Developer Documentation

Nuxeo Platform 5.8

LTS Version

Table of Contents

Developer Documentation Center

Developer Documentation for Nuxeo Platform 5.8

Welcome to the Developer Documentation Center of Nuxeo applications

Architecture tour
Understand the main concepts on which the platform is built

REST API
Documentation about the REST API provided by the platform

Developer Guide
Resources for developers that want to use or extend Nuxeo.

Dev Cook Book
Reference of complete and practical examples to start with

Nuxeo Distributions
Documentation about Nuxeo distributions and associated tools.

Download

Download this documentation in PDF.

Recently updated

Recently Updated

- Using cURL
  - updated Oct 07, 2016 • view change
- How to Change the Default Document Type When Importing a File in the Nuxeo Platform?
  - updated Sep 08, 2016 • view change
- Cross-Origin Resource Sharing (CORS)
  - updated Sep 02, 2016 • view change
- Migrating my Customized Theme
  - updated Sep 02, 2016 • view change
- Navigation URLs
  - updated Sep 02, 2016 • view change
- Document types
  - updated Sep 02, 2016 • view change
- Nuxeo CSV
  - updated Sep 02, 2016 • view change
- Nuxeo Platform User Registration
  - updated Sep 02, 2016 • view change
- How to Remove Tabs
  - updated Sep 02, 2016 • view change
- User Management
  - updated Sep 02, 2016 • view change

Platform Key Features Overview

This page aims at providing an overview of the features you will (or may, depending on the modules you deploy and the features you choose to use) get out-of-the-box by building your application on top of the Nuxeo Platform. This page is not an overview of the technical features from a software engineering point of view, this is addressed in the architecture overview.

A Powerful And Robust Persistence Layer

The Nuxeo repository allows to store structured objects with multiple metadata from simple to complex. It is designed for handling cases as well as documents or media. A document can be present in several places of the repository at the same time: That's what we call proxies. The repository will manage the versioning, the security (with ACLs and custom security policies) the traceability and the indexation: You can leverage the NXQL for querying the repository, including some specific full text search instructions, queries on complex metadata, versions, etc… It is easy to manage, with quota management, possibility to schedule some retention rules, support of multi-tenant.

With Fine Traceability

Ability to search in the audit, to export security and ACLs information regularly, to log custom (project-defined) events in that audit.

Great Authoring Features

WYSIWYG HMTL editing, Markdown support, plugin for integration with live collaborative editing open source platform "Etherpad", complex form management, safe edit (HTML 5 based form backup), ability to revert a previous version of the document. Live Edit provides the ability to edit the content launching automatically the local native editor depending on the file mime-type, without the burden of downloading and re-uploading manually the file. Content preview. Multi-tab editing: Separate all the metadata of your project on various tabs displayed depending on the profile of the connected users. Let some users be able to view or edit a given metadata, or hide that from him, depending on his role. Ability to bulk edit metadata. Tagging support. Support of taxonomies. Support of publishing processes.

In this section

Some Collaborative Features

Document locking, user comments, activity feed, user annotations, versions comparison, user notifications, network based subscriptions, external users registrations, serial review, parallel reviews, polls .

Media Management

Thumbnail extraction, metadata extraction (EXIF, IPTC, XMP), pictures and video conversions, video story boarding, picture watermarking, collections browsing, PDF export of a selection of assets.

Conversions Features

Not only limited to media management field, the platform provides great piping logics for transforming easily your content. It is simple to integrate a third party converter if you don't find what you need in the platform, for example for previewing an AutoCAD document using a

proprietary server side module.

Workflow

Human and services tasks, escalation, merge, fork, exclusive branches, parallel branch, tasks listings, task delegation, task re-assignation. Workflow on several documents at the same time.

Document Synchronization

With Nuxeo Drive, a standalone multi-OS desktop app, users can synchronize the repository as a local file system, with support of off-line modifications, conflict management and versioning policy. The Nuxeo Platform also provides a repository to repository synchronization based on an XML serialization service that can be used for a one-way documents replication.

Email Capture

Either by fetching mails directly from mailboxes to the repository, or by using the Outlook plugins, it is very easy to capture data coming from mails, archiving both body, attachment, and mail metadata.

Document Rendering

There are several options offered for document rendering, so as to produce Open Office Document, Word & Excel documents, PDF, HTML, CSV files where the content comes from the repository, being replaced via a templating logic. You will be able to produce renditions of your data with professional L&F.

Mobile Application

A mobile webapp, touch screen optimized, with additional features on iOS and Android platforms.

Reporting

Ability to produce simple to complex report using the BIRT plugin, or by plugging your BI tools directly to the repository database. Ability to save custom tables of data, choosing which columns to display.

Import/export

CSV import, XML import from scanners and digitization chains, XML export.

User Authentication

Standard LDAP /AD authentication support, OAuth / Open ID compatibility, login with Facebook or GMail account, Shibboleth, Kerberos.

Architecture

The Nuxeo Platform is a mature platform for building modern content apps: Applications managing a lot of semi-structured and structured data and with strong security, life cycle, traceability and transformations dimensions, such as document management, assets management, product life cycle management (PLM), product information management (PIM), Case Management. The Nuxeo Platform is technically highly modular — features and capabilities are spread upon more than 150 modules. On a 10,000 feet overview, you will find the elements below.

- The runtime Java framework, based on OSGi, brings the necessary modularity, flexibility and extensibility that you will like on our platform. This framework exposes the notions of Components, Services and Extension Points.
- The repository transparently translates the document semantics and capabilities (documents, metadata, document query language (NXQL), versioning, audit etc.) into standard SQL and blob storage instructions. The repository works on PostgreSQL, Oracle and SQL servers (and has also been adapted on some other targets such as MySQL and DB2) and stores binary content either on simple file system, on the cloud (S3 connector), or on long term storage devices. The repository has been load-tested against more than 100 millions documents and is continuously improved. The repository is also shipped with high performance multi-threaded injectors and natively implements the standard interface of content repositories: "CMIS".
- The workflow engine allows to run complex processes defined via a graph, with support of many features: parallel node, merge node, n-tasks nodes, sub workflow, escalation, etc.

- The **modular and extensible web application** offers tens of Document Management, Digital Asset Management and Case Management features out-of-the-box. This web application is very easily customizable via Nuxeo Online Services (data model, forms, business views, user actions, workflows, …) so that you can transform the stock app into a 100 % business app that will fit your users expectations mostly by configuration. Furthermore, this web application can be completed by many plugins that bring additional features (ex: polls, permissions export, advanced audit, …).

- The REST API and its client SDKs (Java, JavaScript, PHP, Python, iOS, Android) exposes more than 100 functions, from basic CRUD operations to more advanced repository features such as versioning, document locking, workflow, audit, etc. You can use those clients for implementing connectors (portals, external workflow engines, search engines, …) or for re-implementing a complete new UI on top of the repository in the language and technology of your choice. For example, we provide great accelerators for developing with AngularJS.

- **Connectors, additional plugins and Enterprise world**. The Nuxeo Platform is an "Enterprise" platform: 7 years of integration in many different IT ecosystems made it robust and resilient to many different situations: It supports all the main authentication schemes (form based, OAuth, CAS, Kerberos, SAML2, …), multiple users provisioning strategies, provides different HA deployment options, etc. You will also find some connectors to other enterprise tools such as search engines, mails, etc.

In addition to provide a high quality open source software that is permanently and intensively tested, Nuxeo proposes to the community many tools and means for guarantying your project will be successful:

- Nuxeo Studio, an online tool for configuring the repository, the workflow engine and the web app;
- Nuxeo IDE, an Eclipse plugin offering hot reload, services catalogs, implementation wizards and integration with Nuxeo Studio;
- Funkload, a benchmark framework to streamline the process of load-testing your business application implemented on top of Nuxeo;
- A complete knowledge and set of open source tools for benefiting from a great Continuous Integration chain, the same as Nuxeo's;
- Monitoring tools for going to production safely and quietly.

Section's content:

Component Model

This page describes how the Nuxeo Platform is modular, and how bundles, components and extension points relate to each other to let you create a fully customized application.

The Goal: Easy Customization and Integration

One of the main goals of Nuxeo Platform is to provide an easy and clean way to customize the platform for your application needs. That way:

- No need to hack the system to make it run,
- Your custom code will be based on maintained extension points and interfaces and will be able to be easily upgraded.

For that, Nuxeo Platform provides the following patterns:

- Bundle: A bundle is a "plugin". It is most of the time a ".jar" file with a specific structure that aims at deploying a new set of features on the Nuxeo server. Thanks to this "bundle" notion, developers can deliver their new features in a standalone JAR that the platform will know how to start. As a result, your customization is also delivered as a plugin, like the 10s of plugins that are part of the Nuxeo ecosystem, and that you can find on GitHub or the Nuxeo Marketplace.
- Components and services: A component is a software object declared via XML (and that may reference a Java class) that is used to

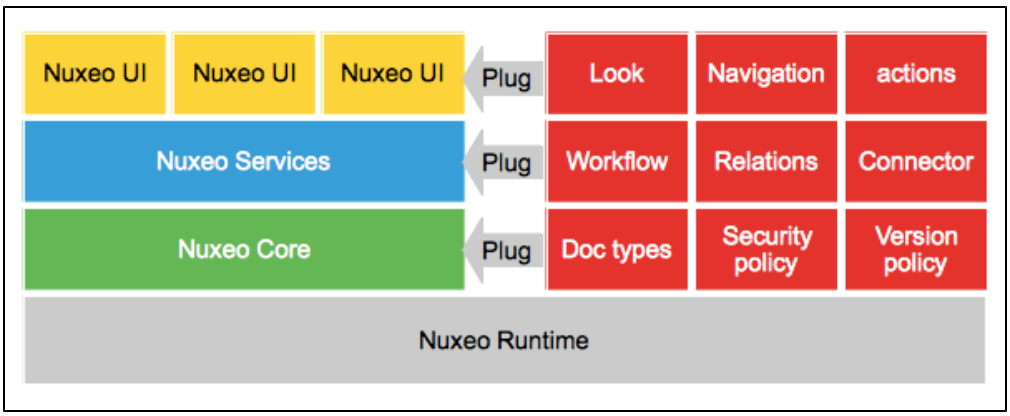
expose some services in the framework. Thanks to this architecture, it is possible to expose a new service anywhere in the Java code executed in the platform. Services are auto-documented: you can see the list on Nuxeo Platform Explorer.
- Extensions: An extension is a mechanism leveraged by the services to let platform users inject customization in the core of the implementation. It is a pattern used frequently on products such as Mozilla, Chrome, or Eclipse. Thanks to this architecture, it is possible to go very deep in product customization only with XML or using our Nuxeo Studio visual environment, without any coding. You can see the list of all extension points in Nuxeo Platform Explorer. Contributions to extensions are delivered in a custom bundle.

Implementing your own *bundle*, your will be able to contribute to existing *extensions* so as to customize things. For instance, you can:

- Define custom schemas and Document types (supported by Nuxeo Studio),
- Define custom forms (supported by Nuxeo Studio),
- Define custom life cycles (supported by Nuxeo Studio),
- Enforce business policies:
  - Use content automation (supported by Nuxeo Studio),
  - Write custom listener scripts,
- Customize the Web UI:
  - Make your own branding (supported by Nuxeo Studio),
  - Add buttons, tabs, links, views (supported by Nuxeo Studio),
  - Build your own theme via the ThemeManager,
- Add workflows.

When existing extensions are not enough, you can declare your own *services* or leverage existing ones in Java code.

In Nuxeo, everything can be configured:



The Recipe: Bundles, Component, Services and Extension Points

Nuxeo Bundles

Inside Nuxeo Platform, software parts are packaged as Bundles. A bundle is a Java archive (JAR) packaged so that it works inside a Nuxeo Application.

It contains:

- An OSGi-based manifest file,
- Java classes,
- XML components,
- Resources,
- A deployment descriptor.

The manifest file is used to:

- Define an id for the bundle,
- Define the dependencies of the bundles (i.e.: other bundles that should be present for this bundle to run),
- List XML components that are part of the bundle.

In this section

Here is an example of a MANIFEST file:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: NXCoreConvert
Bundle-SymbolicName: org.nuxeo.ecm.core.convert
Bundle-Localization: plugin
Require-Bundle: org.nuxeo.ecm.core.api,
 org.nuxeo.ecm.core.convert.api
Bundle-Vendor: Nuxeo
Export-package: org.nuxeo.ecm.core.convert.cache,
 org.nuxeo.ecm.core.convert.extension,
 org.nuxeo.ecm.core.convert.service
Bundle-Category: runtime
Nuxeo-Component: OSGI-INF/convert-service-framework.xml
```

Here we can see that this bundle:

- Is named `org.nuxeo.ecm.core.convert`.
- Depends on two other bundles: `core.api` and `convert.api`.
- Contains one XML component: `convert-service-framework.xml`.

Nuxeo bundles are deployed on the Nuxeo server via the Nuxeo runtime that behaves partially like an OSGi framework. An OSGi framework provides:

- A life cycle model for Java modules,
- A service model.

When an OSGi framework starts it will try to load all bundles installed in the system. When all dependencies of a bundle are resolved, the framework starts the bundle. Starting a bundle means invoking a bundle activator if any is declared in the manifest file.

This way each bundle that registers an activator is notified that it was started - so the bundle activator can do any initialization code required for the bundle to be ready to work. In the same way when a bundle is removed the bundle activator will be notified to cleanup any held resources. OSGi frameworks provides listeners to notify all interested bundles on various framework events like starting a bundle, stopping another one, etc.

This mechanism provides a flexible way to build modular applications which are composed of components that need to take some actions when some resources are become available or are removed. This life cycle mechanism helps bundles react when changes are made in the application. Thus, an OSGi bundle is notified when all its dependencies were resolved and it can start providing services to other bundles. OSGi is also proposing a service model - so that bundles can export services to other bundles in the platform.

There are two major differences between the default Nuxeo Runtime launcher and an OSGi framework:

- Nuxeo is using single class loader for all bundles. It doesn't interpret OSGi dependencies in the manifest.
- Nuxeo services are not exposed as OSGi services.

Components and Services

A **component** is a piece of software defined by a **bundle** used as an entry point by other components that want to contribute some extensions or to ask for some service interface. A component is an abstract concept - it is not necessarily backed by a Java class and is usually made from several classes and XML configuration.

The XML components are XML files, usually placed in the `OSGI-INF` directory, that are used to declare configuration to Nuxeo Runtime.

Each XML component has a unique id and can:

- Declare requirement on other components,
- Declare a Java component implementation,
- Contain a XML contribution,
- Declare a Java contribution.

A Java component is a simple Java class that is declared as component via an XML file.

Components usually derive from a base class provided by Nuxeo Runtime and will be available as a singleton via a simple Nuxeo Runtime call:

```
Framework.getRuntime().getComponent(componentName)
```

Usually, components are not used directly, they are used via a service interface. For that, the XML components can declare which service interfaces are provided by a given component. The component can directly implement the service interface or can delegate service interface implementation to an other class. Components must be accessed using the interfaces they provide and not through real implementation classes. Once declared the service will be available via a simple Nuxeo Runtime call:

```
Framework.getService(ServiceInterface.class)
```

List of services
The list of existing services can be found on the Nuxeo Platform Explorer.

Extension Points

One of the corner stones of the Nuxeo Platform is to provide components and services that can easily be configured or extended. For that, we use the extension point system from Nuxeo Runtime that is inspired from Equinox (Eclipse platform).

This extension point system allows you to:

- Configure the behavior of components (i.e. contribute XML configuration),
- Extend the behavior of components (i.e. contribute Java code or scripting).

Basically, inside the Nuxeo Platform, the pattern is always the same:

- Services are provided by components,
- Components expose extension points.

The same extension point system is used all over the platform:

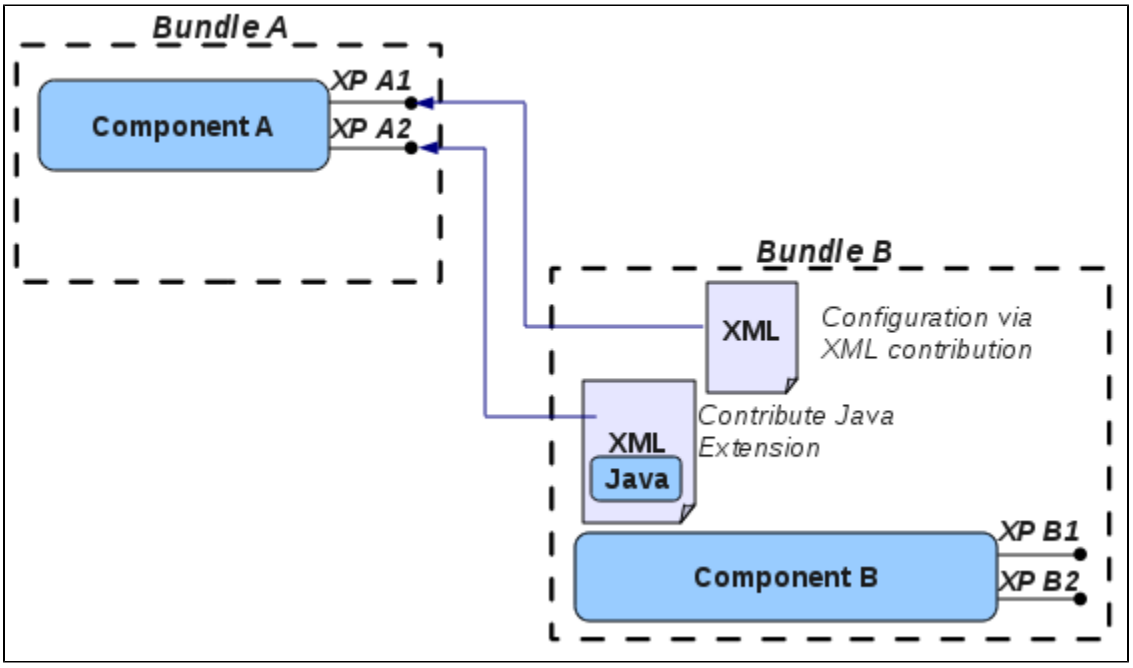
- Inside Nuxeo Runtime itself,
- Inside Nuxeo Core (configure and extend document storage),
- Inside the Nuxeo Service layer (configure and extend ECM services),
- Inside the UI layer (assemble building blocks, contribute new buttons or views, configure navigation, ...).

Each Java component can declare one or several extension points.

These extension points can be used to provide:

- Configuration,
- Additional code (i.e.: plugin system).

So most Nuxeo services are configurable and pluggable via the underlying component.



Declaring an Extension Point

Extension points are declared via the XML component that declares the Java component.

Here is a simple example:

```
<?xml version="1.0"?>
<component name="org.nuxeo.ecm.core.convert.service.ConversionServiceImpl">
  <documentation>
   Service to handle conversions
  </documentation>
  <implementation class="org.nuxeo.ecm.core.convert.service.ConversionServiceImpl"/>*
  <service>
   <provide interface="org.nuxeo.ecm.core.convert.api.ConversionService"/>*
  </service>
  <extension-point name="converter">
    <documentation>
      This extension can be used to register new converters
    </documentation>
    <object class="org.nuxeo.ecm.core.convert.extension.ConverterDescriptor"/>
  </extension-point>
  <extension-point name="configuration">
    <documentation>
      This extension can be used to configure conversion service
    </documentation>
    <object class="org.nuxeo.ecm.core.convert.extension.GlobalConfigDescriptor"/>
  </extension-point>
</component>
```

What we can read in this XML component is:

- A Java component (via the `<component>` tag) with a unique id (into the `name` attribute) is declared;
- This component declares a new service (via the `<implementation>` tag);
- The declaration of the `ConvertService` interface (used to also fetch it) implemented by `ConvertServiceImpl` Java implementation,
- This service expose two extension points:
  - One to contribute configuration (`configuration`),
  - One to contribute some Java code (new `converter` plugin).

Each extension point has his own XML structure descriptor, to specify the XML fragment he's waiting for into this extension point:

- `org.nuxeo.ecm.core.convert.extension.ConverterDescriptor`
- `org.nuxeo.ecm.core.convert.extension.GlobalConfigDescriptor`

This description is defined directly into these classes by annotations. Nuxeo Runtime instanced descriptors and delivers it to the service each time a new contribution of these extension points is detected.

Each Nuxeo extension point uses this pattern to declare configuration possibilities, service integration, behavior extension, etc. You understand this pattern, you will understand all extension points in Nuxeo. And you can use this infrastructure to declare your own business services.

Contributing to an Extension Point

XML components can also be used to contribute to extension points.

For that, the XML component needs to:

- Be referenced in a manifest bundle,
- Specify a target extension point,
- Provide the XML content expected by the target extension point.

Expected XML syntax is defined by the XMap object referenced in the extension point declaration.

Here is an example contribution to an extension point:

```
<?xml version="1.0"?>
<component name="org.nuxeo.ecm.platform.convert.plugins">

  <extension target="org.nuxeo.ecm.core.convert.service.ConversionServiceImpl" point="converter">

    <converter name="zip2html" class="org.nuxeo.ecm.platform.convert.plugins.Zip2HtmlConverter">
      <destinationMimeType>text/html</destinationMimeType>
      <sourceMimeType>application/zip</sourceMimeType>
    </converter>

  </extension>

</component>
```

Extension Points Everywhere

The Nuxeo Platform uses extension points extensively, to let you extend and configure most of the features provided by the platform (see all extension points available in the 5.8 version of the platform for instance).

Nuxeo Runtime

All the bundles included in a Nuxeo Application are part of different plugins (from the core plugins to the high level ones). A minimal application is represented by a single plugin - the framework itself (which is itself packaged as a bundle).
This is what we are naming **Nuxeo Runtime**. Of course launching the Nuxeo Runtime without any plugin installed is useless - except a welcome message in the console nothing happens. But, starting from **Nuxeo Runtime** you can build a complete application by installing different plugins (depending on the type of your application you may end up with tens of bundles).

A basic Nuxeo Application is composed at least of two layers of plugins: the Runtime layer and the core one.

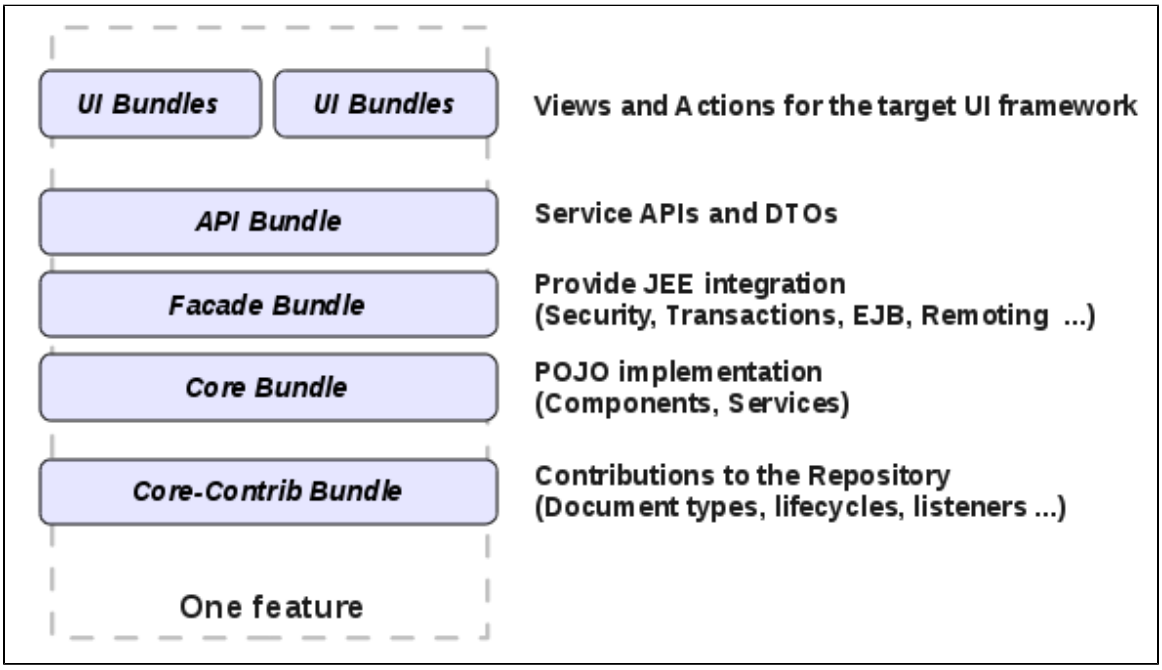
Packaging and Deployment

The layered architecture impacts the way we package features in the Nuxeo Platform.

In order to keep as much deployment options as possible and let you choose what you deploy and where, each feature (workflow, relations, conversions, preview ...) is packaged in several separated bundles.

Typically, this means that each feature will possibly be composed of:

- An **API Bundle** that contains all interfaces and remotable objects needed to access the provided services;
- A **Core Bundle** that contains the POJO implementation for the components and services;
- A **Facade Bundle** that provides the JEE bindings for the services (JTA, Remoting, JAAS ...);
- A **Core Contrib Bundle** that contains all the direct contributions to the Repository (Document types, listeners, security policies ...);
- Client bundles.



All the bundles providing the different layers of the same feature are usually associated to the same Maven artifact group and share the same

parent POM file.
This is basically a bundle group for a given feature.

Now you may want to understand how those packages are deployed on a Nuxeo server.

Content Repository

This page summarizes all the main concepts about documents, access to documents and document storage.
A sub-page gives details of the RBMS mapping layer, "VCS".

Document in Nuxeo

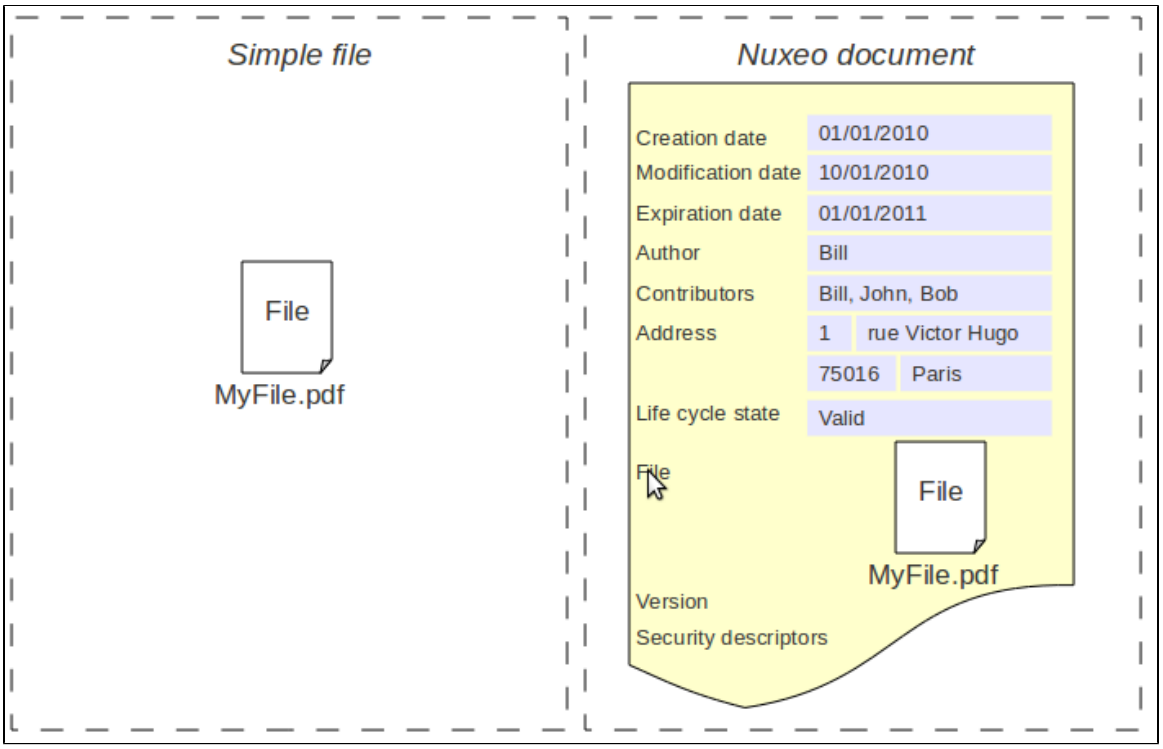
Document vs File

Inside the Nuxeo repository, a document is not just a simple file.

A document is defined as a set of fields.

Fields can be of several types:

- Simple fields (String, Integer, Boolean Date, Double),
- Simple lists (multi-valued simple field),
- Complex types.



On this page

Children pages

A file is a special case of a complex field that contains:

- A binary stream,
- A filename,
- A mime-type,
- A size.

As a result, a Nuxeo Document can contain zero, one or several files.

In fact, inside the Nuxeo repository, even a folder is seen as a document because it holds metadata (title, creation date, creator, ...).

Schemas

Document structure is defined using XSD schemas.

XSD schemas provide:

- A standard way to express structure,
- A way to define metadata blocks.

Each document type can use one or several schemas.

Here is a simple example of a XSD schema used in Nuxeo Core (a subset of Dublin Core):

```
<?xml version="1.0"?>
<xs:schema
        targetNamespace="http://www.nuxeo.org/ecm/schemas/dublincore/"
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns:nxs="http://www.nuxeo.org/ecm/schemas/dublincore/">

    <xs:simpleType name="subjectList">
        <xs:list itemType="xs:string"/>
    </xs:simpleType>
    <xs:simpleType name="contributorList">
        <xs:list itemType="xs:string"/>
    </xs:simpleType>
    <xs:element name="title" type="xs:string"/>
    <xs:element name="description" type="xs:string"/>
    <xs:element name="subjects" type="nxs:subjectList"/>
    <xs:element name="rights" type="xs:string"/>
    <xs:element name="source" type="xs:string"/>
    <xs:element name="coverage" type="xs:string"/>
    <xs:element name="created" type="xs:date"/>
    <xs:element name="modified" type="xs:date"/>
    <xs:element name="issued" type="xs:date"/>
    <xs:element name="valid" type="xs:date"/>
    <xs:element name="expired" type="xs:date"/>
    <xs:element name="format" type="xs:string"/>
    <xs:element name="language" type="xs:string"/>
    <xs:element name="creator" type="xs:string"/>
    <xs:element name="contributors" type="nxs:contributorList"/>
</xs:schema>
```

Document Types

Inside the Nuxeo Repository, each document has a Document Type.

A document type is defined by:

- A name,
- A set of schemas,
- A set of facets,
- A base document type.

Document types can inherit from each other.

By using schemas and inheritance you can carefully design how you want to reuse the metadata blocks.

At pure storage level, the facets are simple declarative markers. These marker are used by the repository and other Nuxeo Platform services to define how the document must be handled.

Default facets include:

- Versionnable,
- HiddenInNavigation,
- Commentable,
- Folderish,
- ...

Here are some Document Types definition examples:

```
<doctype name="File" extends="Document">
    <schema name="common"/>
    <schema name="file"/>
    <schema name="dublincore"/>
    <schema name="uid"/>
    <schema name="files"/>
    <facet name="Downloadable"/>
    <facet name="Versionable"/>
    <facet name="Publishable"/>
    <facet name="Indexable"/>
    <facet name="Commentable"/>
</doctype>

<doctype name="Folder" extends="Document">
    <schema name="common"/>
    <schema name="dublincore"/>
    <facet name="Folderish"/>
    <subtypes>
        <type>Folder</type>
        <type>File</type>
        <type>Note</type>
    </subtypes>
</doctype>
```

At UI level, Document Types defined in the repository are mapped to high level document types that have additional attributes:

- Display name,
- Category,
- Icon,
- Visibility,
- ...

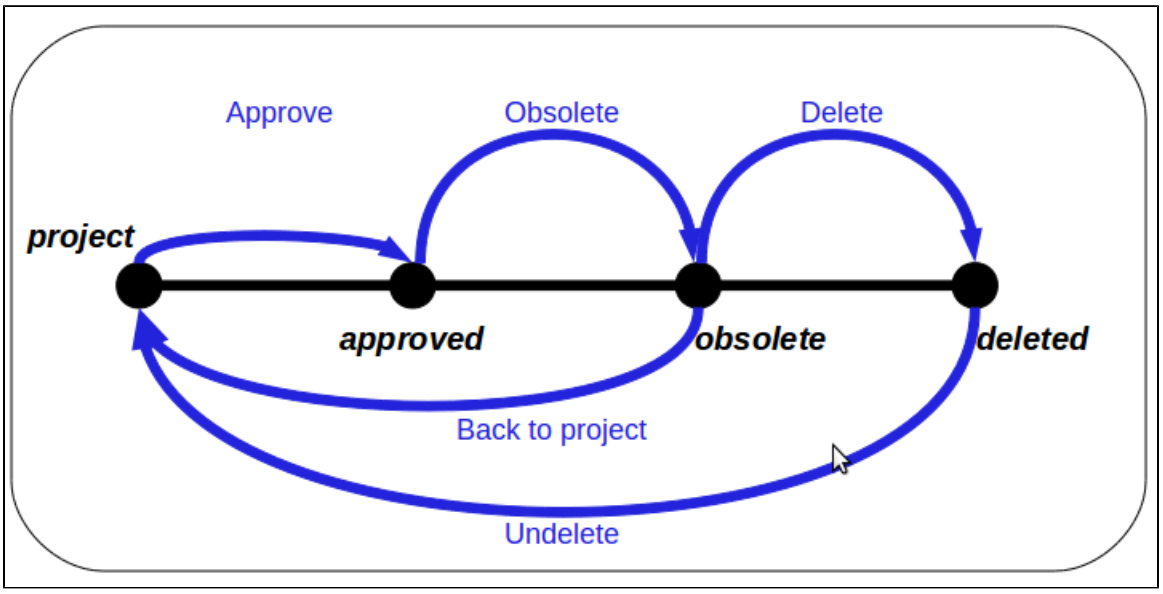
```
<type id="Folder">
    <label>Folder</label>
    <icon>/icons/folder.gif</icon>
    <bigIcon>/icons/folder_100.png</bigIcon>
    <icon-expanded>/icons/folder_open.gif</icon-expanded>
    <category>Collaborative</category>
    <description>Folder.description</description>
    <default-view>view_documents</default-view>
    <subtypes>
        <type>Folder</type>
        <type>File</type>
        <type>Note</type>
    </subtypes>
    <layouts mode="any">
        <layout>heading</layout>
    </layouts>
    <layouts mode="edit">
        <layout>heading</layout>
        <layout>dublincore</layout>
    </layouts>
    <layouts mode="listing">
        <layout>document_listing</layout>
        <layout>document_listing_compact_2_columns</layout>
        <layout>document_listing_icon_2_columns</layout>
    </layouts>
</type>
```

Life Cycle

Nuxeo Core includes a life cycle service.

Each document type can be bound to a life cycle. The life cycle is responsible for defining:

- The possible states of the document (ex: draft, validated, obsolete, ...),
- The possible transitions between states (ex : validate, make obsolete, ...).



Life cycle is not workflow, but:

- Workflows usually use the life cycle of the document as one of the state variable of the process,
- You can simulate simple review process using life cycle and listeners (very easy to do using Nuxeo Studio and Content Automation).

Security Management

By default, security is always on inside Nuxeo repository: each time a document is accessed or a search is issued, security is verified.

The Nuxeo repository security relies on a list of unitary permissions that are used within the repository to grant or deny access. These atomic permissions (Read_Children, Write_Properties ...) are grouped in Permissions Groups (Read, Write, Everything ...) so that security can be managed more easily.

Nuxeo comes with a default set of permissions and permissions groups but you can contribute yours too.

ACL Model

The main model for security management is based on an ACL (Access Control List) system.

Each document can be associated with an ACP (Access Control Policy). This ACP is composed of a list of ACLs that itself is composed of ACEs (Access Control Entry).

Each ACE is a triplet:

- User or Group,
- Permission or Permission group,
- Grant or deny.

ACPs are by default inherited: security check will be done against the merged ACP from the current document and all its parent. Inheritance can be blocked at any time if necessary.

Each document can be assigned several ACLs (one ACP) in order to better manage separation of concerns between the rules that define security:

- The document has a default ACL: The one that can be managed via back-office UI,
- The document can have several workflows ACLs: ACLs that are set by workflows including the document.

Thanks to this separation between ACLs, it's easy to have the document return to the right security if workflow is ended.

Security Policies

The ACP/ACL/ACE model is already very flexible. But is some cases, using ACLs to define the security policy is not enough. A classic example would be confidentiality.

Imagine you have a system with confidential documents and you want only people accredited to the matching confidentiality level to be able to see them. Since confidentiality will be a metadata, if you use the ACL system, you have to compute a given ACL every time this metadata is set. You will also have to compute a dedicated user group for each confidentiality level.

In order to resolve this kind of issue, the Nuxeo repository includes a pluggable security policy system. This means you can contribute custom code that will be run to verify security every time it's needed.

Such polices are usually very easy to write, since in most of the case, it's only a match between a user attribute (confidentiality clearance) and the document's metadata (confidentiality level).

Custom security policy could have an impact on performance, especially when doing open search on a big content repository. To prevent this risk, security policies can be converted in low level query filters that are applied at storage level (SQL when VCS is used) when doing searches.

Indexing and Query

Indexing

All documents stored in the Nuxeo repository are automatically indexed on their metadata. Files contained in Documents are also by default full-text indexed.

For that, Nuxeo Core includes a conversion service that provides full-text conversion from most usual formats (MSOffice, OpenOffice, PDF, HTML, XML, ZIP, RFC 822, ...).

So, in addition to metadata indexing, the Nuxeo repository will maintain a full-text index that will be composed of: all metadata text content + all text extracted from files.

Configuration options depend on the storage back end, but you can define what should be put into the full-text index and even define several separated full-text indexes.

Query Support

Of course, indexing is only interesting if you can issue queries.

The Nuxeo repository includes a query system with a pluggable QueryParser that lets you do search against the repository content. The Nuxeo

repository supports two types of queries:

- NXQL: Native SQL like query language,
- CMISQL: Normalized query language included in CMIS specification.

Both query languages let you search documents based on keyword match (metadata) and/or full-text expressions. You can also manage ordering.

In CMISQL you can do cross queries (i.e. : JOINS).

Here is an example of a NXQL query, to search for all non-folderish documents that have been contributed by a given user:

```
SELECT * FROM Document WHERE
dc:contributors = ?                         // simple match on a multi-valued field
AND ecm:mixinType != 'Folderish'              // use facet to remove all folderish documents
AND ecm:mixinType != 'HiddenInNavigation'     // use facet to remove all documents that should be hidden
AND ecm:isCheckedInVersion = 0               // only get checked-out documents
AND ecm:isProxy = 0 AND                     // don't return proxies
ecm:currentLifeCycleState != 'deleted'        // don't return documents that are in the trash
```

As you may see, there is no security clause, because the repository will always only return documents that the current user can see. Security filtering is built-in, so you don't have to post-filter results returned by a search, even if you use complex custom security policies.

Other Repository Features

Versioning

The Nuxeo Repository includes a versioning system.

At any moment, you can ask the repository to create and archive a version from a document. Versioning can be configured to be automatic (each document modification would create a new version) or on demand (this is bound to a radio button in default UI).

Each version has:

- A label,
- A major version number,
- A minor version number.

The versioning service is configurable so you can define the numbering policy. In fact, even the version storage service is pluggable so you can define your own storage for versions.

Proxies

The Nuxeo Repository includes the concept of Proxy.

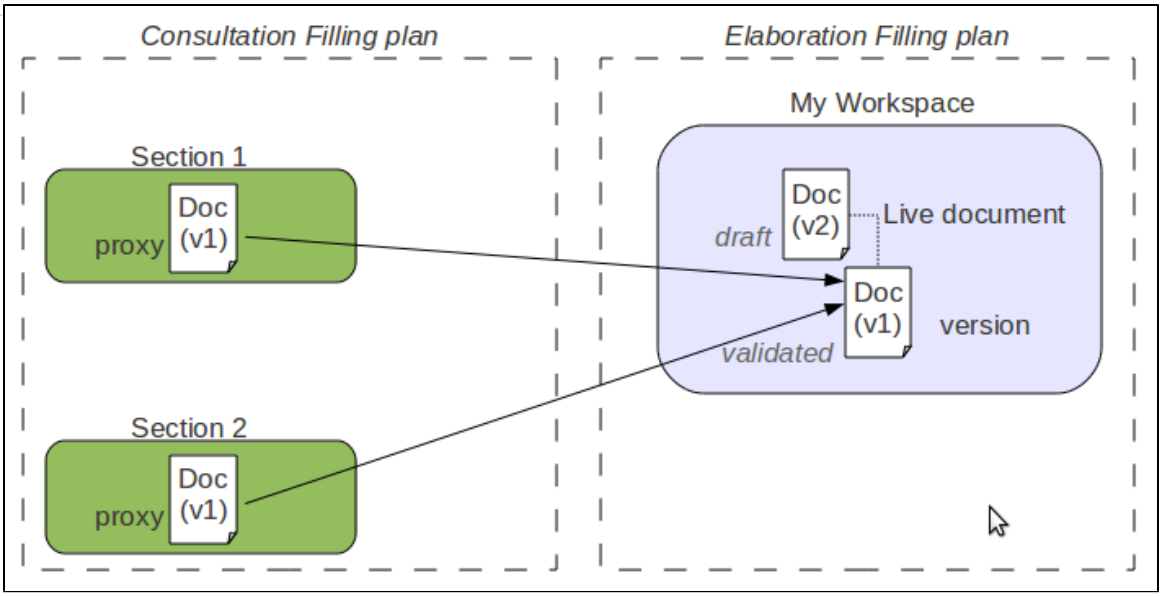
A proxy is very much like a symbolic link on an Unix-like OS: a proxy points to a document and will look like a document from the user point of view:

- The proxy will have the same metadata as the target document,
- The proxy will hold the same files as the target documents (since file is a special kind of metadata).

A proxy can point to a live document or to a version (check in archived version).

Proxies are used to be able to see the same document from several places without having to duplicate any data.

The initial use case for proxies in the Nuxeo Platform is local publishing: when you are happy with a document (and possibly successfully completed a review workflow), you want to create a version for this document. This version will be the one validated and the live document stays in the workspace where you created it. Then you may want to give access to this valid document to several people. For that, you can publish the document into one or several sections: this means creating proxies pointing to the validated version.
Depending on their rights, people that cannot read the document from the workspace (because they can not access it) may be able to see it from one or several sections (that may even be public).



The second use cases for proxies is multi-filling.

If a proxy can not hold metadata, it can hold security descriptors (ACP/ACL). So a user may be able to see one proxy and not an other.

Event Systems

When the Nuxeo repository performs an operation, an event will be raised before and after.

Events raised by the repository are:

- aboutToCreate / emptyDocumentModelCreated / documentCreated
- documentImported
- aboutToRemove / documentRemoved
- aboutToRemoveVersion / versionRemoved
- beforeDocumentModification / documentModified
- beforeDocumentSecurityModification / documentSecurityUpdated
- documentLocked / documentUnlocked
- aboutToCopy / documentCreatedByCopy / documentDuplicated
- aboutToMove / documentMoved
- documentPublished / documentProxyPublished / documentProxyUpdated / sectionContentPublished
- beforeRestoringDocument / documentRestored
- sessionSaved
- childrenOrderChanged
- aboutToCheckout / documentCheckedOut
- incrementBeforeUpdate / aboutToCheckIn

These events are forwarded on the Nuxeo Event Bus and can be processed by custom handlers. As for all events handlers inside the Nuxeo Platform, these handlers can be:

- Synchronous: meaning they can alter the processing of the current operation;
  (ex: change the document content or mark the transaction for rollback);
- Synchronous post-commit: executed just after the transaction has been committed;
  (can be used to update some data before the user gets the result);
- Asynchronous: executed asynchronously after the transaction has been committed.

Inside the Nuxeo repository this event system is used to provide several features:

- Some fields are automatically computed (creation date, modification date, author, contributors, ...),
- Documents can be automatically versioned,
- Full-text extraction is managed by a listener too,
- ...

Using the event listener system for these features offers several advantages:

- You can override the listeners to inject your own logic,
- You can deactivate the listeners if you don't need the processing,
- You can add your own listeners to provide extract features.

Repository Storage: VCS

The Nuxeo repository consists of several services.

One of them is responsible for actually managing persistence of documents. This service is pluggable. The Nuxeo repository uses its own persistence back end: Nuxeo Visible Content Store (VCS)

Nuxeo VCS was designed to provide a clean SQL Mapping. This means that VCS does a normal mapping between XSD schemas and the SQL database:

- A schema is mapped as a table,
- A simple field is mapped as a column,
- A complex type is mapped as a foreign key pointing to a table representing the complex type structure.

Using such a mapping provides several advantages:

- A DBA can see the database content and fine tune indexes if needed,
- You can use standard SQL based BI tools to do reporting,
- You can do low level SQL bulk inserts for data migration.

Binary files are never stored in the database, they are stored via BinaryManager on the file system using their digest. Files are only deleted from the file system by a garbage collector script.

This storage strategy as several advantages:

- Storing several times the same file in Nuxeo won't store it several time on disk,
- Binary storage can be easily snapshotted.

VCS being now the default Nuxeo backend, it also provides some features that were not available when using the previous JCR backend:

- Tag service,
- Possibility to import a document with a fixed UUID (useful for application level synchronization).

In addition, VCS provides a native cluster mode that does not rely on any external clustering system. This means you can have two (or more) Nuxeo servers sharing the same data: you only have to turn on Nuxeo VCS Cluster mode.

Advantages of VCS:

- SQL Storage is usage by DBAs and by BI reporting tools,
- Supports Hot Backup,
- Supports Cluster mode,
- Supports extra features,
- Supports low level SQL bulk imports,
- VCS scales well with big volumes of Documents.

Drawbacks of VCS:

- storage is not JCR compliant.

Advanced Features

Lazy Loading and Binary Files Streaming

In Java API, a Nuxeo document is represented as a *DocumentModel* object.

Because a Document can be big (lots of fields including several files), a DocumentModel object could be big:

- Big object in memory,
- Big object to transfer on the network (in case of remoting),
- Big object to fetch from the storage backend.

Furthermore, even when you have very complex documents, you don't need all these data on each screen: in most screens you just need a few properties (title, version, life cycle state, author, ...).

In order to avoid these problems, the Nuxeo DocumentModel supports lazy-fetching: a DocumentModel is by default not fully loaded, only the field defined as prefetch are initially loaded. The DocumentModel is bound to the repository session that was used to read it and it will transparently fetch the missing data, block per block when needed.

You still have the possibility to disconnect a DocumentModel from the repository (all data will be fetched), but the default behavior is to have a lightweight Java object that will fetch additional data when needed.

The same kind of mechanism applies to files, with one difference: files are transported via a dedicated streaming service that is built-in. Because default RMI remoting is not so smart when it comes to transferring big chuck of binary data, Nuxeo uses a custom streaming for transferring files from and to the repository.

Transaction Management

The Nuxeo repository uses the notion of *Session*.

All the modifications to documents are done inside a session and modifications are saved (written in the back end) only when the session is saved.

In a JTA/JCA aware environment, the repository session is bound to a JCA Connector that allows:

- The repository session to be part of the global JTA transaction,
- The session to be automatically saved when the transaction commits.

This means that in a JTA/JCA compliant environment you can be sure that the repository will always be safe and have the expected transactional behavior. This is important because a single user action could trigger modifications in several services (update documents in repository, update a workflow process state, create an audit record) and you want to be sure that either all these modifications are done, or none of them: you don't want to end up in an inconsistent state.

DocumentModel Adapter

In a lot of cases, documents are used to represent Business Object: invoice, subscription, contract...

The DocumentModel class will let you design the data structure using schemas, but you may want to add some business logic to it:

- Providing helper methods that compute or update some fields,
- Adding integrity checks based on business rules,
- Adding business methods.

For this, Nuxeo Core contains an *adapter* system that lets you bind a custom Java class to a DocumentModel. The binding can be made directly against a document type or can be associated to a facet.

By default, The Nuxeo Platform provides some generic adapters:

- **BlobHolder**: Lets you read and write binary files stored in a document,
- **CommentableDocument**: Encapsulates comment service logic so that you can easily comment a document,
- **MultiViewPicture**: Provides an abstraction and easy API to manipulate a picture with multiple views,
- ...

Binary Store

Repository and BinaryManager

Each content repository has to be associated with a `BinaryManager` implementation. The `BinaryManager` is a low level interface that only deals with binary stream.

```
Binary getBinary(InputStream in) throws IOException;

  Binary getBinary(String digest);
```

On this page

As you can see, the methods do not have any document related parameters. This means the binary storage is independent from the documents:

- Moving a document does not impact the binary stream;
- Updating a document does not impact the binary stream.

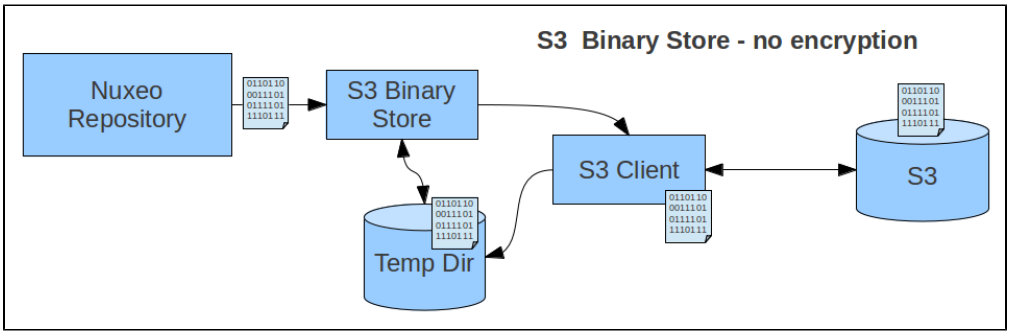
In addition, the streams are stored using their digest, thanks to that:

- `BlobStore` does automatically manage de-duplication;
- `BlobStore` can be safely snapshoted (files are never moved or updated, and they are only removed via a `GarbageCollection` process).

From Simple FS to S3 Binary Manager

The default `BinaryManager` implementation is based on a simple filesystem: considering the storage principles, this is safe to use this implementation even on a NFS like filesystem (since there is no conflicts).

You can also use the S3 Binary Manager to use AWS Cloud File System.



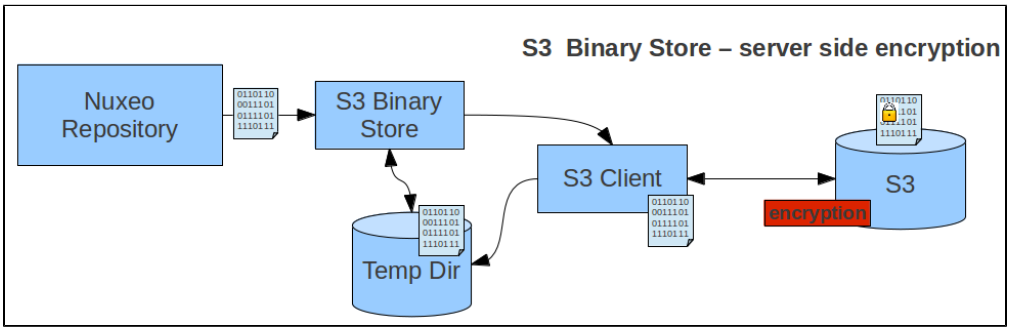
The Temporary storage is used to avoid delays when using the Stream several times (ex: multiple conversions) inside the Nuxeo Server.

Encryption

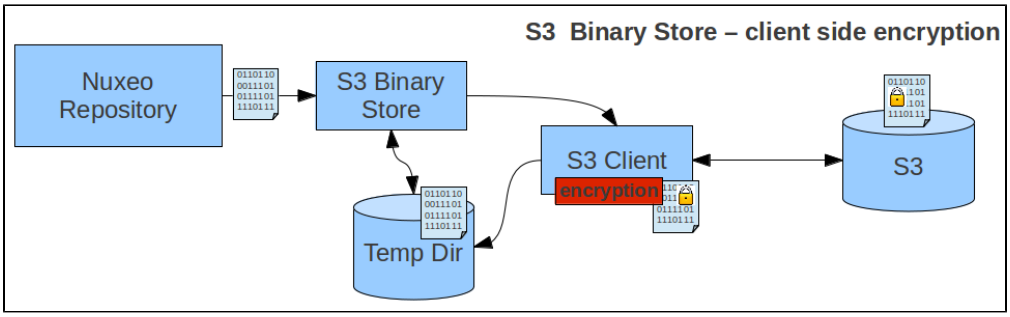
A common question regarding `BinaryManager` is the support for encryption.

Built-in Encryption

If we take the example of the S3 BinaryManager, AWS S3 Client library supports both client side and server side encryption:



With Server side encryption, the encryption is completely transparent.

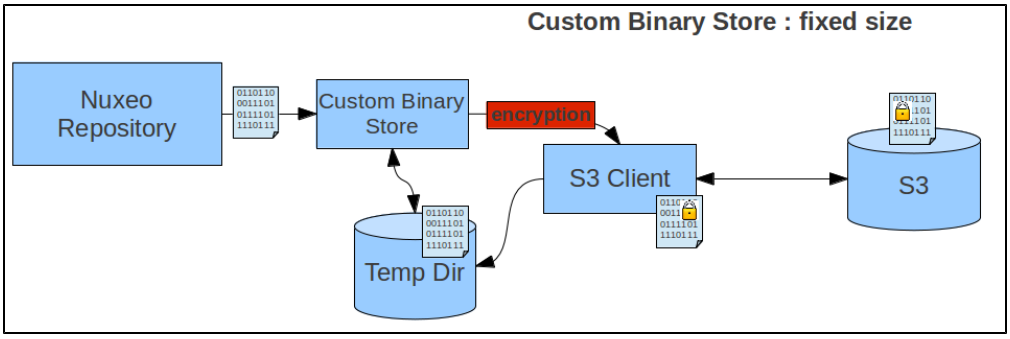


In Client side encryption mode the S3 Client manages the encrypt / decrypt process. The local temporary file is in clear.

Custom Encryption

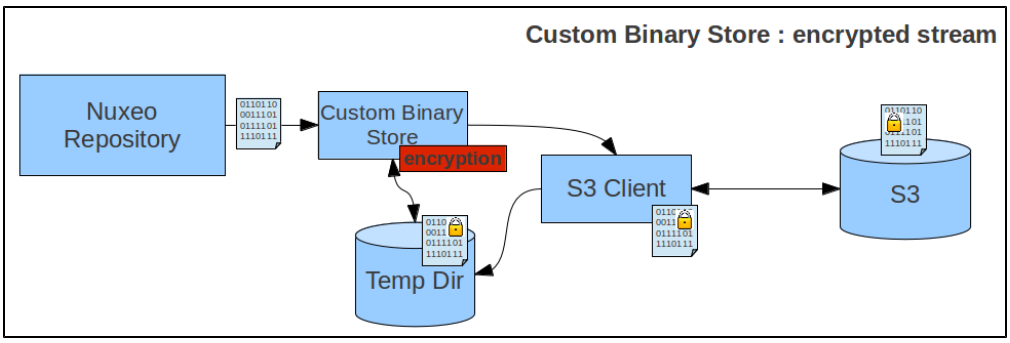
You can contribute custom implementation of the BinaryManager: since the interface is very simple, the implementation is simple too.

- The first possible approach is to handle custom crypt / decrypt on top of AWS S3 Client library:



In that case, the local temporary file is in clear.

- The second possible approach is to handle the crypt/decrypt process on the fly.
  This means that the temp file is crypted, but as a trade off:
  - Decrypting should be run on the fly each time the stream is read.
  - Determining the stream size requires more work.



Deleting Documents

Deleting a document involves several steps before the full document is actually deleted from the database and disk. These steps are described below.

Putting the Document in the Trash

For most standard document types in the Nuxeo Platform, the user interface action of deleting a document (**Delete** button) just puts it in the trash (visible in the **Manage** > **Trash** tab). Once in the trash, the document may be either undeleted (**Restore** button), or removed (**Permanent delete** button).

Putting a document in the trash is done by changing is life cycle state to **deleted**, by following the **delete** transition of the document's life cycle. If no such life cycle exists for the document, then it won't be put in the trash at all but will be immediately permanently deleted using the steps below.

When the trash is purged, all its documents are permanently deleted.

Besides the standard user interface, a document is put in the trash when using WebDAV, Nuxeo Drive, or using the TrashService.trashDocuments API.

In this section

Permanently Deleting the Document

A permanent delete is done by most Nuxeo APIs, typically CoreSession.removeDocument or the higher-level APIs that use it like the CMIS bindings or the Automation Document.Delete operation.

Soft-Delete

If soft-delete is enabled (this is not the case by default), then the document is marked as deleted in the database (using a simple boolean flag) but no rows are actually removed. A search will not be able to find any document marked deleted in this way. From the application's point of view, the document is already fully deleted.

A scheduled periodic process will then hard-delete the documents marked as deleted at a later time, for asynchronous cleanup in fixed-sized batches.

Using Soft-Delete
Soft-delete can be enabled to relieve the database of expected heavy loads if many documents are deleted at the same time.

When Nuxeo has to hard-delete lots of documents, many rows in many tables, themselves related by foreign key constraints, have to be removed. On some databases this can use many resources (like undo segments for Oracle) and take a lot of time, so soft-delete is designed to spread these costly operations over time.

To activate soft-delete, you should change the repository configuration to add `<softDelete enabled="true" />` . See Configuration Templates for more about updating the `default-repository-config.xml.nxftl` file.

Please consult NXP-11335 for more details about soft-delete and the configuration of the periodic cleanup.

Hard-Delete

If soft-delete is not enabled, or when the periodic cleanup process for soft-delete happens, the document's data is actually physically deleted from the database by using `DELETE` statements (hard-delete).

VCS Architecture

The goals of VCS (Visible Content Store) are to:

- Store information in standard SQL databases,
- Use "natural" object mapping to tables,
- Be fast,
- Support full-text searches on databases having that capability,
- Have some flexibility in the storage model to optimize certain cases at configuration time.

This section's sub-pages describe the architecture of VCS.

"Visible SQL" Structures

When using VCS, you can directly access all your data in pure SQL: the schema of the database is a mapping of the XSD schemas you defined in the Nuxeo configuration.

Easy Reporting

Because SQL data can be accessed directly, you may use any SQL reporting tool (Business Object, BIRT, Crystal Reports...) to build custom business reports on your documents data.

Easy Data Injection

For big data migration you can use direct SQL injection. Although we provide importer tools that will work on both JCR and VCS, it will never be as fast as bulk inserts can. Furthermore, in some cases, you don't want the people dealing with data migration to be Nuxeo developers: in this case, using plain SQL is an advantage.

In this section

Children pages

Simple Debugging

Because data are transparently stored in SQL tables, it's very easy to have a look at them and understand exactly what data has been stored.

Easy Schema Migration

With VCS, schema changes in Nuxeo are automatically propagated to the SQL level (columns are added as needed, and you are warned about extra unused columns). If you change a field type, you can just do `ALTER TABLE` to change the column type.

Easy and Safe Hot Backup

You can hot backup your DB and then (i.e., after) hot backup the VCS binary store, data will be consistent:

- You have no risk of having a file being in process of update,
- You have no risk of having a file referenced by the DB that has been removed from the filesystem.

Some additional explanations on why VCS ensures safe hot backups.

Blob Storage

By default VCS stores all blobs on the filesystem in a lock-free, cluster-aware manner, based on the "Content-addressable storage" paradigm. Instead of the filesystem, you can also elect to store the blobs in Amazon S3 using the appropriate plugin.

You can also choose to store blobs in the database but we don't recommend it. In our experience storing blobs in the database leads to a lot of problems:

- Performances really drops,
- Some JDBC drivers load blobs in the JVM memory,
- Database backup/restore is very slow,
- Database sync (Master/Slave) is very very slow too.

If despite those recommendation you still want to take this architectural option, you can use the dedicated plugin available at https://github.com/nuxeo/nuxeo-core-binarymanager-sql/.

Internal VCS Model

The Nuxeo model is mapped internally to a model based on a hierarchy of nodes and properties. This model is similar to the basic JCR (JSR-170) data model.

Nodes, Properties, Children

A node represents a complex value having several properties. The properties of a node can be either simple (scalars, including binaries), or collections of scalars (lists usually). A node can also have children which are other nodes.

Children

The parent-child information for nodes is stored in the `hierarchy` table.

The normal children of a document are mapped to child nodes of the document node. If a document contains complex types, they are also mapped to child nodes of the document mode. There are therefore two kinds of children: child documents and complex types. They have to be quickly distinguished in order to:

- Find all child documents and only them,
- Find all complex properties of a document and only them,
- Resolve name collisions.

To distinguish the two, the hierarchy table has a column holding a `isproperty` flag to decide if it's a complex property or not.

Fragment Tables

A fragment table is a table holding information corresponding to the scalar properties of one schema (simple fragment), or a table corresponding to one multi-valued property of one schema (collection fragment).

For a simple fragment, each of the table's columns correspond to a simple property of the represented schema. One row corresponds to one document (or one instance of a complex type) using that schema.

For a collection fragment, the set of values for the multi-valued property is represented using as many rows as needed. An additional `pos` col umn provides ordering of the values.

A node is the set of fragments corresponding to the schemas of that node.

See the VCS Tables page for more details.

Fields Mapping

Nuxeo fields are mapped to properties or to child nodes:

- A simple type (scalar or array of scalars) is mapped to a property (simple or collection) of the document node,
- A complex type is mapped to a child node of the document node. There are two kinds of complex types to consider:
  - Lists of complex types are mapped to an ordered list of complex property children,
  - Non-list complex types are mapped to a node whose node type corresponds to the internal schema of the complex type.

Security

Security information is stored as an ACL which is a collection of simple ACEs holding basic rights information. This collection is stored in a dedicated table in a similar way to lists of scalars, except that the value is split over several column to represent the rich ACE values.

In this section

VCS Tables

Fragment Tables

Each node has a unique identifier which is a UUID randomly generated by VCS. This random generation has the advantage that different cluster nodes don't have to coordinate with each other to create ids.

All the fragments making up a given node use the node id in their `id` column.

For clarity in the rest of this document simple integers are used, but Nuxeo actually uses UUIDs, like `56e42c3f-db99-4b18-83ec-601e06 53f906` for example.

Hierarchy Table

There are two kinds of nodes: filed ones (those who have a location in the containment hierarchy), and unfiled ones (version frozen nodes, and some other documents like tags).

Each node has a row in the main hierarchy table defining its containment information if it is filed, or just holding its name if it is unfiled. The same tables holds ordering information for ordered children.

Table **hierarchy** :

id	parentid	pos	name	...
1			""	
1234	1		workspace	
5678	1234		mydoc	

Note that:

- The `id` column is used as a `FOREIGN KEY` reference with `ON DELETE CASCADE` from all other fragment tables that refer to it,
- The `pos` is `NULL` for non-ordered children,
- The `parentid` and `pos` are `NULL` for unfiled nodes,
- The `name` is an empty string for the hierarchy's root.

For performance reasons (denormalization) this table has actually more columns; they are detailed below.

Type Information

The node types are accessed from the main `hierarchy` table.

When retrieving a node by its id the `primarytype` and `mixintypes` are consulted. According to their values a set of applicable fragments is deduced, to give a full information of all the fragment tables that apply to this node.

Table **hierarchy** (continued):

id	...	isproperty	primarytype	mixintypes	...
1		FALSE	Root		
1234		FALSE	Bar		
5678		FALSE	MyType	[Facet1,Facet2]	

The `isproperty` column holds a boolean that distinguishes normal children from complex properties,

The `mixintypes` stores a set of mixins (called Facets in the high-level documentation). For databases that support arrays (PostgreSQL), they are stored as an array; for other databases, they are stored as a `|`-separated string with initial and final `|` terminators (in order to allow efficient `LIKE`-based matching) — for the example row 5678 above the mixins would be stored as the string `|Facet1|Facet2|`.

Simple Fragment Tables

Each Nuxeo schema corresponds to one table. The table's columns are all the single-valued properties of the corresponding schema. Multi-valued properties are stored in a separate table each.

A "myschema" fragment (corresponding to a Nuxeo schema with the same name) will have the following table:

Table **myschema** :

id	title	description	created
5678	Mickey	The Mouse	2008-08-01 12:56:15.000

A consequence is that to retrieve the content of a node, a `SELECT` will have to be done in each of the tables corresponding to the node type and all its inherited node types. However lazy retrieval of a node's content means that in many cases only a subset of these tables will be needed.

Collection Fragment Tables

A multi-valued property is represented as data from a separate array table holding the values and their order. For instance, the property "my:subjects" of the schema "myschema" with prefix "my" will be stored in the following table:

Table **my_subjects** :

id	pos	item
5678	0	USA
5678	1	CTU

Files and Binaries

The blob abstraction in Nuxeo is treated by the storage as any other schema, "content", except that one of the columns hold a "binary" value. This binary value corresponds indirectly to the content of the file. Because the content schema is used as a complex property, there are two entries in the `hierarchy` table for each document.

Table **`hierarchy`** :

id	parentid	name	isproperty	primarytype	...
4061	5678	myreport	FALSE	File	
4062	5678	test	FALSE	File	
4063	5678	test2	FALSE	File	
8501	4061	content	TRUE	content	
8502	4062	content	TRUE	content	
8503	4063	content	TRUE	content	

Table **`content`** :

id	name	mime-type	encoding	data	length	digest
8501	report.pdf	application/pdf		ebca0d868ef3	344256	
8502	test.txt	text/plain	ISO-8859-1	5f3b55a834a0	541	
8503	test_copy.txt	text/plain	ISO-8859-1	5f3b55a834a0	541	

Table **`file`** :

id	filename
4061	report.pdf
4062	test.txt
4063	test_copy.txt

The filename is also stored in a separate `file` table just because the current Nuxeo schemas are split that way (the filename is a property of the document, but the content is a child complex property). The filename of a blob is also stored in the `name` column of the `content` table.

The `data` column of the `content` table refers to a binary type. All binary storage is done through the `BinaryManager` interface of Nuxeo.

The default implementation (`DefaultBinaryManager`) stores binaries on the server filesystem according to the value stored in the data column, which is computed as a cryptographic hash of the binary in order to check for uniqueness and share identical binaries (hashes are actually longer than shown here). On the server filesystem, a binary is stored in a set of multi-level directories based on the has, to spread storage. For instance the binary with the hash `c38fcf32f16e4fea074c21abb4c5fd07` will be stored in a file with path `data/c3/8f/c38fcf32f16e4fea074c21abb4c5fd07` under the binaries root.

Relations

Some internal relations are stored using VCS. By default they are the relations that correspond to tags applied on documents, although specific applications could add new ones. Note that most user-visible relations are still stored using the Jena engine in different tables.

Table **`relation`** :

id	source	sourceUri	target	targetUri	targetString
1843	5670		5700		
1844	5670				"some text"

The `source` and `target` columns hold document ids (keyed by the `hierarchy` table). The relation object itself is a document, so its id is present in the `hierarchy` table as well, with the `primarytype` "Relation" or a subtype of it.

In the case of tags, the relation document has type "Tagging", its source is the document being tagged, and its target has type "Tag" (a type with a schema "tag" that contains a field "label" which is the actual tag).

Versioning

You may want to read background information about Nuxeo versioning first.

Versioning uses identifiers for several concepts:

- **Live node id**: the identifier of a node that may be subject to versioning.
- **Version id**: the identifier of the frozen node copy that is created when a version was snapshotted, often just called a "version".
- **versionable id**: the identifier of the original live node of a version, but which keeps its meaning even after the live node may be deleted. Several frozen version nodes may come from the same live node, and therefore have the same versionable id, which is why it is also called also the *version series id*.

Version nodes don't have a parent (they are unfiled), but have more meta-information (versionable id, various information) than live nodes. Live nodes hold information about the version they are derived from (base version id).

Table **`hierarchy`** (continued):

id	...	isversion	ischeckedin	baseversionid	majorversion	minorversion
5675			TRUE	6120	1	0
5678			FALSE	6143	1	1
5710			FALSE			
6120		TRUE			1	0
6121		TRUE			1	1
6143		TRUE			4	3

Note that:

- This information is inlined in the hierarchy table for performance reasons,
- The `baseversionid` represents the version from which a checked out or checked in document originates. For a new document that has never been checked in it is `NULL`.

Table **`versions`** :

id	versionableid	created	label	description	islatest	islatestmajor
6120	5675	2007-02-27 12:30:00.000	1.0		FALSE	TRUE
6121	5675	2007-02-28 03:45:05.000	1.1		TRUE	FALSE
6143	5678	2008-01-15 08:13:47.000	4.3		TRUE	FALSE

Note that:

- The `versionableid` is the id of the versionable node (which may not exist anymore, which means it's not a `FOREIGN KEY` reference), and is common to a set of versions for the same node, it is used as a *version series id*.
- `islatest` is true for the last version created,
- `islatestmajor` is true for the last major version created, a major version being a version whose minor version number is 0,
- The `label` contains a concatenation of the major and minor version numbers for users' benefit.

Proxies

Proxies are a Nuxeo feature, expressed as a node type holding only a reference to a frozen node and a convenience reference to the versionable node of that frozen node.

Proxies by themselves don't have additional content-related schema, but still have security, locking, etc. These facts are part of the node type inheritance, but the proxy node type table by itself is a normal node type table.

Table **proxies** :

id	targetid	versionableid
9944	6120	5675

Note that:

- `targetid` is the id of a version node and is a `FOREIGN KEY` reference to `hierarchy.id`.
- `versionableid` is duplicated here for performance reasons, although it could be retrieved from the target using a `JOIN`.

Locking

The locks are held in a table containing the lock owner and a timestamp of the lock creation time.

Table **locks** :

id	owner	created
5670	Administrator	2008-08-20 12:30:00.000
5678	cobrian	2008-08-20 12:30:05.000
9944	jbauer	2008-08-21 14:21:13.488

When a document is unlocked, the corresponding line is deleted.

Another important feature of the `locks` table is that the `id` column is not a foreign key to `hierarchy.id`. This is necessary in order to isolate the locking subsystem from writing transactions on the main data, to have atomic locks.

Security

The Nuxeo security model is based on the following:

- A single ACP is placed on a (document) node,
- The ACP contains an ordered list of named ACLs, each ACL being an ordered list of individual grants or denies of permissions,
- The security information on a node (materialized by the ACP) also contains local group information (which can emulate owners).

Table **acls** :

id	pos	name	grant	permission	user	group
5678	0	local	true	WriteProperties	cobrian	
5678	1	local	false	ReadProperties		Reviewer
5678	2	workflow	false	ReadProperties	kbauer	

This table is slightly denormalized (names with identical values follow each other by `pos` ordering), but this is to minimize the number of `JOIN` s to get all ACLs for a document. Also one cannot have a named ACL with an empty list of ACEs in it, but this is not a problem given the semantics of ACLs.

The `user` column is separated from the `group` column because they semantically belong to different namespaces. However for now in Nuxeo groups and users are all mixed in the `user` column, and the `group` column is kept empty.

Miscellaneous Values

The life cycle information (life cycle policy and life cycle state) is stored in a dedicated table.

The dirty information (a flag that describes whether the document has been changed since its last versioning) is stored in the same table for convenience.

Two Nuxeo "system properties" of documents in use by the workflow are also available.

Table **misc** :

id	lifecyclepolicy	lifecyclestate	dirty	wfinprogress	wfincoption
5670	default	draft	FALSE		
5678	default	current	TRUE		

9944	publishing	pending	TRUE		

Full-text

The full-text indexing table holds information about the fulltext extracted from a document, and is used when fulltext queries are made. The structure of this table depends a lot on the underlying SQL database used, because each database has its own way of doing fulltext indexing. The basic structure is as follow:

Table **`fulltext`** :

id	jobid	fulltext	simpletext	binarytext
5678	5678	Mickey Mouse USA CTU report pdf reporttitle ...	Mickey Mouse USA CTU report pdf	reporttitle ...

The `simpletext` column holds text extracted from the string properties of the document configured for indexing. The `binarytext` column holds text extracted from the blob properties of the document configured for indexing. The `fulltext` column is the concatenation of the two and is the one usually indexed as fulltext by the database. A database trigger updates `fulltext` as soon as `simpletext` or `binarytext` is changed.

The `jobid` column holds the document identifier of the document being indexed. Once the asynchronous job complete, all the rows that have a `jobid` matching the document id are filled with the computed full-text information. This ensures in most cases that the fulltext information is well propagated to all copies of the documents.

Some databases can directly index several columns at a time, in which case the `fulltext` column doesn't exist, there is no trigger, and the two `simpletext` and `binarytext` columns are indexed together.

The above three columns show the data stored and indexed for the default fulltext index, but Nuxeo allows any number of additional indexes to be used (indexing a separate set of properties). In this case additional columns are present, suffixed by the index name; for instance for index "main" you would find the additional columns:

Table **`fulltext`** (continued):

id	...	fulltext_main	simpletext_main	binarytext_main
5678		bla	bla	

Other System Tables

Repositories

This table hold the root id for each repository. Usually Nuxeo has only one repository per database, which is named "default".

Table **`repositories`** :

id	name
1	default

Note that the `id` column is a `FOREIGN KEY` to `hierarchy.id`.

Clustering

When configured for cluster mode, two additional tables are used to store cluster node information and cluster invalidations.

A new row is created automatically in the cluster nodes table when a new cluster node connects to the database. It is automatically removed when the cluster node shuts down.

Table **`cluster_nodes`** :

nodeid	created
71	2008-08-01 12:31:04.580
78	2008-08-01 12:34:51.663
83	2008-08-01 12:35:27.184

Note that:

- The `nodeid` is assigned by the database itself, its form depends on the database,
- The `created` date is not used by Nuxeo but is useful for diagnostics.

The cluster invalidations are inserted when a transaction commits, the invalidation rows are duplicated for all cluster node ids that are not the current cluster node. Rows are removed as soon as a cluster node checks for its own invalidations, usually at the beginning of a transaction.

Table **`cluster_invals`** :

nodeid	id	fragments	kind
78	5670	hierarchy, dublincore, misc	1
78	5678	dublincore	1
83	5670	hierarchy, dublincore, misc	1
83	5678	dublincore	1

Note that:

- `id` is a node id but is not a `FOREIGN KEY` to `hierarchy.id` for speed reasons,
- `Fragments` is the list of fragments to invalidate; it is a space-separated string, or an array of strings for databases that support arrays,
- `kind` is 1 for modification invalidations, or 2 for deletion invalidations.

Path Optimizations

For databases that support it, some path optimizations allow faster computation of the NXQL `STARTSWITH` operator.

When path optimizations are enabled (this is the default on supported databases), an addition table stores the descendants of every document. This table is updated through triggers when documents are added, deleted or moved.

Table **`descendants`** :

id	descendantid
1	1234
1	5678
1234	5678

Note that `descendantid` is a `FOREIGN KEY` to `hierarchy.id`.

Another more efficient optimization is used instead for PostgreSQL (see NXP-5390). For this optimization, an `ancestors` table stores all the ancestors as an array in a single cell. This table is also updated through triggers:

Table **`ancestors`** :

id	ancestors
1234	[1]
5678	[1, 1234]

The `ancestors` column contains the array of ordered ancestors of each document (not complex properties), with the root at the beginning of the array and the direct parent at the end.

ACL Optimizations

For databases that support it, ACL optimizations allow faster security checks than the `NX_ACCESS_ALLOWED` stored procedure used in standard.

The `hierarchy_read_acl` table stores information about the complete ACL that applies to a document.

Table **`hierarchy_read_acl`** :

id	acl_id
5678	bc61ba9c8dbf034468ac361ae068912b

The `acl_id` is the unique identifier for the complete read ACL (merged with ancestors) for this document. It references the `id` column in the

`read_acls` table, but not using a `FOREIGN KEY` for speed reasons.

The `read_acls` table stores all the possibles ACLs and their unique id.

Table **`aclr`** :

`acl_id`	**`acl`**
bc61ba9c8dbf034468ac361ae068912b	-Reviewer,-kbauer,Administrator,administrators

The unique ACL id is computed through a hash to simplify unicity checks.

When a security check has to be done, the user and all its groups are passed to a stored procedure (usually `NX_GET_READ_ACLS_FOR`), and the resulting values are `JOIN`ed to the `hierarchy_read_acl` table to limit document ids to match.

The `NX_GET_READ_ACLS_FOR` stored procedure has to find all ACLs for a given user, and the results of that can be cached in the `read_acls_cache` table. This cache is invalidated as soon as security on a document changes.

Table **`aclr_user_map`** :

`users_id`	**`acl_id`**
f4bb42d8	1
f4bb42d8	1234
f4bb42d8	5678
c5ad3c99	1
c5ad3c99	1234

Table **`aclr_user`** :

`user_id`	**`users`**
f4bb42d8	Administrator,administrators
c5ad3c99	kbauer,members ...

Note:

- `f4bb42d8` is the MD5 hash for "Administrator,administrators", `c5ad3c99` is the MD5 hash for "kbauer,members".
- A hash is used to make sure this column has a limited size.

An additional table, `aclr_modified`, is used to temporarily log document ids where ACLs are modified.

Table **`aclr_modified`** :

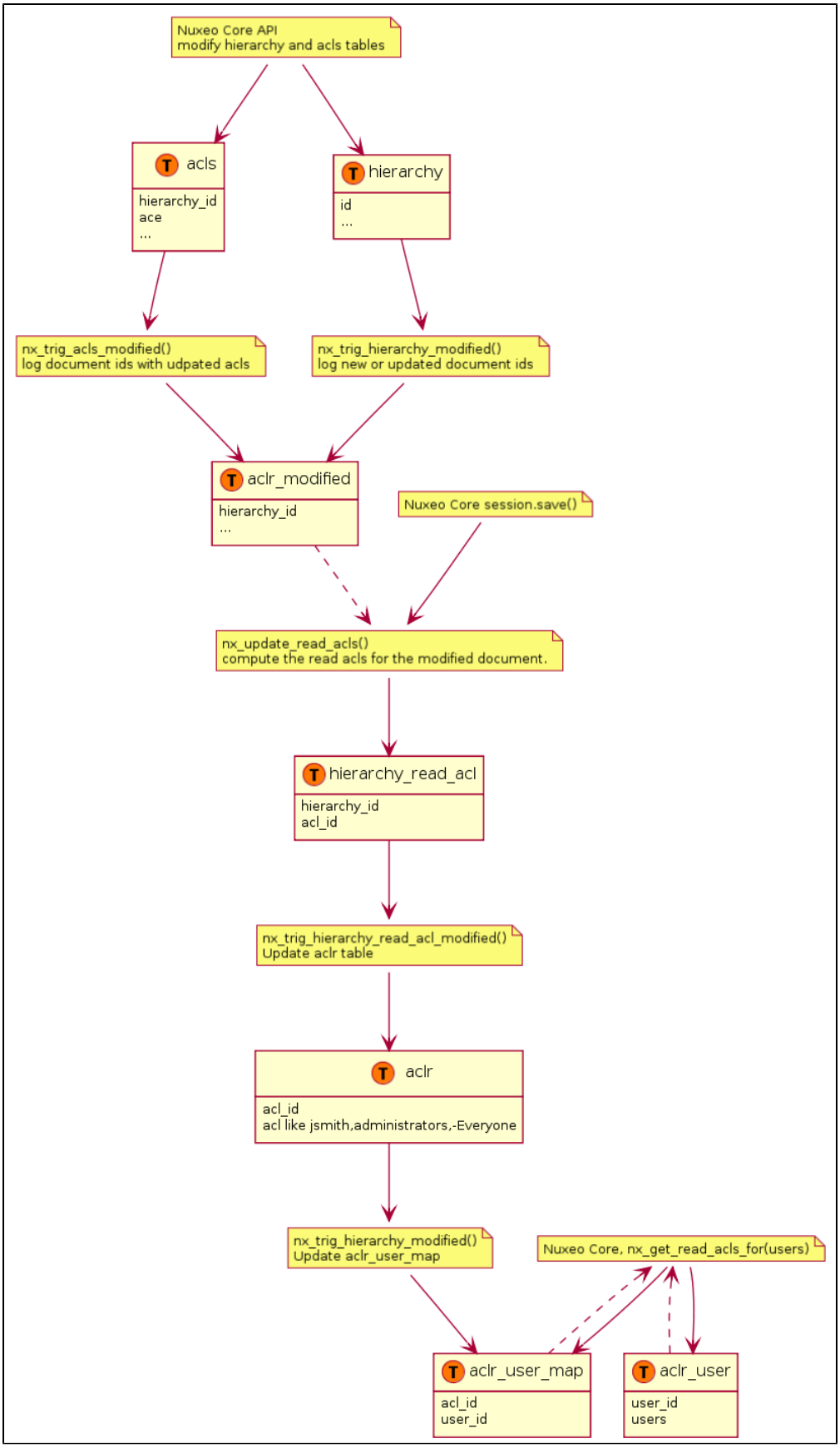
`hierarchy_id`	**`is_new`**
5678	FALSE
5690	TRUE

Note that:

- `id` is a reference to `hierarchy.id` but does not use a `FOREIGN KEY` for speed reasons,
- `is_new` is false for an ACL modification (which has impact on the document's children), and true for a new document creation (where the merged ACL has to be computed).

This table is filled while a set of ACL modifications are in progress, and when the Nuxeo session is saved the stored procedure `NX_UPDATE_READ_ACLS` is called to recompute what's needed according to `hierarchy_modified_acl`, which is then emptied.

Since 5.4.2 for PostgreSQL and since 5.5 for Oracle and MS SQL Server there is a new enhancement to be more efficient in read/write concurrency. Instead of flushing the list of read ACL per user when a new ACL is added, the list is updated. This is done using database triggers. Note that some tables have been renamed and prefixed by aclr_ (for ACL Read). Following is a big picture of the trigger processing:



In this section

Examples of SQL Generated by VCS

Request all Documents as Administrator

NXQL

```
SELECT * FROM Document
```

In this section

SQL (PostgreSQL dialect)

```
-- 1/ Get the result list (only ids)
SELECT _C1 FROM (
  SELECT hierarchy.id AS _C1
  FROM hierarchy
  WHERE ((hierarchy.primarytype IN ($1, ... $58)))
UNION ALL
  SELECT _H.id AS _C1 FROM hierarchy _H
  JOIN proxies ON _H.id = proxies.id
  JOIN hierarchy ON proxies.targetid = hierarchy.id
  WHERE ((hierarchy.primarytype IN ($59, ... $116)))) AS _T
LIMIT 201 OFFSET 0

-- 2/ load hierarchy fragment for the 12 documents
SELECT id, parentid, pos, name, isproperty, primarytype, ...
FROM hierarchy
WHERE id IN ($1, $2, $3, $4, $5, $6, $7, $8, $9, $10, $11, $12)

-- 3/ load prefetch dublincore fragment
SELECT id, creator, source,nature, created, description, ...
FROM dublincore
WHERE id IN ($1, $2,$3, $4, $5, $6, $7, $8, $9, $10, $11, $12)

-- 4/ load dublincore multi valued contributors fields
SELECT id, item FROM dc_contributors
WHERE id IN ($1, $2, $3, $4, $5, $6, $7, $8, $9, $10, $11, $12)
ORDER BY id, pos

-- 5/ load other fragments dc_subject, misc

-- 6/ load ACL
SELECT id, name, grant, permission, user,group
FROM acls
WHERE id IN ($1, $2, $3, $4, $5, $6, $7, $8, $9,$10, $11, $12)
ORDER BY id, pos
```

The main request use a "UNION ALL" to include proxies in the results. If you don't need the proxies you can add a "`AND ecm:isProxy = 0`" clause to reduce the size of the query.
Note that requests to load fragments (steps 2 to 6) are not needed if the rows are already in the cache.

Note that only prefetched properties are loaded. If you need to access a property that is not prefetched for all your documents, you will have an extra database access for each documents (lazy loading).

There is LIMIT because Page Provider for navigation use paging by default. If you have more than 200 documents in a folder you will not see the total size of results. See https://jira.nuxeo.com/browse/NXP-9494 for more information.

List Children of a Folder Ordered by title

NXQL

```
SELECT * FROM Document
WHERE ecm:parentId = ? AND
      ecm:isCheckedInVersion = 0 AND
      ecm:mixinType != 'HiddenInNavigation' AND
      ecm:currentLifeCycleState != 'deleted'
-- defaultSortColumn=dc:title
```

SQL

```
SELECT _C1, _C2 FROM (
  SELECT hierarchy.id AS _C1, _F1.title AS _C2
  FROM hierarchy
  LEFT JOIN dublincore _F1 ON hierarchy.id = _F1.id
  LEFT JOIN misc _F2 ON hierarchy.id = _F2.id
  JOIN hierarchy_read_acl _RACL ON hierarchy.id = _RACL.id
  WHERE ((hierarchy.primarytype IN ($1, ... , $33)) AND
        (hierarchy.parentid = $34) AND
        (hierarchy.isversion IS NULL) AND
        (_F2.lifecyclestate <> $35)) AND
        _RACL.acl_id IN (SELECT * FROM nx_get_read_acls_for($36))
UNION ALL
  -- same select for proxies
ORDER BY _C2
LIMIT 201 OFFSET 0
```

Select on a Complex Type

NXQL

```
SELECT * FROM Document WHERE files/*/file/name LIKE '%.jpg'
```

SQL

```
SELECT DISTINCT _C1 FROM (
  SELECT hierarchy.id AS _C1
  FROM hierarchy
  LEFT JOIN hierarchy _H1 ON hierarchy.id = _H1.parentid AND _H1.name = $1
  LEFT JOIN hierarchy _H2 ON _H1.id = _H2.parentid AND _H2.name = $2
  LEFT JOIN content _F1 ON _H2.id = _F1.id
  WHERE  ((hierarchy.primarytype IN ($3, ... $60)) AND
   (_F1.name LIKE $61))
  UNION ALL
  -- same select for proxies AS _T
 LIMIT 201 OFFSET 0
-- parameters: $1 = 'files', $2 = 'file' .. $61 = '%.jpg'
```

Related Documentation

- VCS Architecture
- Configuring MS SQL Server
- Configuring PostgreSQL

Java Data Structures and Caching

Here is a list of Java objects that hold data:

- **Row:** It holds a single database row using a map (or a list of value for a multi-valued properties).
- **Fragment:** It is a Row with a state, the original data are kept to pinpoint dirty fields that will need to be synchronized with the database. There are two kind of fragments: `SimpleFragment` to hold single database row and `CollectionFragment` to hold multi-valued fields. Fragment and Rows manipulates non-typed data (Serializable).
- **Node**: It holds a map of fragments and it gives access to typed property.
- **Selection:** It holds a list of IDs for a node like the list of children, versions or proxies.
- **DocumentModel**: The high level document representation, it uses a Node and has knowledge about rights, proxies, versions.

When a session is manipulating documents, the underlying Row objects are loaded, updated, deleted using a Mapper.

When a session is saved, the Mapper send SQL DML instructions in batch to minimize database round trip.

The main database caching is done at the Row level.

When performing a NXQL query, the result list of IDs is not cached. Only the database rows needed to represent the documents are cached.

After a commit the session sends cache invalidation to other sessions (or to other Nuxeo instances when in cluster mode). Before starting a new transaction the session processes the invalidation to update its cache.

The default cache implementation uses a cache per session. The cache is done with Java SoftReference map. This means that cache values can be garbage collected on memory pressure. The cache size depends on the size of the JVM heap and on the memory pressure.

The cache implementation is pluggable so it is possible to try other strategies like having an common cache shared by all sessions. There is a beta implementation here: https://github.com/bdelbosc/nuxeo-core-ehcache-mapper/.

The Selection (list of children, proxies or versions) are also cached using SoftReference at the session level.

Both Row and Selection caches expose metrics so it is possible to get the cache hit ratio.

Related documentation

- VCS Architecture
- Performance Recommendations

Performance Recommendations

- Check that common properties are set as prefetched.
- If you don't want to match proxies in your query add a `AND ecm:isProxy = 0` clause.
- f you don't use proxy at all deactivate them at the repository level inside the `<repository>` tag add

```
<proxies enabled="false" />
```

- If you are doing NXQL query that involve custom schema you may need to add custom index to make the request efficient.
- Use groups to manage ACL. Adding a user to a group is free, but adding a user in an ACL at the root level has a cost because optimized read ACLs need to be recomputed.
- Consider disabling the OS swapping (sudo swapoff -a) or try to lower the swapiness (vm .swappiness = 1)
- Check the network latency between the application and the database.
- Configure ImageMagick to use a single thread.
- Monitor everything, JVM, GC, VCS cache hit ratio, database, system.

Related pages in the developer documentation

- How to track the performances of your platform
- Search Results Optimizations

Related pages in the administration documentation
• Metrics and Monitoring • Nuxeo Clustering Configuration

Workflow Engine

The Nuxeo Platform embeds a powerful workflow engine technically called "Content Routing". This workflow engine provides usual features you would expect from a workflow engine and leverages the main modules of the platform: Repository, Automation service, layouts for all the user interaction and process implementation. You need to understand correctly those concepts before playing with the workflow engine.

A workflow is conceptually defined using a graph. Workflow graphs are configured from Nuxeo Studio.

The workflow engine provides means to implement most of BPMN concepts: Fork, merge, decision, branching point, exclusive, inclusive, looping, human tasks, services tasks, multiple instances, events, data objects, subprocess, join. Note that those standard concepts are not all exposed as is on the graph editor, but can still be implemented leveraging what is provided.

The workflow engine provides high level features regarding task management such as filterable tasks lists, reminders, task reassignment, task delegation, task reminders.

Task and workflow REST endpoints are coming soon, stay tuned.

You can use this workflow engine for case management projects, for form digitization, complex documents validation, signature and publishing processes and more!

You will find more information on the customization and development workflow section subpages and practical tutorials on the Studio documentation.

Authentication and Identity Service

This page gives a general idea on how authentication is plugged into the platform.
Authentication and user management is very pluggable so that the Nuxeo Platform can:

- Integrate with a Single Sign On system,
- Integrate with an external source of users and groups,
- Integrate easily any specific business logic tied to user management.

For that the Authentication and Identity services are separated in several components that can all be configured.

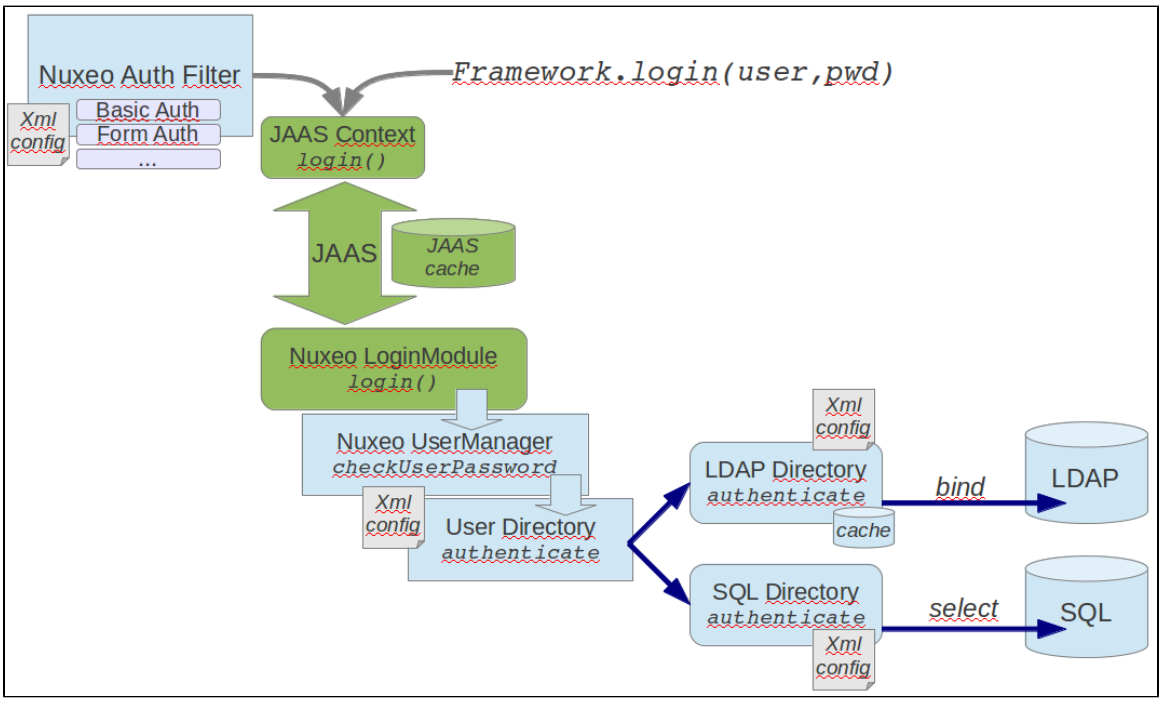
Actors in Authentication and Identity management

The different authentication and identity management actors are:

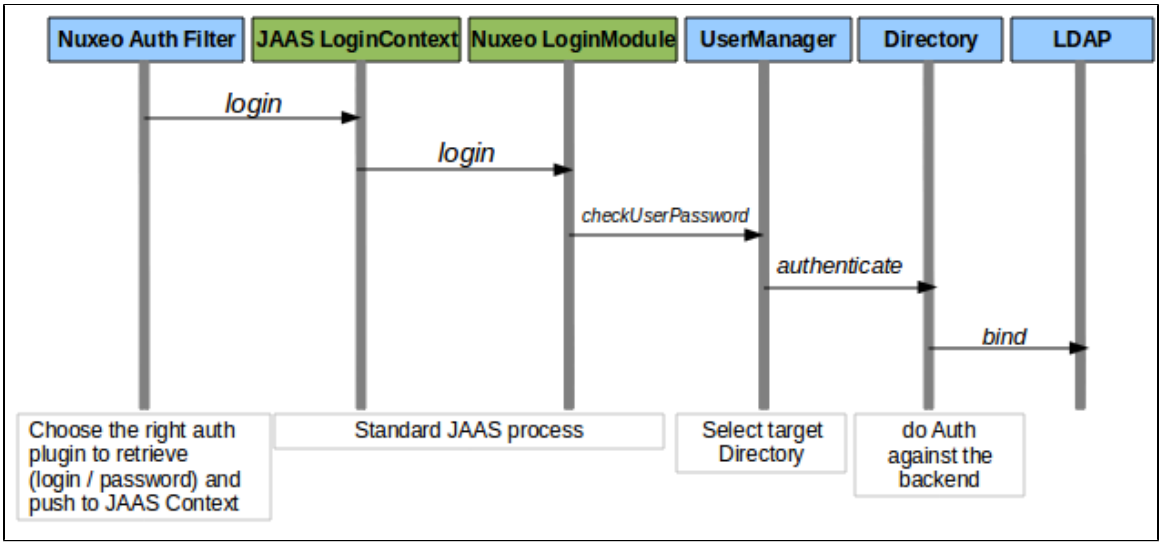
- **Nuxeo Authentication Filter**
  It protects HTTP access to Nuxeo resources.
  It will select and use Authenticator to retrieve credentials (login/password, certificate ...).
  It pushes identity in JAAS Stack.
- **Nuxeo Authenticators**
  These are pluggable modules that encapsulate the logic for getting user's creadentials.
  - They use HTTP Basic Auth.
  - They display a Login/Password form.
  - They redirect to a SSL server and then retrieve a token.
- **JAAS and Login Module**
  This is the standard Java Security framework.
  The default LoginModule uses a LoginModulePlugins to do the credential validation.
  The default LoginModule Plugin will delegate credential validation to the UserManager.

In this section

- **UserManager**
  This is the service responsible for managing Users and Groups. The underlying storage are accessed via the Directory API indirection.
- **Directories**
  Directories provide access to users and groups via LDAP, SQL or any custom implementation. They also provide a validated API to validate the credential against the user source.



The initial identification can be done at Java level via JAAS or at HTTP level via a dedicated filter. The filter is pluggable so that the way of retrieving credentials can be an adapter to the target system. The JAAS login module is also pluggable so that you can define how the credentials are validated. By default, credentials are validated against directory that use LDAP, SQL or an external application.



You will find more information on the Customization and Development section dedicated to user management.

About the Directory Abstraction

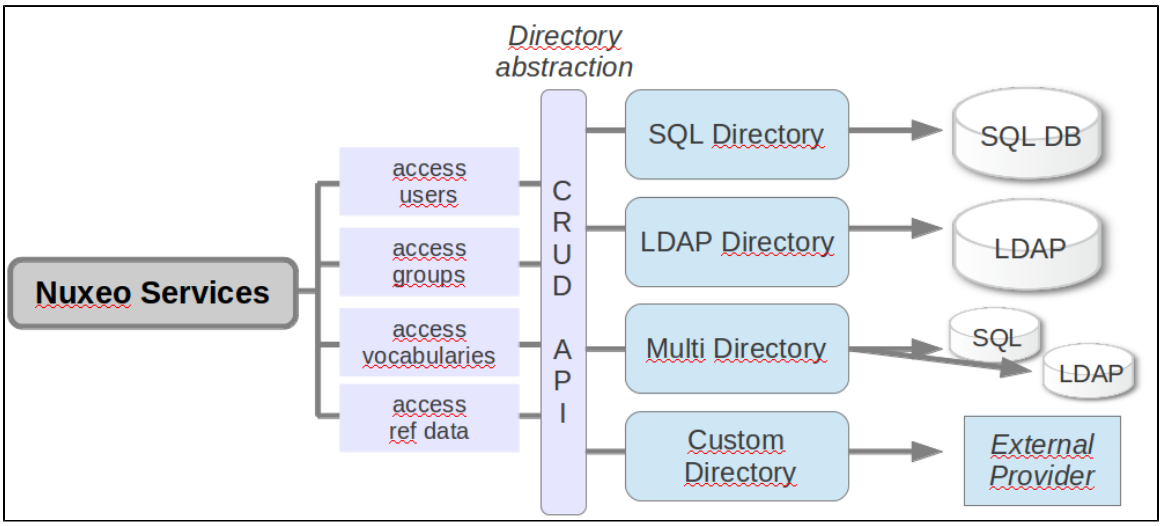
The default LoginModule will delegate Authentication the UserManager that, by default, will delegate this to Directories.

Directories will be used to:

- validate user credentials,
- retrieve user information,
- retrieve groups.

The actual directory implementation can be:

- SQL,
- LDAP (use a LDAP bind for authentication),
- Multi-Directory (wrap several directories in once),
- Custom (ex: wrap a WebService).



SSO Integration Strategies

In the case of SSO integration, the user will be authenticated by a third party service and Nuxeo should only:

- verify that the user has indeed been authenticated,
- retrieve information.

It means that at UserManager level, the validateUserPassword won't be called: UserManager will just check that the user exists so that a Principal can be created for the JAAS stack.

This explains why in most SSO plugin configuration, the target LoginModulePlugin is TrustedLM: Trusted = no need to validate credential, the

SSO server already did it.

Integrating a Ticket Based SSO Server

The Authenticator plugin will be in charge of:

- redirecting the user to the SSO server if needed;
- intercept the ticket:
  - validate it against the SSO server,
  - retrieve information about the user;

The resulting identify will be pushed to JAAS Stack.

CAS is a typical example of such an integration.

Integrating a Proxy SSO Server

In this case the Authenticator plugin will:

- check for a given header in the HTTP request;
- decrypt and validate the header token against the SSO server;

The resulting identify will be pushed to JAAS Stack.

Portal SSO is a typical example of such an integration.

Creating Users on the Fly

For some use cases, you may need to create users on the fly on the Nuxeo Platform side.

You can do this from the Authenticator which, after getting user information from the external server, can create the corresponding entry using the UserManager.

Shibboleth is a typical example of such an implementation.

Integrating SAML

CAS Server can work with SAML using additional module so this is one option.

Integrating with a Webservice Based Identify Provider

You can build a custom directory that will wrap your webservice.

You can use this sample as a starting point.

Related topics in this documentation

- User Management
- Authentication

Related topics in other documentation

- Configuring User & Group Storage and Authentication
- Authentication, users and groups

Platform APIs

This page presents the main APIs and protocols available to integrate the Nuxeo Platform with the IT environment.

Server-Side Customization

When using the extension points mechanism is not enough for customizing your Nuxeo server, you can use the local Java API to extend Nuxeo. As long as you are in the same JVM as the one in which Nuxeo server is started, you have access to the whole Java API. Read the Components and services section for better understanding and see the complete list of exposed Java services on the Nuxeo Platform Explorer. You can also refer to Nuxeo javadoc.

Content Automation is also available for implementing sever-side processing. Automation is the implementation of the Command pattern on top of our previously described Java API. It has a specific sub-section in this Architecture section.

For more information about customizing the Nuxeo server and the user of the Java API, please see the Customization and Development and Advanced topics sections.

In this section

REST API

Concepts

Nuxeo provides a complete API accessible via HTTP and also provides client SDKs in Java, JavaScript, Python, PHP, iOS and Android. This API is the best channel to use when you want to integrate remotely with the Nuxeo repository: Portals access, workflow engines, custom application JavaScript based, etc. This API has several endpoints:

- **Resources endpoints,** for doing CRUD on resources in a 100% REST style.
  Multiple resources are exposed via this API (See the reference documentation page for the existing REST URLs and JSON resources):
  - Documents (`/nuxeo/api/v1/id/{docId} or /nuxeo/api/v1/path/{path}`): to do CRUD on documents (including paginated search);
  - Users (`/nuxeo/api/v1/user/{userId}`): to do CRUD on users;
  - Groups (`/nuxeo/api/v1/group/{groupId}`): to do CRUD on groups;
  - Directories (`/nuxeo/api/v1/directory/{directoryId}`): to do CRUD on directories;
  - Automation (`/nuxeo/api/v1/automation/{Operation id}`): to call a "command", i.e. an operation or chain of operations deployed on the server. This is the main way of exposing the platform services remotely;
  - Workflow and Task endpoints are to be implemented soon!
- A **commands endpoint**, that exposes all the operations of the Automation module offering more than 100 commands for processing remotely the resources. The framework makes it very easy to add a new Java custom operation for completing the API if you miss something, and to chain operations server-side using Nuxeo Studio, so as to expose a coarse-grained API that fits your business logic.

The Nuxeo REST API offers several nice additional features compared to a standard REST API:

- Possibility to call a command on a resource (see example below);
- Possibility to ask for more information when receiving the resources via some request headers, in order to optimize the number of requests you have to do (Ex: Receiving all the children of a document at the same time you receive a document, or receiving all the parents, or all the tasks, ...);
- Possibility to use various "adapters" that will "transform" the resources that are returned;
- Possibility to add new endpoints.

Quick Examples

A blog post has been published presenting all the concepts of the Resources endpoints with some nice examples. We provide here some sample ready-to-play cURL requests to try against the Nuxeo demo site (http://demo.nuxeo.com Administrator/Administrator).

1. Here is how to create a new File document type on the Nuxeo demo instance, right under the default domain (you can copy, paste and test):

```
curl -X POST -H "Content-Type: application/json" -u
Administrator:Administrator -d '{ "entity-type": "document", "name":"newDoc",
"type": "File","properties": { "dc:title": "Specifications", "dc:description":
"Created via a so cool and simple REST API", "common:icon": "/icons/file.gif",
"common:icon-expanded": null, "common:size": null}}'
http://demo.nuxeo.com/nuxeo/api/v1/path/default-domain
```

2. You can get the new resource doing a standard GET (actually the JSON object was already returned in previous response):

```
curl -X GET -u Administrator:Administrator
http://demo.nuxeo.com/nuxeo/api/v1/path/default-domain/newDoc
```

3. Now, "lock" this document we have just created by calling an Automation operation from command API on the document resource.

```
curl -X POST -H "Content-Type: application/json+nxrequest"   -u
Administrator:Administrator -d '{"params":{}}'
http://demo.nuxeo.com/nuxeo/api/v1/path/default-domain/newDoc/@op/Document.Loc
k
```

   Pay attention to the Content-Type that is specific when using the `@op` adapter.

   You can check the result of your request on the web app (http://demo.nuxeo.com/nuxeo/nxpath/default@view_domains, credentials: Administrator/Administrator).
4. You can also directly call an automation operation or chain, from the "Command endpoint". Here we return all the worskpaces of the demo.nuxeo.com instance:

```
curl -H 'Content-Type:application/json+nxrequest' -X POST -d
'{"params":{"query":"SELECT * FROM Document WHERE
ecm:primaryType=\"Workspace\""},"context":{}}'   -u
Administrator:Administrator
http://demo.nuxeo.com/nuxeo/api/v1/automation/Document.Query
```

Adapters

This API also has the concept of "adapter". An adapter is a URL segment that starts with "@" and that transforms the input resource so as to return another resource. For example, using `@blob` will return the file of a document (the one stored on the property given by the next URL segment), and chaining it to `@op` will call an operation (that takes a blob in input):

```
/nuxeo/api/v1/id/{docId}/@blob/file:content/@op/Blob.ToPDF
```

Pluggable Context

It is sometimes useful to optimize the number of requests you send to the server. For that reason we provide a mechanism for requesting more information on the answer, simply by specifying the context you want in the request header.

For example, it is some times useful to get the children of a document while requesting that document. Or its parents. Or its open related workflow tasks.

The blog post "How to Retrieve a Video Storyboard Through the REST API" gives an example of how to use this mechanism and the dedicated extension point:

```
<extension
target="org.nuxeo.ecm.automation.io.services.contributor.RestContributorService"
point="contributor"> ...</extension>
```

Files Upload and Batch Processing

The Platform provides facilities for uploading binaries under a given "batch id" on the server, and then to reference that batch id when posting a document resource, or for fetching it from a custom automation chain.

For instance if you need to create a file with some binary content, first you have to upload the file into the batchManager. It's a place on the system where you can upload temporary files to bind them later.

1. For that you have to generate yourself a `batchId`, which will identify the batch : let's say `mybatchid`.

```
POST http://localhost:8080/nuxeo/site/automation/batch/upload
X-Batch-Id: mybatchid
X-File-Idx:0
X-File-Name:myFile.zip
-----------------------
The content of the file
```

   Or with curl:

```
curl -H "X-Batch-Id: mybatchid" -H "X-File-Idx:0" -H "X-File-Name:Sites.zip"
-F file=@Sites.zip -u Administrator:Administrator
http://localhost:8080/nuxeo/site/automation/batch/upload
```

2. You may verify the content of your batch with the following request.

```
GET http://localhost:8080/nuxeo/site/automation/batch/files/mybatchid
[{"name":"Sites.zip","size":115090}]
```

3. Next you have to create a document of type File and attach the Blob to it by using the specific syntax on the `file:content` property.

```
POST
http://localhost:8080/nuxeo/api/v1/path/default-domain/workspaces/myworkspace
{
  "entity-type": "document",
  "name":"myNewDoc",
  "type": "File",
  "properties" : {
    "dc:title":"My new doc",
    "file:content": {
      "upload-batch":"mybatchid",
      "upload-fileId":"0"
    }
  }
}
```

   Or with curl:

```
curl -X POST -H "Content-Type: application/json" -u
Administrator:Administrator -d '{  "entity-type": "document",
"name":"myNewDoc", "type": "File",  "properties" : { "dc:title":"My new
doc","file:content": {"upload-batch":"mybatchid","upload-fileId":"0"}}}'
http://localhost:8080/nuxeo/api/v1/path/default-domain/workspaces/myworkspace
```

4. Finally you now can access the content of your file by pointing to the following resource:

```
GET
http://localhost:8080/nuxeo/api/v1/path/default-domain/workspaces/myworkspace/
myNewDoc/@blob/file:content
```

Available Client SDKs

We provide several client SDKs for making it even easier to integrate with the Nuxeo Platform.

- Java client,
- JavaScript client,
- iOS client,
- Android client,
- PHP client (partial implementation).

See the Nuxeo Rest API page of the Customization and Development section for a complete documentation.

SOAP Bridge

The Platform doesn't provide a complete SOAP binding of its API. But we provide a bridge and a guideline for implementing the web services you need. You can also use the SOAP binding of CMIS API.

Compatibility APIs: CMIS and WebDAV

The Nuxeo Platform currently supports the following APIs, which gives a content oriented view of the repository:

- **CMIS**, gives you access to a big subset of the data managed by Nuxeo. See the CMIS section for more information.
- **WebDAV** mainly maps Nuxeo's content as a file-system (with all the associated limitations).

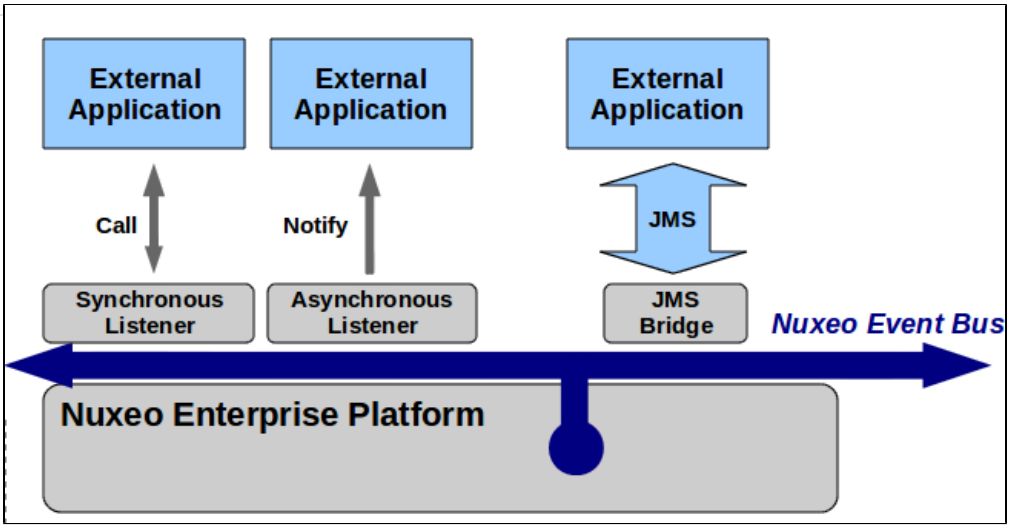
These protocols may be useful in several use cases:

- Desktop integration,
- To allow a portal or a WCM solution to access Nuxeo's content,
- To plug to a digitization chain,
- ...

Windows virtual drive use case
Regarding WebDAV interface gives good results in using it with WebDAV sync tools as well as mobile client, is not recommended for a Windows virtual drive use case, as the implementation of web folders makes it difficult (impossible) to be compatible with each and every versions that MS released.

Event Bus

When you need to integrate some features of an external application into Nuxeo, or want Nuxeo to push data into an external application, using the Nuxeo event system is usually a good solution.



The system allows to contribute event listeners (Java classes) or event handlers (defined in Studio) that subscribe to Platform events such as user login/logout, document creation, document update, document move, workflow started, etc. Those Java classes or automation chains are then executed either synchronously in the transaction, or synchronously out of the transaction, or asynchronously.

The event bus system is a nice way to implement some triggers in the document repository, and to let the repository notify other elements of your IT system of the changes that happen in the repository.

You will find more information on the Customization and development section dedicated to event listeners.

Data Lists and Directories

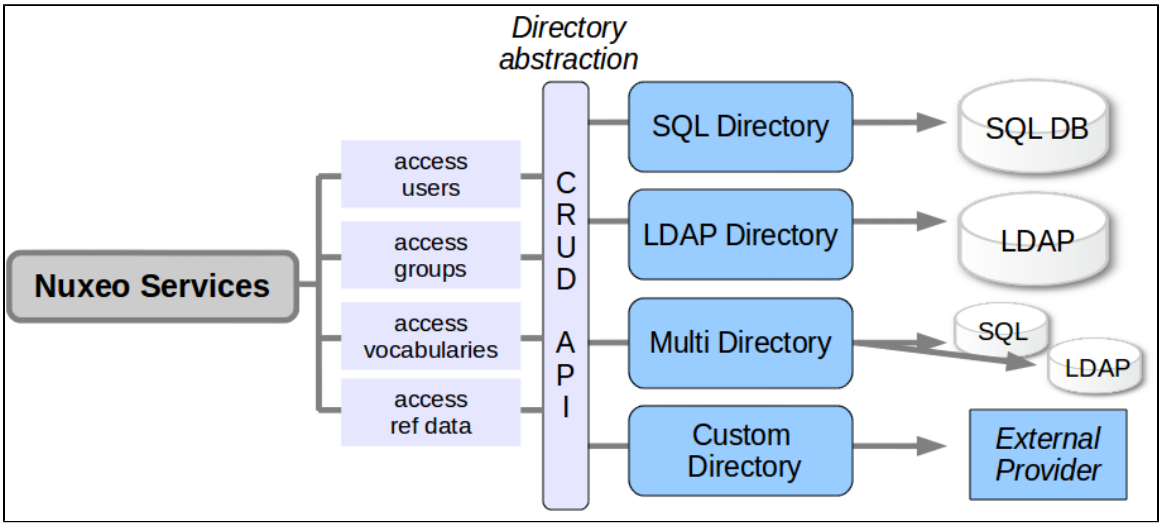
We explain here the philosophy of the directories: A mean to expose to your app some external data.

Inside the Nuxeo Platform, directories are used to provide a abstraction on all referential data that can be manipulated inside the application.

These data can be:

- Users,
- Groups of users,
- Fixed list of values (vocabularies),
- Roles,
- ...

Basically we try to map all data that can be manipulated like record via directories. For that, directories provide a simple CRUD API and an abstraction on the actual implementation. This means that the services of the platform do not have to worry about where and how the data is stored, they just access the API.



Directories comes with several implementations:

- SQL directories that can map SQL tables,
- LDAP directories that can map a LDAP server,
- Multi-directory that allow to combine several directories into a single one.

A frequent use case is also to use the directory abstraction to map a given webservice or application that manages centralized data. (We provide a connector sample for that.)

Content Automation

Content Automation is a Nuxeo service that exposes common actions you do on a Nuxeo application as atomic operations so that one can assemble them to create complex business rules and logic, without writing any Java code.

In other words, content automation provides a high level API over Nuxeo services - an API made of operations that can be assembled in complex automation chains (or macro operations). These operations can be called locally in Java, or remotely as being exposed via the REST API.
The main goal of automation is to enable end users to rapidly build complex business logic without writing any Java code — just by assembling the built-in set of atomic operations into complex chains and then plugging these chains inside Nuxeo as UI actions, event handlers, REST bindings, etc.

You can also create new atomic operations (write a Java class that defines an operation) and contribute them to the set of built-in operations. To define an automation chain, you just need to write an XML contribution that describes the chain by listing each operation in the chain along with the parameter values that will be used to execute the operation. Since 5.7.2, you can define parameters for this chain, injected into the automation context to be fetched at execution time. If you need to define dynamic operation parameters (whose value will be computed at runtime when the operation is executed) you can use scripting (e.g. EL syntax) to fetch the actual parameter value at execution time.

You can easily and graphically create new chains and actions using Nuxeo Studio.

In this section

What is an Operation?

From an end-user point of view: an operation is an action that can be triggered by the user either directly through the user interface, or by responding to an event, or by a REST call to a remote server.

The operations an user can invoke usually deal with the document repository (like creating or updating documents), but they can also do some other tasks like sending emails, converting blobs, etc.

The automation service already provides tens of frequent operations that you may need to build your business logic. More operations can be contributed using a Nuxeo extension point. This involves of course writing the right Java code to implement the operation logic.

From a developer point of view: an operation is a Java class annotated using the right annotations and providing a one or more methods that are doing the actual work.

For more information about how to write and contribute a new operation, see Contributing an operation.

What is an Automation Chain?

The power of operations is that operations can be chained into a sort of macro operation that is composed of atomic operations and which executes each operation in turn by using an operation output as the input of the next operation. This way you can for example construct an automation chain that creates a document, then attaches a blob to the document, then publishes it and so on. Each operation in the chain does the required step by working on the input of the previous operation and when finished outputs a result that will be used by the next operation as its input.

Since 5.7.2, all chains can contain parameters as operation to be used from the automation context along their execution.

This is called in Nuxeo Automation an **automation chain**. Automation chains give you the possibility to build complex business logic only by assembling atomic operations that are provided by the server. Thus, you can script your business logic using automation chains which, thank to Nuxeo Studio, can be done by using drag and drop (without coding anything or even writing XML extension files).

If you are a developer and don't want to use Nuxeo Studio, you can check the Contributing an Automation Chain page to see how you can define and contribute a new operation chain using Nuxeo extension points.

Working with Operations

This section is more about how operations work and how they can be chained to obtain working automation chains.

We've seen that an operation works on an input object by doing something (like updating this object or creating new objects) and at the end it outputs the operation result. When you are constructing a chain of operations, this result will be used as the input of the next operation. The last output in the chain will be the output of the chain itself. As you noticed, an operation works on an input so you should provide an initial input to start an operation (or an operation chain).

More, as there is an Run Chain atomic operation which takes as the argument the name of the chain to execute, you can create very complex chains that can call sub-chains to execute some particular steps in the chain.

In a chain, every operation has a cause and an effect. The effect of one operation in a chain is the cause of the next event. The initial cause is the user action (doing something with a document for example), the final effect is what the chain is assumed to do.

An atomic operation can be viewed as a finite automation chain with an initial cause (the action that triggered it) and having the effect of the execution of the operation itself. This way you can chain as many causes and effects you want. You can actually create your ECM universe using operations.

Now, let's discuss about the bricks that compose an operation:

- An operation has an input (provided by the cause).
- An operation may have zero or more parameters (used to parametrize the way an operation is behaving).
- An operation has an output (that can be used by the next operation in the chain as the input).

The Operation Input

The operation input can be a Document or a Blob (i.e. a file).

The input is provided by the execution context, either by getting the input from the user action (in the case of a single operation or for the first operation in the chain), or from the output of the previous operation when executing a chain.

There are some special operations that don't need any input. For example you may want to run a query in the repository. In this case, you don't need an input for your query operation. Thus, operations can accept void inputs (or nothing as input). To pass a void input to an operation, just use a null value as the input. If an operation doesn't expect any input (i.e, void input) and an input is given, the input will be ignored.

The Operation Parameters

An operation can define parameters to be able to modify its execution at runtime depending on those parameter values.

Any parameter value can be expressed as a string. The string will be converted to the right type at runtime if possible. If not possible an exception is thrown.

There are several types of predefined parameters:

- string: any string,
- boolean: a boolean parameter,
- integer: an integer number,
- float: a floating point number,
- date: a date (in W3C format if is specified as a string),
- resource: an URL to a resource,
- properties: a Java properties content (key=value pairs separated by new lines),
- document: a Nuxeo Document (use its absolute PATH or its UID when expressing it as a string),
- blob: a Nuxeo blob (the raw content of the blob in the case of a REST invocation),
- documents: a list of documents,
- bloblist: a list of blobs,
- any other object that is convertible from a string: you can register new object converters trough the `adapters` extension point of the `org.nuxeo.ecm.core.operation.OperationServiceComponent` component;
- an expression: this represents a MVEL expression (which is compatible with basic EL expressions) that can output dynamic values. When using expressions you must prepend it with the prefix _expr:_.
  Example:

  ```
  expr:Document['dc:title']
  ```

  For the complete list of objects and functions available in an expression see Nuxeo Studio.
- an expression template: this is the same as an expression but it will be interpreted as a string (by doing variable substitution). This is very useful when you want to create expressions like this:

```
expr: SELECT * FROM Document WHERE dc:title LIKE @{mytitle}
```

where `mytitle` is a variable name that will be substituted with its string form.
You can notice that you still need to prepend your template string with an `expr:` prefix.

The Operation Output

The operation output is either a Document, a Blob or void (as the input).

In some rare cases you may want your operation to not return anything (a void operation). For example your operation may send an email without returning anything. When an operation is returning void (i.e. nothing), then a null Java object will be returned .

As said before, the output of an operation is the input of the next operation when running in a chain.

Contributing New Input/Output Types

Since 5.4.2, you can extend the input/output types by contributing the new marshalling logic to automation.

Marshalling and operation binding logic is selected client and server side using the JSON type name. At this stage, we're always using the Java type simple name in lowercase. This makes the operation binding logic being happy.

The logic you need to provide is as follow :

- the JSON type name,
- the POJO class,
- a writing method that puts data extracted from the POJO object into the JSON object,
- a reading method that gets data from the JSON object and builds a POJO object from it,
- a reference builder that extracts the server reference from a POJO object,
- a reference resolver that provides access to a POJO object giving a server reference.

Server and client do not share classes, so you need to provide two marshalling implementation class.

Server side, you should provide a codec. The implementation class is to be contributed to the automation server component using the `codecs` extension point.

```
<extension target="org.nuxeo.ecm.automation.server.AutomationServer"
        point="codecs">
        <codec class="org.nuxeo.ecm.automation.server.test.MyObjectCodec" />
  </extension>
```

`MyObjectCodec` class should extend `org.nuxeo.ecm.automation.server.jaxrs.io.ObjectCodec`. Most common codecs provided by default into Nuxeo server are implemented into `org.nuxeo.ecm.automation.server.jaxrs.io.ObjectCodecService`.

Client side, you should implement the `org.nuxeo.ecm.automation.client.jaxrs.spi.JsonMarshaller<T>` interface. The implementation class is to be registered to the automation client marshalling framework by invoking the static method `org.nuxeo.ecm.automation.client.jaxrs.spi.AbstractAutomationClient#registerPojoMarshaller(Clazz marshaller)`.

Here is an example of the org.nuxeo.ecm.automation.client.jaxrs.spi.marshallers.BooleanMarshaller.

Using Scripting Expression in Operations

Operations can be parametrized using MVEL scripting expressions. For more details about scripting you can look at the page Use of MVEL in Automation Chains.

UI Frameworks

The Nuxeo Platform proposes different technologies for the client side of the application. The choice of one technology vs. another depends on both the project's stakes and its context.
You mainly have two strategies for the UI you will propose to your users:

- Either customize the existing web application, which is JSF/SEAM based. You'll be able to reach a great customization point just using Nuxeo Studio, and you will also be able to extend it writing your own facelets (XHTML) templates and Seam components. In this approach, you can keep the standard layout and just add your "business" flavor from Studio, but you can also completely redesign the layout, as some of the Nuxeo Platform users did.
- Or write your own application UI with the technology of your choice, using our APIs and client SDKs.

In this section

Customizing the Back Office

The Nuxeo Platform's default web UI is based on JSF (a Java EE standard) for the UI component model, and Seam for the navigation and context management. This application is designed to be extended, at every corner: theme, layout, forms, tabs, URLs using our extension point system, and extended using the web app core technologies.

Key points:
The Nuxeo Platform's JSF interface is fully modular and very easy to customize:

- Highly configurable via Nuxeo Studio,
- Resilient: this application is the result of 7 years of continuous development and improvements. It has been tested on thousands of projects, each one bringing its own constraints.
- Great forms framework, configurable document listings, etc ...

Typical use case:

- Business application,
- Document management back-office.

More information on the Studio space and on the "Customizing the web app" section.

Front Apps

The Nuxeo Platform can be seen as a complete remote repository exposing all its features via APIs. In that case, you are the one who chooses your UI framework. Indeed, you can wrap the repository behind:

- An HTML5/JavaScript based application using for instance AngularJS , or any other JavaScript framework (AmberJS, node.js, ...), using our JavaScript client;
- A Java portal, such as Liferay, JBoss Portal, Jahia, uPortal, ... You can use Nuxeo Java client for implementing portlets that will have interactions with the repository;
- Any other technology, such as PHP, Ruby on Rails, ... using our REST API directly.

With this approach you will of course have more developments to do on the UI, but you will be free to use the technology of your choice, and to be as crazy as you want in terms of UI layout!

Nuxeo WebEngine

Nuxeo WebEngine is a light web framework provided by the Nuxeo Platform. You can use it for implementing new UIs. For that reasons, we could have put it in the "Write your own application" category, but since the framework is edited by Nuxeo, let's have a dedicated section

Technologies:

- JAX-RS style
- FreeMarker,
- Java scripting (Groovy or other).

Key points:
Nuxeo WebEngine enables web developers to easily create a customized web interface on top of the Nuxeo Platform. Nuxeo WebEngine consumes URLs and for each segment of the URL instantiates a Java object that is in charge of consuming the remaining part of the URL and also return the web page at some point.

- Simple template engine based on FreeMarker
- Direct and easy access to HTML,
- Java scripting support

Typical use case:
Nuxeo WebEngine is designed to expose the Nuxeo Platform's managed content in a web experience. In many cases, the JSF interface is used for the back-office management while Nuxeo WebEngine provides the front office interface. Furthermore, with the JAX-RS support, Nuxeo WebEngine allows rapid creation of REST applications on top of the Nuxeo Platform.

WebEngine is a light framework
Nuxeo WebEngine is a web-framework because it provides you with a way to structure how you implement the standard "MVC" pattern. But it is light because it doesn't provide any widgets library, any collection of ready to use templates, etc. Nuxeo aims at letting this framework as light as it is now and encourage users to leverage popular web framework when requiring many UI components. Actually, Nuxeo WebEngine is light enough to be easily used in parallel of those JavaScript framework.

Deployment Options

In this section, the different deployment possibilities are described.
Thanks to Nuxeo Runtime and to the bundle system, the Nuxeo Platform deployment can be adapted to your needs:

- Deploy only the bundles you really need,
- Deploy on multiple servers if needed,
- Deploy on multiple infrastructures: Tomcat, Pojo, unit tests

Simple Deployment

For a simple deployment you have to:

- Define the target Nuxeo distribution (in most of the cases Nuxeo DM + some extra plugins),
- Define the target deployment platform:
  - Prepackaged Tomcat Server (including Nuxeo + Transaction manager + JCA + Pooling),
  - Static WAR (see Nuxeo Deployment Model),
  - Embedded mode (mainly for unit tests, but also used for some client side deployment).

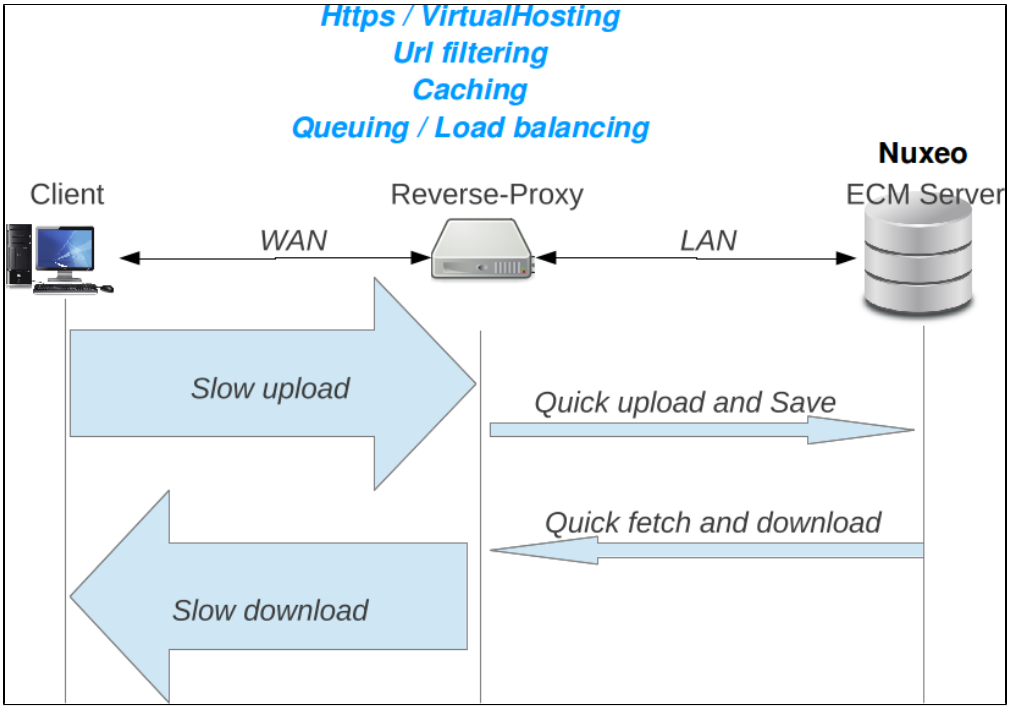
In this section

The default Tomcat packaging is actually not a bare Tomcat. The Tomcat that is shipped with Nuxeo contains:

- A JTA Transaction Manager,
- A JCA pool manager.

In most of the case, the Nuxeo server is behind a reverse proxy that is used to provide:

- HTTPS/SSL encryption,
- HTTP caching,
- URL rewritting.



Additionally, when some clients use a WAN to access the server, the reverse proxy can also be used to protect the server against slow connections that may use server side resources during a long time.

Cluster HA

In order to manage scale out and HA, the Nuxeo Platform provides a simple clustering solution.

When cluster mode is enabled, you can have several Nuxeo Platform nodes connected to the same database server. VCS cluster mode manages the required cache invalidation between the nodes. There is no need to activate any application server level cluster mode: VCS cluster mode works even without application server.

The default caching implementation uses simple JVM Memory, but we also have alternate implementation for specific use cases.

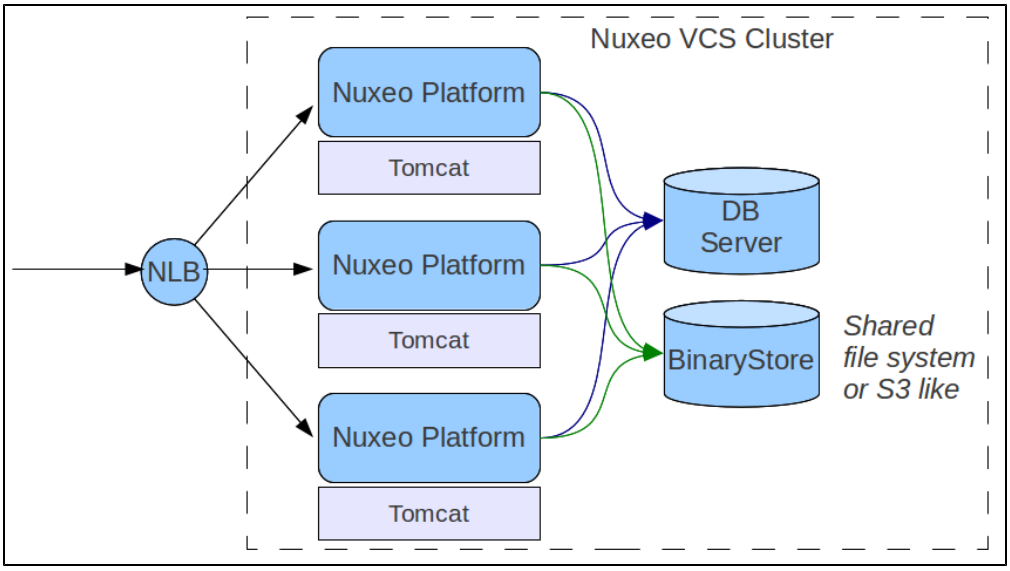
Depending on the UI framework used for presentation layer, the network load balancing may need to be stateful (JSF webapp) or stateless (WebEngine).

Typically:

- JSF Backoffice UI is stateful,
- WebEngine and HTML/JS based UI are mainly stateless.

Anyway, even with JSF:

- Authentication can be transparent if you use a SSO system,
- The Nuxeo Platform knows how to restore a JSF view from a URL (most Nuxeo JSF views are bound to REST URLs).



NB : In this architecture the Database server is still a Single Point of Failure. To correct that, the best option is to use Clustering + Database replication has described in the next paragraph.

For more information, please see the page Setting up a HA Configuration Using the Nuxeo Platform and PostgreSQL.

Hot Standby (DRP)

If you want to provide a Disaster Recovery Plan, you will need to host two separated Nuxeo infrastructures and be sure you can switch from one to an another in case of problem.

The first step is to deploy two Nuxeo infrastructures on two hosting sites. These infrastructure can be mono-VM, cluster or multi-VM. The key point is to provide a way for each hosting site to have the same vision of the data:

- SQL data stored in the SQL database server,
- Filesystem data.

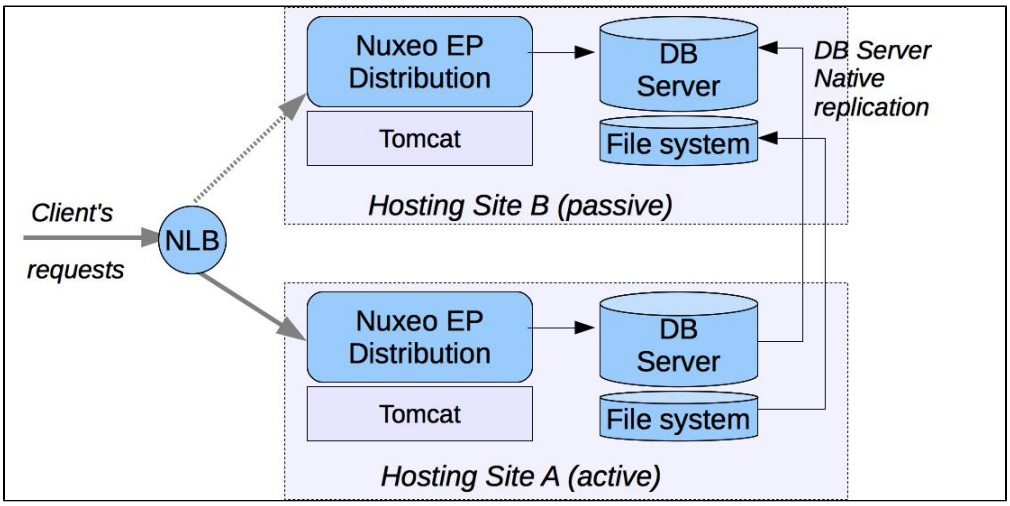
Because Nuxeo storage VCS+Filesystem is safe, you can use a replication system between the two sites. Basically, you can use the replication/standby solution provided by the database server you choose. This replication tool just has to be transactional.

For the filesystem, any replication system like RSync can be used.

Because the blobs are referenced by their digest in the database, you don't have to care about synchronization between the DB and FS. In the worst case, you will have blobs that are not referenced by the DB on the replicated site.

This kind of DRP solution has been successfully tested in production environment using:

- PosgreSQL stand-by solution (WAL shipping),
- RSync for the file system.

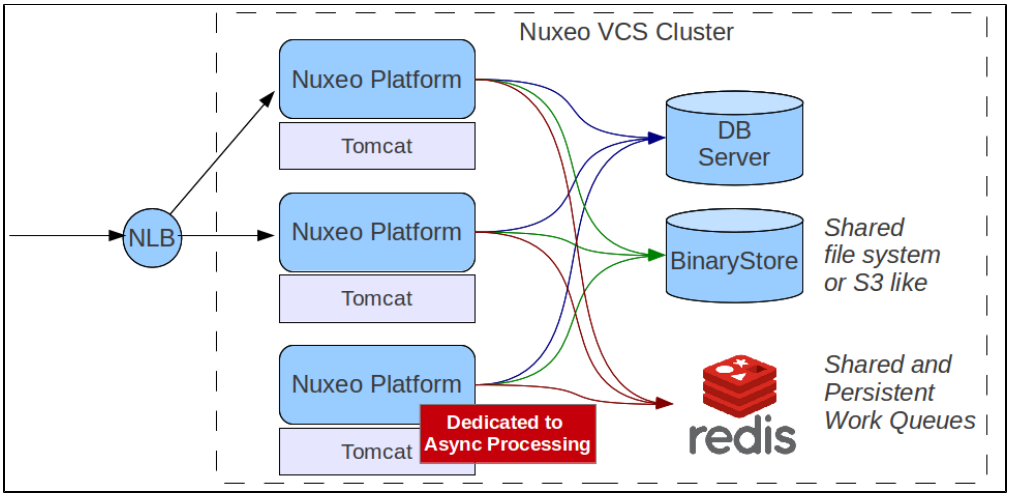


The Warm standby DB nodes are not used by Nuxeo active nodes.

Deploy Dedicated Processing Nodes

Starting with 5.8, the async tasks can be managed in a distributed way using the WorkManager and Redis. This can be used to have some nodes of the Cluster dedicated to some heavy processing like Picture or Video conversions.

Having such a demarcation between nodes is also a good way to be sure that async processing won't slow down Interactive processing.



Multi-Repository

Unlike Nuxeo Nodes, the Database server nodes can not be simply added to handle more load :

- Multi-masters DB architecture (like Oracle RAC) work but don't really provide scale out, only HA.
- For now, the Nuxeo Platform can not leverage the ReadOnly nodes because the Platform is too pluggable to be sure when we start a transaction if we will need to write or not.

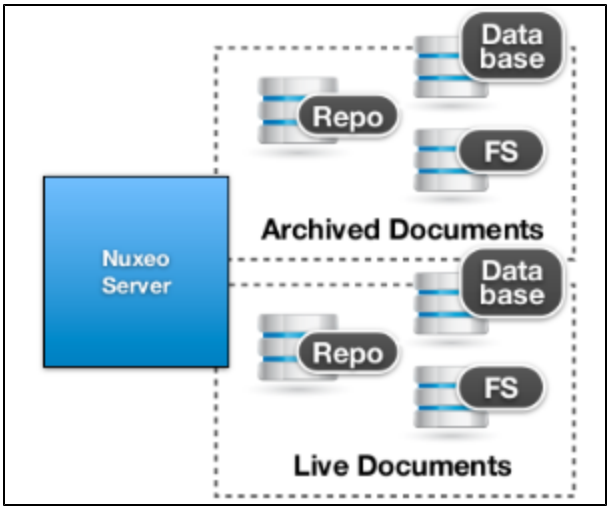
However, the Nuxeo Platform does provide a way to scale out the data: using several repositories. The idea is that a single Nuxeo Application can be bound to several repositories: Each repository being a DB instance and a BlobStore.

The repositories are like mount points:

- Default configuration is to have only one repository named "Default Repository" and mounted under /default/.
- But you can add new repositories as needed.

Using the approach allows you:

- To select your storage type according to the needs:
  - Archive:
    - Massive cheap storage for the Blob Store
    - Add indexes on the Database to make Search faster (few Write)
  - Live
    - Fast storage to make work faster
    - Fast database will few index to maximize write speed
- To select the storage according to the isolation policy:
  - Data for "clientX" goes in "repositoryX"



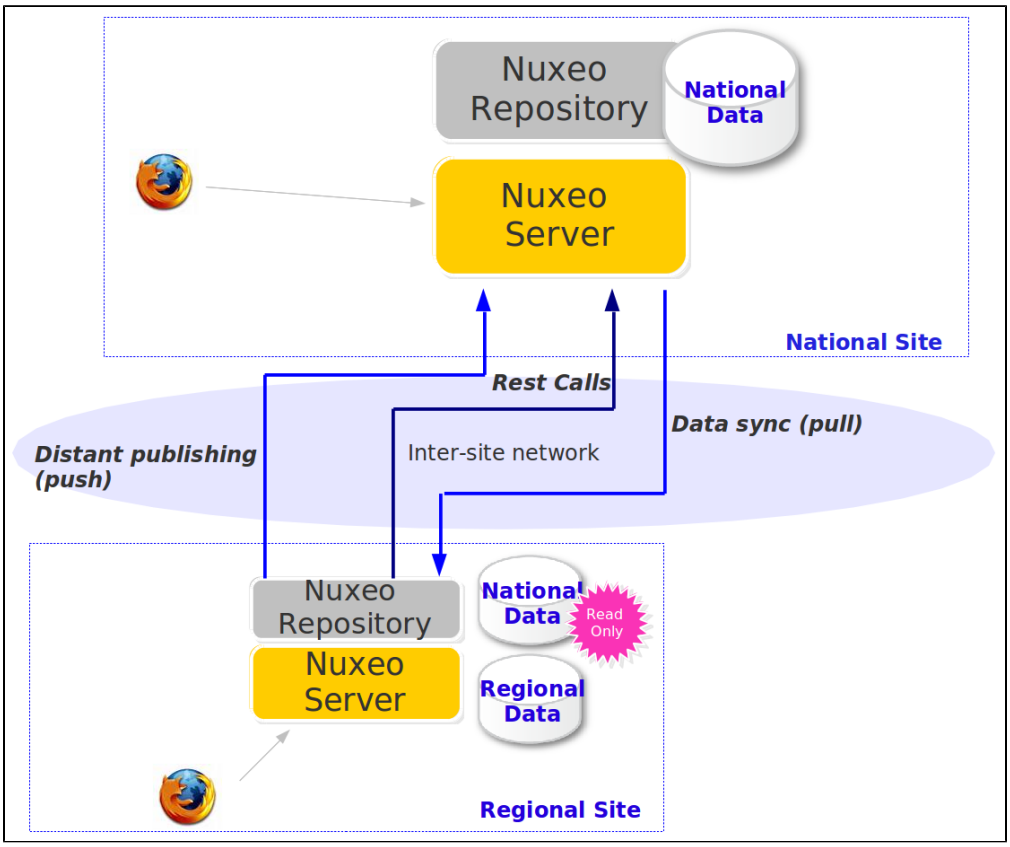
However, the multi-repository approach is not magical:

- There is no aggregated index,
- Data segregation is visible to users.

Read-Only Synchronization

The Nuxeo Platform being flexible, you can use several addons together to achieve a complex architecture:

- Use `nuxeo-platform-sync` to create a read-only copy of a remote repository;
- Use remote plublisher to push information between two distant Nuxeo instances;
- Use multi-repository support to segregate Local vs Central data.



Cloud

Nuxeo can be hosted on various Clouds and virtualization systems:

- Amazon AWS

- MS Azure
- Docker

Related documentation

- Nuxeo and Redis
- Nuxeo Clustering Configuration

Nuxeo and Redis

Principles

Nuxeo provides a Redis integration via the `nuxeo-core-redis` bundle.

The idea is that, at least for now, Redis is not a hard requirement for running the Nuxeo Platform; We simply use Redis as a backend to provide alternate implementation of some services inside the platform. However, we do provide these implementations because we think they can be useful.

Nuxeo Core Cache

Nuxeo Core Cache provides a service to declare and use caches. This cache system:

- is used in Nuxeo Directories (caching directories entries)
- is used in UserManager (caching principals)
- can be used in your custom code

The Cache has a default "in memory" implementation, but `nuxeo-core-redis` provides a an other implementation that allows:

- To have out of JVM memory cache storage.
  It opens the way to have big caches without hurting the JVM.
- To share the same cache between several Nuxeo nodes.
  In cluster mode this can increase cache efficiency.
- To manage cluster wide invalidations.
  Updating the user on one node will impact the central cache: all nodes see the exact same data.

You can configure the backend storage on a per cache basis: Directory A could use Redis while directory B could use in memory.

Work Manager

The WorkManager handles asynchronous jobs:

- Schedule Jobs and store them in queues
- Assign execution slots to queues
- Execute the jobs

In the default implementation, job queues are in the JVM memory. But this model has some limitations:

- Stacking a lot of jobs will consume JVM Memory
- In cluster mode each Nuxeo node maintains its own queue
- When a Nuxeo server is restarted, all the queued jobs are lost

`nuxeo-core-redis` provides an alternate implementation of the queuing system based on Redis:

- Jobs are then stored outside of JVM memory.
  That's why Work have to be serializable: they are serialized and stored inside Redis.
- Jobs can be shared across cluster nodes.
  This allows to dedicate some nodes to some specific processing.
- Jobs survive a Nuxeo restart.
  When Redis persistence is activated, the jobs even survive a Redis restart.

Lock Manager

By default Lock managed on documents are stored inside the repository backend.When the locking system is heavily used, it can create a lot of very small transactions against the database backend.

`nuxeo-core-redis` provides a Redis-based implementation of the locking system that is resilient as the database implementation, but easier to scale.

In this section

- Principles
- Nuxeo Core Cache
- Work Manager
- Lock Manager

Related Documentation

- Redis Configuration
- Work and WorkManager

Licenses

The Nuxeo source code is licensed under various open source licenses, all compatible with each other, non viral and not limiting redistribution. Nuxeo also uses a number of third-party libraries that come with their own licenses.

There is no non-open source "enterprise" version of these products, or, in other words, the "open source versions" are already "enterprise-ready" for our customers. We encourage our customers to purchase a support subscription to ensure that their Nuxeo instances stays fully operational as long as needed by its business users.

Note also that third-parties are perfectly allowed, under the terms of the LGPL, to create non-open source products based on Nuxeo Platform.

Nuxeo Licenses

Nuxeo developers use the following licenses described on the Copyright and license headers page:

- LGPL 2.1: GNU Lesser General Public License v2.1
- EPL 1.0: Eclipse Public License v1.0
- AL 2.0: Apache License v2.0

Third-Party Licenses

Nuxeo may use (depending on which distribution or package is installed) the following libraries. These libraries have been chosen because their licenses are compatible with those of the Nuxeo source code, open source, not viral and do not limit redistribution.

The licenses used are:

- AL 1.1: Apache License v1.1
- AL 2.0: Apache License v2.0
- BSD 2: BSD 2-Clause License
- BSD 3: BSD 3-Clause License
- CC BY 2.5: Creative Commons Attribution License 2.5
- CDDL 1.0: Common Development and Distribution License v1.0
- CDDL 1.1: Common Development and Distribution License v1.1
- EPL 1.0: Eclipse Public License v1.0
- LGPL 2.1: GNU Lesser General Public License v2.1
- LGPL 3: GNU Lesser General Public License v3
- MIT: MIT License
- W3C: W3C Software Notice and License
- PD: Public Domain (not actually a license)

Jar Name	Project	Version	License
activation-1.1.jar	JavaBeans Activation Framework	1.1	CDDL 1.0
ant-1.7.0.jar	Apache Ant	1.7.0	AL 2.0
ant-launcher-1.7.0.jar	Apache Ant	1.7.0	AL 2.0
antisamy-1.4.4.jar	AntiSamy	1.4.4	BSD 3
antlr-2.7.7.jar	ANTLR v2	2.7.7	PD

antlr-runtime-3.1.3.jar	ANTLR v3	3.1.3	BSD 3
aopalliance-1.0.jar	AOP Alliance	1.0	PD
apacheds-bootstrap-extract-1.5.1.jar	ApacheDS	1.5.1	AL 2.0
apacheds-btree-base-1.5.1.jar	ApacheDS	1.5.1	AL 2.0
apacheds-constants-1.5.1.jar	ApacheDS	1.5.1	AL 2.0
apacheds-core-1.5.1.jar	ApacheDS	1.5.1	AL 2.0
apacheds-core-shared-1.5.1.jar	ApacheDS	1.5.1	AL 2.0
apacheds-jdbm-store-1.5.1.jar	ApacheDS	1.5.1	AL 2.0
apacheds-kerberos-shared-1.5.1.jar	ApacheDS	1.5.1	AL 2.0
apacheds-protocol-shared-1.5.1.jar	ApacheDS	1.5.1	AL 2.0
apacheds-schema-bootstrap-1.5.1.jar	ApacheDS	1.5.1	AL 2.0
apacheds-schema-registries-1.5.1.jar	ApacheDS	1.5.1	AL 2.0
apacheds-utils-1.5.1.jar	ApacheDS	1.5.1	AL 2.0
arq-2.8.8.jar	Apache Jena	2.8.8	BSD 3
asm-3.0.jar	ASM	3.0	BSD 3
avalon-framework-4.1.3.jar	Apache Avalon	4.1.3	AL 1.1
backport-util-concurrent-3.1.jar	backport-util-concurrent	3.1	PD
batik-css-1.7.jar	Batik	1.7	AL 2.0
batik-ext-1.7.jar	Batik	1.7	AL 2.0
batik-util-1.7.jar	Batik	1.7	AL 2.0
bcmail-jdk15-1.45.jar	Bouncy Castle	1.45	MIT
bcprov-jdk15-1.45.jar	Bouncy Castle	1.45	MIT
c3p0-0.9.1.jar	c3p0	0.9.1	LGPL 2.1
caja-r3889.jar	Caja	3889	AL 2.0
chemistry-opencmis-client-api-0.7.0.jar	Apache Chemistry OpenCMIS	0.7.0	AL 2.0
chemistry-opencmis-client-bindings-0.7.0.jar	Apache Chemistry OpenCMIS	0.7.0	AL 2.0
chemistry-opencmis-client-impl-0.7.0.jar	Apache Chemistry OpenCMIS	0.7.0	AL 2.0
chemistry-opencmis-commons-api-0.7.0.jar	Apache Chemistry OpenCMIS	0.7.0	AL 2.0
chemistry-opencmis-commons-impl-0.7.0.jar	Apache Chemistry OpenCMIS	0.7.0	AL 2.0
chemistry-opencmis-server-bindings-0.7.0-classes.jar	Apache Chemistry OpenCMIS	0.7.0	AL 2.0
chemistry-opencmis-server-support-0.7.0.jar	Apache Chemistry OpenCMIS	0.7.0	AL 2.0
com.noelios.restlet-1.0.7.jar	Restlet	1.0.7	CDDL 1.0

com.noelios.restlet.ext.servlet-1.0.7.jar	Restlet	1.0.7	CDDL 1.0
commons-beanutils-1.6.jar	Apache Commons BeanUtils	1.6	AL 2.0
commons-betwixt-0.8.jar	Apache Commons Betwixt	0.8	AL 2.0
commons-codec-1.4.jar	Apache Commons Codec	1.4	AL 2.0
commons-collections-3.2.jar	Apache Commons Collections	3.2	AL 2.0
commons-dbcp-1.3-RC1.jar	Apache Commons DBCP	1.3-RC1	AL 2.0
commons-digester-1.8.jar	Apache Commons Digester	1.8	AL 2.0
commons-el-1.0.jar	Apache Commons EL	1.0	AL 2.0
commons-fileupload-1.2.2.jar	Apache Commons FileUpload	1.2.2	AL 2.0
commons-httpclient-3.1.jar	Apache Commons HttpClient	3.1	AL 2.0
commons-io-1.4.jar	Apache Commons IO	1.4	AL 2.0
commons-jexl-1.1.jar	Apache Commons JEXL	1.1	AL 2.0
commons-pool-1.5.4.jar	Apache Commons Pool	1.5.4	AL 2.0
connector-api-1.5.jar	J2EE Connector Architecture (JSR 112)	1.5	CDDL 1.0
dom4j-1.6.1.jar	dom4j	1.6.1	BSD 3
EditableImage-0.9.5.jar	Mistral	0.9.5	AL 2.0
ehcache-core-2.5.2.jar	Ehcache	2.5.2	AL 2.0
ejb-api-3.0.jar	Enterprise JavaBeans (JSR 220)	3.0	CDDL 1.0
ezmorph-1.0.4.jar	EZMorph	1.0.4	AL 2.0
flute-1.3-gg2.jar	Flute	1.3-gg2	W3C
fontbox-1.6.0.jar	Apache PDFBox	1.6.0	AL 2.0
geronimo-connector-2.2.1-NX1.jar	Apache Geronimo	2.2.1-NX1	AL 2.0
geronimo-transaction-2.2.1.jar	Apache Geronimo	2.2.1	AL 2.0
gin-1.5.0.jar	GIN	1.5.0	AL 2.0
gmbal-api-only-3.1.0-b001.jar	Gmbal	3.1.0-b001	CDDL 1.0
google-collections-1.0-rc2.jar	Google Collections	1.0-rc2	AL 2.0
groovy-all-1.5.7.jar	Groovy	1.5.7	AL 2.0
gson-1.4.jar	Google Gson	1.4	AL 2.0
guice-3.0.jar	Google Guice	3.0	AL 2.0
guice-assistedinject-3.0.jar	Google Guice	3.0	AL 2.0
guice-internal-2.0.jar	Google Guice	2.0	AL 2.0
guice-jmx-3.0.jar	Google Guice	3.0	AL 2.0
guice-servlet-3.0.jar	Google Guice	3.0	AL 2.0
gwt-dispatch-1.0.0.jar	GWT Dispatch	1.0.0	BSD 3
gwt-habyt-upload-0.1.jar	GWT HABYT	0.1	AL 2.0
gwt-servlet-2.4.0.jar	GWT	2.4.0	AL 2.0
ha-api-3.1.8.jar	Glassfish	3.1.8	CDDL 1.0

hibernate-annotations-3.4.0.GA.jar	Hibernate	3.4.0.GA	LGPL 2.1
hibernate-commons-annotations-3.1.0.GA.jar	Hibernate	3.1.0.GA	LGPL 2.1
hibernate-core-3.3.2.GA.jar	Hibernate	3.3.2.GA	LGPL 2.1
hibernate-entitymanager-3.4.0.GA.jar	Hibernate	3.4.0.GA	LGPL 2.1
hibernate-validator-3.1.0.GA.jar	Hibernate	3.1.0.GA	LGPL 2.1
howl-1.0.1-1.jar	HOWL	1.0.1-1	BSD 2
hsqldb-1.8.0.1.jar	HSQLDB	1.8.0.1	BSD 3
htmlparser-1.0.7.jar	Validator.nu HTML Parser	1.0.7	MIT + BSD 3
httpclient-4.1.jar	Apache HttpComponents	4.1	AL 2.0
httpcore-4.1.jar	Apache HttpComponents	4.1	AL 2.0
icu4j-4.0.1.jar	ICU	4.0.1	MIT
iri-0.8.jar	Apache Jena	0.8	BSD 3
itext-2.1.7.jar	iText	2.1.7	LGPL 2.1
itext-rtf-2.1.7.jar	iText	2.1.7	LGPL 2.1
jackson-core-asl-1.8.1.jar	Jackson	1.8.1	AL 2.0
jackson-mapper-asl-1.8.1.jar	Jackson	1.8.1	AL 2.0
java-cup-0.11a.jar	CUP	10k	MIT*
javacc-4.0.jar	JavaCC	4.0	BSD 2
javasimon-core-2.5.0.jar	Simon	2.5.0	LGPL 2.1
javasimon-jdbc3-2.5.0.jar	Simon	2.5.0	LGPL 2.1
javasimon-jmx-2.5.0.jar	Simon	2.5.0	LGPL 2.1
javassist-3.9.0.GA.jar	Javassist	3.9.0.GA	AL 2.0
javax.inject-1.jar	Dependency Injection for Java (JSR 330)	1	AL 2.0
jaxb-impl-2.2.4-1.jar	JAXB RI	2.2.4-1	CDDL 1.1
jaxws-rt-2.2.5.jar	JAX-WS RI	2.2.5	CDDL 1.1
jboss-el-1.0_02.CR2.jar	Seam	1.0_02.CR2	AL 2.0
jboss-seam-2.1.0.SP1.jar	Seam	2.1.0.SP1	LGPL 2.1
jboss-seam-excel-2.1.0.SP1.jar	Seam	2.1.0.SP1	LGPL 2.1
jboss-seam-jul-2.1.0.SP1.jar	Seam	2.1.0.SP1	LGPL 2.1
jboss-seam-pdf-2.1.0.SP1.jar	Seam	2.1.0.SP1	LGPL 2.1
jboss-seam-remoting-2.1.0.SP1.jar	Seam	2.1.0.SP1	LGPL 2.1
jboss-seam-rss-2.1.0.SP1.jar	Seam	2.1.0.SP1	LGPL 2.1
jboss-seam-ui-2.1.0.SP1.jar	Seam	2.1.0.SP1	LGPL 2.1
jcip-annotations-1.0.jar	Java Concurrency in Practice	1.0	CC BY 2.5
jdbm-1.0.jar	JDBM	1.0	BSD 3
jdom-1.0.jar	JDOM	1.0	BSD 3

jempbox-1.6.0.jar	Apache PDFBox	1.6.0	AL 2.0
jena-2.6.4-NX.jar	Apache Jena	2.6.4-NX	BSD 3
jericho-html-3.2.jar	Jericho HTML Parser	3.2	LGPL 2.1
jersey-core-1.11-minimal.jar	Jersey	1.11	CDDL 1.1
jersey-server-1.11.jar	Jersey	1.11	CDDL 1.1
jersey-servlet-1.11.jar	Jersey	1.11	CDDL 1.1
jetty-6.1.26.jar	Jetty	6.1.26	AL 2.0
jetty-util-6.1.26.jar	Jetty	6.1.26	AL 2.0
jline-0.9.94.jar	JLine	0.9.94	BSD 3
jmd-0.8.1-tomasol-3e60e36137.jar	JMD	0.8.1-tomasol-3e60e36137	MIT
jmimemagic-0.1.2.jar	jMimeMagic	0.1.2	AL 2.0
joda-time-1.6.jar	Joda Time	1.6	AL 2.0
jodconverter-core-3.0-NX6.jar	JODConverter	3.0-NX6	LGPL 3
jsecurity-0.9.0.jar	JSecurity	0.9.0	AL 2.0
jsf-api-1.2_12.jar	JavaServer Faces	1.2_12	CDDL 1.0
jsf-facelets-1.1.15.B1.jar	Facelets	1.1.15.B1	CDDL 1.0
jsf-impl-1.2_12.jar	JavaServer Faces	1.2_12	CDDL 1.0
json-20070829.jar	JSON	20070829	MIT
json-lib-2.4-jdk15.jar	JSON-lib	2.4	AL 2.0
json-simple-1.1.jar	JSON.simple	1.1	AL 2.0
jsr181-1.0.jar	Web Services Metadata for Java (JSR 181)	1.0	CDDL 1.0
jsr250-api-1.0.jar	Common Annotations for Java (JSR 250)	1.0	CDDL 1.0
jsr311-api-1.1.1.jar	JAX-RS (JSR 311)	1.1.1	CDDL 1.0
jstl-1.1.2.jar	JSP Standard Tag Library	1.1.2	CDDL 1.0
jta-1.1.jar	Java Transaction API	1.1	CDDL 1.0
juel-impl-2.1.2.jar	Java Unified Expression Language	2.1.2	AL 2.0
juh-3.1.0.jar	Java Uno Helper	3.1.0	LGPL 3
jurt-3.1.0.jar	Java Uno Runtime	3.1.0	LGPL 3
jxl-2.6.8-seam.jar	Java Excel API	2.6.8	LGPL 2.1
logkit-1.0.1.jar	Apache Avalon	1.0.1	AL 1.1
management-api-3.0.0-b012.jar	Glassfish	3.0.0-b012	CDDL 1.0
metadata-extractor-2.3.1.jar	Metadata Extractor	2.3.1	AL 2.0
milyn-commons-1.3.1.jar	Smooks	1.3.1	LGPL 2.1
milyn-magger-1.3.1-NXP-7750.jar	Smooks	1.3.1-NXP-7750	LGPL 2.1
mimepull-1.6.jar	MIME Pull	1.6	CDDL 1.1
mina-core-1.1.2.jar	Apache MINA	1.1.2	AL 2.0

mvel2-2.0.19.jar	MVEL	2.0.19	AL 2.0
nekohtml-1.9.9.jar	NekoHTML	1.9.9	AL 2.0
oauth-20090531.jar	OAuth	20090531	AL 2.0
oauth-consumer-20090531.jar	OAuth	20090531	AL 2.0
oauth-httpclient3-20090531.jar	OAuth	20090531	AL 2.0
oauth-provider-20090531.jar	OAuth	20090531	AL 2.0
ooxml-schemas-1.0.jar	Apache POI	1.0	AL 2.0
opencsv-2.1.jar	opencsv	2.1	AL 2.0
Operations-0.9.5.jar	Mistral	0.9.5	AL 2.0
org.eclipse.equinox.common-3.6.0.v20100503.jar	Eclipse Equinox	3.6.0.v20100503	EPL 1.0
org.eclipse.equinox.p2.cudf-1.15-NX.jar	CUDF Resolver	1.15-NX	EPL 1.0
org.osgi.compendium-4.2.0.jar	OSGi	4.2.0	AL 2.0
org.restlet-1.0.7.jar	Restlet	1.0.7	CDDL 1.0
org.restlet.ext.fileupload-1.0.7.jar	Restlet	1.0.7	CDDL 1.0
org.restlet.gwt-1.1.10.jar	Restlet	1.1.10	CDDL 1.0
org.sat4j.core-2.3.1.jar	Sat4j	2.3.1	LGPL 2.1
org.sat4j.pb-2.3.1.jar	Sat4j	2.3.1	LGPL 2.1
oro-2.0.8.jar	Apache Jakarta ORO	2.0.8	AL 2.0
osgi-core-4.1.jar	OSGi	4.1	AL 2.0
pdfbox-1.6.0.jar	Apache PDFBox	1.6.0	AL 2.0
persistence-api-1.0.jar	Java Persistence API	1.0	CDDL 1.0
plexus-utils-1.5.6.jar	Plexus Common Utilities	1.5.6	AL 2.0
poi-3.5-beta6.jar	Apache POI	3.5-beta6	AL 2.0
poi-ooxml-3.5-beta6.jar	Apache POI	3.5-beta6	AL 2.0
poi-scratchpad-3.5-beta6.jar	Apache POI	3.5-beta6	AL 2.0
policy-2.2.2.jar	Glassfish	2.2.2	CDDL 1.0
quartz-all-1.6.3.jar	Quartz Scheduler	1.6.3	AL 2.0
relaxngDatatype-20020414.jar	RELAX NG	20020414	BSD 3
resolver-20050927.jar	JAX-WS RI	20050927	AL 2.0
richfaces-api-3.3.1.GA.jar	RichFaces	3.3.1.GA	LGPL 2.1
richfaces-impl-3.3.1.GA-NX6.jar	RichFaces	3.3.1.GA-NX6	LGPL 2.1
richfaces-ui-3.3.1.GA-NX2.jar	RichFaces	3.3.1.GA-NX2	LGPL 2.1
ridl-3.1.0.jar	Java Runtime Interface Definition Library	3.1.0	LGPL 3
rome-1.0.jar	Rome	1.0	AL 2.0
saaj-api-1.3.3.jar	SAAJ	1.3.3	CDDL 1.0
saaj-impl-1.3.10.jar	SAAJ	1.3.10	CDDL 1.0
sac-1.3.jar	Simple API for CSS	1.3	W3C

sanselan-0.97-incubator.jar	Apache Commons Sanselan	0.97	AL 2.0
shared-asn1-0.9.7.jar	ApacheDS	0.9.7	AL 2.0
shared-ldap-0.9.7.jar	ApacheDS	0.9.7	AL 2.0
shared-ldap-constants-0.9.7.jar	ApacheDS	0.9.7	AL 2.0
shindig-common-1.1-BETA5-incubating.jar	Apache Shindig	1.1-BETA5-incubating	AL 2.0
shindig-features-1.1-BETA5-incubating.jar	Apache Shindig	1.1-BETA5-incubating	AL 2.0
shindig-gadgets-1.1-BETA5-incubating.jar	Apache Shindig	1.1-BETA5-incubating	AL 2.0
shindig-social-api-1.1-BETA5-incubating.jar	Apache Shindig	1.1-BETA5-incubating	AL 2.0
slf4j-api-1.6.0.jar	SLF4J	1.6.0	MIT
slf4j-log4j12-1.6.0.jar	SLF4J	1.6.0	MIT
snakeyaml-1.7.jar	SnakeYAML	1.7	AL 2.0
stax-api-1.0.jar	StAX	1.0	AL 2.0
stax-ex-1.6.jar	StAX	1.6	AL 2.0
stax2-api-3.1.1.jar	StAX	3.1.1	AL 2.0
streambuffer-1.2.jar	Glassfish	1.2	CDDL 1.0
stringtemplate-3.2.jar	StringTemplate	3.2	BSD 3
unoil-3.1.0.jar	Uno Interface Library	3.1.0	LGPL 3
webdav-jaxrs-1.1.1.jar	WebDAV Support for JAX-RS	1.1.1	LGPL 2.1
wem-2.0.2.jar	WikiModel	2.0.2	EPL 1.0
woodstox-core-asl-4.1.1.jar	Woodstox	4.1.1	AL 2.0
xbean-naming-3.9.jar	Apache Geronimo	3.9	AL 2.0
xercesImpl-2.9.1.jar	Apache Xerces2	2.9.1	AL 2.0
xmlbeans-2.3.0.jar	XMLBeans	2.3.0	AL 2.0
xpp3_min-1.1.4c.jar	XPP3	1.1.4c	BSD 3
xsom-20081112.jar	XSOM	20081112	CDDL 1.0
xstream-1.3.1.jar	XStream	1.3.1	BSD 3
yarfraw-0.92.jar	YAFRAW	0.92	AL 2.0

Importing Data in Nuxeo

Different Tools for Different Use Cases

The Nuxeo Platform provides several tools for managing imports. Choosing the right tool will depend on you exact use cases:

- What amount of data do you need to import?
  Hundreds, thousands, millions?
- Do you need to do the import while the application is running?
  Initial / on time import vs everyday import.
- How complex is your import?
  How many business rules are integrated in your import?
- What is the source you want to import from?
  Database, XML, files, ...
- What are your skills?
  SQL, ETL/ESB, Java dev, ...

This page will walk you though the different import options and tries to gives you the pros and cons of each approach.

In this section

Possible Approaches

User Imports

By default, the Nuxeo Platform allows users to import several documents at a time via:

- Drag & Drop,
- WebDAV,
- the DAM web importer,
- Nuxeo Drive.

Import criteria details

Criteria	Value	Comment
Average import speed	Low	A few documents/s.
Custom logic handling	Built-in	All custom logic will be called.
Ability to handle huge volume	No	No transaction batch management.
Production interruption	No	
Blob upload	In transaction	The blob upload is part of the import transaction.
Post import tasks	None	

The key point is that all these user import systems are designed to be easy to use, but are not designed for high performance and huge volume.

For more information

- Drag and Drop user documentation
- WebDAV and WSS user documentation
- WebDAV developer documentation
- DAM import user documentation
- Nuxeo Drive user documentation
- Nuxeo Drive developer documentation

HTTP API

Nuxeo HTTP Automation API can be used to run imports inside the Nuxeo Platform.

You can use Automation from custom code, custom scripting or from tools like:

- ETL : see the Talend Connector
- ESB : see the Mule Connector

Using the API allows you to easily define import custom logic on the client side, but:

- blobs upload will be part of the process,
- doing transaction batch is not easy since it requires to create custom chains.

Import criteria details

Criteria	Value	Comment

Average import speed	Low / Medium	Several document/s (between 5 and 20 docs/s).
Custom logic handling	Built-in	All custom logic will be called.
Ability to handle huge volume	No	No easy transaction batch management.
Production interruption	No	
Blob upload	In process	The blob upload is part of the import process.
Post import tasks	None	

For more information

- REST API
- Automation

Platform Importer

The Platform importer is a framework that can be used to build custom importers that use the Java API.

Unlike the previous methods:

- The Java API is directly used: no network and marshaling overhead.
- Blobs are read from a local filesystem: no network cost.

The importer does handle several aspects that are important for managing performances:

- transaction batch,
- de-activating some listeners,
- process event handles in bulk-mode.

Import criteria details

Criteria	Value	Comment
Average import speed	High	Several hundreds of documents/s (between 50 and 500 docs/s).
Custom logic handling	Built-in	Most custom logic will be called: depending on which listeners are removed.
Ability to handle huge volume	Yes	Native handling of transaction batch + bulk event handler mode.
Production interruption	Yes	The bulk mode is not adapted for a normal usage: at least a dedicated Nuxeo node should be allocated. High speed import is likely to saturate the database: this will slow down all interactive usages.
Blob upload	Separated	Blobs are directly read on the server side FileSystem.
Post import tasks	May need to restart full text indexing. May need to restart process for listeners that were by-passed .	In a lot of cases, the full text indexing is deactivated during processing, as well as other slow processes like video conversation, thumbnails generation, etc. After import, these processes need to be restarted.

For more information

- Nuxeo Bulk Document Importer developer documentation

SQL Import

Thanks to the VCS Repository clear and clean SQL structure, you can directly use SQL injection.

Of course, this is by far the fastest technique, but since you will bypass all the Java business layer, you will need to do some checks and post

processing. In addition, if you want the SQL import to be really fast, you may want to deactivate some of the integrity constraints and triggers.

Import criteria details

Criteria	Value	Comment
Average import speed	Very high	Several thousands of Documents/s (between 500 and 5000 docs/s).
Custom logic handling	Bypass	All Java Layer is by-passed.
Ability to handle huge volume	Yes	Native handling of transaction batch + bulk event handler mode.
Production interruption	Yes	Usually, the database server configuration is changed to make the bulk insert more efficient.
Blob upload	Not handled	Blobs needs to be managed by a separated process.
Post import tasks	May need to restart full text indexing. May need to restart some triggers.	■ Rebuild full text. ■ Rebuild ancestors cache. ■ Rebuild read-ACLs

For more information

- VCS Architecture
- Internal VCS Model
- VCS Tables
- Examples of SQL Generated by VCS
- Java Data Structures and Caching
- Performance Recommendations

Customization and Development

After having gone through an architecture tour in the previous section, this section focuses on providing an extensive documentation on the various modules of the platform. You will learn all the required theoretical knowledge and find some useful snippets of code.

In this section, authors consider you know how to practically apply what you read, by using Nuxeo Studio (for producing the required XML) or writing Java plugins using Nuxeo IDE. The role of the Dev Cook Book section and the tutorial section of Nuxeo Studio documentation is to provide steps by steps guides.

Section's content:

- Extension points configuration — Inside the Nuxeo Platform, pretty much everything is about extension points. Extension points are used to let you contribute XML files to the Nuxeo components. This section explains the basic steps to contribute a new XML file to an extension point.
- Repository features — This chapter encompasses features related to the repository: the concepts used for documents types, how to retrieve documents, how to version them, how to add security policies to access documents, how tags work.
- Authentication and User Management
- REST API — Nuxeo REST API is available on a Nuxeo server at the following URL: http://localhost:8080/nuxeo/api/v1/* . This section describes all the mechanisms that are offered by our REST API as well as all the tools useful for using it (clients, format of queries, etc.).
- Other Repository APIs — This section is about the SOAP bridge, CMIS and WebDAV APIs, as well as URLs to use when downloading a binary content.
- Directories and Vocabularies
- Using the Java API Serverside — This page explains how to use the Nuxeo Java API.
- Automation — In this section you'll find information on how to use the Automation service, how to contribute a new chain, a new operation, etc.
- Customizing the web app — This chapter presents the different ways to customize what is displayed on the application.
- Workflow — This page and its subpages explain all the concepts and provide a documentation on how the workflow engine works.
- WebEngine (JAX-RS)
- Other services
- Other UI Frameworks — Nuxeo uses several UI frameworks beside the default JSF technology.
- Additional Modules — This chapter presents how to apprehend and customize the additional packages available on the Nuxeo Platform, typically from the Nuxeo Marketplace.
- Packaging
- Advanced topics

Extension points configuration

Inside the Nuxeo Platform, pretty much everything is about extension points. Extension points are used to let you contribute XML files to the Nuxeo components. This section explains the basic steps to contribute a new XML file to an extension point.

The extension point system to can be used to:

- Define a new document type,
- Hide a button from the default UI that you want to remove,
- Change the condition that make a particular view available,
- Add a new navigation axis,
- Change the way the documents listings are displayed,
- ...

So before going further, you may want to take a look at the Component Model section.

Once you have understood the notion of extension point and contribution, you can go ahead and start configuring the platform.

For that, you first need to know what you want to configure: find the extension point you want to contribute to. The next sections will give you an overview of the main concepts of the most used extension points. If you need more, you can directly use the Nuxeo Platform Explorer to browse all the available extension points.

Once you have found your target extension point, you simply have to create an XML file that holds the configuration and deploy it inside your Nuxeo server.

The exact XML content will depends on each extension point, but they all start the same:

```
<?xml version="1.0"?>
 <component name="unique.name.for.your.xml.contribution">

  <extension target="target.component.identifier"
    point="extensionPointName">

    <!-- XML Content Depending on the target Extension Point goes HERE -->

  </extension>

</component>
```

In order to have your contribution deployed you need to:

- Have your filename end with `-config.xml`,
- Have your file placed in the config directory (`nuxeo.ear/config` for JBoss distribution or `nxserver/config` for Tomcat distribution)

By default, XML configuration files are only read on startup, so you need to restart your server in order to apply the new configuration.

Easy extension configuration with Nuxeo Studio
Nuxeo Studio is a great way to configure extensions, have a look at the documentation!

Repository features

This chapter encompasses features related to the repository: the concepts used for documents types, how to retrieve documents, how to version them, how to add security policies to access documents, how tags work.

- Document types — This chapter presents the concepts of schemas, facets and document types, which are used to define documents.
- Versioning — This section describes the versioning model of Nuxeo 5.4 and later releases.
- Querying and Searching — In Nuxeo the main way to do searches is through NXQL, the Nuxeo Query Language, a SQL-like query language.
- Tagging — The tag service uses two important concepts: a tag object, and a tagging action. Both are represented as Nuxeo documents.
- Security Policy Service — The Security Policy Service provides an extension point to plug custom security policies that do not rely on the standard ACLs for security. For instance, it can be used to define permissions according to the document metadata, or information about the logged in user.

- Events and Listeners — Events and event listeners have been introduced at the Nuxeo core level to allow pluggable behaviors when managing documents (or any kinds of objects of the site).
- Bulk Edit — The bulk edit feature allows to edit several documents at the same time. This is implemented using the BulkEditService component.

Document types

This chapter presents the concepts of schemas, facets and document types, which are used to define documents.
In Nuxeo EP, a fundamental entity is the *document*. A file, a note, a vacation request, an expense report, but also a folder, a forum, can all be thought of as documents. Objects that contain documents, like a folder or a workspace, are also themselves documents.

Any given document has a *document type*. The document type is specified at creation time, and does not change during the lifetime of the document. When referring to the document type, a short string is often used, for instance "Note" or "Folder".

A document type is defined by several *schemas*. A schema represents the names and structure (types) of a set of fields in a document. For instance, a commonly-used schema is the Dublin Core schema, which specifies a standard set of fields used for document metadata like the title, description, modification date, etc.

In addition to the schemas that the document type always has, a given document instance can receive *facets*. A facet has a name, like "Downloadable" or "Commentable", and can be associated with zero or more schemas. When a document instance receives a facet, the fields of its schemas are automatically added to the document.

Per-document facets and facets associated with schemas are a new feature since Nuxeo EP 5.4.1 (see NXP-6084).

To create a new document type, we start by creating one or more schemas that the document type will use. The schema is defined in a `.xsd` file and is registered by a contribution to the **schema** extension point. The document type is then registered through a contribution to the **doctype** extension point which specifies which schemas it uses. Facets are also registered through the **doctype** extension point.

In this section

In addition to the structural definition for a document type, there's another registration at the UI level, through a different extension point, to define how a given document type will be rendered (its icon, layouts, default view, etc.).

The sections below describe how schemas, facets and document types are defined at a low level in Nuxeo EP using XML configuration files. Unless you're an advanced user, it will be much simpler to use Nuxeo Studio to define them.

Schemas

A schema describes the names and types of some fields. The name is a simple string, like "title", and the type describes what kind of information it stores, like a string, an integer or a date.

A schema is defined in a `.xsd` file and obeys the standard XML Schema syntax.

> *For example, we can create a schema in the `schemas/sample.xsd` file:*

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://project.nuxeo.org/sample/schemas/sample/">
  <xs:element name="sample1" type="xs:string"/>
  <xs:element name="sample2" type="xs:string"/>
</xs:schema>
```

This schema defines two things:

- *an XML namespace that will be associated with the schema (but isn't used by Nuxeo EP),*
- *two elements and their type.*

The two elements are sample1 and sample2. They are both of type "string", which is a standard type defined by the XML Schema specification.

A schema file has to be referenced by Nuxeo configuration to be found and used. The schema must be referenced in the **schema** extension point of the `org.nuxeo.ecm.core.schema.TypeService` component. A reference to a schema defines:

- the schema name,
- the schema location (file),
- an optional (but recommended) schema prefix.

For example, in the configuration file `OSGI-INF/types-contrib.xml` (the name is just a convention) you can define:

```
<?xml version="1.0"?>
<component name="org.nuxeo.project.sample.types">
  <extension target="org.nuxeo.ecm.core.schema.TypeService" point="schema">
    <schema name="sample" src="schemas/sample.xsd" prefix="smp" />
  </extension>
</component>
```

*We name our schema "sample", and the `.xsd` file is referenced through its path, `schemas/sample.xsd`. The schema is registered through the **schema** extension point of the Nuxeo component `org.nuxeo.ecm.core.schema.TypeService`. Our own extension component is given a name, `org.nuxeo.project.sample.types`, which is not very important as we only contribute to existing extension points and don't define new ones — but the name must be new and unique.*

Finally, like for all components defining configuration, the component has to registered with the system by referencing it from the `META-INF/MANIFEST.MF` file of the bundle.

In our example, we tell the system that the `OSGI-INF/types-contrib.xml` file has to be read, by mentioning it in the `Nuxeo-Component` part of the `META-INF/MANIFEST.MF`:

```
Manifest-Version: 1.0
Bundle-SymbolicName: org.nuxeo.project.sample;singleton:=true
Nuxeo-Component: OSGI-INF/types-contrib.xml
...
```

You may need to override an existing schema defined by Nuxeo. As usual, this possible and you have to contribute a schema descriptor with same name. But you must also add an override parameter with value "true".

For instance, you can add your own parameters into the user.xsd schema to add the extra information stored into your ldap and fetch them and store them into the principal instance (that represents every user).
The contribution will be something like:

```
<component name="fr.mycompanyname.myproject.schema.contribution">
  <!-- to be sure to deployed after the Nuxeo default contributions>
  <require>org.nuxeo.ecm.directory.types</require>
  <extension target="org.nuxeo.ecm.core.schema.TypeService" point="schema">
    <schema name="group" src="directoryschema/group.xsd" override="true"/>
  </extension>
</component>
```

*Focus your attention on the **override="true"** that is often missing*

You will need to improve the UI to also display your extra-informations...

Facets

A facet describes an aspect of a document that can apply to several document types or document instances. Facets can have zero, one or more schemas associated to them. Configuration is done in the **doctype** extension point of the same `org.nuxeo.ecm.core.schema.TypeService` component as for schemas.

For example, in the same `OSGI-INF/types-contrib.xml` as above, we add the following:

```
<?xml version="1.0"?>
<component name="org.nuxeo.project.sample.types">
  ...
  <extension target="org.nuxeo.ecm.core.schema.TypeService"
point="doctype">
    <facet name="Rated">
      <schema name="rating"/>
    </facet>
    ...
  </extension>
</component>
```

Facets can be used in two ways:

- on document types, by adding the facet to the `<doctype>` element described below,
- on document instances, by application code.

When a document's type or a document's instance has a facet, the document behaves normally with respect to the added schemas. Facets with no schemas are useful to mark certain types or certain document instances specially, for instance to add additional behavior when they are used.

Standard Nuxeo EP facets are:

- **Folderish**: special facet allowing the creation of children in this document,
- **Orderable**: special facet allowing the children of a folderish type to be ordered,
- **Versionable**: special facet marking the document type as versionable,
- **HiddenInNavigation**: special facet for document types which should not appear in listings.

See the Available Facets page for more details on available facets.

Structural Document Types

By itself, the schema is not very useful, it must be associated with a document type. This is done in the same **doctype** extension point as above. In this extension point, we define:

- the document type to create,
- which standard document type it extends (usually "Document" or "Folder"),
- what schemas it contains,
- what facets it has (this implicitly adds all the facet's schemas).

When extending a document type, all its schemas and facets are inherited as well.

For example, in the same `OSGI-INF/types-contrib.xml` as above, we add the following:

```
<?xml version="1.0"?>
<component name="org.nuxeo.project.sample.types">
  ...
  <extension target="org.nuxeo.ecm.core.schema.TypeService"
point="doctype">
    ...
    <doctype name="Sample" extends="Document">
      <schema name="common"/>
      <schema name="dublincore"/>
      <schema name="sample"/>
      <facet name="Rated"/>
    </doctype>
  </extension>
</component>
```

Here we specify that our document type "Sample" will be an extension of the standard system type "Document" and that it will be composed of three schemas, two standard ones and our specific one, and has one facet.

The standard schemas "common" and "dublincore" already contain standard metadata fields, like a title, a description, the modification date, the document contributors, etc. Adding it to a document type ensures that a minimal level of functionality will be present, and is recommended for all types.

UI Document Types

After the structural document type, a UI registration for our document type must be done for the type to be visible in the Nuxeo DM interface (or in other applications based on Nuxeo EP). This is done through a contribution to the **types** extension point of the `org.nuxeo.ecm.platform.types.TypeService` component (which is a different component than for the structural types, despite also ending in `TypeService`).

For example, in `OSGI-INF/ui-types-contrib.xml` we will define:

```
<?xml version="1.0"?>
<component name="org.nuxeo.project.sample.ecm.types">
  <extension target="org.nuxeo.ecm.platform.types.TypeService"
point="types">
    <type id="Sample">
      <label>...</label>
      <icon>...</icon>
      <bigIcon>...</bigIcon>
      <description>...</description>
      <category>...</category>
      <layouts>...</layouts>
      ...
    </type>
  </extension>
</component>
```

The extension must be added to `META-INF/MANIFEST.MF` so that it will be taken into account by the deployment mechanism:

```
Nuxeo-Component: OSGI-INF/types-contrib.xml,
 OSGI-INF/ui-types-contrib.xml
```

The type element will contain all the information for this type, described below.

General Information

The **label**, **description**, **icon**, **bigIcon** and **category** are used by the user interface, for instance in the creation page when a list of possible types is displayed.

- **label**: a short name for the type.
- **description**: a longer description of the type.
- **icon**: a 16x16 icon path for the type, used in listings for instance. The path points to a resource defined in the Nuxeo WAR.
- **bigIcon**: a 100x100 icon path for the type, used in the creation screen for instance.
- **category**: a category for the type, used to separate types in different sections in the creation screen for instance.

Standard categories used in the Nuxeo DM interface are:

- SimpleDocument: a simple document
- Collaborative: a document or folder-like objects used for collaboration
- SuperDocument: a structural document usually created by the system

Other categories can freely be defined.

> *Example:*
>
> ```
> <type id="Sample">
>   <label>Sample document</label>
>   <description>Sample document to do such and such</description>
>   <icon>/icons/file.gif</icon>
>   <bigIcon>/icons/file_100.png</bigIcon>
>   <category>SimpleDocument</category>
>   ...
> </type>
> ```

Facelet Views

The **default-view** tag specifies the name of the facelet to use to display this document. This corresponds to a file that lives in the webapp, by default `view_documents.xhtml` which is a standard view defined in the base Nuxeo EP bundle. This standard view takes care of displaying available tabs and the document body according to the currently selected type.

Changing it is not advised unless extremely nonstandard rendering is needed.

The **create-view** and **edit-view** tags can point to a specific creation or edit facelets.

Proper defaults are used when these are not specified, so no need to add them to your type.

> *Example:*
>
> ```
> <type id="Sample">
>   ...
>   <default-view>view_documents</default-view>
>   <create-view>create_document</default-view>
>   <edit-view>edit_document</default-view>
>   ...
> </type>
> ```

Layout

A layout is a series of widgets, which makes the association between the field of a schema with a JSF component. The layout is used by the standard Nuxeo modification and summary views, to automatically display the document metadata according to the layout rules.

Layouts Configuration

The **layouts** section (with a final **s**) defines the layouts for the document type for a given mode.

Defaults mode are:

- **create** for creation,
- **edit** for edition,
- **view** for view,
- **any** for layouts that will be merged in all the other modes.

The **layout** names refer to layouts defined on another extension point. Please see the layouts section for more information.

> *Example:*
>
> ```
> <type id="Sample">
>   ...
>   <layouts mode="any">
>     <layout>heading</layout>
>     <layout>note</layout>
>   </layouts>
>   ...
> </type>
> ```

Containment Rules

The **subtypes** section defines a list of **type** elements for the document types that can be created as children objects of other document types. When defining a type, you can specify:

- what child document types can be create in it,
- in what parent document types it can be created.

This can also be defined for a pre-existing type, to add new allowed subtypes. Please make sure you require the components defining the pre-existing type to ensure a good merge of contributions.

> *For example, we can specify that the Sample type can be created in a Folder and a Workspace. Note that we define two new `<type>` sections here, we don't add this information in the `<type id="Sample">` section.*
>
> ```
> <type id="Folder">
>   <subtypes>
>     <type>Sample</type>
>   </subtypes>
> </type>
> <type id="Workspace">
>   <subtypes>
>     <type>Sample</type>
>   </subtypes>
> </type>
> ```

It is also possible to define that some types will not be allowed as children in some cases (creation, copy/paste). To do that, a **hidden** attribute for the **type** element can be used.

The hidden cases are stored in a list, so if a check is needed for a hidden case, then the hidden cases list ought to be verified to check it contains that particular case.

> *Example:*

```
<type id="Workspace">
  <subtypes>
    <type>Workspace</type>
    <type hidden="create, paste">Folder</type>
    <type>File</type>
    <type>Note</type>
  </subtypes>
</type>
```

Summary

The final `OSGI-INF/ui-types-contrib.xml` looks like:

```
<?xml version="1.0"?>
<component name="org.nuxeo.project.sample.ecm.types">

 <!-- Add require to component declaring Workspace and Folder types -->
 <require>org.nuxeo.ecm.platform.types</require>

  <extension target="org.nuxeo.ecm.platform.types.TypeService" point="types">

    <type id="Sample">
      <label>Sample document</label>
      <description>Sample document to do such and such</description>
      <icon>/icons/file.gif</icon>
      <bigIcon>/icons/file_100.png</bigIcon>
      <category>SimpleDocument</category>
      <default-view>view_documents</default-view>
      <layouts mode="any">
        <layout>heading</layout>
        <layout>note</layout>
      </layouts>
    </type>

 <!-- containment rules -->
  <type id="Folder">
      <subtypes>
        <type>Sample</type>
      </subtypes>
    </type>
    <type id="Workspace">
      <subtypes>
        <type>Sample</type>
      </subtypes>
    </type>

 </extension>
</component>
```

Available Facets

Facets are markers on document types that instruct Nuxeo (and any part of the system that cares about them) to behave differently, or that are automatically set on some document instances. The following facets are currently in use by Nuxeo, but others could be defined by code using Nuxeo.

Asset

Marker facet to be set on types which you want to see in the DAM view.

Audio

This facet should be set on any type which you want to store a audio. It comes with the `file`, and `audio` schemas.

BrowseViaSearch

DamSearch

Marker facet to be set on types used as search document types for DAM search content views. Documents with this facet will be listed as DAM saved searches.

Downloadable

FacetedSearch

Marker facet to be set on types used as search document types for content views used in the Faceted search. Documents with this facet will be listed as faceted saved searches.

Folderish

This should be set on types that are to be considered as "folders", and therefore for which a list of children may be displayed in the navigation tree and other user interface areas.

HasRelatedText

This facet can be added dynamically to a document to store arbitrary related text that will be used in the fulltext index for the document. In this way, you can find the document with a fulltext search on more that just what's stored in the document.

This is used for instance to add the Annotations fulltext to the document, or in the Semantic Entities plugin. We'd like to do it for comments as well but it's not done yet.

It can be used by third-parties to add further text.

Of course this requires that the fulltext index includes the `relatedtext` schema. This is the default, but if you redefine the default index it's important to know.

HasStoryboard

Marker facet to be set on types which you want to generate Storyboard, types on which you already added the Video facet.

HasVideoPreview

Marker facet to be set on types which you want to generate a video preview (screenshot), and on which you already added the Video facet.

HiddenInNavigation

This should be set on types which you don't want to see in regular listings of documents and folders.

Immutable

This is automatically set by the system on document instances which correspond to immutable objects (archived versions and proxies).

MultiViewPicture

This is automatically set by the system on document instances which correspond to immutable objects (archived versions and proxies).

NotFulltextIndexable

Since Nuxeo Platform 5.7.1.
The document won't be full-text indexed.

Orderable

This should be set on folderish types for which maintaining the order of children is important. (CoreSession.orderBefore can be used only on

documents contained in orderable folders.)

Picture

This facet should be set on any type which you want to store a picture. It comes with the `file` and `picture` schemas.

Publishable

This should be set on document types for which you want to have publishing.

SystemDocument

The document type corresponds to a system document, not a user-visible document. It is often hidden in navigation as well, but not always, as some system documents (like workflow route models) may need to be visible to administrators or selected users.

Versionable

This should be set on document types for which you want to have versioning.

Video

This facet should be set on any type which you want to store a video. It comes with the `file`, `video` and `picture` schemas.

Task

This facet should be set on any type if you want to use it as the task document created by a workflow. Note that along with the facet, the document type must also have the life cycle "task".

WebView

In this section

Versioning

This section describes the versioning model of Nuxeo 5.4 and later releases.

Concepts

- **Placeful**. A placeful document is one which is stored in a folder, and therefore has a parent in which it is visible as a child.
- **Placeless**. A placeless document isn't stored in a given folder, it's just available in the storage through its id. Having no parent folder it doesn't inherit any security, so it is usually only accessible by unrestricted code.
- **Working Copy**. The document that you edit. It is usually stored in a Workspace's folder but this is just convention. It is also often called the **Live Document**. There is at most one Working Copy per version series. In other systems it is also called the Private Working Copy because only the user that created it can work on it; this is looser in Nuxeo EP.
- **Version**. An immutable, archived version of a document. It is created from a **working copy** by a **check in** operation.

- **Version Number**. The label which is uniquely attached to a version. It formed of two integers separated by a dot, like "2.1". The first integer is the major version number, the second one is the minor version number.
- **Major Version**. A version whose minor version number is 0. It follows that a minor version is a version whose minor version number is not 0.
- **Version Series**. The list of versions that have been successively created from an initial **working copy**. The version series id is a unique identifier that is shared by the working copy and all the versions of the version series.
- **Versionable Document**. The document which can be versioned, in effect the **working copy**. Up to Nuxeo EP 5.4, the versionable document id is used as the version series id.
- **Check In**. The operation by which a new **version** is created from a **working copy**.
- **Check Out**. The operation by which a **working copy** is made available.

In this section

Check In and Check Out

"Check In" and "Check Out" in Nuxeo EP both refer to operations that can be carried out on documents, and to the state a working copy can be in.

Checked In and Checked Out States

A working copy in the Checked Out state can be modified freely by users having access rights to the document. A document ceases to be Checked Out when the Check In operation is invoked. After initial creation a document is in the Checked Out state.

A working copy in the Checked In state is identical to the version that was created when the Check In operation was invoked on the working copy. In the Checked In state, a working copy is (at low level) not modifiable. To be modified it must be switched to the Checked Out state first. This latter operation is automatically done in Nuxeo EP 5.4 when a document is modified.

Check In and Check Out Operations

From a working copy in the Checked Out state, invoking the Check In operation does several things:

- the final version number is determined,
- a new version is created,
- the working copy is placed in the Checked In state.

When invoking the Check In operation, a flag is passed to indicate whether a major version or a minor version should be created. Depending on whether the new version should be major or minor, the version number is incremented differently; for instance, starting from a working copy with the version number "2.1" (displayed as "2.1+"), a minor version would be created as "2.2" and a major version as "3.0".

Given a Checked In working copy, invoking the Check Out operation has little visible effect, it's just a state change for the working copy. A "+" is displayed after the version number to make this apparent, see below.

In other systems than the Nuxeo Platform, the Check In operation that creates a new version removes the Working Copy, whose role has been fulfilled. This is not the case in the Nuxeo Platform, where the Working Copy remains in a special Checked In state. In these other systems, the Check Out operation can also be invoked on a Version to create a new Working Copy (this assumes that there is no pre-existing Working Copy in the system). This kind of operation will be made available in future versions of the platform but is not present at the moment.

Version Number

The initial version number of a freshly created working copy is "0.0".

When displaying the version number for a Checked Out document, the version number is usually displayed with a "+" following it, to show that it's not the final version number but that the working copy is modified and derived from that version. The final version number will be

determined at Check In time. The exception to this display rule is the version "0.0", because displaying "0.0+" would be redundant and misleading as there is actually no previously archived "0.0" version from which it is derived.

The version number is changed by a Check In operation; either the minor version number is incremented, or the major version number is incremented and the minor version number is set to 0.

Plugging In a New `VersioningService` Implementation

For advanced uses, it's possible to plug in a new `VersioningService` implementation to define what happens at creation, save, check in and check out time. See the Javadoc and the versioningService extension point documentation for more about this.

Querying and Searching

In Nuxeo the main way to do searches is through NXQL, the Nuxeo Query Language, a SQL-like query language.

You can read a full description of the NXQL syntax.

The fulltext aspects of the searches are described on a separate page.

In this section:

- NXQL
- Full-Text Queries — Nuxeo documents can be searched using fulltext queries; the standard way to do so is to use the top-right "quick search" box in Nuxeo DM. Search queries are expressed in a Nuxeo-defined syntax, described below.
- Search Results Optimizations — When there are a lot of search results, search becomes slow, which results in very slow response times when users browse the application, which can even be unresponsive. When monitoring the server, memory is overused.

NXQL

NXQL Syntax

The general syntax of a NXQL expression is:

`SELECT` (*|[`DISTINCT`] <*select-clause*>) `FROM` <*from-clause*> [`WHERE` <*where-clause*>] [`ORDER BY` <*order-by-clause*>]

The <*select-clause*> is a comma-separated list of properties. Properties are Nuxeo document property names, for instance `dc:modified`, or special properties, for instance `ecm:uuid` (see below).

The <*from-clause*> is a comma-separated list of document types.

The optional <*where-clause*> is a general <*predicate*>.

The optional <*order-by-clause*> is a comma-separated list of <*identifier*>, each one being optionally followed by `ASC` or `DESC` (`ASC` is the default).

A <*predicate*> can be:

- <*predicate*> <*operator*> <*predicate*>
- <*identifier*> [`NOT`] `IN` (<*literal-list*>)
- <*identifier*> [`NOT`] `BETWEEN` <*literal*> `AND` <*literal*>
- <*identifier*> `IS` [`NOT`] `NULL` (since Nuxeo 5.4.2, see NXP-4339)
- (<*predicate*>)
- `NOT` <*predicate*>
- <*expression*>

In this section

- NXQL Syntax
- List Properties
- Complex Properties
- Special NXQL Properties
- Examples
  - Fulltext Examples

An <*operator*> can be:

- `AND`
- `OR`
- `=`
- <> (or `!=` for Java compatibility and to make it convenient to write XML files)
- `<`
- `<=`

- `>`
- `>=`
- [`NOT`] (`LIKE`|`ILIKE`) (only between an *<identifier>* and a *<string>*)
- `STARTSWITH` (only between an *<identifier>* and a *<string>*)

Be careful with Oracle when comparing a value with an empty string, as in Oracle an empty string is `NULL`. For instance `dc:description <> ''` will never match any document, and `dc:description IS NULL` will also match for an empty description.

An *<expression>* can be:

- *<expression>* *<op>* *<expression>*
- `(` *<expression>* `)`
- *<literal>*
- *<identifier>*

An *<op>* can be:

- `+`
- `-`
- `*`
- `/`

A *<literal>* can be:

- *<string>*: a string delimited by single quotes (`'`) or for Java compatibility double quotes (`"`). To use the string delimiter itself inside the string, it must be escaped by a backslash (`\'` or `\"`) (this is contrary to the standard SQL syntax which would double the delimiter). The backslash itself is also escaped by a backslash (`\\`). The special `\n`, `\r` and `\t` can also be used.
- *<integer>*: an integer with optional minus sign.
- *<float>*: a float.
- `TIMESTAMP` *<timestamp>*: a timestamp in ISO format *yyyy*`-`*MM*`-`*dd hh*`:`*mm*`:`*ss*[`.`sss] (the space separator can be replaced by a `T`).
- `DATE` *<date>*: a date in ISO format *yyyy*`-`*MM*`-`*dd*, converted internally to a timestamp by adding `00:00:00` to it.

A *<literal-list>* is a non empty comma-separated list of *<literal>*.

An *<identifier>* is a property identifier. Before Nuxeo 5.5, this can be only a simple property or a simple list property. Since Nuxeo 5.5, this can also be a complex property element, maybe including wildcards for list indexes (see below).

List Properties

A Nuxeo property representing a list of simple values (like `dc:subjects`) can be queried as if it represented a simple value, and Nuxeo will automatically expand the query to match any of the value in the list. The following example will find the documents where **any** subject is *foo*:

```
SELECT * FROM Document WHERE dc:subjects = 'foo'
```

Note that the above does **not** mean to find the documents where the list of subjects is exactly the list *[foo]*; NXQL (and indeed SQL) does not have enough expressivity for that (and it would be quite slow).

The above example shows the = operator, and the same semantics apply for the operators `IN`, `LIKE` and `ILIKE`.

When using **negative queries**, though, the semantics get a bit more complex. The following example will find the documents where **no** subject is *foo*:

```
SELECT * FROM Document WHERE dc:subjects <> 'foo'
```

Note that the above **does not** mean to find the documents where there is at least one subject that is not *foo*.

The above example shows the <> operator, and the same semantics apply for the other negative operators `NOT IN`, `NOT LIKE` and `NOT ILIKE`.

Since Nuxeo 5.5, the complex property syntax (described in detail further down) can be used to match single list elements. The following two queries will do the same thing:

```
SELECT * FROM Document WHERE dc:subjects = 'foo'
SELECT * FROM Document WHERE dc:subjects/* = 'foo'
```

There is however an important difference in the mechanism with which these two requests are executed internally. The first syntax (which also worked before Nuxeo 5.5) internally uses a SQL `EXISTS` and a subquery. The second one uses a SQL `JOIN` (with a SQL `DISTINCT` if `SELECT *` is used). The end result is usually the same unless you want to use `AbstractSession.queryAndFetch` with no `DISTINCT` to get to the actual matching subjects, then only the second form is usable

In the case where negative queries are used, however, the different execution mechanisms imply that the two syntaxes mean different things:

```
SELECT * FROM Document WHERE dc:subjects <> 'foo'
SELECT * FROM Document WHERE dc:subjects/* <> 'foo'     -- not the same thing as above
```

The first syntax, as already explained, will find the documents where **no** subject is *foo*.

The second syntax will find the documents where there is **at least one** subject which **is not** *foo*.

Complex Properties

Since Nuxeo 5.5 you can refer to complex properties in NXQL, after the `SELECT`, in the `WHERE` clause, and in the `ORDER BY` clause (cf NXP-4464).

A complex property is a property of a schema containing `<xs:simpleType>` lists, or `<xs:complexType>` subelements or sequences of them.

For complex subproperties, like the `length` field of the `content` field of the `file` schema, you can refer to:

- `content/length` for the value of the subproperty.

For simple lists, like `dc:subjects`, you can refer to:

- `dc:subjects/3` for the 4th element of the list (indexes start at 0),
- `dc:subjects/*` for any element of the list,
- `dc:subjects/*1` for any element of the list, correlated with other uses of the same number after `*`.

For complex lists, like the elements of the `files` schema, you can refer to:

- `files/3/file/length` for the length of the 4th file (again, indexes start at 0),
- `files/*/file/length` for any length
- `files/*1/file/length` for any length, correlated with other uses of the same number after `*`.

It's important to note that if you use a `*` then the resulting SQL `JOIN` generated may return several resulting rows, which means that if you use the `AbstractSession.queryAndFetch` API you may get several results for the same document.

The difference between `*` and `*1` gets important when you refer to the same expression twice, for instance if you want the documents with an optional attached of given characteristics, you must correlate the queries.

This returns the documents with an attached text file of length 0:

```
SELECT * FROM Document WHERE files/*1/file/name LIKE '%.txt' AND files/*1/file/length
= 0
```

This returns the documents with an attached text file and an attached file of length 0:

```
SELECT * FROM Document WHERE files/*/file/name LIKE '%.txt' AND files/*/file/length =
0
```

Special NXQL Properties

The following properties are not legal as document property names, but are allowed in NXQL.

ecm:uuid: the document id (`DocumentModel.getId()`).

ecm:parentId: the document parent id.

ecm:path: the document path (`DocumentModel.getPathAsString()`), it cannot be used in the *<select-clause>*, and using it in the *<order-by-clause>* carries a large performance penalty.

ecm:name: the document name (`DocumentModel.getName()`).

ecm:pos: the document position in its parent, this is `NULL` in non-ordered folders. This is mainly used in the *<order-by-clause>*.

ecm:primaryType: the document type (`DocumentModel.getType()`).

ecm:mixinType: a list of the document facets (`DocumentModel.getFacets()`) with some restrictions. 1. the facet *Immutable* is never seen. 2. the facets *Folderish* and *HiddenInNavigation* are never seen on document instances (only if they're on the type). 3. like for other list properties, it can be used only with operators =, <>, `IN` and `NOT IN`.

ecm:currentLifeCycleState: the document lifecycle state (`DocumentModel.getCurrentLifeCycleState()`).

ecm:isCheckedIn: `1` if the document is checked in and `0` if not (the opposite of `DocumentModel.isCheckedOut()`). This can only be compared to `1` or `0`. (Since Nuxeo 5.7.3)

ecm:isProxy: `1` for proxies and `0` for non-proxies (`DocumentModel.isProxy()`). This can only be compared to `1` or `0`.

ecm:isVersion or **ecm:isCheckedInVersion**: `1` for versions and `0` for non-version (`DocumentModel.isVersion()`). This can only be compared to `1` or `0`. (The name **ecm:isVersion** is available since Nuxeo 5.7.3)

ecm:versionLabel: the version label for versions (`DocumentModel.getVersionLabel()` only for a version), `NULL` if it's not a version.

ecm:versionDescription: the version description for versions, `NULL` if it's not a version. (Since Nuxeo 5.7.3)

ecm:versionCreated: the version creation time for versions, `NULL` if it's not a version. (Since Nuxeo 5.7.3)

ecm:versionVersionableId: the id of the versionable document of a version (the versionable document is the one from which the version was created). (Since Nuxeo 5.7.3)

ecm:isLatestVersion: `1` if this is the latest version of a document, `0` if not. This can only be compared to `1` or `0`. (Since Nuxeo 5.7.3)

ecm:isLatestMajorVersion: `1` if this is the latest major version of a document, `0` if not. This can only be compared to `1` or `0`. (Since Nuxeo 5.7.3)

ecm:proxyTargetId: the id of the target of a proxy (usually a version). Implies a search for proxies (`ecm:isProxy = 1`). (Since Nuxeo 5.7.1)

ecm:proxyVersionableId: the id of the versionable document of a proxy (the versionable document is the one from which the version to which the proxy is pointing was created). Implies a search for proxies (`ecm:isProxy = 1`). (Since Nuxeo 5.7.1)

ecm:lockOwner: the lock owner (`DocumentModel.getLockInfo().getOwner()`). (Since Nuxeo 5.4.2)

ecm:lockCreated: the lock creation date (`DocumentModel.getLockInfo().getCreated()`). (Since Nuxeo 5.4.2)

ecm:lock: the old lock. (**Deprecated** since Nuxeo 5.4.2 and NXP-6054, now returns **ecm:lockOwner**, used to return a concatenation of the lock owner and a short-format creation date)

ecm:fulltext: a special field to make fulltext queries, see Full-Text Queries for more.

ecm:tag: a tag of the document. This property, when used multiple times in the same query, always refers to the same tag. If you want to refer to multiple tags in the same query, you can use a wildcard syntax similar to complex properties: every instance of **ecm:tag/*** will always refer to a different tag. If you want to refer several times to the same tag but still have flexibility, use correlated wildcards like for complex properties: **ecm:tag/*1**, **ecm:tag/*2**, etc. See the examples below for more. (Since Nuxeo 5.7.1)

Examples

```
SELECT * FROM Document
SELECT * FROM Folder
SELECT * FROM File
SELECT * FROM Note
SELECT * FROM Note, File WHERE dc:title = 'My Doc'
SELECT * FROM Document WHERE NOT dc:title = 'My Doc'
SELECT * FROM Document WHERE dc:title = 'My Doc' OR dc:title = 'My Other Doc'
SELECT * FROM Document WHERE (dc:title = 'blah' OR ecm:isProxy = 1) AND
dc:contributors = 'bob'
SELECT * FROM Document WHERE file:filename = 'testfile.txt'
SELECT * FROM Document WHERE uid = 'isbn1234'
SELECT * FROM Document WHERE file:filename = 'testfile.txt' OR dc:title =
'testfile3_Title'
SELECT * FROM Document WHERE file:filename = 'testfile.txt' OR dc:contributors = 'bob'
SELECT * FROM Document WHERE dc:created BETWEEN DATE '2007-03-15' AND DATE
'2008-01-01'
SELECT * FROM Document WHERE dc:created NOT BETWEEN DATE '2007-01-01' AND DATE
```

```
'2008-01-01' -- (VCS only)
SELECT * FROM Document WHERE dc:contributors = 'bob'
SELECT * FROM Document WHERE dc:contributors IN ('bob', 'john')
SELECT * FROM Document WHERE dc:contributors NOT IN ('bob', 'john')
SELECT * FROM Document WHERE dc:contributors <> 'pete'
SELECT * FROM Document WHERE dc:contributors <> 'blah'
SELECT * FROM Document WHERE dc:contributors <> 'blah' AND ecm:isProxy = 0
SELECT * FROM Document WHERE dc:title LIKE 'testfile%' ORDER BY dc:description
SELECT * FROM Document WHERE dc:title LIKE 'testfile%' ORDER BY dc:description DESC
SELECT * FROM Document ORDER BY ecm:path
SELECT * FROM Document ORDER BY ecm:path DESC
SELECT * FROM Document ORDER BY ecm:name
SELECT * FROM Document WHERE ecm:path STARTSWITH '/'
SELECT * FROM Document WHERE ecm:path STARTSWITH '/nothere/'
SELECT * FROM Document WHERE ecm:path STARTSWITH '/testfolder1/'
SELECT * FROM Document WHERE dc:title = 'testfile1_Title' AND ecm:path STARTSWITH '/'
SELECT * FROM Document WHERE dc:title LIKE 'testfile%' AND ecm:path STARTSWITH '/'
SELECT * FROM Document WHERE dc:coverage STARTSWITH 'foo'
SELECT * FROM Document WHERE dc:coverage STARTSWITH 'foo/bar'
SELECT * FROM Document WHERE dc:subjects STARTSWITH 'gee'
SELECT * FROM Document WHERE dc:subjects STARTSWITH 'gee/moo'
SELECT * FROM Document WHERE dc:created >= DATE '2007-01-01'
SELECT * FROM Document WHERE dc:created >= TIMESTAMP '2007-03-15 00:00:00'
SELECT * FROM Document WHERE dc:created >= DATE '2007-02-15' AND dc:created <= DATE
'2007-03-15'
SELECT * FROM Document WHERE my:boolean = 1
SELECT * FROM Document WHERE ecm:isProxy = 1
SELECT * FROM Document WHERE ecm:isCheckedInVersion = 1
SELECT * FROM Document WHERE ecm:isProxy = 0 AND ecm:isCheckedInVersion = 0
SELECT * FROM Document WHERE ecm:uuid = 'c5904f77-299a-411e-8477-81d3102a81f9'
SELECT * FROM Document WHERE ecm:name = 'foo'
SELECT * FROM Document WHERE ecm:parentId = '5442fff5-06f1-47c9-ac59-1e10ef8e985b'
SELECT * FROM Document WHERE ecm:primaryType = 'Folder'
SELECT * FROM Document WHERE ecm:primaryType <> 'Folder'
SELECT * FROM Document WHERE ecm:primaryType = 'Note'
SELECT * FROM Document WHERE ecm:primaryType IN ('Folder', 'Note')
SELECT * FROM Document WHERE ecm:primaryType NOT IN ('Folder', 'Note')
SELECT * FROM Document WHERE ecm:mixinType = 'Versionable' AND ecm:mixinType <>
'Downloadable'
SELECT * FROM Document WHERE ecm:mixinType <> 'Rendition'
SELECT * FROM Document WHERE ecm:mixinType = 'Rendition' AND dc:title NOT ILIKE '%pdf'
SELECT * FROM Document WHERE ecm:mixinType = 'Folderish'
SELECT * FROM Document WHERE ecm:mixinType = 'Downloadable'
SELECT * FROM Document WHERE ecm:mixinType = 'Versionable'
SELECT * FROM Document WHERE ecm:mixinType IN ('Folderish', 'Downloadable')
SELECT * FROM Document WHERE ecm:mixinType NOT IN ('Folderish', 'Downloadable')
SELECT * FROM Document WHERE ecm:currentLifeCycleState = 'project'
SELECT * FROM Document WHERE ecm:versionLabel = '1.0'
SELECT * FROM Document WHERE ecm:currentLifeCycleState <> 'deleted'
SELECT * FROM Document WHERE ecm:fulltext = 'world'
SELECT * FROM Document WHERE dc:title = 'hello world 1' ORDER BY
ecm:currentLifeCycleState
SELECT * FROM Document WHERE dc:title = 'hello world 1' ORDER BY ecm:versionLabel
```

```
SELECT * FROM Document WHERE ecm:parentId = '62cc5f29-f33e-479e-b122-e3922396e601'
ORDER BY ecm:pos
```

Since Nuxeo 5.4.1 you can use `IS NULL`:

```
SELECT * FROM Document WHERE dc:expired IS NOT NULL
SELECT * FROM Document WHERE dc:language = '' OR dc:language IS NULL
```

Since Nuxeo 5.5 you can use complex properties:

```
SELECT * FROM File WHERE content/length > 0
SELECT * FROM File WHERE content/name = 'testfile.txt'
SELECT * FROM File ORDER BY content/length DESC
SELECT * FROM Document WHERE tst:couple/first/firstname = 'Steve'
SELECT * FROM Document WHERE tst:friends/0/firstname = 'John'
SELECT * FROM Document WHERE tst:friends/*/firstname = 'John'
SELECT * FROM Document WHERE tst:friends/*1/firstname = 'John' AND
tst:friends/*1/lastname = 'Smith'
SELECT tst:friends/*1/lastname FROM Document WHERE tst:friends/*1/firstname = 'John'
SELECT * FROM Document WHERE dc:subjects/0 = 'something'
SELECT * FROM Document WHERE dc:subjects/* = 'something'
SELECT dc:subjects/*1 FROM Document WHERE dc:subjects/*1 LIKE 'abc%'
```

Since Nuxeo 5.7.1 you can use `ecm:tag`:

```
SELECT * FROM Document WHERE ecm:tag = 'tag1'
SELECT * FROM Document WHERE ecm:tag IN ('tag1', 'tag2')              -- documents
with either tag
SELECT * FROM Document WHERE ecm:tag/* = 'tag1' AND ecm:tag/* = 'tag2' -- documents
with both tags
SELECT ecm:tag FROM Document WHERE dc:title = 'something'             -- with
queryAndFetch
SELECT ecm:tag FROM Document WHERE ecm:tag LIKE 'abc%'                -- with
queryAndFetch
SELECT ecm:tag/*1 FROM Document WHERE ecm:tag/*1 LIKE 'abc%' AND ecm:tag/*2 = 'tag1'
SELECT ecm:tag FROM Document WHERE ecm:tag LIKE 'abc%' AND ecm:tag/* = 'tag1'  --
simpler version of above
```

Since Nuxeo 5.7.1 you can also use `ecm:proxyTargetId` and `ecm:proxyVersionableId`:

```
SELECT * FROM Document WHERE ecm:proxyTargetId =
'62cc5f29-f33e-479e-b122-e3922396e601'
SELECT * FROM Document WHERE ecm:proxyVersionableId =
'5442fff5-06f1-47c9-ac59-1e10ef8e985b'
```

Since Nuxeo 5.7.3 you can match the "checked in" state of a document:

```
 SELECT * FROM Document WHERE ecm:isCheckedIn = 1
```

Since Nuxeo 5.7.3 you can use additional version-related properties:

```
SELECT * FROM Document WHERE ecm:isVersion = 1                        -- new name for
ecm:isCheckedInVersion
SELECT * FROM Document WHERE ecm:isLatestVersion = 1
SELECT * FROM Document WHERE ecm:isLatestMajorVersion = 1
SELECT * FROM Document WHERE ecm:versionCreated >= TIMESTAMP '2007-03-15 00:00:00'
SELECT * FROM Document WHERE ecm:versionLabel = '1.0'                 -- this was
available even before Nuxeo 5.7.3
SELECT * FROM Document WHERE ecm:versionDescription LIKE '%TODO%'
SELECT * FROM Document WHERE ecm:versionVersionableId =
'5442fff5-06f1-47c9-ac59-1e10ef8e985b'
```

Fulltext Examples

This uses standard SQL LIKE:

```
SELECT * FROM Document WHERE dc:title LIKE 'Test%'
SELECT * FROM Document WHERE dc:title ILIKE 'test%'
SELECT * FROM Document WHERE dc:contributors LIKE 'pe%'
SELECT * FROM Document WHERE dc:subjects LIKE '%oo%'
SELECT * FROM Document WHERE dc:subjects NOT LIKE '%oo%'
```

The following uses a fulltext index that has to be additionally configured by administrators:

```
SELECT * FROM Document WHERE ecm:fulltext_title = 'world'
```

The following uses a fulltext index if one is configured for the dc:title field, otherwise it uses ILIKE-based queries:

```
SELECT * FROM Document WHERE ecm:fulltext.dc:title = 'brave'
```

Full-Text Queries

Nuxeo documents can be searched using fulltext queries; the standard way to do so is to use the top-right "quick search" box in Nuxeo DM. Search queries are expressed in a Nuxeo-defined syntax, described below.

In this section

Nuxeo Fulltext Query Syntax

A fulltext query is a sequence of space-separated words, in addition:

- Words are implicitly `AND`-ed together.
- A word can start with `-` to signify negation (the word must not be present).
- A word can end with `*` to signify prefix search (the word must start with this prefix).
- You can use `OR` between words (it has a lower precedence than the implicit `AND`).
- You can enclose several words in double quotes `"` for a phrase search (the words must exactly follow each other).

Examples:

Documents containing both `hello` and `world` and which do not contain `smurf`:

```
hello world -smurf
```

Documents containing `hello` and a word starting with `worl`:

```
hello worl*
```

Documents containing both `hello` and `world`, or documents containing `smurf` but not containing `black`:

```
hello world OR smurf -black
```

Documents containing `hello` followed by `world` and also containing `smurf`:

```
"hello world" smurf
```

Important notes:

1. A query term (sequence of `AND`-ed words without an `OR`) containing only negations will not match anything.
2. Depending on the backend database and its configuration, different word stemming strategies may be used, which means that `universes` and `universal` (for instance) may or may not be considered the same word. Please check your database configuration for more on this, and the "analyzer" parameter used in the Nuxeo configuration for your database.
3. Phrase search using a PostgreSQL backend database is supported only since Nuxeo 5.5 and cannot use word stemming (*i.e.*, a query of `"hello worlds"` will not match a document containing just `hello world` without a final `s`). This is due to way this feature is implemented, which is detailed at NXP-6720.

Also of interest:

1. In Nuxeo 5.3, searches using an `OR` and phrase searches are not supported, these features are only available since Nuxeo 5.4.
2. Prefix search is supported in all databases only since Nuxeo 5.5.
3. Ending a word with `%` instead of `*` for prefix search is also supported for historical reasons.

Using Fulltext Queries in NXQL

In NXQL the fulltext query is part of a WHERE clause that can contain other matches on metadata. Inside the WHERE clause, a fulltext query for "something" (as described in the previous section) can be expressed in several ways:

- `ecm:fulltext = 'something'`
- `ecm:fulltext_someindex = 'something'` if an index called "someindex" is configured in the VCS configuration
- `ecm:fulltext.somefield = 'something'` to search a field called "somefield", using fulltext if the VCS configuration contains a single index for it, or if not using fallback to a standard SQL ILIKE query: `somefield ILIKE '%something%'` (ILIKE is a case-independent LIKE). Note that this will have a serious performance impact if no fulltext is used, and is provided only to help migrations from earlier versions.
- `ecm:fulltext LIKE 'something'` is deprecated but identical to `ecm:fulltext = 'something'`.

Search Results Optimizations

When there are a lot of search results, search becomes slow, which results in very slow response times when users browse the application, which can even be unresponsive. When monitoring the server, memory is overused.

The problem comes from the fact that "DocumentModelImpl" objects are created without making allowance for pagination. As the number of search results increase, so does the number of "DocumentModelImpl" objects. This problem is usually not detected during functional and performance testing because:

- it doesn't cause any functional bug,
- performance tests are usually about concurrent use of the application.

When a query is done, the only reason that explains the lack of pagination is when post-filtering is activated. It is activated by default when security policies are used in the application.

See the line 142 of `org.nuxeo.ecm.core.api.AbstractSession`:

```
postFilterPermission = false;
                String repoName = getRepositoryName();
                postFilterPolicies =
!securityService.arePoliciesExpressibleInQuery(repoName);
                postFilterFilter = filter != null
                        && !(filter instanceof FacetFilter);
                postFilter = postFilterPolicies || postFilterFilter;
```

and line 1442 of `org.nuxeo.ecm.core.api.AbstractSession`:

```
DocumentModelList dms = results.getDocumentModels();
            if (!postFilter) {
                // the backend has done all the needed filtering
                return dms;
            }
            // post-filter the results "by hand", the backend couldn't do it
            long start = limit == 0 || offset < 0 ? 0 : offset;
            long stop = start + (limit == 0 ? dms.size() : limit);
            int n = 0;
            DocumentModelListImpl docs = new DocumentModelListImpl();
            for (DocumentModel model : dms) {
```

Suggested Fix

When there are potentially a lot of search results (content views CoreQueryProvider for instance), security policies should explicitly provide a query "transformer". Otherwise, all the search results (DocumentModelImpl) are cached to perform a post-filtering.

To ignore security policies upon search, the `org.nuxeo.ecm.core.security.AbstractSecurityPolicy` methods are ignored for each security policy used:

```
@Override
    public boolean isExpressibleInQuery(String repositoryName) {
        return true;
    }
    @Override
    public Transformer getQueryTransformer(String repositoryName) {
        return Transformer.IDENTITY;
    }
```

- `isExpressibleInQuery` determines if a NXQL query can be used instead of the security policy. If yes, `getQueryTransformer` will be used.
- `getQueryTransformer` gets the search query and adapts it to add some parameters if needed. If the original query is enough, `Transformer.IDENTITY` can be used.

So, you have to identify all security policies and make sure they can be expressed as a NXQL query.

Checking

Performance and Memory Improvements

To check how performance and memory are impacted:

Using JVisualVM, compare the DocumentModelImpl sampling results when you access a content view that returns a lot of search results with and without the modifications indicated in the above section.

If the results with the modifications are not satisfactory, you need to debug on `AbstractSession` and understand why post-filtering is activated.

You should also test the access to content views in general. The application should be more responsive.

Non-Regression Testing

Since the correction impacts the content displayed when users browse the application, you need to make sure that the expected documents are displayed on the different content views. You can (or even should) run the unit and Selenium tests (or any other tests you may have) to make sure there is no regression.

Tagging

The tag service provides the backbone of the tagging feature. Tags are keywords applied as metadata on documents reflecting (for instance) the user opinion about that document. The tags are either categorizing the content of the document (labels like "document management", "ECM", "complex Web application", etc. can be thought as tags for Nuxeo), or they reflect the user feeling ("great", "user friendly", "versatile", etc.).
The tag service uses two important concepts: a *tag* object, and a *tagging* action. Both are represented as Nuxeo documents.

A **tag** holds a label that does not contain any space ("documentmanagement", "webapplication", etc.). A **tagging** action is a link between a given document and a tag, and belongs to a given user.

On this page

Tag Service Architecture

The following document types are defined by the tag service.

A **Tag** is a document type representing the tag itself (but not its association to specific documents). It contains the usual `dublincore` schem a, and in addition has a specific **tag** schema containing a **tag:label** string field.

A **Tagging** is a relation type representing the action of tagging a given document with a tag. (A relation type is a document type extending the default Relation document type; it works like a normal document type except that it's not found by NXQL queries on `Document`). The important fields of a Tagging document are **relation:source** which is the document id, **relation:target** which is the tag id, and **dc:creator** whi ch is the user doing the tagging action.

Both Tag and Tagging documents managed by the tag service are *unfiled*, which means that they don't have a parent folder. They are therefore not visible in the normal tree of documents, only queries can find them. In addition they don't have any ACLs set on them, which means that only a superuser (and the tag service internal code) can access them.

Tag Service Features

The tag service is accessed through the `org.nuxeo.ecm.platform.tag.TagService` interface.

The tag service allows you to:

- tag and untag a document,
- get all the tags for a document,
- get all the documents for a tag,
- get the tag cloud for a set of documents,
- get suggested tags for a given tag prefix.

A tag cloud is a set of weighted tags, the weights being integers representing the frequency of the tag.
The weights can be just a count of occurrences, or can be normalized to the 0-100 range for easier display.

Tags, Versions and Proxies

Since 5.7.3, the tags are duplicated:

- from the live document when creating a version
- from a version when creating a proxy

When restoring a version, the tags on the live document are also restored from the ones on the version.

Tags can be added and removed independently on live documents, versions and proxies: a tag added on a live document won't be added on all its versions, but only on the versions that will be created after. The same behavior is applied for proxies.

Everything is done in the `org.nuxeo.ecm.platform.tag.TaggedVersionListener` listener.

Disabling Tags on Versions and Proxies

To disable the duplication of tags on versions and proxies, the `org.nuxeo.ecm.platform.tag.TaggedVersionListener` listener needs to be disabled with the following contribution:

```
<require>org.nuxeo.ecm.platform.tag.service.listener</require>
<extension target="org.nuxeo.ecm.core.event.EventServiceComponent"
  point="listener">
  <listener name="taggedVersionListener" enabled="false" />
</extension>
```

Related pages in current documentation

- Publisher service
- Versioning
- Events and Listeners

Related pages in other documentation

- Tags
- Publishing Documents

Security Policy Service

The Security Policy Service provides an extension point to plug custom security policies that do not rely on the standard ACLs for security. For instance, it can be used to define permissions according to the document metadata, or information about the logged in user.

Security Policy Architecture

A security policy is a class implementing the `org.nuxeo.ecm.core.security.SecurityPolicy` interface; it is strongly advised to extend `org.nuxeo.ecm.core.security.AbstractSecurityPolicy` for future compatibility.

The class must be registered through the `policies` extension point of the `org.nuxeo.ecm.core.security.SecurityService` component.

A security policy has two important aspects, materialized by different methods of the interface:

- how security is checked on a given document (method `checkPermission`),
- how security is applied to NXQL and CMISQL searches (methods `getQueryTransformer`).

In this section

Document Security Check

To check security on a given document, Nuxeo Core calls `checkPermission` with a number of parameters, among which the document, the user and the permission to check, on all the security policies registered. The policies should return `Access.DENY` or `Access.UNKNOWN` based on the information provided. If `Access.DENY` is returned, then access to the document (for the given permission) will be denied. If `Access.UNKNOWN` is returned, then other policies will be checked. Finally if all policies return `Access.UNKNOWN` then standard Nuxeo EP ACLs will be checked.

There is a third possible value, `Access.GRANT`, which can immediately grant access for the given permission, but this bypasses ACL checks. This is all right for most permissions but not for the `Browse` permission, because `Browse` is used for NXQL and CMISQL searches in which case it's recommended to implement `getQueryTransformer` instead (see below).

Note that `checkPermission` receives a document which is a `org.nuxeo.ecm.core.model.Document` instance, different from and at a lower level than the usual `org.nuxeo.ecm.core.api.DocumentModel` manipulated by user code.

Unrestricted sessions
If the query has been called in the context of an unrestricted session, the principal will be `system`. It is a good practice to check for that username since if the query is run unrestrictedly, it functionally means that you should not restrict anything with the query transformer

NXQL Security Check

All NXQL queries have ACL-based security automatically applied with the `Browser` permission (except for superusers).

A security policy can modify this behavior but only by adding new restrictions in addition to the ACLs. To do so, it can simply implement the `checkPermission` described above, but this gets very costly for big searches. The efficient approach is to make `isExpressibleInQuery` return `true` and implement `getQueryTransformer`.

The `getQueryTransformer(repositoryName)` method returns a `SQLQuery.Transformer` instance, which is a class with one `transform` method taking a NXQL query in the form of a `org.nuxeo.ecm.core.query.sql.model.SQLQuery` abstract syntax tree. It should transform this tree in order to add whatever restrictions are needed. Note that ACL checks will always be applied after this transformation.

CMISQL Security Check

Since Nuxeo 5.6.0-HF21 and Nuxeo 5.7.2, all CMISQL queries also require implementation of the relevant `getQueryTransformer` API in order to secure CMIS-based searches.

The `getQueryTransformer(repositoryName, "CMISQL")` method returns a `SecurityPolicy.QueryTransformer` instance, which is a class with one `transform` method taking a query in the form of `String`. It should transform this query in order to add whatever restrictions are needed (this will require parsing the CMISQL and adding whatever clauses are needed). Note that ACL checks will always be applied after this transformation.

Example Security Policy Contribution

To register a security policy, you need to write a contribution specifying the class name of your implementation.

```
<?xml version="1.0"?>
<component name="com.example.myproject.securitypolicy">

  <extension target="org.nuxeo.ecm.core.security.SecurityService"
    point="policies">

    <policy name="myPolicy"
      class="com.example.myproject.NoFileSecurityPolicy" order="0" />

  </extension>

</component>
```

Here is a sample contributed class:

```
import org.nuxeo.ecm.core.query.sql.model.*;
import org.nuxeo.ecm.core.query.sql.NXQL;
import org.nuxeo.ecm.core.security.AbstractSecurityPolicy;
import org.nuxeo.ecm.core.security.SecurityPolicy;

/**
 * Sample policy that denies access to File objects.
 */
public class NoFileSecurityPolicy extends AbstractSecurityPolicy implements
SecurityPolicy {

    @Override
    public Access checkPermission(Document doc, ACP mergedAcp,
            Principal principal, String permission,
            String[] resolvedPermissions, String[] additionalPrincipals) {
```

```
        // Note that doc is NOT a DocumentModel
        if (doc.getType().getName().equals("File")) {
            return Access.DENY;
        }
        return Access.UNKNOWN;
    }

    @Override
    public boolean isRestrictingPermission(String permission) {
        // could only restrict Browse permission, or others
        return true;
    }

    @Override
    public boolean isExpressibleInQuery() {
        return true;
    }

    @Override
    public SQLQuery.Transformer getQueryTransformer() {
        return NO_FILE_TRANSFORMER;
    }

    public static final Transformer NO_FILE_TRANSFORMER = new NoFileTransformer();

    /**
     * Sample Transformer that adds {@code AND ecm:primaryType <> 'File'} to the
query.
     */
    public static class NoFileTransformer implements SQLQuery.Transformer {

        /** {@code ecm:primaryType <> 'File'} */
        public static final Predicate NO_FILE = new Predicate(
                new Reference(NXQL.ECM_PRIMARYTYPE), Operator.NOTEQ, new
StringLiteral("File"));

        @Override
        public SQLQuery transform(Principal principal, SQLQuery query) {
            WhereClause where = query.where;
            Predicate predicate;
            if (where == null || where.predicate == null) {
                predicate = NO_FILE;
            } else {
                // adds an AND ecm:primaryType <> 'File' to the WHERE clause
                predicate = new Predicate(NO_FILE, Operator.AND, where.predicate);
            }
            // return query with updated WHERE clause
            return new SQLQuery(query.select, query.from, new WhereClause(predicate),
                    query.groupBy, query.having, query.orderBy, query.limit,
query.offset);
        }
    }
```

```
}
```

CMISQL Security Checks

To find examples of security policies using CMISQL query transformers, please check the `TitleFilteringSecurityPolicy2` in the unit tests.

Events and Listeners

Events and event listeners have been introduced at the Nuxeo core level to allow pluggable behaviors when managing documents (or any kinds of objects of the site).

Whenever an event happens (document creation, document modification, relation creation, etc...), an event is sent to the event service that dispatches the notification to its listeners. Listeners can perform whatever action it wants when receiving an event.

Concepts

A core event has a source which is usually the document model currently being manipulated. It can also store the event identifier, that gives information about the kind of event that is happening, as well as the principal connected when performing the operation, an attached comment, the event category, etc.

Events sent to the event service have to follow the `org.nuxeo.ecm.core.event.Event` interface.

A core event listener has a name, an order, and may have a set of event identifiers it is supposed to react to. Its definition also contains the operations it has to execute when receiving an interesting event.

Event listeners have to follow the `org.nuxeo.ecm.core.event.EventListener` interface.

Several event listeners exist by default in the nuxeo platform, for instance:

- `DublinCoreListener`: it listens to document creation/modification events and sets some Dublin Core metadata accordingly (date of creation, date of last modification, document contributors...)
- `DocUIDGeneratorListener`: it listens to document creation events and adds an identifier to the document if an UID pattern has been defined for this document type.
- DocVersioningListener: it listens to document versioning change events and changes the document version numbers accordingly.

In this section

Registering an Event Listener

Event listeners can be registered using extension points. Here are example event listeners registrations from Nuxeo EP:

```
<component name="DublinCoreStorageService">
  <extension target="org.nuxeo.ecm.core.event.EventServiceComponent" point="listener">
    <listener name="dclistener" async="false" postCommit="false" priority="120"
      class="org.nuxeo.ecm.platform.dublincore.listener.DublinCoreListener">
    </listener>
  </extension>
</component>
```

```
<component name="org.nuxeo.ecm.platform.annotations.repository.listener">
  <extension target="org.nuxeo.ecm.core.event.EventServiceComponent" point="listener">
    <listener name="annotationsVersionEventListener" async="true" postCommit="true"

class="org.nuxeo.ecm.platform.annotations.repository.service.VersionEventListener">
      <event>documentCreated</event>
      <event>documentRemoved</event>
      <event>versionRemoved</event>
      <event>documentRestored</event>
    </listener>
  </extension>
</component>
```

When defining an event listener, you should specify:

- a name, useful to identify the listener and let other components disable/override it,
- whether the listener is synchronous or asynchronous (default is synchronous),
- whether the listener runs post-commit or not (default is false),
- an optional priority among similar listeners,
- the implementation class, that must implement `org.nuxeo.ecm.core.event.EventListener`,
- an optional list of event ids for which this listener must be called.

There are several kinds of listeners:

- **synchronous inline listeners** are run immediately in the same transaction and same thread, this is useful for listeners that must modify the state of the application like the *beforeDocumentModification* event.
- **synchronous post-commit listeners** are run after the transaction has committed, in a new transaction but in the same thread, this is useful for logging.
- **asynchronous listeners** are run after the transaction has committed, in a new transaction and a separate thread, this is useful for any long-running operations whose result doesn't have to be seen immediately in the user interface.

Processing an Event Using an Event Listener

Here is a simple event listener:

```
import org.nuxeo.ecm.core.event.Event;
import org.nuxeo.ecm.core.event.EventContext;
import org.nuxeo.ecm.core.event.EventListener;
import org.nuxeo.ecm.core.event.impl.DocumentEventContext;

public class BookEventListener implements EventListener {

    public void handleEvent(Event event) throws ClientException {
        EventContext ctx = event.getContext();
        if (!(ctx instanceof DocumentEventContext)) {
            return;
        }
        DocumentModel doc = ((DocumentEventContext) ctx).getSourceDocument();
        if (doc == null) {
            return;
        }
        String type = doc.getType();
        if ("Book".equals(type)) {
            process(doc);
        }
    }

    public void process(DocumentModel doc) throws ClientException {
        ...
    }
}
```

Note that if a listener expects to modify the document upon save or creation, it must use events *emptyDocumentModelCreated* or *beforeDocumentModification*, and **not** save the document, as these events are themselves fired during the document save process.

If a listener expects to observe a document after it has been saved to do things on other documents, it can use events *documentCreated* or *documentModified*.

Sending an Event

It is not as common as having new listeners, but sometimes it's useful to send new events. To do this you have to create an event bundle containing the event, then send it to the event producer service:

```
EventProducer eventProducer;
try {
    eventProducer = Framework.getService(EventProducer.class);
} catch (Exception e) {
    log.error("Cannot get EventProducer", e);
    return;
}

DocumentEventContext ctx = new DocumentEventContext(session, session.getPrincipal(),
doc);
ctx.setProperty("myprop", "something");

Event event = ctx.newEvent("myeventid");
try {
    eventProducer.fireEvent(event);
} catch (ClientException e) {
    log.error("Cannot fire event", e);
    return;
}
```

You can also have events be sent automatically at regular intervals using the Scheduling Periodic Events, see that section for mor

Handling Errors

Sometimes, you may want to handle errors that occurred in an inline listener in the UI layer. This is a little bit tricky but do-able.

In the listener, you should register the needed information in a place that is shared with the UI layer.
You can use the document context map for this.

```
...
   DocumentEventContext ctx = (DocumentEventContext)event.getContext();
   DocumentModel doc = ctx.getSourceDocument();
   ScopedMap data = doc.getContextData();
   data.putScopedValue(MY_ERROR_KEY, "some info");
...
```

Another thing is to insure that the current transaction will be roll-backed. Marking the event as rollback
makes the event service throwing a runtime exception when returning from the listener.

```
...
  TransactionHelper.setTransactionRollbackOnly();
  event.markRollback();
  throw new ClientException("rollbacking");
...
```

Then the error handling in the UI layer can be managed like this

```
...
  DocumentModel doc = ...;
  try {
     // doc related operations
  } catch (Exception e) {
     Serializable info = doc.getContextData(MY_ERROR_KEY);
     if (info == null) {
       throw e;
     }
     // handle code
  }
...
```

Asynchronous vs Synchronous Listeners

Asynchronous listeners will run in a separated thread (actually in the Workmanager using a processing queue since 5.6): this means the main transaction, the one that raised the event, won't be blocked by the listener processing.

So, if the processing done by the listener may be long, this is a good candidate for async processing.

However, there are some impacts in moving a sync listener to an async one:

- the listener signature needs to change
  - rather than receiving a single event, it will receive an EventBundle that contains all event inside the transaction
- because the listener runs in a separated transaction it can not rollback the source transaction (it is too late anyway)
- the listener code need to be aware that it may have to deal with new cases
  - typically, the listener may receive an event about a document that has since then been deleted

However, it is easy to have a single listener class that exposes both interfaces and can be use both synchronously and asynchronously.

Performances and Monitoring

Using listeners, especially synchronous one may impact the global performance of the system.

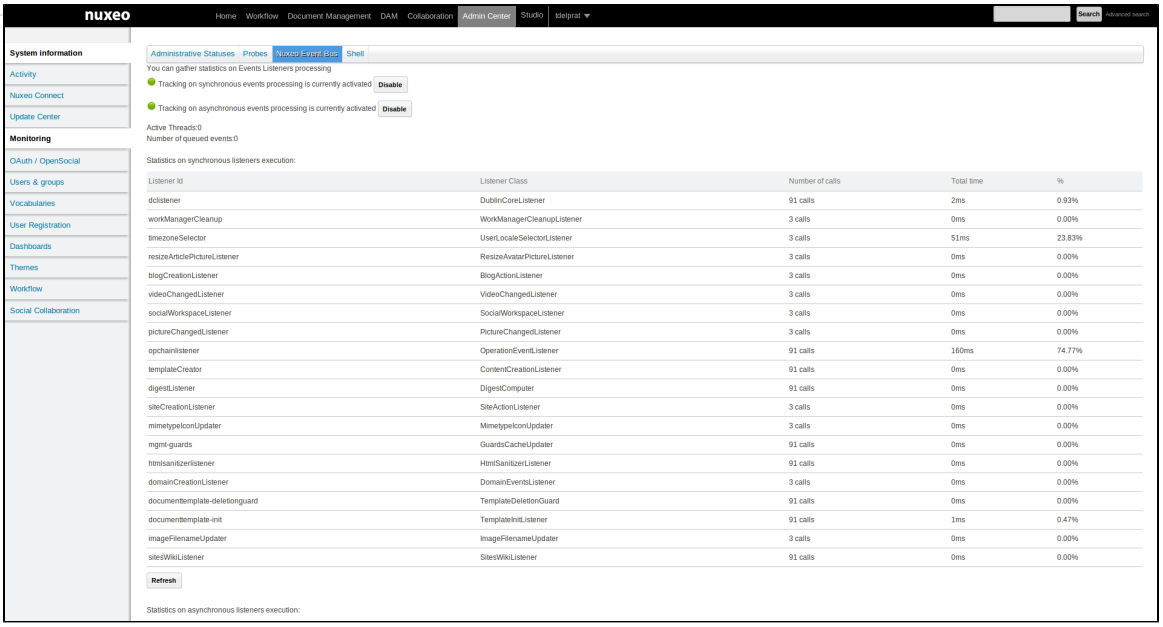
Typically, having synchronous listeners that do long processing will reduce the scalability of the system.

In that kind of case, using asynchronous listener is the recommended approach:

- the interactive transaction is no longer tried to listener execution
- Workmanager allow to configure how many async listeners can run in concurrency

To monitor execution of the listeners, you can use the Admin Center / Monitoring / Nuxeo Event Bus to

- activate the tracking of listeners execution,
- see how much time is spent in each listener.



For diagnostic and testing purpose, you can use the EventAdminService to activate / deactivate listeners one by one.

The EventServiceAdmin is accessible :

- via Java API,
- via JMX.

Common Events

Any Nuxeo code can define its own events, but it's useful to know some of the standard ones that Nuxeo sends by default.

Basic Events

aboutToRemove-/ aboutToRemoveVersion

A document or a version are about to be removed.

beforeDocumentModification

A document is about to be saved after a modification. A synchronous listener may update the DocumentModel but must not save it (this will be done automatically).

beforeDocumentSecurityModification

A document's ACLs are about to change.

documentCreated

A document has been created (this implies that a write to the database has occurred). Be careful, this event is sent for all creations, including for new versions and new proxies.

documentLocked - documentUnlocked

A document has been locked or unlocked.

documentModified

A document has been saved after a modification.

documentRemoved - versionRemoved

A document or a version have been removed.

documentSecurityUpdated

A document's ACLs have changed.

emptyDocumentModelCreated

The data structure for a document has been created, but nothing is saved yet. This is useful to provide default values in document creation forms.

lifecycle_transition_event

A transition has been followed on a document.

sessionSaved

The session has been saved (all saved documents are written to database).

In this section

- Publishing Events
  - Low-Level Events
    - documentProxyPublished
    - documentProxyUpdated
    - sectionContentPublished
  - High-Level Events
    - documentPublicationApproved
    - documentPublicationRejected
    - documentPublished
    - documentUnpublished
    - documentWaitingPublication

Copy and Move Events

aboutToCopy - aboutToMove

A document is about to be copied or moved.

documentCreatedByCopy

A document has been copied, the passed document is the new copy.

documentDuplicated

A document has been copied, the passed document is the original.

documentMoved

A document has been moved.

Versioning Events

beforeRestoringDocument

A document is about to be restored.

documentRestored

A document has been restored.

incrementBeforeUpdate

A document is about to be snapshotted as a new version. The changes made to the passed DocumentModel will be saved in the archived version but will not be seen by the main document being versioned. This is useful to update version numbers and version-related information.

Publishing Events

Low-Level Events

documentProxyPublished

A proxy has been published (or updated).

documentProxyUpdated

A proxy has been updated.

sectionContentPublished

New content has been published in this section.

High-Level Events

documentPublicationApproved

A document waiting for approval has been approved.

documentPublicationRejected

A document waiting for approval has been rejected.

documentPublished

A document has been published (event sent for the base document being published and for the published document).

documentUnpublished

documentWaitingPublication

A document has been published but is not approved yet (event sent for the base document being published and for the published document)

Scheduling Periodic Events

The Scheduler Service is a Nuxeo Platform service to schedule events at periodic times. This is the

best way in Nuxeo Platform to execute things every night, every hour, every five minutes, or at whatever granularity you require.

Scheduler Contribution

To schedule an event, you contribute a `<schedule>` to the `schedule` extension point of the `org.nuxeo.ecm.core.scheduler.SchedulerService` component.

In this section

Component name
Before Nuxeo 5.6, the component name was `org.nuxeo.ecm.platform.scheduler.core.service.SchedulerRegistryService`.

A schedule is defined by:

- **id**: an identifier,
- **username**: the user under which the event should be executed,
- **event**: the identifier of the event to execute,
- **eventCategory**: the event category to use,
- **cronExpression**: an expression to specify the schedule.

The **id** is used for informational purposes and programmatic unregistration.

If the **username** is missing, the event is executed as a system user, otherwise as that user. Note that since Nuxeo EP 5.4.1 no password is needed (the login is done internally and does not need password).

The **event** specifies the event to execute. See the section about Events and Listeners for more.

The **eventCategory** is also used to specify the event, but usually it can be skipped.

The **cronExpression** is described in the following section.

Here is an example contribution:

```
<?xml version="1.0"?>
<component name="com.example.nuxeo.schedule.monthly_stuff">
  <extension target="org.nuxeo.ecm.core.scheduler.SchedulerService"
      point="schedule">
    <schedule id="monthly_stuff">
      <username>Administrator</username>
      <eventId>doStuff</eventId>
      <eventCategory>default</eventCategory>
      <!-- Every first of the month at 3am -->
      <cronExpression>0 0 3 1 * ?</cronExpression>
    </schedule>
  </extension>
</component>
```

Cron Expression

A Scheduler cron expression is similar to a Unix cron expression, except that it has an additional *seconds* field that isn't needed in Unix which doesn't need this kind of precision.

The expression is a sequence of 6 or 7 fields. Each field can hold a number or a wildcard, or in complex cases a sequence of numbers or an additional increment specification. The fields and their allowed values are:

seconds	minutes	hours	day of month	month	day of week	year
0-59	0-59	0-23	1-31	1-12	1-7 or SUN-SAT	optional

A star (*) can be used to mean "all values". A question mark (?) can be used to mean "no specific value" and is allowed for one (but not both) of

the **day of month** and **day of week** fields.

Note that in the **day of week**, 1 stands for Sunday, 2 for Monday, 3 for Tuesday, etc. For clarity it's best to use SUN, MON, TUE, etc.

A range of values can be specified using a dash, for instance `1-6` for the months field or `MON-WED` for the day of week field.

Several values can be specified by separating them with commas, for instance `0,15,30,45` for the minutes field.

Repetitions can be specified using a slash followed by an increment, for instance `0/15` means start at 0 and repeat every 15. This example means the same as the one above.

There's actually more but rarely used functionality; the Scheduler's full cron expression syntax is described in detail in the Quartz CronExpression Javadoc and in the CronTrigger Tutorial.

Cron Expression Examples

Every first of the month at 3:15am:

```
0 15 3 1 * ?
```

At 3:15am every day:

```
0 15 3 * * ?
```

Every minute starting at 2pm and ending at 2:15pm, every day:

```
0 0-15 14 * * ?
```

At 3:15am every Monday, Tuesday, Wednesday, Thursday and Friday:

```
0 15 3 ? * MON-FRI
```

At 3:15a, every 5 days every month, starting on the first day of the month:

```
0 15 3 1/5 * ?
```

Bulk Edit

The bulk edit feature allows to edit several documents at the same time. This is implemented using the BulkEditService component.

Customizing the Bulk Edit Form

The default bulk edit form is based on a layout called `bulkEdit@edit`. To change it you just need to override the default one by your own b`ulkEdit@edit` layout to display your own widgets.

In this section

Default bulk edit layout

```
<require>org.nuxeo.ecm.platform.forms.layouts.webapp</require>
<extension target="org.nuxeo.ecm.platform.forms.layout.WebLayoutManager"
    point="layouts">
  <layout name="bulkEdit@edit">
    <templates>
      <template mode="any">/layouts/layout_bulkedit_template.xhtml
      </template>
    </templates>
    <rows>
      <row>
        <widget>nature</widget>
      </row>
      <row>
        <widget>subjects</widget>
      </row>
      <row>
        <widget>rights</widget>
      </row>
      <row>
        <widget>source</widget>
      </row>
      <row>
        <widget>coverage</widget>
      </row>
      <row>
        <widget>format</widget>
      </row>
      <row>
        <widget>language</widget>
      </row>
      <row>
        <widget>expired</widget>
      </row>
    </rows>
  </layout>
</extension>
```

Customizing the Bulk Edit Versioning Policy

When users edit several documents at the same time using the bulk edit form, a new version of the documents is saved before applying the modifications. The default versioning policy consists in creating a minor version.

This versioning policy can be configured through the versioning extension point:

```
<extension target="org.nuxeo.ecm.webapp.bulkedit.BulkEditService"
  point="versioning">
  <versioning>
    <defaultVersioningOption>MINOR</defaultVersioningOption>
  </versioning>
</extension>
```

Authentication and User Management

- Authentication — Nuxeo Authentication is based on the JAAS standard. Authentication infrastructure is based on two main

components:

- Adding Custom LDAP Fields to the UI — To add a custom LDAP fields to the User interface you have to:
- User Management — An abstraction, the UserManager, centralizes the way a Nuxeo Platform application deals with users (and groups of users). The UserManager is queried by the platform's LoginModule when someone attempts to authenticate against the framework. It is also queried whenever someone wants the last name or email of a user for instance, or to get all users having "Bob" as their first name.
- Using CAS2 Authentication

Authentication

Nuxeo Authentication is based on the JAAS standard. Authentication infrastructure is based on two main components:

- a JAAS Login Module: `NuxeoLoginModule`,
- a Web Filter: `NuxeoAuthenticationFilter`.

Users and groups are managed via the `UserManagerService` that handles the indirection to users and groups directories (SQL or LDAP or else).
The Nuxeo authentication framework is pluggable so that you can contribute new plugins and don't have to rewrite and reconfigure a complete JAAS infrastructure.

Pluggable JAAS Login Module

`NuxeoLoginModule` is a JAAS Login Module. It is responsible for handling all login calls within Nuxeo's security domains:

- nuxeo-ecm: for the service stack and the core,
- nuxeo-ecm-web: for the web application on the top of the service stack.

On JBoss application server, the JBoss Client Login module is used to propagate security between the web part and the service stack.

NuxeoLoginModule

`NuxeoLoginModule` mainly handles two tasks:

On this page

- login user
  This means extracting information from the CallBack stack and validating identity.
  `NuxeoLoginModule` supports several types of CallBacks (including Nuxeo specific CallBack) and uses a plugin system to be able to validate user identity in a pluggable way.

- Principal creation
  For that, `NuxeoLoginModule` uses Nuxeo UserManager service that does the indirection to the users/groups directories.

When used in conjunction with `UserIdentificationInfoCallback` (Nuxeo custom CallBack system), the `LoginModule` will choose the right `LoginPlugin` according to the CallBack information.

NuxeoLoginModule Plugins

Because validating user identity can be more complex that just checking login/password, `NuxeoLoginModule` exposes an extension point to contribute new LoginPlugins.
Each LoginPlugin has to implement the `org.nuxeo.ecm.platform.login.LoginPlugin` interface.

This interface exposes the User Identity validation logic from the `UserIdentificationInfo` object populated by the Authenticator (see the Pluggable Web Authentication Filter section):

```
String validatedUserIdentity(UserIdentificationInfo userIdent)
```

For instance, the default implementation will extract the Login/Password from `UserIdentificationInfo` and call the `checkUsernamePassword` against the `UserManager` that will validate this information against the users directory.

Other plugins can use other information carried by UserIdentificationInfo (token, ticket, ...) to validate the identity against an external SSO system. The `UserIdentificationInfo` also carries the `LoginModule` plugin name that must be used to validate identity. Even if technically, a lot of SSO systems could be implemented using this plugin system, most SSO implementations have been moved to the Authentication Plugin at the Web Filter level, because they need a HTTP dialog.

For now, the `NuxeoLoginModule` has only two ways to handle `validateUserIdentity`:

- default that uses `UserManager` to validate the couple login/password,
- Trusted_LM: this plugin assumes the user identity has already been validated by the authentication filter, so `validatedUserIdentity` will always return true.

Using Trusted_LM, a user will be logged if the user exists in the UserManager. This plugin is used for most SSO systems in conjunction with an Authentication plugin that will actually do the work of validating password or token.

Pluggable Web Authentication Filter

The Web Authentication filter is responsible for:

- Guarding access to web resources. The filter can be parameterized to guard URLs with a given pattern.
- Finding the right plugin to get user identification information. This can be getting a userName/Password, getting a token in a cookie or a header, redirecting user to another authentication server.
- Creating the `LoginContext`. This means creating the needed callBacks and call the JAAS Login.
- Storing and reestablishing login context. In order to avoid recreating a login context for each request, the `LoginContext` is cached.

NuxeoAuthenticationFilter

The `NuxeoAuthenticationFilter` is one of the top level filters in Nuxeo Web Filters stack. For each request, it will try to find a existing `LoginContext` and create a `RequestWrapper` that will carry the `NuxeoPrincipal`.

If no existing `LoginContext` is found, it will try to prompt the client for authentication information and will establish the login context.
In order to execute the task of prompting the client and retrieving `UserIndetificationInfo`, the filter will rely on a set of configured plugins.

Each plugin must:

- Implement `org.nuxeo.ecm.platform.ui.web.auth.interfaces.NuxeoAuthenticationPlugin`. The two main methods are:

```
Boolean handleLoginPrompt(HttpServletRequest httpRequest,HttpServletResponse
httpResponse, String baseURL);
UserIdentificationInfo handleRetrieveIdentity(HttpServletRequest httpRequest,
HttpServletResponse httpResponse);
```

- Define the `LoginModule` plugin to use if needed.
  Typically, SSO `AuthenticationPlugin` will do all the work and will use the Trusted_LM LoginModule Plugin.

- Define if stating URL must be saved.
  `AuthenticationPlugins`, that uses HTTP redirect in order to do the login prompt, will let the Filter store the first accessed URL in order to cleanly redirect the user to the page he asked after the authentication is successful.
  Additionally, `AuthenticationPlugin` can also implement the `org.nuxeo.ecm.platform.ui.web.auth.interfaces.NuxeoAuthenticationPluginLogoutExtension` interface if a specific processing must be done when logging out.

Here is a sample XML descriptor for registering an `AuthenticationPlugin`:

```
<?xml version="1.0"?>
<component name="org.nuxeo.ecm.platform.ui.web.auth.defaultConfig">
  <extension
    target="org.nuxeo.ecm.platform.ui.web.auth.service.PluggableAuthenticationService"
    point="authenticators">
    <authenticationPlugin name="FORM_AUTH" enabled="true"
      class="org.nuxeo.ecm.platform.ui.web.auth.plugins.FormAuthenticator">
      <needStartingURLSaving>true</needStartingURLSaving>
      <parameters>
        <parameter name="LoginPage">login.jsp</parameter>
        <parameter name="UsernameKey">user_name</parameter>
        <parameter name="PasswordKey">user_password</parameter>
      </parameters>
    </authenticationPlugin>
   </extension>
</component>
```

As you can see in the above example, the descriptor contains the parameters tag that can be used to embed arbitrary additional configuration that will be specific to a given `AuthenticationPlugin`. In the above example, it is used to define the field names and the JSP file used for form based authentication.

`NuxeoAuthenticationFilter` supports several authentication system. For example, this is useful to have users using form-based authentication and having RSS clients using Basic Authentication. Because of that, `AuthenticationPlugin` must be ordered. For that purpose, `NuxeoAuthenticationFilter` uses a dedicated extension point that lets you define the `AuthenticationChain`.

```
<component name="org.nuxeo.ecm.anonymous.activation">
   <require>org.nuxeo.ecm.platform.ui.web.auth.WebEngineConfig</require>
    <extension

target="org.nuxeo.ecm.platform.ui.web.auth.service.PluggableAuthenticationService"
      point="chain">
    <authenticationChain>
      <plugins>
        <plugin>BASIC_AUTH</plugin>
        <plugin>ANONYMOUS_AUTH</plugin>
        <plugin>FORM_AUTH</plugin>
      </plugins>
    </authenticationChain>
  </extension>
</component>
```

The `NuxeoAuthenticationFilter` will use this chain to trigger the login prompt. When authentication is needed, the Filter will first call the `handleRetrieveIdentity` method on all the plugins in the order of the authentication chain. Then, if the authentication could not be achieved, the filter will call the `handleLoginPrompt` method in the same order on all the plugins. The aim is to have as much automatic authentications as possible. That's why all the manual authentications (those needing a prompt) are done in a second round.

Some authentication plugins may choose to trigger or not the `LoginPrompt` depending on the situation. For example: the `BasicAuthentication` plugin generates the login prompt (an HTTP basic authentication which takes the form of a popup) only for specific URLs used by RSS feeds or Restlet calls. This allows the platform to be easily called by Restlets and RSS clients without bothering browser clients who are displayed web forms to authenticate.

Built-in Authentication Plugins

`NuxeoAuthenticationFilter` comes with two built-in authentication plugins:

- `FORM_AUTH`: Form based Authentication
  This is a standard form-based authentication. The current implementation lets you configure the name of the Login and Password fields and the name of the page used to display the login page.
- `BASIC_AUTH`: Basic HTTP Authentication

This plugin supports standard HTTP Basic Authentication. By default, this plugin only generates the authentication prompt on configured URLs.
There are also additional components that provide other Authentication plugins (see below).

Additional Authentication Plugins

Nuxeo provides a set of other authentication plugins that are not installed by default with the standard Nuxeo Platform setup. These plugins can be downloaded and installed separately.

Authentication Plugins and Nuxeo Services Specific Authentication

Some Nuxeo services, Drive and Automation for instance, may use a specific authentication. If you want to make them use another authentication mechanism than their default one, you need to overwrite specificChains of the corresponding service, such as the Automation specificChains.

CAS2 Authentication

This plugin implements a client for CAS SSO system (Central Authentication System). It can be configured to use a CAS proxy. It has been tested and reported to work with CAS V2.
It's easy to test this plugin by installing the JA-SIG Central Authentication Service Open Source CAS server.

To install the CAS2 authentication plugin:

1. Make sure there is a CAS server already setup and running.
2. Download the `nuxeo-platform-login-cas2` plugin.
3. Put it in `$TOMCAT_HOME/nxserver/bundles` or `$JBOSS_HOME/server/default/deploy/nuxeo.ear/bundles` and restart the server.
4. Configure the CAS2 descriptor.
5. Put CAS2 plugin into the authentication chain.

In order to configure CAS2 Auth, you need to create an XML configuration file into `nxserver/config`.
Here is a sample file named `CAS2-config.xml`.

```
<component name="org.nuxeo.ecm.platform.authenticator.cas2.sso.config">

  <require>org.nuxeo.ecm.platform.ui.web.auth.WebEngineConfig</require>
  <require>org.nuxeo.ecm.platform.login.Cas2SSO</require>

  <!-- Configure you CAS server parameters -->
  <extension
target="org.nuxeo.ecm.platform.ui.web.auth.service.PluggableAuthenticationService"
point="authenticators">
    <authenticationPlugin name="CAS2_AUTH">
      <loginModulePlugin>Trusting_LM</loginModulePlugin>
      <parameters>
        <parameter name="ticketKey">ticket</parameter>
        <parameter
name="appURL">http://127.0.0.1:8080/nuxeo/nxstartup.faces</parameter>
        <parameter name="serviceLoginURL">http://127.0.0.1:8080/cas/login</parameter>
        <parameter
name="serviceValidateURL">http://127.0.0.1:8080/cas/serviceValidate</parameter>
        <parameter name="serviceKey">service</parameter>
        <parameter name="logoutURL">http://127.0.0.1:8080/cas/logout</parameter>
      </parameters>
    </authenticationPlugin>
  </extension>

  <!-- Include CAS2 into authentication chain -->
  <extension
target="org.nuxeo.ecm.platform.ui.web.auth.service.PluggableAuthenticationService"
point="chain">
    <authenticationChain>
      <plugins>
        <plugin>BASIC_AUTH</plugin>
        <plugin>CAS2_AUTH</plugin>
      </plugins>
    </authenticationChain>
  </extension>
</component>
```

If while authenticating on the CAS server, you get the following exception in the logs, it simply means that the user JOEUSER does not exist in the Nuxeo directory and does not mean that the CAS process is not working.

```
ERROR \[org.nuxeo.ecm.platform.login.NuxeoLoginModule\] createIdentity failed
  javax.security.auth.login.LoginException: principal JOEUSER does not exist
    at
org.nuxeo.ecm.platform.login.NuxeoLoginModule.createIdentity(NuxeoLoginModule.java:304
)
    at
org.nuxeo.ecm.platform.login.NuxeoLoginModule.validateUserIdentity(NuxeoLoginModule.ja
va:362)
    at
org.nuxeo.ecm.platform.login.NuxeoLoginModule.getPrincipal(NuxeoLoginModule.java:216)
    at org.nuxeo.ecm.platform.login.NuxeoLoginModule.login(NuxeoLoginModule.java:271)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
    at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
    at java.lang.reflect.Method.invoke(Method.java:585)
    at javax.security.auth.login.LoginContext.invoke(LoginContext.java:769)
    at javax.security.auth.login.LoginContext.access$000(LoginContext.java:186)
    at javax.security.auth.login.LoginContext$4.run(LoginContext.java:683)
    at java.security.AccessController.doPrivileged(Native Method)
    at javax.security.auth.login.LoginContext.invokePriv(LoginContext.java:680)
    at javax.security.auth.login.LoginContext.login(LoginContext.java:579)
    at
org.nuxeo.ecm.platform.ui.web.auth.NuxeoAuthenticationFilter.doAuthenticate(NuxeoAuthe
nticationFilter.java:205)
```

PROXY_AUTH: Proxy Based Authentication

This plugin assumes Nuxeo is behind a authenticating reverse proxy that transmit user identity using HTTP headers. For instance, you will configure this plugin if an Apache reverse proxy using client certificates does the authentication or for SSO system - example Central Authentication System V2.

To install this authentication plugin:

1. Download the `nuxeo-platform-login-mod_sso` plugin.
2. Put it in `$TOMCAT_HOME/nxserver/bundles/` or `$JBOSS_HOME/server/default/deploy/nuxeo.ear/bundles` and restart the server.
3. Configure the plugin via an XML descriptor.
4. Put the plugin into the authentication chain.

In order to configure this plugin, you need to create an XML configuration file into `nxserver/config`.
Here is a sample file named `proxy-auth-config.xml`:

```
<component name="org.nuxeo.ecm.platform.authenticator.mod.sso.config">

  <require>org.nuxeo.ecm.platform.ui.web.auth.WebEngineConfig</require>
  <require>org.nuxeo.ecm.platform.login.Proxy</require>

  <extension
target="org.nuxeo.ecm.platform.ui.web.auth.service.PluggableAuthenticationService"
point="authenticators">
    <authenticationPlugin name="PROXY_AUTH">
      <loginModulePlugin>Trusting_LM</loginModulePlugin>
      <parameters>
        <\!-\- configure here the name of the http header that is used to retrieve
user identity -->
        <parameter name="ssoHeaderName">remote_user</parameter>
      </parameters>
    </authenticationPlugin>
  </extension>

  <\!-\- Include Proxy Auth into authentication chain -->
  <extension
target="org.nuxeo.ecm.platform.ui.web.auth.service.PluggableAuthenticationService"
point="chain">
    <authenticationChain>
      <plugins>
        <\!-\-  Keep basic Auth at top of Auth chain to support RSS access via
BasicAuth -->
        <plugin>BASIC_AUTH</plugin>
        <plugin>PROXY_AUTH</plugin>
      </plugins>
    </authenticationChain>
  </extension>
</component>
```

NTLM_AUTH: NTLM and IE Challenge/Response Authentication

This plugin uses JCIFS to handle NTLM authentication.

This plugin was partially contributed by Nuxeo Platform users and has been reported to work by several users.

If you have troubles with latest version of IE on POST requests, please see JCIFS instructions on that:

```
http://jcifs.samba.org/src/docs/ntlmhttpauth.html#post
```

To install this authentication plugin:

1. Download the `nuxeo-platform-login-ntlm` plugin.
2. Put it in `$TOMCAT_HOME/nxserver/bundles` or `$JBOSS_HOME/server/default/deploy/nuxeo.ear/bundles` and restart the server.
3. Configure the plugin via an XML descriptor.
4. Put the plugin into the authentication chain.

In order to configure this plugin, you need to create an XML configuration file into `nxserver/config`.
Here is a sample file named `ntlm-auth-config.xml`.

```
<component name="org.nuxeo.ecm.platform.authenticator.ntlm.config">

  <require>org.nuxeo.ecm.platform.ui.web.auth.WebEngineConfig</require>
  <require>org.nuxeo.ecm.platform.login.NTLM</require>

  <extension
target="org.nuxeo.ecm.platform.ui.web.auth.service.PluggableAuthenticationService"
point="authenticators">
    <authenticationPlugin name="NTLM_AUTH">
      <loginModulePlugin>Trusting_LM</loginModulePlugin>
      <parameters>
        <\!-\- Add here parameters for you domain, please ee
[http://jcifs.samba.org/src/docs/ntlmhttpauth.html ];
        <parameter name="jcifs.http.domainController">MyControler</parameter>
        \-->
      </parameters>
    </authenticationPlugin>
  </extension>

  <\!-\- Include NTLM Auth into authentication chain -->
  <extension
target="org.nuxeo.ecm.platform.ui.web.auth.service.PluggableAuthenticationService"
point="chain">
    <authenticationChain>
      <plugins>
        <plugin>BASIC_AUTH</plugin>
        <plugin>NTLM_AUTH</plugin>
        <plugin>FORM_AUTH</plugin>
      </plugins>
    </authenticationChain>
  </extension>
</component>
```

PORTAL_AUTH: SSO Implementation for Portal Clients

This plugin provides a way to handle identity propagation between an external application and Nuxeo. It was coded in order to propagate user identify between a JSR168 portal and a Nuxeo server.

The goal is to let the external application (ex: the portal) call Nuxeo API *"on behalf"* of the interactive users. This ensures:

- that the app/portal will never display data that should not be visible to the user;
- that all actions done via the app/portal will still be logged in the Nuxeo Audit log with the correct information.

Portal_SSO is integrated in Nuxeo Java Automation client and well as in Nuxeo-HTTP-Client sample lib.

To install this authentication plugin:

1. Download the `nuxeo-platform-login-portal-sso` plugin.
2. Put it in `$TOMCAT_HOME/nxserver/bundles` or `$JBOSS_HOME/server/default/deploy/nuxeo.ear/bundles` and restart the server.
3. Configure the plugin via an XML descriptor.
4. Put the plugin into the authentication chain.

In order to configure this plugin, you need to create an XML configuration file into `nxserver/config`.
Here is a sample file named `portal-auth-config.xml`.

```
<component name="org.nuxeo.ecm.platform.authenticator.portal.sso.config">

  <require>org.nuxeo.ecm.platform.ui.web.auth.WebEngineConfig</require>
  <require>org.nuxeo.ecm.platform.login.Portal</require>

  <extension
target="org.nuxeo.ecm.platform.ui.web.auth.service.PluggableAuthenticationService"
point="authenticators">
    <authenticationPlugin name="PORTAL_AUTH">
      <loginModulePlugin>Trusting_LM</loginModulePlugin>
      <parameters>
        <\!-\- define here shared secret between the portal and Nuxeo server -->
        <parameter name="secret">nuxeo5secretkey</parameter>
        <parameter name="maxAge">3600</parameter>
      </parameters>
    </authenticationPlugin>
  </extension>

  <\!-\- Include Portal Auth into authentication chain -->
  <extension
target="org.nuxeo.ecm.platform.ui.web.auth.service.PluggableAuthenticationService"
point="chain">
    <authenticationChain>
      <plugins>
        <\!-\-  Keep basic Auth at top of Auth chain to support RSS access via
BasicAuth -->
        <plugin>BASIC_AUTH</plugin>
        <plugin>PORTAL_AUTH</plugin>
        <plugin>FORM_AUTH</plugin>
      </plugins>
    </authenticationChain>
  </extension>
</component>
```

ANONYMOUS_AUTH: Anonymous Authentication Plugin

This plugin provides anonymous authentication. Users are automatically logged as a configurable Anonymous user. This module also includes additional actions (to be able to login when already logged as Anonymous) and a dedicated Exception handling (to automatically redirect Anonymous users to login screen after a security error).

To activate this authentication plugin:

1. Put it in `$TOMCAT_HOME/nxserver/bundles` or `$JBOSS_HOME/server/default/deploy/nuxeo.ear/bundles` and restart the server.
2. Configure the plugin via an XML descriptor (define who the anonymous user will be).

In order to configure this plugin, you need to create an XML configuration file into `nxserver/config`.
Here is a sample file named `anonymous-auth-config.xml`.

```
<?xml version="1.0"?>
<component name="org.nuxeo.ecm.platform.login.anonymous.config">


  <!-- Add an Anonymous user -->
  <extension target="org.nuxeo.ecm.platform.usermanager.UserService"
point="userManager">
    <userManager>
      <users>
        <anonymousUser id="Guest">
          <property name="firstName">Guest</property>
          <property name="lastName">User</property>
        </anonymousUser>
      </users>
    </userManager>
  </extension>

</component>
```

Adding Custom LDAP Fields to the UI

To add a custom LDAP fields to the User interface you have to:

1. Create a custom schema based on nuxeo's user.xsd schema with custom fields related to the fields in your LDAP system.

schemas/myuser.xsd

```
<?xml version="1.0"?>
 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:nxs="http://www.nuxeo.org/ecm/schemas/myuser"
    targetNamespace="http://www.nuxeo.org/ecm/schemas/myuser">

  <xs:include schemaLocation="base.xsd" />

  <xs:element name="username" type="xs:string" />
  <xs:element name="password" type="xs:string" />
  <xs:element name="email" type="xs:string" />
  <xs:element name="firstName" type="xs:string" />
  <xs:element name="lastName" type="xs:string" />
  <xs:element name="company" type="xs:string" />
  <!-- your custom telephone field -->
  <xs:element name="telephone" type="xs:string" />

  <xs:element name="groups" type="nxs:stringList" />

 </xs:schema>
```

2. Add your schema via Nuxeo's extension system.

OSGI-INF/schema-contrib.xml

```
<?xml version="1.0"?>
 <component name="com.example.myproject.myuser.schema">
  <extension target="org.nuxeo.ecm.core.schema.TypeService" point="schema">
    <schema name="myuser" src="schemas/myuser.xsd" />
  </extension>
 </component>
```

3. Modify your LDAP configuration file in Nuxeo (`default-ldap-users-directory-bundle.xml`) to include:
   a. your custom schema,

default-ldap-users-directory-bundle.xml

```
<extension target="org.nuxeo.ecm.directory.ldap.LDAPDirectoryFactory"
    point="directories">

    <directory name="userDirectory">
      <server>default</server>
      <!-- association between your custom schema and the directory -->
      <schema>myuser</schema>
```

   b. mapping between your schema and your LDAP fields.

default-ldap-users-directory-bundle.xml (continued)

```
<fieldMapping name="username">uid</fieldMapping>
      <fieldMapping name="password">userPassword</fieldMapping>
      <fieldMapping name="firstName">givenName</fieldMapping>
      <fieldMapping name="lastName">sn</fieldMapping>
      <fieldMapping name="company">o</fieldMapping>
      <fieldMapping name="email">mail</fieldMapping>
      <fieldMapping name="telephone">telephoneNumber</fieldMapping>
```

4. Modify the UI.
   a. Add your custom widget to the layout.

default-ldap-users-directory-bundle.xml(continued)

```
<extension target="org.nuxeo.ecm.platform.forms.layout.WebLayoutManager"
    point="layouts">

    <layout name="user">
      <templates>
        <template
mode="any">/layouts/layout_default_template.xhtml</template>
      </templates>
      <rows>
        <row>
          <widget>username</widget>
        </row>
      <row>
      <!-- your custom telephone widget-->
          <widget>telephone</widget>
        </row>
```

b. Define a new widget for your custom field to be used in the layout above.

default-ldap-users-directory-bundle.xml(continued)

```
<widget name="telephone" type="text">
 <labels>
 <label mode="any">telephone</label>
 </labels>
 <translated>true</translated>
 <fields>
 <field schema="myuser">telephone</field>
 </fields>
 <widgetModes>
 <mode value="editPassword">hidden</mode>
 </widgetModes>
 <properties widgetMode="edit">
 <property name="required">true</property>
 <property name="styleClass">dataInputText</property>
 </properties>
 </widget>
```

User Management

In Nuxeo Platform, the concept of a user is needed for two main reasons:

- users are needed for authentication and authorization to work,
- users have associated information that can be displayed, for instance to display someone's full name or email address.

An abstraction, the `UserManager`, centralizes the way a Nuxeo Platform application deals with users (and groups of users). The UserManager is queried by the platform's `LoginModule` when someone attempts to authenticate against the framework. It is also queried whenever someone wants the last

name or email of a user for instance, or to get all users having "Bob" as their first name.

Users and Groups Configuration

The data about users (login, password, name, personal information, etc.) and the groups they belong to (simple members, or any application-related group) are managed through the Directory abstraction. This means that:

- LDAP can store users and groups,
- SQL can store users and groups,
- LDAP can store user and SQL can store groups,
- Nuxeo can aggregate two LDAP servers for user storage and SQL can store groups,
- a part of user can be stored into an LDAP server and into SQL, and SQL can store groups,
- ...

On this page

You understood almost of any configuration is possible... The application doesn't see the difference as long as the connectors are configured properly.

To configure your user management, you basically need to follow these steps:

1. define the schema that describes fields stored into a user. This is exactly the same extension point you will use for document type;
2. define a user manager. The default one will manage user stored into a directory. But you can implement your specific user manager, if you need;
3. If you use the default user manager:
   a. directory definition: As you describe a vocabulary, you will describe the user directory. Instead of using the vocabulary schema, you will use one that defines a username, a first name, ...
   b. configure the Default User Manager to bind it to the directory described above and some search configuration.
4. define how to display the User Profile. Most of the time you do not have to do that.

If you want to declare fields that are not stored into your directory, but that must be locally stored in Nuxeo, this is possible.
Nuxeo Platform defines a User Profile Service that will manage these type of field. These fields will be stored into a hidden Nuxeo Document into the personal workspace of each user.
You will benefit from all the UI infrastructure for these specific fields (Layout Service, Widget Service, ...).

Example of User Manager Configuration

Schema Definition

Here, will be defined a typical example of configuration.

Nuxeo Platform defines a default schema. Most of the time, this schema works for our users:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:nxs="http://www.nuxeo.org/ecm/schemas/user"
    targetNamespace="http://www.nuxeo.org/ecm/schemas/user">

  <xs:include schemaLocation="base.xsd" />

  <xs:element name="username" type="xs:string" />
  <xs:element name="password" type="xs:string" />
  <xs:element name="email" type="xs:string" />
  <xs:element name="firstName" type="xs:string" />
  <xs:element name="lastName" type="xs:string" />
  <xs:element name="company" type="xs:string" />

  <xs:element name="petName" type="xs:string" />

  <!-- inverse reference -->
  <xs:element name="groups" type="nxs:stringList" />

</xs:schema>
```

This schema is registered in an extension point:

```
<extension target="org.nuxeo.ecm.core.schema.TypeService" point="schema">
  <schema name="myuser" src="myuser.xsd" />
</extension>
```

You can choose to define your own schema by adding some field ore remove ones, if you need.

The schema for groups works the same way:

```
<?xml version="1.0"?>

<xs:schema targetNamespace="http://www.nuxeo.org/ecm/schemas/group"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:nxs="http://www.nuxeo.org/ecm/schemas/group">

  <xs:include schemaLocation="base.xsd" />

  <xs:element name="groupname" type="xs:string" />
  <xs:element name="grouplabel" type="xs:string" />
  <xs:element name="description" type="xs:string" />

  <!-- references -->
  <xs:element name="members" type="nxs:stringList" />
  <xs:element name="subGroups" type="nxs:stringList" />

  <!-- inverse reference -->
  <xs:element name="parentGroups" type="nxs:stringList" />

</xs:schema>
```

And the contribution to register this schema is:

```
<extension target="org.nuxeo.ecm.core.schema.TypeService" point="schema">
  <schema name="group" src="directoryschema/group.xsd"/>
</extension>
```

If you want to override these schema, don't forget the require item in your contribution and the override parameter in your schema definition (see the schema documentation warn).

User Manager Definition

You can override the Nuxeo default User Manager. You can look the UserManager definition into explorer.nuxeo.com. But most of the time the default User Manager binded to a directory is enough for our users.

Directory Definition

SQL Case

So the user and group schema can now be used when we define a new directory, called "MyUserDirectory".

SQL directory sample definition

```
<extension target="org.nuxeo.ecm.directory.sql.SQLDirectoryFactory"
point="directories">
  <directory name="MyUserDirectory">

    <schema>myuser</schema>
    <idField>username</idField>
    <passwordField>password</passwordField>

    <dataSource>java:/nxsqldirectory</dataSource>
    <table>myusers</table>
    <dataFile>myusers.csv</dataFile>
    <createTablePolicy>on_missing_columns</createTablePolicy>

    <references>
      <inverseReference field="groups" directory="groupDirectory"
        dualReferenceField="members" />
    </references>

  </directory>
</extension>
```

And we can provide a file, "myusers.csv", which will be used to populate the table if it is missing:

```
username, password, firstName, lastName, company, email, petName
bob,bobSecret,Bob,Doe,ACME,bob@example.com,Lassie
If instead we had used an LDAP directory, the configuration would look like:
```

LDAP Case

In the case of your server is a LDAP server, here is an example of directory definition.

First, define the LDAP Server that will be used as reference into the LDAP directory definition.

LDAP server sample definition

```
<extension target="org.nuxeo.ecm.directory.ldap.LDAPDirectoryFactory" point="servers">
  <server name="default">
    <ldapUrl>ldap://localhost:389</ldapUrl>
    <bindDn>cn=manager,dc=example,dc=com</bindDn>
    <bindPassword>secret</bindPassword>
  </server>
</extension>
```

LDAP directory sample definition

```
<extension target="org.nuxeo.ecm.directory.ldap.LDAPDirectoryFactory"
point="directories">
  <directory name="MyUserDirectory">

    <schema>myuser</schema>
    <idField>username</idField>
    <passwordField>password</passwordField>

    <server>default</server>
    <searchBaseDn>ou=people,dc=example,dc=com</searchBaseDn>
    <searchClass>inetOrgPerson</searchClass>
    <searchScope>subtree</searchScope>

    <fieldMapping name="username">uid</fieldMapping>
    <fieldMapping name="password">userPassword</fieldMapping>
    <fieldMapping name="email">mail</fieldMapping>
    <fieldMapping name="firstName">givenName</fieldMapping>
    <fieldMapping name="lastName">sn</fieldMapping>
    <fieldMapping name="company">o</fieldMapping>

    <references>
      <inverseReference field="groups" directory="groupDirectory"
        dualReferenceField="members" />
    </references>

  </directory>
</extension>
```

Multi-Directories

If you need to mix multiple directories, see the MultiDirectoryFactory.

Multi-directories sample definition

```
<extension point="directories"
target="org.nuxeo.ecm.directory.multi.MultiDirectoryFactory">
  <directory name="userDirectory">
    <schema>user</schema>
    <idField>username</idField>
    <passwordField>password</passwordField>
    <source name="userLDAPSource">
      <subDirectory name="userLDAPDirectory"/>
      <optional>true</optional>
    </source>
    <source creation="true" name="userSQLSource">
      <subDirectory name="userSQLDirectory"/>
    </source>
  </directory>
</extension>
```

UserManager

Simple Case

We can now tell the UserManager that this directory should be the one to use when dealing with users:

```
<extension target="org.nuxeo.ecm.platform.usermanager.UserService"
point="userManager">
  <userManager>
    <users>
      <directory>MyUserDirectory</directory>
      <emailField>email</emailField>
      <searchFields append="true">
        <searchField>username</searchField>
        <searchField>firstName</searchField>
        <searchField>lastName</searchField>
        <searchField>myfield</searchField>
      </searchFields>
    </users>
  </userManager>
</extension>
```

This configuration also sets the email field and search fields that have to be queried when searching for users. It can be completed to set the anonymous user, add virtual users, or set the group directory properties.

Configuring the User Manager with Anonymous User and Other Virtual Users

Virtual users can be added for authentication. Properties are used to create the appropriate model as if user was retrieved from the user directory. This is a convenient way to add custom users to the application when the user directory (using LDAP for instance) cannot be modified. Virtual users with the "administrators" group will have the same rights as the default administrator.

The anonymous user represents a special kind of virtual user, used to represent users that do not need to log in the application. This feature is used in conjunction with the anonymous plugin.

```
<extension target="org.nuxeo.ecm.platform.usermanager.UserService"
point="userManager">
  <userManager>
    <users>
```

```
    <directory>MyUserDirectory</directory>
    <emailField>email</emailField>
    <searchFields append="true">
      <searchField>username</searchField>
      <searchField>firstName</searchField>
      <searchField>lastName</searchField>
      <searchField>myfield</searchField>
    </searchFields>
    <listingMode>tabbed</listingMode>

    <anonymousUser id="Anonymous">
      <property name="firstName">Anonymous</property>
      <property name="lastName">User</property>
    </anonymousUser>
    <virtualUser id="MyCustomAdministrator" searchable="false">
      <password>secret</password>
      <property name="firstName">My Custom</property>
      <property name="lastName">Administrator</property>
      <group>administrators</group>
    </virtualUser>
    <virtualUser id="MyCustomMember" searchable="false">
      <password>secret</password>
      <property name="firstName">My Custom</property>
      <property name="lastName">Member</property>
      <group>members</group>
      <group>othergroup</group>
      <propertyList name="listprop">
        <value>item1</value>
        <value>item2</value>
      </propertyList>
    </virtualUser>
    <virtualUser id="ExistingVirtualUser" remove="true" />

  </users>

  <defaultAdministratorId>Administrator</defaultAdministratorId>
  <!-- available tags since 5.3.1 -->
  <administratorsGroup>myAdmins</administratorsGroup>
  <administratorsGroup>myOtherAdmins</administratorsGroup>
  <disableDefaultAdministratorsGroup>
    false
  </disableDefaultAdministratorsGroup>
  <!-- end of available tags since 5.3.1 -->

  <userSortField>lastName</userSortField>
  <userPasswordPattern>^[a-zA-Z0-9]{5,}$</userPasswordPattern>

  <groups>
    <directory>somegroupdir</directory>
    <membersField>members</membersField>
    <subGroupsField>subgroups</subGroupsField>
    <parentGroupsField>parentgroup</parentGroupsField>
    <listingMode>search_only</listingMode>
  </groups>
  <defaultGroup>members</defaultGroup>
  <groupSortField>groupname</groupSortField>
```

```
  </userManager>
</extension>
```

The default administrator ID can be set either to an existing or virtual user. This user will be virtually member of all the groups declared as administrators (by default, the group named "administrators" is used).

New administrators groups can be added using the "administratorsGroup" tag. Several groups can be defined, adding as many tags as needed. The default group named "administrators" can be disabled by setting the `disableDefaultAdministratorsGroup` to "true" (default is to false): only new defined administrators groups will then be taken into account.

Disabling the default "administrators" group should be done after setting up custom rights in the repository, as this group is usually defined as the group of users who have all permissions at the root of the repository. Administrators groups will have access to vocabulary management, theme editor,... They are also added local rights when blocking permissions to avoid lockups.

The group directory can also be configured to define the groups hierarchy and the contained users. This configuration has to match the user directory inverse references.

Every authenticated user will be placed in the configured default group. This group does not need to exist in the backing group directory, nor does any other group listed in virtual users configuration.

User and Group Display

The default users and groups management pages use some layouts for display. If you're using custom schema and would like to display your new fields, or would like to change the default display, you can redefine the layouts named "user" and "group" by contributing new layouts with these names.

Do not forget to put `<require>org.nuxeo.ecm.platform.forms.layouts.webapp</require>` on your layout contribution to ensure default layouts are overridden.

User Layout Definition

```
<?xml version="1.0"?>
<component name="org.nuxeo.ecm.platform.forms.layouts.usersAndGroups">
  <extension target="org.nuxeo.ecm.platform.forms.layout.WebLayoutManager"
point="layouts">
    <layout name="user">
      <templates>
        <template mode="any">/layouts/layout_default_template.xhtml</template>
      </templates>
      <rows>
        <row>
          <widget>username</widget>
        </row>
        <row>
          <widget>firstname</widget>
        </row>
        <row>
          <widget>lastname</widget>
        </row>
        <row>
          <widget>company</widget>
        </row>
        <row>
          <widget>email</widget>
        </row>
        <row>
          <widget>firstPassword</widget>
        </row>
        <row>
          <widget>secondPassword</widget>
```

```
    </row>
    <row>
      <widget>passwordMatcher</widget>
    </row>
    <row>
      <widget>groups</widget>
    </row>
  </rows>
  <widget name="username" type="text">
    <labels>
      <label mode="any">username</label>
    </labels>
    <translated>true</translated>
    <fields>
      <field schema="user">username</field>
    </fields>
    <widgetModes>
      <mode value="create">edit</mode>
      <mode value="editPassword">hidden</mode>
      <mode value="any">view</mode>
    </widgetModes>
    <properties widgetMode="edit">
      <property name="required">true</property>
      <property name="styleClass">dataInputText</property>
      <property name="validator">
        #{userManagementActions.validateUserName}
      </property>
    </properties>
  </widget>
  <widget name="firstname" type="text">
    <labels>
      <label mode="any">firstName</label>
    </labels>
    <translated>true</translated>
    <fields>
      <field schema="user">firstName</field>
    </fields>
    <widgetModes>
      <mode value="editPassword">hidden</mode>
    </widgetModes>
    <properties widgetMode="edit">
      <property name="styleClass">dataInputText</property>
    </properties>
  </widget>
  <widget name="lastname" type="text">
    <labels>
      <label mode="any">lastName</label>
    </labels>
    <translated>true</translated>
    <fields>
      <field schema="user">lastName</field>
    </fields>
    <widgetModes>
      <mode value="editPassword">hidden</mode>
    </widgetModes>
    <properties widgetMode="edit">
      <property name="styleClass">dataInputText</property>
    </properties>
  </widget>
```

```
<widget name="company" type="text">
  <labels>
    <label mode="any">company</label>
  </labels>
  <translated>true</translated>
  <fields>
    <field schema="user">company</field>
  </fields>
  <widgetModes>
    <mode value="editPassword">hidden</mode>
  </widgetModes>
  <properties widgetMode="edit">
    <property name="styleClass">dataInputText</property>
  </properties>
</widget>
<widget name="email" type="text">
  <labels>
    <label mode="any">email</label>
  </labels>
  <translated>true</translated>
  <fields>
    <field schema="user">email</field>
  </fields>
  <widgetModes>
    <mode value="editPassword">hidden</mode>
  </widgetModes>
  <properties widgetMode="edit">
    <property name="required">true</property>
    <property name="styleClass">dataInputText</property>
  </properties>
</widget>
<widget name="firstPassword" type="secret">
  <labels>
    <label mode="any">password</label>
  </labels>
  <translated>true</translated>
  <fields>
    <field schema="user">password</field>
  </fields>
  <widgetModes>
    <mode value="create">edit</mode>
    <mode value="editPassword">edit</mode>
    <mode value="any">hidden</mode>
  </widgetModes>
  <properties widgetMode="edit">
    <property name="required">true</property>
    <property name="styleClass">dataInputText</property>
  </properties>
</widget>
<widget name="secondPassword" type="secret">
  <labels>
    <label mode="any">password.verify</label>
  </labels>
  <translated>true</translated>
  <widgetModes>
    <mode value="create">edit</mode>
    <mode value="editPassword">edit</mode>
    <mode value="any">hidden</mode>
  </widgetModes>
```

```
      <properties widgetMode="edit">
        <property name="required">true</property>
        <property name="styleClass">dataInputText</property>
      </properties>
    </widget>
    <widget name="passwordMatcher" type="template">
      <labels>
        <label mode="any"></label>
      </labels>
      <translated>true</translated>
      <widgetModes>
        <mode value="create">edit</mode>
        <mode value="editPassword">edit</mode>
        <mode value="any">hidden</mode>
      </widgetModes>
      <properties widgetMode="edit">
        <!-- XXX: depends on firstPassword and secondPassword widget names -->
        <property name="template">
          /widgets/user_password_validation_widget_template.xhtml
        </property>
      </properties>
    </widget>
    <widget name="groups" type="template">
      <labels>
        <label mode="any">label.userManager.userGroups</label>
      </labels>
      <translated>true</translated>
      <fields>
        <field schema="user">groups</field>
      </fields>
      <widgetModes>
        <mode value="edit">
          \#{nxu:test(currentUser.administrator, 'edit', 'view')}
        </mode>
        <mode value="editPassword">hidden</mode>
      </widgetModes>
      <properties widgetMode="any">
        <property name="template">
          /widgets/user_suggestion_widget_template.xhtml
        </property>
        <property name="userSuggestionSearchType">GROUP_TYPE</property>
      </properties>
    </widget>
  </layout>
  <layout name="group">
    <templates>
      <template mode="any">/layouts/layout_default_template.xhtml</template>
    </templates>
    <rows>
      <row>
        <widget>groupname</widget>
      </row>
      <row>
        <widget>members</widget>
      </row>
      <row>
        <widget>subgroups</widget>
      </row>
    </rows>
```

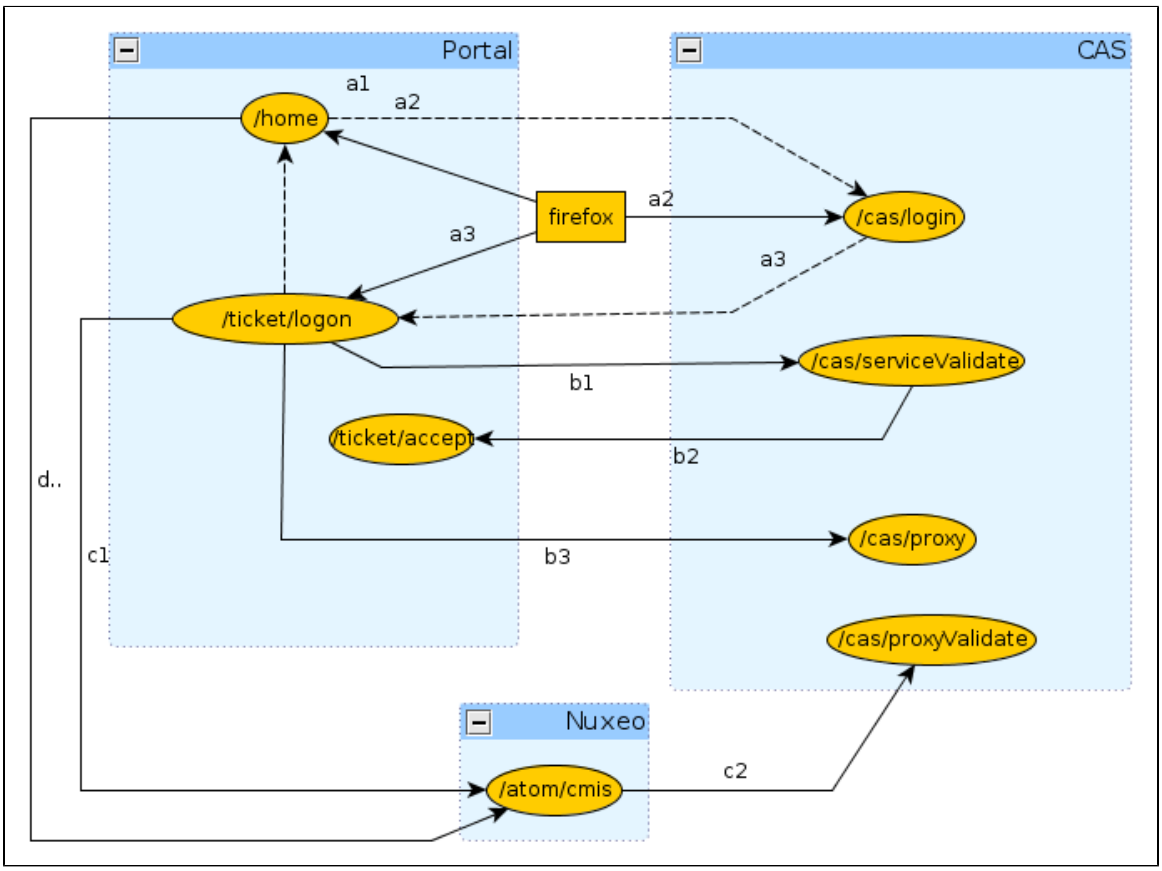
```
    <widget name="groupname" type="text">
      <labels>
        <label mode="any">label.groupManager.groupName</label>
      </labels>
      <translated>true</translated>
      <fields>
        <field schema="group">groupname</field>
      </fields>
      <widgetModes>
        <mode value="create">edit</mode>
        <mode value="any">hidden</mode>
      </widgetModes>
      <properties widgetMode="any">
        <property name="required">true</property>
        <property name="styleClass">dataInputText</property>
      </properties>
    </widget>
    <widget name="members" type="template">
      <labels>
        <label mode="any">label.groupManager.userMembers</label>
      </labels>
      <translated>true</translated>
      <fields>
        <field schema="group">members</field>
      </fields>
      <properties widgetMode="any">
        <property name="template">
          /widgets/user_suggestion_widget_template.xhtml
        </property>
        <property name="userSuggestionSearchType">USER_TYPE</property>
      </properties>
    </widget>
    <widget name="subgroups" type="template">
      <labels>
        <label mode="any">label.groupManager.groupMembers</label>
      </labels>
      <translated>true</translated>
      <fields>
        <field schema="group">subGroups</field>
      </fields>
      <properties widgetMode="any">
        <property name="template">
          /widgets/user_suggestion_widget_template.xhtml
        </property>
        <property name="userSuggestionSearchType">GROUP_TYPE</property>
      </properties>
    </widget>
  </layout>
```

```
  </extension>
</component>
```

Using CAS2 Authentication

A typical CAS use case would be the portal. In this n-tiers architecture, the identity is to be shared between the components.

The following diagram depicts the interactions between a client, a portal, a CAS server and Nuxeo for establishing the authentication context.



In this section

(a) Portal Service Ticket

The first phase is the portal authentication (a1).

```
GET /home
```

The client is redirected to the CAS server for entering its credentials (a2).

```
GET /cas/login?service=http://127.0.0.1:9090/ticket/validate
```

Once the credentials are entered, if they are valid, the CAS server generates a service ticket `ST`.
The client is redirected by the CAS server back onto the portal using the `service` URL (a3).

> In the same time, the CAS server generates a ticket granting and registers it client side using the cookie `CASTGC`. If the cookie is already present in the request headers, then the client is automatically redirected to the portal.

```
http://127.0.0.1:9090/ticket/validate?ticket=ST-81-rCbbm5oj9geCKjvhNCvJ-
cas
```

(b) Proxy Granting Ticket

In the second phase, the portal validates the service ticket and requests for a proxy granting ticket `PGT` (b1).

```
GET /cas/serviceValidate?ticket=ST-81-rCbbm5oj9geCKjvhNCvJ-cas&

service=http://127.0.0.1:9090/ticket/validate&pgtUrl=http://127.0.0.1:90
90/ticket/accept
```

If the ticket is valid, the CAS server invokes the `pgtUrl` callback with two parameters `pgtIou` and `pgtId` (b2).

```
GET /ticket/accept?pgtIou=PGTIOU-34-jJZH23r2wbKUqbc3dLFt-cas&
    pgtId=TGT-45-sSnfcQ7A0TXGsQR2gJONm74rObZ0qRQzhENJWTdZJG5rcGN2T5-cas
```

In case of success, the server responds to the portal with the following content

```
<cas:serviceResponse xmlns:cas='http://www.yale.edu/tp/cas'>
.<cas:authenticationSuccess>
..<cas:user>slacoin</cas:user>
..<cas:proxyGrantingTicket>PGTIOU-34-jJZH23r2wbKUqbc3dLFt-cas</cas:proxyGrantingTick
et>
.</cas:authenticationSuccess>
</cas:serviceResponse>
```

The `pgtIou` is used portal side for retrieving the accepted `PGT`.

(c) Nuxeo Proxy Ticket

In the third phase, the portal asks the CAS server for a new service ticket that will give him access to the Nuxeo server using the client identity (c1).

```
GET
/cas/proxy?pgt=TGT-45-sSnfcQ7A0TXGsQR2gJONm74rObZ0qRQzhENJWTdZJG5rcGN2T5
-cas&
    targetService=http://127.0.0.1:8080/nuxeo/atom/cmis
```

The CAS server generates a new ST and responds to the portal with the following content:

```
<cas:serviceResponse xmlns:cas='http://www.yale.edu/tp/cas'>
.<cas:proxySuccess>
..<cas:proxyTicket>ST-82-20eCHgCqvMCvnP6AmZmz-cas</cas:proxyTicket>
.</cas:proxySuccess>
</cas:serviceResponse>
```

Then the proxy ticket is used by the portal for login into Nuxeo (c2).

```
GET /nuxeo/atom/cmis?ticket=ST-82-20eCHgCqvMCvnP6AmZmz-cas
    &proxy=http://127.0.0.1:9090/ticket/accept
    &service=http:127.0.0.1:8080/nuxeo/atom/cmis
```

The Nuxeo server validates the ticket by invoking the portal server (c3).

```
GET
/cas/proxyValidate?ticket=ST-82-20eCHgCqvMCvnP6AmZmz-cas&service=http:12
7.0.0.1:8080/nuxeo/atom/cmis
```

If the ticket is valid, the CAS server sends the following response:

```
<cas:serviceResponse xmlns:cas='http://www.yale.edu/tp/cas'>
.<cas:authenticationSuccess>
..<cas:user>slacoin</cas>
..<cas:proxyGrantingTicket>PGTIOU-34-jJZH23r2wbKUqbc3dLFt-cas</cas:proxyGrantingTick
et>
..<cas:proxies>
...<cas:proxy>http://127.0.0.1:9090/ticket/accept</cas:proxy>
..</cas:proxies>
.</cas:authenticationSuccess>
</cas>
```

The Nuxeo server creates an HTTP session and sends the AtomPub response message.

```
<?xml version='1.0' encoding='UTF-8'?>
<app:service xmlns:app="http://www.w3.org/2007/app"
             xmlns:atom="http://www.w3.org/2005/Atom"
             xmlns:cmis="http://docs.oasis-open.org/ns/cmis/core/200908/"
             xmlns:cmisra="http://docs.oasis-open.org/ns/cmis/restatom/200908/">
  <app:workspace>...</app:workspace>
</app:service>
```

The portal saves the client context for being able to invoke Nuxeo using the same HTTP session.

(d) Invoking Nuxeo

The Nuxeo HTTP session id is retrieved from the portal session context and invoked.

```
GET /nuxeo/atom/cmis?repositoryId=default
```

CAS2 and Anonymous Authentication

CAS2 and anonymous authenticators have flows that can interfere with each others, creating some side effects like bad redirections.

To avoid that, the CAS2 plugin provides a replacement for the default Anonymous authenticator : basically this is a *"CAS aware Anonymous authenticator"*. You can see a sample configuration available in: https://github.com/nuxeo/nuxeo-platform-login/blob/release-5.8/nuxeo-platform-login-cas2/Sample/CAS2-Anonymous-bundle.xml.

But, basically, wanting to put together both CAS2 and Anonymous authentication means you have two types of users that will access Nuxeo. So, an alternate approach is to define two separated authentication chains, one for each type of user:

- One chain for authenticated users: using CAS2 and some other authentication method you may need;
- One chain for anonymous access.

In most of the case, each type of user will have access via a separated virtual host at reverse proxy level. You can use this so that:

- At reverse proxy level you add a header for tagging the type of request;
  ex: Anonymous requests will have the header `X-anonymous-access` set to "on";
- At nuxeo level you configure the chains depending on the header;
  - Default / main chain is the one using CAS2;
  - You define specific chain for requests having the `X-anonymous-access`.

Sample Xml contribution

```
<specificAuthenticationChain name="anonymous-access">
        <headers>
            <header name="X-anonymous-access">on</header>
        </headers>
        <allowedPlugins>
            <plugin>ANONYMOUS_AUTH</plugin>
        </allowedPlugins>
    </specificAuthenticationChain>
```

You can see specificChains extension point for more info.

REST API

Nuxeo REST API is available on a Nuxeo server at the following URL: `http://localhost:8080/nuxeo/api/v1/*`. This section describes all the mechanisms that are offered by our REST API as well as all the tools useful for using it (clients, format of queries, etc.).

You need to install the REST API module available:

- from the Marketplace,
- from the Update Center of your Nuxeo Platform instance,
- from the configuration wizard when you start your Nuxeo Platform instance for the first time.

Concepts

Nuxeo provides a complete API accessible via HTTP and also provides client SDKs in Java, JavaScript, Python, PHP, iOS and Android. This API is the best channel to use when you want to integrate remotely with the Nuxeo repository: Portals access, workflow engines, custom application JavaScript based, etc. This API has several endpoints:

- **Resources endpoints,** for doing CRUD on resources in a 100% REST style.
  Multiple resources are exposed via this API (See the reference documentation page for the existing REST URLs and JSON resources):
  - Documents (`/nuxeo/api/v1/id/{docId} or /nuxeo/api/v1/path/{path}`): to do CRUD on documents (including paginated search);
  - Users (`/nuxeo/api/v1/user/{userId}`): to do CRUD on users;
  - Groups (`/nuxeo/api/v1/group/{groupId}`): to do CRUD on groups;
  - Directories (`/nuxeo/api/v1/directory/{directoryId}`): to do CRUD on directories;
  - Automation (`/nuxeo/api/v1/automation/{Operation id}`): to call a "command", i.e. an operation or chain of operations deployed on the server. This is the main way of exposing the platform services remotely;
  - Workflow and Task endpoints are to be implemented soon!
- A **commands endpoint**, that exposes all the operations of the Automation module offering more than 100 commands for processing remotely the resources. The framework makes it very easy to add a new Java custom operation for completing the API if you miss something, and to chain operations server-side using Nuxeo Studio, so as to expose a coarse-grained API that fits your business

logic.

The Nuxeo REST API offers several nice additional features compared to a standard REST API:

- Possibility to call a command on a resource (see example below);
- Possibility to ask for more information when receiving the resources via some request headers, in order to optimize the number of requests you have to do (Ex: Receiving all the children of a document at the same time you receive a document, or receiving all the parents, or all the tasks, ...);
- Possibility to use various "adapters" that will "transform" the resources that are returned;
- Possibility to add new endpoints.

Quick Examples

A blog post has been published presenting all the concepts of the Resources endpoints with some nice examples. We provide here some sample ready-to-play cURL requests to try against the Nuxeo demo site (http://demo.nuxeo.com Administrator/Administrator).

1. Here is how to create a new File document type on the Nuxeo demo instance, right under the default domain (you can copy, paste and test):

```
curl -X POST -H "Content-Type: application/json" -u
Administrator:Administrator -d '{ "entity-type": "document", "name":"newDoc",
"type": "File","properties": { "dc:title": "Specifications", "dc:description":
"Created via a so cool and simple REST API", "common:icon": "/icons/file.gif",
"common:icon-expanded": null, "common:size": null}}'
http://demo.nuxeo.com/nuxeo/api/v1/path/default-domain
```

2. You can get the new resource doing a standard GET (actually the JSON object was already returned in previous response):

```
curl -X GET -u Administrator:Administrator
http://demo.nuxeo.com/nuxeo/api/v1/path/default-domain/newDoc
```

3. Now, "lock" this document we have just created by calling an Automation operation from command API on the document resource.

```
curl -X POST -H "Content-Type: application/json+nxrequest"   -u
Administrator:Administrator -d '{"params":{}}'
http://demo.nuxeo.com/nuxeo/api/v1/path/default-domain/newDoc/@op/Document.Loc
k
```

   Pay attention to the Content-Type that is specific when using the @op adapter.

   You can check the result of your request on the web app (http://demo.nuxeo.com/nuxeo/nxpath/default@view_domains, credentials: Administrator/Administrator).
4. You can also directly call an automation operation or chain, from the "Command endpoint". Here we return all the worskpaces of the demo.nuxeo.com instance:

```
curl -H 'Content-Type:application/json+nxrequest' -X POST -d
'{"params":{"query":"SELECT * FROM Document WHERE
ecm:primaryType=\"Workspace\""},"context":{}}'   -u
Administrator:Administrator
http://demo.nuxeo.com/nuxeo/api/v1/automation/Document.Query
```

Adapters

This API also has the concept of "adapter". An adapter is a URL segment that starts with "@" and that transforms the input resource so as to return another resource. For example, using `@blob` will return the file of a document (the one stored on the property given by the next URL segment), and chaining it to `@op` will call an operation (that takes a blob in input):

```
/nuxeo/api/v1/id/{docId}/@blob/file:content/@op/Blob.ToPDF
```

Pluggable Context

It is sometimes useful to optimize the number of requests you send to the server. For that reason we provide a mechanism for requesting more information on the answer, simply by specifying the context you want in the request header.

For example, it is some times useful to get the children of a document while requesting that document. Or its parents. Or its open related workflow tasks.

The blog post "How to Retrieve a Video Storyboard Through the REST API" gives an example of how to use this mechanism and the dedicated extension point:

```
<extension
target="org.nuxeo.ecm.automation.io.services.contributor.RestContributorService"
point="contributor"> ...</extension>
```

Files Upload and Batch Processing

The Platform provides facilities for uploading binaries under a given "batch id" on the server, and then to reference that batch id when posting a document resource, or for fetching it from a custom automation chain.

For instance if you need to create a file with some binary content, first you have to upload the file into the batchManager. It's a place on the system where you can upload temporary files to bind them later.

1. For that you have to generate yourself a `batchId`, which will identify the batch : let's say `mybatchid`.

   ```
   POST http://localhost:8080/nuxeo/site/automation/batch/upload
   X-Batch-Id: mybatchid
   X-File-Idx:0
   X-File-Name:myFile.zip
   ----------------------
   The content of the file
   ```

   Or with curl:

   ```
   curl -H "X-Batch-Id: mybatchid" -H "X-File-Idx:0" -H "X-File-Name:Sites.zip"
   -F file=@Sites.zip -u Administrator:Administrator
   http://localhost:8080/nuxeo/site/automation/batch/upload
   ```

2. You may verify the content of your batch with the following request.

   ```
   GET http://localhost:8080/nuxeo/site/automation/batch/files/mybatchid
   [{"name":"Sites.zip","size":115090}]
   ```

3. Next you have to create a document of type File and attach the Blob to it by using the specific syntax on the `file:content` property.

```
POST
http://localhost:8080/nuxeo/api/v1/path/default-domain/workspaces/myworkspace
{
  "entity-type": "document",
  "name":"myNewDoc",
  "type": "File",
  "properties" : {
    "dc:title":"My new doc",
    "file:content": {
      "upload-batch":"mybatchid",
      "upload-fileId":"0"
    }
  }
}
```

Or with curl:

```
curl -X POST -H "Content-Type: application/json" -u
Administrator:Administrator -d '{  "entity-type": "document",
"name":"myNewDoc", "type": "File",  "properties" : { "dc:title":"My new
doc","file:content": {"upload-batch":"mybatchid","upload-fileId":"0"}}}'
http://localhost:8080/nuxeo/api/v1/path/default-domain/workspaces/myworkspace
```

4. Finally you now can access the content of your file by pointing to the following resource:

```
GET
http://localhost:8080/nuxeo/api/v1/path/default-domain/workspaces/myworkspace/
myNewDoc/@blob/file:content
```

Available Client SDKs

We provide several client SDKs for making it even easier to integrate with the Nuxeo Platform.

- Java client,
- JavaScript client,
- iOS client,
- Android client,
- PHP client (partial implementation).

On this page:

Children pages:

Resources Endpoints

This draft section aims at providing an exhaustive list of resources, URLs and adapters provided by Nuxeo REST API.

The REST API extensive documentation will soon be available on explorer.nuxeo.org, stay tuned!

Document Resources

Getting a Document

There are two ways to access a document resource:

- by its path:

```
GET /nuxeo/site/api/v1/path/{pathOfTheDoc}
```

- by its id:

```
GET /nuxeo/site/api/v1/id/{idOfTheDoc}
```

In any case, it will return an application/JSON+nxentity response with the following content:

```
{
        "entity-type": "document",
        "repository": "default",
        "uid": "37b1502b-26ff-430f-9f20-4bd0d803191e",
        "path": "/default-domain",
        "type": "Domain",
        "state": "project",
        "versionLabel": "",
        "title": "Default domain",
        "lastModified": "2013-07-20T05:58:58.98Z",
        "facets": [
            "SuperSpace",
            "Folderish"
        ],
        "changeToken": "1374299938988",
        "contextParameters": {}

    }
```

As with Automation, if you need some document properties, you have to add the list of schemas in a dedicated header `X-NXDocumentProperties` . This way you get another property in the entity called properties:

```
...
    "lastModified": "2013-07-20T05:58:58.98Z",
    "properties": {
        "common:icon": "/icons/domain.gif",
        "common:icon-expanded": null,
        "common:size": null
    },
    "facets": [
        "SuperSpace",
        "Folderish"
    ],
...
```

In this section

Updating a Document

To update the document it's very simple. You have to make a PUT on the document resource and pass a reduced version of the content of the entity type as the data. You can also pass the full set, but it's not mandatory.

```
PUT /nuxeo/site/api/v1/id/{idOfTheDoc}
    {
        "entity-type": "document",
        "repository": "default",
        "uid": "37b1502b-26ff-430f-9f20-4bd0d803191e",
        "properties": {
            "dc:title": "The new title",
            "dc:description": "Updated via a so cool and simple REST API"
            "common:icon": "/icons/file.gif",
            "common:icon-expanded": null,
            "common:size": null
        }
    }
```

Creating a Document

To create a new document under the current resource, you have to send a POST request with the following data:

```
POST /nuxeo/site/api/v1/id/{idOfParentDoc}
{
    "entity-type": "document",
    "name":"newDoc",
    "type": "File",
    "properties": {
        "dc:title": "The new document",
        "dc:description": "Created via a so cool and simple REST API"
        "common:icon": "/icons/file.gif",
        "common:icon-expanded": null,
        "common:size": null
    }
}
```

In this case, the id of the document is the one for the parent document, and the `name` property in the entity stands for the name of the newly

created document. You don't have to specify a UID since the session will create one for you. It will be returned in the response.

Deleting a Document

Finally, in order to delete a document, you just have to send a DELETE request to the document resource.

```
DELETE /nuxeo/site/api/v1/id/{idOfTheDoc}
```

List Adapters

Children of a Given Document

Given a document resource, it's very simple to get its children:

```
GET
/nuxeo/site/api/v1/path/{pathOfTheDoc}/@children?page=0&pagesize=20&maxResults=100
```

This returns a list of documents with the documents entity type:

```
{
    "entity-type": "documents",
    "isPaginable": true,
    "totalSize": 3,
    "pageIndex": 0,
    "pageSize": 50,
    "pageCount": 1,
    "entries": [
      {
        "entity-type": "document",
        "repository": "default",
        "uid": "afb373f1-08ed-4228-bfe8-9f93131f8c84",
        "path": "/default-domain/sections",
        "type": "SectionRoot",
            ...
        "contextParameters": {
            "documentURL":
"/nuxeo/site/api/v1/id/afb373f1-08ed-4228-bfe8-9f93131f8c84"
        }
       },
       ...
    ]
}
```

Query parameters are not mandatory and are by default:

- `page`: 0
- `pageSize`: 50
- `maxResult`: nolimit

> Notice that in the response, for each document, you have a `documentURL` property that points to the id API endpoint.

Searching Documents

It is possible to search for documents in several ways.

```
GET /nuxeo/site/api/v1/path/{pathOfTheDoc}/@search?fullText=nuxeo&orderBy=dc:title
```

This returns the list of documents under the pointed resource that fulfill the full-text search for the "nuxeo" term. The same query parameters as in the `@children` adapter apply plus the `orderBy` one. If the pointed resource is not a Folder, then the search is issued from the parent document.

You can also make some direct NXQL query with this endpoint, like this:

```
GET /nuxeo/site/api/v1/path/{pathOfTheDoc}/@search?query=SELECT * FROM File
```

It will return a paged result set.

Using Page Provider Adapters

If you have defined some page adapters that rely on only one parameter which is the id of the document, then you can use it with the page adapter endpoint:

```
GET /nuxeo/site/api/v1/path/{pathOfTheDoc}/@pp/{myPageProvider}
```

Same query parameters apply for pagination as for the `@children` adapter.

As it would be great to not only bind the id of a document, but also the document itself as a parameter of the page provider (using EL to define parameters), this API is then subject to change during the 5.7.x series.

Automation Chains

On a Document

As we have a way to point to documents with resources, we provide a way to run an automation chain on them. You can use any chain that takes a document as input. For instance to call that operation `myOperation` on a document, just send an POST request like that:

```
POST /nuxeo/site/api/v1/path/{pathOfTheDoc}/@op/{myOperation}
{
  params: {
    opParam: "value"
  }
}
```

The response will depend on the result of the automation chain.

If you want to run a chain, just use the same endpoint and prefix the chain name by `Chain.` , for instance:

```
POST /nuxeo/site/api/v1/path/{pathOfTheDoc}/@op/Chain.{myChain}
{
  params: {
    chainParam: "value"
  }
}
```

On a List of Documents

We also have a way to point to lists of documents thru the use of list adapters. This way, we are also able to run operations or chains on lists of documents. For instance:

```
POST /nuxeo/site/api/v1/path/{pathOfTheFolder}/@children/@op/Chain.myChain
{
  params: {
 chainParam: "value"
  }
}
```

This way we run the `myChain` operation on the children of the given folder.

> ⚠ Pay attention to the fact that document list adapters are paged. That means that the chain will run on all document of the current page.

Business Object Adapters

Since 5.7.2, you can use Business objects with Automation. They are also bound on the REST API and you just have to use the same semantics than for document resources.

Getting a Business Object

```
GET /nuxeo/site/api/v1/path/{pathOfTheDoc}/@bo/BusinessBeanAdapter
```

will return

```
{
    entity-type: "BusinessBeanAdapter"
    id: "37b1502b-26ff-430f-9f20-4bd0d803191e",
    "type": "Domain",
    "title":"Default domain"
    "description":""
}
```

Updating a Business Object

To update a business object, you just have to send a PUT request one the business object resource with its content data like this:

```
PUT /nuxeo/site/api/v1/path/{pathOfTheDoc}/@bo/BusinessBeanAdapter
{
 "entity-type: "BusinessBeanAdapter"
 "value": {
  id: "37b1502b-26ff-430f-9f20-4bd0d803191e",
     "type": "Domain",
     "title":"Default domain"
     "description":"My new description"
  }
}
```

Creating a Business Object

And then to create a business object, you have to issue a POST on the object resource plus the name of the newly created document, like this:

```
POST /nuxeo/site/api/v1/path/{pathOfTheDoc}/@bo/BusinessBeanAdapter/{newName}
{
 entity-type: "BusinessBeanAdapter",
 value: {
   "type": "Note",
   "title":"A sample note",
   "description":"",
   "note":"The content of my note"
 }
}
```

Related documentation

- Contributing a New Endpoint
- REST API
- Command Endpoint

Contributing a New Endpoint

It is possible to contribute a new endpoint to the API creating new web objects.

Thanks to WebEngine and its pluggability, we provided a way to add other root endpoints that benefit from the same infrastructure. It's just a matter of providing a web object in another bundle, like the following one.

```
@WebObject(type="productRoot")
public class ProductRoot {

    @GET
    public List getProducts() throws ClientException {
        ProductService ps = Framework.getLocalService(ProductService.class);
        return ps.getAllProducts(getContext().getCoreSession());
    }

    @Path("{productId}")
    public Object getProduct(@PathParam("productId") String productId) {
       return newObject("product", productId);
    }

}
```

This will expose two new APIs:

- `GET /nuxeo/api/v1/product`
- `GET /nuxeo/api/v1/product/{productId}`

Since Document Adapters are used as return types, the API will automatically benefit from the integrated adapter serializations (readers and writers). That means that it is very easy to build your own API, and that you inherit all the other pluggability mechanisms.

More Information on Document Resources Endpoints

The up-to-date and complete REST documentation is available on the Resouces Endpoints parent page. Here, we provide more explanations so that you get how to use it.

Updating a Document

To update the document it's very simple. You have to make a PUT on the document resource and pass a reduced version of the entity type content as the data. You can also pass the full set, but it's not mandatory.

```
PUT /nuxeo/site/api/v1/id/{idOfTheDoc}
    {
        "entity-type": "document",
        "repository": "default",
        "uid": "37b1502b-26ff-430f-9f20-4bd0d803191e",
        "properties": {
            "dc:title": "The new title",
            "dc:description": "Updated via a so cool and simple REST API"
            "common:icon": "/icons/file.gif",
            "common:icon-expanded": null,
            "common:size": null
        }
    }
```

Creating a Document

To create a new document under the current resource, you have to send a POST request with the following data:

```
POST /nuxeo/site/api/v1/id/{idOfParentDoc}
{
    "entity-type": "document",
    "name":"newDoc",
    "type": "File",
    "properties": {
        "dc:title": "The new document",
        "dc:description": "Created via a so cool and simple REST API"
        "common:icon": "/icons/file.gif",
        "common:icon-expanded": null,
        "common:size": null
    }
}
```

In this case, the id of the document is the parent document's id, and the `name` property in the entity stands for the name of the newly created document. You don't have to specify a UID since the session will create one for you. It will be returned in the response.

Deleting a Document

Finally, in order to delete a document, you just have to send a DELETE request to the document resource.

```
DELETE /nuxeo/site/api/v1/id/{idOfTheDoc}
```

List Adapters

Children of a Given Document

Given a document resource, it's very simple to get its children:

```
GET /nuxeo/site/api/v1/path/{pathOfTheDoc}/@children?page=0&pagesize=20&maxResults=100
```

This returns a list of documents with the documents entity type:

```
{
    "entity-type": "documents",
    "isPaginable": true,
    "totalSize": 3,
    "pageIndex": 0,
    "pageSize": 50,
    "pageCount": 1,
    "entries": [
      {
        "entity-type": "document",
        "repository": "default",
        "uid": "afb373f1-08ed-4228-bfe8-9f93131f8c84",
        "path": "/default-domain/sections",
        "type": "SectionRoot",
            ...
        "contextParameters": {
            "documentURL":
"/nuxeo/site/api/v1/id/afb373f1-08ed-4228-bfe8-9f93131f8c84"
        }
       },
       ...
    ]
}
```

Query parameters are not mandatory and are by default:

- `page`: 0
- `pageSize`: 50
- `maxResult`: nolimit

In the response, for each document, you have a `documentURL` property that points to the API endpoint's id.

Searching Documents

It is possible to search for documents in several ways.

```
GET /nuxeo/site/api/v1/path/{pathOfTheDoc}/@search?fullText=nuxeo&orderBy=dc:title
```

This returns the list of documents under the pointed resource that fulfills the full-text search for the "nuxeo" term. The same query parameters as in the `@children` adapter apply, plus the `orderBy` one. If the pointed resource is not a Folder, then the search is issued from the parent document.

You can also make some direct NXQL queries with this endpoint, like this:

```
GET /nuxeo/site/api/v1/path/{pathOfTheDoc}/@search?query=SELECT * FROM File
```

It will return a paged result set.

Using Page Provider Adapters

If you have defined some page adapters that rely on only one parameter which is the id of the document, then you can use it with the page adapter endpoint:

```
GET /nuxeo/site/api/v1/path/{pathOfTheDoc}/@pp/{myPageProvider}
```

Same query parameters apply for pagination as for the `@children` adapter.

As it would be great to not only bind the id of a document, but also the document itself as a parameter of the page provider (using EL to define parameters). This API is then subject to change during the 5.9.x series.

Automation Chains

On a Document

As we have a way to point to documents with resources, we provide a way to run an automation chain on them. You can use any chain that takes a document as input. For instance to call that operation `myOperation` on a document, just send an POST request like that:

```
POST /nuxeo/site/api/v1/path/{pathOfTheDoc}/@op/{myOperation}
{
  params: {
    opParam: "value"
  }
}
```

The response will depend on the result of the automation chain.

If you want to run a chain, just use the same endpoint and prefix the chain name by `Chain.`, for instance:

```
POST /nuxeo/site/api/v1/path/{pathOfTheDoc}/@op/Chain.{myChain}
{
  params: {
    chainParam: "value"
  }
}
```

On a List of Documents

We also have a way to point to lists of documents thru the use of list adapters. This way, we are also able to run operations or chains on lists of documents. For instance:

```
POST /nuxeo/site/api/v1/path/{pathOfTheFolder}/@children/@op/Chain.myChain
{
  params: {
 chainParam: "value"
  }
}
```

This way we run the `myChain` operation on the children of the given folder.

Pay attention to the fact that document list adapters are paged. That means that the chain will run on all document of the current page.

Business Object Adapters

Since 5.7.2, you can use Business objects with Automation. They are also bound on the REST API and you just have to use the same semantics than for document resources.

Getting a Business Object

```
GET /nuxeo/site/api/v1/path/{pathOfTheDoc}/@bo/BusinessBeanAdapter
```

will return

```
{
    entity-type: "BusinessBeanAdapter"
    id: "37b1502b-26ff-430f-9f20-4bd0d803191e",
    "type": "Domain",
    "title":"Default domain"
    "description:""
}
```

Updating a Business Object

To update a business object, you just have to send a PUT request one the business object resource with its content data like this:

```
PUT /nuxeo/site/api/v1/path/{pathOfTheDoc}/@bo/BusinessBeanAdapter
{
 "entity-type: "BusinessBeanAdapter"
 "value": {
  id: "37b1502b-26ff-430f-9f20-4bd0d803191e",
     "type": "Domain",
     "title":"Default domain"
     "description:"My new description"
  }
}
```

Creating a Business Object

And then to create a business object, you have to issue a POST on the object resource plus the name of the newly created document, like this:

```
POST /nuxeo/site/api/v1/path/{pathOfTheDoc}/@bo/BusinessBeanAdapter/{newName}
{
 entity-type: "BusinessBeanAdapter",
 value: {
   "type": "Note",
   "title":"A sample note",
   "description:"",
   "note":"The content of my note"
 }
}
```

Command Endpoint

The Automation HTTP / REST Service

The Nuxeo Automation Server module provides a REST API to execute operations on a Nuxeo server.

> **About Automation**
> To know more about Content Automation, you can read the concepts and the practical information in other sections of this space. This section only deals with the REST exposition of operations and chains.

To use the Automation REST service you need to know the URL where the service is exposed, and the different formats used by the service to exchange information. All the other URLs that appear in the content exchanged between the client and server are relative paths to the Automation service URL.

Operations context and parameters as well as response objects (documents) are formatted as JSON objects. To transport blobs, HTTP multipart requests should be used to attach blob binary data along with the JSON objects describing the operation request.

The REST service is bound to the `http://<host>/nuxeo/site/automation` path. To get the service description you should do a GET on the service URL using an Accept type of `application/json+nxautomation`. You will get in response the service description as a JSON document. This document will contain the list of available operations and chains, as well as the URLs for other optional services provided (like login or document type service).

By default all the chains and operations that are not UI related are accessible through REST. Anyway you can filter the set of exposed operations and chains or protect them using security rules. For more details on this see Filtering Exposed Operations.

Example - Getting the Automation Service.

Let say the service is bound to http://localhost:8080/nuxeo/site/automation. Then to get the service description you should do:

```
GET http://localhost:8080/nuxeo/site/automation
Accept: application/json+nxautomation
```

You will get response a JSON document like:

```
HTTP/1.1 200 OK
Content-Type: application/json+nxautomation
```

In this section

```
{
  "paths": {"login" : "login"},
  "operations": [
      "id" : "Blob.Attach",
      "label": "Attach File",
      "category": "Files",
      "description": "Attach the input file to the document given as a parameter. If
the xpath points to a blob list then the blob is appended to the list, otherwise the
xpath should point to a blob property. If the save parameter is set the document
modification will be automatically saved. Return the blob.",
      "url": "Blob.Attach",
      "signature": [ "blob", "blob" ],
      "params": [
         {
           "name": "document",
            "type": "document",
            "required": true,
            "values": []
         },
         {
           "name": "save",
           "type": "boolean",
           "required": false,
           "values": ["true"]
         },
         {
           "name": "xpath",
           "type": "string",
           "required": false,
           "values": ["file:content"]
         }
      ]
  // ... other operations follow here
  ],
  "chains" : [
    // a list of operation chains (definition is identical to regular operations)
  ]
}
```

You can see the automation service is returning the registry of operations and chains available on the server.

Each operation and chain signature is fully described to be able to do operation validation on client side, for instance. Also some additional information that can be used in an UI is provided (operation label, full description, operation category etc.).

The `url` property of an operation (or automation chain) is the relative path to use to execute the operation. For example if the service URL is `http://localhost:8080/nuxeo/site/automation` and the `Blob.Attach` operation has the `url` `Blob.Attach` then the full URL to that operation will be: `http://localhost:8080/nuxeo/site/automation/Blob.Attach`.

The `paths` property is used to specify various relative paths (relative to the automation service) of services exposed by the automation server. In the above example you can see that the "login" service is using the relative path "login".

This service can be used to sign-in and check if the username/password is valid. To use this service you should do a POST to the login URL (e.g. `http://localhost:8080/nuxeo/site/automation/login`) using basic authentication. If authentication fails you will receive a 401 HTTP response. Otherwise the 200 code is returned.

The login service can be used to do (and check) a user login. Note that `WWW-Authenticate` server response is not yet implemented so you need to send the basic authentication header in each call if you are not using cookies (in that case you only need once to authenticate - for example using the login service).

You do not need to be logged in to be able to get the Automation service description.

Executing Operations

The operations registry (loaded doing a GET on the Automation service URL) contains the entire information you need to execute operations.

To execute an operation you should build an operation request descriptor and post it on the operation URL. When sending an operation request you must use the `application/json+nxrequest` content type. Also you need to authenticate your request (using basic authentication) since most of the operations are accessing the Nuxeo repository.

An operation request is a JSON document having the following format:

```
input: "the_operation_input_object_reference",
  params: {key1: "value1", key: "value2", ...},
  context: {key1: "val1", ... }
```

All these three request parameters are optional and depend on the executed operation.

- If the operation has no input (a void operation) then the input parameter can be omitted.
- If the operation has no parameters then `params` can be omitted.
- If the operation does not want to push some specific properties in the operation execution context then context can be omitted. In fact context parameters are useless for now but may be used in future.

The `input` parameter is a string that acts as a reference to the real object to be used as the input. There are four types of supported inputs: void, document, document list, blob, blob list.

- To specify a "void" input (i.e. no input) you should omit the `input` parameter.
- To specify a reference to a document you should use the document absolute path or document UID prefixed using the string "`doc:`".
  Example: "`doc:/default-domain/workspaces/myworkspace`" or "`doc:96bfb9cb-a13d-48a2-9bbd-9341fcf24801`"
- To specify a reference to a list of documents you should use a comma separated list of absolute document paths, or UID prefixed by the string "`docs:`".
  Example: "`docs:/default-domain/workspaces/myworkspace, 96bfb9cb-a13d-48a2-9bbd-9341fcf24801`".
- When using blobs (files) as input you cannot refer them using a string locator since the blob is usually a file on the client file system or raw binary data.
  For example, let say you want to execute the `Blob.Attach` operation that takes as input a blob and set it on the given document (the document is specified through 'params'). Because the file content you want to set is located on the client computer, you cannot use a string reference. In that case you MUST use a multipart/related request that encapsulate as the root part your JSON request as an `application/json+nxrequest` content and the blob binary content in a related part.
- In case you want a list of blobs as input then you simply add one additional content part for each blob in the list.
  The only limitation (in both blob and blob list case) is to put the request content part as the first part in the multipart document. The order of the blob parts will be preserved and blobs will be processed in the same order (the server assumes the request part is the first part of the multipart document - e.g. Content-Ids are not used by the server to identify the request part).

Example 1 - Invoking A Simple Operation

```
POST /nuxeo/site/automation/Document.Fetch HTTP/1.1
Accept: application/json+nxentity, */*
Content-Type: application/json+nxrequest; charset=UTF-8
Authorization: Basic QWRtaW5pc3RyYXRvcjpBZG1pbmlzdHJhdG9y
Host: localhost:8080
```

```
{"params":{"value":"/default-domain/workspaces/myws/file"},"context":{}}
```

This operation will return the document content specified by the "`value`" parameter.

Example 2 - Invoking An Operation Taking a Blob as Input

Here is an example on invoking `Blob.Attach` operation on a document given by its path ("`/default-domain/workspaces/myws/file`" in our example).

```
POST /nuxeo/site/automation/Blob.Attach HTTP/1.1
Accept: application/json+nxentity, */*
Content-Type: multipart/related;
    boundary="----=_Part_0_130438955.1274713628403";
type="application/json+nxrequest"; start="request"
Authorization: Basic QWRtaW5pc3RyYXRvcjpBZG1pbmlzdHJhdG9y
Host: localhost:8080
```

```
------=_Part_0_130438955.1274713628403
Content-Type: application/json+nxrequest; charset=UTF-8
Content-Transfer-Encoding: 8bit
Content-ID: request
Content-Length: 75

{"params":{"document":"/default-domain/workspaces/myws/file"},"context":{}}

------=_Part_0_130438955.1274713628403
Content-Type: image/jpeg
Content-Transfer-Encoding: binary
Content-Disposition: attachment; filename=test.jpg
Content-ID: input

[binary data comes here]

------=_Part_0_130438955.1274713628403--
```

In both examples you can see that the following `Accept` header is used:
`Accept: application/json+nxentity, */*`.

This header is important since is is specifying that the client accept as a response either a JSON encoded entity, either a blob that may have any type (in case of blob download). The `application/json+nxentity` is the first content type to help the server choose the format of the response when returning an encoded object.

Operation Execution Response

An operation can have one of the following output types:

- `document`: A repository document;
- `documents`: A list of documents;
- `blob`: A blob (binary content usually attached to a document);
- `blobs`: A list of blobs;
- `void`: The operation has no output (output is void).

Apart from these possible outputs, the operation can abort due to an exception.

All these cases are represented using the following HTTP responses:

- `document` -> A JSON object describing the document is returned. The used Content-Type is **`application/json+nxentity`**
- `documents` -> a JSON object describing the document list is returned. The used Content-Type is **`application/json+nxentity`**
- `blob` -> The blob raw content is returned. The used Content-Type will be the same as the blob mime-type.
- `blobs` -> A Multipart/Mixed content is returned. Each part will be a blob from the list (order is preserved). Each part will use the right Content-Type as given by the blob mime-type.
- `void` -> HTTP **204** is returned. No content and no Content-Type is returned.
- `exception` -> A status code > **400** is returned and the content will be the server exception encoded as a JSON object. The used Content-Type is **`application/json+nxentity`**.

When an exception occurs, the server tries to return a meaningful status code. If no suitable status code is found, a generic 500 code (server error) is used.

You noticed that each time when returned objects are encoded as JSON objects, the **`application/json+nxentity`** Content-Type will be used. We also saw that only document, documents and exception objects are encoded as JSON.

Here we will discuss the JSON format used to encode these objects.

Document

A JSON document entity contains the minimal required information about the document as top level entries.

These entries are always set on any document and are using the following keys:

- `uid`: The document UID;
- `path`: The document path (in the repository);
- `type`: The document type;
- `state`: The current life cycle state;
- `title`: The document title;
- `lastModifed`: The last modified timestamp.

All the other document properties are contained within a "properties" map using the property XPath as the key for the top level entries.

Complex properties are represented as embedded JSON objects and list properties as embedded JSON arrays.

All `application/json+nxentity` JSON entities always contains a required top level property: "`entity-type`". This property is used to identify which type of object is described. There are three possible entity types:

- `document`
- `documents`
- `exception`

Example of a JSON document entry:

```
{
  "entity-type": "document",
  "uid": "96bfb9cb-a13d-48a2-9bbd-9341fcf24801",
  "path": "/default-domain/workspaces/myws/file",
  "type": "File",
  "state": "project",
  "title": "file",
  "lastModified": "2010-05-24T15:07:08Z",
  "properties":   {
    "uid:uid": null,
    "uid:minor_version": "0",
    "uid:major_version": "1",
    "dc:creator": "Administrator",
    "dc:contributors": ["Administrator"],
    "dc:source": null,
    "dc:created": "2010-05-22T08:42:56Z",
    "dc:description": "",
    "dc:rights": null,
    "dc:subjects": [],
    "dc:valid": null,
    "dc:format": null,
    "dc:issued": null,
    "dc:modified": "2010-05-24T15:07:08Z",
    "dc:coverage": null,
    "dc:language": null,
    "dc:expired": null,
    "dc:title": "file",
    "files:files": [],
    "common:icon": null,
    "common:icon-expanded": null,
    "common:size": null,
    "file:content":     {
      "name": "test.jpg",
      "mime-type": "image/jpeg",
      "encoding": null,
      "digest": null,
      "length": "290096",
      "data": "files/96bfb9cb-a13d-48a2-9bbd-9341fcf24801?path=%2Fcontent"
    },
    "file:filename": null
  }
```


The top level properties "`title`" and "`lastModified`" have the same value as the corresponding embedded properties "`dc:title`" and "`dc:modified`".

The blob data, instead of containing the raw data, contain a relative URL (relative to automation service URL) that can be used to retrieve the real data of the blob (using a GET request on that URL).

Documents

The documents JSON entity is a list of JSON document entities and have the entity type "`documents`". The documents in the list are containing only the required top level properties.

Example:

```
{
  entity-type: "documents"
  entries: [
   {
     "entity-type": "document",
     "uid": "96bfb9cb-a13d-48a2-9bbd-9341fcf24801",
     "path": "/default-domain/workspaces/myws/file",
     "type": "File",
     "state": "project",
     "title": "file",
     "lastModified": "2010-05-24T15:07:08Z",
   },
   ...
   ]
 }
```

Exception

JSON exception entities have a "`exception`" entity type and contain information about the exception, including the server stack trace.

Example:

```
{
  "entity-type": "exception",
  "type": "org.nuxeo.ecm.automation.OperationException",
  "status": 500,
  "message": "Failed to execute operation: Blob.Attach",
  "stack": "org.nuxeo.ecm.automation.OperationException: Failed to invoke operation
Blob.Attach\n\tat
org.nuxeo.ecm.automation.core.impl.InvokableMethod.invoke(InvokableMethod.java:143)\n\
t ..."
}
```

Special HTTP Headers

There are four custom HTTP headers that you can use to have more control on how operations are executed.

X-NXVoidOperation

Possible values: "true" or "false". If not specified the default is "false"

This header can be used to force the server to assume that the executed operation has no content to return (a void operation). This can be very useful when dealing with blobs to avoid having the blob output sent back to the client.

For example, if you want to set a blob content on a document using `Blob.Attach` operation, after the operation execution, the blob you sent to the server is sent back to the client (because the operation is returning the original blob). This behavior is useful when creating operation chains but when calling such an operation from remote it will to much network traffic than necessary.

To avoid this, use the header: `X-NXVoidOperation: true`

Example:

```
POST /nuxeo/site/automation/Blob.Attach HTTP/1.1
Accept: application/json+nxentity, */*
Content-Type: multipart/related;
    boundary="----=_Part_0_130438955.1274713628403";
type="application/json+nxrequest"; start="request"
Authorization: Basic QWRtaW5pc3RyYXRvcjpBZG1pbmlzdHJhdG9y
X-NXVoidOperation: true
Host: localhost:8080
```

```
------=_Part_0_130438955.1274713628403
Content-Type: application/json+nxrequest; charset=UTF-8
Content-Transfer-Encoding: 8bit
Content-ID: request
Content-Length: 75

{"params":&amp;nbsp;{"document":"/default-domain/workspaces/myws/file"}, "context":{}}

------=_Part_0_130438955.1274713628403
Content-Type: image/jpeg
Content-Transfer-Encoding: binary
Content-Disposition: attachment; filename=test.jpg
Content-ID: input

[binary data comes here]

------=_Part_0_130438955.1274713628403--
```

Nuxeo-Transaction-Timeout

This header can be used when you want to control the transaction duration. As an example, if you want to inject a large blob in the repository, the default transaction timeout may be not enough. You can specify a 5 minutes timeout for the chain you're executing:

```
Nuxeo-Transaction-Timeout: 300
```

X-NXDocumentProperties

This header can be used whenever a Document will be returned by the server. The header is forcing the server to fill up the returned document with data from schemas that matches the `X-NXDocumentProperties` filter. So, `X-NXDocumentProperties` is a filter of schemas. If you don't use the header only the minimal required properties of the document are returned.

To have more properties in the returned document, you can specify a list of document schema names:

```
X-NXDocumentProperties: dublincore, file
```

or to have all the document content, you can use the '*' character as the filter value:

```
X-NXDocumentProperties: *
```

X-NXRepository

This header can be used to when you need to access a specific repository. The default value is 'default', as it's the default repository name in

Nuxeo. This is handy if you have change the default name or if you have multiple repositories.

```
X-NXRepository: myCustomRepositoryName
```

Document Property Types

Here you can find more details on the JSON document entity format.

- A document entity is using a string value for any scalar property value.
- Dates are encoded as W3C dates (in UTC timezone)
- Apart from strings, you may have null values for properties that are not set (but are defined by the schema), JSON objects (maps) for complex properties, and JSON arrays for array or list properties.
- Blob data is encoded as a relative URL (relative to Automation service URL) from where you can download the raw data of the blob (using a GET request on that URL).

Property values can be of one of these types:

- `string`
- `long`: Encoded as a string representation of the number (in Java: `Long.toString()`);
- `double`: Encoded as a string representation of the number (in Java: `Double.toString()`);
- `date`: Encoded as a W3C format (UTC timezone preferred);
- `boolean`: "true" or "false";
  For null values the JSON null keyword is used.

Operation Request Parameters

Here you can find more details about the request format.

Request input

As we seen a request may have as input:

- A document,
- A list of documents,
- A blob,
- A list of blobs.

To refer to a document you should use either the absolute path of the document (starting with '/' !) or the document UID.
Examples:

- `input: "/"` -> to refer to the repository root
- `input: "/default-domain/workspaces/ws"` -> to refer to the 'ws' workspace
- `input: "96bfb9cb-a13d-48a2-9bbd-9341fcf24801"` -> to refer to a document havin this UID.

To refer to a list of documents, you must use a comma separated list of document identifiers.
Example:
`input: "/default-domain/workspaces/ws, 96bfb9cb-a13d-48a2-9bbd-9341fcf24801"`

When using a blob as the input of an operation, you cannot use the "input" request property since the blob may contain binary data. In that case you must use a Multipart/Related request as described above and put the blob raw data in the second body part of the request.

To use a list of blobs as the input, you should use the same method as for a single blob, and put each blob in the list in a separate body part. Note that the order of the body parts is important: blobs will be processed in the same order that they appear in the Multipart document.

Also, when using Multipart/Related requests you must always put the JSON encoded request in the first body part.

Request parameter types

The operation parameters specified inside the `params` property of the request are all strings. Operation parameters are typed, so on the server side the operation will know how to decode parameters in real Java classes. The supported parameter types are: string, long (integer number), double (floating point number), date, properties, document, documents, EL expression, EL template.

Here are some rules on how to encode operation parameters:

- `string`: Let it as is.
- `long`: Just use a string representation of the number (in Java `Long.toString()`).
- `double`: Just use a string representation of the number (in Java `Double.toString()`).
- `date`: Use the W3C format (UTC timezone preferred).
- `boolean`: "true" or "false".
- `document`: Use the document UID or the absolute document path.

- `documents`: Use a comma separated list of document references.
- EL expression: put the "`expr:`" string before your EL expression. (E.g. `expr: Document.path`).
- EL template: put the "`expr:`" string before your template. (E.g. `expr: SELECT * FROM Document WHERE dc:title=@{my_var}`)

Note that expressions also you specify relative paths (relative to the context document) using "`expr: ./my/doc`".

In fact all these encoding rules are the same as the one you can use when defining operation chains in Nuxeo XML extensions.

Operation vs. Transactions

The server runs an operation or operation chain in a single transaction. A rollback is done if the operation execution caused an exception.

The transaction is committed when the operation (or operation chain) successfully terminated.

Operations can be executed in two different contexts: either in the context of a stateful repository session, either one session per operation.

By default operations are reusing the same session if your client supports cookies (even in basic authentication).

To enable stateless sessions, you need to modify some Nuxeo configuration. This will not be explained here (TODO: add link). In stateless mode the session is closed at the end of the request.

Note that Automation service is using Nuxeo WebEngine for HTTP request management.

Operation Security

Some operations are allowed to be executed only by some users or groups. This is defined on the server side through Nuxeo XML extensions.

See Filtering Exposed Operations for more details.

Filtering Exposed Operations

Almost all the registered operations and chains are automatically exposed through a REST interface to be invoked from remote clients. The UI-specific operations are not exposed through REST since they require a web user interface to work.

For security reasons, you may want to prevent some operations from being accessed remotely. Or you may want to allow only certain users to be able to invoke them.
The REST operation filters provide an extension point where you can register such security rules on what operations are exposed and for which users.

Here is an example on how to write such an extension:

```
<extension target="org.nuxeo.ecm.automation.server.AutomationServer"
point="bindings">
  <binding name="Document.Delete" disabled="true"/>
  <binding name="audit" chain="true">
    <administrator>true</administrator>
    <secure>true</secure>
    <groups>members</groups>
  </binding>
</extension>
```

The above code is contributing two REST bindings - one for the atomic operation `Document.Delete` which is completely disabled (by using the `disable` parameter) and the second one is defining a security rule for the automation chain named `audit`. You can notice the usage of the `chain` attribute which must be set to true every time a binding refers to an automation chain and not to an atomic operation.

The second binding installs a guard that allows only requests made by an `administrator` user or by users from the `member` group **AND** th e request should be made over a secured channel like HTTPS.

Here is the complete of attributes and elements you can use in the extension:

- **`name`** : The operation or automation chain name that should be protected.
- **`chain`** : "true" if the name refers to an automation chain, "false" otherwise (the default is "false").
- **`disabled`** : Whether or not to completely disable the operation from REST access. The default is "false". If you put this flag on "true" then all the other security rules will be ignored.
- **`administrator`** : Possible values are "true" or "false". The default is "false". If set to "true" the operation is allowed if the user is an administrator.
- **`groups`** : A comma separated list of groups that the user should be member of. If both `administrator` and `groups` are specified the user must be either from a group or an administrator.
- **`secure`** : "true" or "false". Default is "false". If "true" the request must be done through a secured channel like HTTPS. If this guard is used the connection **must** be secured, so that even if the groups guard is matched the operation is not accessible if the connection

is not secured.

Related Documentation
• REST API • Command Endpoint • Automation

Blob Upload for Batch Processing

Motivations

The default way Automation deals with Blobs is to use the standard HTTP MultiPart Encoding.

This strategy can not fit when:

- Your client does not natively support multipart encoding;
  Ex: JavaScript (without using a Form), Android SDK 2.x.
- You have several files to send, but prefer to send them as separated chunk;
  Ex: You have an HTTP proxy that will limit POST size.
- You want to upload files as soon as possible and then run the operation when everything has been uploaded on the server;
  Ex: You upload pictures you select from a mobile device.

In this section

Uploading Files

The principle is that you can upload the file to the server using the URL:

```
POST /site/automation/batch/upload
```

You can do a simple POST with the payload containing your file.

However, you will need to set some custom HTTP headers:

Header name	Description
`X-Batch-Id`	Batch identifier
`X-File-Idx`	Index of the file inside the batch
`X-File-Name`	Name of the file
`X-File-Size`	Size of the file in bytes
`X-File-Type`	Mime type of the file
`Content-Type`	Should be set to "`binary/octet-stream`"

Optionally depending on the HTTP client you are using you might need to add the Content-Length header to specify the size of the file in bytes.

The batch identifier should be common to all the files you want to upload and attach to the same batch. This identifier should be client side generated:

- GUID,

- Timestamp + random number,
- whatever that can be reasonably considered as unique.

The `X-File-Idx` is here in case you later want to reference the file by its index and also to keep track of the client side ordering: because the order the server receives the files may not be the same.

The files attached to the batch are stored on a temporary disk storage (inside `java.io.tmp`) until the batch is executed or dropped.

To drop a batch you must use:

```
GET /site/automation/batch/drop/{batchId}
```

Technically, a batch is automatically "started" when the first upload is received.

Executing a batch will automatically remove it.

Using Files From a Batch

Batch Execute

You can execute an automation chain or an automation operation using the blobs associated to a batch as input.

To place the blobs as input, call a specific batch operation, passing the `operationId` and `batchId` as parameter:

```
POST /site/automation/batch/execute
Accept: application/json+nxentity, */*
Content-Type: application/json+nxrequest; charset=UTF-8
X-NXDocumentProperties: *
```

```
{"params":{"operationId":"Chain.FileManager.ImportInSeam","batchId":"batch-13702823349
17-258", ...},"context":{...}}
```

Optionally you can pass the `fileIdx` parameter to specify the index of the file inside the batch that you want to use as input of the chain or operation to execute.

This way of calling automation operation is actually used in the default UI to manage Drag&Drop:

1. Files are progressively uploaded to the server:
    - You can drop several sets of files,
    - There is a maximum number of concurrent uploads.
2. When upload is finished you can select the operation or chain to execute.

More info about Drag & Drop configuration.

Referencing a Blob From a Batch

An other option is to reference the file within the batch to create input parameters of an operation.

For that you can add a parameter of type `properties` that will automatically be resolved to the correct blob if the provided properties are the correct ones:

```
type = blob
length = 657656
mime-type = application/pdf
name = myfile.pdf
upload-batch = 989676879865765
upload-fileId = myfile.pdf
```

When using Java automation client, this would look like:

```
PropertyMap blobProp = new PropertyMap();
blobProp.set("type", "blob");
blobProp.set("length", new Long(blobUploading.getLength()));
blobProp.set("mime-type", blobUploading.getMimeType());
blobProp.set("name", blobToUpload.getFileName());
// set information for server side Blob mapping
blobProp.set("upload-batch", batchId);
blobProp.set("upload-fileId", blobUploading.getFileName());
```

Referencing a Blob from a JSON Document Resource

You can use the `batchId` property for blob in the JSON document you're sending to the REST API.

```
{
    "entity-type": "document",
    "repository": "default",
    "uid": "531d9636-46c2-497d-996b-1ae7a8f43e89",
    "path": "/default-domain",
    "type": "Domain",
    "state": "project",
    "versionLabel": "",
    "title": "Default domain",
    "lastModified": "2013-09-06T08:53:10.00Z",
    "properties": {
        "file:content": {
             "upload-batch":"batchId-20358",
             "upload-fileId":"0" // referencing the first file of the batch
         }
     },
    "facets": [
        "SuperSpace",
        "Folderish"
    ],
    "changeToken": "1378457590000",
    "contextParameters": {}
}
```

Java API

Clients

The platform already provides multiple clients for accessing the platform remotely. We also provide a test suite from which you can get inspiration if you want to write a new client for Nuxeo (for example in C#, in C, ...) and want to assert your level of compliance.

- Java Automation Client — Nuxeo provides a high level client implementation for Java programmers: Nuxeo Automation Client API simplifies your task since it handles all the protocol level details.
- PHP Automation Client — A PHP automation client is made available on GitHub. You can use it and ask for commit rights on the project if you want to improve it or fix a bug. The project contains the library and some sample use cases.
- Using a Python Client — Alternatively you can use the standard library of Python to access a Nuxeo repository using the Content Automation API. Here is a worked example with screencast that demonstrates how to deploy a custom server side operation developed in Java using the Nuxeo IDE and a client Python script that calls it: Exploring Nuxeo APIs: Content Automation.
- Client API Test suite (TCK) — This chapter provides a test suite that can be used to test the implementation of an automation client library.

- iOS Client
- Android Client
- Using cURL — In this example we are using the UNIX curl command line tool to demonstrate how to invoke remote operations.

Java Automation Client

Nuxeo provides a high level client implementation for Java programmers: Nuxeo Automation Client API simplifies your task since it handles all the protocol level details.

Dependencies

- This documentation applies for `nuxeo-automation-client` versions greater or equal to `5.4`.
- The Nuxeo Automation Client is also available with all dependencies included (JAR shaded - nuxeo-automation-XX-jar-with-dependencies)

To use the Java Automation client you need to put a dependency on the following Maven artifact:

```
<dependency>
    <groupId>org.nuxeo.ecm.automation</groupId>
    <artifactId>nuxeo-automation-client</artifactId>
    <version>...</version>
</dependency>
```

with dependencies

```
<dependency>
    <groupId>org.nuxeo.ecm.automation</groupId>
    <artifactId>nuxeo-automation-client</artifactId>
    <version>...</version>
    <classifier>jar-with-dependencies</classifier>
 </dependency>
```

For a direct download, see https://maven.nuxeo.org/.

The client library depends on:

- `net.sf.json-lib:json-lib`, `net.sf.ezmorph:ezmorph` - for JSON support
- Since Nuxeo 5.6, `net.sf.json-lib:json-lib` is being replaced with `org.codehaus.jackson:jackson-core-asl` and `org.codehaus.jackson:jackson-mapper-asl`.
- `org.apache.httpcomponents:httpcore`, `org.apache.httpcomponents:httpclient` - for HTTP support
- `javax.mail` - for multipart content support

Automation Client API

Let's take an example and execute the "`SELECT * FROM Document`" query against a remote Automation server:

In this section


```
import org.nuxeo.ecm.automation.client.jaxrs.impl.HttpAutomationClient;
import org.nuxeo.ecm.automation.client.model.Documents;
import org.nuxeo.ecm.automation.client.Session;

public static void main(String[] args) throws Exception {
    HttpAutomationClient client = new HttpAutomationClient(
           "http://localhost:8080/nuxeo/site/automation");

    Session session = client.getSession("Administrator", "Administrator");
    Documents docs = (Documents) session.newRequest("Document.Query").set(
           "query", "SELECT * FROM Document").execute();
    System.out.println(docs);

    client.shutdown();
}
```

You can see the code above has three distinctive parts:

1. Opening a connection.
2. Invoking remote operations.
3. Destroying the client.

Opening a connection

1. Before using the Automation client you should first create a new client that is connecting to a remote address you can specify through the constructor URL argument. As the remote server URL you should use the URL of the Automation service.

   ```
   // create a new client instance
   HttpAutomationClient client = new
   HttpAutomationClient("http://localhost:8080/nuxeo/site/automation");
   ```

   No connection to the remote service is made at this step. The Automation service definition will be downloaded the first time you create a session.

   A local registry of available operations is created from the service definition sent by the server.

   > The local registry of operations contains all operations on the server - but you can invoke only operations that are accessible to your user - otherwise a 404 (operation not found) will be sent by the server.

   Once you created a client instance you **must** create a new session to be able to start to invoke remote operations. When creating a

session you should pass the credentials to be used to authenticate against the server.

2. So you create a new session by calling:

```
import org.nuxeo.ecm.automation.client.Session;

Session session = client.getSession("Administrator", "Administrator");
```

This will authenticate you onto the server using the basic authentication scheme.

If needed, you can use another authentication scheme by setting an interceptor.

```
client.setInterceptor(new PortalSSOAuthInterceptor("nuxeo5secretkey",
"Administrator"));
Session session = client.getSession();
```

Invoking remote operations

Using a session you can now invoke remote operations.

1. To create a new invocation request you should pass in the right operation or chain ID.

```
OperationRequest request = session.newRequest("Document.Query");
```

2. Then populate the request with all the required arguments.

```
request.set("query", "SELECT * FROM Document");
```

You can see in the example that you have to specify only the `query` argument. If you have more arguments, call the `set` method in turn for each of these arguments. If you need to specify execution context parameters you can use `request.setContextProperty` method. On the same principle, if you need to specify custom HTTP headers you can use the `request.setHeader` method.

3. After having filled all the required request information you can execute the request by calling the execute method.

```
Object result = request.execute();
```

The client API provides both synchronous and asynchronous execution
For an asynchronous execution you can make a call to the `execute(AsyncCallback<Object> cb)` method.

Beware that you can set the timeout in milliseconds for the wait of the aynchronous thread pool termination at client shutdown by passing it to the `HttpAutomationClient` constructor: `public HttpAutomationClient(String url, long httpConnectionTimeout, long asyncAwaitTerminationTimeout)`.

Default value is 2 seconds. If you don't use any asynchronous call you can set this timeout to 0 in order not to wait at client shutdown.

Setting the HTTP connection timeout
You can set a timeout in milliseconds for the HTTP connection in order to avoid an infinite wait in case of a network cut or if the Nuxeo server is not responding.

Just pass it to the `HttpAutomationClient` constructor: `public HttpAutomationClient(String url, long httpConnectionTimeout, long asyncAwaitTerminationTimeout)`.

Default value is 0 = no timeout.

Executing a request will either throw an exception or return the result. The result object can be null if the operation has no result (i.e. a void operation) - otherwise a Java object is returned. The JSON result is automatically decoded into a proper Java object. The following objects are supported as operation results:

- Document - a document object
- Documents - a list of documents
- Blob - a file
- Blobs - a file list

In case the operation invocation fails - an exception described by a JSON entity will be sent by the server and the client will automatically decode it into a real Java exception derived from `org.nuxeo.ecm.automation.client.jaxrs.RemoteException`.

Before sending the request the client will check the operation arguments to see if they match the operation definition and will throw an exception if some required argument is missing. The request will be sent only after validation successfully completes.

The query example is pretty simply. The query operation doesn't need an input object to be executed. (i.e. the input can be null). But most operations require an input. In that case you must call `request.setInput` method to set the input. We will see more about this in the following examples.

If you prefer a most compact notation you can use the fluent interface way of calling methods:

```
Object result = session.newRequest("OperationId").set("var1", "val1").set("var2",
"val2").execute();
```

Destroying the client

When you are done with the client you must call the `client.disconnect` method to free any resource held by the client. Usually this is done only once when the client application is shutdown. Creating new client instances and destroying them may be costly so you should use a singleton client instance and use it from different threads (which is safe).

If you need different logins then create one session per login. A session is thread safe.

Create Read Update Delete

In this example we assume we already have a session instance.

Here is an example with Document.Create, Document.Update, Document.Delete, Document.Fetch operations.

You can see The Automation Client Service Adapter section to know how CRUD operations are wrapped into DocumentService adapter and how to use it.

```
import org.nuxeo.ecm.automation.client.model.Document;
import org.nuxeo.ecm.automation.client.Session;
import org.nuxeo.ecm.automation.client.Constants;
import org.nuxeo.ecm.automation.client.model.IdRef;

// Fetch the root of Nuxeo repository
Document root = (Document) session.newRequest("Document.Fetch").set("value",
"/").execute();

// Instantiate a new Document with the simple constructor
Document document = new Document("myDocument", "File");
document.set("dc:title", "My File");
document.set("dc:description", "My Description");

// Create a document of File type by setting the parameter 'properties' with String
metadata values delimited by comma ','
document = (Document)
session.newRequest("Document.Create").setHeader(Constants.HEADER_NX_SCHEMAS,
"*").setInput(root).set("type", document.getType()).set("name",
document.getId()).set("properties", document).execute();

// Set another title to update
document.set("dc:title", "New Title");

// Update the document
document = (Document)
session.newRequest("Document.Update").setInput(document).set("properties",
document).execute();

// Delete the document
session.newRequest("Document.Delete").setInput(document).execute();
```

Operations examples above fetch the document with common properties: `common`, `dublincore`, `file`. If you wish to fetch all properties or a specific schema, you can do as this following example:

```
import org.nuxeo.ecm.automation.client.model.Document;
import org.nuxeo.ecm.automation.client.Session;
import org.nuxeo.ecm.automation.client.Constants;
import org.nuxeo.ecm.automation.client.model.IdRef;

Document document = new Document("myDocument", "File");

// For all operations -> use setHeader method to specify specific schemas or every
ones (*)

Document root = (Document)
session.newRequest("Document.Fetch").setHeader(Constants.HEADER_NX_SCHEMAS,
"*").set("value", "/").execute();

document = (Document)
session.newRequest("Document.Create").setHeader(Constants.HEADER_NX_SCHEMAS,
"dublincore,blog").setInput(root).set("type", document.getType()).set("name",
document.getId()).set("properties", document).execute();

document = (Document)
session.newRequest("Document.Update").setHeader(Constants.HEADER_NX_SCHEMAS,
"dublincore,blog").setInput(document).set("properties", document).execute();
```

You can fetch all updated properties during the session with the following method of the document:

```
// Fetch the dirty properties (updated values) from the document
PropertyMap dirties = document.getDirties();
```

To see complex properties example and the use of `propertyMap`, see the Managing Complex Properties section.

Managing Blobs

In this example we assume we already have a session instance.

The example will create a new File document into the root "/" document and then will upload a file into it. Finally we will download back this file.

1. Get the root document and create a new File document at location `/myfile`.

```
import org.nuxeo.ecm.automation.client.model.Document;
import org.nuxeo.ecm.automation.client.Session;

// get the root
Document root = (Document) session.newRequest("Document.Fetch").set("value",
"/").execute();

// create a file document
session.newRequest("Document.Create").setInput(root).set("type",
"File").set("name", "myfile").set("properties", "dc:title=My File").execute();
```

Note the usage of `setInput()` method. This is to specify that the create operation must be executed in the context of the root document - so the new document will be created under the root document. Also you can notice that the input object is a Document instance.

2. Now get the file to upload and put it into the newly created document.

```
File file = getTheFileToUpload();
        FileBlob fb = new FileBlob(file);
        fb.setMimeType("text/xml");
        // uploading a file will return null since we used HEADER_NX_VOIDOP
        session.newRequest("Blob.Attach").setHeader(
                Constants.HEADER_NX_VOIDOP, "true").setInput(fb)
                .set("document", "/myfile").execute();
```

The last `execute` call will return `null` since the `HEADER_NX_VOIDOP` header was used. This is to avoid receiving back from the server the blob we just uploaded.

Note that to upload a file we need to use a `Blob` object that wrap the file to upload.

3. Now get the the file document where the blob was uploaded.
4. Then retrieve the blob remote URL from the document metadata. We can use this URL to download the blob.

```
import org.nuxeo.ecm.automation.client.model.Document;
        import org.nuxeo.ecm.automation.client.Session;
        import org.nuxeo.ecm.automation.client.Constants;

        // get the file document where blob was attached
        Document doc = (Document) session.newRequest(
                "Document.Fetch").setHeader(
                Constants.HEADER_NX_SCHEMAS, "*").set("value",
"/myfile").execute();
        // get the file content property
        PropertyMap map = doc.getProperties().getMap("file:content");
        // get the data URL
        String path = map.getString("data");
```

You can see we used the special `HEADER_NX_SCHEMAS` header to specify we want all properties of the document to be included in the response.

5. Now download the file located on the server under the `path` we retrieved from the document properties:

```
// download the file from its remote location
        blob = (FileBlob) session.getFile(path);
        // ... do something with the file
        // at the end delete the temporary file
        blob.getFile().delete();
```

We can do the same by invoking the `Blob.Get` operation.

```
// now test the GetBlob operation on the same blob
        blob = (FileBlob) session.newRequest("Blob.Get").setInput(doc).set(
                "xpath", "file:content").execute();
        // ... do something with the file
        // at the end delete the temporary file
        blob.getFile().delete();
```

Managing Complex Properties

Complex properties can have different levels complexity. This part of the documentation is about managing them with Automation Client API using

JSON format.

Let's see a complex property schema example:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:mc="http://nuxeo.org/schemas/dataset/"
           elementFormDefault="qualified"
           targetNamespace="http://nuxeo.org/schemas/dataset/">
...
    <xs:complexType name="field">
        <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="description" type="xs:string"/>
            <xs:element name="fieldType" type="mc:fieldType">

            </xs:element>
            <xs:element name="columnName" type="xs:string"/>
            <xs:element name="sqlTypeHint" type="xs:string"/>

            <xs:element name="roles">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="role" type="mc:roleType" minOccurs="0"
maxOccurs="unbounded"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
        </xs:sequence>
    </xs:complexType>
...
</xs:schema>
```

Let's see a JSON example of these complex property values:

creation.json

```
[
    {
        "fieldType": "string",
        "description": "desc field0",
        "roles": [
            "Decision",
            "Score"
        ],
        "name": "field0",
        "columnName": "col0",
        "sqlTypeHint": "whatever"
    },
    {
        "fieldType": "string",
        "description": "desc field1",
        "roles": [
            "Decision",
            "Score"
        ],
        "name": "field1",
        "columnName": "col1",
        "sqlTypeHint": "whatever"
    },
    {
        "fieldType": "string",
        "description": "desc field2",
        "roles": [
            "Decision",
            "Score"
        ],
        "name": "field2",
        "columnName": "col2",
        "sqlTypeHint": "whatever"
    },
    {
        "fieldType": "string",
        "description": "desc field3",
        "roles": [
            "Decision",
            "Score"
        ],
        "name": "field3",
        "columnName": "col3",
        "sqlTypeHint": "whatever"
    },
    {
        "fieldType": "string",
        "description": "desc field4",
        "roles": [
            "Decision",
            "Score"
        ],
        "name": "field4",
        "columnName": "col4",
        "sqlTypeHint": "whatever"
    }
]
```

Finally updating a document with the JSON values into its related metadata looks like this.

```
// Fetch the document
Document document = (Document)
session.newRequest(DocumentService.GetDocumentChild).setInput(new
PathRef("/")).set("name", "testDoc").execute();

// Send the fields representation as json

// Read the json file
File fieldAsJsonFile = FileUtils.getResourceFileFromContext("creation.json");
String fieldsDataAsJSon = FileUtils.readFile(fieldAsJsonFile);

// Don't forget to replace CRLF or LF
fieldsDataAsJSon = fieldsDataAsJSon.replaceAll("\n", "");
fieldsDataAsJSon = fieldsDataAsJSon.replaceAll("\r", "");

// Set the json values to the related metadata
document.set("ds:fields", fieldsDataAsJSon);

// Document Update
session.newRequest(UpdateDocument.ID).setInput(document).set("properties",document).ex
ecute();
```

Managing Business Objects

Since 5.7.2, managing business objects (Plain Old Java Object client side for mapping the Nuxeo Document Model Adapter (Documentation) server side) is now available.

Why manage business object? To manipulate business object to avoid *Nuxeo Document* manipulation on client side:

```
pull : BusinessObject (POJO) <----JSON---- DocumentModelAdapter <---- DocumentModel

push : BusinessObject (POJO) -----JSON---> DocumentModelAdapter ----> DocumentModel
```

Let's see an example.

1. Create a Nuxeo Document Model Adapter registered on server side and which should extend the `BusinessAdapter` class:

```
import org.nuxeo.ecm.automation.core.operations.business.adapter.BusinessAdapter;
import org.codehaus.jackson.annotate.JsonCreator;
import org.codehaus.jackson.annotate.JsonProperty;
import org.nuxeo.ecm.core.api.DocumentModel;

 /**
 * Document Model Adapter example server side
 */
public class BusinessBeanAdapter extends BusinessAdapter {

    private static final Log log = LogFactory.getLog(BusinessBeanAdapter.class);

    /**
     * Default constructor is needed for jackson mapping
     */
    public BusinessBeanAdapter() {
        super();
    }

    public BusinessBeanAdapter(DocumentModel documentModel) {
        super(documentModel);
    }

    public String getTitle() {
        try {
            return (String) getDocument().getPropertyValue("dc:title");
        } catch (ClientException e) {
            log.error("cannot get property title", e);
        }
        return null;
    }

    public void setTitle(String value) {
        try {
            getDocument().setPropertyValue("dc:title", value);
        } catch (ClientException e) {
            log.error("cannot set property title", e);
        }
    }

    public String getDescription() {
        try {
            return (String) getDocument().getPropertyValue("dc:description");
        } catch (ClientException e) {
            log.error("cannot get description property", e);
        }
        return null;
    }

    public void setDescription(String value) {
        try {
            getDocument().setPropertyValue("dc:description", value);
        } catch (ClientException e) {
            log.error("cannot set description property", e);
        }
    }
}
```

2. Declare a POJO on client side as following:

```
import org.nuxeo.ecm.automation.client.annotations.EntityType;

//Automation client File pojo example annotated with EntityType setting the
adapter server side built previously (just the simple name)
@EntityType("BusinessBeanAdapter")
public class BusinessBean {
    protected String title;
    protected String description;
    protected String id;
    protected String type;

    /**
    * this default contructor is mandatory for jackson mapping on the nuxeo
server side
    **/
    public BusinessBean() {
    }

    public BusinessBean(String title, String description, String type) {
        this.title = title;
        this.type = type;
        this.description = description;
    }
    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }
    public String getDescription() {
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
    }
    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
   public String getType() {
        return type;
    }
    public void setType(String type) {
        this.type = type;
    }
 }
```


We assume than the POJO client side stub is similar to the document model adapter stub server side.

So now we have on client side a POJO `BusinessBean` and on server side a Document Model Adapter `BusinessBeanAdapter`.

We are going to use two operations for creating/updating process:

- `Business.BusinessCreateOperation`

- `Business.BusinessUpdateOperation`

> Since 5.7.2, you can see the Automation Client Service Adapter section to know how Business operations are wrapped into `BusinessService` adapter and how to use it.

3. Let's see how to map them directly using these operations:

```
import org.nuxeo.ecm.automation.client.jaxrs.spi.JsonMarshalling;

// Let's instiante a BusinessBean 'File':
BusinessBean file = new BusinessBean("File", "File description", "File");

// BusinessBean 'File' Marshaller has to be registered in Automation Client
following this way:
JsonMarshalling.addMarshaller(PojoMarshaller.forClass(file.getClass()));

// Injecting the BusinessBean 'File' into the operation BusinessCreateOperation
let you create a document server side.
file = (BusinessBean)
session.newRequest("Operation.BusinessCreateOperation").setInput(file).set("name"
,file.getTitle()).set("parentPath","/").execute();

// Update document on server (pojo <-> adapter update) using
BusinessUpdateOperation
file.setTitle("Title Updated");
file = (BusinessBean)
session.newRequest("Operation.BusinessUpdateOperation").setInput(file).execute();
```

GitHub Nuxeo Automation Tests

Here is the complete code of the example. For more examples, see the unit tests in `nuxeo-automation-test` project.

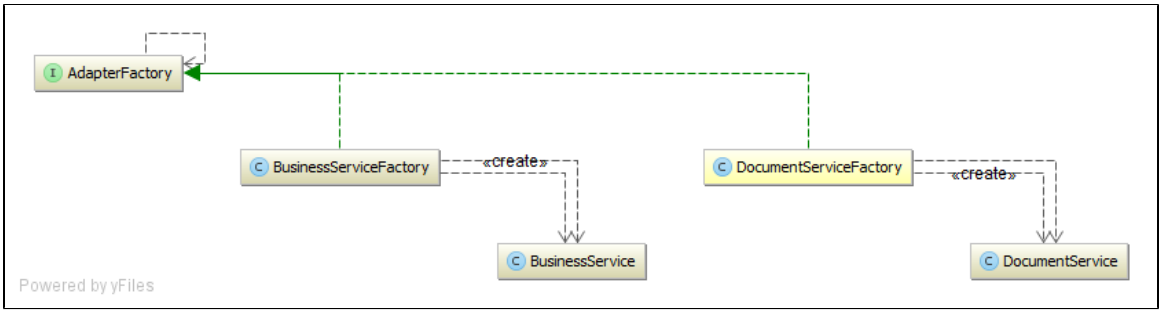
Automation Client Service Adapter

Default Service Adapter

Nuxeo Automation Client provides an "Adapter Service" to encapsulate business logic.

Default services provided:

- `DocumentService` - (Class on GitHub): This API provides access to basic operation like CRUD, lock, version...
- BusinessService (Class on Github): This API provides access to Business Operations.



Powered by yFiles
DocumentService Usage

Here is an example of using `DocumentService` API:

```
import org.nuxeo.ecm.automation.client.model.Document;
import org.nuxeo.ecm.automation.client.adapters.DocumentService;
import org.nuxeo.ecm.automation.client.Session;

// Test document creation
Document myDocument = new Document("myfolder2", "Folder");
myDocument.set("dc:title", "My Test Folder");
myDocument.set("dc:description", "test");
myDocument.set("dc:subjects", "a,b,c\\,d");

DocumentService documentService = session.getAdapter(DocumentService.class);

// Here is the way to create a document by using DocumentService
// Note you can put in first parameter, the Id or Path of the parent document
myDocument = documentService.createDocument(DocumentParent.getId(), myDocument);

// Fetch the document with its dublincore properties
// You can add several schemas like ...(myDocument, "dublincore", "yourSchema");
myDocument = documentService.getDocument(myDocument, "dublincore");

// Update the document title
myDocument.set("dc:title", "New Title");
documentService.update(myDocument);

// Fetch the document with all its properties (all schemas)
myDocument = documentService.getDocument(folder, "*");
```

BusinessService Usage (since 5.7.2)

Here is an example of using `BusinessService` API (this example is the same previous one with Business Objects but this time using `BusinessService`):

```
import org.nuxeo.ecm.automation.client.adapters.BusinessService;
import org.nuxeo.ecm.automation.client.Session;

// A simple custom POJO client side
BusinessBean file = new BusinessBean("File", "File description", "File");

// Fetching the business service adapter
BusinessService<BusinessBean> businessService =
session.getAdapter(BusinessService.class);

// Create server side injecting the POJO to the service method create
file = (BusinessBean) businessService.create(file, file.getTitle(), "/");

// Process to update with POJO
file.setTitle("Update");
file = (BusinessBean) businessService.update(file);
```

Contributing a New Custom Service Adapter

You can contribute your own adapter service by Java to encapsulate your own business logic.

To achieve this registration, you have to provide an adapter service factory that is going to instantiate your adapter. You can find an example of factory for the `DocumentService` class: `DocumentServiceFactory` (source code on GitHub).

Here is an example of registration:

```
HttpAutomationClient client = new
HttpAutomationClient("http://localhost:8080/nuxeo/site/automation");

// Register you own service factory
client.registerAdapter(new MyServiceFactory());

// And then retrieve your custom adapter service
MyService myService = session.getAdapter(MyService.class);
```

Automation Client in OSGi Environment

Since 5.7.1, Nuxeo Automation Client is OSGi compliant.
Example to Get Automation Client in an OSGi Environment

(Here we're using Apache Felix Framework)

```
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceReference;
import javax.inject.Inject;

@Inject
BundleContext bundleContext;

// Example with Felix ServiceReference
ServiceReference serviceReference =
bc.getServiceReference(AutomationClientFactory.class.getName());

// Fetch the Automation Client Factory
AutomationClientFactory factory = (AutomationClientFactory)
bundleContext.getService(serviceReference);

// Get the Automation Client with given URL of you server
AutomationClient client = factory.getClient(new
URL("http://localhost:8080/nuxeo/site/automation"));

// Get client with http connection timeout as parameter in milliseconds
client = factory.getClient(new URL("http://localhost:8080/nuxeo/site/automation"),
3600);
```

MANIFEST.MF information declared into the Automation Client bundle

```
Bundle-SymbolicName: org.nuxeo.ecm.automation.client

Bundle-Activator: org.nuxeo.ecm.automation.client.jaxrs.impl.AutomationClientActivator

Export-Package: org.nuxeo.ecm.automation.client.adapters;version="0.0.
 0.qualifier",org.nuxeo.ecm.automation.client;version="0.0.0.qualifier
 ",org.nuxeo.ecm.automation.client.model;version="0.0.0.qualifier"

Import-Package: javax.activation,javax.mail;version="1.4",javax.mail.i
 nternet;version="1.4",org.apache.commons.lang;version="2.6",org.apach
 e.commons.logging;version="1.1",org.apache.http;version="4.2",org.apa
 che.http.auth;version="4.2",org.apache.http.client;version="4.2",org.
 apache.http.client.methods;version="4.2",org.apache.http.conn;version
 ="4.2",org.apache.http.entity;version="4.2",org.apache.http.impl.clie
 nt;version="4.2",org.apache.http.impl.conn.tsccm;version="4.2",org.ap
 ache.http.params;version="4.2",org.apache.http.protocol;version="4.2"
 ,org.codehaus.jackson;version="1.8",org.codehaus.jackson.map;version=
 "1.8",org.codehaus.jackson.map.annotate;version="1.8",org.codehaus.ja
 ckson.map.deser;version="1.8",org.codehaus.jackson.map.introspect;ver
 sion="1.8",org.codehaus.jackson.map.module;version="1.8",org.codehaus
 .jackson.map.type;version="1.8",org.codehaus.jackson.type;version="1.
 8",org.nuxeo.ecm.automation.client;version="0.0",org.nuxeo.ecm.automa
 tion.client.adapters;version="0.0",org.nuxeo.ecm.automation.client.mo
 del;version="0.0",org.osgi.framework;version="1.5"
```

PHP Automation Client

A PHP automation client is made available on GitHub. You can use it and ask for commit rights on the project if you want to improve it or fix a bug. The project contains the library and some sample use cases.

Queries / Chains

In this examples we are using the PHP automation client to demonstrate how to invoke remote operations.
The following example is executing a simple query against a remote server: `SELECT * FROM Document`. The server will return a JSON document listing all selected documents.

In this section

```
$client = new PhpAutomationClient('http://localhost:8080/nuxeo/site/automation');

$session = $client->getSession('Administrator','Administrator');

$answer = $session->newRequest("Document.Query")->set('params', 'query', "SELECT *
FROM Document" )->sendRequest();

$documentsArray = $answer->getDocumentList();
$value = sizeof($documentsArray);
echo '<table>';
echo '<tr><TH>uid</TH><TH>Path</TH>
<TH>Type</TH><TH>State</TH><TH>Title</TH><TH>Download as PDF</TH>';
for ($test = 0; $test < $value; $test ++){
    echo '<tr>';
    echo '<td> ' . current($documentsArray)->getUid()  . '</td>';
    echo '<td> ' . current($documentsArray)->getPath()  . '</td>';
    echo '<td> ' . current($documentsArray)->getType()  . '</td>';
    echo '<td> ' . current($documentsArray)->getState()  . '</td>';
    echo '<td> ' . current($documentsArray)->getTitle()  . '</td>';
    echo '<td><form id="test" action="../tests/B5bis.php" method="post" >';
    echo '<input type="hidden" name="data" value="'.
    current($documentsArray)->getPath(). '"/>';
    echo '<input type="submit" value="download"/>';
    echo '</form></td></tr>';
    next($documentsArray);
}
echo '</table>';
```

The class `phpAutomationClient` allows you to open a session with the `getSession` (return a session instance). Then, from the session, you can create a new request by using the same named function. The set function is used to configure your automation request, giving the chain or operation to call as well as the loading params, context, and input parts. At last, you send the request with the `sendRequest` function.

You can see here how to use the getters in order to retrieve information from the Document object, built from the request answer.

Using Blobs

Attach Blob

In order to attach a blob, we have to send a Multipart Request to Nuxeo. The first part of the request will contain the body of the request (params, context ...) and the second part will contain the blob (as an input).

```
$client = new PhpAutomationClient('http://localhost:8080/nuxeo/site/automation');

$session = $client->getSession('Administrator','Administrator');

$answer = $session->newRequest("Blob.Attach")->set('params', 'document', $path)
->loadBlob($blob, $blobtype)
->sendRequest();
```

That will attach the blob to an existing file. In order to send a blob, you use the `loadBlob()` function. If a blob is loaded using this function, the `sendRequet()` function will automatically create a multipart request. If you load many blobs without noticing a precise localization in params, the blobs will be send as an attachment of the file (not as a content).

Get a Blob

In order to get a blog, you have to read the content of the 'tempfile' after using the appropriate headers.

```
$client = new PhpAutomationClient('http://localhost:8080/nuxeo/site/automation');

$session = $client->GetSession('Administrator','Administrator');

$answer = $session->NewRequest("Blob.Get")->Set('input', 'doc: ' .
$path)->SendRequest();

if (!isset($answer) OR $answer == false)
    echo '$answer is not set';
else{
    header('Content-Description: File Transfer');
    header('Content-Type: application/octet-stream');
    header('Content-Disposition: attachment; filename='.$filename.'.pdf');
    readfile('tempstream');
}
```

This will download the blob placed in `file:content` of the Nuxeo file designed by `$path`.

Using a Python Client

We have developed a small Python library that implements the main functions of the JSON-RPC API. You can check out the blog post A sample Python library for the Nuxeo Content Automation JSON-RPC API.
Alternatively you can use the standard library of Python to access a Nuxeo repository using the Content Automation API. Here is a worked example with screencast that demonstrates how to deploy a custom server side operation developed in Java using the Nuxeo IDE and a client Python script that calls it: Exploring Nuxeo APIs: Content Automation.

Here is an example building a document query using Python:

```
#!/usr/bin/env python

import urllib2, base64

QUERY_URL = "http://localhost:8080/nuxeo/site/automation/Document.Query"
USER = 'Administrator'
PASSWD = 'Administrator'

auth = 'Basic %s' % base64.b64encode(USER + ":" + PASSWD).strip()
headers = {
   "Content-Type": "application/json+nxrequest",
   "Authorization": auth}
data = '{params: {"query": "SELECT * FROM Document"}, context : {}}'
req = urllib2.Request(QUERY_URL, data, headers)

resp = urllib2.urlopen(req)

print resp.read()
```

Here's a slightly more involved example, that illustrates:

- How to use a `HTTPCookieProcessor` for keeping session state,
- The use of the `input` parameter,
- A few different document-related commands (`Document.Query`, `Document.Create`, `Document.Delete`).

```
#!/usr/bin/env python

import urllib2, base64, sys
import simplejson as json
from pprint import pprint
```

```
URL = "http://localhost:8080/nuxeo/site/automation/"
USER = 'Administrator'
PASSWD = 'Administrator'

cookie_processor = urllib2.HTTPCookieProcessor()
opener = urllib2.build_opener(cookie_processor)
urllib2.install_opener(opener)

def execute(command, input=None, **params):
    auth = 'Basic %s' % base64.b64encode(USER + ":" + PASSWD).strip()
    headers = {
        "Content-Type": "application/json+nxrequest",
        "Authorization": auth}
    d = {}
    if params:
        d['params'] = params
    if input:
        d['input'] = input
    if d:
        data = json.dumps(d)
    else:
        data = None
    req = urllib2.Request(URL + command, data, headers)

    try:
        resp = opener.open(req)
    except Exception, e:
        exc = json.load(e.fp)
        print exc['message']
        print exc['stack']
        sys.exit()
    s = resp.read()
    if s:
        return json.loads(s)
    else:
        return None

print "All automation commands:"
print
for op in execute("")['operations']:
    pprint(op)
    pprint
print

print "All documents in the repository:"
print
for doc in execute("Document.Query", query="SELECT * FROM Document")['entries']:
    print doc['path'], ":", doc['type']
print

print "Fetch workspaces root:"
doc = execute("Document.Fetch", value="/default-domain/workspaces")
pprint(doc)
print

print "Create a new doc:"
new_doc = execute("Document.Create", input="doc:" + doc['uid'], type="Workspace",
name="MyWS",
```

```
        properties="dc:title=My new workspace")
print

print "Delete new doc:"
```

```
execute("Document.Delete", input="doc:" + new_doc['uid'])
print
```

Client API Test suite (TCK)

This page is a work in progress.

This chapter provides a test suite that can be used to test the implementation of an automation client library.

Automation Client Test Suite

The test suite is separated in three sub-parts:

- Basic CRUD operations on documents,
- Blobs management,
- Marshaling extensions.

Only the first part is strictly necessary.

Base CRUD

Create and Read Documents

Scenario

1. Create a Folder on / (`Document.Create`).
2. Create a File child (`Document.Create`).
3. Create a File child (`Document.Create`).
4. Update `dc:description` property on second child (`Document.SetProperty`).
5. Update `dc:subjects` property on second child (`Document.SetProperty`).
6. Get Children of Folder (`Document.GetChildren`).
7. Specify that `dublincore` schema should be fetched.
8. Verify that:
    - the folder has two children,
    - the second child has the correct `dc:description` and `dc:subjects`.

Java Implementation

Github Link for CRUD operations tests suite

In this list

JavaScript Implementation

Link JS
HTTP Captures

HTTP Signature - Document.Create Expand source

```
POST /nuxeo/site/automation/Document.Create HTTP/1.1
Host: 127.0.0.1:8080
Content-Length: 130
Authorization: Basic QWRtaW5pc3RyYXRvcjpBZG1pbmlzdHJhdG9y
Origin: http://127.0.0.1:8080
Content-Type: application/json+nxrequest
Accept: */*
X-NXDocumentProperties: dublincore
X-Requested-With: XMLHttpRequest
X-NXVoidOperation: false
Nuxeo-Transaction-Timeout: 35
{ "input" : "doc:/",
  "params" : { "name" : "TestDocs",
      "properties" : "dc:title=Test Docs \ndc:description=Simple container",
      "type" : "Folder"
    }
}



HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: application/json+nxentity
{ "changeToken" : "1372718688038",
  "contextParameters" : {  },
  "entity-type" : "document",
  "facets" : [ "Folderish" ],
  "lastModified" : "2013-07-01T22:44:48.03Z",
  "path" : "/TestDocs",
  "properties" : { "dc:contributors" : [ "Administrator" ],
      "dc:coverage" : null,
      "dc:created" : "2013-07-01T22:44:48.03Z",
      "dc:creator" : "Administrator",
      "dc:description" : "Simple container",
      "dc:expired" : null,
      "dc:format" : null,
      "dc:issued" : null,
      "dc:language" : null,
      "dc:lastContributor" : "Administrator",
      "dc:modified" : "2013-07-01T22:44:48.03Z",
      "dc:nature" : null,
      "dc:publisher" : null,
      "dc:rights" : null,
      "dc:source" : null,
      "dc:subjects" : [  ],
      "dc:title" : "Test Docs",
      "dc:valid" : null
    },
  "repository" : "default",
  "state" : "project",
  "title" : "Test Docs",
  "type" : "Folder",
  "uid" : "e27bc86b-d5f1-4ba9-a8e4-3ce96340f301",
```

```
  "versionLabel" : ""
}

POST /nuxeo/site/automation/Document.Create HTTP/1.1
Host: 127.0.0.1:8080
Content-Length: 69
Authorization: Basic QWRtaW5pc3RyYXRvcjpBZG1pbmlzdHJhdG9y
Origin: http://127.0.0.1:8080
Content-Type: application/json+nxrequest
Accept: */*
X-NXDocumentProperties: dublincore
X-Requested-With: XMLHttpRequest
X-NXVoidOperation: false
Nuxeo-Transaction-Timeout: 35
{ "input" : "doc:/TestDocs",
  "params" : { "name" : "TestFile1",
      "type" : "File"
    }
}

HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: application/json+nxentity
{ "changeToken" : "1372718688058",
  "contextParameters" : {  },
  "entity-type" : "document",
  "facets" : [ "Downloadable",
      "Commentable",
      "Versionable",
      "Publishable",
      "Thumbnail",
      "HasRelatedText"
    ],
  "lastModified" : "2013-07-01T22:44:48.05Z",
  "path" : "/TestDocs/TestFile1",
  "properties" : { "dc:contributors" : [ "Administrator" ],
      "dc:coverage" : null,
      "dc:created" : "2013-07-01T22:44:48.05Z",
      "dc:creator" : "Administrator",
      "dc:description" : null,
      "dc:expired" : null,
      "dc:format" : null,
      "dc:issued" : null,
      "dc:language" : null,
      "dc:lastContributor" : "Administrator",
      "dc:modified" : "2013-07-01T22:44:48.05Z",
      "dc:nature" : null,
      "dc:publisher" : null,
      "dc:rights" : null,
      "dc:source" : null,
      "dc:subjects" : [  ],
      "dc:title" : null,
      "dc:valid" : null
    },
  "repository" : "default",
  "state" : "project",
  "title" : "TestFile1",
  "type" : "File",
  "uid" : "29e654b3-02ae-4600-a057-d487fbac8fbd",
```

```
  "versionLabel" : "0.0"
}

POST /nuxeo/site/automation/Document.Create HTTP/1.1
Host: 127.0.0.1:8080
Content-Length: 69
Authorization: Basic QWRtaW5pc3RyYXRvcjpBZG1pbmlzdHJhdG9y
Origin: http://127.0.0.1:8080
Content-Type: application/json+nxrequest
Accept: */*
X-NXDocumentProperties: dublincore
X-Requested-With: XMLHttpRequest
X-NXVoidOperation: false
Nuxeo-Transaction-Timeout: 35
{ "input" : "doc:/TestDocs",
  "params" : { "name" : "TestFile2",
      "type" : "File"
    }
}

HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: application/json+nxentity
{ "changeToken" : "1372718688083",
  "contextParameters" : {  },
  "entity-type" : "document",
  "facets" : [ "Downloadable",
      "Commentable",
      "Versionable",
      "Publishable",
      "Thumbnail",
      "HasRelatedText"
    ],
  "lastModified" : "2013-07-01T22:44:48.08Z",
  "path" : "/TestDocs/TestFile2",
  "properties" : { "dc:contributors" : [ "Administrator" ],
      "dc:coverage" : null,
      "dc:created" : "2013-07-01T22:44:48.08Z",
      "dc:creator" : "Administrator",
      "dc:description" : null,
      "dc:expired" : null,
      "dc:format" : null,
      "dc:issued" : null,
      "dc:language" : null,
      "dc:lastContributor" : "Administrator",
      "dc:modified" : "2013-07-01T22:44:48.08Z",
      "dc:nature" : null,
      "dc:publisher" : null,
      "dc:rights" : null,
      "dc:source" : null,
      "dc:subjects" : [  ],
      "dc:title" : null,
      "dc:valid" : null
    },
  "repository" : "default",
  "state" : "project",
  "title" : "TestFile2",
  "type" : "File",
```

```
  "uid" : "7493912e-5d59-4041-9efc-a907c1d1fa07",
  "versionLabel" : "0.0"
}
```

HTTP Signature - Document.Update Expand source

```
POST /nuxeo/site/automation/Document.Update HTTP/1.1
Host: 127.0.0.1:8080
Content-Length: 133
Authorization: Basic QWRtaW5pc3RyYXRvcjpBZG1pbmlzdHJhdG9y
Origin: http://127.0.0.1:8080
Content-Type: application/json+nxrequest
Accept: */*
X-NXDocumentProperties: dublincore
X-Requested-With: XMLHttpRequest
X-NXVoidOperation: false
Nuxeo-Transaction-Timeout: 35
{ "input" : "doc:/TestDocs/TestFile2",
  "params" : { "properties" : "dc:description=Simple
File\ndc:subjects=subject1,subject2",
      "save" : "true"
    }
}

HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: application/json+nxentity
{ "changeToken" : "1372718688104",
  "contextParameters" : {  },
  "entity-type" : "document",
  "facets" : [ "Downloadable",
      "Commentable",
      "Versionable",
      "Publishable",
      "Thumbnail",
      "HasRelatedText"
    ],
  "lastModified" : "2013-07-01T22:44:48.10Z",
  "path" : "/TestDocs/TestFile2",
  "properties" : { "dc:contributors" : [ "Administrator" ],
      "dc:coverage" : null,
      "dc:created" : "2013-07-01T22:44:48.08Z",
      "dc:creator" : "Administrator",
      "dc:description" : "Simple File",
      "dc:expired" : null,
      "dc:format" : null,
      "dc:issued" : null,
      "dc:language" : null,
      "dc:lastContributor" : "Administrator",
      "dc:modified" : "2013-07-01T22:44:48.10Z",
      "dc:nature" : null,
      "dc:publisher" : null,
      "dc:rights" : null,
      "dc:source" : null,
      "dc:subjects" : [ "subject1",
          "subject2"
```

```
      ],
    "dc:title" : null,
    "dc:valid" : null
  },
"repository" : "default",
"state" : "project",
"title" : "TestFile2",
"type" : "File",
"uid" : "7493912e-5d59-4041-9efc-a907c1d1fa07",
```

```
  "versionLabel" : "0.0"
}
```

HTTP Signature - Document.Fetch Expand source

```
POST /nuxeo/site/automation/Document.Fetch HTTP/1.1
Authorization: Basic QWRtaW5pc3RyYXRvcjpBZG1pbmlzdHJhdG9y
Content-Type: application/json+nxrequest
Accept: application/json+nxentity, */*
Content-Length: 48
Host: localhost:8080
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.2.4 (java 1.5)

{"params":{"value":"/TestFolder1"},"context":{}}

HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: application/json+nxentity
Transfer-Encoding: chunked
Date: Thu, 25 Jul 2013 14:20:03 GMT
{
  "entity-type": "document",
  "repository": "default",
  "uid": "6879c9c5-0fdf-4d53-85d0-32c8fa94a9e0",
  "path": "/TestFolder1",
  "type": "Folder",
  "state": "project",
  "versionLabel": "",
  "title": "Test Folder2",
  "lastModified": "2013-07-25T14:20:03.90Z",
  "facets": [
    "Folderish"
  ],
  "changeToken": "1374762003906",
  "contextParameters": {}
}
```

HTTP Signature - Document.GetChildren Expand source

```
POST /nuxeo/site/automation/Document.GetChildren HTTP/1.1
Authorization: Basic QWRtaW5pc3RyYXRvcjpBZG1pbmlzdHJhdG9y
Content-Type: application/json+nxrequest
Accept: application/json+nxentity, */*
Content-Length: 82
Host: localhost:8080
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.2.4 (java 1.5)

{
  "input": {
    "entity-type": "string",
    "value": "/TestFolder1"
  },
```

```
  "params": {},
  "context": {}
}

HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: application/json+nxentity
Transfer-Encoding: chunked
Date: Thu, 25 Jul 2013 14:20:03 GMT
{
  "entity-type": "documents",
  "entries": [
    {
      "entity-type": "document",
      "repository": "default",
      "uid": "dc499ced-e140-46e6-a5c4-58fbc19b589c",
      "path": "/TestFolder1/TestFile1",
      "type": "File",
      "state": "project",
      "versionLabel": "0.0",
      "title": "TestFile1",
      "lastModified": "2013-07-25T14:20:03.95Z",
      "facets": [
        "Downloadable",
        "Commentable",
        "Versionable",
        "Publishable",
        "Thumbnail",
        "HasRelatedText"
      ],
      "changeToken": "1374762003951",
      "contextParameters": {}
    },
    {
      "entity-type": "document",
      "repository": "default",
      "uid": "98181b11-ef91-48fa-b202-091f5d2ee809",
      "path": "/TestFolder1/TestFile2",
      "type": "File",
      "state": "project",
      "versionLabel": "0.0",
      "title": "TestFile2",
      "lastModified": "2013-07-25T14:20:04.04Z",
      "facets": [
        "Downloadable",
        "Commentable",
        "Versionable",
        "Publishable",
        "Thumbnail",
        "HasRelatedText"
      ],
      "changeToken": "1374762004043",
      "contextParameters": {}
```

```
    }
  ]
}
```

Pagination

Scenario

1. Create a Folder on / (`Document.Create`).
2. Create three Files children.
3. Call `Document.PageProvider` with "`select * from Document where ecm:parentId = ?`" with
   - queryParams = Folder uuid
   - pageSize = 2
   - page = 1
4. Verify:
   - pageCount=2
   - the page contains two docs.
5. Call `Document.PageProvider` with "`select * from Document where ecm:parentId = ?`" with
   - queryParams = Folder uuid
   - pageSize = 2
   - page = 2
6. Verify:
   - pageCount=2
   - the page contains one document.

Java Implementation

GitHub Link for Pagination operations tests suite

HTTP Capture

HTTP Signature - Document.PageProvider Expand source

```
 POST /nuxeo/site/automation/Document.PageProvider HTTP/1.1
Authorization: Basic QWRtaW5pc3RyYXRvcjpBZG1pbmlzdHJhdG9y
Content-Type: application/json+nxrequest
Accept: application/json+nxentity, */*
Content-Length: 160
Host: localhost:8080
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.2.4 (java 1.5)

{
  "params": {
    "queryParams": "b583b0a7-cae4-4961-b94d-29c469ca8012",
    "page": "0",
    "query": "select * from Document where ecm:parentId = ?",
    "pageSize": "2"
  },
  "context": {}
}

HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: application/json+nxentity
Transfer-Encoding: chunked
Date: Thu, 25 Jul 2013 15:24:47 GMT
{
  "entity-type": "documents",
  "isPaginable": true,
  "totalSize": 3,
  "pageIndex": 0,
  "pageSize": 2,
```

```
  "pageCount": 2,
  "entries": [
    {
      "entity-type": "document",
      "repository": "default",
      "uid": "3f76a415-ad73-4522-9450-d12af25b7fb4",
      "path": "/TestFolder2/TestFile3",
      "type": "File",
      "state": "project",
      "versionLabel": "0.0",
      "title": "TestFile3",
      "lastModified": "2013-07-25T15:23:58.63Z",
      "facets": [
        "Downloadable",
        "Commentable",
        "Versionable",
        "Publishable",
        "Thumbnail",
        "HasRelatedText"
      ],
      "changeToken": "1374765838639",
      "contextParameters": {
        "documentURL":
"/nuxeo/nxdoc/default/3f76a415-ad73-4522-9450-d12af25b7fb4/view_documents"
      }
    },
    {
      "entity-type": "document",
      "repository": "default",
      "uid": "1e751ded-c659-46da-8982-0963575f48f8",
      "path": "/TestFolder2/TestFile1",
      "type": "File",
      "state": "project",
      "versionLabel": "0.0",
      "title": "TestFile1",
      "lastModified": "2013-07-25T15:23:58.55Z",
      "facets": [
        "Downloadable",
        "Commentable",
        "Versionable",
        "Publishable",
        "Thumbnail",
        "HasRelatedText"
      ],
      "changeToken": "1374765838558",
      "contextParameters": {
        "documentURL":
"/nuxeo/nxdoc/default/1e751ded-c659-46da-8982-0963575f48f8/view_documents"
      }
```

```
    }
  ]
}
```

Blobs

Direct Blob Upload

Scenario

1. Create a Folder on / (`Document.Create`).
2. Call `FileManager.Import` passing a testText.txt file and setting context to Folder.
3. Call `FileManager.Import` passing a testBlob.bin binary file and setting context to Folder.
4. Verify that:
    a. two documents are created, of types File and Note.

Java Implementation

GitHub link for Blob operations tests suite
HTTP Capture

HTTP Signature - FileManager.Import Expand source

```
POST /nuxeo/site/automation/FileManager.Import HTTP/1.1
Authorization: Basic QWRtaW5pc3RyYXRvcjpBZG1pbmlzdHJhdG9y
Content-Type: multipart/related;
    boundary="----=_Part_0_1227289322.1374766516160"
Accept: application/json+nxentity, */*
Transfer-Encoding: chunked
Host: localhost:8080
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.2.4 (java 1.5)
e7
------=_Part_0_1227289322.1374766516160
Content-Type: application/json+nxrequest; charset=UTF-8
Content-Transfer-Encoding: 8bit
Content-ID: request
Content-Length: 57
{"params":{},"context":{"currentDocument":"/FolderBlob"}}
f5
------=_Part_0_1227289322.1374766516160
Content-Type: text/xml
Content-Transfer-Encoding: binary
Content-Disposition: attachment;
    filename=automation-test-5983650304333926591.xml
Content-ID: input
Content-Length: 16
<doc>mydoc</doc>
------=_Part_0_1227289322.1374766516160--

HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: application/json+nxentity
Transfer-Encoding: chunked
Date: Thu, 25 Jul 2013 15:35:16 GMT
{
  "entity-type": "document",
  "repository": "default",
  "uid": "77ec49fb-1832-4124-b8f6-270d53ac20bb",
  "path": "/FolderBlob/automation-test-59836503",
  "type": "File",
  "state": "project",
  "versionLabel": "0.0",
  "title": "automation-test-5983650304333926591.xml",
  "lastModified": "2013-07-25T15:35:16.24Z",
  "facets": [
    "Downloadable",
    "Commentable",
    "Versionable",
    "Publishable",
    "Thumbnail",
    "HasRelatedText"
  ],
  "changeToken": "1374766516241",
  "contextParameters": {}
}
```

HTTP Signature - Blob.Attach Expand source

```
POST /nuxeo/site/automation/Blob.Attach HTTP/1.1
X-NXVoidOperation: true
Authorization: Basic QWRtaW5pc3RyYXRvcjpBZG1pbmlzdHJhdG9y
Content-Type: multipart/related;
    boundary="----=_Part_1_2072469418.1374766516423"
Accept: application/json+nxentity, */*
Transfer-Encoding: chunked
Host: localhost:8080
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.2.4 (java 1.5)

------=_Part_1_2072469418.1374766516423
Content-Type: application/json+nxrequest; charset=UTF-8
Content-Transfer-Encoding: 8bit
Content-ID: request
Content-Length: 46

{"params":{"document":"/myfile"},"context":{}}

------=_Part_1_2072469418.1374766516423
Content-Type: image/jpeg
Content-Transfer-Encoding: binary
Content-Disposition: attachment; filename=creationFields.json
Content-ID: input
Content-Length: 1288
[
    {
        "fieldType": "string",
        "description": "desc field0",
        "roles": [
            "Decision",
            "Score"
        ],
        "name": "field0",
        "columnName": "col0",
        "sqlTypeHint": "whatever"
    },
    {
        "fieldType": "string",
        "description": "desc field1",
        "roles": [
            "Decision",
            "Score"
        ],
        "name": "field1",
        "columnName": "col1",
        "sqlTypeHint": "whatever"
    },
    {
        "fieldType": "string",
        "description": "desc field2",
        "roles": [
            "Decision",
            "Score"
        ],
        "name": "field2",
```

```
        "columnName": "col2",
        "sqlTypeHint": "whatever"
    },
    {
        "fieldType": "string",
        "description": "desc field3",
        "roles": [
            "Decision",
            "Score"
        ],
        "name": "field3",
        "columnName": "col3",
        "sqlTypeHint": "whatever"
    },
    {
        "fieldType": "string",
        "description": "desc field4",
        "roles": [
            "Decision",
            "Score"
        ],
        "name": "field4",
        "columnName": "col4",
        "sqlTypeHint": "whatever"
    }
]
------=_Part_1_2072469418.1374766516423--
```

```
HTTP/1.1 204 No Content
Server: Apache-Coyote/1.1
Date: Thu, 25 Jul 2013 15:35:16 GMT
```

Blobs Upload via Batch Manager

Scenario

1. Upload two blobs via batch manager:
   - testText.txt,
   - testBlob.bin.
2. Call `FileManager.Import` via batch API:
   - set currentDocument to /
3. Verify that:
   - two documents are created, of types File and Note.

Java Implementation

XXX Link the Java unit test
HTTP Capture

XXX Commented HTTP capture

Blobs Attach via Batch Manager

Scenario

1. Create a File on / (`Document.Create`).
2. Upload one blob via batch manager:
   a. testText.txt.
3. Call `Document.Update` with two properties:
   - `dc:description`
   - `file:content` : reference Blob 1 from batch.
4. Fetch the Document:
   - Specify to fetch the `dublincore` schema.
5. Download the blob.
6. Verify that:
   - Description was updated.
   - Blob exists and contains the expected text.

Java Implementation

XXX Link the Java unit test
HTTP Capture

XXX Commented Http capture

Marshaling Extensions

Manage Complex Properties

Scenario

1. Create a File on / (`Document.Create`).
2. Call `Document.Update`:
   a. `dc:description` ,
   b. `dc:subjects` ,
   c. some complex property.
3. Fetch the Document:
   a. specify to fetch all schemas.
4. Verify that:
   a. document content is ok (scalar, list and complex).

Java Implementation

Github link for Complex Properties operations tests suite
HTTP Capture

HTTP Signature - Complex Properties - Document.Create Expand source

```
POST /nuxeo/site/automation/Document.Create HTTP/1.1
Authorization: Basic QWRtaW5pc3RyYXRvcjpBZG1pbmlzdHJhdG9y
```

```
Content-Type: application/json+nxrequest
Accept: application/json+nxentity, */*
Content-Length: 1484
Host: localhost:8080
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.2.4 (java 1.5)

{
  "input": "doc:c3fbb3bd-8646-4b81-9380-63da531856e8",
  "params": {
    "name": "testDoc",
    "properties": "ds:fields=[    {        \"fieldType\": \"string\",
\"description\": \"desc field0\",        \"roles\": [            \"Decision\",
\"Score\"        ],        \"name\": \"field0\",        \"columnName\": \"col0\",
\"sqlTypeHint\": \"whatever\"    },    {        \"fieldType\": \"string\",
\"description\": \"desc field1\",        \"roles\": [            \"Decision\",
\"Score\"        ],        \"name\": \"field1\",        \"columnName\": \"col1\",
\"sqlTypeHint\": \"whatever\"    },    {        \"fieldType\": \"string\",
\"description\": \"desc field2\",        \"roles\": [            \"Decision\",
\"Score\"        ],        \"name\": \"field2\",        \"columnName\": \"col2\",
\"sqlTypeHint\": \"whatever\"    },    {        \"fieldType\": \"string\",
\"description\": \"desc field3\",        \"roles\": [            \"Decision\",
\"Score\"        ],        \"name\": \"field3\",        \"columnName\": \"col3\",
\"sqlTypeHint\": \"whatever\"    },    {        \"fieldType\": \"string\",
\"description\": \"desc field4\",        \"roles\": [            \"Decision\",
\"Score\"        ],        \"name\": \"field4\",        \"columnName\": \"col4\",
\"sqlTypeHint\": \"whatever\"    }]\nds:tableName=MyTable\ndc:title=testDoc\n",
    "type": "DataSet"
  },
  "context": {}
}

HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: application/json+nxentity
Transfer-Encoding: chunked
Date: Thu, 25 Jul 2013 15:45:25 GMT
{
  "entity-type": "document",
  "repository": "default",
  "uid": "ad9f8940-6aa8-450f-80fd-05af3f4836fb",
  "path": "/testDoc",
  "type": "DataSet",
  "state": "undefined",
  "versionLabel": "0.0",
  "title": "testDoc",
  "lastModified": "2013-07-25T15:45:25.03Z",
  "facets": [
    "Downloadable",
    "Commentable",
    "Versionable",
    "Publishable",
    "Thumbnail",
    "HasRelatedText"
  ],
```

```
  "changeToken": "1374767125039",
  "contextParameters": {}
}
```

HTTP Signature - Complex Properties - Document.Update Expand source

```
POST /nuxeo/site/automation/Document.Update HTTP/1.1
Authorization: Basic QWRtaW5pc3RyYXRvcjpBZG1pbmlzdHJhdG9y
Content-Type: application/json+nxrequest
Accept: application/json+nxentity, */*
Content-Length: 465
Host: localhost:8080
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.2.4 (java 1.5)

{
  "input": "doc:ad9f8940-6aa8-450f-80fd-05af3f4836fb",
  "params": {
    "properties": "ds:fields=[    {        \"fieldType\":\"string\",
\"description\":\"desc fieldA\",        \"name\":\"fieldA\",
\"columnName\":\"colA\",        \"sqlTypeHint\":\"whatever\"    },    {
\"fieldType\":\"string\",        \"description\":\"desc fieldB\",
\"name\":\"fieldB\",        \"columnName\":\"colB\",
\"sqlTypeHint\":\"whatever\"    }]\n"
  },
  "context": {}
}

HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: application/json+nxentity
Transfer-Encoding: chunked
Date: Thu, 25 Jul 2013 15:45:25 GMT
{
  "entity-type": "document",
  "repository": "default",
  "uid": "ad9f8940-6aa8-450f-80fd-05af3f4836fb",
  "path": "/testDoc",
  "type": "DataSet",
  "state": "undefined",
  "versionLabel": "0.0",
  "title": "testDoc",
  "lastModified": "2013-07-25T15:45:25.08Z",
  "facets": [
    "Downloadable",
    "Commentable",
    "Versionable",
    "Publishable",
    "Thumbnail",
    "HasRelatedText"
  ],
  "changeToken": "1374767125085",
  "contextParameters": {}
}
```

Custom Marshaling

Scenario

1. Create a Folder on / (`Document.Create`).
2. Create three Files children.
3. Call `Resultset.PageProvider` with "`select dc:title , dc:desciption , ecm:type from Document where ecm:parentId = ?`" with:
    a. `queryParams` = Folder uuid,
    b. `pageSize` = 2,
    c. `page` = 1.
4. Verify that:
    a. `pageCount=2`
    b. the page contains two records,
    c. the content of returned recordset is correct.

Java Implementation

XXX Link the Java unit test
HTTP Capture

XXX Commented HTTP capture

Managing Business Objects

Scenario

1. Create a client side 'Business' bean (simple POJO with getter/setter on properties).
2. Register the POJO in client Marshaling engine.
3. Call `Business.BusinessCreateOperation` with:
    - two properties:
        - `name` ,
        - `parentPath` : reference the parent of the document to create,
    - one input:
        - the POJO itself.
4. Get the returned POJO (with id set).
5. Update the POJO title.
6. Call `Business.BusinessUpdateOperation` with:
    - one property:
        - id: references the document id to update
    - with one input:
        - the POJO itself.
7. Verify that:
    - the id is not null,
    - the title was updated.

Java Implementation

GitHub link for Business Object operations tests suite
HTTP Capture

HTTP Signature - Business.BusinessCreateOperation Expand source

```
 POST /nuxeo/site/automation/Business.BusinessCreateOperation HTTP/1.1
Authorization: Basic QWRtaW5pc3RyYXRvcjpBZG1pbmlzdHJhdG9y
Content-Type: application/json+nxrequest
Accept: application/json+nxentity, */*
Content-Length: 206
Host: localhost:8080
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.2.4 (java 1.5)

{
  "input": {
    "entity-type": "BusinessBeanAdapter",
    "value": {
      "id": null,
      "type": "Note",
      "description": "File description",
      "title": "Note",
      "note": "Note Content"
    }
  },
  "params": {
    "name": "Note",
    "parentPath": "/"
  },
  "context": {}
}

HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: application/json+nxentity
Transfer-Encoding: chunked
Date: Thu, 25 Jul 2013 15:45:45 GMT
{
  "entity-type": "BusinessBeanAdapter",
  "value": {
    "type": "Note",
    "id": "d82d9824-20ca-402a-a586-790103a018b2",
    "description": "File description",
    "note": "Note Content",
    "title": "Note"
  }
}
```

HTTP Signature - Business.BusinessUpdateOperation Expand source

```
 POST /nuxeo/site/automation/Business.BusinessUpdateOperation HTTP/1.1
Authorization: Basic QWRtaW5pc3RyYXRvcjpBZG1pbmlzdHJhdG9y
Content-Type: application/json+nxrequest
Accept: application/json+nxentity, */*
Content-Length: 212
Host: localhost:8080
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.2.4 (java 1.5)

{
  "input": {
    "entity-type": "BusinessBeanAdapter",
    "value": {
      "id": "d82d9824-20ca-402a-a586-790103a018b2",
      "type": "Note",
      "description": "File description",
      "title": "Update",
      "note": "Note Content"
    }
  },
  "params": {},
  "context": {}
}

HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: application/json+nxentity
Transfer-Encoding: chunked
Date: Thu, 25 Jul 2013 15:45:45 GMT
{
  "entity-type": "BusinessBeanAdapter",
  "value": {
    "type": "Note",
    "id": "d82d9824-20ca-402a-a586-790103a018b2",
    "description": "File description",
    "note": "Note Content",
    "title": "Update"
  }
}
```

Client Compatibility Matrix

Compatibility Level

This compatibility matrix focus on two aspects :

- **protocol compliance**: capability to achieve simple usage scenario as defined inside the tests,
- **convenience features**: additional conveniant features that may be provided on top of JSON-RPC library.

Protocol Compliance

Automation being a simple JSON-RPC, you should be able to do everything from every language as long as you can do JSON Marshaling and HTTP calls.

Hence for each test scenario, the target client library will be qualified with one of the three statuses:

- **Ok**: built-in feature,

- **Easy**: test can be achieved with minor tweak (like format JSON partially by hand ...),
- **Fail**: feature is not possible or complex to use.

Convenience Features

This aspect is here to highlight the specific feature that may be provided by each client library implementation.

This feature list currently includes:
Static Check Before Call

Ability to check operation arguments and types before actually doing the server side call.
Working with Document Object

Ability to work fluently with Document objects:

- fetch a document and retrieve it as a Document object,
- update the Document object,
- send it back to the server.

Background Upload/Download Pool

Manage HTTP threads pool for upload/download in background.
Caching

Support client side caching to provide minimal offline capability.
Advanced Exception Management

Retrieve advanced information about potential server side exception.
Advanced Authentication

Support integrated authentication (via portal_sso).
Layout Management

Support for fetching layouts definition associated to documents.

Compatibility Matrix

			Java Client	JavaScript Client	Python Client	PHP Client	C# Client	Android Client	Dart Client
Compatibility									
	Base								
		Create and read docs	testCRUDSuite	CRUD					

		Pagination	testPaginationSuite	Pagination					
	Blobs								
		Direct Blob upload	testBlobSuite	BlobUpload					
		Blob upload via batch manager		BlobUploadBatch					
		Blob update vua batch manager		BlobUpdateBatch					

	Marshaling								
		Complex properties	testComplexPropSuite						
		Custom marshaling							
		Business Objects	testBOSuite						
Convenience									
		Static checks							
		Working with documents							
		Background upload/download							
		Caching							
		Advanced Exception management							
		Advanced Authentication							
		Layout Management							

Running the Tests

Client Side Code

The client side code depends on the following underlying technologies:

- JUnit tests for Java client,
- NUnit tests for .Net client,
- HTML/JS for JavaScript client,
- Python script for Python client,
- JUnit for both Android clients (the nuxeo-android-connector and the deprecated nuxeo-automation-thin-client),
- ? for PHP client,
- ? for Dart client.

Server Side Code

Compatibility tests have to be run against the lastest version of Nuxeo CAP.
For now, no specific additional module is required.

Keep in mind that the tests do create document on the tested Nuxeo instance, don't run it against a production server otherwise you will end up with garbage in your repository since tests don't enforce cleanup.

iOS Client

The iOS client is still under development, you can find the source code on GitHub.

Goal is to integrate correctly with REST tools of the Nuxeo Platform, and provide a document lists synchronization service, so as to be able to access content offline.

Android Client

You can browse the Android Connector section for more information.

Using cURL

In this example we are using the UNIX `curl` command line tool to demonstrate how to invoke remote operations.

In a Windows environment, Nuxeo recommends to use cygwin shell to avoid all formatting request issue using default shell like Powershell

1. Here is how to create a new File document type on the Nuxeo demo instance, right under the default domain (you can copy, paste and test):

```
curl -X POST -H "Content-Type: application/json" -u Administrator:Administrator
-d '{ "entity-type": "document", "name":"newDoc", "type": "File","properties": {
"dc:title": "Specifications", "dc:description": "Created via a so cool and simple
REST API", "common:icon": "/icons/file.gif", "common:icon-expanded": null,
"common:size": null}}' http://demo.nuxeo.com/nuxeo/api/v1/path/default-domain
```

2. You can get the new resource doing a standard GET (actually the JSON object was already returned in previous response):

```
curl -X GET -u Administrator:Administrator
http://demo.nuxeo.com/nuxeo/api/v1/path/default-domain/newDoc
```

3. Now, "lock" this document we have just created by calling an Automation operation from command API on the document resource.

```
curl -X POST -H "Content-Type: application/json+nxrequest"   -u
Administrator:Administrator -d '{"params":{}}'
http://demo.nuxeo.com/nuxeo/api/v1/path/default-domain/newDoc/@op/Document.Lock
```

   Pay attention to the Content-Type that is specific when using the @op adapter.

   You can check the result of your request on the web app (http://demo.nuxeo.com/nuxeo/nxpath/default@view_domains, credentials: Administrator/Administrator).
4. You can also directly call an automation operation or chain, from the "Command endpoint". Here we return all the worskpaces of the demo.nuxeo.com instance:

```
curl -H 'Content-Type:application/json+nxrequest' -X POST -d
'{"params":{"query":"SELECT * FROM Document WHERE
ecm:primaryType=\"Workspace\""},"context":{}}'   -u Administrator:Administrator
http://demo.nuxeo.com/nuxeo/api/v1/automation/Document.Query
```

Other Repository APIs

This section is about the SOAP bridge, CMIS and WebDAV APIs, as well as URLs to use when downloading a binary content.

Nuxeo has generalized the REST approach regarding remote accesses for sake of easiness (from a developer perspective) and wide adoption in Content Management / web applications field. See the complete documentation of this REST API.

We also provide a toolkit and guidelines for wrapping services of the platform SOAP style. Note you'll find only a limited set of SOAP web-services out-of-the box (beside the CMIS implementation), since we didn't make it our main design pattern for accessing the repository remotely.

Finally, the Nuxeo Platform also implements standard CMIS and WebDAV interfaces. Those latter APIs are more strictly "content" oriented, mainly providing CRUD operations on documents.

In this section:

Downloading Files

This page provide the logic for building the correct URL for downloading a file stored on a Nuxeo Document, provided we know the document ID.

You can use the `nxbigfile` pattern that is executed by a standalone servlet:

- `http://127.0.0.1:8080/nuxeo/nxbigfile/default/776c8640-7f19-4cf3-b4ff-546ea1d3d496/file:content/rm.pdf`
- `http://127.0.0.1:8080/nuxeo/nxbigfile/default/776c8640-7f19-4cf3-b4ff-546ea1d3d496/blobholder:0/rm.pdf`
- `http://127.0.0.1:8080/nuxeo/nxbigfile/default/776c8640-7f19-4cf3-b4ff-546ea1d3d496/files:files/0/file/SC-DM-DAM.png`
- `http://127.0.0.1:8080/nuxeo/nxbigfile/default/776c8640-7f19-4cf3-b4ff-546ea1d3d496/blobholder:1/SC-DM-DAM.png`

Within JSF Context Only

The default URL pattern for downloading files from within the JSF environment is:

- `http://{server}:{port}/nuxeo/nxfile/{repository}/{uuid}/blobholder:{blobIndex}/{fileName}`
- `http://{server}:{port}/nuxeo/nxfile/{repository}/{uuid}/{propertyXPath}/{fileName}`

Where :

- `repository` is the identifier of the target repository,
- `uuid` is the uuid of the target document,
- `blobIndex` is the index of the Blob inside the `BlobHolder` adapter corresponding to the target Document Type (starting at 0),
- `propertyXPath` is the xPath of the target Blob property inside the target document,
- `fileName` is the name of the file as it should be downloaded (this information is actually not used to do the resolution).

Here are some examples :

- `http://127.0.0.1:8080/nuxeo/nxfile/default/776c8640-7f19-4cf3-b4ff-546ea1d3d496/file:content/rm.pdf`

- `http://127.0.0.1:8080/nuxeo/nxfile/default/776c8640-7f19-4cf3-b4ff-546ea1d3d496/blobholder:0/rm.pdf`
- `http://127.0.0.1:8080/nuxeo/nxfile/default/776c8640-7f19-4cf3-b4ff-546ea1d3d496/files:files/0/file/SC-DM-DAM.png`
- `http://127.0.0.1:8080/nuxeo/nxfile/default/776c8640-7f19-4cf3-b4ff-546ea1d3d496/blobholder:1/SC-DM-DAM.png`

For Picture document type, a similar system is available to be able to get the attachments depending on the view name:

- `http://{server}:{port}/nuxeo/nxpicsfile/{repository}/{uuid}/{viewName}:content/{fileName}`

where, by default, `viewName` can be Original, OriginalJpeg, Medium, Thumbnail.

Other Documentation About URLs

- Default URL Patterns
- URLs for Files

SOAP Bridge

JAX-WS Bindings

The Nuxeo Platform includes a JAX-WS compliant implementation to expose SOAP based web services. Sun JAX-WS (Metro) is used.

As explained earlier in these pages, JAX-WS is not the preferred way to expose web services on top of Nuxeo Platform. This is the reason why the default distributions don't expose a lot of SOAP based web services. By default, the exposed SOAP web services include:

- A read only access to the Document Repository and the User Manager (NuxeoRemoting),
- A read only access to the Audit trail (NuxeoAudit),
- The CMIS bindings.

In this section

- JAX-WS Bindings

If you want to access the SOAP web service:

- To list all the deployed endpoints:
  - `http://server:port/nuxeo/webservices/nuxeoremoting` to list all the deployed endpoints (when using SUN Metro),
  - `http://server:port/jbossws/services` (when using JbossWS),
- To access NuxeoRemoting WSDL: `http://server:port/nuxeo/webservices/nuxeoremoting?wsdl`,
- To access NuxeoAudit WSDL: `http://server:port/nuxeo/webservices/nuxeoaudit?wsdl`.

The point of SOAP web services is not to have a 1 to 1 mapping with the Java services interfaces. The goal is to provide a "coarse grained" high level API. So it's easy to build new SOAP based web services on top of Nuxeo:

- The JAX-WS infrastructure is already there,
- The authentication system is already in place,
- We provide the base class to manage Repository and security (AbstractNuxeoWebService),
- We already provide JAXB objects for Documents, Security descriptors, properties ...

You can find a step-by-step example on the page Building a SOAP-Based WebService in the Nuxeo Platform.

You might also want to build a **web service client** in Nuxeo to be able to query a remote web service. A simple example is available on the page Building a SOAP-Based WebService Client in Nuxeo.

Building a SOAP-Based WebService Client in Nuxeo

To start with we assume that we have access to a remote WebService, from which we can easily download the WSDL file. For example, let's take a simple WebService that provides weather forecast: http://www.restfulwebservices.net/wcf/WeatherForecastService.svc?wsdl.

The code samples used for this example can be found here.

On this page

Generating the WebService Client Java Source Files with Maven

The `wsimport` goal from the Maven plugin `jaxws-maven-plugin` allows you to generate the WebService client Java source files given a WSDL file.
The WSDL file can either be accessed remotely using an URL or locally using a directory.
Obviously it is better to point at a local (previously downloaded) WSDL file to ensure reproducibility of the build.

For instance, copy http://www.restfulwebservices.net/wcf/WeatherForecastService.svc?wsdl to `src/main/resources/wsdls` in a Nuxeo project and use the following code in your `pom.xml`:

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>jaxws-maven-plugin</artifactId>
  <executions>
    <execution>
      <goals>
        <goal>wsimport</goal>
      </goals>
      <configuration>
        <wsdlDirectory>src/main/resources/wsdls</wsdlDirectory>

<sourceDestDir>${project.build.directory}/generated-sources/wsimport</sourceDestDir>
      </configuration>
    </execution>
  </executions>
</plugin>
```

When you'll build the project, all WSDL files in `src/main/resources/wsdls` will be parsed and the associated Java source files generated in `${project.build.directory}/generated-sources/wsimport`.

The `jaxws:wsimport` goal is automatically executed within the life cycle phase `generate-sources`, ie. before the `compile` phase. This way, running `mvn clean install` first generates the source files and then compiles them.

Be careful with XSD schema imports!
Often a WSDL file needs to import one or more XSD schemas, using the `<xsd:import>` directive.

Let's take this example:
`<xsd:import` **`schemaLocation="http://www.restfulwebservices.net/wcf/WeatherForecastService.svc?xsd=xsd0"`** `namespace="http://www.restfulwebservices.net/ServiceContracts/2008/01"/>`

If your WSDL file is accessed locally, you also have to download all the XSD schemas referenced by the WSDL and use a relative path for the `schemaLocation` attribute of the `<xsd:import>` directives prior to building your project.

Assuming that the XSD schema `WeatherForecastService.xsd` has been downloaded in `src/main/resources/wsdls/xsdschemas`, this is what you get:
`<xsd:import` **`schemaLocation="xsdschemas/WeatherForecastService.xsd"`** `namespace="http://www.restfulwebservices.net/ServiceContracts/2008/01"/>`.

Documentation about `wsimport` is available here: http://jax-ws-commons.java.net/jaxws-maven-plugin/wsimport-mojo.html.

Documentation about using JAX-WS with Maven is available here: http://blogs.oracle.com/enterprisetechtips/entry/using_jax_ws_with_maven.

Getting an Instance of the WebService Client

Once the Java source files have been generated and compiled, you can use them as a third-party library:

```
WeatherForecastService wfs = new WeatherForecastService(
     new URL("http://www.restfulwebservices.net/wcf/WeatherForecastService.svc?wsdl"),
     new QName("http://www.restfulwebservices.net/ServiceContracts/2008/01",
"WeatherForecastService"));
IWeatherForecastService iwfs = wfs.getBasicHttpBindingIWeatherForecastService();
```

The (slightly) tricky part here is to find the actual **WebService client** in the generated classes and the method it provides to get an instance of the **WebService client interface**.

- The actual **WebService client** is the class with the following annotation: @WebServiceClient(name = "WeatherForecastService",...
  In our example: `WeatherForecastService`
  - The URL is the one of the WSDL file.
  - To build the `QName`, we use the `targetNamespace` and `name` attributes of the `<wsdl:definitions>` element in the WSDL file.
    In our example: `<wsdl:definitions name="WeatherForecastService" targetNamespace="http://www.restfulwebservices.net/ServiceContracts/2008/01"....`
- The **WebService client interface** is the class with the following annotation: @WebService(name = "IWeatherForecastService",...
  In our example: `IWeatherForecastService`
- The `WeatherForecastService` class offers the method `getBasicHttpBindingIWeatherForecastService()` that returns an object of type `IWeatherForecastService`.

Calling a WebService Method

Finally, you can call any method available in the WebService client interface, for instance `IWeatherForecastService#getCitiesByCountry(String country)` which returns the cities of a given country.

```
ArrayOfstring cities = iwfs.getCitiesByCountry("FRANCE");
String message = "Cities of country FRANCE: " + cities.getString();
```

You can learn here how to build a server-side SOAP based WebService in Nuxeo.

Advanced Client Configuration Using a Configuration XML File

If you are looking how to create some advanced client configuration, since Apache CXF, you'll need some dependencies. The configuration behavior is the same as the server part, and described in the advanced configuration section of the Building a SOAP-Based WebService in the Nuxeo Platform.

Building a SOAP-Based WebService in the Nuxeo Platform

Let's take the example of a simple Nuxeo WebService that exposes a method to create a document in the default repository. The user credentials are passed to the method to log in and open a core session.

The code samples used for this example can be found on GitHub.

On this page

WebService Interface

First you have to define the interface of your WebService, answering the question: "Which methods do I want to expose?"
If we take a look at `NuxeoSampleWS`, a standard signature defines the `createDocument` method.

```
DocumentDescriptor createDocument(String username, String password,
            String parentPath, String type, String name) throws LoginException,
            ClientException;
```

The different parameters are:

- `username`: used to log in and open a core session
- `password`: used to log in and open a core session
- `parentPath`: document parent path
- `type`: document type
- `name`: doc name

The return type `DocumentDescriptor` is a simple POJO containing minimal information about a document.

Webservice Implementation

You just need a few annotations from the `javax.jws` package to implement your SOAP based WebService.

@WebService

Allows you to define the actual WebService. You need to provide the following parameters:

- `name`: the name of the WebService, used as the name of the `wsdl:portType` when mapped to WSDL 1.1.
- `serviceName`: the service name of the WebService, used as the name of the `wsdl:service` when mapped to WSDL 1.1.

@SOAPBinding

Specifies the mapping of the WebService onto the SOAP message protocol. You can provide the following parameters:

- `style`: defines the encoding style for messages send to and from the WebService. Keep the default value `Style.DOCUMENT`.
- `use`: defines the formatting style for messages sent to and from the WebService. Keep the default value `Use.LITERAL`.

@WebMethod

Exposes a method as a WebService operation. Such a method must be public.

@WebParam

Maps a method parameter to a WebService operation parameter. You need to provide the `name` of the parameter.

Take a look at `NuxeoSampleWSImpl` which uses all these annotations:

```
@WebService(name = "NuxeoSampleWebServiceInterface", serviceName =
"NuxeoSampleWebService")
@SOAPBinding(style = Style.DOCUMENT)
public class NuxeoSampleWSImpl implements NuxeoSampleWS {

    /** The serialVersionUID. */
    private static final long serialVersionUID = 7220394261331723630L;

    /**
     * {@inheritDoc}
     */
    @WebMethod
    public DocumentDescriptor createDocument(@WebParam(name = "username")
    String username, @WebParam(name = "password")
    String password, @WebParam(name = "parentPath")
    String parentPath, @WebParam(name = "name")
    String name, @WebParam(name = "type")
    String type) throws LoginException, ClientException {
        ...
    }
    ...
}
```

WebService End Point and URL Mapping

Finally you have to define your WebService as an end point mapped to an URL of the Nuxeo server so it can be accessed in HTTP.
This is done by adding a contribution to the `endpoint` extension point of the `org.nuxeo.ecm.platform.ws.WSEndpointManager` component in your Nuxeo project.

```
<extension target="org.nuxeo.ecm.platform.ws.WSEndpointManager" point="endpoint">
  <endpoint name="nuxeosample"
    implementor="org.nuxeo.ecm.samples.ws.server.NuxeoSampleWSImpl"
    address="/nuxeosample" />
</extension>
```

Once your plugin is deployed in the Nuxeo server, the Nuxeo sample WebService WSDL should be available at http://server:port/nuxeo/webservices/nuxeosample?wsdl.

Advanced Configuration Requiring a Configuration XML File

Since we moved to Apache CXF as a WebService provider, we encapsulate the service definition in our contribution model to prevent CXF Spring dependency. Sometimes, you'll need a specific configuration that will require a configuration XML file, as described in their documentation.

Here are the steps to do that in a Nuxeo way:

1. Add the following dependencies `spring-core`, `spring-beans`, `spring-context`, `spring-aop`, `spring-expression` and `spring-asm`. Ideally, using a Nuxeo Package.
2. Deploy your `cxf.xml` file in `nuxeo.war/WEB-INF/classes`, using the `deployment-fragment.xml`.
   And the file will be taken into account.

About Code Factorization

If you take a closer look at the `NuxeoSampleWSImpl#createDocument` method, you can see that most of the code is not "business"-related and could be factorized.
Indeed, each time one makes a remote call to a Nuxeo WebService method, the following pattern must be applied:

- log in using the provided credentials,
- start a transaction,
- open a core session,
- do the job,

- manage transaction rollback if an exception occurs when doing the job,
- close the core session,
- commit or rollback the transaction,
- log out.

Which can be implemented as so:

```
// Login
LoginContext loginContext = Framework.login(username, password);

// Start transaction
TransactionHelper.startTransaction();

CoreSession session = null;
try {
    // Open a core session
    session = openSession();

    // Do the job: create a doc with the given params
    DocumentModel doc = session.createDocumentModel(parentPath, name,
        type);
    doc = session.createDocument(doc);
    session.save();

    return new DocumentDescriptor(doc);

} catch (ClientException ce) {
    // Set transaction for rollback if an exception occurs
    TransactionHelper.setTransactionRollbackOnly();
    throw ce;
} finally {
    // Close the core session
    if (session != null) {
        closeSession(session);
    }
    // Commit or rollback transaction
    TransactionHelper.commitOrRollbackTransaction();
    // Logout
    loginContext.logout();
}
```

A way to solve this issue could be to define an abstract class holding this pattern in a generic method, which would call an abstract method responsible for the "business" part of the code, in the same way as for `UnrestrictedSessionRunner`.

You can learn how to build a client-side SOAP based WebService in the Nuxeo Platform on the page Building a SOAP-Based WebService Client in Nuxeo.

Trust Store and Key Store Configuration

Introduction

When you make Nuxeo discuss with other servers through different APIs, you need to add the authentication certificate and your trust store because:

- Establishing connection requires to expose the certificate to the remote server,
- the remote server exposes a self-signed certificate or a certificate signed by a certification authority not known by the standard Key Store.

When your Nuxeo server establishes a remote connection, the remote server exposes a certificate that is his ID card on the network so the Nuxeo server is assured to communicate with a trusted server. Instead passing through detector to trust it, this certificate has been signed by an authority of certification. The trust store contains all certificates of the authorities that will be trusted by the JVM, especially for SSL connections and more particularly HTTPS connections.

The Key Store will contains all the keys needed by the JVM to be authenticated to a remote server.

There are 2 ways to configure these:

- setting the Trust Store and the Key Store statically
- setting it dynamically

In this section

If you set a custom trust store with your authorities exclusively, **Marketplace, Studio and hot fix distribution integration will not work anymore** since these servers expose certificates available in the default trust store. So I suggest that you add your certificates to the default one.

Static Trust Store and Key Store

To set the trust store and key store statically, you just have to add the following parameter into the environment variable:

What for	Parameter name	Comment
Trust Store Path	javax.net.ssl.trustStore	
Trust Store Type	javax.net.ssl.keyStoreType	JKS for instance
Key Store Path	javax.net.ssl.keyStore	
Key Store Password	javax.net.ssl.keyStorePassword	

So if you want to set them at start time, you can add the following parameter either:

- into your JAVA_OPTS:

$NUXEO_HOME/bin/nuxeo.conf

```
-Djavax.net.ssl.trustStore=/the/path/to/your/trust/store.jks
-Djavax.net.ssl.keyStoreType=jks ... etc ...
```

- or into your Java code:

MyClass.java

```
        System.setProperty(
                "javax.net.ssl.trustStore",
                "/the/path/to/your/trust/store.jks");
        System.setProperty(
                "javax.net.ssl.keyStore",
                "/the/path/to/your/key/store.jks");
        System.setProperty("javax.net.ssl.keyStorePassword", "myPassword");
        ...etc...
```

Dynamic Trust Store

...TODO...

Adding your Certificates into the default Trust Store

You will find the default trust store delivered with your JVM in:

```
$JAVA_HOME/lib/security/cacerts
```

For instance in Mac OS, it is in:

```
/System/Library/Frameworks/JavaVM.framework/Home/lib/security/cacerts
```

By default the password for this Trust Store is "changeit".

So to add your certificates to the default trust store:

1. Copy the default trust store.
2. Launch the following command line to add your certificate to the default trust store copy:

```
keytool -import -file /path/to/your/certificate.pem -alias
NameYouWantToGiveOfYourCertificate -keystore
/path/to/the/copy/of/the/default/truststore.jks -storepass changeit
```

3. Set the trust store copy as your either statically or dynamically.
4. Restart your Nuxeo instance.

Troubles

If your Nuxeo instance cannot access to Connect anymore, the Marketplace and Hot Fixes are no longer automatically available (through the Update Center for instance), this can mean that the trust store does not contain the certificates from the authority that signed Nuxeo Servers certificates.

If you have the following error in your logs during the connection establishment:

```
sun.security.validator.ValidatorException: PKIX path building failed:
sun.security.provider.certpath.SunCertPathBuilderException: unable to find valid
certification path to requested target
```

It means that the remote certificate is not trusted.

The following messages mean there is no trust store or key store set for your JVM:

```
java.lang.RuntimeException: Unexpected error:
java.security.InvalidAlgorithmParameterException: the trustAnchors parameter must be
non-empty
```

or

```
java.security.NoSuchAlgorithmException: Error constructing implementation (algorithm:
Default, provider: SunJSSE, class: com.sun.net.ssl.internal.ssl.DefaultSSLContextImpl)
```

This means you must have broken at least the default configuration.

If you have one of the following error, the remote server has been trusted but it asks for authentication and there is no key for that:

```
Received fatal alert: handshake_failure
```

or

```
Remote host closed connection during handshake
```

The following error can mean that the set key store is not available:

```
java.security.NoSuchAlgorithmException: Error constructing implementation (algorithm:
Default, provider: SunJSSE, class: com.sun.net.ssl.internal.ssl.DefaultSSLContextImpl)
```

CMIS for Nuxeo

CMIS is the OASIS specification for content management interoperability. It allows client and servers to talk together in HTTP (SOAP or REST/AtomPub) using a unified domain model. The latest published version is CMIS 1.1.

Status

Nuxeo supports CMIS through the following modules:

- The Apache Chemistry OpenCMIS library, maintained by Nuxeo and others, which is a general-purpose Java library allowing developers to easily write CMIS clients and servers,
- Specific Nuxeo OpenCMIS connector bundles, allowing Nuxeo to be used as a CMIS server with the help of OpenCMIS.

Versions

The Nuxeo OpenCMIS connector supports CMIS 1.1 starting with Nuxeo Platform 5.8.

Online Demo

A demo server has been set up for you to try Nuxeo with CMIS. You can try it here: http://cmis.demo.nuxeo.org/ (login: Administrator / password: Administrator).

The AtomPub service document is here: http://cmis.demo.nuxeo.org/nuxeo/atom/cmis (same credentials).
The SOAP WSDL for the repository service is here: http://cmis.demo.nuxeo.org/nuxeo/webservices/cmis/RepositoryService?wsdl

Downloads

The latest LTS and Fast Track releases include the CMIS connector by default. You can get them from here: http://nuxeo.com/downloads

Usage

Make sure that the Nuxeo server is started: check that there are no ERRORs in the startup logs and that you can normally connect to your server using a browser, at `http://localhost:8080/nuxeo` .

In this section

AtomPub

You can use a CMIS 1.1 AtomPub client and point it at `http://localhost:8080/nuxeo/atom/cmis` .

If you want to check the AtomPub XML returned using the command line, this can be done using `curl` or `wget`:

```
curl -u Administrator:Administrator http://localhost:8080/nuxeo/atom/cmis
```

To do a query you can do:

```
curl -u Administrator:Administrator
"http://localhost:8080/nuxeo/atom/cmis/default/query?q=SELECT+cmis:objectId,+dc:titl
e+FROM+cmis:folder+WHERE+dc:title+=+'Workspaces'&searchAllVersions=true"
```

You should probably pipe this through `tidy` if you want a readable output:

```
... | tidy -q -xml -indent -wrap 999
```


Since Nuxeo 5.5 the `searchAllVersions=true` part is mandatory if you want something equivalent to what you see in Nuxeo (which often contains mostly private working copies).

SOAP

The following SOAP endpoints are available:

- http://localhost:8080/nuxeo/webservices/cmis/RepositoryService
- http://localhost:8080/nuxeo/webservices/cmis/DiscoveryService
- http://localhost:8080/nuxeo/webservices/cmis/ObjectService
- http://localhost:8080/nuxeo/webservices/cmis/NavigationService

- http://localhost:8080/nuxeo/webservices/cmis/VersioningService
- http://localhost:8080/nuxeo/webservices/cmis/RelationshipService
- http://localhost:8080/nuxeo/webservices/cmis/MultiFilingService
- http://localhost:8080/nuxeo/webservices/cmis/ACLService
- http://localhost:8080/nuxeo/webservices/cmis/PolicyService

Note that most SOAP CMIS clients are configured by using just the first URL (the one about RepositoryService), the others are derived from it by changing the suffix.

Authentication is done using Web Services Security (WSS) UsernameToken.

Here is a working example of a SOAP message to the DiscoveryService:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ns="http://docs.oasis-open.org/ns/cmis/messaging/200908/">
  <soapenv:Header>
    <Security
xmlns="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0
.xsd">
      <UsernameToken>
        <Username>Administrator</Username>
        <Password>Administrator</Password>
      </UsernameToken>
    </Security>
  </soapenv:Header>
  <soapenv:Body>
    <ns:query>
      <ns:repositoryId>default</ns:repositoryId>
      <ns:statement>SELECT cmis:objectid, dc:title FROM cmis:document WHERE dc:title
= 'Workspaces'</ns:statement>
      <ns:maxItems>20</ns:maxItems>
      <ns:skipCount>0</ns:skipCount>
    </ns:query>
  </soapenv:Body>
</soapenv:Envelope>
```

CMIS Clients

Several free clients for CMIS 1.1 are available.

The best one is the CMIS Workbench, part of OpenCMIS.

And of course you can use the Chemistry libraries to produce your own client (Java, Python, PHP, .NET). Documentation and sample for using OpenCMIS libraries can be found on the OpenCMIS developer wiki with also example code and how-tos.

From Java Code Within a Nuxeo Component

To create, delete or modify documents, folders and relations just use the regular `CoreSession` API of Nuxeo. To perform CMISQL queries (for instance to be able to perform `JOIN` that are not supported by the default `NXQL` query language, have a look at the following entry in the Knowledge Base: Using CMISQL from Java.

Documentation

You can browse the CMIS 1.1 HTML version or download CMIS 1.1 (PDF) (1.3 MB).

Capabilities

The Nuxeo OpenCMIS connector implements the following capabilities from the specification:

Navigation Capabilities		
	Get descendants supported	Yes
	Get folder tree supported	Yes

	Order By supported	Custom
Object Capabilities		
	Content stream updates	PWC only
	Changes	Object IDs only
	Renditions	Read
Filing Capabilities		
	Multifiling supported	No
	Unfiling supported	No
	Version-specific filing supported	No
Versioning Capabilities		
	PWC updatable	Yes
	PWC searchable	Yes
	All versions searchable	Yes
Query Capabilities		
	Query	Both combined
	Joins	Inner and outer
Type Capabilities		
	Create property types	No
	New type settable attributes	None
ACL Capabilities		
	ACLs	None

Model Mapping

The following describes how Nuxeo documents are mapped to CMIS objects and vice versa.

- Only Nuxeo documents including the "dublincore" schema are visible in CMIS.
- Complex properties are not visible in CMIS, as this notion does not exist in CMIS.
- Proxy documents are not visible in CMIS.
- Secondary content streams are not visible as renditions.
- Documents in the Nuxeo trash (those whose `nuxeo:lifecycleState` is `deleted`) are not visible in CMIS, unless an explicit query using the `nuxeo:lifecycleState` property is done.

This mapping may change to be more comprehensive in future Nuxeo versions.

Nuxeo-Specific System Properties

In addition to the system properties defined in the CMIS specification under the **`cmis:`** prefix, the Nuxeo Platform adds a couple of additional properties under the **`nuxeo:`** prefix:

Since Nuxeo 5.4.2

- **`nuxeo:isVersion`** : to distinguish between archived (read-only revision) and live documents (that can be edited);
- **`nuxeo:lifecycleState`** : to access the life cycle state of a document: by default only document in non `deleted` state will be returned in CMISQL queries unless and explicit **`nuxeo:lifecycleState`** predicate is specified in the `WHERE` clause of the query;
- **`nuxeo:secondaryObjectTypeIds`** : makes it possible to access the facets of a document. Those facet can be static (as defined in the type definitions) or dynamic (each document instance can have declared facets);
- **`nuxeo:contentStreamDigest`** : the low level, MD5 or SHA1 digest of blobs stored in the repository. The algorithm used to compute the digest is dependent on the configuration of the **`BinaryManager`** component of the Nuxeo repository.

`nuxeo:isVersion` , **`nuxeo:lifecycleState`** and **`nuxeo:secondaryObjectTypeIds`** are properties that can be queried upon: they can be used in the `WHERE` clause of a CMISQL query. This is not yet the case for **`nuxeo:contentStreamDigest`** that can only be read in query results or by introspecting the properties of the `ObjectData` description of a document.

Since Nuxeo 5.5

- **`nuxeo:isCheckedIn`** : for live documents, distinguishes between the checked-in and checked-out state.
- **`nuxeo:parentId`** : like `cmis:parentId` but also available on Document objects (which is possible because Nuxeo does not have direct multi-filing).

Since Nuxeo 5.6

- **`nuxeo:pathSegment`** : the last path segment of the document (`ecm:name` in NXQL).

Source Code

The Nuxeo OpenCMIS connector source code is available on GitHub: https://github.com/nuxeo/nuxeo-chemistry.

The Apache Chemistry OpenCMIS source code is available on Apache's Subversion server: https://svn.apache.org/repos/asf/chemistry/opencmis/trunk.

Additional Resources

- CMIS: Overview of a Rapidly Evolving ECM Standard, presentation on SlideShare
- CMIS and Apache Chemistry (ApacheCon 2010), presentation on SlideShare
- Nuxeo World Session: CMIS - What's Next?, presentation on SlideShare

WebDAV

Nuxeo supports the WebDAV (Web-based Distributed Authoring and Versioning) protocol and thus enables you to create and edit Office documents stored in Nuxeo directly from the Windows or Mac OS X desktop, without having to go through your Nuxeo application.

The documentation about installation and usage of WebDAV can be found in the Nuxeo Platform User Guide.

Adding a new WebDAV Client

The plugin comes with a default configuration which supports only a few clients among Windows 7's one, litmus, davfs, cadaver. If your usual client is not listed, you can override this configuration by adding a new file `webdav-authentication-config.xml` under `$NUXEO/nxserver/config/` and update the list associated to the header.

Below is an example where BitKinex is added:

```
<?xml version="1.0"?>
<component name="org.nuxeo.ecm.platform.wi.auth.config.custom">

  <require>org.nuxeo.ecm.platform.wi.auth.config</require>

  <extension
target="org.nuxeo.ecm.platform.ui.web.auth.service.PluggableAuthenticationService"
point="specificChains">

    <specificAuthenticationChain name="WebDAV">
      <headers>
        <header
name="User-Agent">(Microsoft-WebDAV-MiniRedir|DavClnt|litmus|gvfs|davfs|WebDAV|cadav
er|BitKinex).*</header>
      </headers>

      <replacementChain>
        <plugin>DIGEST_AUTH</plugin>
        <plugin>WEBDAV_BASIC_AUTH</plugin>
      </replacementChain>
    </specificAuthenticationChain>
  </extension>

</component>
```

Related Documentation

- Working with WebDAV and WSS

OpenSocial, OAuth and the Nuxeo Platform

About Opensocial

OpenSocial is a community standard managed by the OpenSocial foundation. Several major companies (Google, IBM, SAP, Atlassian ...) are involved in the OpenSocial standard.
You can consult the full OpenSocial specification here, but at a glance OpenSocial scope includes:

JavaScript and HTML gadgets

OpenSocial gadgets are small HTML/JavaScript applications that consume data using REST WebServices.

The idea is similar to the JSR 168 portlet with some differences:

- Rendering is mainly done on the client side (Browser side),
- OpenSocial gadgets are independent of the server side technology (Java, PHP, Python, .Net ...),
- Deploying a new gadget is simply referencing a URL that contains the XML gadget spec.

Gadgets are run in a container that is responsible for providing the gadget with the rendering environment and needed JavaScript APIs.

REST Web Services

OpenSocial defines a set of REST web services that must be implemented by compliant container.

The main services are:

- App Data service: Service to store user specific data for each gadget / app;
- Groups service: Manages relationship between users. (Each user has the ability to manage his own groups: friends, coworkers, ...);
- Activity service: Tracks user activity;
- Person service: Standard access to the users of the system;
- Message service: Simple messaging system.

These web services provide the "social part" of the OpenSocial technology.

Managing Authentication and Security of Data Accesses

In OpenSocial there are usually three actors:

- The user: The human looking at the gadget inside his browser;
- The container: The "server" providing the container page where the gadgets are rendered;
- The service provider: The server providing the REST service accessed by the gadget.

Additionally, the AppId (i.e. the gadget) may be taken into account.

Also, OpenSocial differentiate two types of user identify:

- The viewer: The user in front of the browser;
- The owner: The user owning the page that contains the gadget.

In this section

If you take the example of a gadget that displays your emails from Gmail inside Nuxeo:

- You are the user (OpenSocial viewer and OpenSocial owner),
- Nuxeo is the container,
- Gmail is the service provider,
- the Gmail gadget is the AppId.

Because gadgets are HTML/JavaScript, the gadget can not call Gmail directly (Cross site scripting issue), so the container will proxy all calls from to gadget to Gmail.

So, the container (here Nuxeo) will manage REST call proxying:

- Handling caching if needed,
- Handling security.

The security part is not trivial because there are three constraints in OpenSocial context:

- You need to delegate authentication to a server (i.e. let Nuxeo connect to Gmail on behalf of me);
- Your Gmail and Nuxeo account may be completely unrelated (not the same login for example);
- You may not want to let Nuxeo access all your Gmail data, but only a subset.

In order to manage theses requirements OpenSocial relies on OAuth. The Nuxeo Platform server can be used both as an OAuth Service consumer or as an OAuth Service provider.

You can find more details about OAuth and its integration in Nuxeo later in this page.

OpenSocial in the Nuxeo Platform Use Cases

Dashboard and Portal Use Case

The most direct use cases is to use OpenSocial gadgets to address dashboard / portal use case. This is currently the default usage of OpenSocial within the Nuxeo Platform.

For that, we have created some simple and extensible gadgets:

- My recent documents,
- My Workspaces,
- My Tasks,
- ...

The user can then use Nuxeo's dashboard to have a quick overview of his activity inside the Nuxeo Platform.

Using OpenSocial gadgets to do so is interesting for the following reasons.

Letting Users Control Their Dashboards

Depending on the security policy, the user is able to customize his dashboard:

- adding / removing gadgets,
- changing gadgets configurations.

Leveraging an Open Standard

Gadgets can come from several different services providers:

- My Tasks on Nuxeo,
- My Task on JIRA,
- My Google calendar events.

More and more application are supporting OpenSocial:

- Google Apps (also provides an OpenSocial container in Gmail and iGoogle),
- Atlassian Jira and Confluence (that are also OpenSocial container),
- SAP,
- SalesForce.com,
- Social Oriented services like LinkedIn, MySpace, Hi5, Orkut, etc.
- ...

So you can add external gadgets to Nuxeo's dashboard, but you can also use an external OpenSocial dashboard provided by an other application and use Nuxeo's gadgets from inside this container.

Managing User Identity

Because OpenSocial relies on OAuth you can have a narrow control of what service may be used and what scope of data you accept to share between the applications.

Simple Deployment

Deploying an OpenSocial gadget in a container is as simple as giving a new URL.

Easy Contextual Information Integration

OpenSocial gadgets can also be used to provide a simple and light integration solution between two applications.

For example, you can:

- Display some informations comming from SAP next to the Invoice Document in Nuxeo,
- Diplay links to related Nuxeo documents from within a Confluence wiki page,
- Display specification documents (stored in Nuxeo) from withing the JIRA Issue.

An more generally, if you have enterprise wide REST web service (like Contacts managements, Calendar management ...) you can expose them via OpenSocial gadgets so that theses services are accessible to users of all OpenSocial aware applications.

Social Network Management

Because OpenSocial standardizes a set of social oriented services, you can easily leverage the data from all OpenSocial aware applications. For example you can have an aggregated view on the activity stream.

In the context of Document Management, Collaboration and KM, social APIs really make sense to:

- manage communities,
- manage activity,
- manage user skils,
- ...

OAuth in the Nuxeo Platform

The Nuxeo Platform provides full support for OAuth.

Nuxeo as a Service Provider

You may want to use Nuxeo as an OAuth Service provider:

- If you have an OpenSocial gadget that uses a REST service from Nuxeo,
- If you have a external application that uses a REST service from Nuxeo.

Unless the service consumer is a gadget directly hosted inside Nuxeo, you will need to do some configuration at Nuxeo level to define how you want the external consumer to have access to Nuxeo.

Nuxeo as a Service Consumer

If you need to access an external REST service from within Nuxeo, you may want to use OAuth to manage the authentication.

If you use this external service from within an OpenSocial gadget, you will need to use OAuth.

Unless the service you want to consume is hosted by your Nuxeo instance, you will need to do some configuration do define how Nuxeo must access this external service. For more details about OAuth support in Nuxeo, please see the OAuth dedicated section.

Nuxeo Automation REST Services

OpenSocial gadgets typically use REST services to fetch the data from the service provider. When Nuxeo is the service provider, the recommended target API is Nuxeo Automation. Nuxeo Automation presents the following advantages:

- It provides a solution to easily have custom REST APIs without having to write code (using Nuxeo Studio),
- It is a common access point for all REST services (all operations and chains can be called the same way),
- It has a native support for JSON marshaling (which means easy integration in the Gadget JS),
- There are built-in samples in Nuxeo Gadget (see below).

If you have already built custom automation chains to expose a high level API on top of Nuxeo, you can easily expose theses services via an OpenSocial gadget.

Gadgets in the Nuxeo Platform

Apache Shindig, GWT and WebEngine

OpenSocial integration in Nuxeo is based on Apache Shindig project. Nuxeo OpenSocial modules were originally contributed by Leroy Merlin and include:

- Shindig integration,
- A WebEngine based gadget spec webapp (`/nuxeo/site/gadgets/`),
- A GWT based gadget container,
- A service to contribute new gadgets.

Gadget Spec Generation

Gadget Spec is the XML file defining the gadget.

It's a static XML file that:

- Describes the gadget,
- Lists the required features of the gadget,
- Contains the translation resources,
- Contains the authentication information,
- Contains the JavaScript files and HTML content,
- ...

Inside Nuxeo, the gadget spec is dynamically generated from a FreeMarker template using WebEngine.

Using WebEngine and Freemarker to generate the spec adds more flexibility and power.

Better Reusability

Most of you gadgets will share some common content. For JavaScript it's easy to manage, but for HTML and XML content it not that simple.

In order to avoid that issue, when writing a gadget spec in Nuxeo, you can use FreeMarker includes that will be resolved:

- locally to the gadget directory,
- globally in the `webengine-gadget` bundle (in `skin/resources/ftl`).

The same logic applies for JS et CSS resources that will be resolved:

- locally to the gadget directory,
- in the gadget bundle,
- globally in the `webengine-gadget` bundle (in `skin/resources/scripts` or `skin/resources/css`).

Better Context Management

When generating the gadget spec you need to take into account several parameters:

- URLs: Resources URLs may be dependent of your config:
  - Nuxeo host name may change,
  - Some resources are accessed from the client side (from the browser),
  - Some resources are accessed from the server side (Server to Server communication);
- Authentication: Depending on the target host of the gadget authentication config may change:
  - You want to use 2 legged integrated authentication when the gadget is inside Nuxeo,
  - You want to use 3 legged authentication when gadget is hosted inside an external application.

In order to address these problems the Freemarker template is rendered against the following context:

Variable name	Description
`spec`	Gives access to the InternalGadgetDescriptor object.
`serverSideBaseUrl`	Server side URL used to access the Nuxeo server.
`clientSideBaseUrl`	Client side URL used to access the Nuxeo server.
`contextPath`	Context path of the Nuxeo WebApp (i.e. Nuxeo).
`insideNuxeo`	Boolean flag used to tell if the gadget will be renderd inside Nuxeo or in an external application.
`jsContext`	Automatically generated a string that can be used to dump the FreeMarker context in JavaScript.
`i18n`	Access to a Java Helper class to manage i18n (see later).
`specDirectoryUrl`	Base URL for accessing the gadget virtual directory.
`contextHelper`	Access to a Java Helper class to manage the "Nuxeo context" of the gadget if needed: Repository name, domain path ...

The inside flag is "automatically" determined by Nuxeo, but in case the consumer application and Nuxeo are on the same host (i.e. communication via localhost or 127.0.0.1), you may force the external mode by adding `external=true` to the gadget spec URL:

```
http://127.0.0.1:8080/nuxeo/site/gadgets/userworkspaces/userworkspaces.xml?external=true
```

All request parameters you may pass in the gadget spec URL will also be available at the root of the FreeMarker context.

Integrated Internationalization

Gadget spec needs to provide XML files for all translations.

```
<Locale lang="fr" messages="/nuxeo/site/gadgets/automation/messages_fr.xml"/>
<Locale lang="de" messages="/nuxeo/site/gadgets/automation/messages_de.xml"/>
<Locale lang="it" messages="/nuxeo/site/gadgets/automation/messages_it.xml"/>
<Locale lang="es" messages="/nuxeo/site/gadgets/automation/messages_es.xml"/>
<Locale lang="pt" messages="/nuxeo/site/gadgets/automation/messages_pt.xml"/>
<Locale lang="pl" messages="/nuxeo/site/gadgets/automation/messages_pl.xml"/>
```

So, of course you can use static resources for these translation files. The main problem is that in this case you can not leverage the existing translations in Nuxeo and it's hard to contribute new translations. In order to avoid that, Nuxeo proposes a dynamic translation mode, where messages files are dynamically generated based on the standard messages files used in the JSF webapp.

To activate the dynamic i18n features you need:

- To provide a `dynamic_messages.properties` file that will list the needed label codes and associated default values;
- To use the include that will automatically generate the needed translation files and associated entries in the XML spec.

```
<#include "dynamic-translations.ftl"/>
```

Dynamic Management of Authentication

As explained earlier, depending of the target context, the spec needs to define OAuth 3 legged or 2 legged. You can do it by hand, or you can use the OAuth include:

```
 <#include "default-oauth-prefs.ftl"/>
```

Gadget Toolbox

In order to make gadget creation easier, Nuxeo provides some building blocks for common features:

- Context management,
- Automation REST calls,
- Documents list display.

These building blocks are composed of FreeMarker includes, JavaScripts files and CSS files.

Thanks to these building blocks, writing full featured gadgets based on Automation API is very simple.

```
<Module>
  <ModulePrefs title="Nuxeo Automation"
    description="Simple gadget that uses Nuxeo REST Automation API "
    author="tdeprat" author_email="tdelprat@nuxeo.com"
    height="420">
    <#include "dynamic-translations.ftl"/>
    <Require feature="dynamic-height" />
    <#include "default-oauth-prefs.ftl"/>
  </ModulePrefs>
  <Content type="html">
<![CDATA[
<html>
  <head>
  <link rel="stylesheet" type="text/css"
href="${specDirectoryUrl}documentlists.css"/>
  <!- insert JS Context ->
  ${jsContext}
  <!-- insert JS Automation script -->
  <script src="${specDirectoryUrl}default-automation-request.js"></script>
  <!-- insert default document list display script -->
```

```
  <script src="${specDirectoryUrl}default-documentlist-display.js"></script>
  <script>
   var prefs = new _IG_Prefs(_MODULE_ID_);
   // configure Automation REST call
   var NXRequestParams={ operationId : 'Document.PageProvider',            // id of
operation or chain to execute
     operationParams : { query : "Select * from Document", pageSize : 5},  //
parameters for the chain or operation
     operationContext : {},                                                //
context
     operationDocumentProperties : "common,dublincore",                    // schema
that must be fetched from resulting documents
     entityType : 'documents',                                             // result
type : only document is supported for now
     usePagination : true,                                                 // manage
pagination or not
     displayMethod : displayDocumentList,                                  // js
method used to display the result
     displayColumns : [{ type: 'builtin', field: 'icon'},                 //
minimalist layout listing
                      { type: 'builtin', field: 'titleWithLink', label:
'__MSG_label.dublincore.title__'},
                      { type: 'date', field: 'dc:modified', label:
'__MSG_label.dublincore.modified__'},
                      { type: 'text', field: 'dc:creator', label:
'__MSG_label.dublincore.creator__'}
                      ]
   };
   // execute automation request onload

gadgets.util.registerOnLoadHandler(function(){doAutomationRequest(NXRequestParams);}
);
  </script>
  </head>
   <body>
   <#include "default-documentlist-layout.ftl"/>
   <#include "default-request-controls.ftl"/>
  </body>
</html>
]]>
```

```
  </Content>
</Module>
```

OpenSocial configuration

OpenSocial in Nuxeo can be configured throug the GWT Container parameters.

GWT Container parameters

There are some parameters you can pass to the GWT container, through the getGwtParams() function, to customize the way it works.

Here are the definitions of the different parameters:

- `dndValidation`: `'true'` if the container should wait the validation of the Drag'n Drop before doing the actual move, `'false'` otherwise. If the parameter is not present, default to `'false'`.
- `showPreferences`: `'true'` if the gadget preferences need to be displayed after adding a gadget, `'false'` otherwise. If the parameter is not present, default to `'true'`.
- `resetGadgetTitle`: `'true'` if the gadget title needs to be after its addition to the container, `'false'` otherwise. If the parameter is not present, default to `'true'`.
- `userLanguage`: this parameter is used to store the user language. The user language is used to internationalize the gadgets title by creating the corresponding Locale. If this parameter is not present, we fallback on the default Locale when trying to retrieve the label.

Cross-Origin Resource Sharing (CORS)

If you do cross-domain requests from any JavaScript client to access WebEngine resources or Automation APIs, there's a chance that your browser forbids it. Since version 5.7.2, CORS allows you to communicate with Nuxeo from another domain using `XMLHttpRequests`.

Nuxeo uses a filter to handle those cases. It is based on Vladimir Dzhuvinov's universal CORS filter, and allows you to configure on which URLs cross-origin headers are needed. You'll be able to configure each URL independently.

Here is a the simplest contribution, to allow cross-domain request on the whole `foobar` site:

In this section

Simplest contribution

```
<extension
target="org.nuxeo.ecm.platform.web.common.requestcontroller.service.RequestControllerS
ervice" point="corsConfig">
    <corsConfig name="foobar">
      <pattern>/nuxeo/site/foobar/.*</pattern>
    </corsConfig>
 </extension>
```

Configuration

Here is the list of all contribution attributes. There are all optional.

Attribute name	Description	Default value	Possible values ("|" separates possible values)

`allowGenericHttpRequests`	If false, only valid and accepted CORS requests that be allowed (strict CORS filtering).	`true`	`true` | `false`
`allowOrigin`	The whitespace-separated list of origins that the CORS filter must allow.	`*`	`* | http://example.com http://example.com:8080`
`allowSubdomains`	If true the CORS filter will allow requests from any origin which is a sub-domain origin of the allowed origins.	`false`	`true` | `false`
`supportedMethods`	The list of the supported HTTP methods.	`GET, POST, HEAD, OPTIONS`	"," separates list of HTTP methods
`supportedHeaders`	The names of the supported author request headers.	`*`	* | "," separates list of headers
`exposedHeaders`	The list of the response headers other than simple response headers that the browser should expose to the author of the cross-domain request through the `XMLHttpRequest.getResponseHeader()` method.	`-`	"," separates list of headers
`supportsCredentials`	Indicates whether user credentials, such as cookies, HTTP authentication or client-side certificates, are supported.	`true`	`true` | `false`
`maxAge`	Indicates how long the results of a preflight request can be cached by the web browser, in seconds.	`-1`	`integer`

For instance, a `fooly` complete contribution could looks like:

Fooly contribution

```
<extension
target="org.nuxeo.ecm.platform.web.common.requestcontroller.service.RequestControllerS
ervice" point="corsConfig">
        <corsConfig name="fooly" allowGenericHttpRequests="true"
          allowOrigin="http://example.com http://example.com:8080"
          allowSubdomains="true" supportedMethods="GET"
          supportedHeaders="Content-Type, X-Requested-With"
          exposedHeaders="X-Custom-1, X-Custom-2"
          supportsCredentials="false" maxAge="3600">
          <pattern>/fooly/site/.*</pattern>
        </corsConfig>
      </extension>
```

Making sure the contribution is taken into account

To debug your CORS configuration, you might use cURL and look at the response. If you haven't blocked OPTIONS method, you should test with the preflight request for an expected POST request:

Simulate preflight request

```
curl --verbose -H "Origin: http://www.nuxeo.com" -H "Access-Control-Request-Method:
POST" -H "Access-Control-Request-Headers: X-Requested-With" -X OPTIONS
http://localhost:8080/nuxeo/site/foobar/upload
```

With the default configuration, preflight's response must looks like:

Default response

```
< HTTP/1.1 200 OK
< Server: Apache-Coyote/1.1
< Access-Control-Allow-Origin: http://www.nuxeo.com
< Access-Control-Allow-Credentials: true
< Access-Control-Allow-Methods: HEAD, POST, GET, OPTIONS
< Access-Control-Allow-Headers: X-Requested-With
< Content-Length: 0
```

With these "Access-Control-Allow-*" headers containing expected values.

Legacy Restlets

Restlets used to be the main way of exposing a REST API in Nuxeo. It is no more the case and this module is here for compatibility purpose as some part of the UI still rely on it. You should use the new REST API if you want to use or extend the REST exposition of Nuxeo.

In this section

- Restlets in Nuxeo

Restlets in Nuxeo

The Restlet framework is integrated in Nuxeo since version 5.1 and has been used to expose REST API on top of the Platform. Restlet has been replaced by JAX-RS /Jersey, but Nuxeo still embeds Restlets:

- Because of lot of "old" REST APIs are still used and works perfectly with Restlets,
- Because Restlets have been integrated with Seam (so you can easily mix Seam JS and Restlets calls since they share the same server side conversation).

Unlike Content Automation, Restlets in Nuxeo were never targeted at providing a uniform high level API, these are just helpful REST APIs exposed for:

- Seam Remoting usage,
- Browser helpers usage,
- JavaScript usage.

You can see what Rest APIs are exposed via Restlets by looking and the configured Restlets: http://explorer.nuxeo.org/nuxeo/site/distribution/Nuxeo%20Platform-5.8/viewExtensionPoint/org.nuxeo.ecm.platform.ui.web.restAPI.service.PluggableRestletService--restlets.

Directories and Vocabularies

In the Nuxeo Platform, a **directory** is a source of (mostly) table-like data that lives outside of the VCS document storage database. A directory is typically a connection to an external data source that is also access by other processes than the Nuxeo Platform itself (therefore allowing shared management and usage).

A **vocabulary** is a specialized **directory** with only a few important columns that are used by the Nuxeo Platform to display things like menus and selection lists.

SQL Directories

SQL directories read and store their information in a SQL database. They are defined through the `dir`

ectories extension point of the org.nuxeo.ecm.directory.sql.SQLDirectoryFactory component.

The **directory** element must contain a number of important sub-elements:

- **name** : The name of the directory, used for overloading and in application code;
- **schema** : The schema describing the columns in the directory;
- **dataSource** : The JDBC datasource defining the database in which the data is stored;
- **table** : The SQL table in which the data is stored;
- **idField** : The primary key in the table, used for retrieving entries by id;
- **autoincrementIdField** : Whether the idField is automatically incremented. This value is most of the time at false, because the identifier is a string;
- **querySizeLimit** : The maximum number of results that the queries on this directory should return. If there are more results than this, an exception will be raised;
- **dataFile** : File from which data is read to populate the table, depending on the following element;
- **createTablePolicy** : Indicates how the dataFile will be used to populate the table. Three values are allowed: **never** if the dataFile is never used (the default), **on_missing_columns** if the dataFile is used to create missing columns (when the table is created or each time a new column is added, due to a schema change), **always** if the dataFile is used to create the table as each restart of the application server;
- **cacheTimeout** : The timeout (in seconds) after which an entry is not kept in the cache anymore. The default is 0 which means never time out;
- **cacheMaxSize** : The maximum number of entries in the cache. The default is 0 and means to not use entries caching at all;
- **readOnly** : If the directory should be read-only;
- **substringMatchType** : How a non-exact match is done, possible values are subany, sub initial or subfinal; this is used in most UI searches.

In this section

The following is used by the UI if the directory is a hierarchical vocabulary:

- **parentDirectory** : The parent directory to use.

The following are used only if the directory is used for authentication:

- **password** : Field from the table which contain the passwords;
- **passwordHashAlgorithm** : The has algorithm to use to store new passwords. Allowed values are SSHA and SMD5. The default (nothing specified) is to store passwords in clear.
  Example:

```
<?xml version="1.0"?>
<component name="com.example.project.directories.sql">
  <extension target="org.nuxeo.ecm.directory.sql.SQLDirectoryFactory"
      point="directories">
    <directory name="continent">
      <schema>vocabulary</schema>
      <dataSource>java:/nxsqldirectory</dataSource>
      <cacheTimeout>3600</cacheTimeout>
      <cacheMaxSize>1000</cacheMaxSize>
      <table>continent</table>
      <idField>id</idField>
      <autoincrementIdField>false</autoincrementIdField>
      <dataFile>directories/continent.csv</dataFile>
      <createTablePolicy>on_missing_columns</createTablePolicy>
    </directory>
  </extension>
</component>
```

LDAP Directories

LDAP directories store information in a LDAP database. They are defined through the `servers` and `directories` extension points of the `org.nuxeo.ecm.directory.ldap.LDAPDirectoryFactory` component.

First, one or more servers have to be defined. A **server** defines:

- **`name`** : The name of the server which will be used in the declaration of the directories;
- **`ldapUrl`** : The address of the LDAP server, in `ldap://` or `ldaps://` form. There can be several such tags to leverage clustered LDAP configurations;
- **`bindDn`** : The Distinguished Name used to bind to the LDAP server;
- **`bindPassword`** : The corresponding password.

The bind credentials are used by the Nuxeo Platform to browse, create and modify all entries (irrespective of the actual Nuxeo user these entries may represent).

Optional parameters are:

- **`connectionTimeout`** : The connection timeout (in milliseconds), the default is 10000 (10 seconds);
- **`poolingEnabled`** : Whether to enable internal connection pooling (the default is true).
  Example:

```
<?xml version="1.0"?>
<component name="com.example.project.directories.ldap.srv">
  <extension target="org.nuxeo.ecm.directory.ldap.LDAPDirectoryFactory"
      point="servers">
    <server name="default">
      <ldapUrl>ldap://localhost:389</ldapUrl>
      <bindDn>cn=nuxeo,ou=applications,dc=example,dc=com</bindDn>
      <bindPassword>secret</bindPassword>
    </server>
  </extension>
</component>
```

Once you have declared the server, you can define new LDAP **directories**. The sub-elements of the **directory** element are:

- **`name`** , **`schema`** , **`idField`** and **`passwordField`** : same as for SQL directories;
- **`searchBaseDn`** : Entry point into the server's LDAP tree structure; searches are only made below this root node;
- **`searchClass`** : Restricts the type of entries to return as result;
- **`searchFilter`** : Additional LDAP filter to restrict the search results;
- **`searchScope`** : The scope of the search. It can take two values:`onelevel` to search only under the current node, or `subtree` to

search in the whole subtree;
- **substringMatchType** : Defines how the query is built using wildcard characters. Three different values can be provided:
  - subany: wildcards are added around the string to match (as *foo*);
  - subinitial: wildcard is added before the string (*bar);
  - subfinal: wildcard is added after the string (baz*). This is the default behavior;
- **readOnly** : Boolean value. When set to false, this parameter allows to create new entries or modify existing ones in the LDAP server;
- **cacheTimeout** : Cache timeout in seconds;
- **cacheMaxSize** : Maximum number of cached entries before global invalidation;
- **creationBaseDn** : Entry point in the server's LDAP tree structure where new entries will be created. Useless to provide if the readOnly attribute is set to true;
- **creationClass** : Use as many tag as needed to specify which classes are used to define new people entries in the LDAP server. Example:

```
<?xml version="1.0"?>
<component name="com.example.project.directories.ldap.dir">
  <extension target="org.nuxeo.ecm.directory.ldap.LDAPDirectoryFactory"
      point="directories">
    <directory name="userDirectory">
      <server>default</server>
      <schema>user</schema>
      <idField>username</idField>
      <passwordField>password</passwordField>

      <searchBaseDn>ou=people,dc=example,dc=com</searchBaseDn>
      <searchClass>person</searchClass>
      <searchFilter>(&amp;(sn=foo*)(myCustomAttribute=somevalue))</searchFilter>
      <searchScope>onelevel</searchScope>
      <substringMatchType>subany</substringMatchType>

      <readOnly>false</readOnly>

      <cacheTimeout>3600</cacheTimeout>
      <cacheMaxSize>1000</cacheMaxSize>

      <creationBaseDn>ou=people,dc=example,dc=com</creationBaseDn>
      <creationClass>top</creationClass>
      <creationClass>person</creationClass>
      <creationClass>organizationalPerson</creationClass>
      <creationClass>inetOrgPerson</creationClass>
    </directory>
  </extension>
</component>
```

Multi-directories

Multi-directories are used to combine values coming from different directories. They are defined through the `directories` extension point of the `org.nuxeo.ecm.directory.multi.MultiDirectoryFactory` component.

A multi-directory is made up of one or more **sources**. Each source aggregates one or more **sub-directories**.

A **source** defines:

- **name**: The source name, for identification purposes;
- **creation**: `true` when new entries should be created in this source (default is `false`);
- **subDirectory**: One or more sub-directories.

A **subDirectory** has:

- **name**: The name of a valid directory, from which data will be read and written;
- **optional**: `true` if the sub-directory may have no info about a given entry without this being an error (default is `false`);
- **field**: Zero or more field mapping between the underlying sub-directory and the name it should have in the multi-directory.

A **field** element is of the form: `<field for="foo">bar</field>`. This means that the field `foo` of the underlying directory will be turned into a

field named `bar` in the multi-directory.

When an entry is requested from the multi-directory, each source will be consulted in turn. The first one that has an answer will be used. In a source, the fields of a given entry will come from all the sub-directories, with appropriate field name re-mapping. Each sub-directory has part of the entry, keyed by its main id (which may be remapped).

For the creation of new entries, only the sources marked for *creation* are considered.
Example:

```
<?xml version="1.0"?>
<component name="com.example.project.directories.multi">
  <extension target="org.nuxeo.ecm.directory.multi.MultiDirectoryFactory"
      point="directories">
    <directory name="mymulti">
      <schema>someschema</schema>
      <idField>uid</idField>
      <passwordField>password</passwordField>
      <source name="sourceA" creation="true">
        <subDirectory name="dir1">
          <field for="thefoo">foo</field>
        </subDirectory>
        <subDirectory name="dir2">
          <field for="uid">id</field>
          <field for="thebar">bar</field>
        </subDirectory>
      </source>
      <source name="sourceB">
        ...
      </source>
    </directory>
  </extension>
</component>
```

References Between Directories

Directory references leverage two common ways of string relationship in LDAP directories.

Static Reference as a DN-Valued LDAP Attribute

The static reference strategy is to apply when a multi-valued attribute stores the exhaustive list of distinguished names of reference entries, for example the uniqueMember of the `groupOfUniqueNames` object.

```
<ldapReference field="members" directory="userDirectory"
  staticAttributeId="uniqueMember" />
```

The `staticAttributeId` attribute contains directly the value which can be read and manipulated.

Dynamic Reference as a ldapUrl-Valued LDAP Attribute

The dynamic attribute strategy is used when a potentially multi-value attribute stores a LDAP URL intensively, for example the `memberURL`'s attribute of the `groupOfURL` object class.

```
<ldapReference field="members" directory="userDirectory"
  forceDnConsistencyCheck="false"
  dynamicAttributeId="memberURL" />
```

The value contained in `dynamicAttributeId` looks like `ldap:///ou=groups,dc=example,dc=com??subtree?(cn=sub*)` and will be

resolved by dynamical queries to get all values. The `forceDnConsistencyCheck` attribute will check that the value got through the dynamic resolution correspond to the attended format. Otherwise, the value will be ignored. Use this check when you are not sure of the validity of the distinguished name.

LDAP Tree Reference

The LDAP tree reference can be used to resolve children in the LDAP tree hierarchy.

```
<ldapTreeReference field="children" directory="groupDirectory"
  scope="subtree" />
```

The field has to be a list of strings. It will resolve children of entries in the current directory, and look them up in the directory specified in the reference.The scope attribute. Available scopes are "onelevel" (default), "subtree". Children with same id than parent will be filtered. An inverse reference can be used to retrieve the parent form the children entries. It will be stored in a list, even if there can be only 0 or 1 parent.

Edit is NOT IMPLEMENTED: modifications to this field will be ignored when saving the entry.

Defining Inverse References

Inverse references are defined with the following declarations:

```
<references>
  <inverseReference field="groups" directory="groupDirectory"
    dualReferenceField="members" />
</references>
```

This syntax should be understood as "the member groups value is an inverse reference on `groupDirectory` directory using members reference". It is the group directory that stores all members for a given group. So the groups of a member are retrieved by querying in which groups a member belongs to.

References Defined by a Many-to-Many SQL Table

TODO OG

Using the Java API Serverside

This page explains how to use the Nuxeo Java API.

Java Services

The Nuxeo Platform contains a built-in notion of service. Services are Java interfaces exposed and implemented by a Component.

From within a Nuxeo Runtime aware context, you can access a service locally (in the same JVM) by simply looking up its interface:

In this section


```
RelationManager rm = Framework.getService(RelationManager.class)
```

You can find the list of existing services on the Nuxeo Platform Explorer. You will also need to understand main Java classes, using the javadoc.

Typical Use Cases

You may want to use this API from:

- A Seam component when customizing the default webapp,
- An Event Listener that would do some specific things,
- A custom operation that would use the built-in services,

- ...

Automation

In this section you'll find information on how to use the Automation service, how to contribute a new chain, a new operation, etc.

If you are not familiar with Content Automation, you should first read the Content Automation introduction in the Architecture section.

Here are the different sub-sections of this chapter:

- Operations Index — The lists of operations and their description is available on the Nuxeo Platform Explorer.
- Contributing an Operation — In order to implement an operation you need to create a Java class annotated with @Operation. An operation class should also provide at least one method that will be invoked by the automation service when executing the operation. To mark a method as executable you must annotate it using @OperationMethod.
- Contributing an Automation Chain — An automation chain is a pipe of parametrized atomic operations. This means the automation chain specifies the parametrization of each operation in the chain and not only the list of operations to execute. Because of this, when executing an automation chain, you should only specify the chain's name. The chain will be fetched from the registry and operations will be executed one after the other using the parametrization present in the chain.
- Automation Service API
- Returning a Custom Result with Automation — As automatic marshalling is not implemented into Automation server and client, only Document(s) and Blob(s) can be manipulated. Therefore, the way to return a custom type is to encapsulate the value in a Blob.
- Automation Exception — Since Nuxeo 5.7.3, Automation provides means to debug and handle exception during the Automation operations and chains calls.
- Automation Tracing
- Use of MVEL in Automation Chains

Operations Index

The lists of operations and their description is available on the Nuxeo Platform Explorer.

Contributing an Operation

Implementing an Operation

In order to implement an operation you need to create a Java class annotated with `@Operation`. An operation class should also provide at least one method that will be invoked by the automation service when executing the operation. To mark a method as executable you must annotate it using `@OperationMethod`.

You can have multiple *executable* methods - one method for each type of input/output objects supported by an operation. The right method will be selected at runtime depending on the type of the input object (and of the type of the required input of the next operation when in an operation chain).

In this section

So, an operation method will be selected if the method argument matches the current input object and the return type matches the input required by the next operation if in an operation chain.

The `@OperationMethod` annotation is also providing an optional priority attribute that can be used to specify which method is preferred over the other matching methods. This situation (having multiple method that matches an execution) can happen because the input and output types are not strictly matched. For example if the input of a method is a DocumentModel object and the input of another method is a DocumentRef object then both methods have the same input signature for the automation framework because DocumentModel and DocumentRef are objects of the same kind - they represent a Nuxeo Document. When you need to treat different Java objects as the same type of input (or output) you must create a type adapter (see the interface `org.nuxeo.ecm.automation.TypeAdapter`) that knows how to convert a given object to another type. Without type adapters treating different Java objects as the same type of object is not possible.

Also operations can provide parametrizable variables so that when an user is defining an operation chain he can define values that will be injected in the operation parameters. To declare parameters you should use the `@Param` annotation.

Apart these annotations there is one more annotation provided by the automation service - the `@Context` annotation. This annotation can be used to inject execution context objects or Nuxeo Service instances into a running operation.

When registering an automation chain the chain will be checked to find a path from the first operation to the last one to be sure the chain can be executed at runtime. Finding a path means to identify at least one method in each operation that is matching the signature of the next operation. If such a path could not be found an error is thrown (at registration time). For more detail on registering an operation chains see Contributing an Automation Chain.

To register your operation you should create a Nuxeo XML extension to the `operations` extension point. Example:

```
<extension target="org.nuxeo.ecm.core.operation.OperationServiceComponent"
    point="operations">
    <operation
      class="org.nuxeo.example.TestOperation" />
  </extension>
```

where `org.nuxeo.example.TestOperation` is the class name of your operation (the one annotated with `@Operation).`

Let's look at the following operation class to see how annotations were used:

```
import org.nuxeo.ecm.automation.core.Constants;
import org.nuxeo.ecm.automation.core.annotations.Context;
import org.nuxeo.ecm.automation.core.annotations.Operation;
import org.nuxeo.ecm.automation.core.annotations.OperationMethod;
import org.nuxeo.ecm.automation.core.annotations.Param;
import org.nuxeo.ecm.automation.core.util.DocumentHelper;
import org.nuxeo.ecm.automation.core.util.Properties;
import org.nuxeo.ecm.core.api.CoreSession;
import org.nuxeo.ecm.core.api.DocumentModel;
import org.nuxeo.ecm.core.api.DocumentModelList;
import org.nuxeo.ecm.core.api.DocumentRef;
import org.nuxeo.ecm.core.api.DocumentRefList;
import org.nuxeo.ecm.core.api.impl.DocumentModelListImpl;

@Operation(id = CreateDocument.ID, category = Constants.CAT_DOCUMENT, label =
"Create", description = "Create a new document in the input folder ...")
public class CreateDocument {

    public final static String ID = "Document.Create";

    @Context
    protected CoreSession session;

    @Param(name = "type")
    protected String type;

    @Param(name = "name", required = false)
    protected String name;

    @Param(name = "properties", required = false)
    protected Properties content;

    @OperationMethod
    public DocumentModel run(DocumentModel doc) throws Exception {
        if (name == null) {
            name = "Untitled";
        }
        DocumentModel newDoc = session.createDocumentModel(
                doc.getPathAsString(), name, type);
        if (content != null) {
            DocumentHelper.setProperties(session, newDoc, content);
        }
```

```
        return session.createDocument(newDoc);
    }

    @OperationMethod
    public DocumentModelList run(DocumentModelList docs) throws Exception {
        DocumentModelListImpl result = new DocumentModelListImpl(
                (int) docs.totalSize());
        for (DocumentModel doc : docs) {
            result.add(run(doc));
        }
        return result;
    }

    @OperationMethod
    public DocumentModelList run(DocumentRefList docs) throws Exception {
        DocumentModelListImpl result = new DocumentModelListImpl(
                (int) docs.totalSize());
        for (DocumentRef doc : docs) {
            result.add(run(session.getDocument(doc)));
        }
        return result;
```

```
    }
}
```

You can see how `@Context` is used to inject the current CoreSession instance into the session member. It is recommended to use this technique to acquire a CoreSession instead of creating a new session. This way you reuse the same CoreSession used by all the other operation in the chain. You don't need to worry about closing the session — the automation service will close the session for you when needed.

You can use `@Context` also to inject any Nuxeo Service or the instance of the `OperationContext` object that represents the current execution context and that holds the execution state — like the last input, the context parameters, the core session, the current principal etc.

The attributes of the `@Operation` annotation are required by operation chain creation tools like the one in Nuxeo Studio to be able to generate the list of existing operations and some additional operation information - like its name, a short description on how the operation is working etc. For a complete description of these attributes look into the annotation Javadoc.

You can see the operation above provides three operation methods with different signatures:

- One that takes a Document and return a Document object,
- One that takes a list of document objects and return a list of documents,
- One that takes a list of document references and return a list of documents.

Depending on what the input object is when calling this operation, only one of these methods will be used to do the processing. You can notice that there is no method taking a document reference. This is because the document reference is automatically adapted into a DocumentModel object when needed thanks to a dedicated TypeAdapter.

The initial input of an operation (or operation chain) execution is provided by the caller (the one that create the execution context). The Nuxeo Platform provides several execution contexts:

- An core event listener that executes operations in response to core events,
- An action bean that executes operations in response to the user actions (through the Nuxeo Web Interface),
- A JAX-RS resource which executes operations in response to REST calls,
- A special listener fired by the workflow service to execute an operation chain.

Each of these execution contexts are providing the initial input for the chain (or operation) to be executed. For example the core event listener will use as the initial input the document that is the source of the event. The action bean executor will use the document currently opened in the User Interface.

If no input exists then **null** will be used as the input. In that case the first operation in the chain must be a **void** operation.
If you need you can create your own operation executor. Just look into the existing code for examples (e.g. `org.nuxeo.ecm.automation.jsf.OperationActionBean`).

The code needed to invoke an operation or an operation chain is pretty simple. You need to do something like this:

```
CoreSession session = fetchCoreSession();
  AutomationService automation = Framework.getService(AutomationService.class);
  OperationContext ctx = new OperationContext(session);
  ctx.setInput(navigationContext.getCurrentDocument());
  try {
    Object result = automation.run(ctx, "the_chain_name");
    // ... do something with the result
  } catch (Throwable t) {
    // handle errors
  }
```

To invoke operations is a little more complicated since you also need to set the operation parameters.

Let's look again at the operation class defined above. You can see that operation parameters are declared as class fields using the `@Param` a nnotation.

This annotation has several attributes like a parameter name, a required flag, a default value if any, a widget type to be used by UI operation chain builders like Nuxeo Studio etc.

The parameter name is important since it is the key you use when defining an operation chain to refer to a specific operation parameter. If the parameter is required then its value must be specified in the operation chain definition otherwise an exception is thrown at runtime. The other parameters are useful only for UI tools that introspect the operations. For example when building an operation chain in Nuxeo Studio you need to render each operation parameter using a widget. The default is to use a TextBox if the parameter is a String, a CheckBox if the parameter is a boolean, a ListBox for lists etc. But in some situations you may want to override this default mapping — for example you may

want to use a TextArea instead of a TextBox for a string parameter: in that case you can use the `widget` attribute to specify your desired widget.

Parameter Injection

Executing an operation is done as following:

1. A new operation instance is created (operations are stateless).
2. The context objects are injected if any `@Context` annotation is present.
3. Corresponding parameters specified by the execution context are injected into the fields annotated using `@Param` and identified using the name attribute of the annotation.
4. The method matching the execution input and output types is invoked by passing as argument the current input. (Before invoking the method the input is adapted if any TypeAdapter was registered for the input type.)

Let's look on how parameters are injected into the instance fields.

So, first the field is identified by using the parameter name. Then the value to be injected is checked to see if the value type match with the field type. If they don't match the registered TypeAdapters are consulted for an adapter that knows how to adapt the value type into the field type. If no adapter is found then an exception is thrown otherwise the value is adapted and injected into the parameter.
An important case is when EL expressions are used as values. In that case (if the value is an expression) then the expression is evaluated and the result will be used as the value to be injected (and the algorithm of type matching described above is applied on the value returned by the expression).

This means you can use for almost all field types string values since a string adapter exists for almost all parameter types used by operations.

Here is a list of the most used parameter types and the string representation for each of these types (the string representation is important since you should use it when defining operation chains through Nuxeo XML extensions):

- **document**. Java type: `org.nuxeo.ecm.core.api.DocumentModel`
  Known adapters: from string, from DocumentRef
  String representation: the document UID or the document absolute path. Example: "96bfb9cb-a13d-48a2-9bbd-9341fcf24801", "/default-domain/workspaces/myws" etc.
- **documents**. Java type: `org.nuxeo.ecm.core.api.DocumentModelList`
  Known adapters: from DocumentRefList, from DocumentModel, from DocumentRef
  No String representation exists. Cannot be used as a parameter value in an XML chain descriptor. You should use EL expressions instead.
- **blob**. Java type: `org.nuxeo.ecm.core.api.Blob`
  No String representation exists. Cannot be used as a parameter value in an XML chain descriptor. You should use EL expressions instead.
- **blobs**. Java type: `org.nuxeo.ecm.automation.core.util.BlobList`
  No String representation exists. Cannot be used as a parameter value in an XML chain descriptor. You should use EL expressions instead.
- **properties**. Java type: `org.nuxeo.ecm.automation.core.util.Properties`
  Known adapters: from string.
  String representation: a list of key value pairs in Java properties file format.
- **resource**. Java type: `java.net.URL`
  Known adapters: from string.
- **script**. Java type: `org.nuxeo.ecm.automation.core.scripting.Expression`
  String representation: Use the "expr:" prefix before your EL expression.
  Example: "`expr: Document.title`"
  For the complete list of scripting objects and functions see Use of MVEL in Automation Chains.
- **date**. Java type: `java.util.Date`.
  Known type adapters: from string and from `java.util.Calendar`
  String representation: W3C date format.
- **integer**. Java type: `java.lang.Long` or the long primitive type.
  Natural string representation.
- **float**. Java type: `java.lang.Double` or the double primitive type.
  Natural string representation.
- **boolean**. Java type: `java.lang.Boolean` or the boolean primitive type.
  Natural string representation.
- **string**. Java type: `java.lang.String`
  Already a string.

- **stringlist**. Java Type: `org.nuxeo.ecm.automation.core.util.StringList`
  Known adapters: from string
  String representation: comma separated list of strings. Example: "foo, bar"

Of course, when defining the parameter values that will be injected into an operation you can either specify static values (as hard coded strings) either specify an EL expression to compute the actual value at runtime.

Void Operation Methods

Sometimes operations may not require any input. In that case the operation should use a method with nop parameters. Such methods will match any input - thus it is not indicated to use two void methods in the same operation - since you cannot know which method will be selected for execution.

This is the case for all *fetch* like operations (that are fetching objects from a context). For example a Query operation is not requiring an input since it is only doing a query on the repository. This is the definition of the Query operation:

```
import org.nuxeo.ecm.automation.core.Constants;
import org.nuxeo.ecm.automation.core.annotations.Context;
import org.nuxeo.ecm.automation.core.annotations.Operation;
import org.nuxeo.ecm.automation.core.annotations.OperationMethod;
import org.nuxeo.ecm.automation.core.annotations.Param;
import org.nuxeo.ecm.core.api.CoreSession;
import org.nuxeo.ecm.core.api.DocumentModelList;

@Operation(id = Query.ID, category = Constants.CAT_FETCH, label = "Query",
description = "Perform a query on the repository. The query result will become the
input for the next operation.")
public class Query {

    public static final String ID = "Document.Query";

    @Context
    protected CoreSession session;

    @Param(name = "query")
    protected String query;

    @Param(name = "language", required = false, widget = Constants.W_OPTION, values
= { "NXQL", "CMISQL"})
    protected String lang = "NXQL";

    @OperationMethod
    public DocumentModelList run() throws Exception {
        // TODO only NXQL is supported for now
        return session.query(query);
    }
}
```

Also there are rare cases when you don't want to return anything from an operation. In that case the operation method must use the **void** Jav a keyword and the result of the operation will be the **null** Java object.

Contributing an Automation Chain

An automation chain is a pipe of parametrized atomic operations. This means the automation chain specifies the parametrization of each operation in the chain and not only the list of operations to execute. Because of this, when executing an automation chain, you should only specify the chain's name. The chain will be fetched from the registry and operations will be executed one after the other using the parametrization present in the chain.

Chain contribution is done via the Nuxeo extension point mechanism. The extension point name is `chains` and the component exposing the extension point is `org.nuxeo.ecm.core.operation.OperationServiceComponent`.

Here is an example of a chain extension:

```
<extension target="org.nuxeo.ecm.core.operation.OperationServiceComponent"
point="chains">
    <chain id="downloadAllChain">
     <param type="string" name="chainParameterName">chainParameterValue</param>
     <operation id="Context.FetchDocument"/>
      <operation id="Context.SetVar">
        <param type="string" name="name">aoname</param>
        <param type="object" name="value">expr:@{Document["dc:title"]}</param>
      </operation>
      <operation id="Document.GetChildren"/>
      <operation id="Blob.Get">
        <param type="string" name="xpath">file:content</param>
      </operation>
      <operation id="Blob.CreateZip">
        <param type="string" name="filename">expr:@{aoname}.zip</param>
      </operation>
      <operation id="Seam.DownloadFile"/>
    </chain>
  </extension>
```

This is defining a chain that will do the following:

- Fetching the current document in user interface and setting it as the input of the next operation.
- Setting a context variable named "aoname" to the value of the input document title. The input document is returned as the input for the next operation.
- Getting the children of the input document. The list of children documents is returned as the input of the next operation.
- For each document in the list getting the attached blob (the blob property to use is specified using the `xpath` parameter) and returning the list of blobs as the input of the next operation.
- Creating a zip from the input list of blobs. The `filename` parameter is setting the file name to be used for the zip. The value of the file name is retrieved from the "aoname" context variable that was set before in the chain. Returning the zip blob as the input of the next operation.
- Showing the download file to download the input zip from the browser (this is a UI operation).

To summarize, this chain gets the current document in the user interface, extracts all blobs from its direct children, zips these blobs and offers to download the resulted zip file in the web browser.

You can see that the chain is specifying in order each operation that should be executed along with the parameter values to be used at runtime. The parameters are either hard coded strings or EL expressions that allow dynamic computation of actual values.

An atomic operation in a chain is uniquely identified by its ID. Each parameter should specify the name of the operation parameter to set (see `@Param` annotation in Contributing an Operation) and the type of the value to inject. The type is a hint to the chain compiler to correctly transform the string into an injectable Java object.

Since 5.7.2, all chains can contain parameters as operation to be used from the automation context along their execution (such as the `chainParameterName` which can be fetched from the automation sub-context `@{ChainParameters['chainParameterName ']}`).

You can find the complete list of the supported types in Contributing an Operation.

Since 5.7.2, it is possible to create "composite operations": putting some chains into chain.

Here is an example of how to contribute this kind of automation chain:

```
<extension point="chains"
    target="org.nuxeo.ecm.core.operation.OperationServiceComponent">
    <chain id="contributedchain">
      <operation id="contributedchainLeaf" />
      <param type="string" name="messageChain" />
      <operation id="operation1">
        <param type="string" name="message">Hello 1!</param>
      </operation>
      <operation id="operation2">
        <param type="string" name="message">Hello 2!</param>
      </operation>
      <operation id="operation3">
        <param type="string"
name="message">expr:@{ChainParameters['messageChain']}</param>
      </operation>
    </chain>
    <chain id="contributedchainLeaf">
      <operation id="operation1">
        <param type="string" name="message">Hello 1!</param>
      </operation>
      <operation id="operation2">
        <param type="string" name="message">Hello 2!</param>
      </operation>
    </chain>
  </extension>
```

The `contributedchainleaf` chain is contributed with its operations and is included as an operation into `contributedchain`. During the execution of this chain, a validation is running to check if all inputs/outputs of the different chains/operations in the stack are matching.

Automation Service API

On server side, AutomationService can be used to:

- Run contributed chain with chain/operations
- Run runtime chain created on the fly
- Run contributed operation

This service provides chain(s)/operation(s) parameters setting and OperationContext instantiation to inject Automation input(s).

In this section

Run contributed chain with chain/operations

Chain Contribution:

```
<extension point="chains"
            target="org.nuxeo.ecm.core.operation.OperationServiceComponent">
<chain id="contributedchain">
      <param type="string" name="paramChain" />
      <operation id="o1">
        <param type="string" name="param1">Hello 1!</param>
      </operation>
      <operation id="o2">
        <param type="string" name="param2">Hello 2!</param>
      </operation>
</chain>
</extension>
```

Automation Service API - with chain parameter setting:

```
org.nuxeo.ecm.core.api.DocumentModel doc;
org.nuxeo.ecm.automation.AutomationService service;
org.nuxeo.ecm.core.api.CoreSession session;

// Input setting
org.nuxeo.ecm.automation.OperationContext ctx = new OperationContext(session);
ctx.setInput(doc);
// Setting parameters of the chain
Map<String, Object> params = new HashMap<String, Object>();
params.put("paramChain", "Hello i'm a parameter chain!");
// Run Automation service
service.run(ctx, "contributedchain", params);
```

Run runtime chain created on the fly

Automation Service API - with chain/operations parameters setting:

```
org.nuxeo.ecm.core.api.DocumentModel doc;
org.nuxeo.ecm.automation.AutomationService service;
org.nuxeo.ecm.core.api.CoreSession session;

// Input setting
org.nuxeo.ecm.automation.OperationContext ctx = new OperationContext(session);
ctx.setInput(doc);
org.nuxeo.ecm.automation.OperationChain chain = new
OperationChain("notRegisteredChain");
// Adding operations - operations parameters setting
chain.add("Document.Fetch");
chain.add("o1").set("param1", "Hello 1!");
chain.add("o2").set("param2", "Hello 2!");
// Setting parameters of the chain
Map<String, Object> params = new HashMap<String, Object>();
params.put("messageChain", "Hello i'm a chain!");
chain.addChainParameters(params);
// Run Automation service
service.run(ctx, chain);
```

Run contributed operation

Operation Contribution:

```
<extension point="operations"
            target="org.nuxeo.ecm.core.operation.OperationServiceComponent">
    <operation class="org.nuxeo.ecm.automation.core.test.Operation1" />
  </extension>
```

Java Class Operation:

```
@Operation(id = "o1")
public class Operation1 {

    @Param(name = "message")
    protected String message;

    @OperationMethod
    public DocumentModel run(DocumentModel doc) throws Exception {
        return doc;
    }

}
```

Automation Service API - with operations parameters setting:

```
org.nuxeo.ecm.core.api.DocumentModel doc;
org.nuxeo.ecm.automation.AutomationService service;
org.nuxeo.ecm.core.api.CoreSession session;

// Input setting
org.nuxeo.ecm.automation.OperationContext ctx = new OperationContext(session);
ctx.setInput(doc);
// Operation1 parameter setting
Map<String,Object> params = new HashMap<String,Object>();
params.put("message","messageValue");
service.run(ctx, "o1", params);
```

Returning a Custom Result with Automation

As automatic marshalling is not implemented into Automation server and client, only Document(s) and Blob(s) can be manipulated. Therefore, the way to return a custom type is to encapsulate the value in a Blob.

Below is an example, based on the results returned by the method `QueryAndFetch`.

- Operation code

```
package org.nuxeo.support;

import java.io.ByteArrayInputStream;
import java.io.Serializable;
import java.util.Iterator;
import java.util.Map;

import net.sf.json.JSONArray;
import net.sf.json.JSONObject;

import org.nuxeo.ecm.automation.core.Constants;
import org.nuxeo.ecm.automation.core.annotations.Context;
import org.nuxeo.ecm.automation.core.annotations.Operation;
import org.nuxeo.ecm.automation.core.annotations.OperationMethod;
import org.nuxeo.ecm.automation.core.annotations.Param;
import org.nuxeo.ecm.core.api.Blob;
import org.nuxeo.ecm.core.api.CoreSession;
import org.nuxeo.ecm.core.api.IterableQueryResult;
import org.nuxeo.ecm.core.api.impl.blob.InputStreamBlob;
import org.nuxeo.ecm.core.query.sql.NXQL;

@Operation(id = QueryAndFetch.ID, category = Constants.CAT_FETCH, label =
"QueryAndFetch", description = "Sample to show how to return a blob for any
result type.")
public class QueryAndFetch {

    public static final String ID = "Test.QueryAndFetch";

    @Context
    protected CoreSession session;

    @Param(name = "query")
    protected String query;

    protected String lang = NXQL.NXQL;

    @OperationMethod
    public Blob run() throws Exception {
        IterableQueryResult result = session.queryAndFetch(query, lang);
        Iterator<Map<String, Serializable>> it = result.iterator();

        JSONArray array = new JSONArray();
        while (it.hasNext()) {
            Map<String, Serializable> item = it.next();
            JSONObject object = new JSONObject();
            object.accumulateAll(item);
            array.add(object);
        }

        return new InputStreamBlob(new ByteArrayInputStream(
                array.toString().getBytes("UTF-8")), "application/json");

    }

}
```

- Registering this operation

```
<?xml version="1.0"?>
<component name="org.nuxeo.support.operations">

  <extension target="org.nuxeo.ecm.core.operation.OperationServiceComponent"
    point="operations">
    <operation class="org.nuxeo.support.QueryAndFetch" />
  </extension>
</component>
```

- Sample code to use the result from the operation

```
HttpAutomationClient client = new
HttpAutomationClient("http://localhost:8080/nuxeo/site/automation");

Session session = client.getSession(ADMINISTRATOR, ADMINISTRATOR);
Blob response = (Blob) session.newRequest(QueryAndFetch.ID).set("query", "select
ecm:uuid, dc:title, common:icon from Document").execute();
String json = FileUtils.read(response.getStream());
JSONArray array = JSONArray.fromObject(json);
System.out.println("Objects received : " + array.size());
```

Automation Exception

Since Nuxeo 5.7.3, Automation provides means to debug and handle exception during the Automation operations and chains calls.

Automation exception chain can be added to be executed when an error occurs during an Automation execution.

After contributing your custom chains, you can declare your exception chains:

```
<extension point="chainException"
             target="org.nuxeo.ecm.core.operation.OperationServiceComponent">

    <catchChain id="catchChainA" onChainId="contributedchain">
      <run chainId="chainExceptionA" priority="0" rollBack="true" filterId="filterA"/>
      <run chainId="chainExceptionA" priority="0" rollBack="false"
filterId="filterA"/>
      <run chainId="chainExceptionB" priority="10" rollBack="true"
filterId="filterB"/>
    </catchChain>

    <catchChain id="catchChainB" onChainId="anothercontributedchain">
      <run chainId="chainExceptionA" rollBack="false"/>
    </catchChain>

</extension>
```

- This contribution declares a set of execution chains to be executed after '`contributedchain`' failure.
- Here '`chainExceptionA`' and '`chainExceptionB`' are different candidates.
- Filters can be added (see next chapter).
- Rollback is set to defined if we should rollback transaction after '`contributedchain`' failure.
- Priority can be defined to know which chain is going to take the hand on a candidates range. Higher priority wins.

Here is the contribution to deploy filters that can be added to decide which chain is going to be executed:

```
<extension point="automationFilter"
            target="org.nuxeo.ecm.core.operation.OperationServiceComponent">

    <filter id="filterA">expr:@{Document['dc:title']=='Source'}</filter>

    <filter id="filterB">expr:@{Document['dc:title']=='Document'}</filter>

</extension>
```

These filters are strictly written in MVEL template or expression starting by '`expr:`'.

Exception is cached into the operation context and can be used into filters:

```
<extension point="automationFilter"
            target="org.nuxeo.ecm.core.operation.OperationServiceComponent">

    <filter id="filterA">expr:@{Context['Exception']=='NullPointerException'}</filter>

</extension>
```

If two candidate chains have the same priority and both have their filter evaluated to true, the last one contributed is executed.

Automation Tracing

Automation Tracing

Automation chains and operations calls information is collected during their execution by the Automation Trace feature.

This Automation trace mode can be enabled through the nuxeo.conf file properties:

Property	Default value	Description
`org.nuxeo.automation.trace`	false	• You'll benefit from exhaustive logs to debug all automation chain and/or operation execution. • The automation trace mode is disabled by default (not suitable for production). • It can be activated through JMX via `org.nuxeo:TracerFactory` MBean during runtime.
`org.nuxeo.automation.trace.printable`	*	Since Nuxeo 5.7.3, by default, all automation executions are 'printable' (appear in logs) when automation trace mode is on. • You can filter chain and/or operation execution trace printing by setting this property to chain name and/or operation separated by comma. • Comment this property to get all automation chains/operations back in printing (by default set to * (star))

To display traces even for executions without errors, this appender is added by default in your `nuxeo-***-tomcat/lib/log4j` configuration file:

```
<category name="org.nuxeo.ecm.automation.core">
  <priority value="INFO" />
</category>
```

Examples: Simple chain

```
****** chain ******
Name: chainA
Produced output type: DocumentModelImpl


****** Context.FetchDocument ******
Chain ID: chainA
Class: FetchContextDocument
Method:
 'run' | Input Type: interface org.nuxeo.ecm.core.api.DocumentModel |
Output Type: interface org.nuxeo.ecm.core.api.DocumentModel
Input: DocumentModelImpl(5eff3564-1a69-4d91-ae0c-51fb879a6a5a, path=/default-domain,
title=Default domain)
Context Variables | Key: ChainParameters, Value: {}


****** Seam.AddInfoMessage ******
Chain ID: chainA
Class: AddInfoMessage
Method: 'run' | Input Type: void | Output Type: void
Input: DocumentModelImpl(5eff3564-1a69-4d91-ae0c-51fb879a6a5a, path=/default-domain,
title=Default domain)
Parameters  | Name: message, Value: Message INFO
Context Variables | Key: ChainParameters, Value: {}
```

- 'chainA' is executed, produces a document model and runs two operations: `Context.FetchDocument` and `Seam.AddInfoMessage` .
- These two operations display the following information during their executions:
  - Input Type, Output Type,
  - The Input (here the 'default-domain' document),
  - Parameters defined into each operation with their values,
  - Context Variables (injected during execution).

Examples: Composite Chains

```
****** chain ******
Name: chainA
Produced output type: DocumentModelImpl


****** Context.FetchDocument ******
Chain ID: chainA
Class: FetchContextDocument
Method:
 'run' | Input Type: interface org.nuxeo.ecm.core.api.DocumentModel |
Output Type: interface org.nuxeo.ecm.core.api.DocumentModel
Input: DocumentModelImpl(5eff3564-1a69-4d91-ae0c-51fb879a6a5a, path=/default-domain,
title=Default domain)
Context Variables | Key: ChainParameters, Value: {}


****** Context.RunOperation ******
Chain ID: chainA
```

```
Class: RunOperation
Method: 'run' | Input Type: void | Output Type: void
Input: DocumentModelImpl(5eff3564-1a69-4d91-ae0c-51fb879a6a5a, path=/default-domain,
title=Default domain)
Parameters  | Name: id, Value: chainB | Name: isolate, Value: false
Context Variables | Key: ChainParameters, Value: {}


****** start sub chain ******


****** chain ******
Parent Chain ID: chainA
Name: chainB
Produced output type: DocumentModelImpl


****** Context.FetchDocument ******
Chain ID: chainB
Class: FetchContextDocument
Method:
 'run' | Input Type: interface org.nuxeo.ecm.core.api.DocumentModel |
Output Type: interface org.nuxeo.ecm.core.api.DocumentModel
Input: DocumentModelImpl(5eff3564-1a69-4d91-ae0c-51fb879a6a5a, path=/default-domain,
title=Default domain)
Context Variables | Key: ChainParameters, Value: {}


****** Seam.AddInfoMessage ******
Chain ID: chainB
Class: AddInfoMessage
Method: 'run' | Input Type: void | Output Type: void
Input: DocumentModelImpl(5eff3564-1a69-4d91-ae0c-51fb879a6a5a, path=/default-domain,
title=Default domain)
Parameters  | Name: message, Value: Message INFO
Context Variables | Key: ChainParameters, Value: {}


****** end sub chain ******


****** Seam.AddInfoMessage ******
Chain ID: chainA
Class: AddInfoMessage
Method: 'run' | Input Type: void | Output Type: void
Input: DocumentModelImpl(5eff3564-1a69-4d91-ae0c-51fb879a6a5a, path=/default-domain,
title=Default domain)
```

```
Parameters  | Name: message, Value: test
Context Variables | Key: ChainParameters, Value: {} | Key: Seam.AddInfoMessage, Value:
Message INFO
```

- 'chainA' is executed, produces a document model and runs three operations: Context.FetchDocument, Context.RunOperation and Seam.AddInfoMessage.
- The `Context.RunOperation` runs a second subchain '`chainB`':
  - These chain has parent ID attribute to '`chainA`';
  - It contains two operations: `Context.FetchDocument` and `Seam.AddInfoMessage`;
  - At subchain ending, third `Seam.AddInfoMessage` operation contained into '`chainB`' is executed.

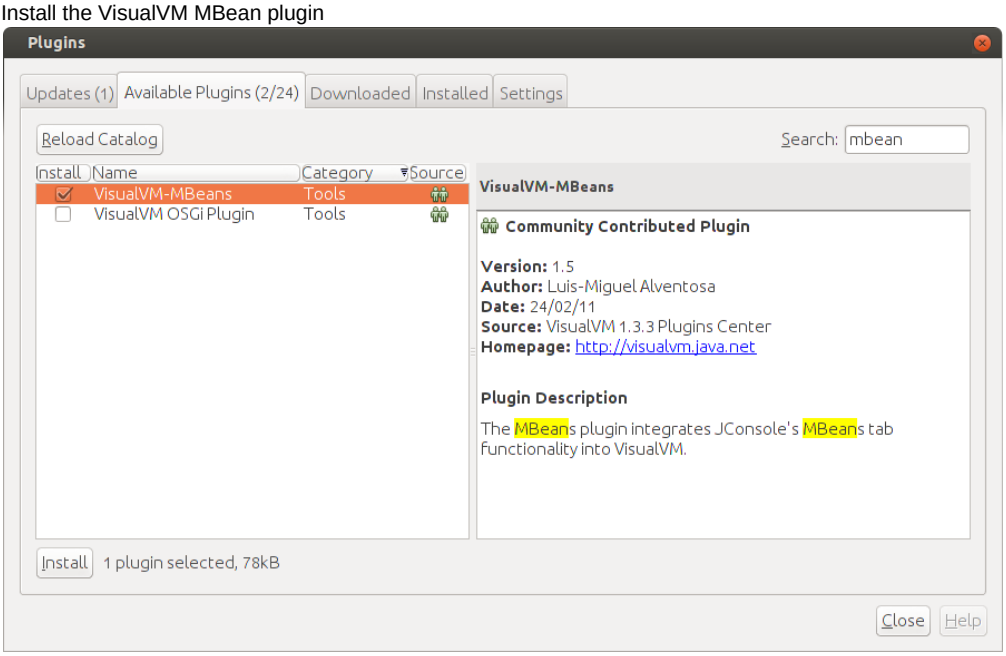
JMX Automation trace activation

The Automation Trace mode can be activated through JMX via `org.nuxeo:TracerFactory` MBean during runtime.

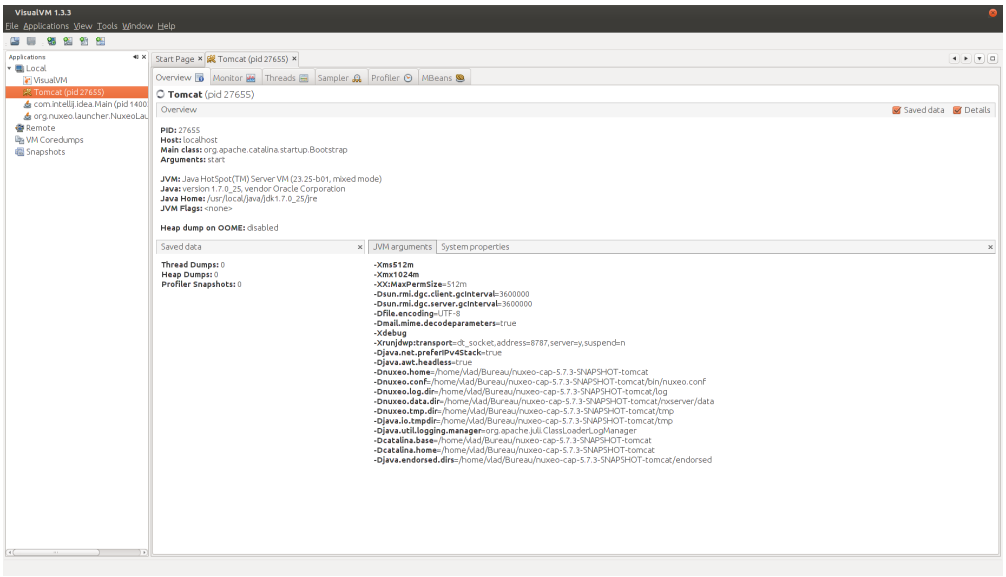
You can also activate the traces (and download them for each chain) from the dynamical documentation of Automation module : http://localhost:8080/nuxeo/site/automation/doc (you should adapt server name and port number).

Please follow guidelines to activate it:

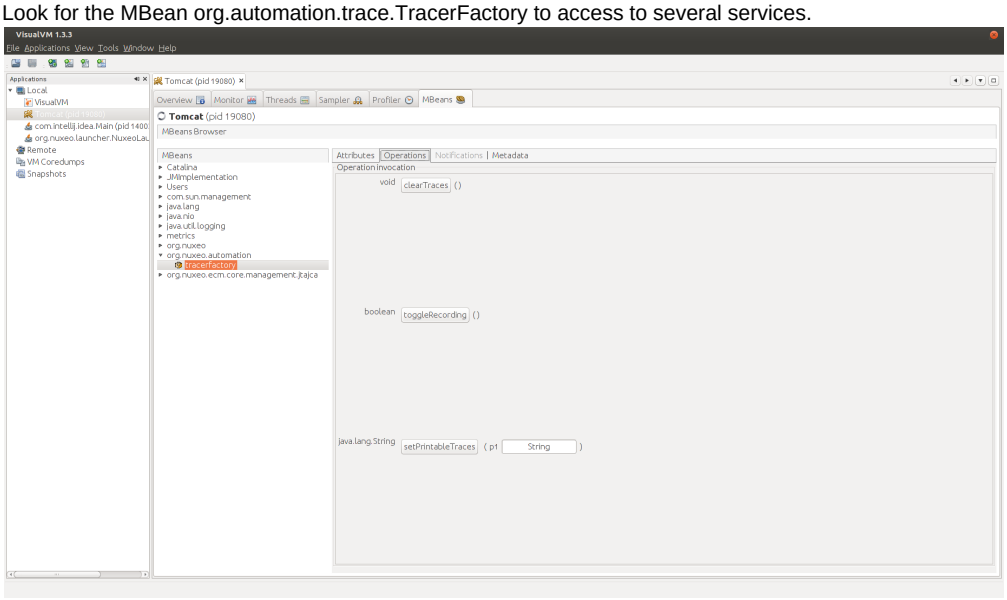
1. Install the VisualVM MBean plugin



2. Run VisualVM and connect it to the Nuxeo server.



3. Look for the MBean org.automation.trace.TracerFactory to access to several services.



- '`clearTraces`' to clean up traces collected during execution.
- '`toggleRecording`' to enable/disable automation trace mode.
- '`setPrintableTraces`' to filter on several chains to print.

Traces are collected and cached for one hour.

Use of MVEL in Automation Chains

This documentation focuses on the MVEL expression language, used in automation chains. For a broader look on that subject, have a look at the Understand Expression and Scripting Languages Used in Nuxeo page.

How to Use Scripting Features in Parameters Values

Values you put in operations parameters are provided with scripting capability.

When filling in a parameter value (like "value" parameter of "update property" in the case we want to update the title of the a document), there are three options:

- `Title`: there is no interpretation (default behavior), the parameter is taken "as is", which means that the title will be "Title".
- `expr:Title` starts with a "`expr:`". The system will interpret the value "Title" as a scripting expression. In that case it means that the value of the title will be updated with the content of the variable *Title*
- `expr:The real @{Title}`: When there is the use of `@{my_variable_name}` in a script expression, the expression between

bracket is considered as a variable name and resolved as a string, and the whole expression is evaluated as the concatenation of the various substrings (included the one interpreted). In our case, if the `Title` variable contains the string literal "Story", the title value will be "The real Story".

When the script is evaluated, you can use both contextual objects and embedded functions.

In this section

Scripting Context

- **Document**: The Document object represents the input document, when the operation takes a document as input. You can use it to get a property value of it:
  `expr:Document["dc:title"]`
  You can also use methods provided by the Document Wrapper, see below.
- **variable_name**: if you set a context variable in a previous operation, you can access it in the parameter value by just referring to its name. In the following sample, if there was a `SetVariable` before in the operation flow that put the path of a document in the variable `path_of_the_workspace`, the parameter's value will be this path.
  `expr:path_of_the_workspace`

  Do not use "-" character in the variable name. Prefer the use of "_".
- **Relative paths**: each time you need to use a path expression (whether it is as a direct parameter of an operation, such as move, or in a an NXQL query, for STARTSWITH operator, you can leverage relative path capability:
  - "." will be replaced by path of input document;
  - ".." will be replaced by path of parent of input document.
- **CurrentDate**: you can use the `CurrentDate` object, that will provide various utility methods to get the current date value, see below.
- **Context.principal.model.getPropertyValue("schema:field")**: gets the current user property. By default users implements user.xsd schema. So if you want for instance the company name, `Context.principal.model.getPropertyValue("user:company")` . But if your users implements other schemas, you choose your "schema prefix (or name if don't set) : field name"
- **CurrentUser.originatingUser**: in a workflow context, all the automation operations executed by the workflow engine are executed using a temporary unrestricted session (if the current user is not an administrator, this is a session with the user "system"). This variable allows you to fetch the current user. This can also be useful when the operation "Users and groups > Login as" has been used in order to retrieve the current username.

  Note that `currentUser` is an alias for `CurrentUser`.

- **Event:** in an event handler context, the Event object can be used to access some of the event's properties. For instance **`@{Event.g`**

etName()} will return the event name.

Document Wrapper

The Document wrapper, used by the system for any document put in scripting context (whether under a variable name, or as the input document ("Document") provides several utility methods:

- `Document.parent`: returns a document wrapper of the parent of the document;
- `Document.workspace`: returns a document wrapper of the parent workspace of the document;
- `Document.domain`: returns a document wrapper of the parent domain of the document;
- `Document.path`: returns a string representing the value of the path of the document, like "/default-domain/workspaces/my-workspace";
- `Document.title`: returns the title of the document;
- `Document.description`: returns the description of the document;
- `Document.type`: returns the Nuxeo EP Document type of the document (like "File", "Folder", ...);
- `Document.lifeCycle`: returns the current life cycle state of the document;
- `Document.name`: returns the name of the document (last part of the path);
- `Document.versionLabel`: returns the version name of the document (like "1.1"...).

Note that `currentDocument` is an alias for `Document`.

Date Wrapper

The Date wrapper is useful to update documents' date properties and to build time-relative NXQL queries.

- `CurrentDate.date`: returns the date. It is the method to use to update a document date field, like "dc:valid", or whatever custom date field.

Some other methods are provided to display the current date as a string:

- `CurrentDate.format("java formatting expression ")`: returns the current date in the specified format;
- `CurrentDate.time`: returns the date in milliseconds;
- `CurrentDate.day`: returns the day of the current time;
- `CurrentDate.month`: returns the month of the current time;
- `CurrentDate.year`: returns the year of the current time;
- `CurrentDate.hour`: returns the hour of the current time;
- `CurrentDate.minut`: returns the minutes of the current time;
- `CurrentDate.second`: returns the seconds of the current time;
- `CurrentDate.week`: returns the week of the current time.

Some others can be used when building an NXQL query to express dates relatively to the current date:

- `CurrentDate.days(-3)`: returns the current date minus three days;
- `CurrentDate.years(10)`: returns the current date plus ten years;
- `CurrentDate.months(-5).weeks(2).seconds(23)`: returns the current date minus five months plus two weeks and 23 seconds;
- ...

• If you want to work on a date that is held by a property of your document, you first need to get a `DateWrapper` object, by using: `@{Fn.calendar(Document["dc:created"])}`.
Ex: `@{Fn.calendar(Document["dc:created"]).format("yyyy-MM-dd")}`

• To set a date property based on the current date, use the `CurrentDate` object, and ends the expression with the `date`

wrapper. So for example, to set up the a field to

...today:

```
@{CurrentDate.date}
```

...in 7 days:

```
@{CurrentDate.days(7).date}
```

• To create a date that you can set on a date property from a string, you can use...

```
@{new java.text.SimpleDateFormat("yyyy-MM-dd").parse(date_str)}
```

...where `date_str` is a Context variable containing a value such as "2013-09-26": You must pass to `SimpleDateFormat` the format of this date, so the parse() method will work.

Functions

The Functions object is providing a set of useful functions. This object is named *Fn* and provide the following functions:

- `Fn.getNextId(String key)` : gets a unique value for the given key. Each time this function is called using the same key a different string will be returned.
- `Fn.getVocabularyLabel(String vocabularyName, String key)` : gets a value from the named vocabulary that is associated with the given key.
- `Fn.getPrincipal(String userName)` : gets a Nuxeo principal object for the given username string.
- `Fn.getEmail(String userName)` : gets the e-mail of the given username.
- `Fn.getEmails(List<String> userNames)` : gets a list of e-mails for the given user name list.
- `Fn.getPrincipalEmails(List<NuxeoPrincipal> principals)` : the same as above but the input object is a list of Nuxeo principals.

Nuxeo Environment Properties

Nuxeo environment properties are accessible in scripts using the `Env` map object. All the properties defined in Nuxeo property files located in Nuxeo `conf` directory are available through the `Env` map. This is very useful when you want to parametrize your operations using values that can be modified later on a running server.

For example let's say you want to make an operation that is creating a document and initialize its description from a Nuxeo property named a `utomation.document.description`.

In order to do this you should:

1. Fetch the property using the `Env` map in your operation parameter: `Env["automation.document.description"]`.
2. And then on the target server to create a property file inside the Nuxeo `conf` directory that defines the variable you are using in the operation chain:
   automation.document.description = My Description

MVEL

The scripting language used is MVEL. See the MVEL language guide. You can use all the features of the scripting language itself.

For instance, you can use the substring method, when dealing with paths:
`expr: Document.path.substring(26)`.

Testing if a variable is null:

```
@{WorkflowVariables["mail"] == empty?"VoidChain":"MyChain"}
```

Usage of `empty` variable allows user to evaluate expression to empty string. For instance to set a property of a document to "" (empty string), you can define the value with `empty:`

```
@{empty}
```

Date Management Example

Operation	Destination field	Source value	Expression Example
Services > Create Task	due date	Current Date	`@{CurrentDate.date}`
		Next Month	`@{CurrentDate.month(1).date}`
		Given Date	2012-03-15T00:00:00Z
		Date Property from the input Document	`@{Document.getProperty("dc:expired")}`
		Date Property from the input Document	`@{Document.getProperty("dc:expired")}`
Document > Update property	value	Current Date	`@{CurrentDate.date}`
		Next Month	`@{CurrentDate.month(1).date}`
Document > Update property	value	Current Date	`@{CurrentDate.date}`

If you have an error like that in your logs:

```
No type adapter found for input: class
org.nuxeo.ecm.automation.core.scripting.DateWrapper and output class java.util.Date
```

This means that you are presenting a DateWrapper value type into a field that waits for a`java.util.Date` object. |

User and Group Management Example

Operation	Destination field	Source value	Expression Example	Remark
Document > Update property	value	String	Administrator	
Document > Update property	value	Static value	Current User	
Services > Create task	additional list of actors prefixed ids	String	`user:Administrator`	
		String	`group:administrators`	
	variable name for actors prefixed ids	In Context	variable	This variable can be set by the "User & Group > Get Users and Groups" with "prefix identifiers" checked.
		Current User	`user:@{CurrentUser.name}`	
User & Group > Get Users and Groups				

Numbers Management example

Operation	Destination field	Source value	Expression Example	Remark
Conversion > Resize a picture	maxHeight/maxWidth	integer	10	

Document management example

Field management

Operation	Destination field	Source value	Expression Example	Remark
Document > Copy	target	String	`/default-domain/workspaces`	Here is specified the path of the container where the document is to be created. The path is given by the names of ancestors of the document (can be different from the title and stored into a different field of the document).
Document > Copy	target	from context	`@{Context"documentStored".name}`	This will work if you have previously added the **Execution Context > Set Context Variable** into your automation chain to store the document into this `documentStored` variable.
	name	String	name-of-my-document	This field is a technical one and used to create the notion of path (see above). **This is not the title**. This is also used for URL generation. Look at the URL after navigating to a document and you will see his path based on the names of the document's ancestor and its name.
Document > Create	type	String	File	Here you must use the name of the document type (not the label).

More Advanced Scripts

In this section, we will gather useful scripts so as to share experience on using scripting in automation, especially in the Run Script operation.

Working with a list of properties

```
new ArrayList(Arrays.asList(WorkflowVariables["contributors"]));
x.add(Context["workflowInitiator"]); WorkflowVariables["contributors"]=x;
// This works for workflow variables, (NodeVariables and WorkflowVariables) but
would also work for a list property of a document, see this question on answers
(http://answers.nuxeo.com/questions/4240/add-string-to-a-list-string-property-in-pur
e-studioautomation-for-prototypes)
```

Related Documentation

- Understand Expression and Scripting Languages Used in Nuxeo
- Automation
- Automation in Nuxeo Studio

Customizing the web app

This chapter presents the different ways to customize what is displayed on the application.

- Seam JSF Webapp Overview — The Nuxeo Platform provides a web framework to build business applications for thin clients. This framework is based on the standard JEE view technology: Java Server Faces (JSF).
- Layouts and Widgets (Forms, Listings, Grids) — Layouts and widgets have been originally designed to display forms, and have been improved to handle listings as well as summary pages. These configuration elements are used to generate page fragments from XML configurations contributed to Nuxeo Runtime extension points.
- Content Views — A content view is a notion to define all the elements needed to get a list of items and perform their rendering. The most obvious use case is the listing of a folderish document content, where we would like to be able to perform several actions.
- Documents Display Configuration — The views on documents, the forms to create or edit them, how lists of documents are presented, all that can be changed in a Nuxeo application, to make sure the information displayed are meaningful. To enable the customization of how documents, forms and listings are presented, Nuxeo Platform-based application use layouts and content views.
- Searches Customization — This chapter presents the different search screens available on the application, and how to customize them.
- Actions (Links, Buttons, Icons, Tabs and More) — Actions usually stand for commands that can be triggered via user interface interaction (buttons, links, etc...).
- Document List Management — Management of a lost of documents is useful for clipboard and generally document selection features.
- Navigation URLs — There are two services that help building GET URLs to restore a Nuxeo context. The default configuration handle restoring the current document, the view, current tab and current sub tab.
- Theme — The theme is in charge of the global layout or structure of a page (composition of the header, footer, left menu, main area...), as well as its branding or styling using CSS. It also handles additional resources like JavaScript files.
- JSF and Ajax Tips and How-Tos

Seam JSF Webapp Overview

Work is still in progress!

The Nuxeo Platform provides a web framework to build business applications for thin clients. This framework is based on the standard JEE view technology: Java Server Faces (JSF).

Nuxeo JSF Technical Stack

Nuxeo JSF framework integrates several technologies in order to make the development of web applications fast and efficient.

The Nuxeo JSF stack includes:

- JSF 1.2 (SUN RI) as MVC and UI component model,
- Facelets as rendering engine and templating system,
- Ajax4JSF to add support for Ajax behaviors,
- RichFaces (3.3) for high level UI components,
- Seam (2.1) as Web Framework

In this section

Inside the Nuxeo Platform, Seam Framework is used only for the JSF (client) layer.

The usage of Seam has several benefits:

- usage of JSF is simpler,
- powerful context management,
- dependency injection and Nuxeo Service lookup via injection,
- Nuxeo Web Component are easily overridable,
- decoupling of Web Components (that can communicate via Seam event bus).

The Nuxeo JSF framework also comes with additional concepts and tools:

- Action service is used to make buttons, tabs and views configurable.
- Layout and Content View allow to define how you want to see documents and listings.
- URL Service: the Nuxeo Platform provides REST URLs for all pages so that you can bookmark pages or send via email a link to a specific view on a specific document.

- Nuxeo Tag Libraries: extend existing tags and provides new Document Oriented tags.
- Theme engine.

Nuxeo JSF Approach

We built Nuxeo JSF framework with two main ideas in mind:

- Make the UI simple,
- Make the UI pluggable.

For the first point, we choose to have an "File Explorer" like navigation. So you have tools (tree, breadcrumb, search, tags) to navigate in a document repository and when on a document you can see several views on this document (Summary, Relations, Workflows, Rights, History ...).

We also choose to make the UI very pluggable, because each project needs to have a slightly different UI. In order to achieve that, each page/view is in fact made of several fragments that are assembled based on the context. This means you can easily add, remove or change a button, a link, a tab or a HTML/JSF block. You don't need to change or override the Nuxeo Platform code for that, neither do you need to change the default Nuxeo Platform templates. The assembly of the fragments is governed by "Actions", so you can change the filters and conditions for each fragment. Of course each project also needs to define it's own views on Document, for that we use the Layout and Content View system.

All this means that you can start from a standard Nuxeo Platform, and with simple configuration have a custom UI.

Nuxeo JSF Key Concepts

Seam JSF Webapp Limitations

This chapter presents the limitations to the Seam/JSF web application.

- Back and Next Buttons Paradigm and JSF in the Nuxeo Platform — Although this library is not designed to take advantage of the back and next buttons of the browser, these buttons work in most cases when called on GET actions, but some inconsistent display could happen if used after a user action modifying data. However, those cache-related display inconsistency aren't harmful in anyway for the system.
- I Get an Error When I Click on Two Links Quickly — Sometimes if you click on a link and then, after a very short time, click on another link without waiting the response from the server, you can get a JSF error. This can happen very frequently when using Ajax requests as users does not always see that the server did not answer yet, so it can be easier from them to trigger an additional request.

Back and Next Buttons Paradigm and JSF in the Nuxeo Platform

Nuxeo Platform navigation is based solely on the JSF library.
Although this library is not designed to take advantage of the back and next buttons of the browser, these buttons work in most cases when called on GET actions, but some inconsistent display could happen if used after a user action modifying data. However, those cache-related display inconsistency aren't harmful in anyway for the system.

Those unwanted displays are hard to fix: it could be done by pushing "by hand" some history info into a queue whenever the Nuxeo Platform does a navigation, and try to return to that when an application-based back button is pressed. But this would be quite complex and browser dependent.

So if you're massively using POST action, the solution is to train the users to never activate/use the Back and the Next buttons when using Nuxeo.

I Get an Error When I Click on Two Links Quickly

This has been fixed in Nuxeo Platform 5.8 (see NXP-12487).

Sometimes if you click on a link and then, after a very short time, click on another link without waiting the response from the server, you can get a JSF error. This can happen very frequently when using Ajax requests as users does not always see that the server did not answer yet, so it can be easier from them to trigger an additional request.

There is no ideal solution to this problem, so it is important to understand why it is happening before picking a solution.

Seam does not handle concurrent requests in the same conversation simultaneously: when a request arrives, if a preceding request is already being processed, it'll wait for the other request to finish before processing the new one. It does not wait indefinitely: it waits for a given, configurable, amount of time. When this timeout is reached, as the first request is still being processed, the corresponding conversation is locked. So the new request is processed in a "temporary" conversation. This temporary conversation does not hold the contextual information that the request may need, and this would lead to an error.

By default, this timeout (`concurrentRequestTimeout`) is configured to 1 s, so if your server is quite slow you can set up a higher timeout to avoid this issue. Note that it is important not to setup a value too high: concurrent requests may keep piling up on the server, and may slow it down even more.

To configure a concurrent request timeout, in the `OSGI-INF/deployment-fragment.xml` of your own project add the definition:

```
<extension target="components#SEAM_CORE_MANAGER" mode="replace">
  <!-- 30 min = 1800 s = 1800000 ms -->
  <property name="conversationTimeout">1800000</property>
  <!-- 5 s = 5000 ms (default value is 1 s) -->
  <property name="concurrentRequestTimeout">5000</property>
</extension>
```

At startup, you'll get in `nxserver/nuxeo.war/WEB-INF/components.xml` a block like:

```
<component name="org.jboss.seam.core.manager">
  <property name="conversationTimeout">1800000</property>
  <property name="concurrentRequestTimeout">5000</property>
</component>
```

If this is not enough, or if this problem is happening on some specific pages, the other options you have are:

- try to optimize the request that takes time: if some request is very slow, users may be tempted to click elsewhere waiting for the server to answer.
- use Ajax configurations to handle concurrent requests: ajax makes it possible to put requests in a queue so that similar requests are not sent to the server. It also allows to wait for a given amount of time before sending a request, etc...
- use JavaScript tricks to make it impossible for the user to click somewhere else (or even to keep on clicking on the costly link) while the server has not answered.

Related topics

- Double Click Shield

Layouts and Widgets (Forms, Listings, Grids)

Layouts and widgets have been originally designed to display forms, and have been improved to handle listings as well as summary pages. These configuration elements are used to generate page fragments from XML configurations contributed to Nuxeo Runtime extension points.

In this chapter we will see how to define layouts and associated widgets, and use them in XHTML pages.

Layouts and Widgets can be configured using Studio: Check out the Form Layouts documentation.

Online demo
You might want to check out the Layout Showcase for a demo.

Key Concepts

Layouts

In a document oriented perspective, layouts are mostly used to display a document metadata in different use cases: present a form to set its schema fields when creating or editing the document, and present these fields values when simply displaying the document. A single layout definition can be used to address these use cases as it will be rendered for a given document and in a given mode.

A layout is a group of widgets that specifies how widgets are assembled and displayed. It manages widget rows and has global control on the rendering of each of its widgets.

Widgets

> *There's a widget in the closet.*

> – R. Kellise

A widget defines how one or several fields from a schema will be presented on a page. It can be displayed in several modes and holds additional information like for instance the field label. When it takes user entries, it can perform conversion and validation like usual JSF components.

Widgets have been made more generic to handle rendering of elements in listings, for instance, or to render information that is not directly linked to a document metadata, like presenting its relations.

Widget Types

A widget definition includes the mention of its type. Widget types make the association between a widget definition and the JSF component tree or xhtml template that will be used to render it in a given mode.

Table of Content

The following pages explain how to work with layouts and widgets.

Layout and Widget Definitions

Custom layouts and widgets can be contributed to the web layout service, using its extension points.

These layout definitions are then available through the service to control how they will be displayed in a given mode.

This chapter explains how to contribute new layouts and widgets and gives information about modes management:

Layout Definitions

Layouts can be used to display various kinds of information, in various renderings.

Layout Registration

Layouts are registered using a regular extension point on the Nuxeo ECM layout service. Here is a sample contribution.

```
<?xml version="1.0"?>

<component name="org.nuxeo.ecm.platform.forms.layouts.webapp">

  <extension target="org.nuxeo.ecm.platform.forms.layout.WebLayoutManager"
    point="layouts">

    <layout name="heading">
      <templates>
        <template mode="any">/layouts/layout_default_template.xhtml</template>
      </templates>
      <rows>
        <row>
          <widget>title</widget>
        </row>
        <row>
          <widget>description</widget>
        </row>
      </rows>
      <widget name="title" type="text">
        <labels>
          <label mode="any">label.dublincore.title</label>
        </labels>
        <translated>true</translated>
        <fields>
          <field>dc:title</field>
        </fields>
        <properties widgetMode="edit">
          <property name="required">true</property>
        </properties>
      </widget>
      <widget name="description" type="textarea">
        <labels>
          <label mode="any">label.dublincore.description</label>
        </labels>
        <translated>true</translated>
        <fields>
          <field>dc:description</field>
        </fields>
      </widget>
    </layout>

  </extension>

</component>
```

In this section

Layout Definition

The above layout definition is used to display the title and the description of a document. Here are its properties:

- `name`: the string used as an identifier. In the example, the layout name is `heading`.
- `templates`: the list of templates to use for this layout global rendering. In the example, the layout template in any mode is the XHTML file at "`/layouts/layout_default_template.xhtml`". Please refer to section about custom layout templates for more information.
- `rows`: the definition about which widgets will have to be displayed on this row. Each row can hold several widgets, and an empty `widget` tag can be used to control the alignment. The widget has to match a widget name given in this layout definition. If none is found, it will lookup a global widget with this name in the global registry of widgets.
  In the example, two rows have been defined: the first one will hold the `title` widget, and the second one will hold the `description` widget.
- `widget`: a layout definition can hold any number of widget definitions. If the widget is not referenced in the rows definition, it will be ignored. Referencing a global widget instead of defining it locally is a convenient way to share widget definitions between layouts. Please refer the Widget Definitions section.

Listing Layout Definition

Layouts can also be used to render table rows, as long as their mode (or their widgets mode) do not depend on the iteration variable (as the layout is built when building the JSF tree, too early in the JSF construction mechanism for most iteration variables).

For this usage, columns/column aliases have been defined because they are more intuitive when describing a row in the layout. The layout `layout_listing_template.xhtml` makes it possible to define new properties to take care of when rendering the table header or columns.

```
<?xml version="1.0"?>

<component name="org.nuxeo.ecm.platform.forms.layouts.webapp.listing">

  <extension target="org.nuxeo.ecm.platform.forms.layout.WebLayoutManager"
    point="widgets">

    <widget name="listing_selection_box_with_current_document"
      type="listing_selection_box_with_current_document">
      <labels>
        <label mode="any"></label>
      </labels>
      <fields>
        <field>selected</field>
        <field>data.ref</field>
      </fields>
    </widget>

    <widget name="listing_icon_type" type="listing_icon_type">
      <labels>
        <label mode="any"></label>
      </labels>
      <fields>
        <field>data</field>
        <field>data.ref</field>
        <field>data.type</field>
        <field>data.folder</field>
      </fields>
    </widget>

    <widget name="listing_title_link" type="listing_title_link">
      <labels>
        <label mode="any">label.content.header.title</label>
      </labels>
      <translated>true</translated>
      <fields>
        <field>data</field>
        <field>data.ref</field>
```

```
      <field>data.dc.description</field>
      <field>data.file.content</field>
      <field>data.file.filename</field>
    </fields>
    <properties mode="any">
      <property name="file_property_name">file:content</property>
      <property name="file_schema">file</property>
    </properties>
  </widget>

  <widget name="listing_modification_date" type="datetime">
    <labels>
      <label mode="any">label.content.header.modified</label>
    </labels>
    <translated>true</translated>
    <fields>
      <field>data.dc.modified</field>
    </fields>
    <properties widgetMode="any">
      <property name="pattern">#{nxu:basicDateAndTimeFormater()}</property>
    </properties>
  </widget>

</extension>

<extension target="org.nuxeo.ecm.platform.forms.layout.WebLayoutManager"
  point="layouts">

  <layout name="document_listing_sample">
    <templates>
      <template mode="any">/layouts/layout_listing_template.xhtml</template>
    </templates>
    <properties mode="any">
      <property name="showListingHeader">true</property>
      <property name="showRowEvenOddClass">true</property>
    </properties>
    <columns>
      <column>
        <properties mode="any">
          <property name="isListingSelectionBoxWithCurrentDocument">
            true
          </property>
          <property name="useFirstWidgetLabelAsColumnHeader">false</property>
          <property name="columnStyleClass">iconColumn</property>
        </properties>
        <widget>listing_selection_box_with_current_document</widget>
      </column>
      <column>
        <properties mode="any">
          <property name="useFirstWidgetLabelAsColumnHeader">false</property>
          <property name="columnStyleClass">iconColumn</property>
        </properties>
        <widget>listing_icon_type</widget>
      </column>
      <column>
        <properties mode="any">
          <property name="useFirstWidgetLabelAsColumnHeader">true</property>
          <property name="sortPropertyName">dc:title</property>
        </properties>
```

```
        <widget>listing_title_link</widget>
      </column>
      <column>
        <properties mode="any">
          <property name="useFirstWidgetLabelAsColumnHeader">true</property>
          <property name="sortPropertyName">dc:modified</property>
        </properties>
        <widget>listing_modification_date</widget>
      </column>
    </columns>
  </layout>

</extension>
```

```
</component>
```

Here widgets have been defined globally, as well as their types. New widget types, or simply widget templates, can be made taking example on the existing ones, see the layouts-listing-contrib.xml and chapter about Listing Widget Types.

Note that field bindings are a bit different here since they do not hold just the property path, but the `data` prefix. For more information about this, please read Field Binding and Expressions.

More information about how to write a listing layout template can be read in section Custom Layout and Widget Templates. If you need to define listing layouts that handle column selection, please refer to the Advanced Search chapter as it gives a complete example on how this is achieved for this feature.

Grid Layout Definition

Layouts can also be used to render grid layouts, visible on documents "Summary" tab for instance.

Here is a sample contribution showing how to define grid slots and corresponding size. The online Style Guide shows detailed information about grids styling.

```
<layout name="grid_summary_layout">
  <templates>
    <template mode="any">
      /layouts/layout_grid_template.xhtml
    </template>
  </templates>
  <rows>
    <row>
      <properties mode="any">
        <property name="nxl_gridStyleClass_0">gridStyle12</property>
      </properties>
      <widget>summary_panel_top</widget>
    </row>
    <row>
      <properties mode="any">
        <property name="nxl_gridStyleClass_0">gridStyle7</property>
        <property name="nxl_gridStyleClass_1">gridStyle5</property>
      </properties>
      <widget>summary_panel_left</widget>
      <widget>summary_panel_right</widget>
    </row>
    <row>
      <properties mode="any">
        <property name="nxl_gridStyleClass_0">gridStyle12</property>
      </properties>
      <widget>summary_panel_bottom</widget>
    </row>
  </rows>
</layout>
```

Other Kinds of Layout Definitions

Above examples show that depending on the layout template used to perform the rendering, specific configuration can be held by the layout definition to allow performing customized behavior and rendering.

As an example, Studio generates specific layout templates to handle `colspans` in generated rendering. Colspan information is kept on layout rows properties.

Related pages in current documentation

- Layout and Widget Modes
- Generic Layout Usage

Related pages in Studio documentation

- Form Layouts in Nuxeo Studio
- Tabs in Nuxeo Studio

Widget Definitions

This page describes the different elements that can be used to define widgets.

> *Last night a widget saved my life*
>
> *– ui:Indeep*

Sample Definition

Here is a sample contribution of widget definitions inside a layout definition:

In this section

```
<?xml version="1.0"?>

<component name="org.nuxeo.ecm.platform.forms.layouts.webapp">

  <extension target="org.nuxeo.ecm.platform.forms.layout.WebLayoutManager"
    point="layouts">

    <layout name="heading">
      <templates>
        <template mode="any">/layouts/layout_default_template.xhtml</template>
      </templates>
      <rows>
        <row>
          <widget>title</widget>
        </row>
        <row>
          <widget>description</widget>
        </row>
      </rows>
      <widget name="title" type="text">
        <labels>
          <label mode="any">label.dublincore.title</label>
        </labels>
        <translated>true</translated>
        <fields>
          <field>dc:title</field>
        </fields>
        <properties widgetMode="edit">
          <property name="required">true</property>
        </properties>
      </widget>
      <widget name="description" type="textarea">
        <labels>
          <label mode="any">label.dublincore.description</label>
        </labels>
        <translated>true</translated>
        <fields>
          <field>dc:description</field>
        </fields>
      </widget>
    </layout>

  </extension>

</component>
```

Two widget definitions are presented on the above example. Let's look into the `title` widget and present its properties:

- `name`: the string used as an identifier in the layout context.
  In the example, the widget name is `title`.
- `type`: the widget type that will manage the rendering of this widget.
  In this example, the widget type is `text`. This widget type is a standard widget types, more information about widget types is available on the Standard Widget Types page.
- `labels`: the list of labels to use for this widget in a given mode. If no label is defined in a specific mode, the label defined in the "any" mode will be taken as default.
  In the example, a single label is defined for any mode to the `label.dublicore.title` message. If no label is defined at all, a default label will be used following the convention: `label.widget.[layoutName].[widgetName]`.
- `translated`: the string representing a boolean value ("true" or "false") and defaulting to "false". When set as translated, the widget labels will be treated as messages and displayed translated. In the example, the `label.dublincore.title` message will be

translated at rendering time.

- `fields`: the list of fields that will be managed by this widget. In the example, we handle the field `dc:title` where "dc" is the prefix for the "dublincore" schema. If the schema you would like to use does not have a prefix, use the schema name instead. Note that most of standard widget types only handle one field. Since 5.8, the field definition also accepts an EL expression. For more information about this please read Field Binding and Expressions.

  When dealing with an attribute from the document that is not a metadata, you can use the property name as it will be resolved like a value expression of the form `#{document.attribute}`, like `#{document.id}` for instance.

- `properties`: the list of properties that will apply to the widget in a given mode. Properties listed in the "any" mode will be merged with properties for the specific mode. Depending on the widget type, these properties can be used to control what JSF component will be used and/or what attributes will be set on these components. In standard widget types, only one component is used given the mode, and properties will be set as attributes on the component. For instance, when using the `text` widget type, every property accepted by the `<h:inputText />` tag can be set as properties on "edit" and "create" modes, and every property accepted by the `<h:outputText />` tag can be set as properties. Properties can also be added in a given widget mode.

Additional Configuration

Additional configuration can be set on a widget:

- `helpLabels`: a list that follows the same pattern as labels, but used to set help labels.
- `widgetModes`: the list of local mode mappings that override the default one. Local mode is deduced from the layout mode or from the parent widget mode.
- `subWidgets`: the list of widget definitions, as the widget list, used to describe subwidgets use to help the configuration of some complex widget types.
- `controls`: an additional configuration to control the behavior of a widget (see the Widget Controls section below).

Here is a more complex layout contribution that shows the syntax to use for these additional properties:

```
<?xml version="1.0"?>

<component name="org.nuxeo.ecm.platform.forms.layouts.webapp">

  <!-- WARNING: this extension point is only available from versions 5.1.7 and 5.2.0
-->
  <extension target="org.nuxeo.ecm.platform.forms.layout.WebLayoutManager"
    point="widgets">

    <!-- global definition of a widget so that it can be used
      in several layouts -->
    <widget name="description" type="textarea">
      <labels>
        <label mode="any">description</label>
      </labels>
      <translated>true</translated>
      <fields>
        <field>dc:description</field>
      </fields>
      <properties widgetMode="edit">
        <property name="styleClass">dataInputText</property>
      </properties>
    </widget>

  </extension>

  <extension target="org.nuxeo.ecm.platform.forms.layout.WebLayoutManager"
    point="layouts">

    <layout name="complex">
      <templates>
        <template mode="any">/layouts/layout_default_template.xhtml</template>
      </templates>
      <rows>
```

```
        <row>
          <widget>identifier</widget>
        </row>
        <row>
          <!-- reference a global widget -->
          <widget>description</widget>
        </row>
      </rows>
      <widget name="identifier" type="text">
        <labels>
          <label mode="any">label.dublincore.title</label>
        </labels>
        <translated>true</translated>
        <fields>
          <field>uid:uid</field>
        </fields>
        <widgetModes>
          <!-- not shown in create mode -->
          <mode value="create">hidden</mode>
        </widgetModes>
        <properties widgetMode="edit">
          <!-- required in widget mode edit -->
          <property name="required">true</property>
        </properties>
        <properties mode="view">
          <!-- property applying in view mode -->
          <property name="styleClass">cssClass</property>
        </properties>
      </widget>
    </layout>

</extension>
```

```
</component>
```

Here is a sample contribution showing subwidgets configuration:

```
<widget name="dam_text_search" type="container">
  <handlingLabels>true</handlingLabels>
  <labels>
    <label mode="any">label.dam.search.textSearch</label>
  </labels>
  <translated>true</translated>
  <properties widgetMode="any">
    <property name="hideSubLabels">true</property>
  </properties>
  <subWidgets>
    <widget name="ecm_fulltext" type="text">
      <labels>
        <label mode="any"></label>
      </labels>
      <translated>false</translated>
      <fields>
        <field>dams:ecm_fulltext</field>
      </fields>
      <properties widgetMode="edit">
        <property name="placeholder">
          #{messages['label.dam.search.text.placeholder']}
        </property>
      </properties>
    </widget>
  </subWidgets>
</widget>
```

Since 5.6, you can also reference global widgets for subwidgets. Here is a sample configuration:

```
<widget name="dam_text_search" type="container">
  <handlingLabels>true</handlingLabels>
  <labels>
    <label mode="any">label.dam.search.textSearch</label>
  </labels>
  <translated>true</translated>
  <properties widgetMode="any">
    <property name="hideSubLabels">true</property>
  </properties>
  <subWidgetRefs>
    <widget>globalSubWidgetName</widget>
  </subWidgetRefs>
</widget>
```

If you need a richer structure to handle subwidgets, you can also consider using a "layout" widget type to render a layout via a widget.

Widget Controls

Since 5.8, widgets definitions also accept a notion of "control".

Controls can be seen as additional properties that can be set on the widget definition, and are managed by mode.

They hold additional configurations that will not be filled on the underlying JSF component attributes, but will more likely be used by this widget parent layout or widget.

Default layout templates and widget types accepting subwidgets handle the following controls:

- `requireSurroundingForm`: this makes it possible to surround the rendered widget by a form. If the control `useAjaxForm` is also set to true, then the form will be ajaxified.
- `handlingLabels`: this makes it possible to avoid making the parent widget or layout handle the label, maybe keeping an unneeded empty space visible. Widgets using a type that knows how to handle their label will then display the label themselves. Widgets using a type that does not know how to handle the label will not display the label at all.

Here is a sample configuration:

```
<widget name="widgetWithControls" type="test">
  <controls mode="any">
    <control name="requireSurroundingForm">true</control>
    <control name="useAjaxForm">true</control>
    <control name="handlingLabels">true</control>
  </controls>
</widget>
```

Related pages in current documentation

- Standard Widget Types
- Custom Widget Types

Layout and Widget Modes

Both layouts and widgets have modes, that makes it possible to render the same layout in different use cases, even if some only support a simple "view" mode.
The **layout modes** can be anything although some default modes are included in the application: `create`, `edit`, `view`. Some additional modes are available: `listing`, `search`, `bulkEdit`, `header`, `csv`, `pdf` and `plain`.

The **widget modes** are more restricted and widget types will usually only handle two modes: `edit` and `view`. Some additional modes are available by default: `pdf`, `csv` and `plain` (very close to the `view` mode except it's not supposed to include HTML tags). These additional modes are useful when exporting listings. The widget mode is computed from the layout mode (or from its parent widget mode).

Here is a table of the default mappings:

Layout Mode	Default Widget Mode
create*, edit*, search*, bulkEdit*	edit
view*, summary*	view
csv*	csv
pdf*	pdf
any other value	'view' before 5.4.2, 'plain' after

It is possible to override this behavior in the widget definition and state that, for instance, whatever the layout mode, the widget will be in `view` mode. This enables to make the widget display only read-only values. The pseudo-mode `hidden` can also be used in a widget definition to exclude this widget from the layout in a given mode.

The pseudo mode `any` is only used in layouts and widgets definitions to set up default values.

Related pages in current documentation

- Layout Definitions
- Widget Definitions

Related pages in Studio documentation

- Form Layouts

Field Binding and Expressions

This chapter explains how field bindings are resolved, what is their purpose, and what variables are available for expressions depending on the context.

Field Bindings

The final binding used by the JSF component is built by the layout system at runtime so that it applies to the value you give it in the template defining the `nxl:layout` tag.

Simple Use Case

You can use this kind of `nxl:layout` tag in a XHTML template.

In this section


```
<nxl:layout name="myLayoutName" mode="view" value="#{currentDocument}" />
```

The layout contains a widget mapped to the `dc:title field` (see the Widget Definitions page):

```
<widget name="title" type="text">
 <labels>
   <label mode="any">label.dublincore.title</label>
 </labels>
 <translated>true</translated>
 <fields>
   <field>dc:title</field>
 </fields>
 <properties widgetMode="edit">
   <property name="required">true</property>
 </properties>
</widget>
```

When the layout service builds the corresponding JSF view (dynamically via facelets), it will build the value expression `#{currentDocument.dc.title}` (or a similar expression achieving this) and give it to the JSF component that would be usually used for a `h:outputText` JSF tag.

Using the field property means that you expect the layout system to perform this mapping.

Listing Use Case

Depending on the value given to the `nxl:layout` tag, the field definition may change. For instance, in a listing, the layout is not usually rendered on the document, but on a `PageSelection` element, wrapping the `DocumentModel` to handle selection information. So a `text` widget showing the widget `title` would have the field binding `data['dc']['title']` instead of `dc:title`.

```
<widget name="title" type="text">
  <labels>
    <label mode="any">Title</label>
  </labels>
  <translated>false</translated>
  <fields>
    <field>data['dc']['title']</field>
  </fields>
</widget>
```

Using the syntax `data.dc.title` is also an option as it will produce a valid EL expression. But the `data['dc']['title']` syntax makes it possible to correctly handle cases where the schema or field name contains a dash "-" character. The "data" binding is mentioned here because it will resolve the `PageSelection#getData` method, which resolves to the `DocumentModel`. Subsequent "dc" and "title" configurations make it resolve the document field named "title" in the "dublincore" schema (defined to use prefix "dc").

Overriding the Layout System Mapping

If you would like to map your widget to another object than the one the layout is applying to, you can use an EL expression in a custom property named `value` instead. It will override the field mapping generated by the layout system.

Since 5.8, you can even do so in the field definition (which makes it possible to define EL expressions for several fields). This can also be useful to display values resolved dynamically, like providing the difference between two integers kept on your document metadata, for instance `#{layoutValue.data.mySchema.myFirstNumber - layoutValue.data.mySchema.mySecondNumber}`. Also, remember that if you are displaying such expression while the widget is not in a listing, you cannot use `data`. The expression will be: `#{layoutValue.mySchema.myFirstNumber - layoutValue.mySchema.mySecondNumber}`.

EL Expressions in Layouts and Widgets

Variables in Definitions

Some variables are available within layout and widget definitions, and can be referenced in properties, modes and field bindings definitions.

- `layoutValue`: represents the value (evaluated) passed in a `nxl:layout` or `nxl:documentLayout` tag attribute.
- `layoutMode`: represents the mode (evaluated) passed in a `nxl:layout` or `nxl:documentLayout` tag attribute.

Variables in Templates

Some variables are made available to the EL context when using layout or widget templates.

- Inside the **layout context**, the following global variables are available:
  - `layoutValue`: represents the value (evaluated) passed in a `nxl:layout` or `nxl:documentLayout` tag attribute.
  - `layoutMode`: represents the mode (evaluated) passed in a `nxl:layout` or `nxl:documentLayout` tag attribute.
  - `value`: represents the current value as manipulated by the tag. In both `nxl:layout` and `nxl:widget` tags, it will represent the value resolved from the `value` tag attribute. This value will work with field information passed in the widget definition to resolve fields and subfields. The variable `document` is available as an alias of `value`, although it does not always represent a document model (as layouts can apply to any kind of object).
  - `value_n`: represents the current value as manipulated by the tag, as above, excepts it includes the widget level (`value_0`, `value_1`, etc...). This is useful when needing to use the value as defined in a parent widget, for instance.
- Inside a **layout template**, the variable "layout" is available. It make it possible to access the generated layout object. Since Nuxeo 5.8, the layout properties are also exposed for easier resolution of EL expressions: for instance, the variable `layoutProperty_title` represents the resolved property with name `title`.
- Inside a **`nxl:layoutRow` tag**, the variables `layoutRow` and `layoutRowIndex` are available to access the generated layout row, and its index within the iteration over rows. The equivalent `layoutColumn` and `layoutColumnIndex` variables are also available for the **`nxl:layoutColumn` tag**.
- Inside a **`nxl:layoutRowWidget`** , or equivalent `nxl:layoutColumnWidget` tag, the variables `widget` and `widgetIndex` are available to access the generated current widget, and its index in the row or column. The variables with level information are also available: `widget_0`, `widget_1`, etc. and `widgetIndex_0`, `widgetIndex_1`, etc. This is useful when needing to use the widget

as defined in a higher level.

- Inside a **widget template**, some `field_n` variables are available: `field` or `field_0` represents the resolved first field value, `field_1` the second value, etc.
  The widget properties are also exposed for easier resolution of EL expressions: for instance, the variable `widgetProperty_onchange` represents the resolved property with name `onchange`.
  The variable `fieldOrValue` is also available, in case the widget should be bound to the layout value (or parent widget value) when field definitions are empty.
  Since Nuxeo 5.8, the variable `widgetProperties` is also available, exposing the complete map of resolved widget properties.

The complete reference is available at http://community.nuxeo.com/api/nuxeo/release-5.8/tlddoc/nxl/tld-summary.html.

There are some inconsistencies between variables exposed in layout/widget definitions and variables exposed in layout/widget templates. You can follow issue NXP-10423 for improvements about this.

Related pages in current documentation

- How to Add a New Widget to the Default Summary Layout
- Standard Widget Types
- Custom Layout and Widget Templates

Standard Widget Types

A series of widget types has been defined for the most generic uses cases.

Please refer to the tag library documentation available at http://community.nuxeo.com/api/nuxeo/5.8/tlddoc/ for Nuxeo JSF tags, and consider reading the page Default Widget Types Known Limitations.

All widgets types available on your Nuxeo application are visible at http://localhost:8080/nuxeo/site/layout-manager/widget-types.

Here is some documentation about the most common widget types that are used in the application:

- Basic Widget Types — A series of widget types is available for the most basic uses cases.
- Listing Widget Types — A series of widget types useful for listings.
- Summary Widget Types — A series of higher-level widget types useful to display information on a document Summary tab.
- Tab Designer Widget Types — Some higher level widget types are useful to design tab content, and come as an addition to Summary Widget Types.
- Decoration Widget Types — A series of widget types that are only useful to handle display of subwidgets, or just add tags surrounding other widgets.
- Advanced Widget Types — A series of widget types for advanced usage.
- Default Widget Types Known Limitations — Some widgets have limitations in some specific conditions of use. We maintain a list of known problems here.
- Suggestion Widget Types — A series of widget types for suggestions.

Basic Widget Types

A series of widget types is available for the most basic uses cases.

text

The `text` widget displays an input text in create or edit mode, with an additional message tag for errors, and a regular text output in any other mode. Widgets using this type can provide properties accepted on a `<h:inputText />` tag in create or edit mode, and properties accepted on a `<h:outputText />` tag in other modes.

View online demo: http://layout.demo.nuxeo.org/nuxeo/layoutDemo/textWidget.

int

The `int` widget displays an input text in create or edit mode, with an additional message tag for errors, and a regular text output in any other mode. It uses a number converter. Widgets using this type can provide properties accepted on a `<h:inputText />` tag in create or edit mode, and properties accepted on a `<h:outputText />` tag in other modes.

View online demo: http://layout.demo.nuxeo.org/nuxeo/layoutDemo/intWidget.

secret

The `secret` widget displays an input secret text in create or edit mode, with an additional message tag for errors, and nothing in any other mode. Widgets using this type can provide properties accepted on a `<h:inputSecret />` tag in create or edit mode.

View online demo: http://layout.demo.nuxeo.org/nuxeo/layoutDemo/secretWidget.

In this section

textarea

The `textarea` widget displays a textarea in create or edit mode, with an additional message tag for errors, and a regular text output in any other mode. Widgets using this type can provide properties accepted on a `<h:inputTextarea />` tag in create or edit mode, and properties accepted on a `<h:outputText />` tag in other modes.

View online demo: http://layout.demo.nuxeo.org/nuxeo/layoutDemo/textareaWidget.

datetime

The `datetime` widget displays a JavaScript calendar in create or edit mode, with an additional message tag for errors, and a regular text output in any other mode. It uses a date time converter. Widgets using this type can provide properties accepted on a `<nxu:inputDatetime />` tag in create or edit mode, and properties accepted on a `<h:outputText />` tag in other modes. The converter will also be given these properties.

View online demo: http://layout.demo.nuxeo.org/nuxeo/layoutDemo/datetimeWidget.

file

The `file` widget displays a file uploader/editor in create or edit mode, with an additional message tag for errors, and a link to the file in other modes. Widgets using this type can provide properties accepted on a `<nxu:inputFile />` tag in create or edit mode, and properties accepted on a `<nxu:outputFile />` tag in other modes.

View online demo: http://layout.demo.nuxeo.org/nuxeo/layoutDemo/fileWidget.

htmltext

The `htmltext` widget displays an HTML text editor in create or edit mode, with an additional message tag for errors, and a regular text output in other modes (without escaping the text). Widgets using this type can provide properties accepted on a `<nxu:editor />` tag in create or edit mode, and properties accepted on a `<nxu:outputText />` tag in other modes.

View online demo: http://layout.demo.nuxeo.org/nuxeo/layoutDemo/htmltextWidget.

selectOneDirectory

The `selectOneDirectory` widget displays a selection of directory entries in create or edit mode, with an additional message tag for errors, and the directory entry label in other modes. Widgets using this type can provide properties accepted on a `<nxd:selectOneListbox />` tag in create or edit mode, and properties accepted on a `<nxd:directoryEntryOutput />` tag in other modes.

View online demo: http://layout.demo.nuxeo.org/nuxeo/layoutDemo/selectOneDirectoryWidget.

selectManyDirectory

The `selectManyDirectory` widget displays a multiselection of directory entries in create or edit mode, with an additional message tag for errors, and the directory entries labels in other modes. Widgets using this type can provide properties accepted on a `<nxd:selectManyListbox />` tag in create or edit mode, and properties accepted on a `<nxd:directoryEntryOutput />` tag in other modes.

View online demo: http://layout.demo.nuxeo.org/nuxeo/layoutDemo/selectManyDirectoryWidget.

checkbox

The `checkbox` widget displays a checkbox in create, edit and any other mode, with an additional message tag for errors. Widgets using this type can provide properties accepted on a `<h:selectBooleanCheckbox />` tag in create, edit mode, and other modes.

View online demo: http://layout.demo.nuxeo.org/nuxeo/layoutDemo/checkboxWidget.

list

The `list` widget displays an editable list of items in create or edit mode, with an additional message tag for errors, and the same list of items in other modes. Items are defined using subwidgets configuration. This actually a template widget type whose template uses a `<nxu:inputList />` tag in edit or create mode, and a table iterating over items in other modes. It also offers alternative renderings.

View online demo: http://layout.demo.nuxeo.org/nuxeo/layoutDemo/listWidget

complex

The `complex` widget displays its subwidgets, binding them to a map-like structure suitable for complex field types definitions. It offers different kinds of renderings and is available since Nuxeo 5.4.2.

View online demo:http://layout.demo.nuxeo.org/nuxeo/layoutDemo/complexWidget

template

The `template` widget displays a template content whatever the mode. Widgets using this type must provide the path to this template; this template can check the mode to adapt the rendering.

Information about how to write a template is given in the custom widget template section.

Related Documentation

- Standard Widget Types
- Custom Widget Types
- Layout and Widget Modes

Related How-tos

- How to Refresh the Task Widget on the Summary Tab
- How to Add a New Widget to the Default Summary Layout

Listing Widget Types

A series of widget types useful for listings.

Listing widget types usually apply to a `PageSelection` element, wrapping the `DocumentModel` to handle selection information.

Basic widget types can also be used in listings, but this has an impact on field bindings configuration.

For instance, when displaying the document title in a listing layout, here is the corresponding configuration:

```
<widget name="title" type="text">
  <labels>
    <label mode="any">Title</label>
  </labels>
  <translated>false</translated>
  <fields>
    <field>data['dc']['title']</field>
  </fields>
</widget>
```

You can see that the "data" binding is mentioned. It will resolve `PageSelection#getData` method, which resolves to the `DocumentModel`. Subsequent "dc" and "title" configurations make it resolve the document field named "title" in the "dublincore" schema (defined to use prefix "dc").

Please refer to the section Field Binding and Expressions for more information about field bindings configuration.

In this section

Title with Link

Id: `listing_title_link`

This widget type displays the document title with a link to it. It can also display a download link provided the blob field path in its properties.

View online demo: http://showcase.nuxeo.com/nuxeo/layoutDemo/listing_title_linkWidget

Author

Id: `listing_author`

This widget type displays the document author/creator first and last name, with its identifier in a tooltip.

View online demo: http://showcase.nuxeo.com/nuxeo/layoutDemo/listing_authorWidget

Last Contributor

Id: `listing_last_contributor`

This widget type displays the document last contributor first and last names, with its identifier in a tooltip.

View online demo: http://showcase.nuxeo.com/nuxeo/layoutDemo/listing_last_contributorWidget

Big Icon with Type

Id: `listing_big_icon_type_link`

This widget type displays a big icon (usually 100x100 px) with the document type in a tooltip. The icon can represent the document type, or the attached file type when there is one.

View online demo: http://showcase.nuxeo.com/nuxeo/layoutDemo/listing_big_icon_type_linkWidget

Icon with Type

Id: `listing_icon_type`

This widget type displays an icon (usually 16x16 px) with the document type in a tooltip. The icon can represent the document type, or the attached file type when there is one.

View online demo: http://showcase.nuxeo.com/nuxeo/layoutDemo/listing_icon_typeWidget

Version

Id: `listing_version`

This widget type displays the document version.

View online demo: http://showcase.nuxeo.com/nuxeo/layoutDemo/listing_versionWidget

Lifecycle

Id: `listing_lifecycle`

This widget type displays the document life cycle state.

View online demo: http://showcase.nuxeo.com/nuxeo/layoutDemo/listing_lifecycleWidget

Related topics in this documentation

- Standard Widget Types
- Custom Widget Types

Summary Widget Types

A series of higher-level widget types useful to display information on a document Summary tab.

> Summary widget types are usually using hardcoded values to display from the context, hence their field bindings are usually ignored.
>
> For this reason, these widget types will not behave correctly if used in listing layouts.

Comments

Id: `summary_current_document_comments`

This widget type displays the current document comments.

Description

Id: `summary_current_document_description`

This widget type displays the current document description, creation date, author and version.

Dublincore

Id: `summary_current_document_dublincore`

This widget type displays the dublincore layout for the current document (in view mode).

In this section

Files

Id: `summary_current_document_files`

This widget type displays attached files to the current document. It also displays a drop zone for Drag & Drop features.

Life Cycle State and Version

Id: `summary_current_document_lc_and_version`

This widget type displays the current document life cycle state and version.

Publications

Id: `summary_current_document_publications`

This widget type displays the current document publications.

Relations

Id: `summary_current_document_relations`

This widget type displays the current document incoming and outgoing relations.

States

Id: `summary_current_document_states`

This widget type displays the current document life cycle state and lock status.

Tagging

Id: `summary_current_document_tagging`

This widget type displays the current document tags, with the possibility to add new ones.

> **Requirements**
> This feature requires the Document Management module to be installed.

Tasks

Id: `summary_current_document_single_tasks`

This widget type displays the current document workflow tasks assigned to current user.

View Layout

Id: `summary_current_document_view`

This widget type displays the current document view layout (as configured on the document type definition).

Workflow Process

Id: `summary_document_route`

This widget type displays a selector with the list of available workflows that can be started on the current document. When a workflow is already started, it displays the workflow name and a button to view its graph.

Related topics in this documentation

- Standard Widget Types
- Custom Widget Types

Tab Designer Widget Types

Some higher level widget types are useful to design tab content, and come as an addition to Summary Widget Types.

Toggleable Layout

Id: `toggleableLayoutWithForms`

This widget type presents a layout in view mode, with toggle buttons to present the edit mode, as well as usual Save and Cancel buttons by default (these can be customized).

Content View

Id: `contentViewWithForms`

This widget type presents a content view, with possibility to override some of the content view definition properties in the widget properties.

In this section

- Toggleable Layout
- Content View
- Tabs
- Form Action
- Form Actions
- Toolbar Action
- Toolbar Actions

Tabs

Id: `documentTabsWithForms`

This widget type presents document tabs. It is tied to the document on which this widget is rendered as filters are evaluated in this context.

Form Action

Id: `documentAction`

This widget type presents an action, not surrounded by a form. It is tied to the document on which this widget is rendered as filters are evaluated in its context. The action to present is retrieved by id.

This is useful in combination with the "layout" widget for instance, or to build incremental layouts using "Summary Panel" categories (as widgets contributed to this category may already contain a form).

Form Actions

Id: `documentActions`

This widget type presents actions, not surrounded by a form. It is tied to the document on which this widget is rendered as filters are evaluated in its context. The actions to present are retrieved by category.

This is useful in combination with the `layout` widget for instance, or to build incremental layouts using "Summary Panel" categories (as widgets contributed to this category may already contain a form).

Toolbar Action

Id: `documentActionWithForms`

This widget type presents an action, surrounded by a form. It is tied to the document on which this widget is rendered as filters are evaluated in its context.

Form management performed by this widget type makes it possible to use FancyBox actions for instance (so that FancyBox content can itself contain a form).

Toolbar Actions

Id: `documentActionsWithForms`

This widget type presents document tabs. It is tied to the document on which this widget is rendered as filters are evaluated in its context.

Form management performed by this widget type makes it possible to use FancyBox actions for instance (so that FancyBox content can itself contain a form).

Related topics in this documentation

- Standard Widget Types
- Custom Widget Types

Related topics in Studio documentation

- Filtering Options Reference Page
- Tabs

Decoration Widget Types

A series of widget types that are only useful to handle display of subwidgets, or just add tags surrounding other widgets.

Container

Id: `container`

The container widget is a "decorative" widget, used to control rendering of its subwidgets. It's available since Nuxeo 5.6 and is used on the default Summary layout.

View online demo: http://layout.demo.nuxeo.org/nuxeo/layoutDemo/containerWidget

In this section

- Container
- Drop Zone

Drop Zone

Id: `dropZone`

This widget defines a drop zone, used to define a zone where some actions will be called on drop. The Drag and Drop Service for Content Capture (HTML5-Based) documentation details how it can be used.

Related pages in this documentation

- Actions Overview
- Custom Action Types
- Drag and Drop Service for Content Capture (HTML5-Based)

Related pages in Studio documentation

- User Actions
- User actions categories

Advanced Widget Types

A series of widget types for advanced usage.

Layout

Id: `layout`

The layout widget type displays a layout. This is useful to build high-level layouts, in combination with the Form Actions widget type for instance.

Set Variable

Id: `setVariable`

This widget type uses the `nxu:set` tag to expose variables in the JSF context, and displays its subwidgets. It can be useful to make a variable available for the subwidgets rendering.

For instance, this is useful when manipulating selection entries, to be able to cache them in the page (and avoid retrieving them for each item rendered in selection options).

In this section

Template

Id: `template`

The template widget type is highly customizable and allows to define a XHTML template to use for rendering the widget depending on its mode. Please refer to the dedicated documentation.

Action

Id: `action`

This widget type displays an action, with possibility to surround it by a form or not (and additional rendering properties).

The difference with the Form Action or Toolbar Action widget types is that it needs the action to be already resolved. This makes it possible to display actions that would have been retrieved using a custom filtering context.

Actions

Id: `actions`

This widget type displays actions, with possibility to surround it by a form or not (and additional rendering properties).

The difference with the Form Actions or Toolbar Actions widget types is that it needs the action to be already resolved. This makes it possible to display actions that would have been retrieved using a custom filtering context.

Single Generic Suggestion

Id: `select2Widget`

This widget types displays a select2 suggestion for a single element selection. It requires advanced configuration for the operation to call and JavaScript functions for formatting of JSON result of suggestions.

Multiple Generic Suggestion

Id: `select2WidgetMultiple`

This widget types displays a select2 suggestion for multiple elements selection. It requires advanced configuration for the operation to call and JavaScript functions for formatting of JSON result of suggestions.

Related Documentation

- Standard Widget Types
- Custom Widget Types
- Layout and Widget Modes

Related How-tos

- How to Refresh the Task Widget on the Summary Tab
- How to Add a New Widget to the Default Summary Layout

Default Widget Types Known Limitations

Some widgets have limitations in some specific conditions of use. We maintain a list of known problems here.

- **Widgets using HTML text editor cannot be used in a list (fixed from 5.8.0-HF26).**
- **File widget cannot be used in an Ajax form.**
- **File widget selection may be lost when adding element in a list**: this issue happens when using a file widget inside a list. When adding a new element in the list, the previously selected file can be lost. There is ad-hoc JavaScript code that help keeping this selection, a sample widget template handling a list of files can taken as an example.
- You might have some troubles using **chain select** in Ajax forms or when using lists. This occurs in some very specific conditions, so you have to check if it is ok for your use case first. We plan to rewrite completely this widget for solving this issue.
- **Widgets using Rich Faces suggestion components doesn't work correctly in a list.** Limitation is at lower Rich Faces level, so there is no short term fix planned.
- Widgets displaying URLs to files/images may misbehave inside of lists. This is because the URL might need information about the complete property XPath, and this variable is not exposed in the context for now (see **NXP-10423** - Improve variables exposed in layouts (Open)). Some specific widget templates can be used to workaround this issue, and some of them are available by default:
  - `/widgets/extended_subfile_widget.xhtml` Displays a link to a file with additional information (icon, PDF export) for a file within a list.
  - `/widgets/image_subwidget_template.xhtml` Displays an image when within a list (available only from 5.9.3, 5.8.0-HF09, 5.6.0-HF32.

  These widget templates are specific to this use case, and will not work as expected inside lists of lists, for instance, but custom widget templates can be defined for this behavior, taking care of the following issues:
  - The URL system may rely on a document to provide the RESTful link, hence the `layoutValue` variable is used for it: these widget templates will assume that the layout applies to a document
  - The parent widgets field definitions, used to build the document property XPath, can be retrieved using variables `widget_0` for instance: this assumes that the widget at the first level of the layout holds the list property path.
  - The widget template needs to know the file/image index, and is relying on variable `model.rowData` for it: this is only available within a widget of type `list`.
  - Facelets tags evaluated at build time (like a c:if tag) cannot check elements within the list, because these items are only exposed at render time.

Other Widget Related Documentation

- Standard Widget Types
- Custom Widget Types

Suggestion Widget Types

A series of widget types for suggestions.
Since 5.7.2, Nuxeo uses select2 for the suggestion widgets:

- the user suggestion and the document suggestion widgets were previously based on the RichFaces suggestion box, and they now use select2;
- a new directory suggestion widget is provided.

Unlike the older JSF suggestion widgets, these select2-based widgets rely upon an automation operation to retrieve suggestions according to the search term typed in the select2 box. The backing operation returns suggestions as an array of JSON objects that select2 is able to manipulate.

As a result, it is quite easy to customize the display of selected/suggested entries.

Also, these suggestion widgets can be used inside lists (older suggestion widgets had limitations).

In this section

The Different Types of Widgets

Each widget type is available for single and multiple suggestions.

Directory Suggestion Widget Type

This widget type lets the user select an entry among a list of vocabulary entries, based on the user input. Widget type names are:

- suggestOneDirectory
- suggestManyDirectory

These new widgets come in addition to the existing selectOneDirectory and selectManyDirectory. The suggestion are provided by the `SuggestDirectoryEntries` operation.

Suggestion widgets can present multi-level chained vocabulary out of the box. A multi-level chained vocabulary is a vocabulary whose entries have a parent entry such as ln10coverage and ln10subjects. These widgets will be able to group entries by their parents only if the parent is also defined in the vocabulary.
About Label Localization

By default, suggestion widgets assume the vocabulary labels are not localized. In case of need of localization, these widgets support two ways of label localization:
Properties File Translation

The translations are available is `message_xx.properties` files and the vocabulary only handles the label key. As of Nuxeo 5.8, labels will be translated if the widget is marked as translated in its configuration (but this should be tied to the "localize" property for better consistency so this will likely change in the future, see NXP-13629):

```
<translated>true</translated>
```

Column-Based Translation

The vocabulary handles as many label columns as supported locales. Again this is the case for ln10coverage and ln10subjects which have two columns for English and French translations. To support such vocabulary structure, the `dbl10n` widget property must be set to true:

```
<property name="dbl10n">true</property>
```

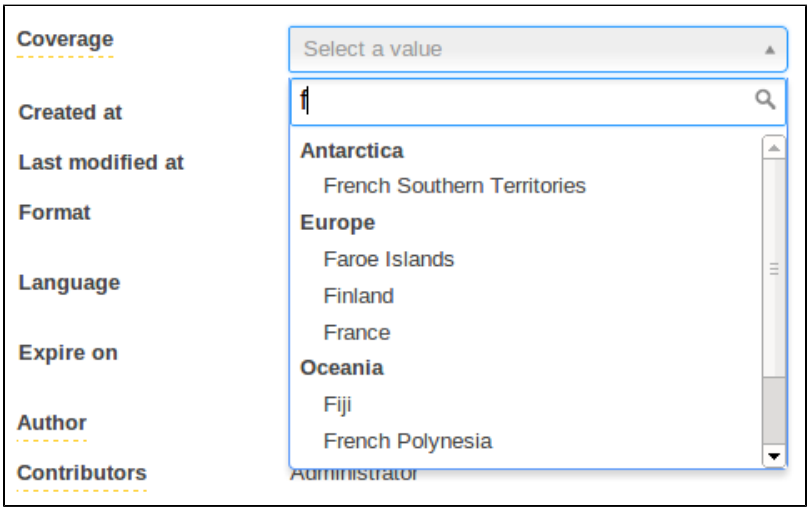
By convention, the field name associated to these label columns must respect the pattern **label_xx** where xx is the locale code. However this can be changed with the `labelFieldName` widget parameter. For instance, the following configuration means that labels are translated using the columns `language_en`, `language_fr`, etc. of your vocabulary.

```
<property name="labelFieldName">language_{lang}</property>
```

Using column-based translation allows to directly query suggestions in the directory associated to the vocabulary. This is much more efficient than filtering on translations handled by properties file in term of performance.

Check out the layout showcase for more details on available widget properties:

- Vocabulary suggestion showcase
- Multiple vocabulary showcase



Picture 1: *`suggestOneDirectory` presenting l10ncoverage vocabulary. Countries are grouped by continent.*

User Suggestion Widget Type

This widget type lets the user select an entry among a list of users or groups based on the user input. Widget type names are:

- singleUserSuggestion
- multipleUsersSuggestion

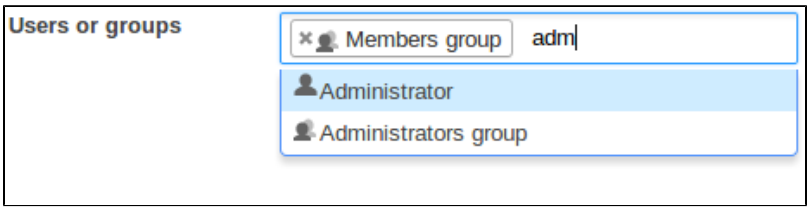
Previously based on RichFaces suggestion box in Nuxeo Platform 5.6 and earlier versions, these widgets now use select2 to present user suggestions returned by the `SuggestUserEntries` operation. All previous widget properties are still available.

However, here are noticeable new properties:
groupRestriction

This property allows to specify a group id to restrict the suggested users.

Check out the layout showcase for more details on available widget properties:

- Single user showcase
- Multiple user showcase



Picture 2: *multipleUsersSuggestion*

Document Suggestion Widget Type

This widget type lets the user select an entry among a list of existing documents of the repository that results from a search query based on the user input. Widget type names are:

- singleDocumentSuggestion
- multipleDocumentsSuggestion

Previously based on RichFaces suggestion box for Nuxeo Platform 5.6 and earlier versions, these suggestion widgets now use select2 to present document suggestions returned by the `DocumentPageProviderOperation` operation.
Modifying the Way of Suggesting Documents

This operation uses the `default_document_suggestion` page provider. However, you can specify the NXQL query to be executed via the widget property named `query`.

Query to suggest documents with title starting with the entered term

```
<property name="query">
  SELECT * FROM  Document WHERE dc:title LIKE ? AND ecm:mixinType !=
'HiddenInNavigation' AND ecm:isCheckedInVersion = 0 AND ecm:currentLifeCycleState !=
'deleted'
</property>
```

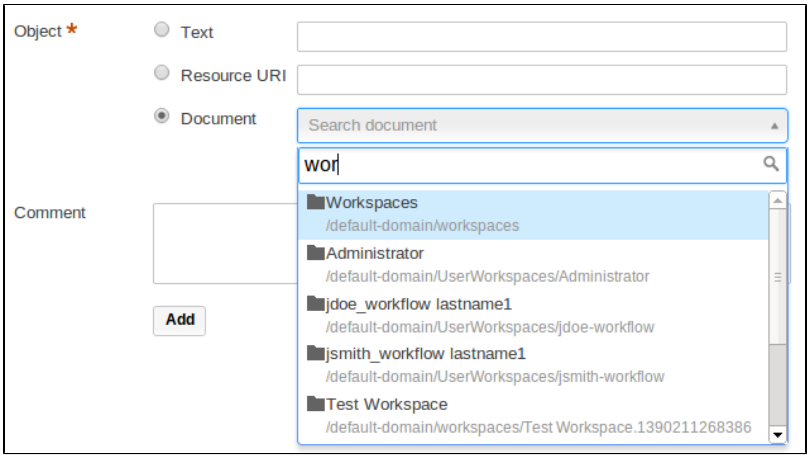
The specified query must have one parameter (the '?' character in the query above). This parameter will be be valued with the term entered in the select2. Note that a '%' character will be automatically appended to the entered value.

Finally, it is also possible to tell these widgets to use another page provider with the widget property `pageProviderName`. See the pages Custom Page Providers and Page Providers without Content Views.
Which Document Properties Are Accessible?

As explained above, the `DocumentPageProviderOperation` will return documents serialized as JSON objects. By default, the properties of the `dublincore` and `common` schemas are serialized. However, you can extend the list of schemas that must be returned in the document JSON serialization with the `documentSchemas` widget property:

```
<property name="documentSchemas">dublincore,common</property>
```

This is useful if you need to customize the display of the suggested/selected entries with specific document properties.



Picture 3: singleDocumentSuggestion

Advanced Suggestion Widget Type

- select2Widget
- select2WidgetMultiple

This section deals with advanced settings and requires some development skills.

In case you'd like to build your own advanced suggestion widget, here are additional widget properties.
operationId

Specify the id of an automation operation that will be used to feed the widget with suggestion.

```
<property name"operationId">your_operation_id</property>
```

The term entered in select2 will be passed to the specified operation under the parameter `searchTerm`. Your operation must therefore at least have this parameter:

```
@Param(name = "searchTerm", required = false)
protected String searchTerm;
```

The specified operation must return an array of JSON objects which, to be interpreted by select2, must contains the following elements:

- `id`: the value or reference to be submitted (i.e. saved in the bound field). In case of a JSON serialization of a document, it is the document uid. However, you can modify which element of the JSON object must be submitted with the `idProperty` widget property.
- `displayLabel`: the label to displayed by select2 in the UI. However, if you use custom formatter, it will be ignored.

Custom Display

How to Format the Displayed Entries

Suggestion widget appearance can be customized in a quite flexible way through the use of JavaScript formatters. You can separately tune the formatting of both the suggested and selected entries. All suggestion widgets have the following widget properties:

- `suggestionFormatter`
- `selectionFormatter`
- `inlinejs`

As explained above, each suggested entries are JSON objects returned by an automation operation. The idea is to define a JavaScript function that will generate the HTML to render this JSON object. For instance, the formatter used to render suggested documents in the above Picture 3 is:

Example of formatter used in Picture 3

```
function myDocFormatter(doc) {
    var markup = "<table><tbody>";
    markup += "<tr><td>";
    if (doc.properties && doc.properties['common:icon']) {
      markup += "<img src='" + window.nxContextPath
          + doc.properties['common:icon'] + "'/>"
    }
    markup += "</td><td>";
    markup += doc.title;
    if (doc.warn_message) {
      markup += "<img src='" + window.nxContextPath
          + "/icons/warning.gif' title='" + doc.warn_message + "'/>"
    }
    if (doc.path) {
      markup += "<span class='displayB detail' style='word-break:break-all;'>" +
doc.path + "</span>";
    }
    markup += "</td></tr></tbody></table>"
    return markup;
  }
```

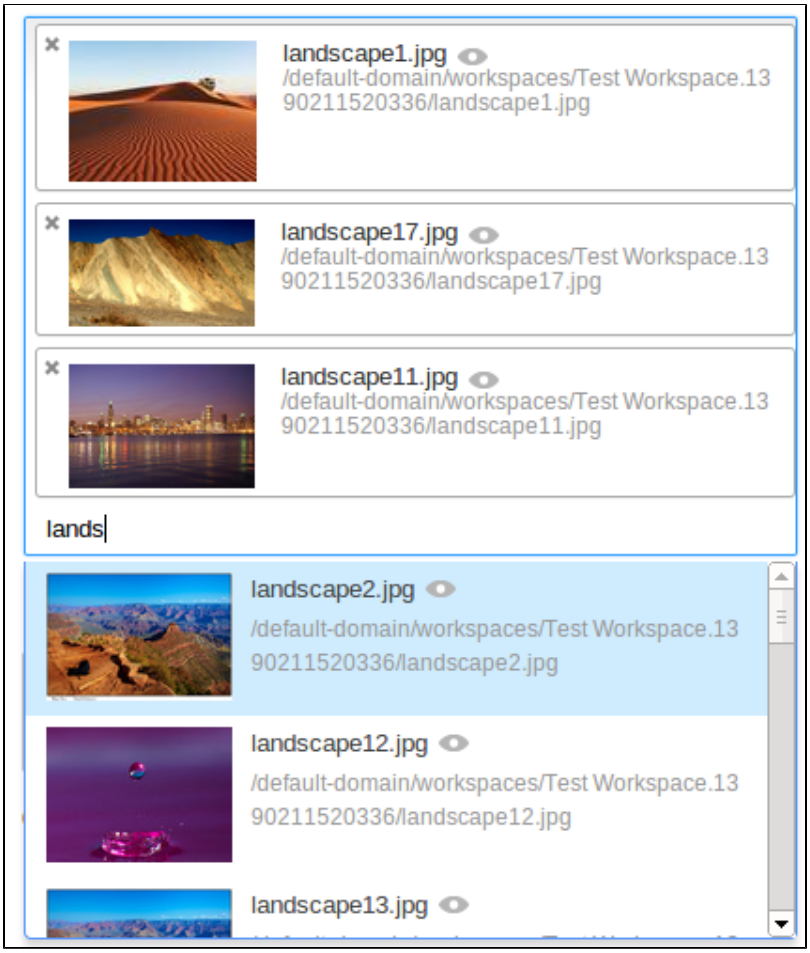
You can define the JS code of your custom formatters directly in the inlinejs widget property, it will be injected in the rendered HTML page. Alternatively, you can define your formatters as global JavaScript resources by contributing them to the Theme.

Once you have defined a custom formatter, you can tell your widgets to use it:

```
<property name="suggestionFormatter">myformatter</property>
<property name="selectionFormatter">myformatter</property>
<property name="inlinejs">
   <![CDATA[
      function myformatter(doc) {
        return doc.title;
      }
   ]]>
</property>
```

The JSON serialization of documents is not flat! Here is how you can access the document properties, for instance, based on the snippet above (line 4):

```
doc.properties['common:icon']
```



Picture 4: multipleDocumentsSuggestion *with custom formatter to suggest documents with "Picture" facet. See this cookbook for more information.*

How to Customize CSS Styling

You can finally customize the CSS style of the select2-based widgets through the use of the following widget properties:

- `containerCssClass`
- `dropdownCssClass`

The container is the div handling the current selection and the dropdown is the popup box suggesting available entries.

These properties are passed through to select2, that handles these properties as standard select2 parameters. Please refer to the select2 documentation for more details about these parameters.

Finally you can specify the `width` property, it accepts pixel as well as percentage (i.e. 300px and 100%).

Internationalization

Select2 widgets localization is provided by the select2 library and the control labels are therefore not translated in `messages_xx.properties` files but directly in select2 resources files. Many locales are provided.

However, if you're a missing a locale, you must create your own `select2_locale_xx.js` file and add it to your own bundle. This file must be deployed in `${NUXEO_HOME}/nxserver/nuxeo.war/scripts/select2/` directory.

Custom Layout and Widget Templates

Some templating features have been made available to make it easier to control the layouts and widgets rendering.

Custom Layout Template

A layout can define an XHTML template to be used in a given mode. Let's take a look at the default template structure.

In this section


```
<f:subview
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:nxl="http://nuxeo.org/nxforms/layout"
  xmlns:nxu="http://nuxeo.org/nxweb/util"
  id="#{layout.id}"
  styleClass="#{layoutProperty_styleClass}">

  <c:set var="isEditMode" value="#{nxl:isBoundToEditMode(layout.mode)}" />
  <nxu:set var="layoutWidgetsDisplay"
    value="#{layout.properties.widgetsDisplay}"
    resolveTwice="true">

    <table class="dataInput">
      <tbody>

        <nxl:layoutRow>
          <nxu:set var="rowStyleClass"
            value="#{layoutRow.properties.styleClass}"
            resolveTwice="true">
            <tr class="#{rowStyleClass}">
              <nxl:layoutRowWidget>

                <c:if test="#{layoutWidgetsDisplay == 'label_top'}">
                  <nxu:set var="fieldColspan"
                    value="#{layout.columns/layoutRow.size}">
                    <td class="fieldColumn" colspan="#{fieldColspan}" dir="auto">
                      <c:if test="#{not widget.handlingLabels}">
                        <div>
                          <ui:include
src="/widgets/incl/widget_label_template.xhtml">
                            <ui:param name="labelStyleClass"
                              value="boldLabel #{widgetProperty_subLabelStyleClass}"
/>
                          </ui:include>
                        </div>
                      </c:if>
                      <ui:decorate template="/widgets/incl/form_template.xhtml">
                        <ui:param name="addForm"
```

```
value="#{widgetControl_requireSurroundingForm}" />
                        <ui:param name="formId" value="#{widget.id}_form" />
                        <ui:param name="useAjaxForm"
value="#{widgetControl_useAjaxForm}" />
                        <ui:define name="form_template_content">
                          <nxl:widget widget="#{widget}" value="#{value}" />
                        </ui:define>
                      </ui:decorate>
                    </td>
                  </nxu:set>
                </c:if>

                <c:if test="#{layoutWidgetsDisplay != 'label_top'}">
                  <c:if test="#{layoutWidgetsDisplay != 'no_label'}">
                    <c:if test="#{not widget.handlingLabels}">
                      <td class="labelColumn">
                        <ui:include src="/widgets/incl/widget_label_template.xhtml"
/>
                      </td>
                    </c:if>
                  </c:if>
                  <nxu:set var="fieldColspan"
                    value="#{2*layout.columns/layoutRow.size -1 +
nxu:test(widget.handlingLabels, 1, 0)}">
                    <td class="fieldColumn" colspan="#{fieldColspan}" dir="auto">
                      <ui:decorate template="/widgets/incl/form_template.xhtml">
                        <ui:param name="addForm"
value="#{widgetControl_requireSurroundingForm}" />
                        <ui:param name="formId" value="#{widget.id}_form" />
                        <ui:param name="useAjaxForm"
value="#{widgetControl_useAjaxForm}" />
                        <ui:define name="form_template_content">
                          <nxl:widget widget="#{widget}" value="#{value}" />
                        </ui:define>
                      </ui:decorate>
                    </td>
                  </nxu:set>
                </c:if>

              </nxl:layoutRowWidget>
            </tr>
          </nxu:set>
        </nxl:layoutRow>

      </tbody>
    </table>

  </nxu:set>

<script>
  jQuery(document).ready(function() {
    jQuery(".widgetHelpLabel").tooltip({relative: true, position: 'bottom center'});
  });
```

```
</script>

</f:subview>
```

This template is intended to be unused in any mode, so the layout mode is checked to provide a different rendering in "edit", "create", "view" modes and other modes.

When this template is included in the page, several variables are made available:

- `layout`: the computed layout value; its mode and number of columns can be checked on it.
- `value` or `document`: the document model (or whatever item used as value).

The layout system integration using facelets features requires that iterations are performed on the layout rows and widgets. The `<nxl:layoutRow>` and `<nxl:layoutRowWidget />` trigger these iterations. Inside the `layoutRow` tag, two more variables are made available: `layoutRow` and `layoutRowIndex`. Inside the `layoutRowWidget`, two more variables are made available: `widget` and `widgetIndex`.

These variables can be used to control the layout rendering. For instance, the default template is the one applying the "required" style on widget labels, and translating these labels if the widget must be translated. It also makes sure widgets on the same rows are presented in the same table row.

This template also shows that:

- Layout templates can handle properties set on the layout definition (here to control the display of its subwidgets via a property named `widgetsDisplay`).
- Layout templates can check controls set on widgets configuration (here to add a form around the widget or not).

Listing Template

This layout intends to render columns within a table: each line will be filled thanks to a layout configuration. It is only used in view mode. Let's take a look at the default listing template structure.

```
<f:subview
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:nxl="http://nuxeo.org/nxforms/layout"
  xmlns:nxu="http://nuxeo.org/nxweb/util"
  xmlns:nxd="http://nuxeo.org/nxweb/document"
  xmlns:a4j="https://ajax4jsf.dev.java.net/ajax"
  id="#{layout.id}">

<c:if test="false">
  Layout template applying to an item instance of
PageSelections&amp;lt;DocumentModel&amp;gt; named "documents"

  Other needed parameters are:
  - provider: instance of a PageProvider&amp;lt;DocumentModel&amp;gt; to handle sort
  - layoutListingStatus: iteration status, used to print table header
    matching widget label.
  - formId: the surrounding form id
</c:if>

<nxu:set var="hasSeveralSorts"
  value="#{provider.getSortInfos().size() > 1}"
  cache="true">

<c:if test="#{showListingHeader and layout.properties.showListingHeader}">
  <thead>
    <tr>
      <nxl:layoutColumn>
        <th dir="auto">
          <c:choose>
```

```
            <c:when test="#{layoutColumn.properties.isListingSelectionBox}">
              <h:selectBooleanCheckbox id="#{layoutColumn.widgets[0].name}_header"
                title="#{messages['tooltip.content.select.all']}"
                value="#{documents.selected}">
                <a4j:support event="onclick"

action="#{documentListingActions.processSelectPage(contentView.name,
contentView.selectionListName, documents.selected)}"
                  onclick="javascript:handleAllCheckBoxes(this.form.name,
this.checked)"
                  reRender="#{formId}_buttons:ajax_selection_buttons" />
              </h:selectBooleanCheckbox>
            </c:when>
            <c:when
test="#{layoutColumn.properties.isListingSelectionBoxWithCurrentDocument}">
              <h:selectBooleanCheckbox id="#{layoutColumn.widgets[0].name}_header"
                title="#{messages['tooltip.content.select.all']}"
                value="#{documents.selected}">
                <a4j:support event="onclick"
                  onclick="javascript:handleAllCheckBoxes(this.form.name,
this.checked)"

action="#{documentListingActions.checkCurrentDocAndProcessSelectPage(contentView.nam
e, contentView.selectionListName, documents.selected, currentDocument.ref)}"
                  reRender="#{formId}_buttons:ajax_selection_buttons" />
              </h:selectBooleanCheckbox>
            </c:when>
            <c:when
test="#{layoutColumn.properties.useFirstWidgetLabelAsColumnHeader}">
              <c:choose>
                <c:when test="#{provider.sortable and !empty
layoutColumn.properties.sortPropertyName}">
                  <nxu:set var="ascIndex"

value="#{provider.getSortInfoIndex(layoutColumn.properties.sortPropertyName, true)}"
                    cache="true">
                  <nxu:set var="descIndex"

value="#{provider.getSortInfoIndex(layoutColumn.properties.sortPropertyName,
false)}"
                    cache="true">
                    <span class="contentViewHeaderSortTooltip">
                      <h:commandLink immediate="true"

action="#{provider.setSortInfo(layoutColumn.properties.sortPropertyName,
nxu:test(ascIndex != -1, false, true), true)}"
                        id="#{layoutColumn.widgets[0].name}_header_sort">
                        <h:outputText value="#{layoutColumn.widgets[0].label}"
                          rendered="#{!layoutColumn.widgets[0].translated}" />
                        <h:outputText
value="#{messages[layoutColumn.widgets[0].label]}"
                          rendered="#{layoutColumn.widgets[0].translated}" />
                      </h:commandLink>
                    </span>
                    <div class="tooltip">
                      #{messages['contentview.setSort.help']}
                    </div>
                    <f:verbatim>&amp;nbsp;</f:verbatim>
                    <c:if test="#{ascIndex != -1}">
```

```
                      <span class="contentViewHeaderSortTooltip">
                        <h:commandLink immediate="true"

action="#{provider.setSortInfo(layoutColumn.properties.sortPropertyName, false,
false)}"
                          id="#{layoutColumn.widgets[0].name}_header_sort_desc">
                          <h:graphicImage value="/icons/sort_selected_up.png" />
                          <c:if test="#{hasSeveralSorts}">
                            #{ascIndex + 1}
                          </c:if>
                        </h:commandLink>
                      </span>
                      <div class="tooltip">
                        #{messages['contentview.addSort.help']}
                      </div>
                    </c:if>
                    <c:if test="#{descIndex != -1}">
                      <span class="contentViewHeaderSortTooltip">
                        <h:commandLink immediate="true"

action="#{provider.setSortInfo(layoutColumn.properties.sortPropertyName, true,
false)}"
                          id="#{layoutColumn.widgets[0].name}_header_sort_asc">
                          <h:graphicImage value="/icons/sort_selected_down.png" />
                          <c:if test="#{hasSeveralSorts}">
                            #{descIndex + 1}
                          </c:if>
                        </h:commandLink>
                      </span>
                      <div class="tooltip">
                        #{messages['contentview.addSort.help']}
                      </div>
                    </c:if>
                    <c:if test="#{ascIndex == -1 and descIndex == -1}">
                      <span class="contentViewHeaderSortTooltip">
                        <h:commandLink immediate="true"

action="#{provider.addSortInfo(layoutColumn.properties.sortPropertyName, true)}"
                          id="#{layoutColumn.widgets[0].name}_header_sort_add">
                          <h:graphicImage value="/icons/sort_up.png" />
                        </h:commandLink>
                      </span>
                      <div class="tooltip">
                        #{messages['contentview.addSort.help']}
                      </div>
                    </c:if>
                  </nxu:set>
                  </nxu:set>
                </c:when>
                <c:otherwise>
                  <h:outputText value="#{layoutColumn.widgets[0].label}"
                    rendered="#{!layoutColumn.widgets[0].translated}" />
                  <h:outputText value="#{messages[layoutColumn.widgets[0].label]}"
                    rendered="#{layoutColumn.widgets[0].translated}" />
                </c:otherwise>
              </c:choose>
            </c:when>
          </c:choose>
          <script>
```

```
            jQuery(document).ready(function() {
              jQuery(".contentViewHeaderSortTooltip").tooltip();
            });
          </script>
        </th>
      </nxl:layoutColumn>
    </tr>
  </thead>
</c:if>

</nxu:set>

<c:set var="trStyleClass" value="#{nxu:test(layoutListingStatus.index%2 ==0,
'dataRowEven', 'dataRowOdd')}" />
<tr class="#{nxu:test(layout.properties.showRowEvenOddClass, trStyleClass, '')}">
  <nxl:layoutColumn>
    <td class="#{layoutColumn.properties.columnStyleClass}" dir="auto">
      <nxl:layoutColumnWidget>
        <nxl:widget widget="#{widget}" value="#{value}" />
        <c:if test="#{layoutColumn.size > 1 and layoutColumn.size > widgetIndex + 1
and widgetIndex > 0}">
          <br />
        </c:if>
      </nxl:layoutColumnWidget>
    </td>
  </nxl:layoutColumn>
</tr>
```

```
</f:subview>
```

As you can see, this layout makes it possible to use the first defined widget in a given column to print a label, and maybe translate it. It also relies on properties defined in the layout or layout column properties to handle selection, column style class, sorting on the provider, etc.

Any custom template can be defined following this example to handle additional properties to display on the final table header and columns.

Custom Summary Template

The summary uses a custom template to use "div" elements instead of tables, more appropriate to a dashboard-like view.

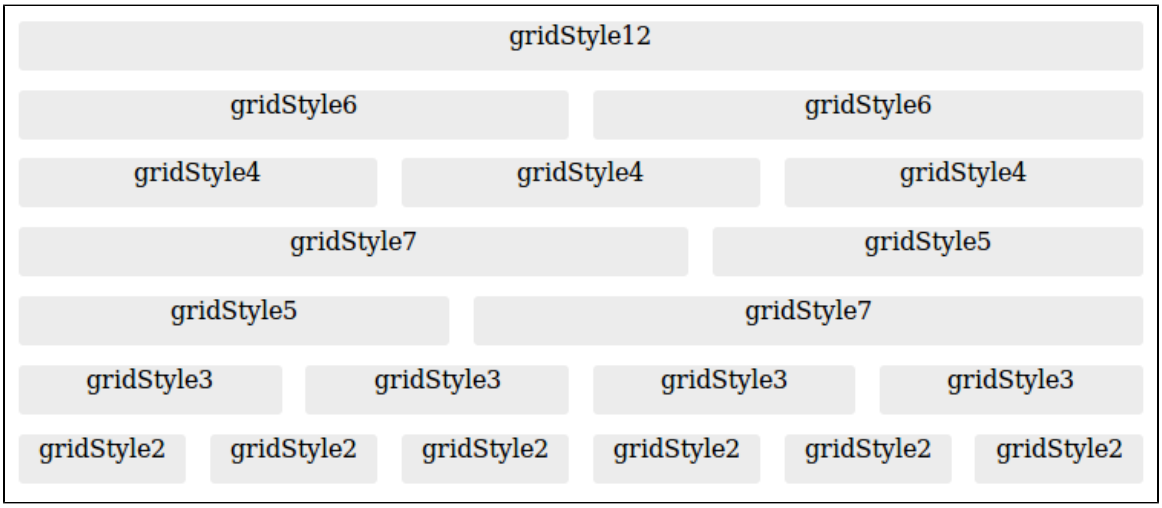
Since version 5.6, it uses a "grid" rendering allowing fine-grained control over the place used by widgets. It combines the following layout template with the use of the standard "container" widget type. The container widgets pile up any number of widgets displaying information about the document metadata, its state, relations, publications, etc.

```
<f:subview
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:nxl="http://nuxeo.org/nxforms/layout"
  id="#{layout.id}">

  <c:if test="false">
    Handles grid layouts, using style classes defined by row properties.
  </c:if>

  <div class="gridContainer">
    <nxl:layoutRow>
      <div class="gridRow">
        <nxl:layoutRowWidget>
          <c:set var="gridStyleClassProp" value="nxl_gridStyleClass_#{widgetIndex}"
/>
          <div class="gridBox #{layoutRow.properties[gridStyleClassProp]}">
            <nxl:widget widget="#{widget}" value="#{value}" />
          </div>
        </nxl:layoutRowWidget>
      </div>
    </nxl:layoutRow>
  </div>

</f:subview>
```

When using this layout template, the layout definition will use properties defined on rows to allow more or less place to the widgets. Here is the default summary definition:

```
<layout name="grid_summary_layout">
  <templates>
    <template mode="any">
      /layouts/layout_grid_template.xhtml
    </template>
  </templates>
  <rows>
    <row>
      <properties mode="any">
        <property name="nxl_gridStyleClass_0">gridStyle7</property>
        <property name="nxl_gridStyleClass_1">gridStyle5</property>
      </properties>
      <widget>summary_left_panel</widget>
      <widget>summary_right_panel</widget>
    </row>
  </rows>
</layout>
```

Here the first widget, containing widgets to display on the left part of the page, will take approximatively 60% of the page. Here is a diagram to help you design layouts using grids:



Custom Widget Template

The template widget type makes it possible to set a template to use as an include.

Let's have a look at a sample template used to present contributors to a document.

```
<f:subview xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:a4j="https://ajax4jsf.dev.java.net/ajax"
  xmlns:nxu="http://nuxeo.org/nxweb/util"
  xmlns:nxdir="http://nuxeo.org/nxdirectory"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:nxp="http://nuxeo.org/nxweb/pdf"
  id="#{widget.id}">
  <div>
    <c:forEach var="username" items="#{field}" varStatus="status">
      <c:if test="#{!status.first}">#{status.last ? andLabel : ', '}</c:if>
      <h:outputText value="#{nxu:userFullName(username)}"
        title="#{username}" onmouseover="tooltip.show(username, 500);"
        onmouseout="tooltip.hide();"/>
    </c:forEach>
  </div>
</f:subview>
```

This widget presents the contributors of a document with specific links on each on these user identifier information, whatever the widget mode.

Having a widget type just to perform this kind of rendering would be overkill, so using a widget with type "template" can be useful here.

Template widgets should handle the new 'plain' and 'pdf' modes for an accurate rendering of the layout in PDF (content view and document export) and CSV (content view export). CSV export does not need any specific CSV rendering, so the widget rendering in 'plain' mode should be enough.

Some helper methods make it easier to check the widget mode, here is the complete current definition of the contributors widget type in the Nuxeo Platform.

```
<f:subview xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:a4j="https://ajax4jsf.dev.java.net/ajax"
  xmlns:nxu="http://nuxeo.org/nxweb/util"
  xmlns:nxdir="http://nuxeo.org/nxdirectory"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:nxp="http://nuxeo.org/nxweb/pdf"
  id="#{widget.id}">
<c:set var="andLabel" value=" #{messages['label.and']} " scope="page" />
<c:if test="#{nxl:isLikePlainMode(widget.mode)}"><nxu:inputList
  value="#{field}" model="contributorsModel"><h:outputText
  value="#{nxu:userFullName(contributorsModel.rowData)}" /><h:outputText
  rendered="#{contributorsModel.rowIndex != contributorsModel.rowCount -1}"
  value="#{nxu:test(contributorsModel.rowIndex == contributorsModel.rowCount -2,
andLabel, ', ')}" /></nxu:inputList></c:if>
<c:if test="#{widget.mode == 'pdf'}">
  <nxp:html>
    <c:forEach var="username" items="#{field}" varStatus="status">
      <c:if test="#{!status.first}">#{status.last ? andLabel : ', '}</c:if>
      <h:outputText value="#{nxu:userFullName(username)}" />
    </c:forEach>
  </nxp:html>
</c:if>
<c:if test="#{nxl:isLikeViewMode(widget.mode)}">
  <div>
    <c:forEach var="username" items="#{field}" varStatus="status">
      <c:if test="#{!status.first}">#{status.last ? andLabel : ', '}</c:if>
      <h:outputText value="#{nxu:userFullName(username)}"
        title="#{username}" onmouseover="tooltip.show(username, 500);"
        onmouseout="tooltip.hide();"/>
    </c:forEach>
  </div>
</c:if>
</f:subview>
```

Note that extra spaces have been removed when rendering in the "plain" mode as these spaces may appear on the final rendering (in CSV columns for instance).

When this template is included in the page, the `widget` variable is made available. For a complete list of available variables, please refer to the EL expressions documentation.

Some rules must be followed when writing XHTML to be included in templates:

- Use the widget id as identifier: the widget id is computed to be unique within the page, so it should be used instead of fixed id attributes so that another widget using the same template will not introduce duplicated ids in the JSF component tree.
- Use the variable with name following the field_n pattern to reference field values. For instance, binding a JSF component value attribute to `#{field_0}` means binding it to the first field definition. The expression `#{field}` is an alias to `#{field_0}`.

Built-in Templates to Handle Complex Properties

List Widget Template

The standard widget type "list" is actually a widget of type "template" using a static template path: `/widgets/list_widget_template.xhtml`. If this default behavior does not suit your needs, you can simply copy this template, make your changes, and use a widget of type "template" with the new template path.

This template assumes that each element of the list will be displayed using subwidgets definitions.

For instance, to handle a list of String elements, you can use the definition:

```
<widget name="contributors" type="list">
  <fields>
    <field>dc:contributors</field>
  </fields>
  <subWidgets>
    <widget name="contributor" type="text">
      <fields>
        <field></field>
      </fields>
    </widget>
  </subWidgets>
</widget>
```

The empty field definition in the subwidget is used to specify that each element of the list is itself the element to display.

Complex Widget Template

A builtin template has been added to handle complex properties. It is available at `/widgets/complex_widget_template.xhtml`. It assumes that each element of the complex property will be displayed using subwidgets definitions.

To handle a complex property (the value is a map with keys 'name' and 'email' for instance, you can use the definition:

```
<widget name="manager" type="template">
  <fields>
    <field>company:manager</field>
  </fields>
  <properties mode="any">
    <property name="template">
      /widgets/complex_widget_template.xhtml
    </property>
  </properties>
  <subWidgets>
    <widget name="name" type="text">
      <fields>
        <field>name</field>
      </fields>
    </widget>
    <widget name="email" type="text">
      <fields>
        <field>email</field>
      </fields>
    </widget>
  </subWidgets>
</widget>
```

Lists of Lists

A builtin template has been added to handle sublists: the original "list" widget is equivalent to a widget of type "template" using the file `/widgets/list_widget_template.xhtml`. To handle the sublist, this template needs to be changed. The file `list_subwidget_template.xhtml` is available for it.

To handle a sublist property, you can use take example on this definition:

```
<widget name="employees" type="list">
  <fields>
    <field>company:employees</field>
  </fields>
  <subWidgets>
    <widget name="employee" type="template">
      <labels>
        <label mode="any"></label>
      </labels>
      <properties mode="any">
        <property name="template">
          /widgets/complex_list_item_widget_template.xhtml
        </property>
      </properties>
      <!-- subwidgets for complex -->
      <subWidgets>
        <widget name="phoneNumbers" type="template">
          <fields>
            <field>phoneNumbers</field>
          </fields>
          <properties mode="any">
            <property name="template">
              /widgets/list_subwidget_template.xhtml
            </property>
          </properties>
          <subWidgets>
            <widget name="phoneNumber" type="text">
              <label mode="any"></label>
              <fields>
                <field></field>
              </fields>
            </widget>
          </subWidgets>
        </widget>
      </subWidgets>
    </widget>
  </subWidgets>
</widget>
```

Related topics

- Incremental Layouts and Actions
- How to Add a New Widget to the Default Summary Layout
- Standard Widget Types
- Custom Widget Types

Custom Widget Types

Custom widget types can be added to the standard list thanks to another extension point on the web layout service.

Usually widget types are template widgets that are declared as widget types to make them easily reusable in different layouts, have a clear widget types library, and make them available in Studio.

Simple Widget Type Registration

Here is a sample widget type registration, based on a widget template:

In this section


```
<component name="org.nuxeo.ecm.platform.forms.layout.MyContribution">

  <extension target="org.nuxeo.ecm.platform.forms.layout.WebLayoutManager"
    point="widgettypes">

    <widgetType name="my_widget_type">
      <handler-class>

org.nuxeo.ecm.platform.forms.layout.facelets.plugins.TemplateWidgetTypeHandler
      </handler-class>
      <property name="template">
        /widgets/my_own_widget_template.xhtml
      </property>
    </widgetType>

  </extension>

</component>
```

Before this contribution, the widgets needing this template were declaring (for instance):

```
<widget name="my_widget" type="template">
  <labels>
    <label mode="any">My label</label>
  </labels>
  <translated>false</translated>
  <fields>
    <field>dc:description</field>
  </fields>
  <properties widgetMode="any">
    <property name="template">/widgets/my_own_widget_template.xhtml</property>
  </properties>
</widget>
```

With this configuration, the following widget definition can now be used:

```
<widget name="my_widget" type="my_widget_type">
  <labels>
    <label mode="any">My label</label>
  </labels>
  <translated>false</translated>
  <fields>
    <field>dc:description</field>
  </fields>
</widget>
```

Complex Widget Type Registration

Here is a more complex sample widget type registration:

```
<?xml version="1.0"?>

<component name="org.nuxeo.ecm.platform.forms.layout.MyContribution">

  <extension target="org.nuxeo.ecm.platform.forms.layout.WebLayoutManager"
    point="widgettypes">

    <widgetType name="customtype">
      <handler-class>
        org.myproject.MyCustomWidgetTypeHandler
      </handler-class>
      <property name="foo">bar</property>
    </widgetType>

  </extension>

</component>
```

The custom widget type class must follow the `org.nuxeo.ecm.platform.forms.layout.facelets.WidgetTypeHandler` interface.

Additional properties can be added to the type registration so that the same class can be reused with a different behavior given the property value.

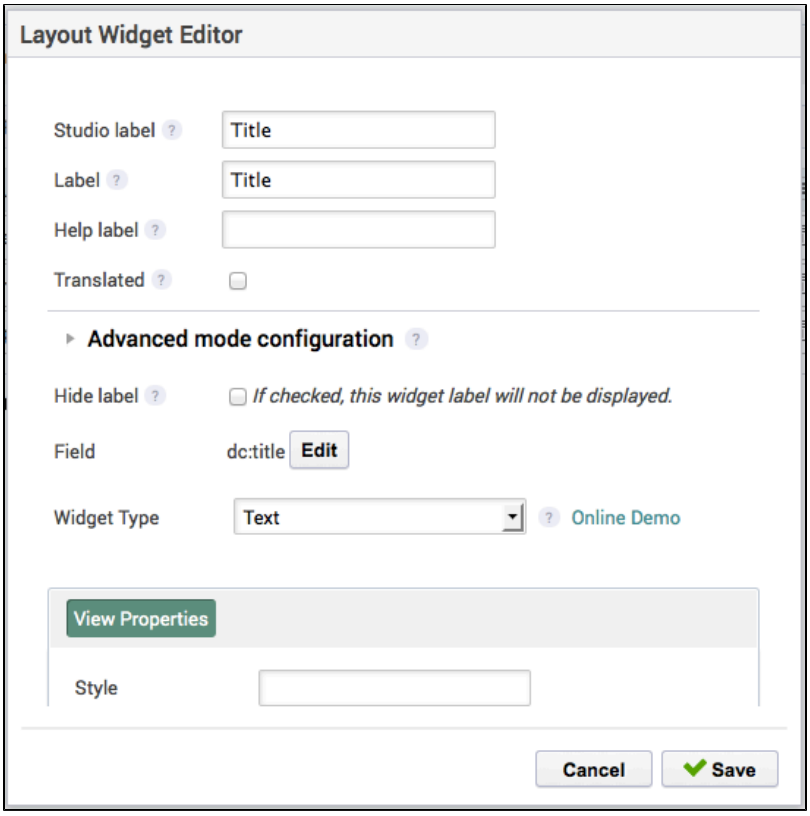
The widget type handler is used to generate facelet tag handlers dynamically taking into account the mode and any other properties that can be found on a widget.

The best thing to do before writing a custom widget type handler is to go see how standard widget type handlers are implemented, as some helper methods can be reused to ease implementation of specific behaviors.

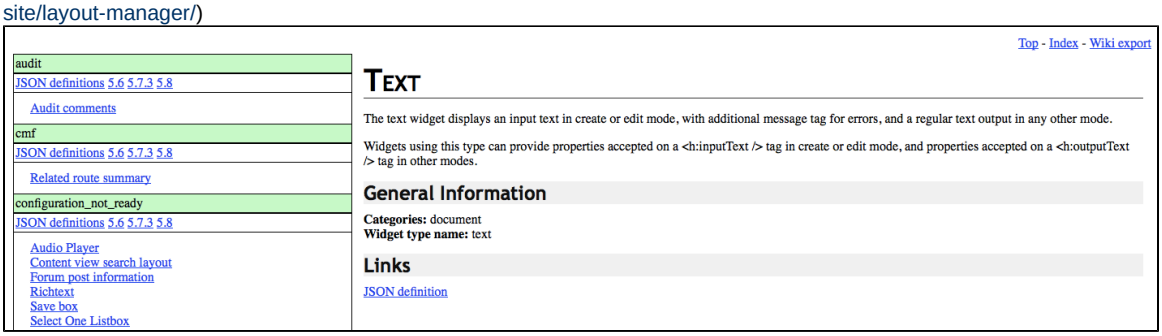
Additional Widget Type Configuration

Some additional information can be put on a widget type for several purposes:

- configuration of widgets made available in Studio



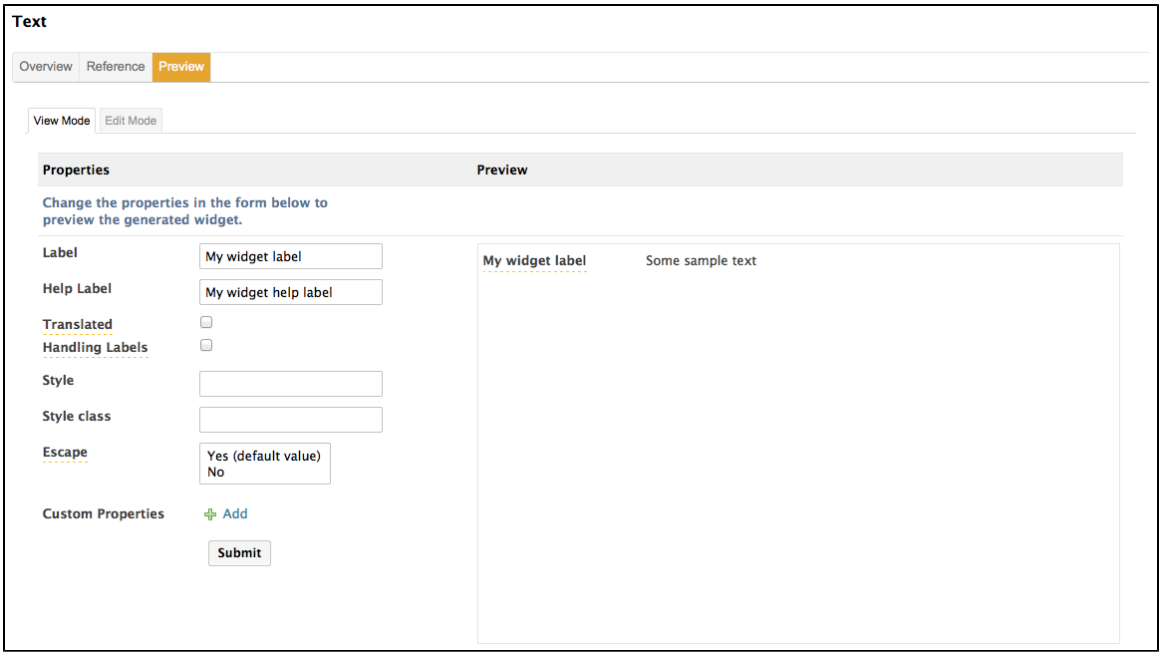
- documentation of available layouts and widget types on a given Nuxeo instance (see on your Nuxeo instance: http://localhost:nuxeo/site/layout-manager/)



- Sample JSON export URLs:

http://demo.nuxeo.com/nuxeo/site/layout-manager/widget-types/widgetTypes/document	All widget types with category "document"
http://demo.nuxeo.com/nuxeo/site/layout-manager/widget-types/widgetTypes/document?version=5.4.0	All widget types with category "document", filtering widget types with a version strictly higher than 5.4.0
http://demo.nuxeo.com/nuxeo/site/layout-manager/widget-types/widgetTypes?categories=studio%20document&version=5.4.0	All widget types with both categories "document" and "studio", filtering widget types with a version strictly higher than 5.4.0.

- documentation and showcase of this widget type (see http://showcase.nuxeo.com/layout):



Here is a sample configuration extract:

```
<widgetType name="text">
  <configuration>
    <title>Text</title>
    <description>
      <p>
        The text widget displays an input text in create or edit mode, with
        additional message tag for errors, and a regular text output in any
        other mode.
      </p>
      <p>
        Widgets using this type can provide properties accepted on a
        &lt;h:inputText /&gt; tag in create or edit mode, and properties
        accepted on a &lt;h:outputText /&gt; tag in other modes.
      </p>
    </description>
    <demo id="textWidget" previewEnabled="true" />
    <supportedModes>
      <mode>edit</mode>
      <mode>view</mode>
    </supportedModes>
    <fields>
      <list>false</list>
      <complex>false</complex>
      <supportedTypes>
        <type>string</type>
        <type>path</type>
      </supportedTypes>
      <defaultTypes>
        <type>string</type>
      </defaultTypes>
    </fields>
    <categories>
      <category>document</category>
    </categories>
    <properties>
```

```
    <layouts mode="view">
      <layout name="text_widget_type_properties_view">
        <rows>
          <row>
            <widget>style</widget>
          </row>
          <row>
            <widget>styleClass</widget>
          </row>
          [...]
        </rows>
        <widget name="style" type="text">
          <labels>
            <label mode="any">Style</label>
          </labels>
          <fields>
            <field>style</field>
          </fields>
        </widget>
        <widget name="styleClass" type="text">
          <labels>
            <label mode="any">Style class</label>
          </labels>
          <fields>
            <field>styleClass</field>
          </fields>
        </widget>
        [...]
      </layout>
    </layouts>
    <layouts mode="edit">
      <layout name="text_widget_type_properties_edit">
        <rows>
          <row>
            <widget>required</widget>
          </row>
          <row>
            <widget>maxlength</widget>
          </row>
          <row>
            <widget>title</widget>
          </row>
          [...]
        </rows>
        <widget name="maxlength" type="int">
          <labels>
            <label mode="any">Max length</label>
          </labels>
          <fields>
            <field>maxlength</field>
          </fields>
        </widget>
        <widget name="required" type="checkbox">
          <labels>
            <label mode="any">Required</label>
          </labels>
          <fields>
            <field>required</field>
          </fields>
```

```
        </widget>
        <widget name="title" type="text">
          <labels>
            <label mode="any">Title</label>
          </labels>
          <fields>
            <field>title</field>
          </fields>
          <widgetModes>
            <mode value="any">hidden</mode>
            <mode value="view_reference">view</mode>
          </widgetModes>
        </widget>
        [...]
      </layout>
    </layouts>
  </properties>
</configuration>
<handler-class>
  org.nuxeo.ecm.platform.forms.layout.facelets.plugins.TextWidgetTypeHandler
```

```
  </handler-class>
</widgetType>
```

The `configuration` element is optional, but when defined it'll be used to define the following information:

- `title`: the widget type title.
- `description`: the widget type description, that accepts HTML content.
- `demo`: this refers to this widget type representation in the layout demo (see the online demo, for instance http://layout.demo.nuxeo.org/nuxeo/layoutDemo/textWidget).
- `supportedModes`: the list of supported modes (for instance some widget types are read-only). This is useful for Studio configuration: if the edit mode is not available, the corresponding panel for properties configuration will not be shown.
- `fields`: this configuration is subject to change, but it is currently used to define what kind of widgets types are available for a given field type.
- `categories`: the list of categories for this widget type. This is a marker for display and it can also be used to facilitate exports. The default categories are "document", "summary", "listing" and "dev".
- `properties`: the layouts to use to display the available widget properties depending on the mode. This is a standard layout configuration, using the property name as field. Properties hidden in the mode "view_reference" will only be displayed on the reference table, and will not be displayed for configuration in Studio or preview in the Layout showcase.

Layout and Widget Display

Layouts can be displayed thanks to a series a JSF tags that will query the web layout service to get the layout definition and build it for a given mode.
The more common way to display a given layout for a document is to use the `nxl:layout` tag:

```
<div xmlns:nxl="http://nuxeo.org/nxforms/layout">
  <nxl:layout name="heading" mode="view" value="#{currentDocument}" />
</div>
```

Layouts that are referenced on a document type definition can use other helper tags, see the corresponding documentation.

> ⚠ You can include a layout in a `dataTable` tag, but cannot make its mode depend on the iteration variable. If you need to do so, recommendation is to use the `c:forEach` tag and handle all the <table>, <tr>, <td>... tags by yourself.

For instance, here is a sample display of a listing layout. The layout template is configured to display table rows. It will display header rows when the parameter `showListingHeader` is true.

```
<table class="dataOutput">
  <c:forEach var="row" items="#{documents.rows}" varStatus="layoutListingStatus">
    <c:set var="showListingHeader" value="#{layoutListingStatus.index == 0}" />
    <nxl:layout name="#{layoutName}" value="#{row}" mode="view"
      selectedColumns="#{selectedResultLayoutColumns}" />
  </c:forEach>
</table>
```

Some other advanced tags make it possible to display a global widget for instance, or even to create a widget from scratch by specifying its definition using the tag attributes.

Here is a sample usage of the `nxl:widget` tag:

```
<nxl:widget name="widgetName" mode="#{myMode}" value="#{myObject}" required="true"
/>
```

Here is a sample usage of the `nxl:widgetType` tag (creating a widget definition on the fly):

```
<nxl:widgetType name="text" mode="#{myMode}" value="#{myText}" required="true" />
```

Please refer to the tag library documentation available at http://community.nuxeo.com/api/nuxeo/release-5.8/tlddoc/nxl/tld-summary.html.

Related sections in this documentation

- Layout and Widget Definitions
- Layout and Widget Modes
- Custom Layout and Widget Templates

Related section in Studio documentation

- Form Layouts
- Tabs

Generic Layout Usage

Layouts can be used with other kind of objects than documents.

The field definition has to match a document property for which setters and getters will be available, or the `value` property must be passed explicitly for the binding to happen. Depending on the widget, other kinds of bindings can be done.

Since Nuxeo Platform 5.6, the field definition can contain the complete binding itself and be independent from the value passed to the tag. It just needs to be formatted like an EL expression, for instance: `#{myBinding}`.

Customize the Versioning and Comment Widget on Document Edit Form

On documents edit form, a **Comment** textarea is displayed, and this text is visible in the **History** tab. When document is versionable, versioning options are also displayed. This page provides some examples to customize this behavior. These examples can be contributed in Nuxeo Studio (Advanced Settngs > XML Extensions) or in Nuxeo IDE.

When using a **Toggleable Form** (`toggleableLayoutWithForms`) widget type, these fields can be shown by setting the property **Show Edit Options** (`showEditOptions`) to `true`, and they can be hidden by setting the property to `false`.

On the standard edit form, on the **Edit** tab, the layout showing these fields is included by default. Customizing it or hiding it can be done overriding the layout named `document_edit_form_options` .

This layout holds three widgets that can be customized independently:

- `document_edit_current_version`,
- `document_edit_versioning_options`,
- `document_edit_comment` .

Here is the original definition of this layout:

```
<extension target="org.nuxeo.ecm.platform.forms.layout.WebLayoutManager"
  point="layouts">
  <layout name="document_edit_form_options">
    <templates>
      <template mode="any">/layouts/layout_default_template.xhtml
      </template>
    </templates>
    <rows>
      <row>
        <widget>document_edit_comment</widget>
      </row>
      <row>
        <widget>document_edit_current_version</widget>
      </row>
      <row>
        <widget>document_edit_versioning_options</widget>
      </row>
    </rows>
  </layout>
</extension>
```

Emptying This Layout for All Documents

```
<require>org.nuxeo.ecm.platform.forms.layouts.webapp.base</require>

<extension target="org.nuxeo.ecm.platform.forms.layout.WebLayoutManager"
  point="layouts">
  <layout name="document_edit_form_options">
    <templates>
      <template mode="any">/layouts/layout_default_template.xhtml</template>
    </templates>
    <rows />
  </layout>
</extension>
```

Removing the Comment but Keeping Versioning Options

```
<require>org.nuxeo.ecm.platform.forms.layouts.webapp.base</require>

<extension target="org.nuxeo.ecm.platform.forms.layout.WebLayoutManager"
  point="layouts">
  <layout name="document_edit_form_options">
    <templates>
      <template mode="any">/layouts/layout_default_template.xhtml</template>
    </templates>
    <rows>
      <row>
        <widget>document_edit_current_version</widget>
      </row>
      <row>
        <widget>document_edit_versioning_options</widget>
      </row>
    </rows>
  </layout>
</extension>
```

Or you can play with the `hidden` widget mode:

```
<require>org.nuxeo.ecm.platform.forms.layouts.webapp.base</require>

<extension target="org.nuxeo.ecm.platform.forms.layout.WebLayoutManager"
  point="widgets">
  <widget name="document_edit_comment" type="textarea">
    <widgetModes>
      <mode value="any">hidden</mode>
    </widgetModes>
  </widget>
</extension>
```

Hiding the Comment Only on Some Document Types

```
<require>org.nuxeo.ecm.platform.forms.layouts.webapp.base</require>

<extension target="org.nuxeo.ecm.platform.forms.layout.WebLayoutManager"
  point="widgets">
  <widget name="document_edit_comment" type="textarea">
    <labels>
      <label mode="any">label.editComment</label>
    </labels>
    <helpLabels>
      <label mode="any">label.editComment.tooltip</label>
    </helpLabels>
    <translated>true</translated>
    <fields>
      <field>contextData['request/comment']</field>
    </fields>
    <widgetModes>
      <mode value="any">
        #{layoutMode == 'create' or layoutValue.type == 'myType'?'hidden':'edit'}
      </mode>
    </widgetModes>
  </widget>
</extension>
```

Making the Comment Mandatory

```
<require>org.nuxeo.ecm.platform.forms.layouts.webapp.base</require>

<extension target="org.nuxeo.ecm.platform.forms.layout.WebLayoutManager"
  point="widgets">
  <widget name="document_edit_comment" type="textarea">
    <labels>
      <label mode="any">label.editComment</label>
    </labels>
    <helpLabels>
      <label mode="any">label.editComment.tooltip</label>
    </helpLabels>
    <translated>true</translated>
    <fields>
      <field>contextData['request/comment']</field>
    </fields>
    <widgetModes>
      <mode value="create">hidden</mode>
    </widgetModes>
    <properties widgetMode="edit">
      <property name="required">true</property>
    </properties>
  </widget>
</extension>
```

Making the Comment Mandatory on a given Document Type

```
<require>org.nuxeo.ecm.platform.forms.layouts.webapp.base</require>

<extension target="org.nuxeo.ecm.platform.forms.layout.WebLayoutManager"
  point="widgets">
  <widget name="document_edit_comment" type="textarea">
    <labels>
      <label mode="any">label.editComment</label>
    </labels>
    <helpLabels>
      <label mode="any">label.editComment.tooltip</label>
    </helpLabels>
    <translated>true</translated>
    <fields>
      <field>contextData['request/comment']</field>
    </fields>
    <widgetModes>
      <mode value="create">hidden</mode>
    </widgetModes>
    <properties widgetMode="edit">
      <property name="required">
        #{layoutValue.type == 'myType'?true:false}
      </property>
    </properties>
  </widget>
</extension>
```

On this page

Content Views

Content Views can be configured using Studio, check out the Content Views documentation.

Definition

A content view is a notion to define all the elements needed to get a list of items and perform their rendering. The most obvious use case is the listing of a folderish document content, where we would like to be able to perform several actions.

- Defining the NXQL query that will be used to retrieve the documents, filtering some of them (documents in the trash for instance).
- Passing on contextual parameters to the query (the current container identifier).
- Defining a filtering form to refine the query.

- Defining what columns will be used for the rendering of the list, and how to display their content.
- Handling selection of documents, and actions available when selecting them (copy, paste, delete...).
- Handling sorting and pagination.
- Handling caching, and refresh of this cache when a document is created, deleted, modified, etc.

The Nuxeo Content View framework makes it possible to define such an object, by registering content views to the service. Here is a sample contribution, that will display the children of the current document:

```
<extension
target="org.nuxeo.ecm.platform.ui.web.ContentViewService
"
  point="contentViews">

  <contentView name="document_content">

    <coreQueryPageProvider>
      <property
name="coreSession">#{documentManager}</property>
      <pattern>
        SELECT * FROM Document WHERE ecm:parentId = ?
        AND ecm:isCheckedInVersion = 0
        AND ecm:mixinType != 'HiddenInNavigation'
        AND ecm:currentLifeCycleState != 'deleted'
      </pattern>
      <parameter>#{currentDocument.id}</parameter>
      <sort column="dc:title" ascending="true" />
      <pageSize>20</pageSize>
    </coreQueryPageProvider>

    <cacheKey>#{currentDocument.id}</cacheKey>
    <cacheSize>10</cacheSize>
    <refresh>
      <event>documentChanged</event>
      <event>documentChildrenChanged</event>
    </refresh>

    <resultLayouts>
      <layout name="document_listing_ajax"
title="document_listing"
        translateTitle="true"
iconPath="/icons/document_listing_icon.png"
        showCSVExport="true" showPDFExport="true"
showSyndicationLinks="true" />
      <layout
name="document_listing_ajax_compact_2_columns"
        title="document_listing_compact_2_columns"
        translateTitle="true"

iconPath="/icons/document_listing_compact_2_columns_icon
.png" />
      <layout
name="document_listing_ajax_icon_2_columns"
        title="document_listing_icon_2_columns"
        translateTitle="true"

iconPath="/icons/document_listing_icon_2_columns_icon.pn
g" />
    </resultLayouts>

    <selectionList>CURRENT_SELECTION</selectionList>
    <actions category="CURRENT_SELECTION_LIST" />

  </contentView>

</extension>
```

In this section

The Content View Query

`coreQueryPageProvider` element

The `coreQueryPageProvider` element makes it possible to define what query will be performed. Here it is a query on a core session, using a pattern with one parameter.

`parameter` and `property` elements

The `coreQueryPageProvider` element accepts any number of property elements, defining needed context variables for the page provider to perform its work. The `coreSession` property is mandatory for a core query to be processed and is bound to the core session proxy named `documentManager` available in a default Nuxeo application.

It also accepts any number of `parameter` elements, where order of definition matters: this EL expression will be resolved when performing the query, replacing the '?' characters it holds.

The main difference between `properties` and `parameters` is that properties will not be recomputed when refreshing the provider, whereas parameters will be. Properties will only be recomputed when resetting the provider.

`sort` elements

The `sort` element defines the default sort, that can be changed later through the interface. There can be any number of `sort` elements. The `sortInfosBinding` element can also be defined: it can resolve an EL expression in case the sort info are held by a third party instance (document, Seam component...) and will be used instead of the default sort information if not null or empty. The EL expression can either resolve to a list of `org.nuxeo.ecm.core.api.SortInfo` instances, or a list of map items using keys `sortColumn` (with a String value) and `sortAscending` (with a boolean value).

`pageSize` elements

The `pageSize` element defines the default page size, it can also be changed later. The `pageSizeBinding` element can also be defined: it can resolve an EL expression in case the page size is held by a third party instance (document, seam component...), and will be used instead of the default page size if not null.

The optional `maxPageSize` element can be placed at the same level than `pageSize`. It makes it possible to define the maximum page size so that the content view does not overload the server when retrieving a large number of items. When not set, the default value "100" will be used: even when asking for all the results with a page size with value "0" (when exporting the content view in CSV format for instance), only 100 items will be returned. Since 5.6 (and some previous hotfixed versions, see NXP-9052) this is configurable globally using the runtime property `nuxeo.pageprovider.default-max-page-size`.

`maxResults` elements

Since version 5.6, you can set a maximum number of results. This is useful for performance reasons, because it takes time to get the total number of results (and thus the number of pages). If there are more results than the limit, then the number of pages will be unknown. Nevertheless you can still navigate to the next page if it exists.

To set this limit you need to add a `maxResults` parameter to `coreQueryPageProvider`, either using an integer value or one of the

following keywords:

- DEFAULT_NAVIGATION_RESULTS: Used by most of the navigation page provider, the default is 200 and it can be overridden using Java options:

```
JAVA_OPTS=$JAVA_OPTS
-Dorg.nuxeo.ecm.platform.query.nxql.defaultNavigationResults=1000
```

  or you can directly add this line in nuxeo.conf:

```
org.nuxeo.ecm.platform.query.nxql.defaultNavigationResults=1000
```

- PAGE_SIZE: this is useful when you are interested in a single page or if you don't need a total count.

`whereClause` element

This kind of core query can also perform a more complex form of query, using a document model to store query parameters. Using a document model makes it easy to:

- use a layout to display the form that will define query parameters;
- save this document in the repository, so that the same query can be replayed when viewing this document.

Here is an example of such a registration:

```
<extension target="org.nuxeo.ecm.platform.ui.web.ContentViewService"
  point="contentViews">

  <contentView name="document_content">

    <coreQueryPageProvider>
      <property name="coreSession">#{documentManager}</property>
      <property name="maxResults">DEFAULT_NAVIGATION_RESULTS</property>
      <whereClause docType="AdvancedSearch">

        <predicate parameter="dc:title" operator="FULLTEXT">
          <field schema="advanced_search" name="title" />
        </predicate>

        <predicate parameter="dc:created" operator="BETWEEN">
          <field schema="advanced_search" name="created_min" />
          <field schema="advanced_search" name="created_max" />
        </predicate>

        <predicate parameter="dc:modified" operator="BETWEEN">
          <field schema="advanced_search" name="modified_min" />
          <field schema="advanced_search" name="modified_max" />
        </predicate>

        <predicate parameter="dc:language" operator="LIKE">
          <field schema="advanced_search" name="language" />
        </predicate>

        <predicate parameter="ecm:currentLifeCycleState" operator="IN">
          <field schema="advanced_search" name="currentLifeCycleStates" />
        </predicate>

        <fixedPart>
          ecm:parentId = ? AND ecm:isCheckedInVersion = 0 AND ecm:mixinType !=
          'HiddenInNavigation' AND ecm:currentLifeCycleState != 'deleted'
        </fixedPart>

      </whereClause>
      <parameter>#{currentDocument.id}</parameter>
      <sort column="dc:title" ascending="true" />
      <pageSize>20</pageSize>
    </coreQueryPageProvider>

    <searchLayout name="document_content_search"
      filterDisplayType="quick" />
    <showFilterForm>true</showFilterForm>

    ...

</extension>
```

The above definition holds a `whereClause` element, stating the search document type and predicates explaining how the document model properties will be translated into a NXQL query. It can also state a `fixedPart` element that will added as to the query string. This fixed part can also take parameters using the '?' character and `parameter` elements.

The `fixedPart` element also accepts attributes to better control its behavior:

- `escapeParameters` is a boolean value that allows to avoid escaping parameters when building the fixed part (defaults to true).

- `quoteParameters` is a boolean value that allows to avoid adding quotes around the parameter (defaults to true). This is useful when parameters actually hold a complete predicate, for instance (and in this case, the `escapeParameters` element must also be set to true.

Attributes `escapeParameters` and `quoteParameters` are also accepted on the `pattern` element.
`searchLayout` element

The `searchLayout` element defines what layouts needs to be used when rendering the search document model: it will be in charge of displaying the search form. Since 5.5, this element accepts a `filterDisplayType` attribute. When set to "quick", it will display a form showing only the first row of the layout, visible directly above the content view results. The whole filter form is then displayed in a popup. Otherwise, the default rendering is used, and the filter form is visible in a foldable box.

The `showFilterForm` element makes it possible to show this form above the content view results.

The Content View Result Layouts

The result layouts control the display of resulting documents. It states different kinds of rendering so that it's possible to switch between them. They also accept a title and an icon, useful for rendering.

The layout configuration is standard and has to follow listing layouts configuration standards. The layout template, as well as widgets displaying selection checkboxes, need to perform an Ajax selection of documents, and re-render the action buttons region.

The Content View Selection List

The `selectionList` element will be used to fill the document list with given name.

Selection is done through Ajax, so that selection is not lost when not performing any action thanks to this selection.

The Content View Actions

The `actions` element can be repeated any number of times: it states the actions category to use to display buttons applying to this table ("copy", "paste", "delete",...). Each `actions` element will generate a new row of buttons.

These actions will be displayed under the table in default templates, and will be re-rendered when selecting an item of the table so that they are enabled or disabled. This is performed using adequate filters, performing checks on selected items.

Additional Configuration

`searchDocument`

The `searchDocument` variable can be used in EL expressions to bind the page size, the sort information and the result columns to the search document properties.

Sample usage:

```
<contentView name="myContentView">
  <coreQueryPageProvider>
    <property name="coreSession">#{documentManager}</property>
    <whereClause docType="AdvancedSearch">
      <fixedPart>
        ecm:currentLifeCycleState != 'deleted'
      </fixedPart>
      <predicate parameter="dc:title" operator="FULLTEXT">
        <field schema="dublincore" name="title" />
      </predicate>
      <pageSizeBinding>#{searchDocument.cvd.pageSize}</pageSizeBinding>
      <sortInfosBinding>#{searchDocument.cvd.sortInfos}</sortInfosBinding>
    </whereClause>
  </coreQueryPageProvider>
  [...]
</contentView>
```

The `searchDocument` element can be filled on a content view using an EL expression: it will be used as the search document model. Otherwise, a bare document will be generated from the document type.

Sample usage, showing how to add a clause to the search depending on title set on the current document (will display non deleted document

with the same title):

```
<contentView name="sampleContentViewWithCustomSearchDocument">
  <searchDocument>#{currentDocument}</searchDocument>
  <coreQueryPageProvider>
    <property name="coreSession">#{documentManager}</property>
    <whereClause docType="AdvancedSearch">
      <fixedPart>
        ecm:currentLifeCycleState != 'deleted'
      </fixedPart>
      <predicate parameter="dc:title" operator="FULLTEXT">
        <field schema="dublincore" name="title" />
      </predicate>
    </whereClause>
  </coreQueryPageProvider>
</contentView>
```

`resultColumns` element

The `resultColumns` element can be filled on a content view using an EL expression: it will be used to resolve the list of selected columns for the current result layout. If several result layouts are defined, they should be configured so that their rows are always selected in case the selected column names do not match theirs.

Sample usage, showing how to reuse the same selected columns than the one selected on the advanced search page:

```
<contentView name="myContentView">
  [...]
  <resultColumns>
    #{documentSearchActions.selectedLayoutColumns}
  </resultColumns>
</contentView>
```

Additional rendering information

Additional rendering information can also be set, to be used by templates when rendering the content view:

```
<contentView name="CURRENT_DOCUMENT_CHILDREN">
  <title>label.current.document.children</title>
  <translateTitle>true</translateTitle>
  <iconPath>/icons/document_listing_icon.png</iconPath>
  <emptySentence>label.content.empty.search</emptySentence>
  <translateEmptySentence>true</translateEmptySentence>
  <translateEmptySentence>true</translateEmptySentence>
  <showPageSizeSelector>true</showPageSizeSelector>
  <showRefreshCommand>true</showRefreshCommand>

...
</contentView>
```

The element `showTitle` can be used to define a title for the content view, without displaying it on the default rendering. It can also be used when exporting the content view in CSV format, for instance.

The elements `emptySentence` and `translateEmptySentence` are used to display the message stating that there are no elements in the content view.

The elements `showPageSizeSelector` and `showRefreshCommand` are used to control the display of the page size selector, and of the "refresh current page" button. They both default to true.

Caching

The `cacheKey` element, if filled, will make it possible to keep content views in cache in the current conversation. It accepts EL expressions, but a static cache key can be used to cache only one instance.

The `cacheSize` element is useful to use a queue of cached instances. In the example, 10 instances of content views with a different cache key will be kept in cache. When the 11th entry, with a new cache key, is generated, the first content view put in the cache will be removed, and will need to be re-generated again. This cache configuration will make it possible to navigate to 10 different folderish document pages, and keep the current page, the current sort information, and current result layout.

When caching only one instance, setting the `cacheSize` element to more than "1" is useless. Caching only one instance can be useful when several features need to retrieve information from the same content view.

If a cache key is given, but no cache size is set, "5" will be used by default. Using "0" means no caching at all (and the cache key will be ignored).

Caching is done by a Seam component named `contentViewActions`. Although the cache key, cache size and events configurations handle the most common use cases, it is sometimes useful to call this bean methods directly when forcing a refresh.

The `refresh` and `reset` elements configurations make it possible to refresh/reset this content view when receiving the listed Seam event names. Only `documentChanged` and `documentChildrenChanged` are handled by default, but it is possible to react to new events by adding a method with an observer on this event on a custom Seam component, and call the method `contentViewActions.refreshOnSeamEvent(String seamEventName)` or `contentViewActions.resetPageProviderOnSeamEvent(String seamEventName)`.

Refresh will keep current settings, and will force the query to be done again. Reset will delete content views completely from the cache, and force complete re-generation of the content view, its provider, and the search document model if set.

> **cache size "0" behaviour**
> Before 5.7.1, selection actions were misbehaving when using a cache of size "0", so content views with selections actions needed a cache size of at least "1". Since 5.7.1 (and 5.6-HF02), when using value "0", the content view is cached anyhow, but its page provider is refreshed every time it is rendered.
>
> As this behavior is costly, using refresh events can be enough most of the time. But it is not possible to trigger a refresh for other users when using Seam events, so this configuration makes it possible to make sure the content view is up to date when other users may have an impact on its content.

Document Content Views

It is possible to define content views on a document type. This makes it easier to define folderish documents views.

Here is the default configuration of content views for Nuxeo folderish documents:

```
<type id="Folder">
  <label>Folder</label>
  ...
  <contentViews category="content">
    <contentView>document_content</contentView>
  </contentViews>
  <contentViews category="trash_content">
    <contentView showInExportView="false">document_trash_content</contentView>
  </contentViews>
</type>
```

The `document_content` content view will be displayed on this folder default view, and the `document_trash_content` content view will be displayed on the Trash tab.

The `category` attribute is filled from XHTML templates to render all content views defined in a given category.

The `showInExportView` attribute is used to check whether this content view should be displayed in the document export view (and PDF export).

If several content views are filled in the same category, both will be displayed on the same page.

Rendering

Rendering is done using methods set on generic Seam components: `contentViewActions` (`org.nuxeo.ecm.webapp.contentbrowser.ContentViewActions`) and `documentContentViewActions` (`org.nuxeo.ecm.webapp.contentbrowser.DocumentContentVi`

ewActions) to handle document content views categories.

A typical usage of content views, to render the results, would be:

```
<nxu:set var="contentViewName" value="my_content_view_name">

  <ui:decorate template="/incl/content_view.xhtml" />

</nxu:set>
```

The template `/incl/content_view.xhtml` handles generic rendering of the given content view (content view title, pagination, result layout selection, list rendering, actions rendering) . It inserts names region that can be overridden when using the `ui:decorate` tag.

The current version of this template is here: https://github.com/nuxeo/nuxeo-jsf/blob/release-5.8/nuxeo-platform-webapp-base/src/main/resources/web/nuxeo.war/incl/content_view.xhtml.

Here is the sample rendering of the search form defined on a content view named "document_content_filter":

```
<nxu:set var="contentView"

value="#{contentViewActions.getContentViewWithProvider('document_content_filter')}"
  cache="true">
  <c:if test="#{contentView != null}">
    <nxl:layout name="#{contentView.searchLayout.name}" mode="edit"
      value="#{contentView.searchDocumentModel}" />
  </c:if>
</nxu:set>
```

Here is a typical way of refreshing or resetting a provider named "advanced_search" from the interface:

```
<div>
  <h:commandButton value="#{messages['command.search']}"
    action="search_results_advanced"
    styleClass="button">
    <nxu:actionListenerMethod
value="#{contentViewActions.refresh('advanced_search')}" />
  </h:commandButton>
  <h:commandButton value="#{messages['command.clearSearch']}"
    action="#{contentViewActions.reset('advanced_search')}"
    immediate="true"
    styleClass="button" />
</div>
```

Related topics in this documentation

- Document Content Views

Related topics in Studio documentation

- Content Views

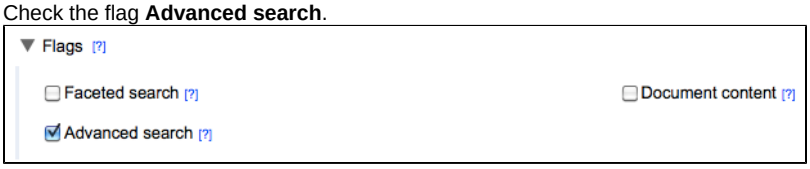
Configure a Domain Specific Advanced Search

Nuxeo Studio enables you to create new search forms to leverage your specific document types and metadata. Using Nuxeo DM local configuration, you can choose to override the default search form with your own. For instance, you can define a new advanced search form in Nuxeo Studio, that you will be able to use instead of the default advanced search form on a domain.

Search forms are based on content views.

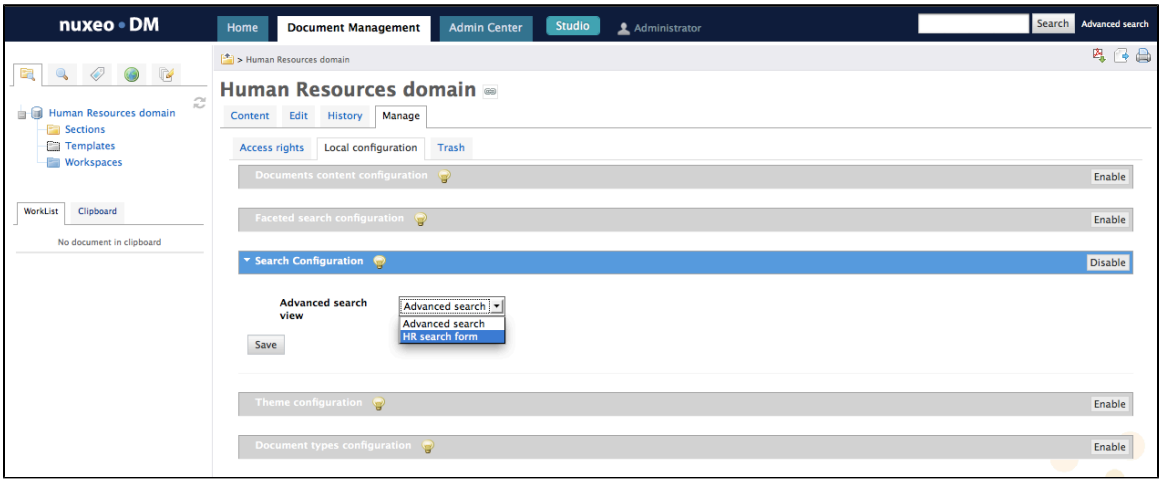
To create a new advanced search form:

1. Define your query filter, that is to say the fixed part of the query that will be executed when the user clicks on the "Search" button.
2. Check the flag **Advanced search**.



3. Design the search form that will be displayed to users:
   The page Define a new content view does not exist.
4. Configure the results layout.

After you have configured your new advanced search form and updated your Nuxeo DM instance with your last Studio changes, you can select your new advanced search form in your domain's local configuration.



Related How-to's

- How to Add a New Virtual Navigation Entry
- Configure a Domain Specific Advanced Search
- Configure a Domain Specific Advanced Search
- How to Customize the Default Content and Trash Listings
- How-To Index

Related Documentation

- Content Views
- Content Views in Studio Documentation
- Custom Page Providers
- Documents Display Configuration
- Default Search

Custom Page Providers

This chapter focuses on writing custom page providers, for instance when you'd like to use content views to query and display results from an external system.

For an introduction to content views, please refer to the Content Views chapter.

Page Providers Configuration

The `<coreQueryPageProvider>` element makes it possible to answer to most common use cases. If you would like to use another kind of query, you can use an alternate element and specify the `PageProvider` class to use.

Here is a sample example of a custom page provider configuration:

```
<extension target="org.nuxeo.ecm.platform.ui.web.ContentViewService"
  point="contentViews">

  <contentView name="CURRENT_DOCUMENT_CHILDREN_FETCH">
    <genericPageProvider
      class="org.nuxeo.ecm.platform.query.nxql.CoreQueryAndFetchPageProvider">
      <property name="coreSession">#{documentManager}</property>
      <pattern>
        SELECT dc:title FROM Document WHERE ecm:parentId = ? AND
        ecm:isCheckedInVersion = 0 AND ecm:mixinType != 'HiddenInNavigation'
        AND ecm:currentLifeCycleState != 'deleted'
      </pattern>
      <parameter>#{currentDocument.id}</parameter>
      <sort column="dc:title" ascending="true" />
      <pageSize>2</pageSize>
    </genericPageProvider>

    ...
  </contentView>

</extension>
```

The `<genericPageProvider>` element takes an additional `class` attribute stating the page provider class. This class has to follow the `org.nuxeo.ecm.core.api.PageProvider` interface and does not need to list document models: content views do not force the item type to a given interface. The abstract class `org.nuxeo.ecm.core.api.AbstractPageProvider` makes it easier to define a new page provider as it implements most of the interface methods in a generic way.

As result layouts can apply to other objects than document models, their definition can be adapted to fit to the kind of results provided by the custom page provider.

In the given example, another kind of query will be performed on a core session, and will return a list of maps, each map holding the "dc:title" key and corresponding value on the matching documents.

The `<genericPageProvider>` element accepts all the other configurations present on the `<coreQueryPageProvider>` element: it is up to the `PageProvider` implementation to use them to build its query or not. It can also perform its own caching.

The properties can be defined as EL expressions and make it possible for the query provider to have access to contextual information. In the above example, the core session to the Nuxeo repository is taken from the Seam context and passed as the property with name `coreSession`.

Page Providers without Content Views

Content views are very linked to the UI rendering as they hold pure UI configuration and need the JSF context to resolve variables. Sometimes it is interesting to retrieve items using page providers, but in a non-UI context (event listener), or in a non-JSF UI context (WebEngine).

Page providers can be registered on their own service, and queried outside of a JSF context. These page providers can also be referenced from content views, to keep a common definition of the provider.

Here is a sample page provider definition:

```
<extension target="org.nuxeo.ecm.platform.query.api.PageProviderService"
  point="providers">

  <coreQueryPageProvider name="TREE_CHILDREN_PP">
    <pattern>
      SELECT * FROM Document WHERE ecm:parentId = ? AND ecm:isProxy = 0 AND
      ecm:mixinType = 'Folderish' AND ecm:mixinType != 'HiddenInNavigation'
      AND ecm:isCheckedInVersion = 0 AND ecm:currentLifeCycleState !=
      'deleted'
    </pattern>
    <sort column="dc:title" ascending="true" />
    <pageSize>50</pageSize>
  </coreQueryPageProvider>

</extension>
```

This definition is identical to the one within a content view, except it cannot use EL expressions for variables resolution. A typical usage of this page provider would be:

```
PageProviderService ppService = Framework.getService(PageProviderService.class);
Map<String, Serializable> props = new HashMap<String, Serializable>();
props.put(CoreQueryDocumentPageProvider.CORE_SESSION_PROPERTY,
        (Serializable) coreSession);
PageProvider<DocumentModel> pp = (PageProvider<DocumentModel>)
ppService.getPageProvider(
        "TREE_CHILDREN_PP", null, null, null, props,
        new Object[] { myDoc.getId() });
List<DocumentModel> documents = pp.getCurrentPage();
```

Here you can see that the page provider properties (needed for the query to be executed) and its parameters (needed for the query to be built) cannot be resolved from EL expressions: they need to be given explicitly to the page provider service.

A typical usage of this page provider, referenced in a content view, would be:

```
<extension target="org.nuxeo.ecm.platform.ui.web.ContentViewService"
  point="contentViews">

  <contentView name="TREE_CHILDREN_CV">
    <title>tree children</title>

    <pageProvider name="TREE_CHILDREN_PP">
      <property name="coreSession">#{documentManager}</property>
      <property name="checkQueryCache">true</property>
      <parameter>#{currentDocument.id}</parameter>
    </pageProvider>

  </contentView>

</extension>
```

Here you can see that properties and parameters can be put on the referenced page provider as content views all have a JSF context.

Documents Display Configuration

The views on documents, the forms to create or edit them, how lists of documents are presented, all that can be changed in a Nuxeo application,

to make sure the information displayed are meaningful. To enable the customization of how documents, forms and listings are presented, Nuxeo Platform-based application use layouts and content views.

In this section:

- Document Views — Pages visible for a given document can be configured depending on the document type.
- Document Layouts — Layouts can be linked to a document type by specifying the layout names in its definition.
- Document Content Views — Content Views can be linked to a document type by specifying the content view names in its definition.
- Drag and Drop Service for Content Capture (HTML5-Based) — Starting with Nuxeo 5.4.2, you can use the native HTML5 Drag and Drop features on recent browsers (Firefox 3.6+, Google Chrome 9+, Safari 5+). This new Drag and Drop import model is pluggable so you can adapt the import behavior to your custom needs.

Document Views

Pages visible for a given document can be configured depending on the document type.

First of all, we have to make the difference between a view in a standard JSF way (navigation case view id, navigation case output) and views in the Nuxeo Platform (document type view, creation view).

Standard JSF Navigation Concepts

A standard JSF navigation rule can be defined in the `OSGI-INF/deployment-fragment.xml` files, inside the `faces-config#NAVIGATION` directive.

Example of a navigation rule case definition:

```
<extension target="faces-config#NAVIGATION">

    <navigation-case>
      <from-outcome>create_document</from-outcome>
      <to-view-id>/create_document.xhtml</to-view-id>
      <redirect />
    </navigation-case>

    <navigation-case>
      <from-outcome>view_documents</from-outcome>
      <to-view-id>/view_documents.xhtml</to-view-id>
      <redirect />
    </navigation-case>

</extension>
```

Nuxeo Platform Views

A certain Nuxeo document type can have defined a default view (used to view/edit the document) and a create view (used to create the document). These views are specified in the `OSGI-INF/ecm-types-contrib.xml` file, as in the following example.

```
<extension target="org.nuxeo.ecm.platform.types.TypeService" point="types">
    <type id="Workspace">
      <label>Workspace</label>
      <icon>/icons/workspace.gif</icon>
      <icon-expanded>/icons/workspace_open.gif</icon-expanded>
      <default-view>view_documents</default-view>
      <create-view>create_workspace</create-view>
    </type>
</extension>
```

The default view of a document is rendered as a list of tabs. The document tabs are defined as actions in the `OSGI-INF/actions-contrib.xml` file, having as category VIEW_ACTION_LIST. A tab can be added to a document default view as shown in the following example.

```
<extension target="org.nuxeo.ecm.platform.actions.ActionService" point="actions">
    <action id="TAB_EDIT" link="/incl/tabs/document_edit.xhtml" enabled="true"
      order="20" label="action.view.edit" icon="/icons/file.gif">
      <category>VIEW_ACTION_LIST</category>
      <filter-id>edit</filter-id>
      <filter-id>mutable_document</filter-id>
    </action>
</extension>
```

Document Layouts

Layouts can be linked to a document type by specifying the layout names in its definition.

Here is a sample configuration excerpt:

```
<type id="Note">
  [...]
  <layouts mode="any">
    <layout>heading</layout>
    <layout>note</layout>
  </layouts>
</type>
```

Layouts are defined in a given mode; layouts in the "any" mode will be used as default when no layouts are given in specific modes.

It is possible to merge layouts when redefining the document type, adding a property `append="true"`:

```
<layouts mode="any" append="true">
  <layout>newLayout</layout>
</layouts>
```

A new mode "listing" can be used for folderish documents. Their default content will use the given layouts to make it possible to switch between the different presentations. Since 5.4.0, this configuration is deprecated as it is now possible to configure it through Content Views. Anyhow, some default listing layouts have been defined, the one used by default when no layout is given in this mode is `document_listing`. To remove the layouts defined by default on a document type, override it without listing any modes.

```
<layouts mode="listing">
</layouts>

<layouts mode="listing">
  <layout>document_listing</layout>
  <layout>document_listing_compact_2_columns</layout>
  <layout>document_icon_2_columns</layout>
</layouts>
```

Layouts with a name that ends with `2_columns` will be displayed on two columns by default. The layout name will be used as a message key for the selector label.

Since 5.8, the property named `display` set on the layout configuration is enough to handle a two columns rendering, it can be set to value table_2_columns for this purpose.

Document Layouts Display

The documentLayout tag can be used to display the layouts of a document:

```
<div xmlns="http://www.w3.org/1999/xhtml"
    xmlns:nxl="http://nuxeo.org/nxforms/layout">
  <nxl:documentLayout mode="view" value="#{currentDocument}" />
</div>
```

It is possible to make a distinction between the layouts defined in a given mode on the document, and the mode used to render layouts, for instance:

```
<nxl:documentLayout documentMode="header" mode="view"
  value="#{currentDocument}" defaultLayout="document_header"
  includeAnyMode="false" />
```

Document Content Views

Content Views can be linked to a document type by specifying the content view names in its definition.

Some default document views present content views for listing this folderish document content, for instance, or to present the result of a search using some of the document properties as parameters (like in the Smart Search addon).

A `category` has been added to make the distinction between the different views, here is a sample configuration:

```
<type id="Workspace">
  [...]
  <contentViews category="content">
    <contentView>document_content</contentView>
  </contentViews>
  <contentViews category="trash_content">
    <contentView showInExportView="false">
      document_trash_content
    </contentView>
  </contentViews>
</type>
```

The category `content` is looked up by the default tab showing a folderish document content. The category `trash_content` is looked up by the default tab showing a folderish document trash content.

Several content views can be shown on each of these views.

Drag and Drop Service for Content Capture (HTML5-Based)

Drag and Drop from the Desktop to Nuxeo HTML UI has been available for a long time using a browser plugin.
Starting with Nuxeo 5.4.2, you can use the native HTML5 Drag and Drop features on recent browsers (Firefox 3.6+, Google Chrome 9+, Safari 5+). This new Drag and Drop import model is pluggable so you can adapt the import behavior to your custom needs.

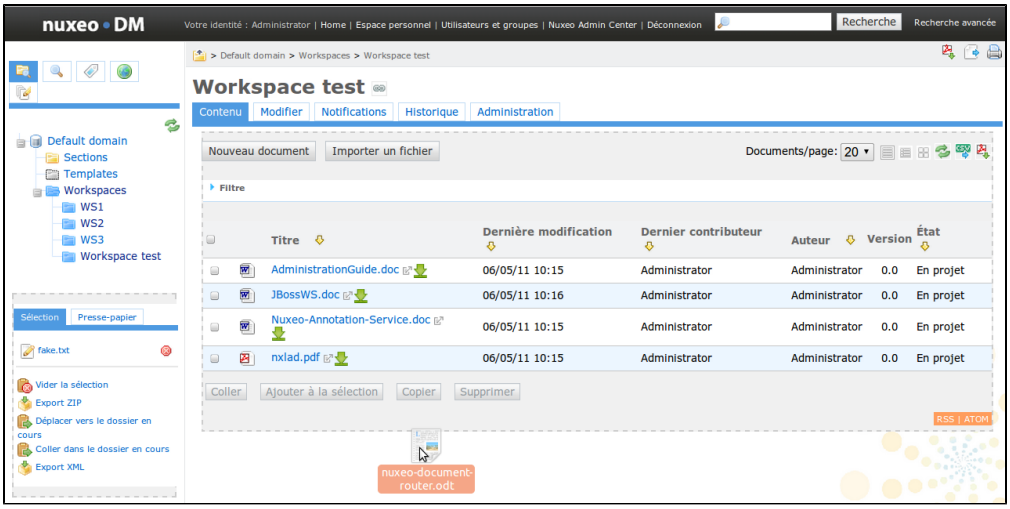
How to Use it

Selecting the DropZone

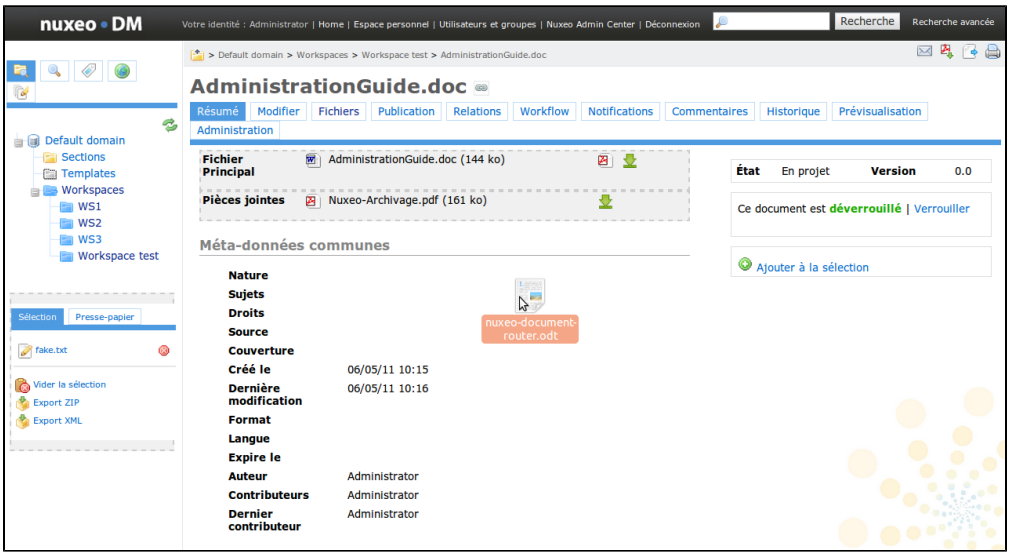
If you drag some files from the Desktop to the Nuxeo WebUI, the possible DropZones will be highlighted.

In Nuxeo DM there are 5 different DropZones (depending on the page):

- `ContentView`: the content listing for a folderish Document;



- `Clipboard_CLIPBOARD`: the user's Clipboard;
- `Clipboard_DEFAULT`: the user's Worklist;
- `mainBlob`: the main attachment of the current Document;



- `otherBlobs`: additional attachments of the current Document.

On this page

Depending on the DropZone you select, the import action will be different:

- Content view: create documents from files in the current container;
- Clipboard: create documents from files in the user's personal workspaces and add them to the clipboard;
- Worklist: create documents from files in the user's personal workspaces and add them to the worklist;
- main blob: attach a file to the document;
- other blobs: attach file(s) as additional files in the document.

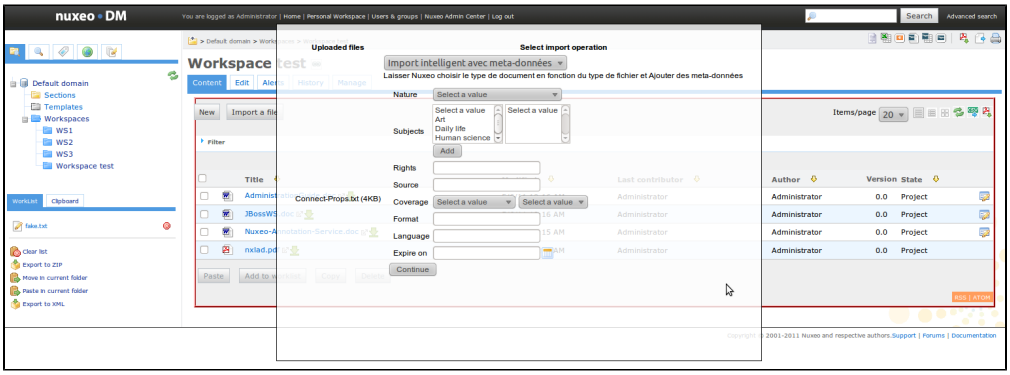
Default Mode vs Advanced Mode

In the default mode the file you drop will be automatically uploaded and imported into the document repository.

By using the advanced mode you can have more control over the import process:

- you can do several file imports but still keep all files part of the same batch,
- you can select the automation chain that will be executed.

To trigger the extended mode, just maintain the drag over the DropZone for more than 2.5 seconds: the drop zone will be highlighted in green indicating you are now in extended mode.



How to Customize it

Defining a New DropZone

You can very simply define a new DropZone in your pages; you simply need to add an HTML element (like a `div`) which has:

- a unique id,
- the '`dropzone`' CSS class,
- a context attribute.

Drop zone declaration

```
<div id="myDropZone" class="dropzone" context="myDropZone"> ... </div>
```

Associating Automation Chains

Each dropzone context is associated with a set of content automation operations or automation chains. This association is configured via the action service:

Binding an operation chain to a drop zone

```
<action id="Chain.FileManager.ImportInSeam"
        link="" order="10" label="label.smart.import"
        help="desc.smart.import.file">
  <category>ContentView</category>
  <filter-id>create</filter-id>
  <properties>
    <property name="chainId">FileManager.ImportInSeam</property>
  </properties>
</action>
```

Where:

- the operation or automation chain is configured through the action properties (since 5.7.1). The `chainId` property is used to configure the automation chain to execute. If not present, the `operationId` property is tried.
  For backward compatibility, if both properties are not present, we fallback using the action `id` to get the automation chain or operation to execute (for automation chains, append `chain.` as a prefix for id).
- `category` represents the dropzone context;
- `filter` / `filter-id` are the filter used to define if operation should be available in a given context;
- `link` points to a page that can be used to collect parameters for the automation chain.

The operation or chain that will be called for the import will receive:

- as input: a BlobList representing the files that have been uploaded;

- as context: the current page context.

```
typically : { currentDocument : '#{currentDocument.id}',
currentDomain : '#{currentDomain.id}',
currentWorkspace : '#{currentWorkspace.id}',
conversationId : '#{org.jboss.seam.core.manager.currentConversationId}',
lang : '#{localeSelector.localeString}',
repository : '#{currentDocument.repositoryName}'};
```

- as parameters: what has been collected by the form if any.

The output of the chains does not really matter.

At some point, inside your automation chain you may need to access Seam Context. For that, new operations were introduced:

- `Seam.RunOperation`: that can run an operation or a chain in the Seam context.
  For example, if you want to get available actions via the "Actions.GET" operation, but want to leverage Seam context for actions filters:

Running an operation in Seam Context

```
<chain id="SeamActions.GET">
  <operation id="Seam.RunOperation">
    <param type="string" name="id">Actions.GET</param>
  </operation>
</chain>
```

- `Seam.InitContext` / `Seam.DestroyContext`: that can be used to initialize / destroy seam context:

Manual Seam context management

```
<chain id="ImportClipboard">
  <operation id="Seam.InitContext" />
  <operation id="UserWorkspace.CreateDocumentFromBlob" />
  <operation id="Document.Save"/>
  <operation id="Seam.AddToClipboard"/>
  <operation id="Seam.DestroyContext" />
</chain>
```

Parameters Management

In some cases, you may want user to provide some parameters via a form associated to the import operation he wants to run. For that, you can use the `link` attribute of the action used to bind your automation chain to a dropzone. This URL will be used to display your form within an iframe inside the default import UI.

In order to send the collected parameters to the import wizard, you should call a JavaScript function inside the parent frame:

Calling back the import wizard

```
window.parent.dndFormFunctionCB(collectedData);
```

where `collectedData` is a JavaScript object that will then be sent (via JSON) as parameter of the operation call.

In the default JSF WebApp you can have a look at `DndFormActionBean` and `dndFormCollector.xhtml`.

Related Documentation
• Enabling Drag and Drop for Internet Explorer • Working Using Drag and Drop • Drag and Drop Compatibility Table • How to Change the Default Document Type When Importing a File in the Nuxeo Platform? • Blob Upload for Batch Processing

Searches Customization

This chapter presents the different search screens available on the application, and how to customize them.

- Simple Search — The simple search is configured to work in conjunction with a content view. This section describes the document type and layouts used in the default simple search.
- Advanced Search — The advanced search is configured to work in conjunction with a content view. This section describes the document type and layouts used in the default advanced search.
- Faceted Search — The faceted search is configured to work in conjunction with a content view. This section describes the document type and layouts used in the default faceted search.

Simple Search

The simple search is configured to work in conjunction with a content view. This section describes the document type and layouts used in the default simple search.

Simple Search Content View

The simple search content view is named `simple_search` and can be overridden see the contribution at search-contentviews-contrib.xml.

It shares part of its configuration with the Advanced Search to make user experience better (for instance search results layout and selected columns results).

Simple Search Box

Since 5.8, the simple search box is shown thanks to an action. To customize this box, the corresponding action contribution can be overriden. See http://explorer.nuxeo.org/nuxeo/site/distribution/Nuxeo%20Platform-5.8/viewContribution/org.nuxeo.ecm.platform.actions--actions

Suggestions

> The suggestions on simple search come with the Document Management module.

When the Document Management module is enabled, the simple search content view fallback on the Faceted Search tab and results.

It takes over the simple search by disabling the action showing the simple search box, and adding the suggester one. See https://github.com/nuxeo/nuxeo-platform-suggestbox/blob/release-5.8/nuxeo-platform-suggestbox-jsf/src/main/resources/OSGI-INF/suggestbox-actions-contrib.xml.

Suggesters can be contributed to this search box. See the contribution at http://explorer.nuxeo.org/nuxeo/site/distribution/Nuxeo%20Platform-5.8/viewContribution/org.nuxeo.ecm.platform.suggestbox.core.defaultSuggesters--suggesters.

Advanced Search

The advanced search is configured to work in conjunction with a content view. This section describes the document type and layouts used in the default advanced search.

"advanced_search" Content View

The content view used to configure the advanced search is named advanced_search and can be overridden to customize this page. See the contribution at search-contentviews-contrib.xml.

Additional content views for advanced search can be configured to be made available in Local Search Configurations, but using the flag named ADVANCED_SEARCH .

This is easy to perform using Studio, see Configure a Domain Specific Advanced Search.

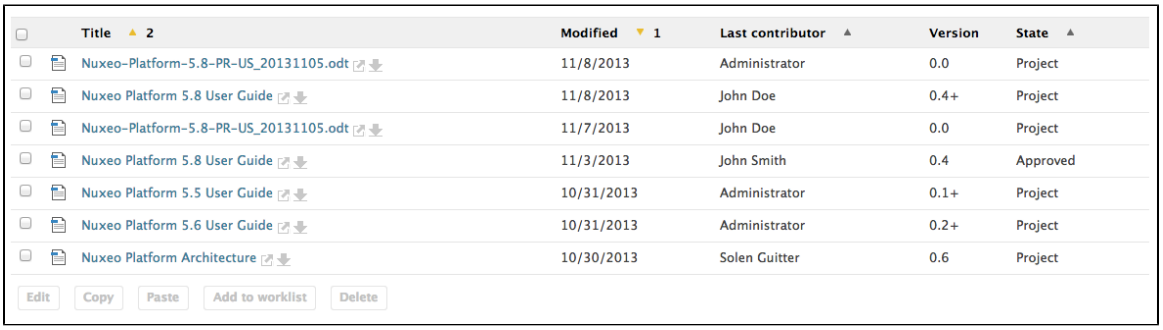
In this section

"AdvancedSearch" Document Type

The `AdvancedSearch` document type is attached to this content view, and will store query parameters. Its default schema definition can be found here: advanced_search.xsd. It can be overridden to add new custom fields, or redefined completely as the document type is only referenced in the content view.

Its content view definition shows the mapping between this document properties and the query to build (see search-contentviews-contrib.xml).

It also references the search layout (defined here: search-layouts-contrib.xml) and the result layouts (defined here: layouts-listing-contrib.xml).



Title ▲ 2	Modified ▼ 1	Last contributor ▲	Version	State ▲
Nuxeo-Platform-5.8-PR-US_20131105.odt	11/8/2013	Administrator	0.0	Project
Nuxeo Platform 5.8 User Guide	11/8/2013	John Doe	0.4+	Project
Nuxeo-Platform-5.8-PR-US_20131105.odt	11/7/2013	John Doe	0.0	Project
Nuxeo Platform 5.8 User Guide	11/3/2013	John Smith	0.4	Approved
Nuxeo Platform 5.5 User Guide	10/31/2013	Administrator	0.1+	Project
Nuxeo Platform 5.6 User Guide	10/31/2013	Administrator	0.2+	Project
Nuxeo Platform Architecture	10/30/2013	Solen Guitter	0.6	Project

Edit Copy Paste Add to worklist Delete

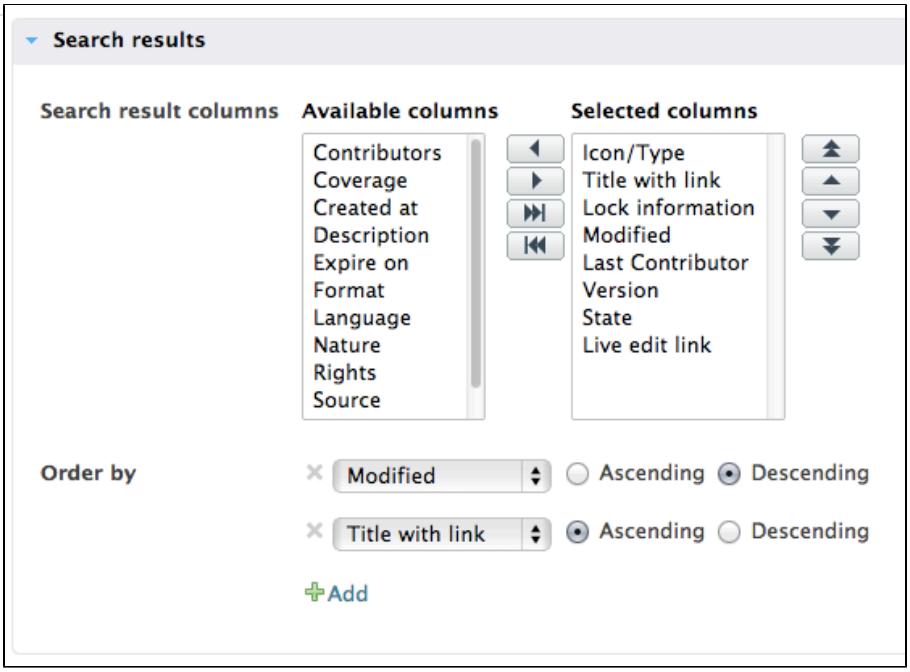
"search_form" and "search_results_advanced" Layouts

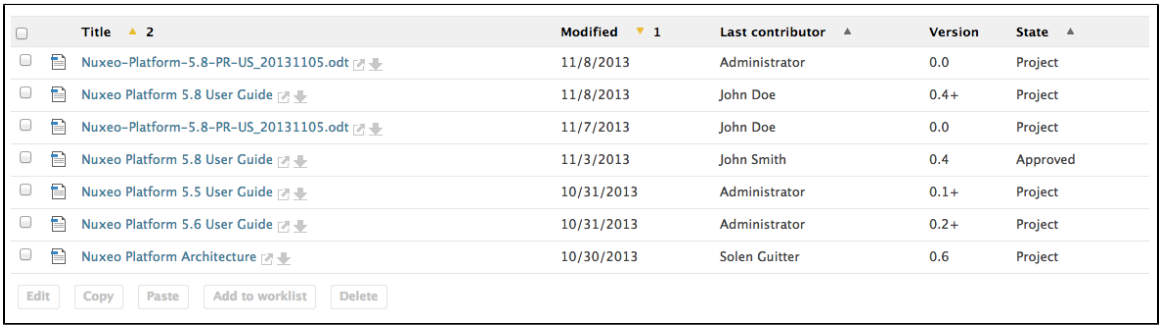
The search form and search results reference the content view name (see search_form.xhtml and search_results_advanced.xhtml) and also use the Seam component DocumentSearchActions to store the sort information and selected result columns.

The result layout used here is named `search_listing_ajax`. It is configured as a standard listing layout, and holds additional information on its columns definition so that it can be used to display the search columns selection and the sort information available columns. It is used with the following modes:

- "edit_columns" when displaying the column selection widget as shown above.
- "edit_sort_infos" or "edit_sort_infos_map" when displaying the sort information list widget as shown above. These two modes are equivalent, but the new item to add to the list is a `org.nuxeo.ecm.core.api.SortInfo` instance in the first case, and a map with keys "sortColumn" and "sortAscending" in the second case.
- "view" when displaying the search results table (listing layout).

Here are screenshots of this layout rendered in these modes:





Here is an excerpt of this layout definition:

```
<layout name="search_listing_ajax">
  <templates>
    <template mode="any">
      /layouts/layout_listing_ajax_template.xhtml
    </template>
    <template mode="edit_columns">
      /layouts/layout_column_selection_template.xhtml
    </template>
    <template mode="edit_sort_infos">
      /layouts/layout_sort_infos_template.xhtml
    </template>
    <template mode="edit_sort_infos_map">
      /layouts/layout_sort_infos_template.xhtml
    </template>
  </templates>
  <properties mode="any">
    <property name="showListingHeader">true</property>
    <property name="showRowEvenOddClass">true</property>
  </properties>
  <properties mode="edit_columns">
    <property name="availableElementsLabel">
      label.selection.availableColumns
    </property>
    <property name="selectedElementsLabel">
```

```
    label.selection.selectedColumns
  </property>
  <property name="selectedElementsHelp"></property>
  <property name="selectSize">10</property>
  <property name="required">true</property>
  <property name="displayAlwaysSelectedColumns">false</property>
</properties>
<properties mode="edit_sort_infos">
  <property name="newSortInfoTemplate">
    #{documentSearchActions.newSortInfo}
  </property>
  <property name="required">false</property>
</properties>
<properties mode="edit_sort_infos_map">
  <property name="newSortInfoTemplate">
    #{documentSearchActions.newSortInfoMap}
  </property>
  <property name="required">false</property>
</properties>
<columns>
  <column name="selection" alwaysSelected="true">
    <properties mode="any">
      <property name="isListingSelectionBox">true</property>
      <property name="useFirstWidgetLabelAsColumnHeader">false</property>
      <property name="columnStyleClass">iconColumn</property>
    </properties>
    <widget>listing_ajax_selection_box</widget>
  </column>
  <column name="title_link">
    <properties mode="any">
      <property name="useFirstWidgetLabelAsColumnHeader">true</property>
      <property name="sortPropertyName">dc:title</property>
      <property name="label">label.selection.column.title_link</property>
    </properties>
    <properties mode="edit_sort_infos">
      <property name="showInSortInfoSelection">true</property>
    </properties>
    <properties mode="edit_sort_infos_map">
      <property name="showInSortInfoSelection">true</property>
    </properties>
    <widget>listing_title_link</widget>
  </column>
  [...]
  <column name="description" selectedByDefault="false">
    <properties mode="any">
      <property name="useFirstWidgetLabelAsColumnHeader">true</property>
      <property name="sortPropertyName">dc:description</property>
      <property name="label">description</property>
    </properties>
    <properties mode="edit_sort_infos">
      <property name="showInSortInfoSelection">true</property>
    </properties>
    <properties mode="edit_sort_infos_map">
      <property name="showInSortInfoSelection">true</property>
    </properties>
    <widget>listing_description</widget>
  </column>
  <column name="subjects" selectedByDefault="false">
    <properties mode="any">
```

```
      <property name="useFirstWidgetLabelAsColumnHeader">true</property>
      <property name="label">label.dublincore.subject</property>
    </properties>
    <widget>listing_subjects</widget>
  </column>
  [...]
</columns>
```

```
</layout>
```

All the columns have names defined so that this value can be used as the key when computing the list of selected columns. If not set, the name will be generated according to the column position in the layout definition, but as this definition may change, it is recommended to set specific names for better maintenance and upgrade.

The columns that should not be selected by default hold the additional parameter `selectedByDefault`, and it is set to "false" as all columns (and rows) are considered selected by default. Hence the "description" and "subjects" columns are not selected by default, and shown in the left selector when displaying this layout in mode "edit_sort_infos" or "edit_sort_infos_map".

Properties defined on the layout in mode "edit_columns" are used by the layout template layout_column_selection_template.xhtml.

Properties defined on the layout and columns in mode "edit_sort_infos" or "edit_sort_infos_map" are used by the layout template layout_sort_infos_template.xhtml. This template filters presentation of columns that do not hold the property "`showInSortInfoSelection`" set to "true" as some columns may not support sorting (for instance, as sorting cannot be done on the "subjects" complex property, the associated column should not be made available in the sort selection widget).

The column selection and sort information will be taken into account by the content view if:

- the template displaying results binds this information to the backing bean holding the values, for instance:

```
<nxu:set var="contentViewId" value="advanced_search">
<nxu:set var="contentViewName" value="advanced_search">

  <ui:decorate template="/incl/content_view.xhtml">
    <ui:param name="selectedResultLayoutColumns"
      value="#{documentSearchActions.selectedLayoutColumns}" />
    <ui:param name="contentViewSortInfos"
      value="#{documentSearchActions.searchSortInfos}" />
[...]
```

- or the content view definition holds these bindings, for instance:

```
<contentView name="advanced_search">

  <coreQueryPageProvider>
    [...]
    <pageSize>20</pageSize>
    <sortInfosBinding>
      #{documentSearchActions.searchSortInfos}
    </sortInfosBinding>
  </coreQueryPageProvider>

  [...]

  <resultLayouts>
    <layout name="search_listing_ajax" title="document_listing"
      translateTitle="true" iconPath="/icons/document_listing_icon.png" />
   [...]
  </resultLayouts>
  <resultColumns>
    #{documentSearchActions.selectedLayoutColumns}
  </resultColumns>

  [...]
</contentView>
```

Faceted Search

The faceted search is configured to work in conjunction with a content view. This section describes the document type and layouts used in the default faceted search.

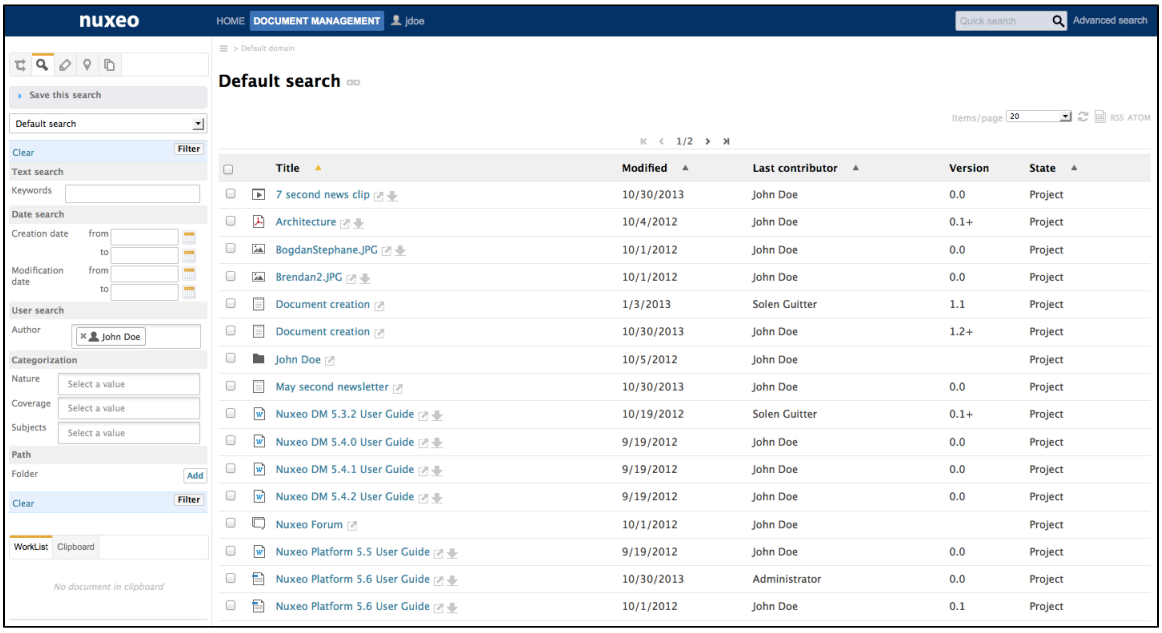
Overview

The faceted search adds a new way to browse the whole document repository using visual filters on metadata.

> This feature comes with the Document Management module.

The new tab appears on the left, just after the navigation tree.

Here is a screenshot of the default faceted search:



In this section

Saved Faceted Searches

The saved searches are stored in the personal workspace of the user who saved the search.

The folder where the searches are stored can be configured through an extension point on the `FacetedSearchService` :

```
<extension
target="org.nuxeo.ecm.platform.faceted.search.jsf.service.FacetedSearchService"
 point="configuration">

 <configuration>
   <rootSavedSearchesTitle>Saved Searches</rootSavedSearchesTitle>
 </configuration>

</extension>
```

How to Contribute a New Faceted Search

We will see how to contribute a new faceted search with the default faceted search in Nuxeo DM as an example.

Content Views Contribution

A faceted search is just a content view with the `FACETED_SEARCH` flag set.
When defining the content view for your faceted search, you'll end up defining the `CoreQueryPageProvider` that will be the definition of the query done to retrieve the documents matching your criteria.

To register your content view as a faceted search, don't forget to add the correct flag in the contribution:

```
<flags>
  <flag>FACETED_SEARCH</flag>
</flags>
```

To understand all the parameters of the contribution, have a look at: Content Views

The key attributes are:

- `docType`: defines which document type to use to populate the values in the query.
- `searchLayout`: defines which layout will be rendered for this faceted search.

Here is the whole contribution of the content view used for the default faceted search in Nuxeo DM:

```
<extension target="org.nuxeo.ecm.platform.ui.web.ContentViewService"
  point="contentViews">

  <contentView name="faceted_search_default">
    <title>label.faceted.search.default</title>
    <translateTitle>true</translateTitle>

    <coreQueryPageProvider>
      <property name="coreSession">#{documentManager}</property>
      <whereClause docType="FacetedSearchDefault">
        <fixedPart>
          ecm:mixinType != 'HiddenInNavigation' AND
          ecm:mixinType != 'HiddenInFacetedSearch' AND ecm:isCheckedInVersion = 0
          AND ecm:currentLifeCycleState != 'deleted'
        </fixedPart>
        <predicate parameter="ecm:fulltext" operator="FULLTEXT">
          <field schema="faceted_search_default" name="ecm_fulltext" />
        </predicate>
        <predicate parameter="dc:created" operator="BETWEEN">
          <field schema="faceted_search_default" name="dc_created_min" />
          <field schema="faceted_search_default" name="dc_created_max" />
        </predicate>
        <predicate parameter="dc:modified" operator="BETWEEN">
```

```
        <field schema="faceted_search_default" name="dc_modified_min" />
        <field schema="faceted_search_default" name="dc_modified_max" />
      </predicate>
      <predicate parameter="dc:creator" operator="IN">
        <field schema="faceted_search_default" name="dc_creator" />
      </predicate>
      <predicate parameter="dc:coverage" operator="STARTSWITH">
        <field schema="faceted_search_default" name="dc_coverage" />
      </predicate>
      <predicate parameter="dc:subjects" operator="STARTSWITH">
        <field schema="faceted_search_default" name="dc_subjects" />
      </predicate>
      <predicate parameter="ecm:path" operator="STARTSWITH">
        <field schema="faceted_search_default" name="ecm_path" />
      </predicate>
    </whereClause>
    <sort column="dc:title" ascending="true" />
    <pageSize>20</pageSize>
  </coreQueryPageProvider>

  <searchLayout name="faceted_search_default" />

  <useGlobalPageSize>true</useGlobalPageSize>
  <refresh>
    <event>documentChanged</event>
    <event>documentChildrenChanged</event>
  </refresh>
  <cacheKey>only_one_cache</cacheKey>
  <cacheSize>1</cacheSize>

  <resultLayouts>
    <layout name="document_virtual_navigation_listing_ajax"
      title="document_listing" translateTitle="true"
      iconPath="/icons/document_listing_icon.png" />
  </resultLayouts>

  <selectionList>CURRENT_SELECTION</selectionList>
  <actions category="CURRENT_SELECTION_LIST" />

  <flags>
    <flag>FACETED_SEARCH</flag>
  </flags>
</contentView>
```

```
</extension>
```

Schema and Document Type Contribution

As seen in the content view we just defined, we need a document type, `FacetedSearchDefault`. To be a correct document type used in a faceted search, it must extend the `FacetedSearch` document type.

According to the predicates set in the content view, we need to add a schema to the new document type to handle each predicate.

Schema definition

```
<?xml version="1.0"?>
<xs:schema targetNamespace="http://www.nuxeo.org/ecm/schemas/common/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:nxs="http://www.nuxeo.org/ecm/schemas/common/">

  <xs:include schemaLocation="base.xsd" />

  <xs:element name="ecm_fulltext" type="xs:string" />
  <xs:element name="dc_creator" type="nxs:stringList" />
  <xs:element name="dc_created_min" type="xs:date" />
  <xs:element name="dc_created_max" type="xs:date" />
  <xs:element name="dc_modified_min" type="xs:date" />
  <xs:element name="dc_modified_max" type="xs:date" />

  <xs:element name="dc_coverage" type="nxs:stringList" />
  <xs:element name="dc_subjects" type="nxs:stringList"/>

  <xs:element name="ecm_path" type="nxs:stringList"/>

</xs:schema>
```

Document Type and Schema registration

```
<extension target="org.nuxeo.ecm.core.schema.TypeService" point="schema">
  <schema name="faceted_search_default" src="schemas/faceted_search_default.xsd"
prefix="fsd"/>
</extension>

<extension target="org.nuxeo.ecm.core.schema.TypeService" point="doctype">

  <doctype name="FacetedSearchDefault" extends="FacetedSearch">
    <schema name="faceted_search_default"/>
    <facet name="HiddenInFacetedSearch"/>
  </doctype>

</extension>
```

Search Layout Contribution

The search layout is just a standard layout. It's the layout that will be used in the left tab to display all the widgets that will perform the search.

Define your widgets and map them to the right field on your newly created schema.
For instance, for a filter on the `dc:creator` property, the widget looks like:

```
<widget name="people_search" type="faceted_search_wrapper">
  <labels>
    <label mode="any">label.faceted.search.peopleSearch</label>
  </labels>
  <translated>true</translated>
  <subWidgets>
    <widget name="dc_creator" type="faceted_search_users_suggestion">
      <labels>
        <label mode="any">label.dublincore.creator</label>
      </labels>
      <fields>
        <field>fsd:dc_creator</field>
      </fields>
      <properties widgetMode="any">
        <property name="userSuggestionSearchType">USER_TYPE</property>
        <property name="displayHorizontally">false</property>
        <property name="hideSearchTypeText">true</property>
        <property name="displayHelpLabel">false</property>
      </properties>
    </widget>
  </subWidgets>
</widget>
```

Then you just need to create the layout referenced in the content view:

```
<extension target="org.nuxeo.ecm.platform.forms.layout.WebLayoutManager"
           point="layouts">

  <layout name="faceted_search_default">
    <templates>
      <template mode="any">/layouts/layout_faceted_search_template.xhtml
      </template>
    </templates>
    <rows>
      <row>
        <widget>faceted_searches_selector</widget>
      </row>
      <row>
        <widget>saved_faceted_searches_selector</widget>
      </row>
      <row>
        <widget>actions_bar</widget>
      </row>
      <row>
        <widget>text_search</widget>
      </row>
      <row>
        <widget>date_search</widget>
      </row>
      <row>
        <widget>people_search</widget>
      </row>
      <row>
        <widget>categorization_search</widget>
      </row>
      <row>
        <widget>path_search</widget>
      </row>
      <row>
        <widget>actions_bar</widget>
      </row>
    </rows>
  </layout>
</extension>
```


Do not forget to update the `searchLayout` attribute of the content view if you change the layout name.

Available Widgets Types

Here are the widgets types defined in the Faceted Search module. You can reuse them in your own faceted search contribution. You can also use all the existing widget already defined in Nuxeo.

You can have a look here to see how the widgets are used in the default faceted search.

If you depend on Nuxeo DM, you can use some widgets directly without redefining them (for instance, the ones that do not depend on a metadata property)

`faceted_search_wrapper`

This widget is used to wrap other subwidgets. It displays the widget label, and list the subwidgets below according to the wrapperMode. The subwidgets can use three wrapperMode (to be defined in the subwidget properties):

- row: the subwidget label is displayed on one row, and the subwidget content on another row.
- column: the subwidget label and content are displayed on the same row (default mode if not specified)

- noLabel: the subwidget label is not displayed at all.

For instance, here is the definition of the Text search part:

```
<widget name="text_search" type="faceted_search_wrapper">
  <labels>
    <label mode="any">label.faceted.search.textSearch</label>
  </labels>
  <translated>true</translated>
  <subWidgets>
    <widget name="ecm_fulltext" type="text">
      <labels>
        <label mode="any">label.faceted.search.fulltext</label>
      </labels>
      <translated>true</translated>
      <fields>
        <field>fsd:ecm_fulltext</field>
      </fields>
      <properties widgetMode="edit">
        <property name="wrapperMode">row</property>
      </properties>
    </widget>
  </subWidgets>
</widget>
```

`date_range`

Widget used to search on a date range.

```
<widget name="dc_modified" type="date_range">
  <labels>
    <label mode="any">label.dublincore.modificationDate</label>
  </labels>
  <translated>true</translated>
  <fields>
    <field>fsd:dc_modified_min</field>
    <field>fsd:dc_modified_max</field>
  </fields>
  <properties widgetMode="edit">
    <property name="styleClass">dataInputTextDate</property>
  </properties>
</widget>
```

`faceted_search_users_suggestion`

Widget allowing to search and select one or more users with a suggestion box.

```
<widget name="dc_creator" type="faceted_search_users_suggestion">
  <labels>
    <label mode="any">label.dublincore.creator</label>
  </labels>
  <fields>
    <field>fsd:dc_creator</field>
  </fields>
  <properties widgetMode="any">
    <property name="userSuggestionSearchType">USER_TYPE</property>
    <property name="displayHorizontally">false</property>
    <property name="hideSearchTypeText">true</property>
    <property name="displayHelpLabel">false</property>
  </properties>
</widget>
```

`faceted_searches_selector`

Widget displaying all the registered faceted searches. Hidden in case only one faceted search is registered.

```
<widget name="faceted_searches_selector"
  type="faceted_searches_selector">
  <widgetModes>
    <!-- not shown in edit and view modes -->
    <mode value="view">hidden</mode>
    <mode value="edit">hidden</mode>
  </widgetModes>
</widget>
```

In this sample, the widget is hidden in view and edit mode, so that the widget is not displayed when you are on the Summary or Edit tab of a saved search.

`saved_faceted_searches_selector`

Widget displaying all the saved faceted searches. It displays two categories:

- Your searches: your saved faceted searches.
- All searches: all the other users shared saved faceted searches.

The `outcome` property needs to be defined: on which JSF view should we redirect after selecting a saved search.

```
<widget name="saved_faceted_searches_selector"
        type="saved_faceted_searches_selector">
  <widgetModes>
    <!-- not shown in edit and view modes -->
    <mode value="view">hidden</mode>
    <mode value="edit">hidden</mode>
  </widgetModes>
  <properties widgetMode="any">
    <property name="outcome">faceted_search_results</property>
  </properties>
</widget>
```

`faceted_search_directory_tree`

Widget allowing to select one or more values from a tree constructed from the directory tree specified in the `directoryTreeName` property.

```
<widget name="dc_coverage" type="faceted_search_directory_tree">
  <labels>
    <label mode="any">label.faceted.search.coverage</label>
  </labels>
  <translated>true</translated>
  <fields>
    <field>fsd:dc_coverage</field>
  </fields>
  <properties widgetMode="any">
    <property name="directoryTreeName">byCoverageNavigation</property>
    <property name="wrapperMode">noLabel</property>
  </properties>
</widget>
```

`faceted_search_path_tree`

Widget allowing to select one or more values from a tree constructed from the navigation tree.

```
<widget name="ecm_path" type="faceted_search_path_tree">
  <labels>
    <label mode="any">label.faceted.search.path</label>
  </labels>
  <translated>true</translated>
  <fields>
    <field>fsd:ecm_path</field>
  </fields>
  <properties widgetMode="any">
    <property name="wrapperMode">noLabel</property>
  </properties>
</widget>
```

`actions_bar`

This widget is only defined in the `nuxeo-platform-faceted-search-dm` module.

```
<widget name="actions_bar" type="template">
  <properties widgetMode="any">
    <property name="template">
      /widgets/faceted_search_actions_widget_template.xhtml
    </property>
  </properties>
  <widgetModes>
    <!-- not shown in edit and view modes -->
    <mode value="view">hidden</mode>
    <mode value="edit">hidden</mode>
  </widgetModes>
</widget>
```

You can use directly the widget in your custom search layout:

```
<extension target="org.nuxeo.ecm.platform.forms.layout.WebLayoutManager"
  point="layouts">
  <layout name  ="faceted_search_default">
    <templates>
      <template mode="any">/layouts/layout_faceted_search_template.xhtml
      </template>
    </templates>
    <rows>
      ...
      <row>
        <widget>actions_bar</widget>
      </row>
      ...
    </rows>
  </layout>
</extension>
```

You probably want at least one action bar in your layout to perform the search!

Related pages in this documentation

- Layouts and Widgets (Forms, Listings, Grids)

Related pages in other documentations

- Faceted Search

Actions (Links, Buttons, Icons, Tabs and More)

Actions usually stand for commands that can be triggered via user interface interaction (buttons, links, etc...).

Usually, it will describe a link and other information that may be used to manage its display (the link label and an icon for instance).

By extension, actions are also used for conditional rendering and sorting of page fragments (tabs, summary widgets,...).

This chapter explains how to define actions and display them in pages.

- Actions Overview — In this chapter, an action will stand for any kind of command that can be triggered via user interface interaction. In other words, it will describe a link and other information that may be used to manage its display (the link label, an icon, security information for instance).
- Standard Action Types — A series of action types is available for the most basic uses cases.
- Custom Action Types — Since 5.8, it is easy to add your own action type to handle its configuration and display, rather than defining an action of type "template" and specifying the template each time it needs to be used.
- Filters and Access Controls — Filters configuration allows to control activation of an action, to control its visibility depending on the user rights, for instance, or selected documents, etc.
- Actions Display — Actions are grouped in categories to be able to display them in the same area of a page. Widgets can be used to handle rendering of these actions.
- Incremental Layouts and Actions — Actions are leveraged by the layout framework to include widgets inside layouts dynamically, benefiting from sorting and filtering features of actions within layouts.

Actions Overview

In this chapter, an action will stand for any kind of command that can be triggered via user interface interaction. In other words, it will describe a link and other information that may be used to manage its display (the link label, an icon, security information for instance).

Registering a New Action

Custom actions can be contributed to the actions service, using its extension point. Their description is then available through this service to control where and how they will be displayed.

An action can be registered using the following example extension:

In this section


```
<extension target="org.nuxeo.ecm.platform.actions.ActionService" point="actions">

  <action id="newWorkspace" label="command.create.workspace"
 link="#{documentActions.createDocument('Workspace')}"
    type="link" icon="/icons/action_add.gif">
    <category>SUBVIEW_UPPER_LIST</category>
    <filter id="newWorkspace">
      <rule grant="true">
        <permission>AddChildren</permission>
        <type>WorkspaceRoot</type>
      </rule>
    </filter>
  </action>

</extension>
```

The above action will be used to display the creation page for a Workspace document type. Here are its properties:

- `id`: the string identifying the action. In the example, the action id is `newWorkspace`. The same identifier can be reused to override (or merge) the original action configuration.
- `label`: the action name that will be used when displaying the link. In the example, the label is `command.create.workspace`. This label is a message that will always be translated at display (there is no boolean property stating if the action label should be translated like on widget definitions).
- `link`: the string representing the command the action will trigger. This string may have a different form and syntax depending on the action type. In the example, a JSF command link will be used, so it represents an action method expression. The Seam component called `documentActions` holds a method named `createDocument` that will perform the navigation to the creation page.
- `category`: a string useful to group actions that will be rendered in the same area of a page. An action can define several categories. Here, the only category defined is `SUBVIEW_UPPER_LIST`. It is designed to group all the actions that will be displayed on the right top corner of any page of the site. Some default categories are available on Studio documentation at User actions categories.
- `type` (available since 5.6): an optional typing of the action, so that actions needing different kinds of rendering can be mixed up in the same category (see chapter Adapt templates to display an action). When the type is not defined, a default type can be deduced from the action category (for compatibility), or else the type "link" is used.

Other elements can be used to define an action. They are listed here but you can have a look at the main actions contribution files for more examples, like `nuxeo-platform-webapp-core/src/main/resources/OSGI-INF/actions-contrib.xml` or `nuxeo-platform-webapp-base/src/main/resources/OSGI-INF/actions-contrib.xml`.

- `filter-ids`: the id of a filter that will be used to control the action visibility. An action can have several filters: it is visible if all of its filters grant the access (see chapter about Filters and Access Controls).
- `filter`: a filter definition can be done directly within the action definition. It is a filter like others and can be referred by other actions. This way of defining filters is here for compatibility, defining filters on the filters extension points is recommended.
- `icon`: the optional icon path for this action.
- `confirm`: an optional JavaScript confirmation string that can be triggered when executing the command.
- `order`: an optional integer used to sort actions within the same category.
- `enabled`: a boolean indicating whether the action is currently active. This can be used to disable/hide existing actions when customizing the application.
- `immediate`: an optional boolean to execute action immediately, without validating the form.
- accessKey (available since 5.6): an optional key that can be used for keyboard navigation.
- `properties` (available since 5.6): a tag that allows to attach any kind of named string, list or map-like property to the action. Since 5.8, most of the above elements are looked up in properties (label, icon, link, immediate, accessKey). Depending on the type of the action, some custom properties may be available. Properties usually accept EL expressions for dynamic resolution of values. Please refer to the documentation at http://showcase.nuxeo.com/nuxeo/layoutDemo/linkAction for instance.

Here is a small sample to add a custom confirmation message on an action:

```
<extension point="actions" target="org.nuxeo.ecm.platform.actions.ActionService">

  <action id="JenkinsReportSendMail" label="Send Mail" order="0" type="link"
    icon="/img/jenkins_send_email.png"
    link="#{operationActionBean.doOperation('JenkinsReportSendMail')}">
    <category>DOCUMENT_UPPER_ACTION</category>
    <properties>
      <property name="confirmMessage">label.jenkins.sendMail.confirm</property>
      <propertyList name="confirmMessageArgs">

<value>#{docSuggestionActions.getDocumentWithId(currentSuperSpace.id).getPropertyVal
ue('jenkinsreports:report_email')}</value>
      </propertyList>
    </properties>
  </action>

</extension>
```

The confirmation label `label.jenkins.sendMail.confirm` is holding a variable parameter:

```
label.jenkins.sendMail.confirm=This will send an email to {0}, are you sure that you
would like to continue?
```

This definition makes it possible to show the following confirmation message at runtime: "This will send an email to myemail@example.com, are you sure that you would like to continue?".

Of course the confirmation message does not need to hold arguments.

Redefining an Action

Actions extension point provides merging features: you can change an existing action definition in your custom extension point provided you use the same identifier. Properties holding single values (label, link for instance) will be replaced using the new value. Properties holding multiple values (categories, filters) will be merged with existing values. Properties will be merged if they hold the attribute `append="true"`.

Here are some actions override examples:

```
<extension target="org.nuxeo.ecm.platform.actions.ActionService" point="actions">

  <action id="newWorkspace" enabled="false" />

  <action id="newWorkspace" icon="/icons/my_icon.png" />

</extension>
```

Examples

Here are more examples showing how to use actions in the default application.

A Tab Supporting Key Access and Ajax

Tab items on document views, for instance, are using actions to benefit from sorting and filtering features. The tab link is a RESTful link to the current document, with additional request parameters to show this tab as selected.

So tab actions are using the type "Rest document link" and state in the "link" property the template to include for the tab content:

```
<extension target="org.nuxeo.ecm.platform.actions.ActionService" point="actions">

  <action id="TAB_EDIT" link="/incl/tabs/document_edit.xhtml" order="20"
    label="action.view.modification" icon="/icons/file.gif" accessKey="e"
    type="rest_document_link">
    <category>VIEW_ACTION_LIST</category>
    <filter-id>edit</filter-id>
    <filter-id>mutable_document</filter-id>
    <properties>
      <property name="ajaxSupport">true</property>
    </properties>
  </action>

</extension>
```

This action is showing the "Edit" tab on documents. It displays the content of the "/incl/tabs/document_edit.xhtml" template, defines an access key, and specifies that it supports Ajax (in case action is displayed in an ajaxified context).

Displaying a Permalink inside a Fancybox

Here is another sample that displays the document permanent link inside a FancyBox:

```
<extension target="org.nuxeo.ecm.platform.actions.ActionService" point="actions">

  <action id="permalinkAction" order="20" label="label.permalink" type="fancybox"
    icon="/icons/contextual_menu/share.png" accessKey="k">
    <category>DOCUMENT_UPPER_ACTION</category>
    <properties>
      <property name="include">/incl/permalink_box.xhtml</property>
      <property name="ajaxSupport">true</property>
    </properties>
  </action>

</extension>
```

The FancyBox content is defined by "/incl/permalink_box.xhtml". Note that if the fancybox content holds a form, you must make sure that the action is not itself already placed within a bigger form (as nested forms are not supported).

Related sections

- Actions Display
- Standard Action Types
- Custom Action Types

Standard Action Types

A series of action types is available for the most basic uses cases.

link

Link actions display a command link, and can be ajaxified or not.

View online reference: http://showcase.nuxeo.com/nuxeo/layoutDemo/linkAction

bare_link

Bare link actions display links to external URLs.

View online reference: http://showcase.nuxeo.com/nuxeo/layoutDemo/bare_linkAction

In this section

fancybox

Fancybox actions open a FancyBox (aka modal panel) when the button is clicked. The fancybox content can either be a XHTML template fragment, or a complete iframe.

Note that fancybox actions cannot currently be used as form actions (e.g present forms that can be submitted inside another form) as nested sub-forms are not supported.

When referencing a XHTML template via the `include` property, if this template holds a form, it should be using the variable `fancyboxFormId` as its form id for the fancybox to be reopened on validation errors.

The bulk edit action can be taken as an example, see the action "CURRENT_SELECTION_EDIT" registration and referenced template.

View online reference: http://showcase.nuxeo.com/nuxeo/layoutDemo/fancyboxAction

rest_document_link

Rest document link actions display a link to a given document view. These actions are used to display document tabs, as they can reference a tab to be marked as currently selected.

View online reference: http://showcase.nuxeo.com/nuxeo/layoutDemo/rest_document_linkAction

main_tab

Main tab actions display a link to a given module/application, for instance "Home", "Document Management" or "Admin Center".

These actions are not really tabs, because the content of the "tab" depends on the view, not on the corresponding action configuration. They use a specific action type to resolve specific Navigation URLs and current document restoration depending on the module/application.

View online reference: http://showcase.nuxeo.com/nuxeo/layoutDemo/main_tabAction

widget

Widget actions are mainly useful when building incremental layouts: they allow to display a given widget (on a summary page for instance) depending on action filters and orders configuration (for a given category of actions).

View online reference: http://showcase.nuxeo.com/nuxeo/layoutDemo/widgetAction

template

Template actions are useful to display a custom content freely. For instance, they're used to display the search box on the top right corner.

View online reference: http://showcase.nuxeo.com/nuxeo/layoutDemo/templateAction

Related pages in this documentation

- Actions Overview
- Custom Action Types

Related pages in Studio documentation

- User Actions
- User actions categories

Custom Action Types

Since 5.8, it is easy to add your own action type to handle its configuration and display, rather than defining an action of type "template" and specifying the template each time it needs to be used.
You can take example on the "link" action type registration for this. It is actually relying on widget types definition, here is a minimal definition

for a custom action type:

```
<extension target="org.nuxeo.ecm.platform.forms.layout.LayoutStore"
  point="widgettypes">

  <widgetType name="my_link">
    <categories>
      <category>jsfAction</category>
    </categories>
    <configuration>
      <properties>
        <defaultValues mode="any">
          <property name="discardSurroundingForm">false</property>
          <property name="supportAjax">#{canUseAjaxTabs}</property>
        </defaultValues>
      </properties>
    </configuration>
    <handler-class>
      org.nuxeo.ecm.platform.forms.layout.facelets.plugins.TemplateWidgetTypeHandler
    </handler-class>
    <property name="template">
      /incl/action/my_link_action_template.xhtml
    </property>
  </widgetType>

</extension>
```

The category `jsfAction` is used to filter this widget type from other widget types and make this one specific to actions display.

The configuration can hold default values to control default behavior:

- `discardSurroundingForm`: makes it possible to avoid adding a form around an action that does not need it anyway (like a RESTful link).
- `supportAjax`: makes it possible to avoid stating this property in all actions of this type (if Ajax is always supported, for instance).

The action template at "/incl/action/my_link_action_template.xhtml" may need to handle two modes specific to actions:

- `tab_content`: used to define what should be displayed in the content of a tab for this action type.
- `after_view`: used to define what should be displayed after the main rendering (e.g the action button), because that's useful to include content after the main form displaying an action bar, for instance. This mode is currently useful when defining fancyboxes that include a form.

Default action templates can also be browsed for examples.

Related pages in this documentation

- Standard Action Types
- Actions Overview
- Actions and Filters How-tos

Related pages in Studio documentation

- User Actions
- User actions categories

Filters and Access Controls

Filters configuration allows to control activation of an action, to control its visibility depending on the user rights, for instance, or selected documents, etc.

Managing Filters to Control an Action Visibility

An action visibility can be controlled using filters. An action filter is a set of rules that will apply - or not - given an action and a context.

Filters can be registered using their own extension point, or registered implicitly when defining them inside of an action definition.

In this section

Example of a filter registration:

```
<filter id="view_content">
  <rule grant="true">
    <permission>ReadChildren</permission>
    <facet>Folderish</facet>
  </rule>
  <rule grant="false">
    <type>Root</type>
  </rule>
</filter>
```

Example of a filter registration inside an action registration:

```
<action id="newSection" link="#{documentActions.createDocument('Section')}"
    enabled="true" label="command.create.section"
    icon="/icons/action_add.gif">
  <category>SUBVIEW_UPPER_LIST</category>
  <filter id="newSection">
    <rule grant="true">
      <permission>AddChildren</permission>
      <type>SectionRoot</type>
    </rule>
  </filter>
</action>
```

A filter can accept any number of rules. It will grant access to an action if, among its rules, no denying rule (`grant=false`) is found and at least one granting rule (`grant=true`) is found. A general rule to remember is that if you would like to add a filter to an action that already has one or more filters, it has to hold constraining rules: a granting filter will be ignored if another filter is already too constraining.

If no granting rule (`grant=true`) is found, the filter will grant access if no denying rule is found. If no rule is set, the filter will grant access by default.

The default filter implementation uses filter rules with the following properties:

- `grant`: a boolean indicating whether this is a granting rule or a denying rule.
- `permission`: a permission like "Write" that will be checked on the context for the given user. A rule can hold several permissions: it applies if user holds at least one of them.
- `facet`: a facet like "Folderish" that can be set on the document type (`org.nuxeo.ecm.core.schema.types.Type`) to describe the document type general behavior. A rule can hold several facets: it applies if current document in context has at least one of them.
- `condition`: an EL expression that can be evaluated against the context. The Seam context is made available for conditions evaluation. A rule can hold several conditions: it applies if at least one of the conditions is verified.
- `type`: a document type to check against current document in context. A rule can hold several types: it applies if current document is one of them. The fake `Server` type is used to check the server context.
- `schema`: a document schema to check against current document in context. A rule can hold several schemas: it applies if current document has one of them.
- `group`: a group like "members" to check against current user in context. A rule can hold several groups: it applies if current user is in one of them.

Filters do not support merging, so if you define a filter with an id that is already used in another contribution, only the first contribution will be

taken into account.

EL Expressions and Available Context Variables

When defining an action which visibility that depends on the current document, the `currentDocument` variable can be used inside a `condition` element.

The `currentUser` variable is also available, resolving to a NuxeoPrincipal instance, and can also be used inside a `condition` element.

All other variables available in the Seam context can also be used (Seam components for instance).

In some specific use cases, some additional variables are available, for instance when displaying actions on content view results. These actions are still displayed, but disabled, when filters resolve to false: often the filter will check if there are selected documents. These filters can use the custom variable `selectedDocuments`, representing the list of entries that have been selected, or `contentView`, representing the corresponding content view POJO.

Using Filters without Actions

Since 5.6, filters resolution can be done independently from actions. This makes it possible to benefit from filters configurability and override/merging features.

For instance, `directoriesManagementAccess` makes it possible to control access to a given directory:

```
<extension target="org.nuxeo.ecm.platform.actions.ActionService"
  point="filters">
  <filter id="directoriesManagementAccess">
    <rule grant="true">
      <condition>currentUser.administrator</condition>
      <condition>currentUser.isMemberOf('powerusers')</condition>
    </rule>
  </filter>
</extension>
```

This can be looked up in the code, by calling the action service, and evaluating the filter given an evaluation context:

```
ActionManager actionManager = Framework.getLocalService(ActionManager.class);
return actionManager.checkFilter("directoriesManagementAccess",
createActionContext(ctx));
```

This also can be looked up in XHTML templates, calling Seam component webActions and using its default evaluation context:

```
<c:if test="#{webActions.checkFilter('directoriesManagementAccess')}">
  [...]
</c:if>
```

Related pages in this documentation

- Field Binding and Expressions

Related pages in Studio documentation

- Filtering Options Reference Page

Actions Display

Actions are grouped in categories to be able to display them in the same area of a page. Widgets can be used to handle rendering of these actions.

Actions are referencing the same category when they need to be displayed in the same area of a page.

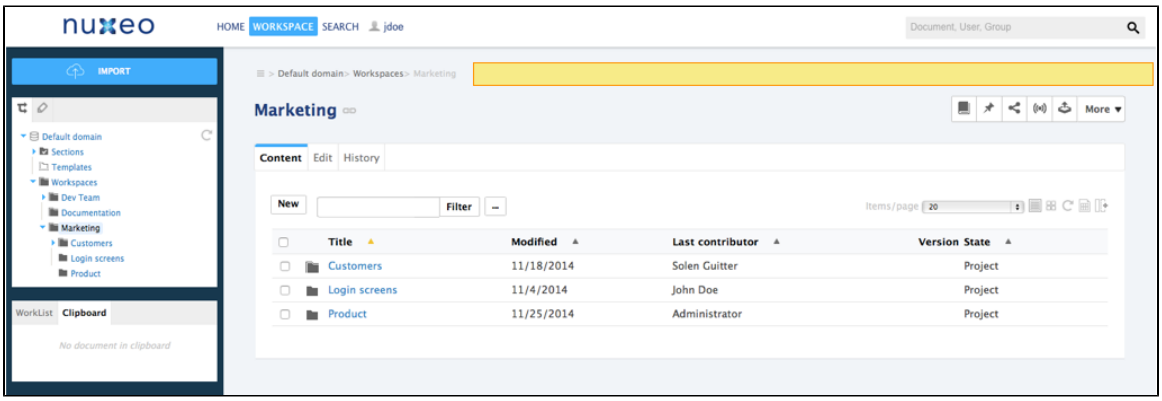
The available categories are listed below and can also be found by checking action contributions on the explorer.

CAP Categories

Breadcrumbs

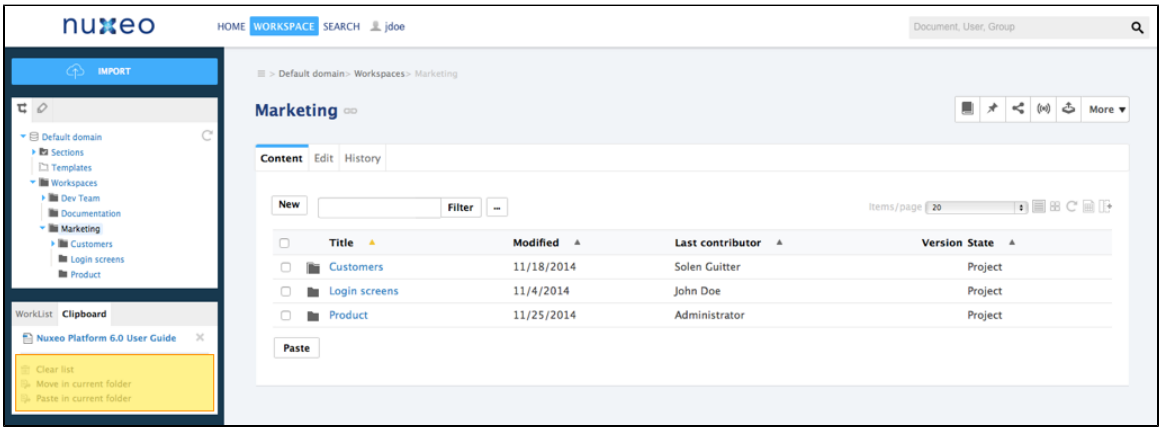
Technical name: `BREADCRUMBS_ACTIONS`

Available since Nuxeo Platform 5.9.2.



Clipboard

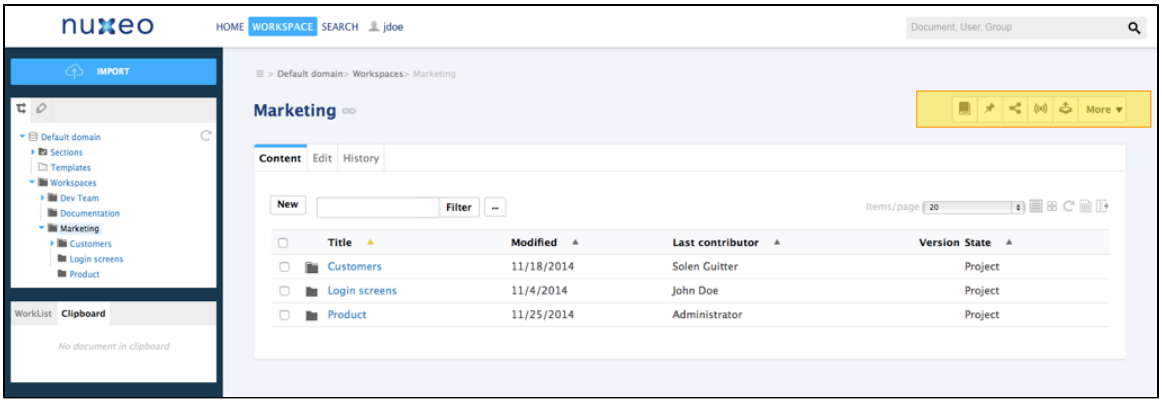
Technical name: `CLIPBOARD_LIST`



Contextual Tools

Technical name: `DOCUMENT_UPPER_ACTION`

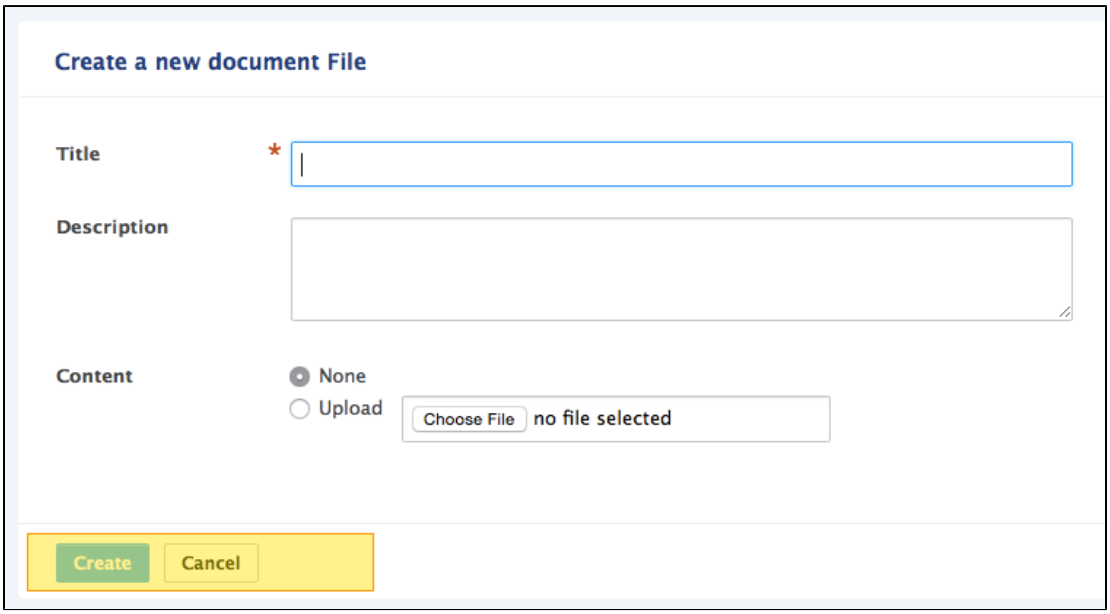
Unless you specify an icon, the following icon will be used: 



Document Creation Form

Technical name: `CREATE_DOCUMENT_FORM`

Available since Nuxeo Platform 5.4.2.

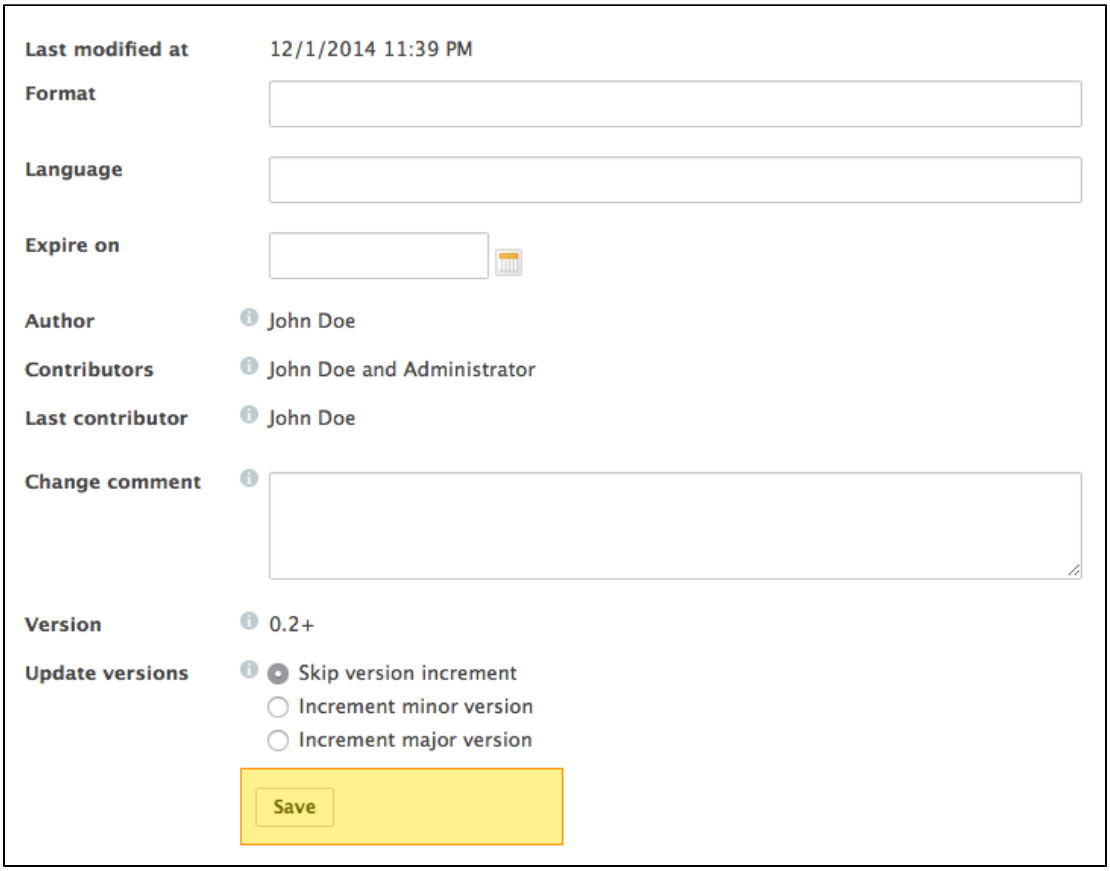


In this section

Document Edition Form

Technical name: `EDIT_DOCUMENT_FORM`

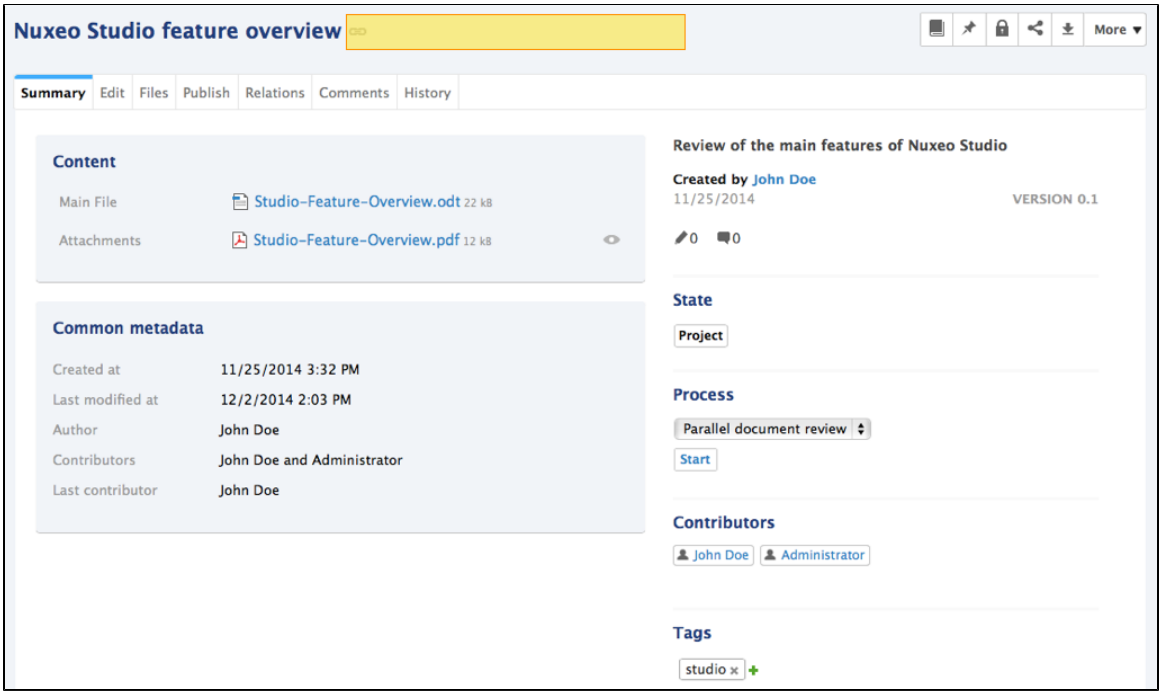
Available since Nuxeo Platform 5.4.2.



Document Header Actions

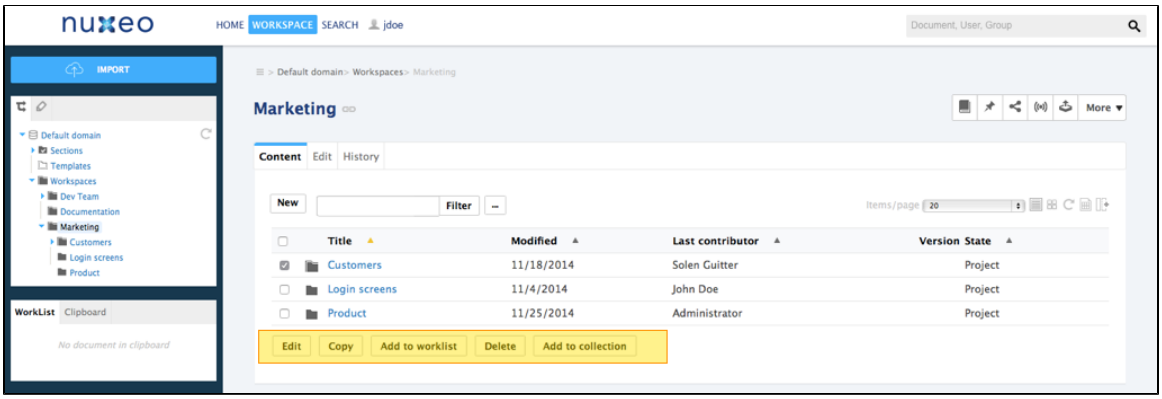
Technical name: `DOCUMENT_HEADER_ACTIONS_LIST`

Available since Nuxeo Platform 5.4.2. This action category requires to use an icon, otherwise it won't be displayed on the UI.



Document List Toolbar

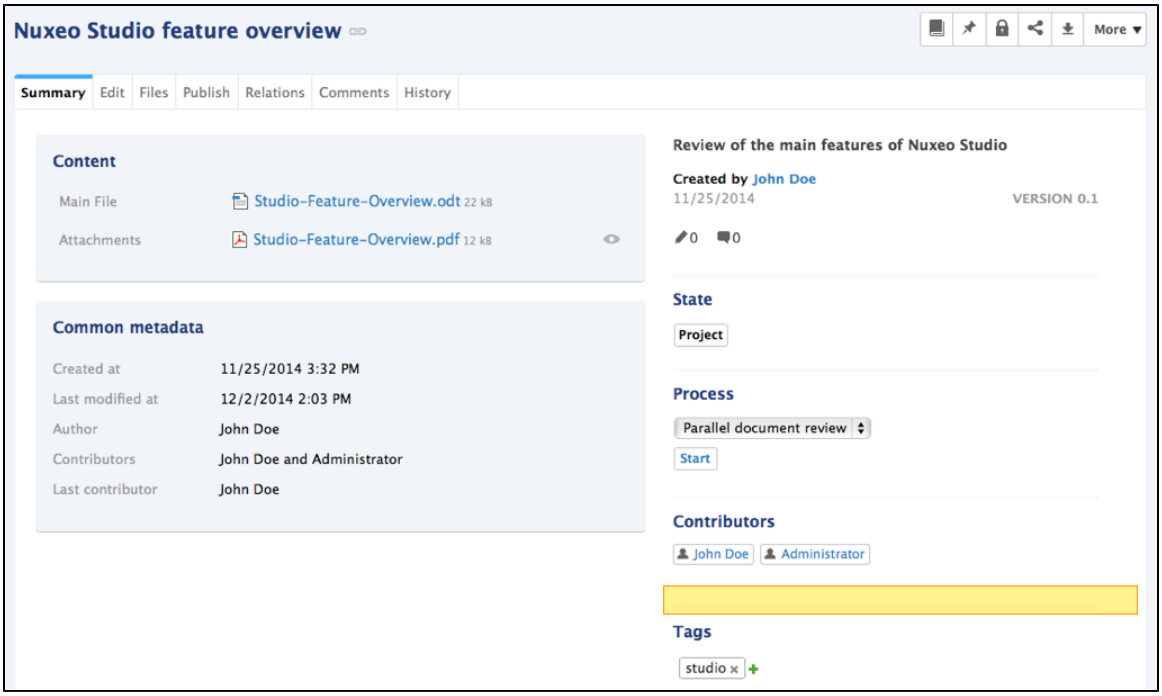
Technical name: `CURRENT_SELECTION_LIST`



Document Summary Actions

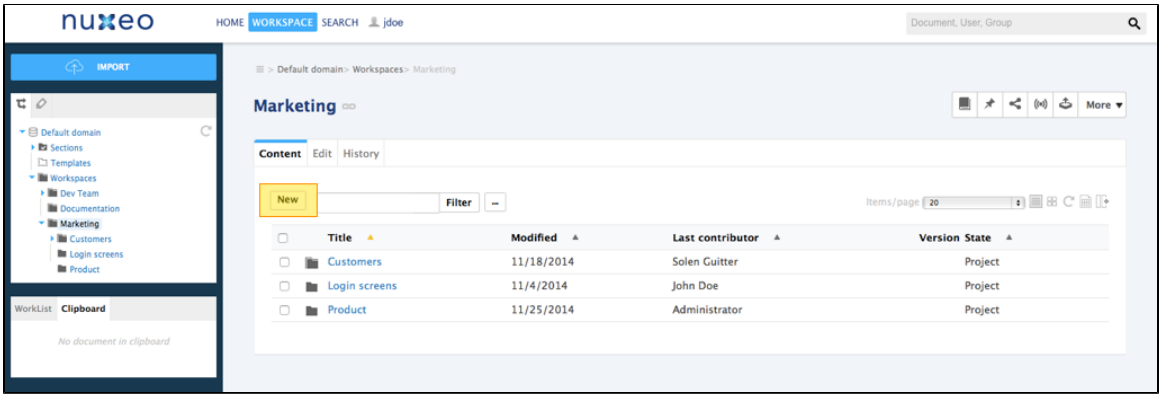
Technical name: `DOCUMENT_SUMMARY_CUSTOM_ACTIONS`

Available since Nuxeo Platform 5.4.2.



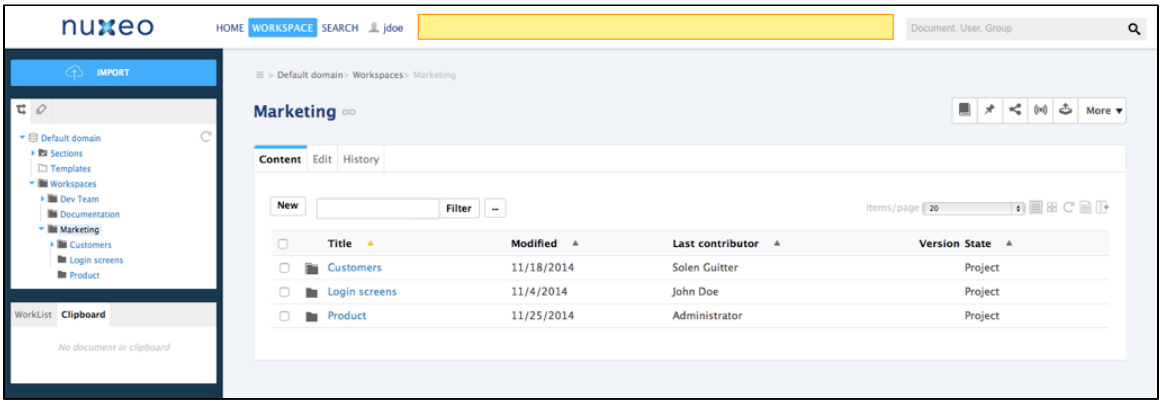
Folder Toolbar

Technical name: `SUBVIEW_UPPER_LIST`



Header Links

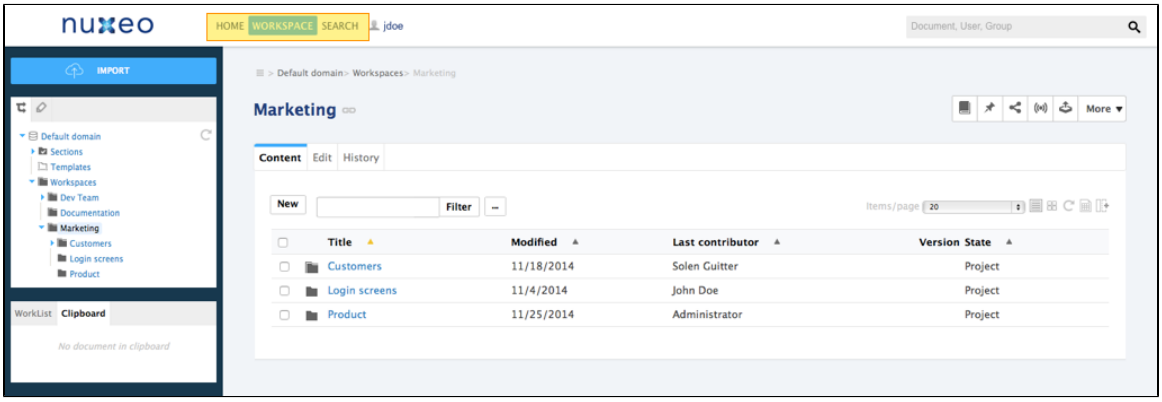
Technical name: `USER_SERVICES`



Header Main Tabs

Technical name: `MAIN_TABS`

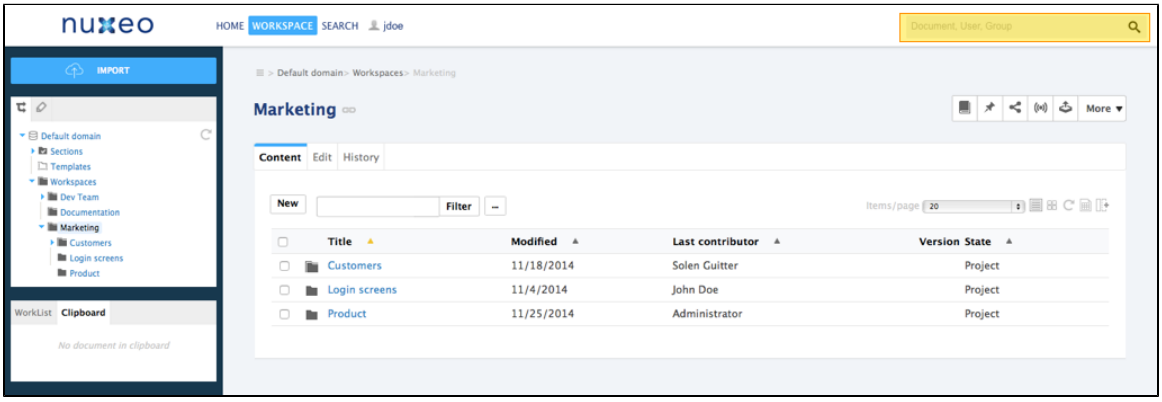
Available since Nuxeo Platform 5.8.



Header Search Actions

Technical name: `SEARCH ACTIONS`

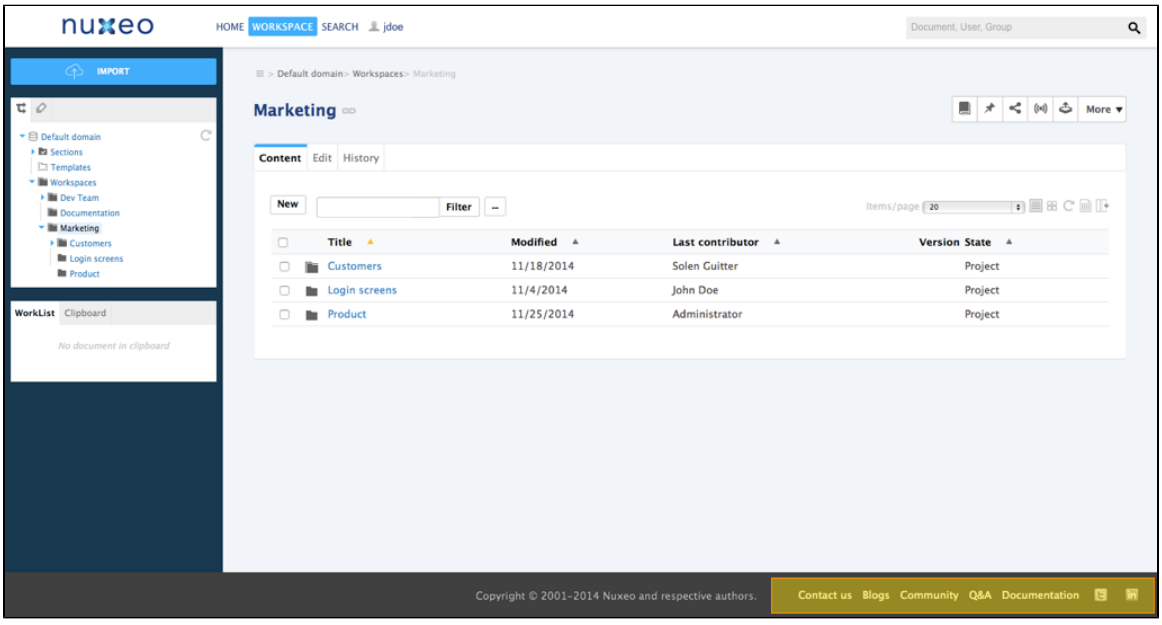
Available since Nuxeo Platform 5.8.



Footer Links

Technical name: `FOOTER`

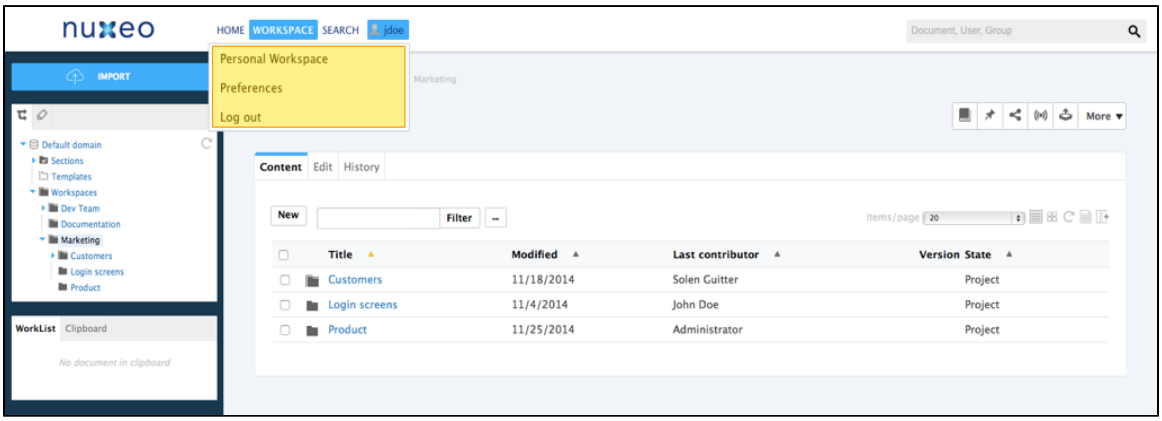
Available since Nuxeo Platform 5.8.



User Menu Actions

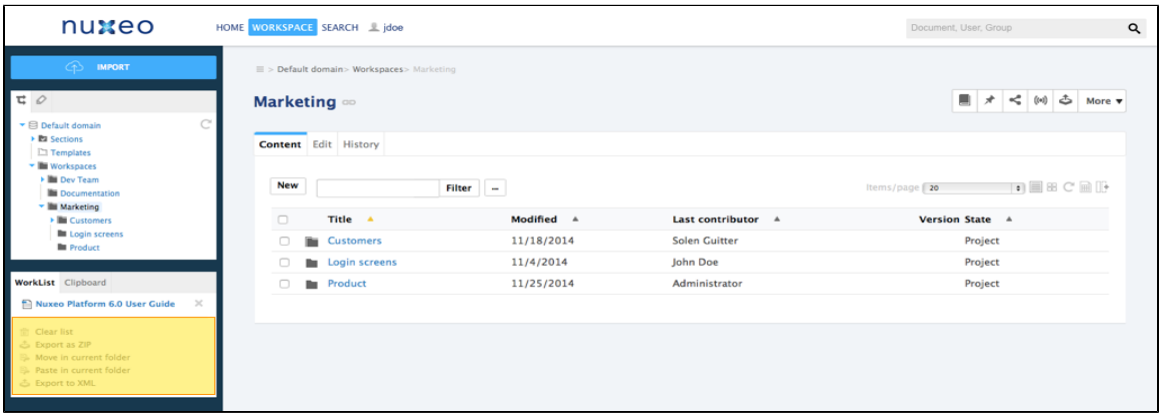
Technical name: `USER_MENU_ACTIONS`

Available since Nuxeo Platform 5.4.2.



Worklist

Technical name: `DEFAULT_LIST`



CAP Advanced Categories

These categories can be useful when defining custom actions, that will reference the widgets to display. This is useful when building incremental layouts, like the default summary layout starting from 5.6: the action order and filter information are useful to contribute/display/hide some widgets to the summary default layout.

These categories are not really useful when defining user actions, and the associated features can be broken when migrating from 5.6 to 5.8, as the form around the summary layout has been removed for 5.8 to allow fine-grained form management on this page.

View Action List

Technical name: `VIEW_ACTION_LIST`.

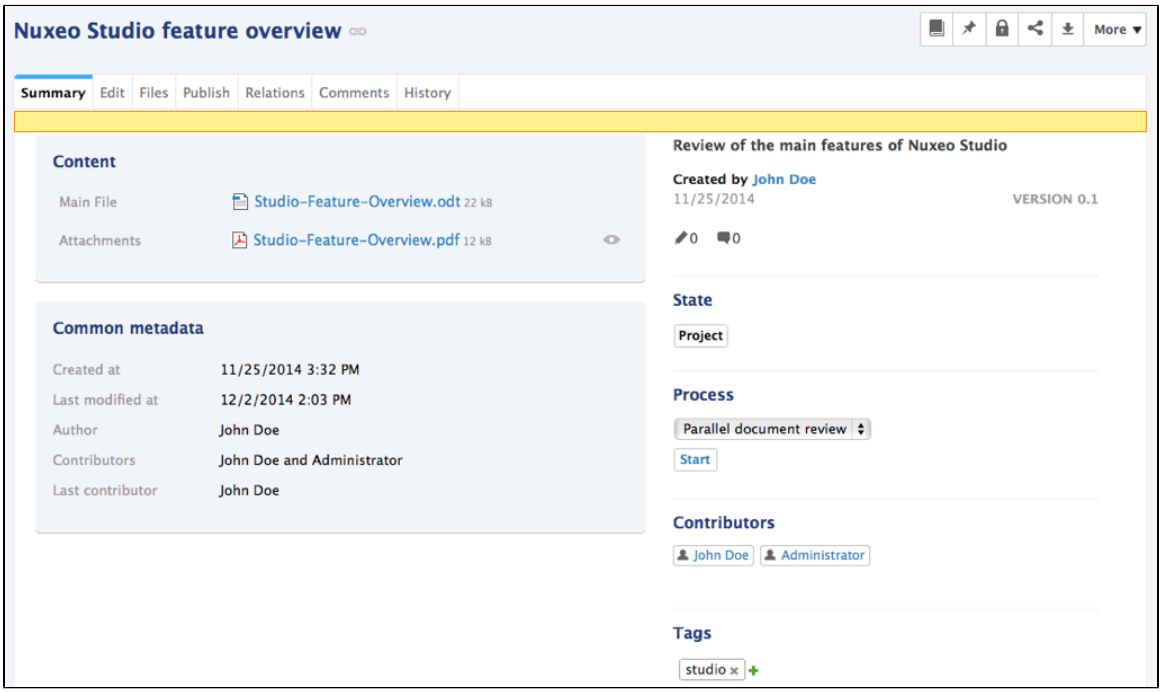
This categories is used for tabs displayed on every document.

```
<action id="TAB_VIEW" link="/incl/tabs/document_view.xhtml" enabled="true"
  order="0" label="action.view.summary" type="rest_document_link">
  <category>VIEW_ACTION_LIST</category>
  <filter-id>view</filter-id>
</action>

<action id="TAB_CONTENT" link="/incl/tabs/document_content.xhtml" order="10"
  enabled="true" label="action.view.content" type="rest_document_link">
  <category>VIEW_ACTION_LIST</category>
  <filter-id>view_content</filter-id>
</action>
```

Summary Top Panel

Technical name: `SUMMARY_PANEL_TOP`

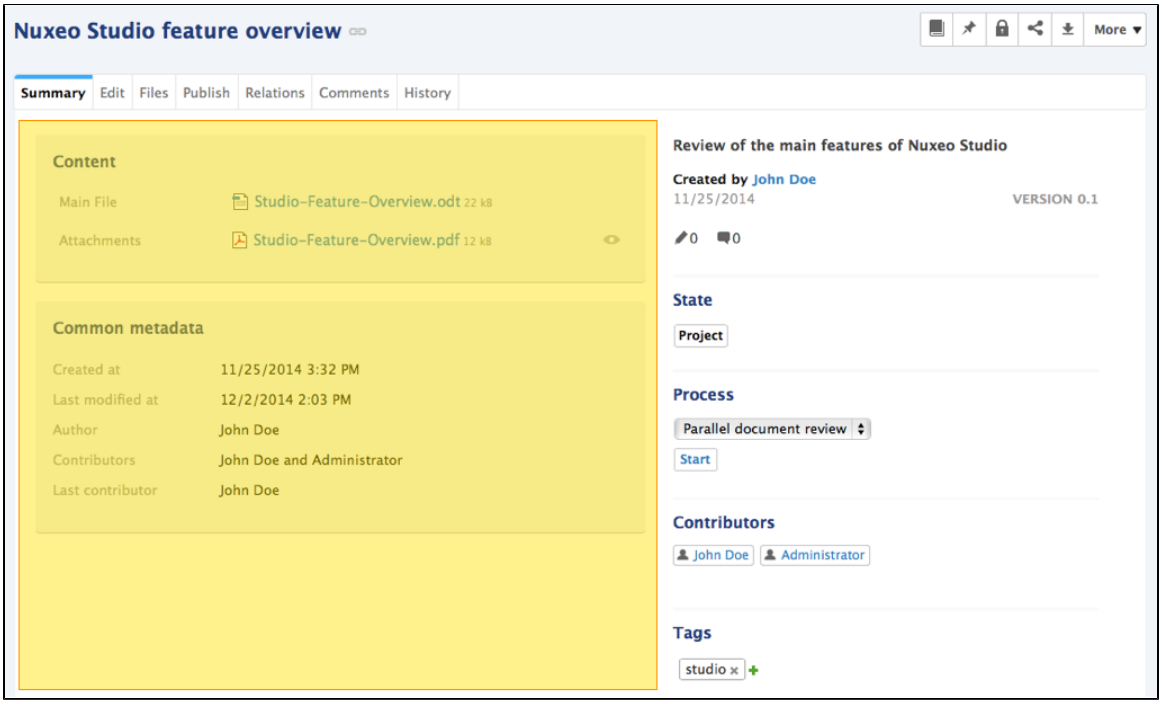
This user action category is not yet fully implemented in Studio (available since Nuxeo Platform 5.6).



Summary Left Panel

Technical name: `SUMMARY_PANEL_LEFT`

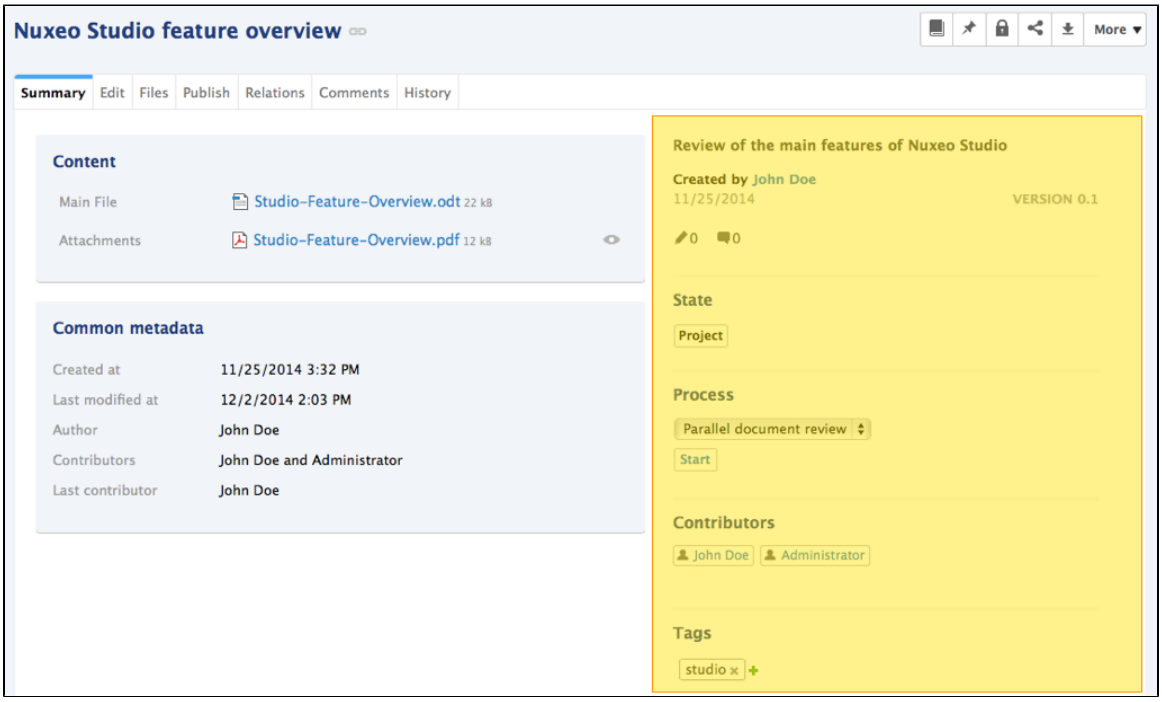
This user action category is not yet fully implemented in Studio (available since Nuxeo Platform 5.6).



Summary Right Panel

Technical name: `SUMMARY_PANEL_RIGHT`

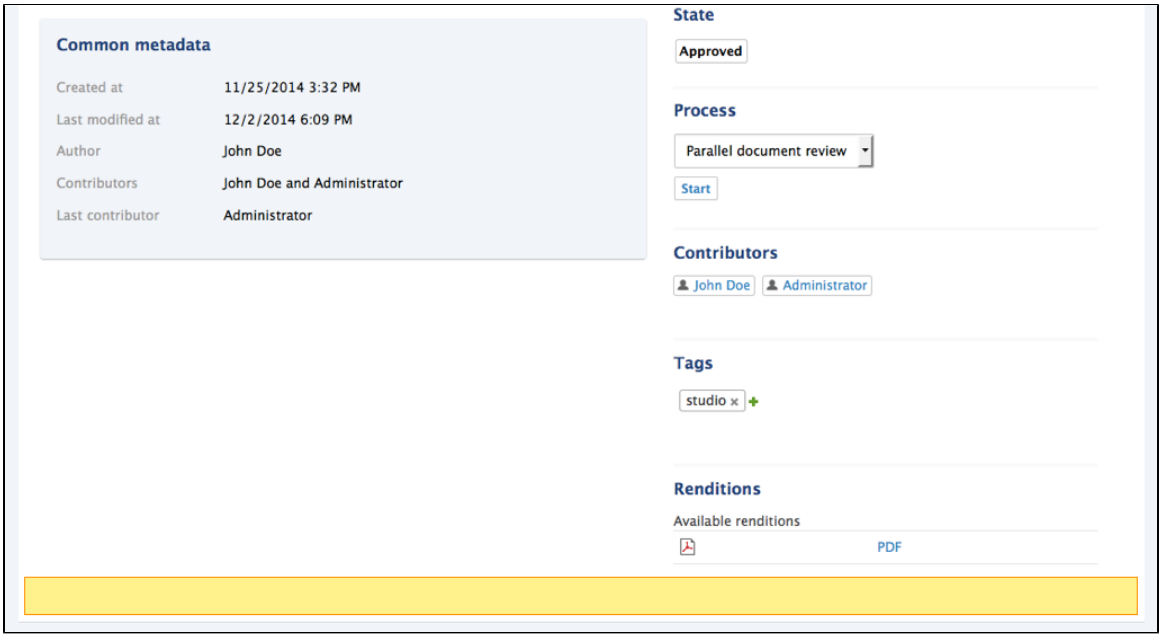
This user action category is not yet fully implemented in Studio (available since Nuxeo Platform 5.6).



Summary Bottom Panel

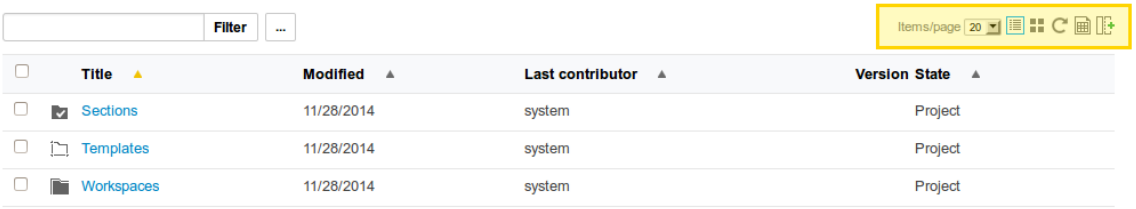
Technical name: `SUMMARY_PANEL_BOTTOM`

This user action category is not yet fully implemented in Studio (available since Nuxeo Platform 5.6).



Content View Actions

Technical name: `CONTENT_VIEW_ACTIONS`



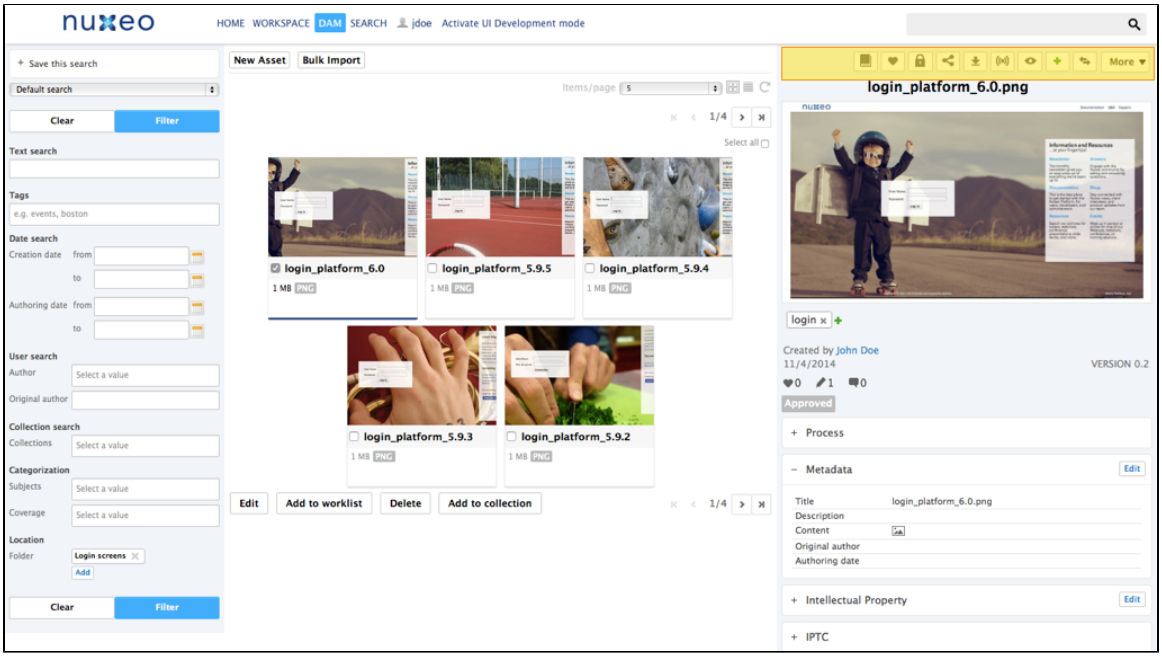
DAM Categories

DAM Asset view actions

Technical name: `DAM_ASSET_VIEW_ACTIONS`

Available since Nuxeo Platform 5.7.1.

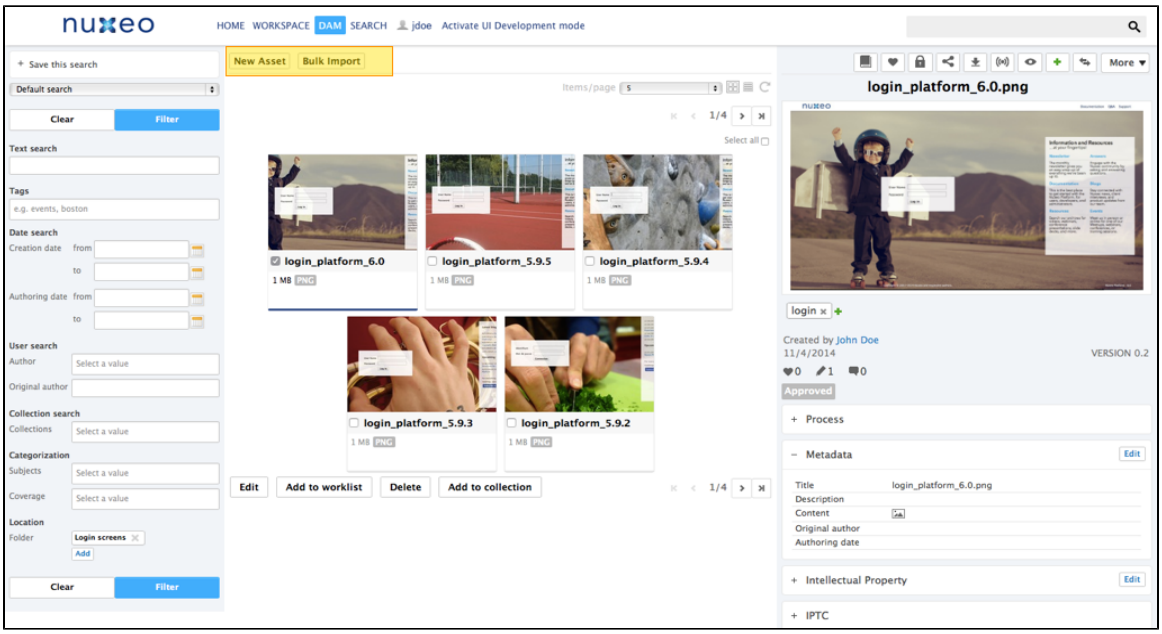
Unless you specify an icon, the following icon will be used: 



DAM Search results actions

Technical name: `DAM_SEARCH_RESULTS_ACTIONS`

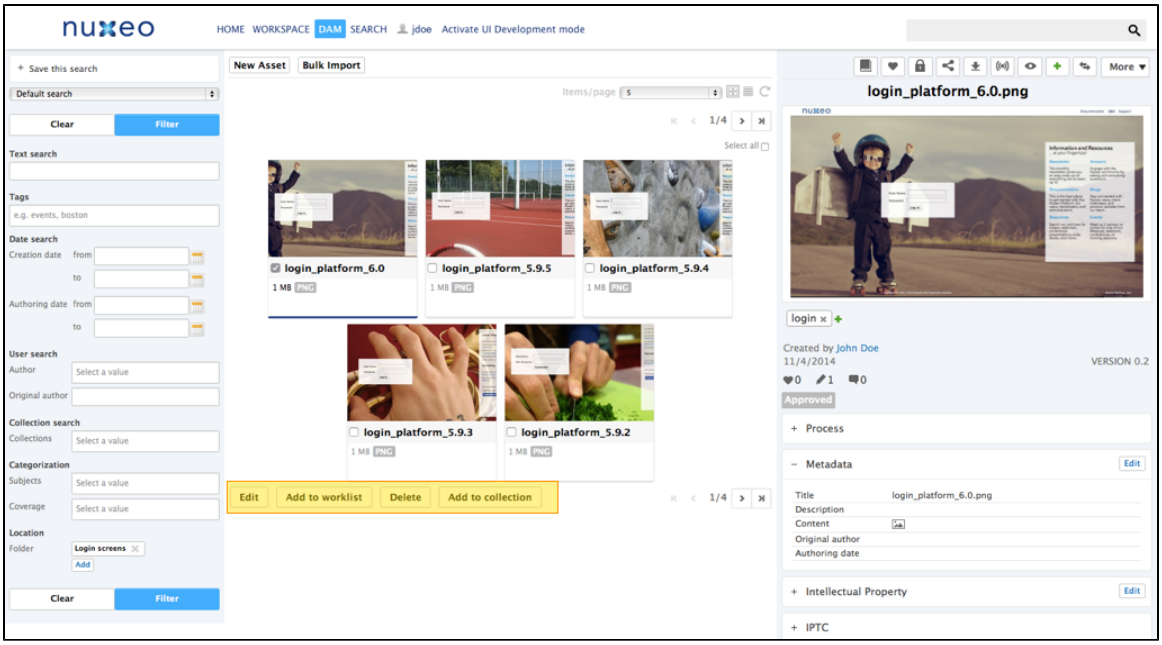
Available since Nuxeo Platform 5.7.1.



DAM Current selection lists

Technical name: `DAM_CURRENT_SELECTION_LIST`

Available since Nuxeo Platform 5.7.1.



Adapting Templates to Display an Action

Since Nuxeo Platform 5.6, an action can define the way it will be rendered by using the `type` attribute. This make it easier to combine different kinds of rendering for a group of actions, and this is done by using widget types for action types, to leverage features from the Nuxeo Layout Framework.

The `template` action type makes it possible to define a custom action rendering. New action types can also be contributed to the framework.

A series of widget types displaying are available by default, see the pages Tab Designer Widget Types and Advanced Widget Types. These widget types include rendering configuration options that are implemented by default action widget types (CSS styling, display as buttons or links, for instance).

Here are two ways of rendering actions.

Rendering Actions via Widget Definitions

Here is a sample widget definition to render actions using category `MY_CATEGORY`:

```
<extension target="org.nuxeo.ecm.platform.forms.layout.WebLayoutManager"
  point="widgets">
  <widget name="userActions" type="documentActionsWithForms">
    <properties mode="view">
      <property name="category">MY_CATEGORY</property>
      <property name="actionsDisplay">links</property>
      <property name="overallDisplay">horizontal_block</property>
      <property name="styleClass">userActions</property>
    </properties>
  </widget>
</extension>
```

This widget can be displayed on a page directly using the following sample code:

```
<div xmlns:nxl="http://nuxeo.org/nxforms/layout">
  <nxl:widget name="userActions" mode="view" value="#{currentDocument}" />
</div>
```

Of course this widget definition can also be included within a layout definition, as it's done for Incremental Layouts configuration.

Rendering Actions via Dynamically Computed Widget

It can also be useful to generate the widget definition dynamically from the widget template, by passing the widget properties as tag attributes to the `nxl:widgetType` tag:

```
<div xmlns:nxl="http://nuxeo.org/nxforms/layout">
  <nxl:widgetType name="documentActionsWithForms"
    widgetName="documentActionsUpperButtons"
    mode="view"
    label=""
    actionStyleClass="button"
    actionsDisplay="icons"
    overallDisplay="horizontal_block"
    widgetProperty_category="MY_CATEGORY"
    maxActionsNumber="5"
    value="#{currentDocument}" />
</div>
```

Notice the tag attribute `widgetProperty_category` used to define the actions category: as widget types also have a notion of category, adding `widgetProperty_` prefix to the attribute makes it possible to explicitly state that this is a widget property.

See also chapter about Layout and Widget Display.

Related sections in this documentation

- Actions Overview
- Standard Action Types
- Custom Action Types

Related sections in Studio documentation

- User Actions

Incremental Layouts and Actions

Actions are leveraged by the layout framework to include widgets inside layouts dynamically, benefiting from sorting and filtering features of actions within layouts.
Configuration is a bit overkill for now; this could be simplified in the future, but the main principle is to:

1. Define a layout including action widgets.
2. Define action widgets referencing an action category (depending on the use case).
3. Define actions with type `widget` within this category, referencing a widget name.
4. Define widgets with corresponding names.

Here is a complete example, taken from the default Nuxeo Summary layout, and showing how some widgets are shown conditionally on the page.

Summary Layout Definition

```
<extension target="org.nuxeo.ecm.platform.forms.layout.WebLayoutManager"
  point="layouts">
  <layout name="grid_summary_layout">
    <templates>
      <template mode="any">
        /layouts/layout_grid_template.xhtml
      </template>
    </templates>
    <rows>
      <row>
        <properties mode="any">
          <property name="nxl_gridStyleClass_0">gridStyle12</property>
        </properties>
        <widget>summary_panel_top</widget>
      </row>
      <row>
        <properties mode="any">
          <property name="nxl_gridStyleClass_0">gridStyle7</property>
          <property name="nxl_gridStyleClass_1">gridStyle5</property>
        </properties>
        <widget>summary_panel_left</widget>
        <widget>summary_panel_right</widget>
      </row>
      <row>
        <properties mode="any">
          <property name="nxl_gridStyleClass_0">gridStyle12</property>
        </properties>
        <widget>summary_panel_bottom</widget>
      </row>
    </rows>
  </layout>
</extension>
```

This layout is using a grid, so that widgets are piled up within grid slots and stacked up correctly when some widgets are not displayed.

Let's look at the `summary_panel_left` widget configuration:

Action Widget Definition

```
<extension target="org.nuxeo.ecm.platform.forms.layout.WebLayoutManager"
  point="widgets">
  <widget name="summary_panel_left" type="documentActions">
    <handlingLabels>true</handlingLabels>
    <labels>
      <label mode="any"></label>
    </labels>
    <properties widgetMode="any">
      <property name="category">SUMMARY_PANEL_LEFT</property>
      <property name="subStyleClass">summaryActions</property>
    </properties>
  </widget>
</extension>
```

Actions can be displayed by this widget when using the category `SUMMARY_PANEL_LEFT`. Here is a sample action configuration:

```
<extension target="org.nuxeo.ecm.platform.actions.ActionService"
  point="actions">
  <action id="summary_note_text" type="widget" order="100">
    <category>SUMMARY_PANEL_LEFT</category>
    <properties>
      <property name="widgetName">summary_note_text</property>
    </properties>
    <filter-id>hasNote</filter-id>
  </action>
</extension>
```

This action is using the type `widget`, referencing the widget named `summary_note_text`. Note that it's also using a filter that makes sure the widget will be shown only when the document has a schema named `note`:

```
<extension target="org.nuxeo.ecm.platform.actions.ActionService"
  point="filters">
  <filter id="hasNote">
    <rule grant="true">
      <schema>note</schema>
    </rule>
  </filter>
</extension>
```

Finally, here is the widget configuration:

```
<extension target="org.nuxeo.ecm.platform.forms.layout.WebLayoutManager"
  point="widgets">
  <widget name="summary_note_text" type="richtext_with_mimetype">
    <fields>
      <field>note:note</field>
      <field>note:mime_type</field>
    </fields>
    <properties mode="view">
      <property name="translatedHtml">
        #{noteActions.translateImageLinks(field_0)}
      </property>
      <property name="cssClass">note_content_block</property>
    </properties>
    <controls mode="any">
      <control name="requireSurroundingForm">true</control>
    </controls>
  </widget>
</extension>
```

Note that the widget holds a control to make sure a form is added around it: the summary layout is not surrounded by a form since version 5.8 to allow defining fine-grained forms inside it.

Related sections in this documentation

- How to Add a New Widget to the Default Summary Layout
- Layout and Widget Definitions
- Actions Overview

Related section in Studio documentation
• Form Layouts • Tabs • User Actions

Document List Management

Management of a lost of documents is useful for clipboard and generally document selection features.

The Document List Manager provides a service to manage lists of Nuxeo documents.

These lists of documents can have properties such as:

- A name, defined by `name` attribute.
- A scope (session or conversation), defined by `<isSession/>` tag - it defines if the memory storage occurs in the Seam session context or in the Seam conversation context.
- A persistence (SQL directory or not present), defined by `<persistent/>` tag - the service persists only the list of the document references, not the real documents; the lists of document references is persisted in a SQL directory, which is generic and does not need any configuration.

The lists of documents can be invalidated when Seam events are raised. This is useful, for example, for resetting `CURRENT_SELECTION` lists when the user change the current folder or when a new search is performed.

Documents lists can be defined like in the following example (`OSGI-INF/documentslists-contrib.xml`):

```
<extension target="org.nuxeo.ecm.webapp.documentsLists.DocumentsListsService"
point="list">

  <documentsList name="CLIPBOARD">
    <category>CLIPBOARD</category>
    <imageURL>/img/clipboard.gif</imageURL>
    <title>workingList.clipboard</title>
    <defaultInCategory>false</defaultInCategory>
    <supportAppends>false</supportAppends>
  </documentsList>

  <documentsList name="CURRENT_SELECTION">
    <events>
      <event>folderishDocumentSelectionChanged</event>
      <event>searchPerformed</event>
    </events>
    <isSession>false</isSession>
  </documentsList>

</extension>
```

Here is a sample code to get the list of selected documents within a Seam Component:

```
@In(create = true)
protected transient DocumentsListsManager documentsListsManager;

public boolean getCanCopy() {
    if (navigationContext.getCurrentDocument() == null) {
        return false;
    }
    return
!documentsListsManager.isWorkingListEmpty(DocumentsListsManager.CURRENT_DOCUMENT_SELEC
TION);
}
```

Navigation URLs

There are two services that help building GET URLs to restore a Nuxeo context. The default configuration handle restoring the current document, the view, current tab and current sub tab.

Document View Codec Service

The service handling document views allows registration of codecs. Codecs manage coding of a document view (holding a document reference, repository name as well as key-named string parameters) in to a URL, and decoding of this URL into a document view.

In this section

Example of a document view codec registration:

```
<extension
  target="org.nuxeo.ecm.platform.url.service.DocumentViewCodecService"
  point="codecs">

  <documentViewCodec name="docid" enabled="true" default="true" prefix="nxdoc"
    class="org.nuxeo.ecm.platform.url.codec.DocumentIdCodec" />
  <documentViewCodec name="docpath" enabled="true" default="false" prefix="nxpath"
    class="org.nuxeo.ecm.platform.url.codec.DocumentPathCodec" />

</extension>
```

In this example, the `docid` codec uses the document uid to resolve the context. URLs are of the form `http://site/nuxeo/nxdoc/demo/docuid/view` . The `docpath` codec uses the document path to resolve the context. URLs are of the form `http://site/nuxeo/nxpath/demo/path/to/my/doc@view` .

Additional parameters are coded/decoded as usual request parameters.

Note that when building a document view, the URL service will require a view id. Other information (document location and parameters) are optional, as long as they're not required for your context to be initialized correctly.

URL Policy Service

The service handling URLs allows registration of patterns. These patterns help saving the document context and restoring it thanks to information provided by codecs. The URL service will iterate through its patterns, and use the first one that returns an answer (proving decoding was possible).

Example of a URL pattern registration:

```
<extension target="org.nuxeo.ecm.platform.ui.web.rest.URLService"
  point="urlpatterns">

  <urlPattern name="default" enabled="true">
    <defaultURLPolicy>true</defaultURLPolicy>
    <needBaseURL>true</needBaseURL>
    <needRedirectFilter>true</needRedirectFilter>
    <needFilterPreprocessing>true</needFilterPreprocessing>
    <codecName>docid</codecName>
    <actionBinding>#{restHelper.initContextFromRestRequest}</actionBinding>
    <documentViewBinding>#{restHelper.documentView}</documentViewBinding>
    <newDocumentViewBinding>#{restHelper.newDocumentView}</newDocumentViewBinding>
    <bindings>
      <binding name="tabId">#{webActions.currentTabId}</binding>
      <binding name="subTabId">#{webActions.currentSubTabId}</binding>
    </bindings>
  </urlPattern>

</extension>
```

In this example, the "default" pattern uses the above `docid` codec. Its is set as the default URL policy, so that it's used by default when caller does not specify a pattern to use. It needs the base URL: the `docid` codec only handles the second part if the URL. It needs redirect filter: it will be used to provide the context information to store. It needs filter preprocessing: it will be used to provide the context information to restore. It's using the `docid` codec.

The action binding method handles restoring of the context in the Seam context. It takes a document view as parameter. It requires special attention: if you're using conversations (as Nuxeo does by default), you need to annotate this method with a `@Begin` tag so that it uses the conversation identifier passed as a parameter if it's still valid, or initiates a new conversation in other cases. The method also needs to make sure it initializes all the seam components it needs (documentManager for instance) if they have not be in intialized yet.

The additional document view bindings are used to pass document view information through requests. The document view binding maps to corresponding getters and setters. The new document view binding is used to redirect to build GET URL in case request is a POST: it won't have the information in the URL so it needs to rebuild it.

Other bindings handle additional request parameters. In this example, they're used to store and restore tab and sub tab information (getters and setters have to be defined accordingly).

Since 5.5, a new element `documentViewBindingApplies` can be set next to document view bindings: it makes it possible to select what pattern descriptor will be used on a POST to generate or get the document view, as well as building the URL to use for the redirect that will come just after the POST. It has to resolve to a boolean expression, and if no suitable pattern is found, the default one will be used.

Additional Configuration

The URL patterns used need to be registered on the authentication service so that they're considered as valid URLs. Valid URLs will be stored in the request, so that if authentication is required, user is redirected to the URL after login.

Example of a start URL pattern registration:

```
<extension
  target="org.nuxeo.ecm.platform.ui.web.auth.service.PluggableAuthenticationService"
  point="startURL">

  <startURLPattern>
    <patterns>
      <pattern>nxdoc/</pattern>
    </patterns>
  </startURLPattern>

</extension>
```

Just the start of the URL is required in this configuration. Contributions are merged: it is not possible to remove an existing start pattern.

The URL patterns used also need to be handled by the default Nuxeo authentication service so that login mechanism (even for anonymous) applies for them.

Example authentication filter configuration:

```
<extension target="web#STD-AUTH-FILTER">
  <filter-mapping>
    <filter-name>NuxeoAuthenticationFilter</filter-name>
    <url-pattern>/nxdoc/*</url-pattern>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>FORWARD</dispatcher>
  </filter-mapping>
</extension>
```

This is a standard filter mapping configuration.

URL JSF Tags

There are some JSF tags and functions helping you to define what kind of GET URL should be displayed on the interface.

Example of `nxd:restDocumentLink` use:

```
<nxd:restDocumentLink document="#{doc}">
  <nxh:outputText value="#{nxd:titleOrId(doc)}" />
</nxd:restDocumentLink>
```

In this example, the tag will print a simple link, using the default pattern, and build the document view using given document model, using its default view.

Please refer to the tag library documentation available at http://community.nuxeo.com/api/nuxeo/5.8/tlddoc/nxd/restDocumentLink.html for additional parameters: it's possible to set the tab, sub tab, and use a specific URL pattern.

Note that you can also use JSF functions to build the GET URL. This is what's done for file links: the function queries the URL policy service to build the URL.

Example of a JSF function use:

```
<nxh:outputLink rendered="#{doc.hasSchema('file') and !empty doc.file.content}"
  value="#{nxd:fileUrl('downloadFile', doc, 'file:content', doc.file.filename)}">
  <nxh:graphicImage value="/icons/download.png" style="vertical-align:middle"
    title="#{doc.file.filename}" />
</nxh:outputLink>
```

Here is an the `fileURL` method code as an example:

```
public static String fileUrl(String patternName, DocumentModel doc,
        String blobPropertyName, String filename) {
    try {
        DocumentLocation docLoc = new DocumentLocationImpl(doc);
        Map<String, String> params = new HashMap<String, String>();
        params.put(DocumentFileCodec.FILE_PROPERTY_PATH_KEY,
                blobPropertyName);
        params.put(DocumentFileCodec.FILENAME_KEY, filename);
        DocumentView docView = new DocumentViewImpl(docLoc, null, params);

        // generate url
        URLPolicyService service = Framework.getService(URLPolicyService.class);
        if (patternName == null) {
            patternName = service.getDefaultPatternName();
        }
        return service.getUrlFromDocumentView(patternName, docView,
                BaseURL.getBaseURL());

    } catch (Exception e) {
        log.error("Could not generate url for document file", e);
    }

    return null;
}
```

Default URL Patterns

This page introduces the URLs used in Nuxeo Platform and tries to bind them to services/addons.
URLs:

- `/nuxeo/nxdoc`: navigation by document ID in Document Management
- `/nuxeo/nxpath`: navigation by document path in Document Management
- /`nuxeo/nxfile`: URL to download a file
- `/nuxeo/nxbigfile`: URL to download big files or automatic switch from nxfile if file size > 5 MB (default value)
- `/nuxeo/nxhome`: navigation under Home tab
- `/nuxeo/nxcollab`: navigation in Social Collaboration
- `/nuxeo/nxdam`: navigation in DAM
- `/nuxeo/nxadmin`: navigation in Admin Center

The complete list of contributions can be browsed on the explorer.

Other URLs exposed by WebEngine module are of the form `/nuxeo/site/*` (where * is a service offered by a WebEngine module), please refer to Default WebEngine Applications for the list of corresponding URLs.

Other URLs:

- `/nuxeo/atom/cmis` : CMIS service

Other URL related documentation

- URLs for Files
- Navigation URLs

URLs for Files

For a single file, in schema "file", where blob field is named "file" and file name field is named "filename":

```
<nxh:outputLink
  value="#{nxd:fileUrl('downloadFile', currentDocument, field.fullName,
currentDocument.file.filename)}">
  <nxh:graphicImage
    value="#{nxd:fileIconPath(currentDocument[field.schemaName][field.fieldName])}"
    rendered="#{! empty
nxd:fileIconPath(currentDocument[field.schemaName][field.fieldName])}" />
  <nxh:outputText value="#{currentDocument.file.filename}" />
</nxh:outputLink>
```

For a list of files, in schema "files", where list name is "files" and in each item, blob field is named "file" and file name field is named "filename":

```
<nxu:inputList value="#{currentDocument.files.files}" model="model"
rendered="#{not empty currentDocument.files.files}">
  <nxh:outputLink
    value="#{nxd:complexFileUrl('downloadFile', currentDocument, 'files:files',
model.rowIndex, 'file', currentDocument.files.files[model.rowIndex].filename)}">
    <nxh:graphicImage
      value="#{nxd:fileIconPath(currentDocument.files.files[model.rowIndex].file)}"
      rendered="#{! empty
nxd:fileIconPath(currentDocument.files.files[model.rowIndex].file)}" />
    <nxh:outputText value="#{currentDocument.files.files[model.rowIndex].filename}"
/>
  </nxh:outputLink>
  <t:htmlTag value="br" />
</nxu:inputList>
```

This gives you get URLs of the form:

```
http://localhost:8080/nuxeo/nxfile/default/8f5aca13-e9d9-4b7b-a1d9-a1dcd74cc709/blob
holder:0/mainfile.jpg
http://localhost:8080/nuxeo/nxfile/default/47ad14f2-c7a6-4a3f-8e4b-6c2cf1458f5a/file
s:files/0/file/firstfile.jpg
```

Namespaces:

```
xmlns:t="http://myfaces.apache.org/tomahawk"
xmlns:nxh="http://nuxeo.org/nxweb/html"
xmlns:nxl="http://nuxeo.org/nxforms/layout"
xmlns:nxu="http://nuxeo.org/nxweb/util"
```

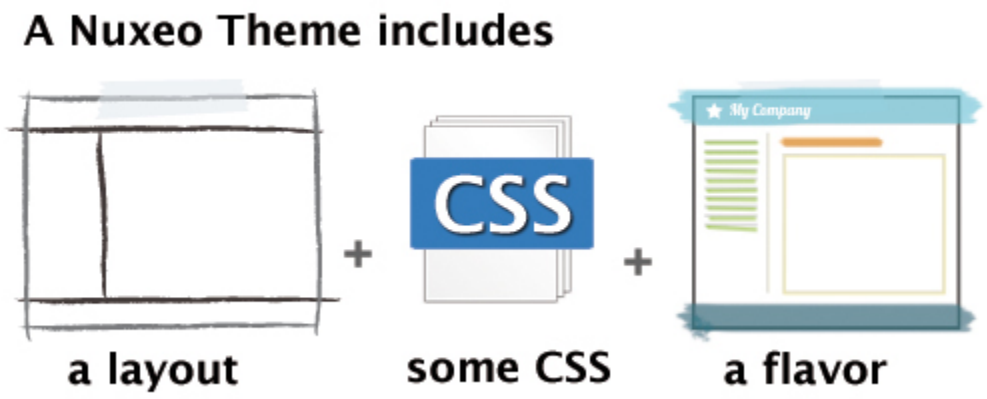
Related topics in this documentation

- Default URL Patterns
- Navigation URLs

Theme

The theme is in charge of the global *layout* or *structure* of a page (composition of the header, footer, left menu, main area...), as well as its *branding* or *styling* using CSS. It also handles

additional resources like JavaScript files.



If you already understand theme concepts, maybe you'd like to check Theme and Style How-Tos.

Note that since Nuxeo 5.5, the theme system has been improved to better separate the page layout from the page branding. The page layout is still defined using XML, but the branding now uses traditional CSS files. These files can now hold dynamic configuration to handle flavors, but they can also be simply defined as static resources like JS files (as in a standard web application). Although Nuxeo 5.5 is fully backward compatible with the old system, we encourage you to migrate your custom theme to the new system as it will ease maintenance and further upgrades.

On this page

Online Style Guide
An online UI Style Guide is available to help you design your pages, customize and style them.

Page Layout

The layout for the different theme pages is defined in an XML file with the following structure.
For instance, in 5.5, the default *galaxy* theme is defined in file `themes/document-management.xml`. It represents the structure displayed when navigating in the *Document Management* tab of the application (the *galaxy* name has been kept for compatibility).

```
<theme name="galaxy" template-engines="jsf-facelets">
  <layout>
    <page name="default" class="documentManagement">
      <section class="nxHeader">
        <cell class="logo">
          <!-- logo -->
          <fragment type="generic fragment"/>
        </cell>
        ...
      </section>
      ...
    </page>
  </layout>
  <properties element="page[1]/section[2]/cell[2]/fragment[2]">
    <name>body</name>
    ...
  </properties>
  <formats>
    <widget element="page[1]|page[2]|page[3]">
      <view>page frame</view>
    </widget>
    <widget element="page[1]/section[1]/cell[1]/fragment[1]">
      <view>Nuxeo DM logo (Galaxy Theme)</view>
    </widget>
    ...
  </formats>
</theme>
```

It is now possible to add a class attribute to the `<page>`, `<section>`, `<cell>` and `<fragment>` elements. They will be part of the generated stylesheet.

This structure is declared to the theme service using the following extension point:

```
<extension target="org.nuxeo.theme.services.ThemeService" point="themes">
  <theme>
    <src>themes/document-management.xml</src>
  </theme>
</extension>
```

The file referenced by the "src" tag is looked up in the generated jar.

Views referenced in this layout still need to be defined in separate extension points. For instance here is the logo contribution:

```
<extension target="org.nuxeo.theme.services.ThemeService" point="views">
  <view name="Nuxeo DM logo (Galaxy Theme)" template-engine="jsf-facelets">
    <format-type>widget</format-type>
    <template>incl/logo_DM_galaxy.xhtml</template>
  </view>
</extension>
```

The global application settings still need to be defined on the "applications" extension point:

```
<extension target="org.nuxeo.theme.services.ThemeService" point="applications">
  <application root="${org.nuxeo.ecm.contextPath}"
    template-engine="jsf-facelets">
    <negotiation>
      <strategy>nuxeo5</strategy>
      <default-engine>default</default-engine>
      <default-theme>galaxy/default</default-theme>
      <default-perspective>default</default-perspective>
    </negotiation>
    <!-- Cache control for theme resources (/nxthemes-lib/) -->
    <resource-caching>
      <lifetime>36000</lifetime>
    </resource-caching>
    <!-- Cache control for theme styles (/nxthemes-css/) -->
    <style-caching>
      <lifetime>900</lifetime>
    </style-caching>
    <view id="/editor_link_search_document.xhtml">
      <theme>galaxy/popup</theme>
    </view>
  </application>
</extension>
```

In the above example, you can see that the *galaxy/default* page is the default page of the application. Other pages like *galaxy/popup* can also be explicitely set for some specific views.

Branding: Use Flavors and Styles!

CSS configuration used to be held by the same XML file than the one defining the pages structures. This was a problem for modularity, as some addons needed to add styling to existing pages without changing the global structure of the page.

Since 5.5, extension points have been added to allow style contributions separately from the structure. A new notion of *flavor* has also been added, to make it possible to switch some elements held by the branding (colors, logo...) without changing the associated styles: the local configuration makes it possible to change this flavor so that branding can be customized when navigating in some parts of the document hierarchy.

The pages Extension Point

Each theme page needs to have a `themePage` contribution to the `pages` extension point: this is the main link between the page structure and its branding.
Here is an example of the *galaxy* theme default page:

```
<extension target="org.nuxeo.theme.styling.service" point="pages">
  <themePage name="galaxy/default">
    <defaultFlavor>default</defaultFlavor>
    <flavors>
      <flavor>default</flavor>
      <flavor>rainbow</flavor>
    </flavors>
    <styles>
      <style>basics</style>
      <style>buttons_and_actions</style>
      <style>header</style>
      <style>body</style>
      <style>footer</style>
      <style>my_custom_styles<style>
      ...
    </styles>
    <resources>
      <resource>static_styles.css</resource>
      <resource>jquery.fancybox.js</resource>
    </resources>
  </themePage>
  ...
</extension>
```

- `<defaultFlavor>` refers to the default flavor used for the page. See below for `flavor` definition.
- `<flavors>` refers to the different flavors available for this page in the Theme local configuration.
- `<styles>` refers to the dynamic CSS stylesheets that will be embedded in this page. See below for `styles` definition.
- `<resources>` refers to static resources (CSS stylesheet that do not use variables, and Javascript file). See below for `resources` definition.

Style order is important!
The theme engine will actually concatenate all the stylesheets defined in the `<styles>` element, in the order of their declaration, to build one big stylesheet named `myCustomTheme-styles.css`.
So if you need to override a CSS class existing in one of the Nuxeo stylesheets, for example `.nxHeader .logo` in `header.css`, you will have to do this in a custom stylesheet declared after the Nuxeo one.
For example `my_custom_styles` should come after `header`.

The styles Extension Point

Each dynamic CSS stylesheet embedded in a theme page needs to have a `style` contribution to the `styles` extension point.
Here is an example of some of the basic stylesheet contributions:

```
<extension target="org.nuxeo.theme.styling.service" point="styles">
  <!-- Global styles -->
  <style name="breadcrumb">
    <src>themes/css/breadcrumb.css</src>
  </style>
  <style name="buttons_and_actions">
    <src>themes/css/buttons_and_actions.css</src>
  </style>
  <style name="basics">
    <src>themes/css/basics.css</src>
  </style>
  <style name="body">
    <src>themes/css/body.css</src>
  </style>
</extension>
```

The files referenced by the "src" tags are looked up in the generated JAR.

Here is an excerpt of the "buttons_and_actions" CSS file:

```
a.button { text-decoration: none }

.major, input.major {
 background: none repeat scroll 0 0 #0080B0;
 border: 1px solid #014C68;
 color: #fff;
 text-shadow: 1px 1px 0 #000 }

input.button[disabled=disabled], input.button[disabled=disabled]:hover,
input.button[disabled], input.button[disabled]:hover {
 background: none "button.disabled (__FLAVOR__ background)";
 border: 1px solid; border-color: "button.disabled (__FLAVOR__ border)";
 color: "link.disabled (__FLAVOR__ color)";
 cursor: default;
 visibility: hidden }
```

It can mix standard CSS content, and also hold variables that can change depending on the selected flavor (see below).

The flavor Extension Point

Each flavor needs to have a `flavor` contribution to the `flavors` extension point.
Here is an example of the default flavor:

```
<extension target="org.nuxeo.theme.styling.service" point="flavors">
  <flavor name="default">
    <presetsList>
      <presets category="border" src="themes/palettes/default-borders.properties" />
      <presets category="background"
src="themes/palettes/default-backgrounds.properties" />
      <presets category="font" src="themes/palettes/default-fonts.properties" />
      <presets category="color" src="themes/palettes/default-colors.properties" />
    </presetsList>
    <label>label.theme.flavor.nuxeo.default</label>
    <palettePreview>
      <colors>
        <color>#cfecff</color>
        <color>#70bbff</color>
        <color>#4e9ae1</color>
        ...
      </colors>
    </palettePreview>
    <logo>
      <path>/img/nuxeo_logo.png</path>
      <previewPath>/img/nuxeo_preview_logo_black.png</previewPath>
      <width>92</width>
      <height>36</height>
      <title>Nuxeo</title>
    </logo>
  </flavor>
</extension>
```

- `<presets>` contributions define a list of CSS property presets, as it was already the case in the old system. They represent variables that can be reused in several places in the CSS file.
  For example in `themes/palettes/default-borders.properties`we can define:

  ```
  color.major.medium=#cfecff
  color.major.strong=#70bbff
  color.major.stronger=#4e9ae1
  ```

  A preset can be referenced in a dynamic CSS file, using the pattern "*preset_name* (*__FLAVOR__ preset_category*)".

  ```
  .button:hover, input.button:hover {
    background: none "color.major.medium (__FLAVOR__ background)";
    border: 1px solid; border-color: "button.hover (__FLAVOR__ border)" }
  ```

- `<label>` and `<palettePreview>` are used to display the flavor in the Theme local configuration.
- `<logo>` is used to customize the default logo view `logo_DM_galaxy.xhtml`.

Theme Static Resources

JS and CSS static files (ie. that don't need to be configured with flavors) should be defined as static resources, using the `resources` extensi on point as it was the case in the old system.

Note that dynamic CSS files are parsed, and this parsing may not accept CSS properties that are not standard (like CSS3 properties for instance). Placing these properties in static resources is a workaround for this problem if you do not need to make them reference flavors.

Here are sample contributions to this extension point:

```
<extension target="org.nuxeo.theme.styling.service" point="resources">
  <resource name="static_styles.css">
    <path>css/static_styles.css</path>
  </resource>
  <resource name="jquery.fancybox.js">
    <path>scripts/jquery/jquery.fancybox.pack.js</path>
  </resource>
</extension>
```

Note that the resource name **must** end by ".css" for CSS files, and ".js" for JavaScript files.

The files referenced by the "path" tags are looked up in the generated war directory (nuxeo.war) after deployment, so you should make sure the `OSGI-INF/deployment-fragment.xml` file of your module copies the files from the JAR there.

Related pages in this documentation

- How to Override a Default Style
- How to Show Theme Fragment Conditionaly
- How to Declare the CSS and Javascript Resources Used in Your Templates

Related pages in other documentation

- Unicolor Flavors Set
- Branding
- Applying a Preset Look to a Space

Migrating my Customized Theme

If you have customized your Nuxeo theme by coding directly the (huge) XML theme file and adding contributions in a `theme-contrib.xml` file, such as `views`, `resources` and `presets`, you will want to know how to migrate this customized theme to use the new system available since version 5.5.

First make sure you have read the Theme page to understand the basic concepts.

Theme XML File

Let's say you have a theme file called `theme-custom.xml` where you have defined your own page layouts and styles.
You now want to keep in this file only the page layouts and get rid of all the styles.

Page Layouts

Nothing has changed here in terms of page layouts.
Keep your existing `page/section/cell/fragment` structure for each page in the `<layout>` elem ent.

On this page

```
<layout>
  <page name="default">
    <section>
      <cell>
        <fragment type="generic fragment"/>
      </cell>
    </section>
  </page>
  <page name="popup">
    ...
  </page>
</layout>
```

Page, Section, Cell and Fragment Formats

With the old system, to add a style on one of these elements, you had to use:

```
<formats>
  <layout element="page[1]/section[1]/cell[1]">
    <width>18%</width>
    <text-align>center</text-align>
    <vertical-align>bottom</vertical-align>
  </layout>
  ...
</formats>
```

If you take a look at the new default theme file `document-management.xml`, you will see that a `class` attribute has been added on each `page`, `section`, `cell` and `fragment` element. It references a CSS class in one of the CSS stylesheets declared in the `styles` extension point. Taking the same example, we now have:

```
<page name="default" class="documentManagement">
  <section class="nxHeader">
    <cell class="logo">
```

And in `themes/css/header.css`:

```
.nxHeader .logo { text-align: center; vertical-align: middle; width: 18% }
```

Therefore, in order to migrate, you need to:

- Remove all `<layout>` elements embedded in `<formats>`.
  Copy/paste them in your favorite text editor since you will need to translate them into CSS later!
- Add a `class` attribute on each `page`, `section`, `cell` and `fragment` element.
  Here you can take advantage of the classes that exist in the Nuxeo default theme file, especially if you did not customize these elements.
- Make sure these CSS classes are defined in a CSS stylesheet embedded for each page it should apply to (see below about CSS stylesheets).
  If you use the classes that apply to the Nuxeo default theme (`nxHeader, logo, ...`), they will already be defined!
  Else you will have to define them in a custom CSS file.

Styles

With the old system, CSS styles were defined in the XML theme file:

```
<style name="header and footer style" inherit="default buttons">
  <selector path="a, a:link, a:visited">
    <text-decoration>none</text-decoration>
  </selector>
</style>
```

With the new system these styles are defined in CSS stylesheets. In this case in `themes/css/basic.css`:

```
a, a:link, a:visited { color: "link (__FLAVOR__ color)"; text-decoration: none }
```

Therefore, in order to migrate, you need to:

- Remove all `<style>` elements embedded in `<formats>`.
  Again, copy/paste them in a text editor since you will need to translate them into CSS later!

Theme Contributions: Pages, Flavors and Styles

You now have to translate all styles from the old XML format to the new (and nicer) CSS one.
If you take a look at the new theme contribution file `theme-contrib.xml`, you will see that 3 extension points have been added.
For each one, you need to check that the existing contributions match your specific need and if not you will have to add yours.
Lets say you have a customized theme named *myCustomTheme* with a specific page *myPage* for which you want to use the *myFlavor* flavor and the *my_custom_styles.css* stylesheet.

Pages

Add a `<themePage>` contribution to the `pages` extension point.

```
<extension target="org.nuxeo.theme.styling.service" point="pages">
  <themePage name="myCustomTheme/myPage">
    <defaultFlavor>myFlavor</defaultFlavor>
    <flavors>
      <flavor>default</flavor>
      <flavor>rainbow</flavor>
    </flavors>
    <styles>
      <style>basics</style>
      <style>buttons_and_actions</style>
      <style>body</style>
      <style>header</style>
      <style>footer</style>
      <style>navigation</style>
      <style>tables</style>
      <style>my_custom_styles<style>
    </styles>
  </themePage>
</extension>
```

Style order is important!
The theme engine will actually concatenate all the stylesheets defined in the `<styles>` element, in the order of their declaration, to build one big stylesheet named `myCustomTheme-styles.css`.
So if you need to override a CSS class existing in one of the Nuxeo stylesheets, for example `.nxHeader .logo` in `header.css`, you will have to do this in a custom stylesheet declared after the Nuxeo one.
For example `my_custom_styles` should come after `header`.

Flavors

Add a `<flavor>` contribution to the `flavors` extension point.

```
<extension target="org.nuxeo.theme.styling.service" point="flavors">
  <flavor name="myFlavor">
    <label>label.theme.flavor.myFlavor</label>
    <presetsList>
      <presets category="border" src="themes/palettes/myFlavor-borders.properties" />
      <presets category="background"
src="themes/palettes/myFlavor-backgrounds.properties" />
      <presets category="font" src="themes/palettes/myFlavor-fonts.properties" />
      <presets category="color" src="themes/palettes/myFlavor-colors.properties" />
    </presetsList>
    <palettePreview>
      <colors>
        <color>#cfecff</color>
        <color>#70bbff</color>
        <color>#4e9ae1</color>
      </colors>
    </palettePreview>
  </flavor>
</extension>
```

Of course you need to create the corresponding file for each preset (in this example in the `themes/palettes/` directory).
You can take example on the Nuxeo default presets.

Styles

Add a `<style>` contribution to the `styles` extension point.

```
<extension target="org.nuxeo.theme.styling.service" point="styles">
  <style name="my_custom_styles">
    <src>themes/css/my_custom_styles.css</src>
  </style>
</extension>
```

Finally you can take advantage of the flavor system by making your CSS stylesheet dynamic! For example:
In `my_custom_styles.css`:

```
.nxHeader { background-color: "header (__FLAVOR__ background)"; overflow: auto;
width:100% }
```

In `myFlavor-backgrounds.properties` :

```
header=#CCCCCC
```

Old Palette Contributions

Finally, you can get rid of the old `palette` contributions to the `presets` extension point, if you had any. They are not used anymore.

```
<extension target="org.nuxeo.theme.services.ThemeService" point="presets">
  <palette name="Custom borders" src="themes/palettes/custom-borders.properties"
    category="border" />
  ...
</extension>
```

JSF and Ajax Tips and How-Tos

Topics in this section:

- Ajax4jsf Best Practices
- Ajax Forms and Actions — Here is some helpful tips and tricks when working with Ajax forms or actions.
- JSF number of views configuration
- JSF and Javascript — Getting a tag client id with JavaScript might be an issue. Here's how you can get your tag using DOM.
- JSF tag library registration
- JSF troubleshoot
- Double Click Shield — A new feature called "Double Click Shield" has been introduced in Nuxeo 5.8. It is enabled in the JSF interface and it addresses the issue described in "I Get an Error When I Click on Two Links Quickly".

Ajax4jsf Best Practices

Use a4j:region to Speed up Ajax Rendering

Wrap JSF components that are affected by rerender targets or `a4j:support` tags with a `a4j:region renderRegionOnly="true"` in order to both lower the network bandwith usage and the CPU pressure on the server since a much smaller part of the JSF tree will be processed:

TODO: add an example here

Use a4j:outputPanel as Target of Rerender Clauses

If you have JSF elements with dynamic render clause that are evaluated to false in the first page rendering, a4j reRendering will fail to make them appear afterward since will not be present in the JSF rendering tree. Wrapping them with a `a4j:outputPanel` that holds the reRender target id will solve the issue.

TODO: add an example here

Also see for reference: http://www.jroller.com/a4j/entry/ajax_regions_and_output_panels

On this page

Use Attribute layout="block" on a4j:outputPanel Tag

If the `a4j:outputPanel` tag will contain divs or tables, for instance, you should use attribute `layout="block"` so that it's displayed using a `div` instead of a `span`, as a span tag containing divs or tables will produce invalid HTML.

See the reference documentation at http://docs.jboss.org/richfaces/latest_3_3_X/en/devguide/html/a4j_outputPanel.html

a4j:commandLink Does Not Behave Like h:commandLink

For instance if you naively replace `h:commandLink` by `a4j:commandLink` in the sort column links of a document listings you can get no effect on the first click (but works the second time but with the wrong arrow position).

The reason is that `h:commandLink` is a traditional JSF JavaScript postback that first sends a POST request to apply the new values and then redirect the browser using HTTP code 302 to the updated page using a second GET request that renders the results. In the `a4j:commandLink` case this is a different story since there is only one HTTP request to perform both the model values update and the new HTML fragment rendering. Hence if you use Seam variables computed by EVENT scoped factory, it is computed twice in the first case and only once (and then "cached") in the second case.

For instance the `SortActionsBean` has been improved to allow the caller to invalidate Seam components using an additional `f:param` with name `invalidateSeamVariables` to invalidate the `filterSelectModel` that gets cached too early (before we change the order with the Ajax action): http://hg.nuxeo.org/nuxeo/nuxeo-jsf/rev/7e388aa56786.

Hence the sort title controls can be ajaxified in the following way:

```
<a4j:commandLink id="title" action="#{sortActions.repeatSearch}"
              rendered="#{provider.sortable}" reRender="filterResultTable" >
      <h:outputText value="#{messages['label.content.header.title']}" />
      <f:param name="providerName" value="#{providerName}" />
      <f:param name="sortColumn" value="dc:title" />
      <f:param name="invalidateSeamVariables" value="filterSelectModel" />
      <h:panelGroup rendered="#{provider.sortInfo.sortColumn == 'dc:title' }"
        <h:graphicImage value="/icons/arrow_down.gif"
            rendered="#{provider.sortInfo.sortAscending}" />
        <h:graphicImage value="/icons/arrow_up.gif"
            rendered="#{!provider.sortInfo.sortAscending}" />
      </h:panelGroup>
</a4j:commandLink>
```

Use the New a4j Namespace

The URI for `a4j:` and `rich:` namespaces are changed to "http://richfaces.org/a4j" and "http://richfaces.org/rich". The previous URIs are also available for a backward compatibility but it is recommended to switch to the new names for consistency.

Testing with Selenium

Add the following functions in your user-extensions.js of your Selenium tests suite to be able to setup smart waiters in your Selenium tests suite that have a far better stability than "pause" steps that may slowdown then set suite a lot while still breaking randomly if the Selenium server is otherwise put under too much load: http://www.nuxeo.com/blog/development/2009/03/selenium-ajax-requests/.

There seems to be bugs remaining with this method: it works most of the time but sometimes breaks later unexpectedly. More investigations are needed to make it robust.

Ajax Forms and Actions

Here is some helpful tips and tricks when working with Ajax forms or actions.

Generic Advice

Always try to use the Ajax library features to help avoiding multiple calls to the server (ignoreDupResponses, eventsQueue,...), and re-render only needed parts of the page. This will prevent you from experiencing problems like " I Get an Error When I Click on Two Links Quickly.

Ajax4JSF library documentation is available at http://docs.jboss.org/richfaces/latest_3_3_X/en/tlddoc/a4j/tld-summary.html.

Submitting the Form When Hitting The "Enter" Key

The browser `form` tag will natively select the first input submit button in the form: be aware that tag `a4j:commandLink` is not one of them, so you should replace it with a `h:commandButton`. You will have to change the form into an Ajax form for the submission to be done in Ajax.

Here is an example:

```
<a4j:form id="#{quickFilterFormId}" ajaxSubmit="true"
  reRender="#{elementsToReRender}"
  ignoreDupResponses="true"
  requestDelay="100"
  eventsQueue="contentViewQueue"
  styleClass="action_bar">
  <nxl:layout name="#{contentView.searchLayout.name}" mode="edit"
    value="#{contentView.searchDocumentModel}"
    template="content_view_quick_filter_layout_template.xhtml" />
  <h:commandButton
    value="#{messages['label.contentview.filter.filterAction']}"
    id="submitFilter"
    styleClass="button"
    action="#{contentView.resetPageProvider()}" />
</a4j:form>
```

Ajax Re-Rendering

If you need to perform some Ajax re-rendering when submitting this button, they'd better be placed directly on the form tag itself: adding a tag `a4j:support` for this inside the `h:commandButton` tag will generates an additional call to the server when using some browsers (visible using Firefox 8.0) and may lead to errors when server is under load.

Command Button Visibility

Note also that the command button must be visible: if for some reasons you'd like it to be hidden, placing it inside a div using `display: none` will break the submission using some browsers (visible using Chrome 15).

Disabling Buttons after Form Submission

Last but not least, if you'd like the buttons to be disabled when submitting the form (to make sure people do not keep on clicking on it), they should be disabled by the form itself when the `onsubmit` JavaScript event is launched: if it is disabled when the `onclick` event is received on the button, some browsers will not send the request at all (visible with Chrome).

JavaScript Form Tricks

When the form is not generated by a reusable layout (like in the example above), you can also use more traditional JavaScript tricks to submit the form, like for instance:

```
<h:inputText value="#{searchActions.simpleSearchKeywords}"
  onkeydown="if (event.keyCode == 13) {Element.next(this).click();} else return
true;" />
<h:commandButton action="#{searchActions.performSearch}"
  value="#{messages['command.search']}" styleClass="button" />
```

JSF number of views configuration

The JSF/Seam stack used in the Nuxeo Platform uses server-side state saving: for each JSF component in the tree of graphical components, a state is saved server-side.

Each JSF component tree state is associated to a unique identifier, and the number of states is configurable using the JSF parameters `com.sun.faces.numberOfViewsInSession` and `com.sun.faces.numberOfLogicalViews`.

When the number of clicks exceeds this number of states, the page cannot be restored, and JSF may throw a `ViewExpiredException`.

Since version 5.7.1 (and 5.6.0-HF12), the default value in the Nuxeo Platform is set to 4 for both parameters (instead of 50 which was the previous default value) because it has an impact on performance.

Since version 5.7.2 (and 5.6.0-HF20), these values can be changed by adding (for example) the following line to the `bin/nuxeo.conf` file:

```
nuxeo.jsf.numberOfViewsInSession=25
nuxeo.jsf.numberOfLogicalViews=25
```

Related JIRA issues:

- NXP-9177: Reduce default number of view in JSF
- NXP-11423: Allow configuring the number of JSF logical views
- NXP-11677: Make JSF/Seam more resiliant to mulithreaded navigation

JSF and Javascript

Getting a tag client id with JavaScript might be an issue. Here's how you can get your tag using DOM.

```
document.getElementById('div-id').childNodes[0]
```

JSF tag library registration

When registering a new tag library, you would usually declare the facelets taglib file in the `web.xml` configuration file.

As this parameter can only be declared once, and is already declared in the nuxeo base ui module, you cannot declare it using the Nuxeo deployment feature.

A workaround is to put you custom taglib file `mylibary.taglib.xml` in the `META-INF` folder of your custom jar: it will be registered automatically (even if it triggers an error log at startup).

As a reminder, the tag library documentation file, `mylibrary.tld`, is usually placed in the same folder than the taglib file, but it is only used for documentation: it plays no role in the tags registration in the application.

JSF troubleshoot

Altough they are not all specific to Nuxeo framework, here are some troubleshooting issues that can be encountered with JSF.

NullPointerException? when rendering a page

If you have a stack trace that looks like:

```
java.lang.NullPointerException
        at
org.apache.myfaces.trinidadinternal.renderkit.core.xhtml.FormRenderer.encodeEnd(FormRe
nderer.java:210)
        at
org.apache.myfaces.trinidad.render.CoreRenderer.encodeEnd(CoreRenderer.java:211)
        at
org.apache.myfaces.trinidadinternal.renderkit.htmlBasic.HtmlFormRenderer.encodeEnd(Htm
lFormRenderer.java:63)
        at javax.faces.component.UIComponentBase.encodeEnd(UIComponentBase.java:833)
        at javax.faces.component.UIComponent.encodeAll(UIComponent.java:896)
        at javax.faces.component.UIComponent.encodeAll(UIComponent.java:892)
```

Then you probably have a `<h:form>` tag inside another `<h:form>` tag, and this is not allowed.

My action is not called when clicking on a button

If an action listener is not called when clicking on a form button, then you probably have conversion or validation errors on the page. Add a `<h:messages />` tag to your page to get more details on the actual problem.

Also make sure you don't have any `<h:form>` tag inside another `<h:form>` tag.

My file is not uploaded correctly although other fields are set

Make sure your `<h:form>` tag accepts multipart content: `<h:form enctype="multipart/form-data">`.

My ajax action does not work

There could be lots of reasons for this to happen. The easiest way to find the cause is to add a `<a4j:log />` tag to your xhtml page, and then open it using CTRL+SHIFT+l. You'll get ajax debug logs in a popup window and hopefully will understand what is the problem.

Also make your you have the right namespace: `xmlns:a4j="https://ajax4jsf.dev.java.net/ajax"`.

My variable is not resolved correctly

A lot of different causes could explain with a variable is not resolved properly, but here are some recommandations to avoid known problems.

When exposing a variable for EL resolution, note that there are some reserved keywords. For instance in tags like:

```
<nxu:dataList var="action" value="#{actions}">
  ...
</nxu:dataList>
```

or

```
<nxu:inputList model="model" value="#{myList}">
...
</nxu:inputList>
```

or even:

```
<c:set var="foo" value="bar" />
```

The reserved keywords are: "application", "applicationScope", "cookie", "facesContext", "header", "headerValues", "initParam", "param", "paramValues", "request", "requestScope", "session", "sessionScope", and "view".

All values in the EL context for one of these keywords will resolve to request information instead of mapping your value.

IE throws errors like 'Out of memory at line 3156'

This is probably due to some forgotten a4j:log tag on one of your xhtml pages: IE does not play well with this tag from the RichFaces ajax library, and useful for debugging.
It usually happens when entering values on a form, and it may not happen on every page holding the tag.

Double Click Shield

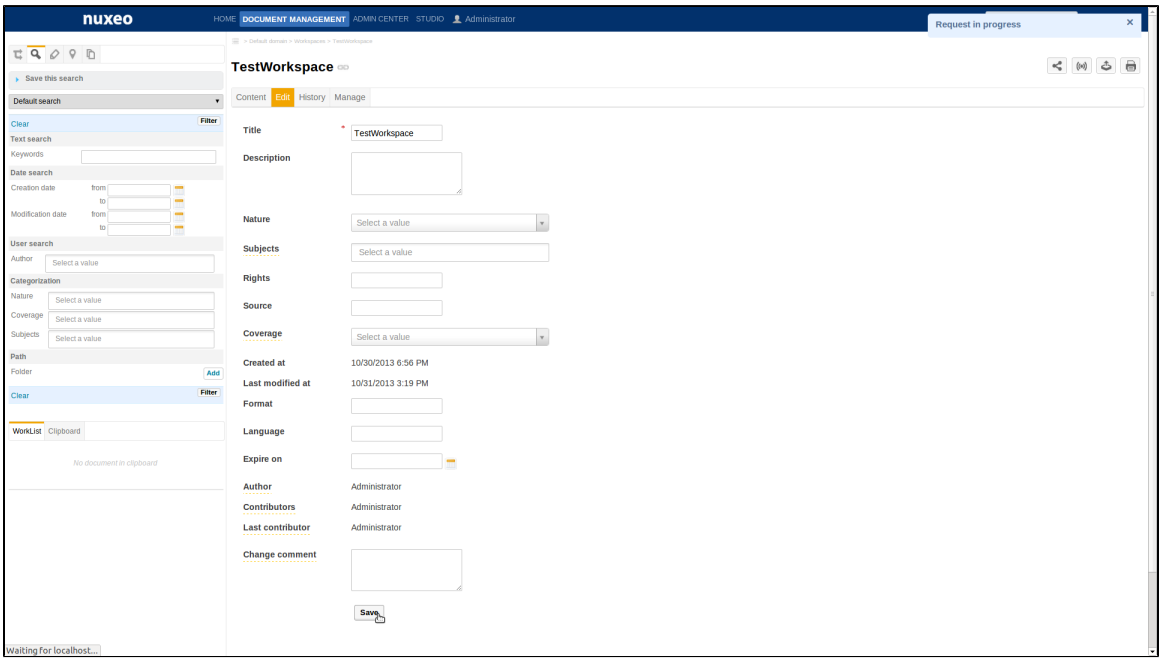
A new feature called "Double Click Shield" has been introduced in Nuxeo 5.8. It is enabled in the JSF interface and it addresses the issue described in "I Get an Error When I Click on Two Links Quickly".

The Double Click Shield prevents a form from being submitted twice. Typically, when you click many times the same button that launches a server-side action, the feature will let the first click launch the action but will prevent any additional click. This is possible thanks to a jQuery plugin which listens to the submit event of a given form.

Through a custom JSF Form Renderer, we add the dedicated CSS style class `doubleClickShielded` to all `h:form` handled by Nuxeo. When a page is loaded, the jQuery plugin will then enable the shield on all form flagged with this CSS class. (Note that the shield is not enabled on `a4j:form` since it serializes its submits).

On this page

You can see the shield in action when a "Request in progress" message is displayed at the top right corner of the window:



Limitations

The shield will intercept the submit event of a form. There are cases where a form is submitted without firing the submit event and the shield will therefore not be active.

h:commandLink vs. h:commandButton

- `h:commandLink` renders a classic HTML a link with some custom JSF JavaScript, bound to the click event, that obscurely submits the form in a uncommon way. Then submitting it does **not** fire the submit event.
- `h:commandButton` renders a HTML `input` tag with `type="submit"` which properly fires the submit event.

When designing your own template, if you'd like to protect a form from double submit, use the `h:commandButton` .

How to Locally Disable the Shield

There are use cases where the shield might break the proper functioning of a form. For instance, if you have a component that needs to perform several Ajax submits. This is the case when using `rich:fileupload` which performs as many submits as there are files to upload. With the shield enabled, only the first file will be uploaded and the others will be ignored. To address these particular use cases, you can disable the shield on a particular form by adding the `disableDoubleClickShield="true"` attribute.

Disabling Double Click Shield on a particular form

```
<h:form enctype="multipart/form-data" id="document_files_edit"
disableDoubleClickShield="true">

 <rich:fileUpload uploadData="#{FileManageActions.uploadedFiles}"
      listHeight="150" maxFilesQuantity="5"
      id="upload"
      locale="#{localeSelector.localeString}"
      immediateUpload="true" >
      <a4j:support  onsubmit="removeUploadedFile(event.memo.entry);"
event="onclear"/>
 </rich:fileUpload>

</h:form>
```

How to Globally Disable the Shield

You can disable the shield for the whole Nuxeo application by adding the following parameter in your nuxeo.conf file .

```
nuxeo.jsf.enableDoubleClickShield=false
```

Related topics

- I Get an Error When I Click on Two Links Quickly

Workflow

This page and its subpages explain all the concepts and provide a documentation on how the workflow engine works.
The Nuxeo Platform embeds a powerful workflow engine technically called "Content Routing". This workflow engine provides usual features you would expect from a workflow engine and leverages the main modules of the platform: Repository, Automation service, layouts for all the user interaction and process implementation. You need to understand correctly those concepts before playing with the workflow engine.

A workflow is conceptually defined using a graph. Workflow graphs are configured from Nuxeo Studio.

The workflow engine provides means to implement most of BPMN concepts: Fork, merge, decision, branching point, exclusive, inclusive, looping, human tasks, services tasks, multiple instances, events, data objects, subprocess, join. Note that those standard concepts are not all exposed as is on the graph editor, but can still be implemented leveraging what is provided.

The workflow engine provides high level features regarding task management such as filterable tasks lists, reminders, task reassignment, task delegation, task reminders.

Task and workflow REST endpoints are coming soon, stay tuned.

You can use this workflow engine for case management projects, for form digitization, complex documents validation, signature and publishing processes and more!

Section's content:

- Models Packaging — This page deals with how a workflow model is created on a Nuxeo instance. Note that when using Nuxeo Studio, everything is done transparently. The goal of this page is to let you understand what Studio generates in the JAR when you use the Studio feature and how the workflow is created and persisted when deploying the Studio project.
- Runtime Instantiation & Execution Logic — This page gives you technical details about the execution of workflow.
- Instance Properties — This page describes information that is persisted on a workflow instance.
- Node Properties — This page lists all the properties of a node. A workflow instance is made of several nodes linked by some transitions.
- Escalation Service — The Nuxeo workflow engine comes with an escalation service useful for having some automated evolution in the workflow graph.
- About Tasks — You'll find here what you need to know about tasks that are created via the workflow service.
- Workflow APIs — We list here all the useful available workflow APIs if you want to programmatically change a workflow instance.
- Variables Available in the Automation Context
- Workflow Naming Conventions — We provide in this page a few rules for facilitating maintenance of your workflow models.
- Useful Definitions — The main concepts that are used to design a workflow are listed below:
- How to Refresh the Task Widget on the Summary Tab — If you start workflow automatically using the Workflow > StartWorkflow operation and that your workflow's first node creates a task to the workflow initiator, you need to use in the input chain the User Interface > Refresh operation, with the value "workflowNewProcessStarted" for the event name.

Models Packaging

This page deals with how a workflow model is created on a Nuxeo instance. Note that when using Nuxeo Studio, everything is done transparently. The goal of this page is to let you understand what Studio generates in the JAR when you use the Studio feature and how the workflow is created and persisted when deploying the Studio project.

Nuxeo workflows are declared via the following contribution:

```
<extension target="org.nuxeo.ecm.platform.routing.service" point="routeModelImporter">
<template-resource id="TimeOffRequest" path="data/TimeOffRequest.zip"/>
<template-resource id="TimeOffUpdateCancelRequest"
path="data/TimeOffUpdateCancelRequest.zip"/>
</extension>
```

As you can see, the workflow definition is referenced in a zip "data/TimeOffRequest.zip" (that is in the JAR bundle). In this zip you find the serialization in core-io format of the workflow graph: the parent document corresponds to the workflow model, and you have as many children as there are nodes in your graph.

Workflow and nodes, with their properties configured from Studio, are serialized as Nuxeo documents.

Once the runtime deploys the bundle that contains such a contribution, the workflows are created from the archive, under the path `/document-route-models-root`. The workflow model is persisted as a document of type "`DocumentRoute`" and children nodes as documents of type "`RouteNode`". You can see the list of workflow models in the Admin Center (see the workflows deployed on the demo site).

Once deployed, depending on what has been set as the availability criteria for your workflow in Studio (stored on the DocumentRoute document, field `docri:availabilityFilter`) you will see the workflow in the "start workflow" drop down list on the summary of the document. You can now start a new workflow.

Runtime Instantiation & Execution Logic

This page gives you technical details about the execution of workflow.

Running a workflow

The page Models Packaging explained how the Studio JAR containing the workflow definitions is deployed on a Nuxeo instance and how the

corresponding workflow models are created.

What is happening when starting a new workflow? How is the running workflow persisted?

Running a workflow means creating a new workflow instance from one of the existing workflow models. The new instance is actually created by copy from the workflow model that is run.

So the new workflow is also a document of type DocumentRoute (and the nodes are its children of type RouteNode) with the path "`/document-route-instances-root/2013/10/15`". The date part of the path corresponds to the day the workflow instance has been created. The life cycle state for a workflow model is "validated", while the instances are "running" when created and will be "done" or "canceled", depending on how the workflow is ended. The document(s) associated at the workflow start is(are) set on the field `docri:participatingDocuments`.

The DocumentRoute document and the underlying RouteNodes are used by the workflow engine to:

- read and store the workflow and node variables,
- read the definition of the graph with the transitions between nodes,
- persist the state of the workflow.

In this section

- Running a workflow
- Runtime Execution Logic

Those DocumentRoute and RouteNode objects are useless once the workflow is achieved. They are never visible by the end user and they are removed by default. The clean-up is triggered daily at midnight.

In order to keep the completed workflows enable the following line in nuxeo.conf:

```
nuxeo.routing.disable.cleanup.workflow.instances = true
```

How are workflow transitions persisted?

Workflow transitions are stored as properties of the node document. A node has an unique id generated by Studio, set on the `rnode:nodeId` property. The list of possible transitions from a node is a complex metadata: a list of 'transition' where each 'transition' has (among others) a name, a targetId (the id of the target node) and a condition.

What are workflow tasks?

How tasks are created by the workflow is discussed in the section About tasks, but at this point it's worth mentioning that workflow tasks are also persisted as documents of type TaskDoc.

Since workflows, nodes and task are all documents, the NXQL query language can be used, like on any other document type. Check the page How to query workflow objects to find out more.

Runtime Execution Logic

Workflow nodes and tasks

Workflow nodes can be automatic steps or nodes of type task. An automatic node doesn't wait and doesn't require any external intervention to be completed. For a node of type "task", that means a task (or more) is(are) created when the workflow is executing that node and the node is paused as long as all its originating tasks are not completed.

The life cycle state of a document representing the node waiting for a task to be completed is 'suspended'.

Plug-in custom business logic

Custom business logic is added in the workflow by using automation chains at the node level. There are three entry points on the node where a chain can be added: in input, in output and on transitions. The moment when these chains are executed depends on the state of the workflow.

Execution logic

When a workflow is run, the engine starts with the node having the property 'start' set to true and after runs the nodes following, depending on the transitions in the graph.

A workflow node is run after the following algorithm:

1. Check if this is a node of type 'merge' or not.

- If true, mark the node as 'waiting' and leave it in the list of pending nodes as long as not all its incoming transitions are evaluated to true.
- If false, move to the next step.

2. Execute the "input" automation chain.
   - If the node is of type "task", create one or more tasks and mark the node as 'suspended'. The node will stay 'suspended' until its tasks are completed, then move to the next step.
   - If not, move to the next step.
3. Execute the "output" automation chain.
4. Evaluate all the conditions for all outgoing transitions from the current node.
   For all the transitions evaluated to 'true':
   a. Execute the "transition" automation chain.
   b. Add its target node to the list of pending nodes (the nodes to be run next in the workflow).

Fork node
A node of type "fork" is a node having all conditions for its outgoing transitions evaluated to 'true'.

Node of type task "Accept/Reject"
The conditions of the outgoing transitions from a node of this type depend on how the task originating from the node was completed. When the user ends the task, the id of the button the user clicked is stored on the node in a predefined property. This property is used in the conditions of the outgoing transitions and when these are evaluated: the workflow knows where to go next, depending if the user accepted or rejected the task.

All the automation chained executed by the workflow engine are executed using a temporary unrestricted session (if the current user is not an administrator, this is a session with the user "system"). The documents bound to the workflow are set as the input for these chains.

Instance Properties

This page describes information that is persisted on a workflow instance.

Workflow Variables

A workflow instance can persist some custom variables that can be used all along the workflow life.

These are properties of the document of type 'DocumentRoute' stored on a dynamic facet, in a schema named as following:

- the name of the schema (and its prefix too): `var_$WorkflowModelName`,

Usual property types are available beside complex properties that are currently disabled.

In this section

Workflow variables can be accessed from the automation context during the execution of the process using the context variable `WorkflowVariables["my_variable"]`. They can be updated using the Automation operation Set Workflow Variable.

Availability

The workflow availability information is stored on the workflow instance. This filter is actually an action filter used to control the visibility of workflow models in the list of workflows displayed by the widget type "Workflow Process" (the widget that displays the list of workflows that can be started). Note that it is not currently used at low level for controlling the right for a user to execute a workflow.

Node Properties

This page lists all the properties of a node. A workflow instance is made of several nodes linked by some transitions.

Node Variables

Node variables are persisted on the node document. These are metadata stored on a dynamic facet, in a schema named `var_$NodeId`.

Node Properties

Principle

At low level, there is only one type of node and the workflow engine behavior will change depending on the value of the properties of this node. In the Nuxeo Studio designer, you will find several kinds of node, that are actually just some sort of presets on top of the generic node.

There are more than 40 parameters that can be configured on a node. They are all stored on the "route_node" schema of the "RouteNode" document.

Identification

`nodeId`: Id of the node, that is generated from Studio.

In this section

Graph Behavior

Property	Description
`start`	Must be equal to `true` only on the node where the workflow engine should start to execute the given workflow.
`stop`	Sends a Stop workflow event. When the workflow arrives on such a node, it stops the execution of the workflow definitively.
`merge`	When this is 'true', this is a merge node: the workflow is suspended on this node, waiting for all incoming transitions to this node to be evaluated to 'true'.
`executeOnlyFirstTransition`	Follows the first transition that is equal to `true`, and only that one.
`subRouteModelExpr`	When this one is filled, the workflow engine runs the given process model on the same bound document list. This is used for the "sub workflow node" template in Studio.
`subRouteInstanceId`	The id of the instance of the subworkflow once it has been launched.
`subRouteVariables`	The variables to pass to the subworkflow when starting it.
`transitions`	There must be at least one transition. Each transition has the following :
`transition:chain`	The chain to execute when going into a transition.
`transition:condition`	Workflow engine goes into that transition only if the condition is evaluated to `true`.
`transition:name`	The name of the transition.
`transition:targetId`	The id of the target node of the transition.
`transition:label`	The displayed name of the transition.
`inputChain`	The chain executed when the workflow engine starts running the node.
`outputChain`	The chain executed when the workflow engine quits the node, before entering a "transition".

Execution History

Property	Description

`startDate`	Workflow engines stores here when it started running the node. If the node is run multiple times in a workflow (in a loop for example), it stores the last date the workflow started running this node.
`endDate`	Workflow engines store here when it goes out of the node, the last time the node is run.
`lastActor`	The last actor having done an action on the node. Useful most of the time in the output Automation chain to know who resumed the node.
`count`	The number of times the workflow engine ran this node.
`canceled`	When the workflow is canceled, all nodes are marked as canceled.
`tasksInfo`	Holds information about all tasks created by a node. For every task: the life cycle state (ended or not) , the user who ended the task, the comment if any and the id of the button the user clicked to complete the task (status).

Tasks

Property	Description
`hasTask`	If true, the workflow engine creates a task when running this node. The workflow is resumed only when this task is completed.
`hasMultipleTasks`	If true, the workflow engine creates a task for each assignee. The workflow is resumed only when all tasks created by this node are completed.
`taskDocType`	The tasks created by the workflow engine (by calling the TaskService) are documents of type "TaskDoc". You can change this if you need custom metadata on the task document (to be displayed on the task dashboard for example). Your document type must have the facets "Task" and "RoutingTask" and the "task" life cycle.
`taskDescription`	Currently not used.
`taskDirective`	The directive of the task. Should put instructions here for the user to close his task.
`taskDueDate`	The due date for the task. The workflow engine does nothing with this information by default. Can be leveraged by an escalation rule.
`taskDueDateExpr`	The task dueDate will be dynamically computed from this MVEL expression.
`taskAssignees`	The assignees for the task. Users should be prefixed by "`user:`" and groups by "`group:`"
`taskAssigneesExpr`	Assignees are dynamically computed from this MVEL expression when the workflow is run.
`taskAssigneesPermission`	Grant specific permission to the task assignees on the documents following the workflow (automatically removed by the workflow engine once the task is completed).
`allowTaskReassignment`	When true, the reassign user action is displayed to the assignee.
`taskNotificationTemplate`	The template of the email that is sent to the assignee when she is assigned a task. If no template is selected, no notification is sent. **workflowTaskAssigned** is the default mail template.
`taskLayout`	The layout (form) that is displayed to the assignee for resolving the task.
`taskButtons`	The buttons that are displayed to the assignee for solving the task.

`name`	The id of the button, can be used in the transitions condition.
`label`	The label used for the button in the UI.
`filter`	Filtering information, used in the UI too (no low level control).
`taskY`	The position of the node on the Y axis.
`taskX`	The position of the node on the X axis.

Escalation

`escalationRules`

Property	Description
`name`	The name of the escalation rule (technical id)
`multipleExecution`	When set to true, the rule is evaluated periodically, otherwise it is evaluated only once.
`condition`	The condition to know if the rule should be executed or not.
`chain`	The name of the chain that should be executed if the condition is true.

Escalation Service

The Nuxeo workflow engine comes with an escalation service useful for having some automated evolution in the workflow graph.

Principle

It is possible to set escalation rules on a given node. An escalation rule has:

- A name,
- A condition (MVEL based expression),
- An automation chain id,
- Multiple execution properties.

The escalation service periodically queries for all suspended node (a node is suspended while the engine is waiting for all tasks originating from the node to be completed). For each suspended node, it fetches all its escalation rules. It verifies the condition and if it is true, it runs the corresponding automation chain.

In this section

Example of Conditions For Escalation Rules

Remind Every Day as Soon as Task Due Date is Over

Given a `last_reminder` node variable of type Date, and making sure to update that `last_reminder` to the current date in the automation chain executed (by using **Workflow Context** > **Set Node Variable** | name: `last_reminder`, value `@{CurrentDate.calendar}`)::

```
@{!(NodeVariables["last_reminder"]==empty) &&
Context["taskDueTime"].compareTo(CurrentDate.calendar)<0&&Fn.date(NodeVariables["las
t_reminder"]).days(1).calendar.compareTo(CurrentDate.calendar)<0}
```

Remind Every 2 Min 30 Sec as Soon as the Task Has Been Assigned

Given a `last_reminder` node variable of type Date, that has been set to `CurrentDate.calendar` in the input node, and making sure to update that `last_reminder` to the current date in the automation chain executed several times by the rule:

```
@{!(NodeVariables["last_reminder"]==empty) &&
nodeStartTime.compareTo(CurrentDate.calendar)<0&&Fn.date(NodeVariables["last_reminde
r"]).seconds(150).calendar.compareTo(CurrentDate.calendar)<0}
```

In the roadmap
You can see that variables definition always have the same structure:

- How much do you add to the `last_reminder` (depending on the frequency at which you want the rule to be executed)
- From where to start: task due date, nodeStartTime, ...

==> We plan to add a Fn helper for making the expression of this kind of rule easier, and to have a "last_reminder" field part of the built-in variables.

Rules Evaluation Frequency

Rules are evaluated by default every five minutes. You can override the related scheduler contribution if you want to change it. You may want to reduce the frequency while you are doing the configuration, but then don't forget to set it back to a reasonable value when in production!

```
<extension
 target="org.nuxeo.ecm.platform.scheduler.core.service.SchedulerRegistryService"
 point="schedule">
 <!-- every 30 seconds -->
 <schedule id="escalationScheduler">
 <eventId>executeEscalationRules</eventId>
 <eventCategory>escalation</eventCategory>
 <cronExpression>0/30 * * * * ?</cronExpression>
 </schedule>
</extension>
```

About Tasks

You'll find here what you need to know about tasks that are created via the workflow service.

Dashboard

The user tasks dashboard on the Workflow tab in the Home menu is based on querying the tasks. It is possible to redefine the query and the complete content view (so as to choose the columns that are displayed for instance) by redefining the content view `user_open_tasks`.

Resolution Screen

The resolution screen is made as a "Tab", based on a grid layout. One of the pieces you may want to override on this screen is the list of metadata displayed for the bound documents. You can for this override the `task_target_documents` content view. More exactly merge a new listing layout in it.

In this section

TasksInfo

When a task is closed, the workflow engine stores useful information on the node about the task: The name of the user who solved the task, his comment and the id of the button that was clicked. Information can be accessed in the automation context via the object "`NodeVariables["tasks"]`" that returns a TasksInfoWrapper object that is a list of TaskInfo. When a node creates multiple tasks, all the resolution information of each task can be found on the `NodeVariables["tasks"]` object.

Delegation

The workflow module allows task delegation. Delegatees are stored on the task in a separate property than the assignees (`task:delegatedActors`). That way it is possible to know when someone is assigned a task directly or by delegation. The workflow engine sets the the same ACL on the bound documents for the delegatee as for the main assignee. When the task is closed, an audit entry is set on the document tracking that the user acted as a delegatee of the main assignee.

It is also possible to reassign a task. In that case the value in the "Actors" list is changed directly.

Task Type Definition

The default document type for tasks created by the workflow engine is `TaskDoc`. It is possible to use of any type of document as long as the document type has the facets `Task` and `RoutingTask and the task life cycle`. You can specify on the node properties which type of task to create.

Workflow APIs

We list here all the useful available workflow APIs if you want to programmatically change a workflow instance.

Java API

You can read the javadoc of the workflow engine main services:

- The DocumentRouting Service,
- The Task Service.

Note that most of the time provided Automation operations will do what you want to do and are easier to use.

In this section

Automation

The framework provides a few interesting Automation operation, in the category "Workflow Context".

- Start Workflow: Starts a workflow with the given `id` and to initialize its workflow variable. The document id of the created workflow instance is available under the `workflowInstanceId` context variable.
  The `id` parameter is the id of the workflow definition, as it was configured in Studio.
- Cancel Workflow: Cancels a workflow giving its id. The `id` parameter is the id of the document representing the workflow instance.
- Resume Workflow: Allows to resume a node of the workflow. It probably was suspended waiting for a task to be solved. This operation allows to force the resuming, and will let the task in a "cancelled" state.
- Complete task: Allows to close a task as if it was done via the user interface, with the ability to pass some data, as if it came from a form.
- Set Workflow Variable: Allows to set workflow variables, either from within the execution of a workflow automation chain (input, output, transition) or externally, provided the workflow instance id.
- Set Node Variable: Allows to set node variables within the execution of a workflow automation chain (input, output, transition).

REST Endpoints

Some REST Endpoints for Task and Workflow instance are coming!

Variables Available in the Automation Context

For a broader look about variables available in different contexts, have a look at the Understand Expression and Scripting Languages Used in Nuxeo page.

You need to be familiar with the concept of Automation for this section.

In the chains run on nodes (Input chain, output chain, transition and escalation chains) you have access to some data linked to the workflow being run.

- `WorkflowVariables`: A hashmap, used like this: `WorkflowVariables['my_workflow_level_variable']`
- `NodeVariables`: A hashmap, used like this: `NodeVariables['my_Node_level_variable']`
- `workflowInitiator`: The user who launched the workflow.
- `workflowStartTime`: The time when the workflow was started, useful for example when computing a due date (Eg: MVEL Due date expression: `workflowStartTime.days(8)`).
- `documents`: The documents bound to the workflow. These documents are also set as input of all executed chains.
- `button`: The button id that was clicked for processing the task, if the current node is a task node.
- `nodeId`
- `state`: The node state. A node waiting for its originating tasks to be completed is `suspended`. At the end of the workflow the node is `done`.
- `nodeStartTime`: The time when the node was started, useful for example when computing a due date (Eg: MVEL Due date expression: `nodeStartTime.days(8)`)
- `nodeEndTime`: The time when the node was ended.
- `nodeLastActor`: The last actor on the node. Useful for instance to know who closed a task when the task was assigned to a group.
- `CurrentUser.originatingUser:` All the automation operations executed by the workflow engine are executed using a temporary unrestricted session (if the current user is not an administrator, this is a session with the user "system"). This variable allows you to fetch the current user.
- `ChainParameters`: A hashmap, used like this: `ChainParameters['my_chain_parameter']`. Since 5.7.2, all chains are able to contain parameters as operation to be used from the automation context along their execution.
- `workflowInstanceId`: The id of the workflow instance
- `NodeVariables["tasks"]:` Holds information about all tasks created by a node. Is a list of objects of type TaskInfo. You can iterate on this list and fetch for every task: the life cycle state (ended or not), the user who ended the task, the comment if any, and the id of the button the user clicked to complete the task (status).
- taskDueTime: Get the due time for the current task. Returns a Java Calendar object

You can use these variables on the node configuration form Studio: in automation chains, in the fields "Due Date expression" , "Compute additional assignees" and when setting up conditions on transitions or escalation rules .

Workflow Naming Conventions

We provide in this page a few rules for facilitating maintenance of your workflow models.

In this section

Graph object names

- The workflow name is the workflow definition name as it was configured in Studio. Only alphanumeric characters, '_' and '-' are allowed. Also follow this rule when building workflows without Studio.
  Ex: Purchase Order = por
- Start transition technical name and nodes not-translated labels with this prefix. Main reason is that the transition name and node labels are directly used for translations. Buttons id doesn't need to be prefixed. Buttons label translation key should be: *workflow_prefix.node_name.button_id*
- Use *node_name*.directive for the content of the directive ex: prl_choose_particiants.directive. This improves maintenability of the translation as you better now what it matches.
- In translations, use verbs for tasks (ex: "Choose Participants"), verbs for buttons (ex: "Reject") and nouns for transitions (ex: "Validation"). This will greatly improve readability of your graph, and facilitate the flow.

Automation chains

Consider having two kinds of automaton chains.

- Automation chains that are "entry point" automation chains, bound to input, output, transition or escalation. Name those with the following convention:
  - input chain: project_prefix-workflow_prefix-node_name-input
  - output chain: project_prefix-workflow_prefix-node_name-output
  - transition chain: project_prefix-workflow_prefix-node_name-transition_name
  - escalation rule chain: project_prefix-workflow_prefix-node_name-escalation_rule_name
- Automations chains that are target to do one thing: they are focused on an objective and do nothing else. Name those with the following convention
  - project_prefix-workflow_prefix-doThis (You can even remove the workflow prefix if you think of using it in many workflows).

Usually you will reference the second kind of automation chains in the first one. That way, you will very easily add new features/ small implementation details to each node whenever you want, without having to refactor everything each time. (Note that in the future, we will want to be able to contribute several input, output, etc. … chains to avoid the first kind of operations chains.)

Workflow and versioning

A workflow instance is created by copy form the workflow model when you start a new workflow. There is no lazy loading. If you change the graph, the workflow instances that were launched before the deployment of the new workflow graph will keep on using the former graph, while new ones will use the new version of the model.

You can have some troubles if you change the behavior of a chain let's say the chain "wkf-validate-input" in such a way that the new behavior is not compatible with what expects the former graph definition. One way to handle the situation correctly is to have a suffix versioning on automation chain ids. *You duplicate the automaton chain and add the suffix _1 , _2, 3 to the new name, to be able to easily do some maintenance on both chains if necessary. So for the new version of your graph with an impacting change, you would use "wkf-validate-input_01" and you will not delete wkf-validate-input as long as some old instances are using it*.

Useful Definitions

The main concepts that are used to design a workflow are listed below:

- Node
- Transition
- Graph
- Task
- Variables
- User Task Form
- Notifications
- Automation Chains
- Queues
- Sub workflow

Additional useful notions are explained on the following pages:

- Workflow engine FAQ

Main concepts

Node

A workflow node is a “step” of the workflow model. The node holds a set of configurable properties that define what the workflow engine should do at this step. For instance, it can create a task for someone, send some notifications, update some document properties.

Transition

In the workflow terminology a transition is what links two nodes in the graph of a workflow model. A transition is always associated to a condition. When there is a transition between nodes n1 and n2, the workflow engine will check if the transition condition is evaluated to true. If yes, it will execute n2 immediately after n1 is processed.

Graph

A graph is the definition of a workflow process. It tells the server what should happen first, how execution is orchestrated, which nodes are orchestrate, etc. The graph has a start node, and one or multiple end nodes and between them a set of nodes linked by transitions.

Task

A task is a persisted object that represents what a workflow instance expects from a user or a group of users. Usually they are expected to give some information through a form and click on a button (like Accept, Reject, Confirm, Notify, Transform, etc.). A task usually has one or several assignees, a title, a directive and an expected date of achievement. A task is either to be done or done. A task is created by a node for which the property of task creation has been activated. The task assignees can be specified directly in the task definition or can be dynamically computed.

Variables

As we said before, the process, defined by its graph, is the orchestrated execution of nodes. It is very often necessary to have access to some data all along the life of the process: for example an action on a node needs some information captured earlier in the process execution. For that reason, the workflow engine offers a persistence system through the "variables". There are two types of variables:

- Node variables, defined on a node and accessible on the node scope,
- Workflow variables, defined at workflow level, and accessible from any node.

A variable has a type: string, float, blob, etc. So a variable can be a date picked up in a calendar, a file, a constrained value, …

User Task Form

A form and submission buttons can be bound to a task generated at a given node. The form helps capturing data from the user and set it on variables (workflow or local node).

Notifications

When a task is assigned to someone, you can configure the node to send a notification via email to the assignee(s), and author the content of the email yourself, including HTML formatting. Note that when a task is assigned to a group, the prefix "`group:`" must be used in front of the group name in the task assignee if you need to send a notification mail to each of the group members.

Automation Chains

The Automation module is used to express what the engine does when going through each node. Workflow engine handles several kinds of automation chain:

- Node input: executed before entering a node (and if this node is of type task, before the task is created),
- Node output: executed just after having gone through a node,
- Transition chain: executed when the workflow engine goes through a transition condition one is evaluated to true.
- Escalation rule (available since 5.7.2): executed when the system is awaiting for a user to complete an action, depending on a condition. For example when the user has not completed the task before its due date.

Queues

Depending on the implemented workflows, users may receive many tasks that they need to process. Those tasks can be grouped for presentation to the users in "queues". A queue can be functionally defined as a set of tasks assigned to a user or a group of users. Generally, a queue allows to do bulk task processing. A user can have access to several queues. Technically, we will see that queues are implemented in Nuxeo using content views.

Sub workflow

Since 5.7.2, a workflow can call another workflow and pass hard-coded or computed variables to it.

Workflow engine FAQ

- **Does Nuxeo workflow engine implement BPEL or BPMN?**
  Content Routing is not a Business Process management tool; it is designed for content-centric applications, so it would not make sense to implement BPEL. BPMN could be interesting, but Studio proposes a 100% integrated experience with the Nuxeo Platform and its tools. Implementing support of BPMN so as to define the forms with a generic designer would make experience not as good as in Studio.
- **Does Nuxeo workflow engine support multiple templates of workflows?**
  Yes, workflow models can be designed in Studio, and multiple workflows can be used at the same time in a given Nuxeo instance. It is possible to filter workflows on user properties, current document properties, ...
- **How does the workflow engine behaves with versioning of templates?**
  An instance started using a workflow template will use a frozen copy of it until the end of the process. But new instances will use the new model.
- **Does it support escalation (time based output of nodes, etc...)?**
  Since 5.7.2, escalation rules have been added to task nodes. They allow to execute an automation chain depending on a set condition, which can be time based.
- **Is there a console to manage all the workflow instances?**
  This will be included in the next release of Nuxeo. It can be added on your project very easily, as a route is persisted as a "document" in the repository.
- **Does the workflow accept notifications?**
  It is possible to configure completely customized notifications.
- **Is Content Routing for programmers only?**
  Content Routing requires a technical mind if you want to capture all its power, but can be configured very simply by non developers guys through Studio.

- **Is it complex to design a new workflow?**
  It can be very simple, depending on how much logic there is inside. Chaining nodes in Studio is very intuitive though.
- **Does the workflow engine work on a 5.5?**
  No, you need to be on a Nuxeo 5.6 version at least to fully benefit from this feature.
- **How can I disable the default existing workflow SerialDocumentReview for the documents of type File and Note?**
  It's very simple, just use the following contribution to override its availability filter:

```
<extension target="org.nuxeo.ecm.platform.actions.ActionService" point="filters">
    <filter id="filter@SerialDocumentReview" append="true">
      <rule grant="false">
        <type>File</type>
        <type>Note</type>
      </rule>
     </filter>
 </extension>
```

- **How can I enable the default workflow SerialDocumentReview for new document types?**

  Use the following contribution:

```
<extension target="org.nuxeo.ecm.platform.actions.ActionService" point="filters">
    <filter id="filter@SerialDocumentReview" append="true">
      <rule grant="true">
        <type>MyDocTYpe</type>
      </rule>
     </filter>
 </extension>
```

- **How can I have all the comments submitted by users when completing tasks logged in by the audit?**
  You just need to use a node variable called "comment" and you'll find all the comments stored in the Event log, on the "Workflow task completed" event.

- **How can I get the current user name in an operation executed by the workflow?**
  All the automation operations executed by the workflow engine are executed using a temporary unrestricted session (if the current user is not an administrator, this is a session with the user "system"). In order to fetch the current user, you have to use: `CurrentUser.originatingUser==null?CurrentUser.name:CurrentUser.originatingUser`.
- **Is the availability filter configured on the workflow also evaluated when the workflow is started using the operation "StartWorkflow"?**
  No, this filter is actually an Action Filter used to control the visibility of workflow models in the list of workflows displayed by the widget type "Workflow Process".
- **Can I have a workflow variable and a node variable with the same name?**
  No, the workflow engine doesn't handle this case. They could have the same name, but as long as they are not both used on the same node. But we recommend you to choose different names and to avoid this situation.

How to Refresh the Task Widget on the Summary Tab

If you start workflow automatically using the Workflow > StartWorkflow operation and that your workflow's first node creates a task to the workflow initiator, you need to use in the input chain the **User Interface > Refresh** operation, with the value "**`workflowNewProcessStarted`**" for the event name.

Related Documentation
• Workflow in Nuxeo Studio • Fulltext Queries • NXQL • Variables Available in the Automation Context • Workflow

WebEngine (JAX-RS)

Nuxeo WebEngine is a web framework for JAX-RS applications based on a Nuxeo document repository. It provides a template mechanism to facilitate the dynamic generation of web views for JAX-RS resources.

Templating is based on the Freemarker template engine.

WebEngine also provides support for writing JAX-RS resources using the Groovy language, or other scripting languages supported by the `javax.scripting` infrastructure.

Besides templating support, WebEngine provides an easy way to integrate native JAX-RS application on top of the Nuxeo platform - it provides all the glue and logic to deploy JAX-RS application on a Nuxeo server.

We will describe here the steps required to register a JAX-RS application, and how to use WebEngine templating to provide Web views for your resources.

This tutorial assumes you are already familiarized with JAX-RS. If not, please read first a JAX-RS tutorial or the specifications. For beginners, you can find a JAX-RS tutorial here.

This page presents the WebEngine concepts. For more details about using these concepts, see the WebEngine Tutorials. You can download the tutorials sources from examples.zip.

Outline of this chapter:

Quick checklist

These are the key points to keep in mind to have a running simple WebEngine application. They will be detailed further in this page.

- make your class extend `ModuleRoot`,
- annotate your class with `@WebObject`,
- define a `module.xml` referencing the type in the above annotation,
- have `maven-apt-plugin` in the pom.xml (and use `mvn install` to run it).

Declaring a JAX-RS Application in Nuxeo

To deploy your JAX-RS application in Nuxeo you should create a JAX-RS application class (see specifications) and declare it inside the `MANIFEST.MF` file of your Nuxeo bundle.

To define a JAX-RS application, you must write a Java (or Groovy) class that extends the `javax.ws.rs.core.Application` abstract class.

Then, you need to declare your application in your bundle `MANIFEST.MF` file as following:

```
Nuxeo-WebModule: org.MyApplicationClass
```

where `org.MyApplicationClass` is the full name of your JAX-RS application class.

Now you simply put your JAR in Nuxeo bundles directory (e.g. `nuxeo.ear/system` under JBoss) and your Web Application will be deployed under the URL: http://localhost:8080/nuxeo/site.

Example

Let's define a JAX-RS application as follows:

```
public class MyWebApp extends Application {
    @Override
    public Set<Class<?>> getClasses() {
        HashSet<Class<?>> result = new HashSet<Class<?>>();
        result.add(MyWebAppRoot.class);
        return result;
    }
}
```

where the `MyWebAppRoot` class is the entry point of the application and is implemented as follows:

```
@Path("mysite")
public class MyWebAppRoot {
    @GET
    public Object doGet() {
        return "Hello World!";
    }
}
```

Lets say the full name of `MyWebApp` is `org.nuxeo.example.MyWebApp`. Now you should tell to Nuxeo WebEngine that you have a JAX-RS application n your bundle. To do this, add a line to your `MANIFEST.MF` file as follows:

```
Manifest-Version: 1.0
...
Nuxeo-WebModule: org.nuxeo.example.MyWebApp
...
```

Build your application JAR and put it into your Nuxeo bundles directory. After starting the server you will have a new web page available at http://localhost:8080/nuxeo/site/mysite.

Under the Jetty core server distribution (which is a development distribution), the URL of your application will be http://localhost:8080/m

ysite

Automatic discovery of JAX-RS resources at runtime

If you don't want to explicitly declares your resources in a JAX-RS application object you can use a special application that will discover resources at runtime when it will be registered by the JAX-RS container.
For this you should use the `org.nuxeo.ecm.webengine.jaxrs.scan.DynamicApplicationFactory` application in your manifest like this:

```
Nuxeo-WebModule: org.nuxeo.ecm.webengine.jaxrs.scan.DynamicApplicationFactory
```

When your JAX-RS module will be registered the bundle declaring the module will be scanned and any class annotated with `@Provider` or `@Path` will be added to the module.
If you want to avoid scanning the entire bundle you can use the attribute `package` to perform scanning only inside the given package (and on its sub-packages). Example:

```
Nuxeo-WebModule:
org.nuxeo.ecm.webengine.jaxrs.scan.DynamicApplicationFactory;package=org/my/root/package
```

The JAX-RS container used by Nuxeo is Jersey.

Declaring a WebEngine Application in Nuxeo

To declare a WebEngine Application you should create a new JAX-RS Application as in the previous section - but using the `org.nuxeo.ecm.webengine.app.WebEngineModule` base class and declare any Web Object Types used inside your application (or use runtime type discovery). You will learn more about Web Object Types in the following sections.

The simplest way to declare a WebEngine module is to add a line like the following one in your manifest:

```
Nuxeo-WebModule:
org.nuxeo.ecm.webengine.app.WebEngineModule;name=myWebApp[;extends=base;package=org/mywebapp]
```

the **name** attribute is mandatory, **extends** and **package** are optional and are explained above.
When declaring in that way a WebEngine module all the Web engine types and JAX-RS resources will be discovered at runtime - at each startup.

A WebEngine Application is a regular JAX-RS application plus an object model to help creating Nuxeo web front ends using Freemarker as the templating system.

Compatibility Note
This way of declaring WebEngine Applications is new - used since Nuxeo 5.4.2 versions. The old declaration mode through `module.xml` (and web types discovery at build time) is still supported but it is deprecated. See WebEngine Modules for more details.

If you'd like your WEB module to be deployed in a distinct JAX-RS application than the default one (handling all webengine modules), you need to decare a host:

```
Nuxeo-WebModule: org.nuxeo.ecm.webengine.app.WebEngineModule;host=MyHost
```

and bind the servlet to this host in the deployment-fragment.xml file:

```
<extension target="web#SERVLET">
  <servlet>
    <servlet-name>My Application Servlet</servlet-name>
    <servlet-class>
      org.nuxeo.ecm.webengine.app.jersey.WebEngineServlet
    </servlet-class>
    <init-param>
      <param-name>application.name</param-name>
      <param-value>MyHost</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>My Application Servlet</servlet-name>
    <url-pattern>/site/myapp/*</url-pattern>
  </servlet-mapping>
</extension>
```

This will isolate your top level resources from the ones declared by other modules (like root resources, message body writer and readers, exception mapper etc.
Example: this will make it possible, for instance, to use a distinct exception mapper in your application.

Upgrading your old-style WebEngine module to work on 5.4.2

Before 5.4.2 you were able to declare a WebEngine module only by using a module.xml file. If you have such a module and want to upgrade it to run under >= 5.4.2 versions you should:

- Add the following header in the MANIFEST.MF file of your plugin:

```
Nuxeo-WebModule: org.nuxeo.ecm.webengine.app.WebEngineModule
```

- Update your root object (the one referenced in module.xml as root-type="...") and add a @Path annotation on the class that specify the module path (as specified in module.xml as path="...")

This is the easiest way to upgrade an existing module - but it is only a compatibility mode introduced to ease the upgrade.

If you want to use the new features provided by the new way of declaring WebEngine modules you should refactor your module and define a JAX-RS application as described in the example below.
The benefit of this approach is that you can have any number of root objects in the same WebEngine module, and also you don't need to reference them in module.xml (i.e. module attributes are no more needed if you take this approach).
This provides a standard way of declaring your JAX-RS root resources - and also add strong typing for your JAX-RS root resources.

Example

To define a WebEngine Application you should override the `org.nuxeo.ecm.webengine.app.WebEngineModule` class and declare any web types you are providing:

```
public class AdminApp extends WebEngineModule {

    @Override
    public Class<?>[] getWebTypes() {
        return new Class<?>[] { Main.class, User.class, Group.class,
                UserService.class, EngineService.class, Shell.class };
    }

}
```

If you want automatic discovery of resources you can just use the WebEngineModule in your MANIFEST.MF - without extending it with

your own application class.

Of course as for JAX-RS applications you should specify a Manifest header to declare your application like:

```
Nuxeo-WebModule: org.nuxeo.ecm.webengine.admin.AdminApp;name=admin;extends=base
```

You can see there are some additional attributes in the manifets header: 'name' for the module name and 'extends' if you want to extend another module. The 'name' attribute is mandatory. You can also optionally use the 'headless=true' attribute to avoid displaying your module in the module list on the root index.

If you want to customize how your module is listed in that module index you can define 'shortcuts' in the module.xml file. Like this:

```
<?xml version="1.0"?>
<module>
  <shortcuts>
    <shortcut href="/admin">
      <title>Administration</title>
    </shortcut>
    <shortcut href="/shell">
      <title>Shell</title>
    </shortcut>
  </shortcuts>
</module>
```

Note that the module.xml file is optional. You can use it if you want to make some customization like adding shortcuts or defining links.

To define a WebEngine Application root resource you should override the `org.nuxeo.ecm.webengine.model.impl.ModuleRoot` class:

```
@WebObject(type = "Admin", administrator = Access.GRANT)
@Produces("text/html;charset=UTF-8")
@Path("/admin")
public class Main extends ModuleRoot {

    public Main(@Context UriInfo info, @Context HttpHeaders headers) {
        super(info, headers, "Admin");
    }

    ...

}
```

What is WebEngine good for?

We've seen that using WebEngine you can deploy your JAX-RS applications without many trouble. You don't need to care about servlet declaration etc. You simply need to declare your JAX-RS application class in the MANIFEST file. The JAX-RS servlet provided by WebEngine will be used to invoke your application when its URL is hit.

So for now, we've seen how to create a JAX-RS application and deploy it into a Nuxeo server. You can stop here if you just want to use JAX-RS and don't care about WebEngine templating and Web Views for your resources.

JAX-RS is a very good solution to build REST applications and even Web Sites. The problem is that JAX-RS focus more on REST applications and doesn't define a flexible way to build modular Web Sites on top of the JAX-RS resources.

This is where **WebEngine** is helping by providing Web Views for your JAX-RS resources.

I will first explain how you can do templating (using Freemarker) for a regular JAX-RS resource. Then I will enter deeper into the WebEngine templating model.

JAX-RS Resource Templating

To create a Freemarker template for your JAX-RS resource you need to put the template file (a Freemarker template like `index.ftl`) in your bundle so the template could be located using the Java class loader at runtime.

Example

Put a file with the following content in the `src/main/resources/org/nuxeo/example/index.ftl`:

```
Hello ${Context.principal.name}!
```

And then modify the `MyWebAppRoot` class as following:

```
public class MyWebAppRoot {
    @GET
    public Object doGet() {
        return new TemplateView(this, "index.ftl");
    }
}
```

If your are logged as Guest and you go to http://localhost:8080/nuxeo/site/mysite you will see a message like: *Hello Guest!*

In WebEngine if you doesn't sign in as a real user you will be automatically considered a *Guest* user.

WebEngine Template Variables

Here is the list of all variables available in a template file:

- `Context` - the context object; see `org.nuxeo.ecm.webengine.model.WebContext` for the provided API.
- `This` - the target JAX-RS resource. (the object that returned the template)
- `Root` - the first JAX-RS resource in the call chain. (the first JAX-RS resources that delegate the call to the leaf resource). This variable **is not available** for pure JAX-RS resources. You should use WebEngine objects to have it defined.
- `Session` - the Nuxeo repository session. The session is always non null when the JAX-RS application is installed into a Nuxeo server.
- `Document` - this object is equivalent to **This** when the current JAX-RS resource is wrapping a Nuxeo Document. See `org.nuxeo.ecm.platform.rendering.fm.adapters.DocumentTemplate` for the provided API. This variable is not set when using pure JAX-RS resources. You should use WebEngine objects to have it defined.
- `Adapter` - the current WebEngine adapter - only set when using WebEngine objects and the current JAX-RS resource is an adapter.
- `Module` - *deprecated* - this is the module instance (the root object) and is provided only for compatibility with previous WebEngine implementations.
- `Engine` - this is the singleton WebEngine service; see the `org.nuxeo.ecm.webengine.WebEngine` interface for the provided API.
- `basePath` - the `contextPath+"/"+servletPath` (see `javax.servlet` specifications)
- `contextPath` - *deprecated* - special variable that identify the context path set using the runtime variable `org.nuxeo.ecm.contextPath`. Tis is useful for proxy redirections. See **WebEngine Resources** section for how to locate resources.
- `skinPath` - *deprecated* - represent the path to the WebEngine module resources. Should no more be used since it is not safe when rewriting requests using a proxy HTTP server. See **WebEngine Resources** section for how to locate resources.

You notice that when using pure JAX-RS objects you only have the following built-in variables defined in the template context: `This`, `Context`, `Engine`, `basePath`, `contextPath`.

Custom Template Variables

You can add your custom variables to the template context as follows:

```
public class MyWebAppRoot {
    @GET
    public Object doGet() {
        return new TemplateView(this, "index.ftl").arg("country",
"France").arg("city", "Paris");
    }
}
```

You can now write a template file named `index.ftl` that uses these variables to render the response:

```
Hello ${Context.principal.name}! Your country is ${country}, and your city is ${city}.
```

WebEngine Modules

The module declaration documented here is deprecated for Java based modules since Nuxeo 5.4.2 versions. See Declaring a WebEngine Application in Nuxeo for the new declaration. The `module.xml` file is now optional - and should be used to declare module shortcuts or to describe links. Groovy based modules still use the old declaration method.

The problem with the templating described above is that template files are inside the application JAR and are located using the class loader. This make difficult to create web sites that are composed from tens of template files and images. A more flexible approach will be to put web files into directories on the file system. This way, the lookup is faster and you can easily modify web files without restarting the server when in development mode.

This is one of the reason the WebEngine module concept was introduced.

A module is a bundle (i.e. JAR file) that contains JAX-RS resources and web resources (such as images, HTML files or templates). The module is usually defining a JAX-RS application but it can also contribute resources to other applications. So a module is defined by:

- a module name - a unique key used to identify the module in the module registry.
- a module path - the path of the root resource in a module.
- a module entry point - a JAX-RS resource class that will be served when the module path matches a client request. The module entry point is used to directly send responses or to dispatch the request to other JAX-RS resources.

To define a module you need to create a `module.xml` file and put it in the root of your JAR. Here is the minimal content of a `module.xml` file:

```
<module name="Admin" root-type="Admin" path="/admin" />
```

This module file is declaring a module named `Admin` with path `/admin`. The module path is relative to the WebEngine servlet so the full URL of this module will be http://localhost:8080/nuxeo/site/admin.

You notice there is a third required attribute `root-type`. This attribute is used to locate the module entry point.

How the entry point is located will be discussed in the next section.

WebEngine Objects

A WebEngine module is made from web resources and web objects. Resources are usually HTML, JavaScript, CSS, images or template files and are used to create web **views** for the JAX-RS objects provided by the module.

To be able to bind views to your JAX-RS resources you must declare them as WebEngine objects. This is done by using the annotation: `@WebObject` and extending the `org.nuxeo.ecm.webengine.model.impl.DefaultObject` class. Example:

```
@WebObject(type = "User")
@Produces("text/html;charset=UTF-8")
public class User extends DefaultObject {

    @GET
    public Object doGet() {
        return getView("index").arg("user", principal);
    }

    ...
```

In the previous example we defined a WebObject of type `User`. You notice the object is a JAX-RS resource and extends the `DefaultObject` base class. The `@WebObject` annotation is used to declare the JAX-RS resource as a WebObject.

There is a special WebObject - the entry point of a module. To define a module entry point you need to create a WebObject that extends the `org.nuxeo.ecm.webengine.model.impl.ModuleRoot.ModuleRoot` class. Example:

```
@WebObject(type = "Admin", administrator=Access.GRANT)
@Produces("text/html;charset=UTF-8")
public class Main extends ModuleRoot {

    @Path("users")
    public Object getUserManagement() {
        return newObject("UserManager");
    }

    @Path("engine")
    public Object getEngine() {
        return newObject("Engine");
    }

    ...
}
```

As we've seen above when a module is loaded the entry point class is located using the `root-type` attribute in module.xml. This attribute is pointing to the WebObject having the same type as the `root_type` value. So in our case the `root-type="Admin"` attribute is telling to WebEngine to use the the class `Main` annotated with `@WebObject(type = "Admin")` as the entry point JAX-RS resource.

In the example above we can see that WebObjects methods annotated with `@GET`, `@POST` etc. are used to return the response to the client. The right method is selected depending on the HTTP method that were used to make the request. `@GET` methods are used to serve GET requests, `@POST` methods are used to serve POST requests, etc. So the method:

```
@GET
    public Object doGet() {
        return getView("index").arg("user", principal);
    }
```

will return a view (i.e. template) named "index" for the current object. The returned view will be processed and serialized as a HTML document and sent to the client.

We will see in next section how view templates are located on the file system.

The method:

```
@Path("users")
    public Object getUserManagement() {
        return newObject("UserManager");
    }
```

delegates the request to the WebObject having the type `UserManager`. This Web Object is a JAX-RS resource annotated with `@WebObject(type="UserManager")`.

WebEngine Adapters

A WebAdapter is a special kind of Web Object that can be used to extend other Web Objects with new functionalities. To extend existing objects using adapters you don't need to modify the extended object. This type of resource make the life easier when you need to add more API on an existing object but cannot modify it because for example it may be a third party web object or the new API is too specialized to be put directly on the object. In this cases you can create web adapters that adapts the target object to a new API.

To declare an adapter use the `@WebAdapter` annotation and extend the `DefaultAdapter` class:

```
@WebAdapter(name = "audits", type = "AuditService", targetType = "Document")
public class AuditService extends DefaultAdapter {
...
}
```

An adapter has a `name` that will be used to select the adapter depending on the request path, and as any Web Object a type. An adapter also has a `targetType` that represent the type name of the object to adapt.

See more on using adapters in Adapter Tutorial.

@Path and HTTP method annotations.

Lets discuss now how JAX-RS annotations are used to match requests.

If a method is annotated using one of the HTTP method annotations (i.e. `@GET`, `@POST`, `@PUT`, `@DELETE`, etc.) then it will be invoked when the current object path matches the actual path in the user request. These methods **must** return the object that will be used as the response to be sent to the client. Regular Java objects as String, Integer etc. are automatically serialized by the JAX-RS engine and sent to the client. If you return other type of objects you must provide a writer that will handle the object serialization. See more about this in JAX-RS specifications.

Methods that are annotated with both `@Path` and one of the HTTP method annotations are uses tin the same manner as the ones without a `@Path` annotation. The `@Path` annotation can be added if you want to match a sub-path of the current object path. `@Path` annotations may contain regular expression patterns that should be enclosed in brackets {}.

For example, let's say the following object matches the `/nuxeo/site/users` path:

```
@WebObject(type = "Users")
@Produces("text/html;charset=UTF-8")
public class Users extends DefaultObject {

    @GET
    public Object doGet() {
        return getView("index");
    }

    @Path("\{name\}")
    @GET
    public Object getUser(@PathParam("name") String username) {
        return getView("user").arg("user", username);
    }
    ...
```

The `doGet` method will be invoked when a request is exactly matching the `/nuxeo/site/users` path and the HTTP method that was used is

GET.

The `getUser` method will be invoked when a request is matching the path `/nuxeo/site/users/{name}` where `{name}` matches any path segment. So all requests on paths like `/nuxeo/site/users/foo`, `/nuxeo/site/users/bar` will match all the `getUser` method. Because the path contains a pattern variable, you can use the `@PathParam` annotation to inject the actual value of that variable into the method argument.

You can also use a `@Path` annotation to redirect calls to another JAX-RS resource. If you want this then you **must not** use use any HTTP method annotations in conjunction with `@Path` - otherwise the method will be treated as a terminal method that is returning the response object to be sent to the client.

Example:

```
@WebObject(type = "Users")
@Produces("text/html;charset=UTF-8")
public class Users extends DefaultObject {

    @Path("name")
    public Object getUser(@PathParam("name") String username) {
        return new User(username);
    }
    ...
```

You can see in the example above the if the request matches a path like `/nuxeo/site/users/{name}` then the `Users` resource will dispatch the call to another JAX-RS resource (i.e. `User` object) that will be used to handle the response to the user (or to dispatch further the handling to other JAX-RS resources).

Dispatching requests to WebObject sub-resources.

To dispatch the call to another WebObject instance you must use the `newObject(String type)` method to instantiate the WebObject by specifying its type as the argument. Example:

```
@Path("users")
    public Object getUserManagement() {
        return newObject("UserManager");
    }
```

This method will dispatch the requests to the `UserManager` WebObject. The WebObject class is identified using the type value: "UserManager".

Module deployment

At server startup JARs containing `module.xml` files will be unzipped under `install_dir/web/root.war/modules/module_dir` directory. The `module_dir` name is the bundle symbolic ID of the unzipped JAR. This way you can find easily which bundle deployed which module.

The `install_dir` is the installation directory for a standalone installation or the `jboss/server/default/data/NXRuntime` for a JBoss installation.

To deploy a module as a directory and not as an OSGi bundle you can simply copy the module directory into `install_dir/web/root.war/deploy`. If deploy directory doesn't exists you can create it.

Note that when you deploy the module as an OSGi bundle the JAR will be unzipped only at first startup. If you update the JAR (the last modified time changes) then the JAR will be unzipped again and will override any existing files.

Module structure

A web module can be deployed as a JAR file or as a directory. When deployed as a JAR file it will be unzipped in a directory at first startup.

The structure of the root of a deployed web module should follows this schema:

```
/module.xml
 /i18n
 /skin
 /skin/resources
 /skin/views
 /META-INF
```

Every module must have a `module.xml` descriptor in its root. This file is used by WebEngine to detect which bundles should be deployed as web modules.

- The `/i18n` directory contains message bundle property files.
- The `/skin` directory should contain the templates used to generate web pages and all the client resources (e.g. images, style sheets, client side scripts). The content of this directory is inherited from the super module if your module extend another module. This means if a resource is not found in your module skin directory the super module will be asked for that resource and so on until the resource is found or no more super modules exists.
- The `/skin/resources` directories contains all client resources. Here you should put any image, style sheet or script you want to use on the client. The content of this directory is directly visible in your web server under the path: `{base_path}/module_name/skin` (see **Static Resources**).
- The `/skin/views` directory should be used to store object views. An object view is usually a Freemarker template file that will be rendered in the request context and served when necessarily by the web object.
- The `/META-INF`} directory is usually storing the `MANIFEST.MF` and other configuration or generated files that are internally used by the server.

⚠ Be aware that the root directory of a module is added to the WebEngine classpath.

For that reason module classes or Groovy scripts must be put in well named packages and the package root must reside directly under the the module root. Also, avoid to put classes in script directory.

Look into an existing WebEngine module like admin, base or wiki for examples on how to classes are structured.

Web Object Views

We saw in the examples above that WebObjects can return views as a response to the client. Views are in fact template files bound to the object. To return a view from a WebObject you should do something like:

```
@GET
    public Object doGet() {
        return getView("my_view");
    }
```

where `my_view` is the view name. To bind a view to an object, you should put the view file in a folder named as the object type in the `/skin/views` directory. The view file name should have the same name as the view + the `.ftl` extension.

Example: Suppose we have an web object of type `MyObject`. To define a view named `myview` for this object, you should pit the view template file into `/skin/views/MyObject/myview.ftl`. Doing this you can now use send the view to the client using a method like:

```
@WebObject(type = "MyObject")
@Produces("text/html;charset=UTF-8")
public class MyObject extends DefaultObject {
    @GET
    public Object doGet() {
        return getView("myview");
    }
}
```

If a view file is not found inside the module directory then all super types are asked for that view. If the view is not found then the super modules (if any) are asked for that view.

Extending Web Objects

You can extend an existing Web Object with your own object by defining the `superType} attribute in the {{@WebObject` annotation. Example:

```
@WebObject(type = "JSONDocument", superType = "Document")
```

When extending an object you inherit all object views.

Extending Modules

When defining a new module you can extend existing modules by using the `extends` attribute in your module.xml file:

```
<module name="Admin" root-type="Admin" path="/admin" extends="base" />
```

The `extends` attribute is pointing to another module name that will become the base of the new module.

All WebObjects from a base module are visible in the derived modules (you can instantiate them using `newObject("object_type") method)`.

Also, static resources and templates that are not found in a module will be searched into the base module if any until the resource is found or all module hierarchy is consumed.

You can use this feature to create a base module that is providing a common look and feel for your applications and then extending it in modules by overriding resources (or web page areas - see **Template Model**) that you need to change for your application.

Template Model

WebEngine defines also a template model that is used to build responses. The template engine is based on FreeMarker, plus some custom extensions like template blocks. Using blocks you can build your web site in a modular fashion. Blocks are dynamic template parts that can be extended or replaced using derived blocks. Using blocks, you can write a base template that may define the site layout (using blocks containing empty or generic content) and then write final skins for your layout by extending the base template and redefining blocks you are interested in.

Templates are stored as files in the module bundle under the skin directory. Templates are resolved in the context of the current module. This way, if a module is extending another module, a template will be first looked up into the derived module, then in its super modules until a template it's found or no more parent modules exists.

There is a special type of templates that we call views. The difference between views and regular templates is that views are always attached to an Web Object Resource. This means, views are inherited from super types. Because of this the view file resolution is a bit different from templates.

Views are first searched in the current module, by iterating over all resource super types. If not found then the super module is searched (if any) and so on until a view file is found or no more parent modules exists.

Static resources

Template files are usually referencing external resources like static CSS, JavaScript or image files.

To refer to this type of resources you **must always** use relative paths to the module root (the entry point object). By doing this you avoid problems generated by URL rewrite when putting your server behind a proxy.

Lets suppose we have a module entry point as follows:

```
@WebObject(type = "Admin", administrator=Access.GRANT)
@Produces("text/html;charset=UTF-8")
public class Main extends ModuleRoot {

    public Object doGet() {
        return getView("index");
    }

    ...
}
```

Suppose the object is bound to the `nuxeo/site/admin` path, and the `index.ftl` view is referencing an image located in the module directory in `skin/resources/images/myimage.jpg`. Then the image should be referenced using the following path:

```
<img src="skin/images/myimage.jpg" />
```

the path is relative to the current object so the image absolute path is `/nuxeo/site/admin/skin/images/myimage.jpg`.

So all static resources in a Web module are accessible under the `/module_path/skin/` path. You can get the module path from any view by using the variable `${Root.path}` or `${basePath}/module_name`.

Headless modules

By default WebEngine modules are listed in the WebEngine home page (i.e. `/nuxeo/site`). If you don't want to include your module in that list you can use the `headless` attribute in your `module.xml` file:

```
<module name="my_module" root-type="TheRoot" path="/my_module" extends="base"
headless="true" />
```

Groovy Modules

You can write your Web Objects in Groovy and put object sources in the module directory.

Groovy support is not enabled by default. If you want to define web objects using Groovy you need to enable the 'groovy' nature on the module. This is an example on how to do this:

```
<module name="my_module" root-type="TheRoot" path="/my_module" >
  <nature>groovy</nature>
  ...
</module>
```

Building WebEngine projects

WebEngine is using types to classify web objects. When a request will be resolved to an web object the object type is retrieved first and asked to instantiate a new object of that type which will be used to handle the request. This means all types in a module and in super modules must be known after a module is loaded. Types are declared using annotations so detecting types at runtime may be costly. For this reason types are discovered at build time and written to a file in the `META-INF` directory of the module.

```
/META-INF/web-types
```

To have this file generated you must use a correct maven `pom.xml` for your project that will load a custom 'java apt' processor during the build. See WebEngine pom files for how to write your pom files. Here is the declaration of the apt plugin that you need to put in the your WebEngine module project:

```
<build>
  <plugins>
  ...
      <!-- APT plugin for annotation preprocessing -->
      <plugin>
        <groupId>org.apache.myfaces.tobago</groupId>
        <artifactId>maven-apt-plugin</artifactId>
        <executions>
          <execution>
            <id>generate-bindings</id>
            <goals>
              <goal>execute</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
      <!-- end APT plugin -->
  ...
  </plugins>
</build>
```

Also, it is recommended to use `mvn clean install` to build the JAR of your module - this way you force the type file to be generated again.

Default WebEngine Applications

This page is a work in progress. See NXDOC-229 for details.

URLs exposed by WebEngine module are of the form `/nuxeo/site/*` (where * is a service offered by a WebEngine module):

- `/nuxeo/site`: root page listing the "available WebEngine applications"
- `/nuxeo/site/admin`: simple WebEngine UI to access administrative features
- `/nuxeo/site/automation`: base URL for Automation services, documentation is available at `/nuxeo/site/automation/doc`
- `/nuxeo/site/gadgets`: base URL for gadgets Rest requests
- `/nuxeo/site/dav`: WebDAV service
- `/nuxeo/site/connectClient`: URL to request Nuxeo Connect services (packages lists, registration ...)
- `/nuxeo/site/shell`: Nuxeo Shell applet
- `/nuxeo/site/layout-manager`: Layouts and Widgets (Forms, Listings, Grids) services and documentation

Depending on the addons you've installed, you could also expose these URLs:

- /`nuxeo/site/mobile`: URL for the mobile application
- `/nuxeo/site/sites`: WebEngine front-end for Site (mini-site) documents
- `/nuxeo/site/blogs`: WebEngine front-end for Blog documents

Related topics

- WebEngine (JAX-RS)
- Navigation URLs

Session and Transaction Management

Transaction Management

By default WebEngine will automatically start a transaction for any request to a JAX-RS application (or WebEngine module). The default locations of static resources are omitted (so that no transaction will be started when requesting a static resource). The static resources locations is `*/skin/*` — this will match any path that targets a resource inside a `skin` directory in a JAX-RS application.

You may want to adapt the transaction timeout per-request. In that case, you should specify in the HTTP headers the timeout value in seconds using the header `Nuxeo-Transaction-Timeout`.

On this page

Session Management

WebEngine provides a managed `CoreSession` to any JAX-RS resource that wants to connect to the repository. WebEngine will close the managed `CoreSession` when no more needed (at the end of the request) so you should not worry about leaks. This session can be used either in a JAX-RS resource method, or in JAX-RS a `MessageBodyReader` or `MessageBodyWriter`.

To get the managed `CoreSession` from a JAX-RS resource you can use the following code:

```
UserSession.getCurrentSession(httpRequest);
```

If you don't have access to the current HTTP request object you can use this code (in that case a ThreadLocal variable will be used to retrieve the UserSession):

```
WebEngine.getActiveContext().getUserSession();
```

Then using the `UserSession` object you can get either the current `Principal` or a `CoreSession`:

```
UserSession userSession = WebEngine.getActiveContext().getUserSession();
Principal principal = userSession.getPrincipal();
CoreSession session1 = userSession.getCoreSession();
CoreSession session2 = userSession.getCoreSession("myrepo");
```

When calling the `getCoreSession()` method and no managed `CoreSession` was yet created for the target repository then a new `CoreSession` is created and returned. If a `CoreSession` already exists then it is returned.

You can see that there are two flavors of `getCoreSession()` method:

- `getCoreSession()`
- `getCoreSession(String repositoryName)`

The first one is returning a session for the default repository. The second one will return a session for the given repository. By default the `getCoreSession()` method will use the default repository as configured in Nuxeo server, but you can change the repository that will be used on a request basis. See next section for how to change the default repository used by this method.

Not that the `UserSession` object is available only in the context of a WebEngine request (*i.e.*, inside JAX-RS applications or WebEngine modules).

Selecting the Default Repository

You can choose from the client side which will be the repository used to create a managed `CoreSession`. To do this you can either use an HTTP request header:

```
X-NXRepository: myrepo
```

or a request parameter:

```
nxrepository=myrepo
```

If not specified the default repository defined by the Nuxeo server will be used.

Cleaning Up at the End of the Request

Some JAX-RS resources will need to create temporary files or open other system resources that cannot be removed in the JAX-RS method because they are used by a `MessageBodyWriter`. In that case you can register a cleanup handler that will be invoked at the request end (after all JAX-RS objects finished their work and response was sent to the servlet output stream).

To register a cleanup handler you can do the following:

```
UserSession.addRequestCleanupHandler(httpRequest, new RequestCleanupHandler() {
    @Override
    public void cleanup(HttpServletRequest httpRequest) {
        ...
    }
});
```

The `cleanup` method will be invoked after the request is processed and the response created.

Configuring Transaction Management on a Path Basis

You can also configure how the transaction is managed on a subset of resources in your JAX-RS application. To do this you can contribute an extension as follows:

```
<extension target="org.nuxeo.ecm.webengine.WebEngineComponent"
point="request-configuration">
  <path value="/mymodule1" autoTx="false" />
  <path value="/mymodule2/resources" autoTx="true" />
  <path value="/mymodule3/.*\.gif" autoTx="false" regex="true"/>
</extension>
```

The first rule says that for any resource which path begins with "/mymodule1" then automatic transaction is off.
The second one says that for any resource which path begins with "/mymodule2/resources" then automatic transaction is true.
The third rule says that for any .gif file inside "/mymodule3" automatic transaction is off.

By default the `value` attribute represent a path prefix. If you want to use a regular expression you must specify `regex="true"`.

The recommended way to define rules is to use prefixes and not regular expression.

Path Rule Matching

All the contributed path matching rules will be ordered from the longest path to the shortest one in lexicographical order. Regular expression rules will be always put after the prefix based rules (i.e. prefix based rules are *stronger*). Then to find the best matching rule, the path rules are iterated until a match occurs.

Paths specified in a rule must begin with a `/` (if not, a `/` will be automatically added by WebEngine). These paths are matched against the `HttpServletRequest` path info (which will always begin with a `/`).

For example, given the following path matching rule contributions:

```
/a/b/d/.*\.gif
/a
/a/b/c
/b
/b/c
```

they will be ordered as follow:

```
/a/b/c
/b/c
/a
/b
/a/b/d/.*\.gif
```

Therefore for the URL path `/a/b` the first match will be `/a,` and this rule will be used to define the transaction management for this resource.

WebEngine Tutorials

In this section we will go deeper into WebEngine model by proposing 5 samples on how to use common WebEngine concepts.

To install the sample modules you need to download the compiled jar and copy them s to a `<NXSERVER>/plugins` directory. `<NXSERVER>` is the `jboss/server/default/deploy/nuxeo.ear` directory on JBoss distribution or the root `nxserver` directory on other distributions.

To correctly understand the tutorials you need to look into all .java and .ftl files you find in the corresponding sample modules. Each sample is well documented in the corresponding classes using java docs.

You should download the binaries and sources (https://maven.nuxeo.org/nexus/index.html#nexus-search;gav~~nuxeo-webengine-samples*~~~) from our maven repository (since 5.4.2).

Tutorial 1 - Hello World.

This tutorial demonstrates how to handle requests. This is the simplest object. It requires only one java class which represents the Resource (the entry point).

You can access the tutorial sample at: http://localhost:8080/nuxeo/site/samples/hello

Tutorial 2 - Using Templates

This tutorial demonstrates how to use templates to render dynamic content.

You can access the tutorial sample at: http://localhost:8080/nuxeo/site/samples/templating

Tutorial 3 - Web Object Model

This tutorial demonstrates the basics of the WebEngine Object Model. You can see how to create new Module Resources, Object Resources, Adapter Resources and views.

You can access the tutorial sample at: http://localhost:8080/nuxeo/site/samples/basics

Tutorial 4 - Working with Documents

This tutorial demonstrates how to access Nuxeo Platform Documents through WebEngine.
You can access the tutorial sample at: http://localhost:8080/nuxeo/site/samples/documents

Tutorial 5 - Module Extensibility

This tutorial demonstrates how modules can be extended and how the links you are using in your templates can be managed to create easy to maintain and modular applications.

You can access the tutorial sample at: http://localhost:8080/nuxeo/site/samples/extended

Hello World

Module definition

module.xml

```
<?xml version="1.0"?>

<module name="sample1" root-type="sample1" path="/sample1">
  <nature>groovy</nature>
</module>
```

JAX-RS resources

Sample1.groovy

```
import javax.ws.rs.*;
import javax.ws.rs.core.*;
import org.nuxeo.ecm.core.rest.*;
import org.nuxeo.ecm.webengine.model.*;
import org.nuxeo.ecm.webengine.model.impl.*;
import org.nuxeo.ecm.webengine.model.exceptions.*;
import org.nuxeo.ecm.webengine.*;

/**
 * Web Module Main Resource Sample.
 * <p>
 * This demonstrates how to define the entry point for a WebEngine module.
 * <p>
 * The module entry point is a regular JAX-RS resource named 'Sample1' and with an
additional @WebModule annotation.
 * This annotation is mainly used to specify the WebModule name. I will explain the
rest of @WebModule attributes in the following samples.
 * A Web Module is implicitly defined by its entry point. You can also configure a Web
Module using a module.xml file located
 * in the module root directory. This file can be used to define: root resources (as
we've seen in the previous example), links, media type IDs
 * random extensions to other extension points; but also to define new Web Modules
without an entry point.
 * <p>
 * A Web Module's Main resource is the entry point to the WebEngine model build over
JAX-RS resources.
 * If you want to benefit of this model you should define such a module entry point
rather than using plain JAX-RS resources.
 * <p>
 * This is a very simple module example, that prints the "Hello World!" message.
 *
 * @author <a href="mailto:bs@nuxeo.com">Bogdan Stefanescu</a>
 */
@WebObject(type="sample1")
@Produces(["text/html"])
public class Sample1 extends ModuleRoot {

  @GET
  public String doGet() {
    return "Sample1: Hello World!";
  }

  @GET
  @Path("{name}")
  public String doGet(@PathParam("name") String name) {
    return "Sample1: Hello "+name+"!";
  }

}
```

Using FreeMarker Template Language (FTL)

Module definition

module.xml

```
<?xml version="1.0"?>

<module name="sample2" root-type="sample2" path="/sample2">
  <nature>groovy</nature>
</module>
```

JAX-RS resources

Sample2.groovy

```
import javax.ws.rs.*;
import javax.ws.rs.core.*;
import org.nuxeo.ecm.webengine.model.impl.*;
import org.nuxeo.ecm.webengine.model.*;

/**
 * Templates sample.
 *
 * This demonstrates how to use template files to build client responses.
 * JAX-RS provides a flexible mechanism to send responses based on the mime type that
the client expects.
 * To send a response to the client you simply return the Object you want as the
response.
 * JAX-RS engines will usually know how to render common Java objects like String,
InputStream, File etc.
 * If you need to output specific objects you need to register a custom
MessageBodyWriter class.
 * In JAX-RS you are not able to modify the HttpServletResponse object directly from a
resource method. (add headers, cookies etc)
 * Anything you may want to output must be returned from the resource method back to
JAX-RS engine, and the engine will output it for you.
 * This is a very good thing, even if for some people this approach may seem strange.
 * You may ask yourself, ok cool, The response rendering is pretty well separated from
the resource logic.
 * But how can I modify response headers?
 * In that case you must return a javax.ws.rs.Response that may be used to customize
your response headers.
 * <p>
 * WebEngine is adding a new type of response objects: templates.
 * Templates are freemarker based templates that can be used to render your objects
depending on the request context.
 * WebEngine is adding some cool extensions to freemarker templates that let you build
your web site in a modular fashion.
 * These extensions are called blocks. Blocks are dynamic template parts that can be
extended or replaced using derived blocks.
 * Using blocks, you can write a base template that may define the site layout (using
blocks containing empty or generic content) and then
 * write final <i>skins</i> for your layout by extending the base template and
redefining blocks you are interested in.
 * See the <i>skin</i> directory for template examples.
 * <p>
 * Templates are stored in files under the <i>skin</i> directory. Templates are always
resolved relative to the <i>skin</i> directory,
 * even if you are using absolute paths.
```

```
 * The following variables are accessible from a template when rendered at rendering
time:
 * <ul>
 * <li> <code>Context</code>  - the WebContext instance
 * <li> <code>Engine</code>   - the WebEngine instance
 * <li> <code>This</code>     - the target Web Object.
 * <li> <code>Root</code>     - the root WebObject.
 * <li> <code>Document</code> - the target Document if any otherwise null.
 * <li> <code>Session</code>  - the Repository Session. (aka Core Session)
 * <li> <code>basePath</code> - the request base path (context path + servlet path)
 * </ul>
 * To render a template as a response you need to instantiate it and then return it
from the resource method.
 * The template will be processed by the corresponding MessageBodyWriter and rendered
on the client stream.
 *
 * @author <a href="mailto:bs@nuxeo.com">Bogdan Stefanescu</a>
 */
@WebObject(type="sample2")
@Produces(["text/html"])
public class Sample2 extends ModuleRoot {

  @GET
  public Object doGet() {
    return "Sample2: Hello World!";
  }

  /**
   * Return the template index.ftl from 'skin' directory
   */
  @GET
  @Path("index1")
  public Object getIndex1() {
    return getTemplate("index1.ftl");
  }


  /**
   * Inject the variable 'name' in the template context and then return the template.
   */
  @GET
  @Path("index1/{name}")
  public Object getIndex1(@PathParam("name") String name) {
    return getTemplate("index1.ftl").arg("name", name);
  }

  /**
   * Render the index2 template
   */
  @GET
  @Path("index2")
  public Object getIndex2() {
    return getTemplate("index2.ftl");
  }

  /**
   * Example of using redirect.
   * The redirect method inherited from DefaultModule is returning a Response object
that is doing a redirect
```

```
 */
@GET
@Path("redirect/{whereToRedirect}")
public Response doRedirect(@PathParam("whereToRedirect") String path) {
  return redirect(ctx.getModulePath() + "/"+ path);
}

/**
 * Example of using a Response object.
 * This method is sending a 403 HTTP error.
 */
@GET
@Path("error/{errorCode}")
public Response sendError(@PathParam("errorCode") String errorCode) {
  try {
    int statusCode = Integer.parseInt(errorCode);
    Response.Status status = Response.Status.fromStatusCode(statusCode);
    if (status != null) {
     return Response.status(status).build();
    }
  } catch (Exception e) {
  }
  return Response.status(500).entity("Invalid error code: " + errorCode).build();
}
```

```
}
```

Object views

skin/base.ftl

```
<!-- Base template that defines the site layout -->
<html>
  <head>
    <title><@block name="title"/></title>
  </head>
  <body>
    <table width="100%" border="1">
      <tr>
        <td><@block name="header">Header</@block></td>
      </tr>
      <tr>
        <td><@block name="content">Content</@block></td>
      </tr>
      <tr>
        <td><@block name="footer">Footer</@block></td>
      </tr>
    </table>
  </body>
</html>
```

skin/index1.ftl

```
<@extends src="base.ftl">
  <@block name="title">Index 2</@block>
  <@block name="header">
    <#if name>
    Hello ${name}!
    <#else>
    Hello World!
    </#if>
  </@block>
  <@block name="content">
    This is the <i>index1</i> skin.
  </@block>
  <@block name="footer">
    The footer here ...
  </@block>
</@extends>
```

skin/index2.ftl

```
<@extends src="base.ftl">
  <@block name="title">Index 2</@block>
  <@block name="content">
    This is the <i>index2</i> skin.
  </@block>
</@extends>
```

Web Object Model

Module definition

module.xml

```
<?xml version="1.0"?>

<module name="sample3" root-type="sample3" path="/sample3">
  <nature>groovy</nature>
</module>
```

JAX-RS resources

Sample3.groovy

```
import javax.ws.rs.*;
import javax.ws.rs.core.*;
import org.nuxeo.ecm.webengine.model.impl.*;
import org.nuxeo.ecm.webengine.model.*;

/**
 * WebEngine Object Model.
 *
 * This sample is explaining the basics of Nuxeo WebEngine Object Model.
 * <p>
 *
 * <h3>Resource Model</h3>
 * Resources are objects used to serve the request. WebEngine Resources are always
stateless (a new instance is created on each request).
 * There are three type of resources defined by WebEngine:
 * <ul>
 * <li> Module Resource - this is the Web Module entry point as we've seen in sample3.
 * This is the root resource. The other type of resources are JAX-RS sub-resources.
 * A WebModule entry point is a special kind of WebObject having as type name the
module name.
 * <li> Web Object - this represents an object that can be requested via HTTP methods.
 *  This resource is usually wrapping some internal object to expose it as a JAX-RS
resource.
 * <li> Web Adapter - this is a special kind of resource that can be used to adapt Web
Objects
 * to application specific needs.
 * These adapters are useful to add new functionalities on Web Objects without
breaking application modularity
 * or adding new methods on resources.
```

```
 * This is helping in creating extensible applications, in keeping the code cleaner
and in focusing better on the REST approach
 * of the application.
 * For example let say you defined a DocumentObject which will expose documents as
JAX-RS resources.
 * A JAX-RS resources will be able to respond to any HTTP method like GET, POST, PUT,
DELETE.
 * So let say we use:
 * <ul>
 * <li> <code>GET</code> to get a view on the DocumentObject
 * <li> <code>POST</code> to create a DocumentObject
 * <li> <code>PUT</code> to update a document object
 * <li> <code>DELETE</code> to delete a DocumentObject.
 * </ul>
 * But what if I want to put a lock on the document? Or to query the lock state? or to
remove the lock?
 * Or more, to create a document version? or to get a document version?
 * A simple way is to add new methods on the DocumentObject resource that will handle
requests top lock, unlock, version etc.
 * Somethig like <code>@GET @Path("lock") getLock()</code> or <code>@POST
@Path("lock") postLock()</code>.
 * But this approach is not flexible because you cannot easily add new fonctionalities
on existing resources in a dynamic way.
 * And also, doing so, you will end up with a cluttered code, having many methods for
each new aspect of the Web Object you need to handle.
 * To solve this problem, WebEngine is defining Web Adapters, so that they can be used
to add new fonctionality on existing objects.
 * For example, to handle the lock actions on an Web Object we will define a new class
LockAdapter which will implement
 * the <code>GET</code>, <code>POST</code>, <code>DELETE</code> methods to manage the
lock fonctionality on the target Web Object.
 * Adapters are specified using an '@' prefix on the segment in an HTTP request path.
This is needed by WebEngine to differentiate
 * Web Objects from Web Adapters.
 * Thus in our lock example to request the lock adapter on an object you will use a
request path of like the following:
 * <code>GET /my/document/@lock</code> or <code>POST /my/document/@lock</code> etc.
 * <p>
 * When defining a Web Adapter you can specify on which type of Web Object it may be
used. (this is done using annotations)
 * </ul>
 * All WebEngine resources have a type, a super type, an optional set of facets and an
optional guard (these are declared using annotations)
 * By using types and super types you can create hierarchies or resources, so that
derived resource types will inherit attributes of the super types.
 * <p>
 *
 * There is a builtin adapter that is managing Web Objects views. The adapter name is
<code>@views</code>.
 * You will see in the view model an example on how to use it.
 * <p>
 *
 * Thus, request paths will be resolved to a resource chain usually of the form:
WebModule -> WebObject -> ... -> WebObject [ -> WebAdapter ].
 * <br>
 * Each of these resource objects will be <i>served</i> using the <i>sub-resource</i>
mechanism of JAX-RS until the last resource is reached.
 * The last resource will usually return a view to be rendered or a redirection
response.
```

```
 * The request resource chain is exposed by the WebContext object, so that one can
programatically retrieve any resource from the chain.
 * In a given resource chain there will be always 2 special resources: a <b>root</b>
and a <b>target</b> resource
 * The root resource is exposed in templates as the <code>Root</code> object and the
target one as the contextual object: <code>This</code>.
 * <br>
 * <b>Note</b> that the root resource is not necessarily the first one, and the target
resource is not necessarily the last one!
 * More, the root and the target resources are never WebAdapters. They can be only
WebObjects or WebModule entry points
 * (that are aspecial kind of WebObjects).
 * <p>
 * The root resource is by default the module entry point (i.e. the first resource in
the chain) but can be programatically set to point to any other
 * WebObject from the chain.
 * <p>
 * The target resource will be always the last WebObject resource from the chain.(so
any trailing WebAdapters are excluded).
 * This means in the chain: <code>/my/space/doc/@lock</code>, the root will be by
default <code>my</code> which is the module entry point,
 * and the target resource will be <code>doc</doc>. So it means that the
<code>$This</code> object exposed to templates (and/or views) will
 * never points to the adapter <code>@lock</code> - but to the last WebObject in the
chain.
 * So when an adapter view is rendered the <code>$This</code> variable will point to
the adapted WebObject and not to the adapter itself.
 * In that case you can retrieve the adapter using <code>${This.activeAdapter}</code>.
 * This is an important aspect in order to correctly understand the behavior of the
<code>$This</code> object exposed in templates.
 * <p>
 * <p>
 * <h3>View Model</h3>
 * The view model is an extension of the template model we discussed in the previous
sample.
 * The difference between views and templates is that views are always attached to an
Web Object. Also, the view file resolution is
 * a bit different from template files. Templates are all living in <code>skin</skin>
directory. Views may live in two places:
 * <ul>
 * <li> in the skin/views/${type-name} folders where type-name is the resource type
name the view is applying on.
 * This location will be consulted first when a view file is resolved, so it can be
used by derived modules to replace views on already defined objects.
 * <li> in the same folder (e.g. java package) as the resource class.
 * This location is useful to defining views inside JARs along with resource classes.
 * </ul>
 * Another specific property of views is that they are inherited from resource super
types.
 * For example if you have a resource of type <code>Base</code> and a resource of type
<code>Derived</code> then all views
 * defined on type <code>Base</code> apply on type <code>Dervied</code> too.
 * You may override these views by redefining them on type <code>Derived</code>
 * <br>
 * Another difference between templates and views is that views may vary depending on
the response media-type.
 * A view is identified by an ID. The view file name is computed as follow:
 * <pre>
 * view_id + [-media_type_id] + ".ftl"
```

```
 * </pre>
 * The <code>media_type_id</code> is optional and will be empty for media-types not
explicitely bound to an ID in modules.xml configuration file.
 * For example, to dynamically change the view file corresponding to a view
 * having the ID <code>index</code> when the response media-type is
<code>application/atom+xml</code>
 * you can define a mapping of this media type to the media_type_id <code>atom</code>
and then you can use the file name
 * <code>index-atom.ftl</code> to specify a specific index view when <code>atom</code>
output is required.
 *
 * @author <a href="mailto:bs@nuxeo.com">Bogdan Stefanescu</a>
 */
@WebObject(type="sample3")
@Produces(["text/html"])
public class Sample3 extends ModuleRoot {

  /**
   * Get the index view. The view file name is computed as follows:
index[-media_type_id].ftl
   * First the skin/views/sample4 is searched for that file then the current
directory.
   * (The type of a module is the same as its name)
   */
  @GET
  public Object doGet() {
    return getView("index");
  }

  /**
   * Get the WebObject (i.e. a JAX-RS sub-resource) bound to "users".
   * Look into "users" directory for the UserManager WebObject. The location of
WebObjects is not explictely specified by the programmer.
   * The module directory will be automatically scanned for WebObject and WebAdapters.
   */
  @Path("users")
  public Object getUserManager() {
    // create a new instance of an WebObject which type is "UserManager" and push this
object on the request chain
    return newObject("UserManager");
  }
```

```
}
```

users/UserManager.groovy

```
package users;

import javax.ws.rs.*;
import javax.ws.rs.core.*;
import org.nuxeo.ecm.webengine.model.impl.*;
import org.nuxeo.ecm.webengine.model.*;

/**
 * UserManager object.
 * You can see the @WebObject annotation that is defining a WebObject of type
"UserManager"
 * @author <a href="mailto:bs@nuxeo.com">Bogdan Stefanescu</a>
 */
@WebObject(type="UserManager")
@Produces(["text/html", "*/*"])
public class UserManager extends DefaultObject {

  /**
   * Get the index view. The view file name is computed as follows:
index[-media_type_id].ftl
   * First the skin/views/UserManager is searched for that file then the current
directory.
   * (The type of a module is the same as its name)
   */
  @GET
  public Object doGet() {
    return getView("index");
  }

  /**
   * A hack to accept users as user?name=xxx query parameters
   */
  @GET
  @Path("user")
  public Object getUserByQueryString(@QueryParam("name") String name) {
    if (name == null) {
      return doGet();
    } else {
      return redirect(getPath()+"/user/"+name);
    }
  }

  /**
   * Get the user JAX-RS resource given the user name
   */
  @Path("user/{name}")
  public Object getUser(@PathParam("name") String name) {
    // create a new instance of a WebObject which type is "User" and push this object
on the request chain
    // the User object is intialized with the String "Username: name"
    return newObject("User", "Username: "+name);
  }

}
```

users/User.groovy

```
package users;

import javax.ws.rs.*;
import javax.ws.rs.core.*;
import org.nuxeo.ecm.webengine.model.impl.*;
import org.nuxeo.ecm.webengine.model.*;

/**
 * User object.
 * You can see the @WebObject annotation that is defining a WebObject of type "User"
 * @author <a href="mailto:bs@nuxeo.com">Bogdan Stefanescu</a>
 */
@WebObject(type="User")
@Produces(["text/html", "*/*"])
public class User extends DefaultObject {

  String displayName;

  /**
   * Initialize the object.
   * args values are the one passed to the method newObject(String type, Object ...
args)
   */
  protected void initialize(Object... args) {
    displayName = args[0];
  }

  /**
   * Getter the variable displayName. Would be accessible from views with
${This.displayName}
   */
  public String getDisplayName() {
    return displayName;
  }


  /**
   * Get the index view of the User object.
   * The view file is located in <code>skin/views/User</code> so that it can be easily
extended
   * by a derived module. See extensibility sample.
   */
  @GET
  public Object doGet() {
    return getView("index");
  }

  /**
   * This method is not implemented but demonstrates how DELETE requests can be used
   */
  @DELETE
  public Object doRemove(@PathParam("name") String name) {
    //TODO ... remove user here ...
    // redirect to the UserManager (the previous object in the request chain)
    return redirect(getPrevious().getPath());
  }
```

```
/**
 * This method is not implemented but demonstrates how PUT requests can be used
 */
@PUT
public Object doPut(@PathParam("name") String name) {
  //TODO ... update user here ...
  // redirect to myself
  return redirect(getPath());
}
```

```
}
```

users/UserBuddies.groovy

```
package users;

import javax.ws.rs.*;
import javax.ws.rs.core.*;
import org.nuxeo.ecm.webengine.model.impl.*;
import org.nuxeo.ecm.webengine.model.*;

/**
 * UserBuddies object.
 * You can see the @WebAdapter annotation that is defining a WebAdapter of type
"UserBuddies" that applies to any User WebObject.
 * The name used to access this adapter is the adapter name prefixed with a '@'
character: <code>@buddies</code>
 *
 * @author <a href="mailto:bs@nuxeo.com">Bogdan Stefanescu</a>
 */
@WebAdapter(name="buddies", type="UserBuddies", targetType="User")
@Produces(["text/html", "*/*"])
public class UserBuddies extends DefaultAdapter {

  /**
   * Get the index view. The view file name is computed as follows:
index[-media_type_id].ftl
   * First the skin/views/UserBuddies is searched for that file then the current
directory.
   * (The type of a module is the same as its name)
   */
  @GET
  public Object doGet() {
    return getView("index");
  }


}
```

Object views

skin/views/sample3/index.ftl

```
<html>
  <head>
    <title>Sample3</title>
  </head>
  <body>
    <h3>Sample3 Index View.</h3>
    <p><a href="${This.path}/users">User Management</a></p>
  </body>
</html>
```

skin/views/UserManager/index.ftl

```
<html>
  <head>
    <title>Sample3</title>
  </head>
  <body>
    <h3>UserManager Index</h3>
    <form method="GET" action="${This.path}/user" onSubmit="">
    Enter a fictive User name: <input type="text" name="name" value=""/>
    </form>
  </body>
</html>
```

skin/views/User/index.ftl

```
<html>
  <head>
    <title>Sample3</title>
  </head>
  <body>
    <h3>${This.displayName}</h3>
    View my <a href="${This.path}/@buddies">buddies</a>
  </body>
</html>
```

skin/views/UserBuddies/index.ftl

```
<html>
  <head>
    <title>Sample3 - Adapter example</title>
  </head>
  <body>
    <#-- Look here how $This is used to access current user and not Buddies adapter
-->
    <h3>Buddies for user ${This.name}!</h3>
    <#-- Look here how to access the adapter instance: ${This.activeAdapter} -->
    This is an adapter named  ${This.activeAdapter.name}
    <ul>
    Buddies:
        <li><a href="${This.previous.path}/user/Tom">Tom</li>
        <li><a href="${This.previous.path}/user/Jerry">Jerry</li>
    </ul>
  </body>
</html>
```

Working with Documents

Module definition

module.xml

```
<?xml version="1.0"?>

<module name="sample4" root-type="sample4" path="/sample4" extends="base">
  <nature>groovy</nature>
</module>
```

JAX-RS resources

Sample4.groovy

```
import javax.ws.rs.*;
import javax.ws.rs.core.*;
import org.nuxeo.ecm.core.rest.*;
import org.nuxeo.ecm.webengine.model.impl.*;
import org.nuxeo.ecm.webengine.model.*;
import org.nuxeo.ecm.webengine.model.exceptions.*;

/**
 * Working with Nuxeo Documents.
 *
 * Nuxeo Documents are transparently mapped to WebObjects so that you can easily
access your documents
 * through WebEngine.
 * Nuxeo Documents are defined by a document type and can be structured in a hierarchy
based on their type.
 * The ancestor of all document types is the "Document" type.
 * Document types are transparently mapped to WebObject types, so that you don't need
to explicitely declare
```

```
 * WebObjects that expose documents. By default all documents are exposed as
DocumentObject instances (which is an WebObject).
 * If you need specific control over your document type you need then to explicitely
declare a new WebObject using the same
 * type name as your document type. This way, the default binding to DocumentObject
will be replaced with your own WebObject.
 * <p>
 * <b>Note</b> that it is recommended to subclass the DocumentObject when redefining
document WebObjects.
 * <p>
 * Also, Documents as WebObjects may have a set of facets. Documents facets are
transparently exposed as WebObject facets.
 * When redefining the WebObject used to expose a Document you can add new facets
using @WebObject annotation
 * (these new facets that are not existing at document level but only at WebObject
level).
 * <p>
 * To work with documents you need first to get a view on the repository. This can be
done using the following methods:
 * <br>
 * <code>DocumentFactory.getDocumentRoot(ctx, path)</code> or
<code>DocumentFactory.getDocument(ctx, path)</code>
 * <br>
 * The difference between the two methods is that the getDocumentRoot is also setting
 * the newly created document WebObject as the root of the request chain.
 * The document WebObject created using the DocumentFactory helper class will
represent the root of your repository view.
 * To go deeper in the repository tree you can use the <code>newDocument</code>
methods on the DocumentObject instance.
 * <p>
 * <b>Remember</b> that when working with documents you may need to log in to be able
to access the repository.
 * (it depends on whether or not the repository root is accessible to Anonymous user)
 * For this reason we provide in this example a way to login into the repository.
 * This also demonstrates <b>how to handle errors</b> in WebEngine. The mechanism is
simple:
 * At your module resource level you redefine a method
 * <code>public Object handleError(WebApplicationException e)</code> that will be
invoked each time
 * an uncatched exception is thrown during the request. From that method you should
return a suitable response to render the error.
 * To ensure exceptions are correclty redirected to your error handler you must catch
all exceptions thrown in your resource methods
 * and rethrowing them as following: <code> ... } catch (Throwable t) { throw
WebException.wrap(t); } </code>.
 * The exception wrapping is automatically converting exceptions to the ones defined
by WebEngine model.
 * <p>
 * The default exception handling defined in ModuleRoot class is simply printing the
exception on the output stream.
 *
 * @author <a href="mailto:bs@nuxeo.com">Bogdan Stefanescu</a>
 */
@WebObject(type="sample4")
@Produces(["text/html"])
public class Sample4 extends ModuleRoot {

  @GET
  public Object doGet() {
```

```
  return getView("index");
}

/**
 * Get a repository view rooted under "/default-domain".
 */
@Path("repository")
public Object getRepositoryView() {
  return DocumentFactory.newDocumentRoot(ctx, "/default-domain");
}


/**
 * Example on how to handle errors
 */
public Response handleError(WebApplicationException e) {
  if (e instanceof WebSecurityException) {
    // display a login page
    return Response.status(401).entity(getTemplate("error/error_401.ftl")).build();
  } else if (e instanceof WebResourceNotFoundException) {
    return Response.status(404).entity(getTemplate("error/error_404.ftl")).build();
  } else {
    // not interested in that exception - use default handling
    return super.handleError(e);
  }
}
```

```
}
```

Object views

skin/base.ftl

```
<!-- base template -->
<html>
  <head>
    <title><@block name="title"/>Sample4</title>
  </head>
  <body>
    <@block name="content"/>
  </body>
</html>
```

skin/views/sample4/index.ftl

```
<@extends src="base.ftl">
<@block name="title">Sample 4: Working with documents</@block>
<@block name="content">

Browse <a href="${This.path}/repository">repository</a>

</@block>
</@extends>
```

skin/views/Document/index.ftl

```
<@extends src="base.ftl">

<@block name="content">
    <h2>${Document.title}</h2>
    <div>Document ID: ${Document.id}
    <div>Document path: ${Document.path}
    <div>Document name: ${Document.name}
    <div>Document type: ${Document.type}

    <#-- Here we declare a nested block. Look in sample6 how nested block can be
redeclared -->
    <@block name="info">

    <div>
    Document schemas:
    <ul>
    <#list Document.schemas as schema>
      <li> ${schema}
    </#list>
    </ul>
    </div>
    <div>
    Document facets:
    <ul>
    <#list Document.facets as facet>
      <li> ${facet}
    </#list>
    </ul>
    </div>
  </@block>

  <#if Document.isFolder>
    <hr>
    <div>
    Document children:
    <ul>
    <#list Document.children as doc>
      <li> <a href="${This.path}/${doc.name}">${doc.name}</a>
    </#list>
    </ul>
    </div>
  </#if>

</@block>
</@extends>
```

Templates

skin/error/error_401.ftl

```
<@extends src="base.ftl">
<@block name="header"><h1><a href="${appPath}">Error</a></h1></@block>
<@block name="content">

<h1>401 - Unauthorized</h1>

<p>
You don't have privileges to access this page
</p>
<p>
<br/>
</p>
<#include "error/login.ftl">

</@block>
</@extends>
```

skin/error/error_404.ftl

```
<@extends src="base.ftl">
<@block name="header"><h1><a href="${appPath}">Error</a></h1></@block>
<@block name="content">

<h1>404 - Resource Not Found</h1>

The page you requested doesn't exists

</@block>
</@extends>
```

skin/error/login.ftl

```
<!-- Login Form -->
<form action="${Context.loginPath}" method="POST">
<table cellpadding="4" cellspacing="1">
  <tr>
    <td>Username:</td>
    <td><input name="username" type="text"></td>
  </tr>
  <tr>
    <td>Password:</td>
    <td><input name="password" type="password"></td>
  </tr>
  <tr align="right">
    <td colspan="2">
      <input type="submit" value="Sign In"/>
    </td>
  </tr>
  <#if Context.getProperty("failed") == "true">
  <tr align="center">
    <td colspan="2"><font color="red">Authentication Failed!</font></td>
  </tr>
  </#if>
</table>
</form>
```

Module Extensibility

The module defined here extends the module defined in Tutorial 4.

Module definition

module.xml

```
<?xml version="1.0"?>

<module name="sample5" root-type="sample5" path="/sample5" extends="sample4">
  <nature>groovy</nature>
</module>
```

JAX-RS resources

Samples5.groovy

```
import javax.ws.rs.*;
import javax.ws.rs.core.*;
import org.nuxeo.ecm.core.rest.*;
import org.nuxeo.ecm.webengine.model.impl.*;
import org.nuxeo.ecm.webengine.model.*;
import org.nuxeo.ecm.webengine.model.exceptions.*;

/**
```

```
 * Web Module Extensibility.
 *
 * This sample is demonstrating how existing web modules can be extended.
 * To extend another module you should use the <code>base="BaseModule"</code> in the
<code>@WebModule</code>
 * annotation. This way the new module will inherit all templates and resources
defined in the base module.
 * You can thus create a chain of inherited web modules.
 * <p>
 * Here is how template resolval will be impacted by the module inheritance:
 * <br>
 * <i>If a template T is not found in skin directory of derived module then search the
template inside the base module and so on
 * until a template is found or no more base module exists.</i>
 * The view resolval is similar to the template one but it will use the WebObject
inheritance too:
 * <br>
 * <i></i>
 * <br>
 * <b>Note</b> that only the <i>skin</i> directory is stacked over the one in the base
module.
 * The other directories in the module are not inheritable.
 * <p>
 * Also, resource types defined by the base module will become visible in the derived
one.
 * <p>
 * In this example you will also find a very useful feature of WebEngine: the builtin
<b>view service adapter</b>.
 * This adapter can be used on any web object to locate any view declared on that
object.
 * Let's say we define a view named <i>info</i> for the <i>Document</i> WebObject
type.
 * And the following request path will point to a Document WebObject:
<code>/my/doc</code>.
 * Then to display the <i>info</i> view we can use the builtin views adapter this way:
 * <code>/my/doc/@views/info</code>.
 * <p>
 * Obviously, you can redefine the WebObject corresponding to your document type and
add a new method that will dispatch
 * the view <info>info</info> using a pretty path like <code>/my/doc/info</code>. But
this involves changing code.
 * If you don't want this then the views adapter will be your friend.
 *
 * <p>
 * <p>
 * This example will extend the module defined in sample5 and will reuse and add more
templates.
 * Look into template files to see how base module templates are reused.
 *
 * @author <a href="mailto:bs@nuxeo.com">Bogdan Stefanescu</a>
 */
@WebObject(type="sample5")
@Produces(["text/html"])
public class Sample5 extends Sample4 {

  /**
   * We are reusing bindings declared in the main class from sample5 and only a new
one.
   */
```

```
@Path("info")
@GET
public Object getInfo() {
  return "This is the 'info' segment added by the derived module";
}
```

```
}
```

Object views

h5 skin/views/sample5/index.ftl

```
<#-- we are reusing the base template from the base module -->
<@extends src="base.ftl">

<#-- we are redefining only the title block -->
<@block name="title">Sample 5: Web Module Extensibility</@block>

<@block name="content">
Browse <a href="${This.path}/repository">repository</a>
</@block>

</@extends>
```

skin/views/Document/index.ftl

```
<#-- we reuse base.ftl from base module -->
<@extends src="base.ftl">

<@block name="content">
  <h2>${Document.title}</h2>
  <div>Document ID: ${Document.id}</div>
  <div>Document path: ${Document.path}</div>
  <div>Document name: ${Document.name}</div>
  <div>Document type: ${Document.type}</div>

  <p>
    <#-- we redefine the nested block info by adding a link to another view named
'info' on the document -->
    <@block name="info">
    <#-- look how the builtin view service adapter is used to locate the 'info' view
-->
    <a href="${This.path}/@views/info">More Info</a>
    </@block>
  </p>

  <#if Document.isFolder>
    <hr/>
    <div>
    Document children:
    <ul>
    <#list Document.children as doc>
      <li> <a href="${This.path}/${doc.name}">${doc.name}</a> </li>
    </#list>
    </ul>
    </div>
  </#if>

</@block>
</@extends>
```

skin/views/Document/info.ftl

```
<@extends src="base.ftl">

<#--
Here is an additional view on a document added by the derived  module.
You can display the view by using the builtin View Service adapter.
Example: /my/doc/@views/info
-->

<@block name="content">
  <h2>More info on document ${Document.title}</h2>
  <h3>Last modified: ${Document["dc:modified"]}</h3>
  <div>
    Document schemas:
    <ul>
    <#list Document.schemas as schema>
      <li> ${schema} </li>
    </#list>
    </ul>
  </div>
  <div>
    Document facets:
    <ul>
    <#list Document.facets as facet>
      <li> ${facet} </li>
    </#list>
    </ul>
  </div>
</@block>

</@extends>
```

Managing Links

Module definition

module.xml

```
<?xml version="1.0"?>
<module name="sample6" root-type="sample6" path="/sample6" extends="base">
  <nature>groovy</nature>

  <links>
  <!-- link IDs are normally used as the keys of i18n messages - but in this example
we are displaying them directly without using i18n mechanism-->
  <!-- link to a info view available for all Documents (i.e. WebObjects that are
derived from Document type) -->
  <link id="Info" path="/@views/info">
    <category>TOOLS</category>
    <category>INFO</category>
    <type>Document</type>
  </link>

  <!-- Link enabled only for folderish documents -->
  <link id="Children" path="/@views/children">
    <facet>Folderish</facet>
    <type>Document</type>
    <category>TOOLS</category>
  </link>

  <!-- this is only demonstrating link conditions - this is not a real link...
       This link will be enabled only for WebObject derived from Workspace
  -->
  <link id="I am a workspace" path="">
    <type>Workspace</type>
    <category>TOOLS</category>
  </link>
  </links>

</module>
```

JAX-RS resources

Sample6.groovy

```
import javax.ws.rs.*;
import javax.ws.rs.core.*;
import org.nuxeo.ecm.core.rest.*;
import org.nuxeo.ecm.webengine.model.impl.*;
import org.nuxeo.ecm.webengine.model.*;
import org.nuxeo.ecm.webengine.model.exceptions.*;

/**
 * Managing links.
 * <p>
 * Almost any template page will contain links to other pages in your application.
 * These links are usually absolute paths to other WebObjects or WebAdapters
(including parameters if any).
 * Maintaining these links when application object changes is painful when you are
using modular applications
 * (that may contribute new views or templates).
 * <p>
```

```
 * WebEngine is providing a flexible way to ease link management.
 * First, you should define all of your links in <i>module.xml</i> configuration file.
 * A Link is described by a target URL, an enablement condition, and one or more
categories that can be used to organize links.
 * <ul>
 * Here are the possible conditions that you can use on links:
 * <li> type - represent the target Web Object type. If present the link will be
enabled only in the context of such an object.
 * <li> adapter - represent the target Web Adapter name. If present the link will be
enabled only if the active adapter is the same as this one.
 * <li> facet - a set of facets that the target web object must have in order to
enable the link.
 * <li> guard - a guard to be tested in order to enable the link. This is using the
guard mechanism of WebEngine.
 * </ul>
 * If several conditions are specified an <code>AND</code> will be used between them.
 * <p>
 * Apart conditions you can <i>group</i> links in categories.
 * Using categories and conditions you can quickly find in a template which are all
enabled links that are part of a category.
 * This way, you can control which links are written in the template without needing
to do conditional code to check the context if links are enabled.
 * <p>
 * Conditions and categories manage thus where and when your links are displayed in a
page. Apart this you also want to have a target URL for each link.
 * <ul>
 * You have two choices in specifying such a target URL:
 * <li> define a custom link handler using the <code>handler</handler> link attribute.
 * The handler will be invoked each time the link code need to be written in the
output stream so that it can programatically generate the link code.
 * <li> use the builtin link handler. The builtin link handler will append the
<code>path</code> attribute you specified in link definition
 * to the current WebObject path on the request. This behavior is good enough for most
of the use cases.
 * <li>
 * </ul>
 * <p>
 * <p>
 * This example will demonstrate how links work. Look into <code>module.xml</code> for
link definitions
 * and then in <code>skin/views/Document/index.ftl</code> on how they are used in the
template.
 *
 * @author <a href="mailto:bs@nuxeo.com">Bogdan Stefanescu</a>
 */
@WebObject(type="sample6")
@Produces(["text/html"])
public class Sample6 extends ModuleRoot {

  @GET
  public Object doGet() {
    return getView("index");
  }

  /**
   * Get a repository view rooted under "/default-domain".
   */
  @Path("repository")
  public Object getRepositoryView() {
```

```
  return DocumentFactory.newDocumentRoot(ctx, "/default-domain");
}

/**
 * Example on how to handle errors
 */
public Response handleError(WebApplicationException e) {
  if (e instanceof WebSecurityException) {
    // display a login page
    return Response.status(401).entity(getTemplate("error/error_401.ftl")).build();
  } else if (e instanceof WebResourceNotFoundException) {
    return Response.status(404).entity(getTemplate("error/error_404.ftl")).build();
  } else {
    // not interested in that exception - use default handling
    return super.handleError(e);
  }
}
```

```
}
```

Object views

skin/base.ftl

```
<html>
  <head>
    <title>Sample 6: Working with links</title>
  </head>
  <body>
    <@block name="content"/>
  </body>
</html>
```

skin/views/sample6/index.ftl

```
<@extends src="base.ftl">

  <@block name="content">
    Browse Repository: <a href="${This.path}/repository">repository</a>
  </@block>

</@extends>
```

skin/views/Document/index.ftl

```
<@extends src="base.ftl">

<@block name="content">
  <h2>${Document.title}</h2>

<table width="100%" border="1">
  <tr>
    <td>
    <div>Document ID: ${Document.id}
    <div>Document path: ${Document.path}
    <div>Document name: ${Document.name}
    <div>Document type: ${Document.type}
    <hr>
    <#if Document.isFolder>
    <div>
    Document children:
    <ul>
    <#list Document.children as doc>
      <li> <a href="${This.path}/${doc.name}">${doc.name}</a> </li>
    </#list>
    </ul>
    </div>
    </#if>
    </td>
    <td>
      <#-- display here the links available in the current context in category INFO
-->
      <ul>
      <b>Tools</b>
      <#list This.getLinks("INFO") as link>
        <li> <a href="${link.getCode(This)}">${link.id}</a> </li>
      </#list>
      </ul>
      <#-- display here the links available in the current context in category TOOLS
-->
      <ul>
      <b>Adminitsration</b>
      <#list This.getLinks("TOOLS") as link>
        <li> <a href="${link.getCode(This)}">${link.id}</a> </li>
      </#list>
      </ul>
    </td>
  </tr>
</table>

</@block>
</@extends>
```

skin/views/Document/children.ftl

```
<@extends src="base.ftl">

<@block name="content">
  <#if Document.isFolder>
    <div>
    Document children:
    <ul>
    <#list Document.children as doc>
      <li> <a href="${This.path}/${doc.name}">${doc.name}</a> </li>
    </#list>
    </ul>
    </div>
  </#if>
</@block>

</@extends>
```

skin/views/Document/info.ftl

```
<@extends src="base.ftl">

<@block name="content">
  <h2>More info on document ${Document.title}</h2>
  <h3>Last modified: ${Document["dc:modified"]}</h3>
  <div>
    Document schemas:
    <ul>
    <#list Document.schemas as schema>
      <li> ${schema} </li>
    </#list>
    </ul>
  </div>
  <div>
    Document facets:
    <ul>
    <#list Document.facets as facet>
      <li> ${facet} </li>
    </#list>
    </ul>
  </div>

</@block>

</@extends>
```

Templates

skin/error/error_401.ftl

```
<@extends src="base.ftl">
<@block name="header"><h1><a href="${appPath}">Error</a></h1></@block>
<@block name="content">

<h1>401 - Unauthorized</h1>

<p>
You don't have privileges to access this page
</p>
<p>
<br/>
</p>
<#include "error/login.ftl">

</@block>
</@extends>
```

skin/error/error_404.ftl

```
<@extends src="base.ftl">
<@block name="header"><h1><a href="${appPath}">Error</a></h1></@block>
<@block name="content">

<h1>404 - Resource Not Found</h1>

The page you requested doesn't exists

</@block>
</@extends>
```

skin/error/login.ftl

```
<!-- Login Form -->
<form action="${Context.loginPath}" method="POST">
<table cellpadding="4" cellspacing="1">
  <tr>
    <td>Username:</td>
    <td><input name="username" type="text"></td>
  </tr>
  <tr>
    <td>Password:</td>
    <td><input name="password" type="password"></td>
  </tr>
  <tr align="right">
    <td colspan="2">
      <input type="submit" value="Sign In"/>
    </td>
  </tr>
  <#if Context.getProperty("failed") == "true">
  <tr align="center">
    <td colspan="2"><font color="red">Authentication Failed!</font></td>
  </tr>
  </#if>
</table>
</form>
```

Other services

This section lists additional services and modules that are part of the default distribution of the Nuxeo Platform. You can also refer to the additional packages section for understanding how to integrate features offered by additional plugins.

- Publisher service — There are three ways to publish a document:
- Thumbnail service — Since Nuxeo Platform 5.7.1, documents can have a thumbnail. A thumbnail is a reduced-size version of a picture used to help in recognizing and organizing documents. It will stand for any kind of document according to the type and/or facet.
- Nuxeo Core Import / Export API — The import / export service is providing an API to export a set of documents from the repository in an XML format and then re-importing them back. The service can also be used to create in batch document trees from valid import archives or to provide a simple solution of creating and retrieving repository data. This could be used for example to expose repository data through REST or raw HTTP requests.
- Work and WorkManager — The WorkManager service allows you to run code later, asynchronously, in a separate thread and transaction.

Publisher service

There are three ways to publish a document:

- on local sections, i.e. the sections created in your Nuxeo DM instance,
- on remote sections, i.e. the sections of a remote Nuxeo server,
- on the file system.

Publication is configured using the `PublisherService`.

About the `PublisherService`

When using the `PublisherService`, you only need to care about three interfaces:

- **`PublishedDocument`** : represents the published document. It can be created from a DocumentModel, a proxy or a file on the file system.
- **`PublicationNode`** : represents a Node where you can publish a DocumentModel. It can be another DocumentModel (mainly Folder / Section) or a directory on the file system.
- **`PublicationTree`** : the tree which is used to publish / unpublish documents, to approve / reject publication, list the already published documents in a PublicationNode, ... See the java doc of the PublicationTree.

On this page

The PublisherService mainly works with three concepts:

- **`factory`** : the class which is used to actually create the published document. It also manages the approval / rejection workflow on published documents.
- **`tree`** : a `PublicationTree` instance associated to a name: for instance, we have a `SectionPublicationTree` which will publish in Sections, a `LocalFSTree` to publish on the file system, ...
- **`tree instance`** : an actual publication tree where we define the factory to use, the underlying tree to use, its name / title, and some parameters we will see later.

Configuring Local Sections Publishing

Publishing in local sections was the only way to publish on versions < 5.3GA. From Nuxeo DM 5.3GA, it is the default way to publish documents.

Here is the default contribution you can find in Nuxeo `publisher-jbpm-contrib.xml` in `nuxeo-platform-publisher-jbpm`. This contribution overrides the one in `publisher-contrib.xml` located in the `nuxeo-platform-publisher-core` project:

```
<extension target="org.nuxeo.ecm.platform.publisher.impl.service.PublisherServiceImpl"
  point="treeInstance">

  <publicationTreeConfig name="DefaultSectionsTree" tree="RootSectionsCoreTree"
    factory="CoreProxyWithWorkflow" localSectionTree="true"
    title="label.publication.tree.local.sections">
    <parameters>
      <!-- <parameter name="RootPath">/default-domain/sections</parameter> -->
      <parameter name="RelativeRootPath">/sections</parameter>
      <parameter name="enableSnapshot">true</parameter>
      <parameter name="iconExpanded">/icons/folder_open.gif</parameter>
      <parameter name="iconCollapsed">/icons/folder.gif</parameter>
    </parameters>
  </publicationTreeConfig>

</extension>
```

In this contribution, we define an instance using the `RootSectionsCoreTree` tree and the `CoreProxyWithWorkflow` factory. We give it a name, a title and configure it to be a `localSectionTree` (which means we will publish the documents in the Sections of the Nuxeo application the documents are created in).

The available parameters are:

- `RootPath`: it's used when you want to define the root publication node of your `PublicationTree`. You can't use both `RootPath` AND `RelativeRoothPath` parameters.
- `RelativeRootPath`: used when you just want to define a relative path (without specifying the domain path). A `PublicationTree` inst ance will be created automatically for each domain, appending the `RelativeRootPath` value to each domain.
  For instance, let's assume we have two domains, domain-1 and domain-2, and the `RelativeRootPath` is set to "/sections".
  Then, two `PublicationTree` instances will be created: the first one with a `RootPath` set to "/domain-1/sections", and the second one with a `RootPath` set to "/domain-2/sections",
  In the UI, when publishing, you can chose the `PublicationTree` you want. The list of trees will automatically be updated when creating and deleting domains.
- `iconExpanded` and `iconCollapsed`: specify which icons to use when displaying the `PublicationTree` on the user interface.

Configuring Remote Sections Publishing

To make the remote publication work, both the Nuxeo server instance and Nuxeo client instance need to be configured.

Server Configuration

You should create a new configuration file, `publisher-server-config.xml` for instance, in the `nuxeo.ear/config` folder of your Nuxeo acting as a server (ie the Nuxeo application on which the documents will be published).

Here is a sample configuration:

```
<?xml version="1.0"?>
<component name="org.nuxeo.ecm.platform.publisher.contrib.server">

  <extension
target="org.nuxeo.ecm.platform.publisher.impl.service.PublisherServiceImpl"
    point="treeInstance">

    <publicationTreeConfig name="ServerRemoteTree" tree="CoreTreeWithExternalDocs"
factory="RemoteDocModel" >
      <parameters>
        <parameter name="RootPath">/default-domain/sections</parameter>
      </parameters>
    </publicationTreeConfig>

  </extension>

</component>
```

The available parameters are:

- `RootPath`: its value must be the path to the document which is the root of your `PublicationTree`. Here, it will be the document `/default-domain/sections`, the default Sections root in Nuxeo.
  This parameter can be modified to suit your needs.

  Don't forget to put the whole path to the document.

Client Configuration

You should create a new configuration file, `publisher-client-config.xml` for instance, in the `nuxeo.ear/config` folder of your Nuxeo acting as a client (ie the Nuxeo application from which documents are published).

Here is a sample configuration:

```
<?xml version="1.0"?>
<component name="org.nuxeo.ecm.platform.publisher.contrib.client">

  <extension
target="org.nuxeo.ecm.platform.publisher.impl.service.PublisherServiceImpl"
    point="treeInstance">

    <publicationTreeConfig name="ClientRemoteTree" tree="ClientForRemoteTree"
      factory="ClientProxyFactory">
      <parameters>
        <parameter name="title">label.publication.tree.remote.sections</parameter>
        <parameter name="userName">Administrator</parameter>
        <parameter name="password">Administrator</parameter>
        <parameter name="baseURL">
          http://myserver:8080/nuxeo/site/remotepublisher/
        </parameter>
        <parameter name="targetTree">ServerRemoteTree</parameter>
        <parameter name="originalServer">localserver</parameter>
        <parameter name="enableSnapshot">true</parameter>
      </parameters>
    </publicationTreeConfig>

  </extension>

</component>
```

The available parameters:

- `targetTree`: this parameter corresponds to the name of the tree defined on the server Nuxeo application, here `ServerRemoteTree`.
- `username`, `password`: the user account defined by those parameters will be the one used to connect to the remote Nuxeo and so to create documents in the `PublicationTree`. This account MUST exist on the server.
- `baseURL`: the URL used by the PublisherService on the client side to communicate with the server Nuxeo application.
- `originalServer`: identifies the Nuxeo application used as client.

Configuring File System Publishing

To publish on the file system, you just need to define a new `TreeInstance` using the `LocalFSTree` and the `RootPath` of your tree.

Here is a sample configuration:

```
<extension
target="org.nuxeo.ecm.platform.publisher.impl.service.PublisherServiceImpl"
  point="treeInstance">

  <publicationTreeConfig name="FSTree" tree="LocalFSTree"
    factory="LocalFile" localSectionTree="false"
    title="label.publication.tree.fileSystem">
    <parameters>
      <parameter name="RootPath">/opt/publishing-folder</parameter>
      <parameter name="enableSnapshot">true</parameter>
      <parameter name="iconExpanded">/icons/folder_open.gif</parameter>
      <parameter name="iconCollapsed">/icons/folder.gif</parameter>
    </parameters>
  </publicationTreeConfig>

</extension>
```

The available parameters are:

- `RootPath`: the root folder on the file system to be used as the root of the publication tree.
- `iconExpanded` and `iconCollapsed`: specify which icons to use when displaying the `PublicationTree` on the user interface.
- `enableSnapshot`: defines if a new version of the document is created when the document is published.

Enabling Duplication of Relations upon Publication

By default, the relations on the document in the workspace are not duplicated on the published document. But it is possible to have them duplicated.

To enable this duplication of relations, you need to add the following contribution to the Platform:

```
<extension target="org.nuxeo.ecm.core.event.EventServiceComponent"
    point="listener">
  <listener name="publishRelationsListener" async="false" postCommit="false"
    class="org.nuxeo.ecm.platform.relations.core.listener.PublishRelationsListener"
    priority="-50">
    <event>documentProxyPublished</event>
  </listener>
</extension>
```

See the How to Contribute to an Extension page to add the contribution in Nuxeo Studio.

Thumbnail service

Since Nuxeo Platform 5.7.1, documents can have a thumbnail. A thumbnail is a reduced-size version of a picture used to help in recognizing and organizing documents. It will stand for any kind of document according to the type and/or facet.
Custom thumbnail factories can be contributed to the thumbnail service, using its extension point. Thumbnails are then available through this service to define how they will be computed and fetched.

Thumbnail Service Architecture

In this section

Here are the different components of the thumbnail feature:

- **Thumbnail service**
  The service that handles thumbnail factories contributions.
  - Interface: `ThumbnailService`,
  - Implementation: `ThumbnailServiceImpl`,
  - Component: `org.nuxeo.ecm.core.api.thumbnail.ThumbnailService`,
  - Extension point: `thumbnailFactory`.

- **Default Thumbnail factories**
  - `ThumbnailDocumentFactory`: Default thumbnail factory for all non-folderish documents. Returns the main blob converted in thumbnail or get the document's big icon as a thumbnail.
  - `ThumbnailFolderishFactory`: Default thumbnail factory for all folderish documents.
  - `ThumbnailPictureFactory`: Default Picture thumbnail factory.
  - `ThumbnailVideoFactory`: Video thumbnail factory from DAM.
  - `ThumbnailAudioFactory`: Audio thumbnail factory from DAM.

- **Thumbnail listeners**
  - `UpdateThumbnailListener`: Thumbnail listener handling creation and update of document event to store document thumbnail preview (only for the File document type).
  - `CheckBlobUpdateListener`: Thumbnail listener handling document blob update and checking changes. Fires a `scheduleThumbnailUpdate` event if it's the case that will trigger `UpdateThumbnailListener`.

Thumbnail factory on GitHub
Here are Nuxeo thumbnail factory implementations on GitHub:

- `ThumbnailDocumentFactory`,
- `ThumbnailVideoFactory`,
- `ThumbnailAudioFactory`,
- `ThumbnailPictureFactory`.

Registering a New Thumbnail Factory

A thumbnail factory can be registered using the following example extension:

```
<extension target="org.nuxeo.ecm.core.api.thumbnail.ThumbnailService"
    point="thumbnailFactory">

    <thumbnailFactory name="thumbnailFolderishFactory" facet="Folderish"

factoryClass="org.nuxeo.ecm.platform.thumbnail.factories.ThumbnailFolderishFactory" />

    <thumbnailFactory name="thumbnailAudioFactory" docType="Audio"
      factoryClass="org.nuxeo.ecm.platform.audio.extension.ThumbnailAudioFactory" />

 </extension>
```

The above thumbnail factories will be used to compute and fetch specific thumbnails for folderish documents on one hand, and audio documents on the other hand. Here are their properties:

- `docType`: string identifying the related document type. In the example, the type is "Audio".
- `facet`: string identifying the related document facet. In the example, the facet is "Folderish".
- `factoryClass`: string representing the class name of the factory to use.

Each factory should implement the interface `ThumbnailFactory`. This interface contract contains two methods to implement:

- `Blob getThumbnail(DocumentModel doc, CoreSession session)`: gets the document thumbnail (related to the doc type/facet).
- `Blob computeThumbnail(DocumentModel doc, CoreSession session)`: computes the thumbnail (related to the document type/facet).

The listener `UpdateThumbnailListener` is calling `YourFactory#computeThumbnail` that allows developers to compute the document blob when creating a document and after updating it (if the blob related to this document has been changed).

When computing your thumbnail, `UpdateThumbnailListener` stores it into a specific metadata `thumb:thumbnail` provided by the following schema:

```
<xs:schema xmlns:nxs="http://www.nuxeo.org/ecm/schemas/thumbnail"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.nuxeo.org/ecm/schemas/thumbnail">

  <xs:include schemaLocation="core-types.xsd" />

  <xs:element name="thumbnail" type="nxs:content" />

  </xs:schema>
```

This process can be useful to avoid lazy loading when displaying documents list.

Picture Thumbnail Example

Here is an example with the picture thumbnail factory, which is fetching a image existing into the picture schema.

```
import org.nuxeo.common.utils.FileUtils;
import org.nuxeo.ecm.core.api.Blob;
import org.nuxeo.ecm.core.api.CoreSession;
import org.nuxeo.ecm.core.api.DocumentModel;
import org.nuxeo.ecm.core.api.impl.blob.FileBlob;
import org.nuxeo.ecm.platform.picture.api.PictureView;
import org.nuxeo.ecm.platform.picture.api.adapters.MultiviewPicture;
import org.nuxeo.ecm.platform.types.adapter.TypeInfo;

public class ThumbnailPictureFactory implements ThumbnailFactory {
    @Override
    public Blob getThumbnail(DocumentModel doc, CoreSession session)
            throws ClientException {
        if (!doc.hasFacet("Picture")) {
            throw new ClientException("Document is not a picture");
        }
        // Choose the nuxeo default thumbnail of the picture views if exists
        MultiviewPicture mViewPicture = doc.getAdapter(MultiviewPicture.class);
        PictureView thumbnailView = mViewPicture.getView("Small");
        if (thumbnailView == null || thumbnailView.getBlob() == null) {
            // try thumbnail view
            thumbnailView = mViewPicture.getView("Thumbnail");
            if (thumbnailView == null || thumbnailView.getBlob() == null) {
                TypeInfo docType = doc.getAdapter(TypeInfo.class);
                return new FileBlob(
                        FileUtils.getResourceFileFromContext("nuxeo.war"
                                + File.separator + docType.getBigIcon()));
            }
        }
        return thumbnailView.getBlob();
    }
    @Override
    public Blob computeThumbnail(DocumentModel doc, CoreSession session) {
        return null;
    }
}
```

And then the usage of `ThumbnailAdapter`:

```
import org.nuxeo.common.utils.FileUtils;
import org.nuxeo.ecm.core.api.Blob;
import org.nuxeo.ecm.core.api.CoreSession;
import org.nuxeo.ecm.core.api.DocumentModel;
import org.nuxeo.ecm.core.api.blobholder.BlobHolder;
import org.nuxeo.ecm.core.api.impl.DocumentModelImpl;
import org.nuxeo.ecm.core.api.impl.blob.FileBlob;
import org.nuxeo.ecm.core.api.thumbnail.ThumbnailAdapter;
import org.nuxeo.ecm.core.api.CoreSession;
import com.google.inject.Inject;

@Inject
CoreSession session;

// Create a picture
DocumentModel root = session.getRootDocument();
DocumentModel picture = new DocumentModelImpl(root.getPathAsString(),"pic",
"Picture");
picture.setPropertyValue("picture:views", (Serializable) createViews());
picture = session.createDocument(picture);
session.save();

// Using BlobHolder adapter
BlobHolder bh = picture.getAdapter(BlobHolder.class);
Blob blob = new FileBlob(getFileFromPath("test-data/big_nuxeo_logo.gif"),
"image/gif",null, "big_nuxeo_logo.gif", null);
bh.setBlob(blob);
session.saveDocument(picture);
session.save();

// Using ThumbnailAdapter
ThumbnailAdapter pictureThumbnail = picture.getAdapter(ThumbnailAdapter.class);
Blob thumbnail = pictureThumbnail.getThumbnail(session);
String fileName = thumbnail.getFilename();
```

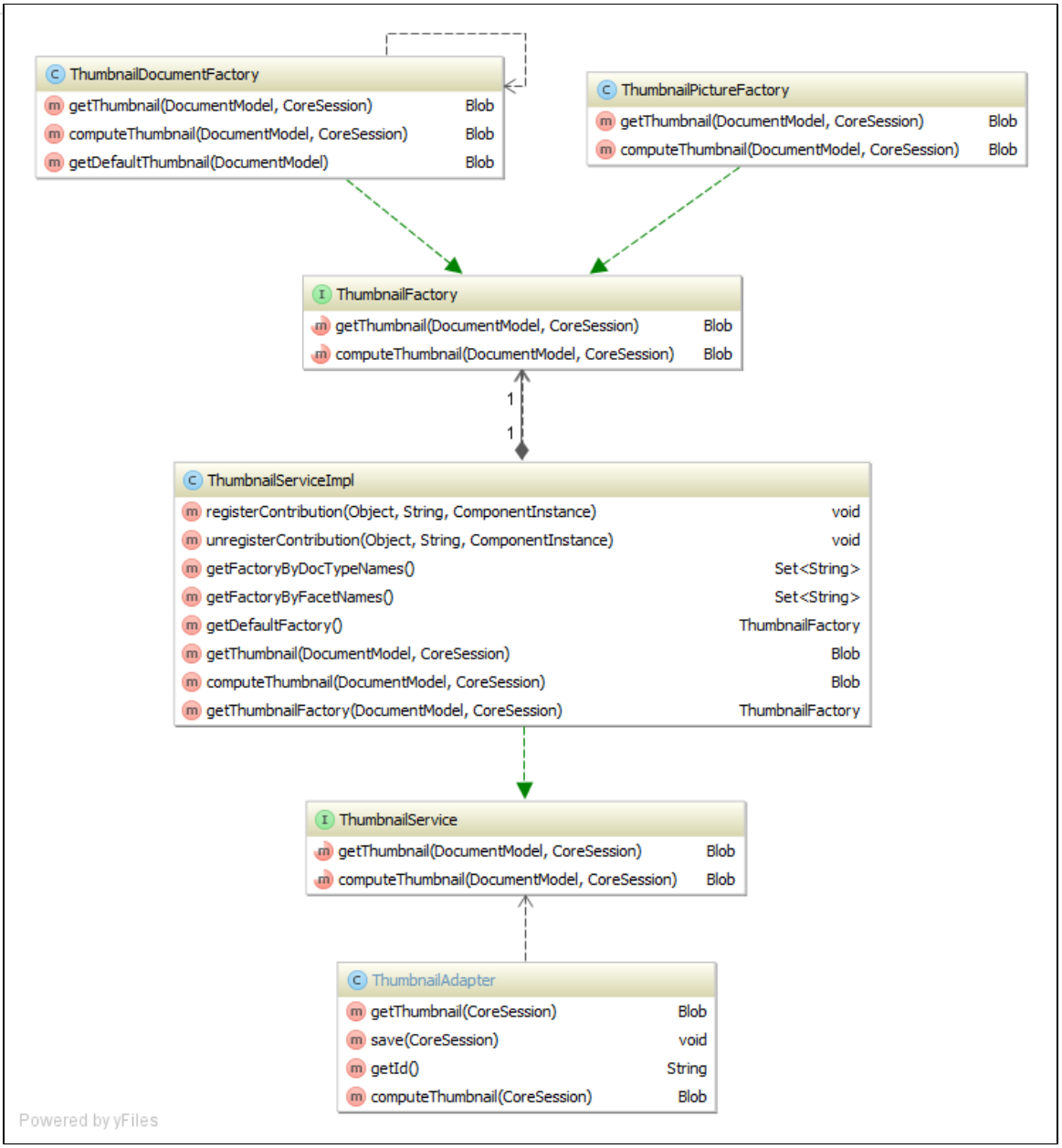
Default Nuxeo thumbnail factories fall back on **Nuxeo Document BigIcon** if no thumbnail has been found.

Here is a way to find it properly:

```
Blob getDefaultThumbnail(DocumentModel doc) {
        if (doc == null) {
            return null;
        }
        TypeInfo docType = doc.getAdapter(TypeInfo.class);
        String iconPath = docType.getBigIcon();
        if (iconPath == null) {
            iconPath = docType.getIcon();
        }
        if (iconPath == null) {
            return null;
        }
        FacesContext ctx = FacesContext.getCurrentInstance();
        if (ctx == null) {
            return null;
        }
        try {
            InputStream iconStream = ctx.getExternalContext().getResourceAsStream(
                    iconPath);
            if (iconStream != null) {
                return new FileBlob(iconStream);
            }
        } catch (IOException e) {
            log.warn(String.format(
                    "Could not fetch the thumbnail blob from icon path '%s'",
                    iconPath), e);
        }
        return null;
    }
```

Thumbnail Architecture

Here, we can see the `ThumbnailAdapter` to use and factories like the default one `ThumbnailDocumentFactory` and `ThumbnailPictureFactory`:



Powered by yFiles

Nuxeo Core Import / Export API

The import / export service is providing an API to export a set of documents from the repository in an XML format and then re-importing them back. The service can also be used to create in batch document trees from valid import archives or to provide a simple solution of creating and retrieving repository data. This could be used for example to expose repository data through REST or raw HTTP requests.
Export and import mechanism is extensible so that you can easily create you custom format for exported data. The default format provided by the Nuxeo Platform is described below.

The import / export module is part of the `nuxeo-core-api` bundle and it is located under the `org.nuxeo.ecm.core.api.io` package.

Export Format

A document will be exported as a directory using the document node name as a name and containing a `document.xml` file which holds the document metadata and properties as defined by document schemas. Document blobs, if any, are by default exported as separate files inside the document directory. There is also an option to export inlined blobs as Base64 encoded data inside the `document.xml`.

When exporting trees, document children are put as subdirectories inside the document parent

directory.

Optionally each service in Nuxeo that stores persistent data related to documents, like the workflow, relation or annotation services, may also export their own data inside the document folder as XML files.

A document tree will be exported as directory tree. Here is an example of an export tree containing relations information for a workspace named `workspace1`:

In this section


```
+ workspace1
    + document.xml
    + relations.xml

    + doc1
      + document.xml
      + relations.xml

    + doc2
      + document.xml
      + relations.xml
      + file1.blob

    + doc3
      + document.xml
```

`document.xml` Format

Here is an XML that corresponds to a document containing a blob. The blob is exported as a separate file:

```
<document repository="default" id="633cf240-0c03-4326-8b3b-0960cf1a4d80">
  <system>
    <type>File</type>
    <path>/default-domain/workspaces/ws/test</path>
    <lifecycle-state>project</lifecycle-state>
    <lifecycle-policy>default</lifecycle-policy>
    <access-control>
      <acl name="inherited">
        <entry principal="administrators" permission="Everything" grant="true"/>
        <entry principal="members" permission="Read" grant="true"/>
        <entry principal="members" permission="Version" grant="true"/>
        <entry principal="Administrator" permission="Everything" grant="true"/>
      </acl>
    </access-control>
  </system>
  <schema xmlns="http://www.nuxeo.org/ecm/schemas/files/" name="files">
    <files/>
  </schema>
  <schema xmlns:dc="http://www.nuxeo.org/ecm/schemas/dublincore/" name="dublincore">
    <dc:valid/>
    <dc:issued/>
    <dc:coverage></dc:coverage>
    <dc:title>test</dc:title>
    <dc:modified>Fri Sep 21 20:49:26 CEST 2007</dc:modified>
    <dc:creator>Administrator</dc:creator>
    <dc:subjects/>
    <dc:expired/>
    <dc:language></dc:language>
    <dc:rights>test</dc:rights>
    <dc:contributors>
      <item>Administrator</item>
    </dc:contributors>
    <dc:created>Fri Sep 21 20:48:53 CEST 2007</dc:created>
    <dc:source></dc:source>
    <dc:description/>
    <dc:format></dc:format>
  </schema>
  <schema xmlns="http://www.nuxeo.org/ecm/schemas/file/" name="file">
    <content>
      <encoding></encoding>
      <mime-type>application/octet-stream</mime-type>
      <data>cd1f161f.blob</data>
    </content>
    <filename>error.txt</filename>
  </schema>
  <schema xmlns="http://project.nuxeo.com/geide/schemas/uid/" name="uid">
    <minor_version>0</minor_version>
    <uid/>
    <major_version>1</major_version>
  </schema>
  <schema xmlns="http://www.nuxeo.org/ecm/schemas/common/" name="common">
    <icon-expanded/>
    <icon/>
    <size/>
  </schema>
</document>
```

You can see that the generated document is containing one <system> section and one or more <schema> sections.
The system section contains all system (internal) document properties like document type, path, lifecycle state and access control configuration. For each schema defined by the document type, there is a schema entry which contains the document properties belonging to that schema. The XSD schema that corresponds to that schema can be used to validate the content of the schema section. Anyway this is true only in the case of inlined blobs. By default, for performance reasons, blobs are put outside the XML file in their own file. So instead of encoding the blob in the XML file a reference to an external file is preserved: cd1f161f.blob

Here is how the same blob will be serialized when inlining blobs (an option of the repository reader):

```
<schema xmlns="http://www.nuxeo.org/ecm/schemas/file/" name="file">
    <content>
      <encoding></encoding>
      <mime-type>application/octet-stream</mime-type>
      <data>
       b3JnLmpib3NzLnJlbW90aW5nLkNhbm5vdENvbm5lY3RFeGNlcHRpb246IENhbiBub3QgZ2V0IGNv
       bm5lY3Rpb24gdG8gc2VydmVyLiAgUHJvYmxlbSBlc3RhYmxpc2hpbmcgc29ja2V0IGNvbm5lY3Rp
       [...]
       </data>
    </content>
    <filename>error.txt</filename>
  </schema>
```

Inlining Blobs

There is an option to inline the blob content in the XML file as a Base64 encoded text. This is less optimized but this is the canonic format to export a document data prior to XSD validation of document schemas.

Of course this is less optimized than writing the raw blob data in external files but provides a way to encode the entire document content in a single file and in a well known and validated format.

By default when exporting documents from the repository, blobs are not inlined. To activate the inlining option you must set call the method on the `DocumentModelReader` you are using to fetch data from the repository:

```
reader.setInlineBlobs(boolean inlineBlobs);
```

Document Pipe

An export process is a chain of three sub-processes:

- fetching data from the repository,
- transforming the data if necessary,
- writing the data to an external system.

In the same way an import can be defined as a chain of three sub-processes:

- fetching data from external sources,
- transforming the data if necessary
- writing the data into the repository.

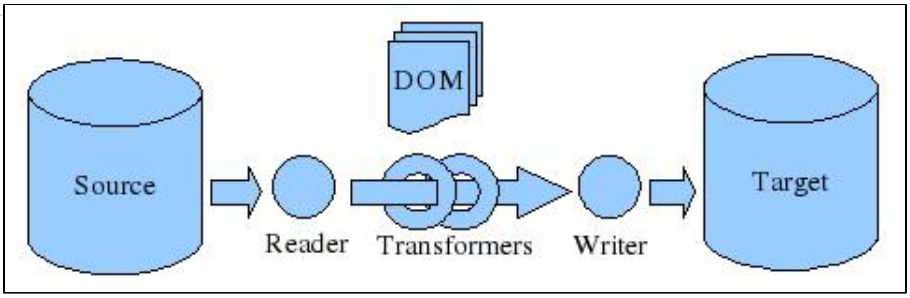
We will name the process chain used to perform imports and exports as a Document Pipe.

In both cases (imports and exports) a document pipe is dealing with the same type of objects:

- A document reader,
- Zero or more document transformers,
- A document writer.

So the document pipe will use a reader to fetch data that will be passed through registered transformers and then written down using a document writer.

See the API Examples for examples on how to use a Document Pipe.



Document Reader

A document reader is responsible for reading some input data and converting it into a DOM representation. The DOM representation is using the format explained in Document XML section. Currently dom4j documents are used as the DOM objects.

For example a reader may extract documents from the repository and output it as XML DOM objects. Or it may be used to read files from a file system and convert them into DOM objects to be able to import them in a Nuxeo repository.

To change the way documents are extracted and transformed to a DOM representation, you can implement your own document reader. Currently Nuxeo provides several flavors of document readers:

- Repository readers - this category of readers is used to extract data from the repository as DOM objects. All of these readers are extending `DocumentModelReader`:
  - `SingleDocumentReader` - this one reads a single document given its ID and export it as a dom4j document;
  - `DocumentChildrenReader` - this one reads the children of a given document and export each one as dom4j document;
  - `DocumentTreeReader` - this one reads the entire subtree rooted in the given document and export each node in the tree as a dom4j document;
  - `DocumentListReader` - this one is taking a list of document models as input and export them as dom4j documents. This is useful when wanting to export a search result for example.
- External readers used to read data as DOM objects from external sources like file systems or databases. The following readers are provided:
  - `XMLDirectoryReader` - reads a directory tree in the format supported by Nuxeo (as described in Export Format section). This can be used to import deflated Nuxeo archives or hand-created document directories;
  - `NuxeoArchiveReader` - reads Nuxeo Platform exported archives to import them in a repository. Note that only zip archives created by the Nuxeo exporter are supported;
  - `ZipReader` - reads a zip archive and output DOM objects. This reader can read both Nuxeo zip archives and regular zip archives (hand-made).
    Reading a Nuxeo archive is more optimized, because Nuxeo zip archives entries are added to the archive in a predefined order that makes it possible to read the entire archive tree on the fly without unziping the content of the archive on the filesystem first. If the zip archive is not recognized as a Nuxeo archive the zip will be deflated in a temporary folder on the file system and the `XMLDirectoryReader` will be used to read the content.

To create a custom reader you need to implement the interface `org.nuxeo.ecm.core.api.io.DocumentReader`.

Document Writer

A document writer is responsible for writing the documents that exit the pipe in a document store. This storage can be a file system, a Nuxeo repository or any database or data storage as long as you have a writer that supports it.

The following `DocumentWriters` are provided by Nuxeo:

- Repository Writers - These ones are writing documents to a Nuxeo repository. They are useful to perform imports into the repository:
  - `DocumentModelWriter` - writes documents inside a Nuxeo Repository. This writer is creating new document models for each of the imported documents;
  - `DocumentModelUpdater` - writes documents inside a Nuxeo Repository. This writer is updating documents that have the same ID as the imported ones or create new documents otherwise.
- External Writers - are writers that write documents on an external storage. They are useful to perform exports from the repository.
  - `XMLDocumentWriter` - writes a document as a XML file with inlined blobs;
  - `XMLDocumentTreeWriter` - writes a list of documents inside a unique XML file with inlined blobs. The document tags will be included in a `root` tag:

    ```
    <documents> .. </documents>
    ```

  - `XMLDirectoryWriter` - writes documents as a folder tree on the file system. To read back the exported tree you may use `XMLDirectoryReader`;
  - `NuxeoArchiveWriter` - writes documents inside a Nuxeo zip archive. To read back the archive you may use the `NuxeoArchi`

veReader.

To create a custom writer you need to implement the interface `org.nuxeo.ecm.core.api.io.DocumentWriter`.

Document Transformer

Document transformers are useful to transform documents that enter the pipe before they are sent to the writer. This way you can remove, add or modify some properties from the documents, or other information contained by the exported DOM object.

As documents are expressed as XML DOM objects you can also use XSLT transformations inside your transformer.

To create a custom transformer you need to implement the interface `org.nuxeo.ecm.core.api.io.DocumentTransformer`.

API Examples

Performing exports and imports can be done by following these steps:

1. Instantiate a new document pipe:

   ```
   // create a pipe that will process 10 documents on each iteration
   DocumentPipe pipe = new DocumentPipeImpl(10);
   ```

   The page size argument is important when you are running the pipe on a machine different than the one containing the source of the data (the one from where the reader will fetch data). This way you can fetch several documents at once and improve performances.
2. Create a new document reader that will be used to fetch data and put it into the pipe. Depending on the data you want to import you can choose between existing an document reader implementation or you may write your own if needed:

   ```
   reader = new DocumentTreeReader(docMgr, src, true);
   pipe.setReader(reader);
   ```

   In this example we use a `DocumentTreeReader` which will read an entire sub-tree form the repository rooted in 'src' document.

   The `docMgr` argument represents a session to the repository, the 'src' is the root of the tree to export and the 'true' flag means to exclude the root from the exported tree.
3. Create a document writer that will be used to write down the outputed by the pipe.

   ```
   writer = new XMLDirectoryWriter(new File("/tmp/export"));
   pipe.setWriter(writer);
   ```

   In this example we instantiate a writer that will write exported data onto the file system as a folder tree.
4. Optionally you may add one or more document transformers to transform documents that enter the pipe.

   ```
   MyTransformer transformer = new MyTransformer();
   pipe.addTransformer(transformer);
   ```

5. And now run the pipe.

   ```
   pipe.run();
   ```

Exporting Data from a Nuxeo Repository to a ZIP Archive

```
File zip = File.createTempFile("MyFile", ".zip");
DocumentReader reader = null;
DocumentWriter writer = null;
try {
  DocumentModel src = getTestWorkspace();
  reader = new DocumentTreeReader(docMgr, root, true);
  writer = new NuxeoArchiveWriter(zip);
  // creating a pipe
  DocumentPipe pipe = new DocumentPipeImpl(10);
  pipe.setReader(reader);
  pipe.setWriter(writer);
  pipe.run();
} finally {
  if (reader != null) {
    reader.close();
  }
  if (writer != null) {
    writer.close();
  }
}
```

Importing Data from a ZIP Archive to a Nuxeo Repository

```
DocumentReader reader = null;
DocumentWriter writer = null;
try {
  DocumentModel src = getTestWorkspace();
  reader = new ZipReader(new File("/tmp/export.zip"));
  writer = new DocumentModelWriter(docMgr, "import-domain/Workspaces/ws");

  // creating a pipe
  DocumentPipe pipe = new DocumentPipeImpl(10);
  pipe.setReader(reader);
  pipe.setWriter(writer);
  pipe.run();
} finally {
  if (reader != null) {
    reader.close();
  }
  if (writer != null) {
    writer.close();
  }
}
```

Exporting a Single Document as an XML with Inlined Blobs

```
DocumentReader reader = null;
DocumentWriter writer = null;
try {
  DocumentModel src = getTestWorkspace();
  reader = new SingleDocumentReader(docMgr, src);

  // inline blobs
  ((DocumentTreeReader)reader).setInlineBlobs(true);
  writer = new XMLDocumentWriter(new File("/tmp/export.zip"));

  // creating a pipe
  DocumentPipe pipe = new DocumentPipeImpl();

  // optionally adding a transformer
  pipe.addTransformer(new MyTransformer());
  pipe.setReader(reader);
  pipe.setWriter(writer); pipe.run();

} finally {
  if (reader != null) {
    reader.close();
  }
  if (writer != null) {
    writer.close();
  }
}
```

REST Exposition of Core IO Export

CoreIO default XML exports as bound using Restlet framework that is still available even if this has been superseded via JAX-RS.

- Export a single document as XML:
  `http://{server}:{port}/nuxeo/restAPI/{repository}/{uuid}/export?format=XML`
- Export a single document as XML + blobs in a ZIP:
  `http://{server}:{port}/nuxeo/restAPI/{repository}/{uuid}/export?format=ZIP`
- Export a document tree as XML + blobs in a ZIP:
  `http://{server}:{port}/nuxeo/restAPI/{repository}/{uuid}/exportTree`

Work and WorkManager

The WorkManager service allows you to run code later, asynchronously, in a separate thread and transaction.

Work Instance

The code to be executed asynchronously must be implemented in an `AbstractWork` subclass. In theory you could just implement `Work` you rself but this is strongly discouraged, for forward-compatibility reasons.

Work Construction

Each Work instance must have a unique id describing the Work in its entirety. The id can either be random (just call the empty `AbstractWork()` constructor), or specified by the caller (`AbstractWork(id)`).

At construction time, the Work instance should also have its `setDocument()` method called to set the repository name and document(s) id(s) that this Work instance is going to be dealing with, and a flag specifying whether the work is actually about a whole subtree under the given document. This is important for monitoring and locking purposes.

In this section

Work Implementation

A Work instance must be `Serializable` in order for it to be persisted between server restarts. All fields that are used only during work execution but don't hold any configuration state must be marked `transient`. Persistence is done through a Redis server, please check Redis Configuration for more.

A Work instance must implement the following methods:

- `getCategory()` should return the Work category, which is used to choose which Work Queue to use when queuing it for later execution.
- `getTitle()` should return a human-readable name for the Work instance, for monitoring purposes.
- `getRetryCount()` should return the number of retries allowed; this is used if `retryableWork()` is implemented.
- `work()` or `retryableWork()` should be implemented to do the actual work.
- `cleanUp()` can be overridden if there is some cleanup to do when the work is done. Make sure you call `super`.
- `rollbackAndRetryTransaction()` can be overridden for a retryable Work if there is some cleanup to do before a Work is retried. Make sure you call `super`.

The meat of the Work instance is the `work()` method (or `retryableWork()`, but we won't repeat this each time). The following is available during method execution:

- `setStatus()` should be called with a human-readable status at key times during the execution (for instance Starting, Extracting, Done), for monitoring purposes.
- `setProgress()` should be called periodically during the execution, if there is any progress to measure, for monitoring purposes.
- `initSession()` can be called to initialize the `session` field (a `CoreSession`) on the repository specified at construction time.

For long-running Work, `work()` should periodically check for `isSuspending()` and if `true` save the work state in non-transient fields of the Work instance and call `suspended()` and then return. This is how a server shutdown can interrupt a long-running Work, persist it, and relaunch it when the server restarts.

Work Scheduling

Once a Work instance is constructed, it must be sent to the WorkManager for execution using the `schedule()` method. The recommended way to call this method is:

```
WorkManager workManager = Framework.getLocalService(WorkManager.class);
workManager.schedule(work, true);
```

The `schedule()` method takes a second parameter which is `afterCommit`, and should be `true` in most cases otherwise the Work may be scheduled immediately even after the current transaction has committed, which would prevent the Work from seeing the results of the current transaction.

You should avoid using the `Scheduling` argument to the `schedule()` method, as it may become deprecated in the near future.

Work Queues

Every Work instance is queued in a Work Queue. Each queue is associated with a thread pool of one or more threads, which will execute the Work instances whenever a thread becomes available.

```
<extension point="queues" target="org.nuxeo.ecm.core.work.service">
  <queue id="myqueue">
    <name>My Queue</name>
    <maxThreads>2</maxThreads>
    <category>somecategory1</category>
    <category>somecategory2</category>
  </queue>
</extension>
```

The details of the configuration can be seen in the extension point documentation. Here we'll concentrate on the important points:

- The name is a human-readable name, for monitoring purposes.
- The categories define which Work instances go to this queue.
- The number of threads define how many parallel executions of Work instances can be happening.

If a Work instance has a category that isn't registered with any queue, then it will be sent to the `default` queue.

Persistent Work Queues

When using persistent queues (using Redis, see Redis Configuration), then any Nuxeo instances where the queue is configured will pick Work instances from the persistent queue according to the availability of a thread in its thread pool.

Other UI Frameworks

Nuxeo uses several UI frameworks beside the default JSF technology.

- GWT Integration — GWT is a web toolkit to build rich clients in Java programming language. The Java code is transcoded in JavaScript at build time so the build process generates a fully HTML+JavaScript application ready to be deployed on an HTTP server.
- Extending The Shell — This section is intended for developers who wants to provide new Shell commands, namespaces or Shell Features.
- Nuxeo Android Connector — Nuxeo Android Connector is a SDK to build Android Applications that communicate with a Nuxeo Server.

GWT Integration

This documents assumes you are familiar with GWT and have the basic knowledge on how to build GWT applications. You can find a complete introduction to GWT here:

http://code.google.com/webtoolkit/gettingstarted.html
GWT is a web toolkit to build rich clients in Java programming language. The Java code is transcoded in JavaScript at build time so the build process generates a fully HTML+JavaScript application ready to be deployed on an HTTP server.

GWT applications may contain both server side code (which is Java byte code) and client side code (which is Java in development mode but is transcoded in JavaScript at build time).

When using the GWT RPC mechanism you usually need to share the client code that makes up your application model (the data objects). This code is both compiled to JavaScript and to Java byte code.

Only a small subset of JRE classes can be transcoded by GWT to JavaScript (e.g. most of the classes in `java.lang` and `java.util`s).

Developing a GWT Application

Requirements

To develop a GWT based application for Nuxeo, you need first to install the GWT Eclipse plugin. Here is the list of update sites for each supported Eclipse distribution:

In this section

- Eclipse 3.6 (Helios)
- Eclipse 3.5 (Galileo)
- Eclipse 3.4 (Ganymede)
- Eclipse 3.3 (Europa)

Also you will need a Nuxeo version >= 5.3.1-SNAPSHOT for your Nuxeo dependencies and to be able to deploy your GWT applications.

Creating a Hello World Application

Create a new *Web Application Project*. Uncheck Use *Google App Engine* in the wizard page. The GWT wizard will create a project structure like:

```
src
    org/my/app/client
    org/my/app/server
    your_module.gwt.xml
war
   WEB-INF/web.xml
   your_module.css
   your_module.html
```

The client package will contain the Java code that must be transcoded into JavaScript. The data objects defined here can be shared on the server side too. The server package will contain code that will be used on the server side (as Java byte code).

As you noticed, a "`war`" directory was generated in the module root. Here you need to define any servlet or filter used in development mode (in the `web.xml` file). Also this directory contains the HTML home page of your application.

When working with a Nuxeo Server, what you need is to be able to start a Nuxeo Server when GWT starts the application in development mode. If you don't have a running Nuxeo inside the same Java process as the debugged application, you cannot use Nuxeo APIs or access the repository to be able to tests your GWT servlets.

To achieve this you need to follow these steps:

1. Add the JARs of `nuxeo-webengine-gwt` and `nuxeo-distribution-tools` v. 1.1 classifier "all" to your project classpath. When using Maven, this can be done by the following POM fragment:

```
<dependency>
  <groupId>org.nuxeo.ecm.webengine</groupId>
  <artifactId>nuxeo-webengine-gwt</artifactId>
</dependency>
<dependency>
  <groupId>org.nuxeo.build</groupId>
  <artifactId>nuxeo-distribution-tools</artifactId>
  <classifier>all</classifier>
  <version>1.1</version>
  <scope>test</scope>
</dependency>
```

2. Add to `war/WEB-INF/web.xml` a filter as following:

```
<filter>
  <display-name>WebEngine Authentication Filter</display-name>
  <filter-name>NuxeoAuthenticationFilter</filter-name>
  <filter-class>
  org.nuxeo.ecm.webengine.gwt.dev.NuxeoLauncher
  </filter-class>
  <init-param>
    <param-name>byPassAuthenticationLog</param-name>
    <param-value>true</param-value>
  </init-param>
  <init-param>
    <param-name>securityDomain</param-name>
    <param-value>nuxeo-webengine</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>NuxeoAuthenticationFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

This filter will be used to start an Embedded Nuxeo Server when the GWT application is started in development mode. You can find more details on how to control what version of Nuxeo is started, where the home directory will be created, etc. in the `nuxeo-webengine-gwt` sources in `org/nuxeo/ecm/webengine/gwt/web.xml`.

This filter will load any JAR or Project found in your classpath as a Nuxeo Bundle if the JAR contains a valid OSGi Manifest. So if your current project is a Nuxeo Plugin, it will be correctly deployed on the embedded Nuxeo.

If you want to deploy additional Nuxeo XML components on the embedded server, you need to extend the NuxeoLauncher and implement the method `frameworkStarted()` where you can deploy additional test components using the default mechanism in Nuxeo. (e.g. `Framework.getRuntime().getContext().deploy(url)`).

This filter is only usable in development mode and it must not be deployed on a real server.

Now you can start a debugging session by right-clicking the project and then clicking on *Debug As* > *Web Application*.

The Embedded Nuxeo will be started before the GWT application is initialized.

By default the web server in the embedded Nuxeo will listen on localhost:8081.

So you can connect to the address for the WebEngine UI if you want to introspect the repository.

The Nuxeo Server Embedded home directory used by default is

```
{user.home}/.nxserver-gwt
```

Now in your GWT servlets you can use calls to Nuxeo API, create and browse documents, etc. Of course you need to add the required dependencies on your project class path.

Deploying the GWT Application on a Nuxeo Server

To be able to deploy your GWT in a real Nuxeo Server you need to package it as a Nuxeo bundle that:

1. defines an OSGi `Bundle-Activator` in your `MANIFEST.MF` that points to `org.nuxeo.ecm.webengine.gwt.GwtBundleActivator`.

> If you need a custom activator you can override the `gwtBundleActivator` and add you own login in the `start()` method after calling `super.start()`, or you can directly implement `BundleActivator` and call `instart()` method the following code: `GwtBundleActivator.install(context)`.

2. contains the generated GWT application files into a directory named "`gwt-war`" at the root of the JAR.

So the JAR will have a content similar to the following one:

```
your-gwt-module.jar
  META-INF/MANIFEST.MF
  OSGI-INF/deployment-fragment.xml
  ...
  org/
  gwt-war/
    your_gwt_module.html
    ...
```

Then simply put your JAR into a Nuxeo Server. (Make sure the server contains `nuxeo-webengine-gwt.jar` in bundles directory).

At first startup the "`gwt-war`" directory from your JAR will be copied into {web}`/root.war/gwt/` where {web} is the webengine root directory.

At each startup, the GWT activator you added to your bundle will check if it needs to unzip again the "`gwt-war`" directory content.

This is true if the timestamp of the bundle JAR will be greater than the timestamp of the file {web}`/root.war/gwt/.metadata/{your_bundle_symbolic_name}`.

This way if you upgrade the JAR the GWT application files will be updated too.

At the end of this document you will find a fragment of a POM that can be used to correctly build a Nuxeo GWT module and that can also generate the GWT project in eclipse by running "`mvn eclipse:eclipse`".

Accessing your GWT Module From the Client

But how to expose the GWT application to clients?

For this you need to create a simple WebEngine module that expose the GWT application through a JAX-RS resource.

(You can either use WebEngine objects or raw JAX-RS resources - or even a custom servlet your registered in `web.xml`).

If your are using a WebEngine Module you only need to override the abstract class: `org.nuxeo.ecm.webengine.gwt.GwtResource` and implement a `@GET` method to server the GWT application home page like:

```
@WebObject(type="myGwtApp")
public class MyGwtApp extends GwtResource {
    @GET @Produces("text/html")
    public Object getIndex() {
        return getTemplate("studio.ftl");
    }
}
```

You can do the same from a raw JAX-RS resource by integrating the method from `GwtResource` into your resource:

```
@GET
    @Path("{path:.*}")
    public Response getResource(@PathParam("path") String path) {
        //System.out.println(">>> "+GWT_ROOT.getAbsolutePath());
        // avoid putting automatic no cache headers
        ctx.getRequest().setAttribute("org.nuxeo.webengine.DisableAutoHeaders",
"true");
        File file = new File(GwtBundleActivator.GWT_ROOT, path);
        if (file.isFile()) {
            ResponseBuilder resp = Response.ok(file);
            String fpath = file.getPath();
            int p = fpath.lastIndexOf('.');
            String ext = "";
            if (p > -1) {
                ext = fpath.substring(p+1);
            }
            String mimeType = ctx.getEngine().getMimeType(ext);
            if (mimeType == null) {
                mimeType = "text/plain";
            }
            resp.type(mimeType);
            return resp.build();
        }
        return Response.status(404).build();
    }
```

This method simply locates the file requested by the GWT client (in `{web}/root.war/gwt`) to send it to the client.

You can apply the same logic if you prefer to write a servlet as an entry point for your GWT module.

Using GWT RPC Mechanism in a Nuxeo GWT Module

If you want to use GWT RPC inside Nuxeo GWT modules you must be sure your RPC servlet classes extends `org.nuxeo.ecm.webengine.gwt.WebEngineGwtServlet` instead of `RemoteServiceServlet`.

This is required since the default `RemoteServiceServlet` is assuming a WAR structure that is not present in a Nuxeo Server.

The `WebEngineGwtServlet` locates correctly the resources needed by the GWT Serializer and then it dispatches the request back to `RemoteServiceServlet`.

And also don't forget to define your RPC servlets in the `web.xml`! You can use for this the regular approach in Nuxeo (through `deployment-fragment.xml` files).

Note: Your GWT RPC servlets are executed in an authenticated context since the Nuxeo Authentication filter is in place.

Using Other Server / Client Communication Mechanisms

Apart the GWT RPC mechanism you can communicate with the server using any mechanism you need as far as you define a servlet and a protocol between your client and server applications.

Event if custom communication works well in web mode (when the application is deployed in a real Nuxeo server) you will have problems to debug and use them in development mode (when Nuxeo is embedded in GWT IDE). This is because of the SOP (Same origin Policy) problems in Ajax applications.

As I said before the embedded Nuxeo server will listen at a different port (by default to 8081) than the GWT embedded HTTP server. This means you will not be able to do calls from GWT client to servlets registered in the embedded Nuxeo server since they belong to another domain (i.e. different port).

There are 2 ways of fixing the problem and make it work in development mode:

1. Register your servlets in the GWT HTTP server (in `war/WEB-INF/web.xml`)
   This method has many limitations since the servlet you want to use may need to be started by Nuxeo - or you may want to use Nuxeo JAX-RS or WebEngine objects.
2. Use the redirection service provided by `nuxeo-webengine-gwt`. This is the recommended method since you don't have any limitation.

How is This Working?

Let say your servlet is installed at http://localhost:8081/myservice. Doing an asynchronous HTTP request from GWT to this URL will not work as expected because of the SOP limitation.

Instead of this you can rewrite the URL by pre-pending a prefix "redirect" to your servlet path and using the GWT HTTP server domain like this: http://localhost:*8080*/*redirect*/myservice.

The call will be transparently redirected to the Nuxeo HTTP server (and the "redirect" prefix will be removed).

In GWT client code you can use GWT helpers to detect if you are in development mode or not and to use the correct URL in each case. You can for example send the service URL from the server at page load by using javascript to inject the service URL in your module HTML page or you can simply compute the service URL based on the GWT module URL - see GWT.getModuleBaseURL() or GWT.getHostPageBaseURL().

To know if you are in development you can use `GWT.isScript()` on the client side or `"true".equals(System.getProperty("nuxeo.gwt_dev_mode"))` on the server side.

You can also configure the prefix used for redirection and enabling tracing of the redirected requests (that will be printed in the Eclipse console). This can be configured into `war/WEB-INF/web.xml` by adding some parameters to `NuxeoLauncher` filter:

```
<!-- enable redirected request content tracing -->
<init-param>
  <param-name>redirectTraceContent</param-name>
  <param-value>true</param-value>
</init-param>
<init-param>
  <param-name>redirectPrefix</param-name>
  <param-value>/foo/bar</param-value>
</init-param>
```

If you want to trace only headers use "redirectTrace" instead of "redirectTraceContent"

Example of a pom.xml

```
<properties>
    <gwtVersion>2.0</gwtVersion>
    <gwt.module>org.your_gwt_module</gwt.module>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.osgi</groupId>
      <artifactId>osgi-core</artifactId>
    </dependency>
    <dependency>
      <groupId>org.nuxeo.runtime</groupId>
      <artifactId>nuxeo-runtime</artifactId>
```

```
  </dependency>
  <dependency>
    <groupId>org.nuxeo.common</groupId>
    <artifactId>nuxeo-common</artifactId>
  </dependency>
  <dependency>
    <groupId>org.nuxeo.ecm.webengine</groupId>
    <artifactId>nuxeo-webengine-gwt</artifactId>
  </dependency>
  <dependency>
    <groupId>org.nuxeo.build</groupId>
    <artifactId>nuxeo-distribution-tools</artifactId>
    <classifier>all</classifier>
    <version>1.1</version>
    <scope>test</scope>
  </dependency>

  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
  </dependency>

  <dependency>
    <groupId>com.google.gwt</groupId>
    <artifactId>gwt-servlet</artifactId>
    <version>${gwtVersion}</version>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>com.google.gwt</groupId>
    <artifactId>gwt-user</artifactId>
    <version>${gwtVersion}</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>com.google.gwt</groupId>
    <artifactId>gwt-dev</artifactId>
    <version>${gwtVersion}</version>
    <scope>runtime</scope>
  </dependency>

 </dependencies>

 <build>
  <!-- gwt compiler needs the java sources to correctly work -->
  <resources>
    <resource>
      <directory>src/main/java</directory>
    </resource>
    <resource>
      <directory>src/main/resources</directory>
    </resource>
  </resources>

  <plugins>
    <!-- correctly generate eclipse files with GWT nature -->
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-eclipse-plugin</artifactId>
```

```
        <configuration>
          <downloadSources>false</downloadSources>
          <additionalProjectnatures>
            <projectnature>com.google.gwt.eclipse.core.gwtNature</projectnature>
            <projectnature>com.google.gdt.eclipse.core.webAppNature</projectnature>
          </additionalProjectnatures>
          <additionalBuildcommands>
            <buildCommand>
              <name>com.google.gwt.eclipse.core.gwtProjectValidator</name>
              <arguments>
              </arguments>
              <name>com.google.gdt.eclipse.core.webAppProjectValidator</name>
              <arguments>
              </arguments>
            </buildCommand>
          </additionalBuildcommands>
          <classpathContainers>

<classpathContainer>org.eclipse.jdt.launching.JRE_CONTAINER</classpathContainer>

<classpathContainer>com.google.gwt.eclipse.core.GWT_CONTAINER</classpathContainer>
          </classpathContainers>
          <buildOutputDirectory>war/WEB-INF/classes</buildOutputDirectory>
        </configuration>
      </plugin>
      <!--
        After compiling java sources compile java to JS using GWT compiler. This
        must be done process-classes after compile step finished to be sure we
        have all the needed files in classes directory. I am using ant for this
        since the maven exec plugin is not able to run correctly this
      -->
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-antrun-plugin</artifactId>
        <executions>
          <execution>
            <id>compile-js</id>
            <phase>process-classes</phase>
            <configuration>
              <tasks>
                <property name="compile_classpath" refid="maven.compile.classpath"
/>
                <property name="runtime_classpath" refid="maven.runtime.classpath"/>
                <java failonerror="true" fork="true"
classname="com.google.gwt.dev.Compiler">
                  <classpath>
                    <pathelement location="${project.build.outputDirectory}" />
                    <pathelement path="${compile_classpath}" />
                    <pathelement path="${runtime_classpath}" />
                  </classpath>
                  <jvmarg value="-Xmx256M" />
                  <jvmarg value="${gwt.arg}" />
                  <!--arg value="-style" />
                  <arg value="DETAILED" /-->
                  <!-- to speed up compiler
                  <arg value="-draftCompile" /-->
                  <arg value="-war" />
                  <arg value="${project.build.outputDirectory}/gwt-war" />
                  <arg value="${gwt.module}" />
```

```
            </java>
          </tasks>
        </configuration>
        <goals>
          <goal>run</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
```

```
    </plugins>
  </build>
```

Extending The Shell

This section is intended for developers who wants to provide new Shell commands, namespaces or Shell Features.

Table of contents:

Shell Features

In order to install new commands to existing namespaces or to register new namespaces, completors, injector providers or other Shell objects you must create a new Shell Feature.
If you need to modify or add some built-in commands (this is more for Nuxeo developers) - you can directly modify the Nuxeo Shell implementation - so you don't need to create new Shell Features.

Creating new Features is the way to do when you:

1. Need to declare a new namespace (even if you are modifying the main Nuxeo Shell JAR).
2. Cannot modify the main Nuxeo Shell JAR.
3. Need to patch an existing feature but cannot modify the main Nuxeo Shell JAR.
4. Want to provide optional features in additional JARs

What is a Feature?

A feature is a Java class that implements the `org.nuxeo.shell.ShellFeature` interface:

In this section

```
public interface ShellFeature {
    /**
     * Install the feature in the given shell instance. This is typically
     * registering new global commands, namespaces, value adapters or
     * completors.
     *
     * @param shell
     */
    public void install(Shell shell);
}
```

The feature has only one method: `install(Shell shell)`. This method will be called by the shell at startup on every registered feature to register new things in the shell.

Shell Feature Registration

In order for the shell to discover your feature you need to register it. Nuxeo Shell is using the `Java ServiceLoader` mechanism to discover services.
So, to register a feature you must create a file named `org.nuxeo.shell.ShellFeature` inside the `META-INF/services` folder of your JAR. And then write into, every ShellFeature implementation (i.e. the implementation class name) you provide.
Each class name should be specified on one line.

This is the `org.nuxeo.shell.ShellFeature` file coming into the Nuxeo Shell JAR which declares the built-in features:

```
org.nuxeo.shell.fs.FileSystem
org.nuxeo.shell.automation.AutomationFeature
```

Of course to enable the shell discover your features you need to put your JAR on the Java classpath used by the shell application.

Examples

The FileSystem Feature

Here is an excerpt from the built-in FileSystem feature that provides the **local** namespace and file name completors.

```
public class FileSystem implements ShellFeature {
    ...

    public void install(Shell shell) {
        shell.putContextObject(FileSystem.class, this);
        shell.addValueAdapter(new FileValueAdapter());
        shell.addRegistry(FileSystemCommands.INSTANCE);
    }

    ...
}
```

Let's look at the content of the install method.
The first line is registering the feature instance as a context object of type FileSystem.class.

> **Context Objects**
> are object instances that are available for injection into any command field using the @Context annotation.

The second line contribute a new Value Adapter to the shell.

> **Value Adapters**
> are objects used to adapt an input type to an output type. They are used to adapt string values specified on the command line to a

value of the real type specified by the command field which was bound to a command line parameter.

The third line is registering a new namespace named **local** and which is implemented by FileSystemCommands class.

Command Registry
A command registry object is the materialization of a namespace. It must extend the abstract class `org.nuxeo.shell.CommandRegistry`

To activate a namespace at shell startup you can use:

```
shell.setActiveRegistry("myNamespaceName");
```

in your **install** method but this is too intrusive since it may override some other namespace that also want to be the default one.

To activate a namespace at shell startup set the **shell** Java system property to point to your namespace name when starting the shell application. Example:

```
java -Dshell=myNamespace -jar nuxeo-shell.jar
```

Contributing a Command to the Global Namespace

In this example we will see how to contribute a command to an existing namespace - in our example it will be the **global** namespace.

```
public class MyFeature implements ShellFeature {
    ...

    public void install(Shell shell) {
       GlobalCommands.INSTANCE.addAnnotatedCommand(MyCommand.class);
    }

    ...
}
```

You can see this feature is really simple. It is registering in the GlobalCommands registry a new command created from a Command annotated class.
If the registry you want to access is not providing a static INSTANCE field you can use the shell to lookup a registry by its name.
So you can also do the following:

```
shell.getRegistry("global").addAnnotatedCommand(MyCommand.class);
```

Registering a Namespace at Demand (When a Command is Executed)

This use case is useful in features using connection like commands.
When such a feature is installed - it will register only the **connect** like command in the global namespace. But when **connect** command is executed the feature will execute the **remote** command namespace since a connection was established and remote commands can be used.

This is how Automation Feature is implemented.

```
public class MyRemoteFeature implements ShellFeature {
    ...

    public void install(Shell shell) {
       GlobalCommands.INSTANCE.addAnnotatedCommand(MyConnectCommand.class);
    }

    public CommandRegistry createRemoteRegistry() {
        CommandRegistry reg = new CommandRegistry();
        // add commands to registry here
        return reg;
    }

    ...
}
```

You can see the feature is simply installing the **connect** like command into the global namespace.
Also, it is providing a factory method for the remote registry which should be registered only when connected to server.

Now let's look at the `MyConnectCommand` implementation:

```
@Command(name = "connect", help = "Connect to a remote server")
public class MyConnectCommand implements Runnable {

    @Context
    protected Shell shell;

    @Argument(name = "url", index = 0, required = false, help = "The url of the
automation server")
    protected String url;

    @Parameter(name = "-u", hasValue = true, help = "The username")
    protected String username;

    @Parameter(name = "-p", hasValue = true, help = "The password")
    protected String password;

    public void run() {
       try {
         doConnect(url, username, password);
         CommandRegistry reg =
shell.getFeature(MyRemoteFeature.class).createRemoteRegistry();
         shell.addRegistry(reg);
         shell.setActiveRegistry(reg.getName());
       } catch (Exception e) {
         throw new ShellException("Failed to connect to "+url, e);
       }
    }

}
```

You can see here that after successfully connecting to the remote server we ask our feature instance to create our remote command registry, and then we simply register it in the shell.
Then we activate this registry so that the active namespace in the interactive shell will be the **remote** one.

In the same manner we can implement the disconnect method - after disconnecting it will unregister the remote namespace and switch on the **local** namespace.

Related documentation
• Extending The Shell • Nuxeo Shell administration documentation

Shell Commands

Let's look now into Command implementation details.

Creating A Command

When creating a command you should think about:

1. What my command is doing - choose a name, write a short description and define the command parameters.
2. Map the command parameters on object fields.
3. Specify any completors you want for the parameter values. If you need a new completor, write it and register it through the feature contributing your command.
4. implement the command logic.
5. register your command in a namespace.

I will demonstrate this on creating a command that is listing the children of remote document with an optional filter on the child document type.

Define the Command Syntax

My command is listing children given a type so I will name it **lst**. So it will take as parameters an option which is the type to filter on, and an optional argument which is the target document to list its children. If no type is given all children will be listed.
The document argument is optional because if not specified the current document will be used as the target document.

In this section

Any command should implement the `java.lang.Runnable` interface - since the **run** method will be invoked by the shell to execute the command.

```
@Command(name = "lst", help = "List children documents filtered by type")
public class Lst implements Runnable {
    public void run() {
    }
}
```

So, I've put a name on my command and a short description which will be used to generate the documentation.
You can also specify alias names for your command by filling the **aliases** attribute of the @Command annotation.
Example `aliases = "listByType"`

Let's go to the next step.

Map the Command Parameters on Object Fields

Now I need to map the command line arguments to fields on my Command object.

```
@Command(name = "lst", help = "List children documents filtered by type")
public class Lst implements Runnable {

    @Argument(name = "doc", index = 0, required = false, help = "A document to list
its content. If not specified list the current document content.")
    protected DocRef doc;

    @Parameter(name = "-type", hasValue = true, help = "The type to filter on
children. If not specified all children will be listed.")
    protected boolean uid = false;

    public void run() {
    }
}
```

So I mapped the target document on my **doc** field, and the -type option on my **type** field.
So when I will run my command these fields will be injected with the user options specified through the command line.

But how the document target which is specified as a path will become a **DocRef** object? This is because the Automation Feature is providing an Object Adapter from String to DocRef. It will convert a document path into a Java DocRef object which will be used by the automation client to reference a remote document.

Also, the **doc** field is not require since if it is not specified I will use the current document as the target of my command.

You can see that the **-type** option specify that waits for a value (the **hasValue** attribute). If this attribute is not specified the option will be assumed to be a flag (i.e. boolean value that activate a behavior when used).

Let's look at the next step.

Add Completors if Any

Now I want to add completors for my parameter values. Automation Feature is already providing a completor for documents. I will create a completor for possible children types.
Here is the type completor:

```
public class SimpleDocTypeCompletor extends SimpleCompletor {
    public DocTypeCompletor() {
        super(new String[] { "Workspace", "Section", "Folder",
                "File", "Note" });
    }
}
```

The `SimpleCompletor` is a helper provided by JLine to create simple completors. To create complex completors you may want to directly

implement the JLine `Completor` interface.

My completor only allows a few types to filter on: Workspace, Section, Folder, File and Note. To create a more useful completor we will want to make a query to the server for the available types in the repository.

Let's now specify my completor for the **-type** option and the `DocRefCompletor` provided by the Automation Filter for my **doc** argument.

```
@Command(name = "lst", help = "List children documents filtered by type")
public class Lst implements Runnable {

    @Argument(name = "doc", index = 0, required = false,
completor=DocRefCompletor.class, help = "A document to list its content. If not
specified list the current document content.")
    protected DocRef doc;

    @Parameter(name = "-type", hasValue = true,
completor=SimpleDocTypeCompletor.class, help = "The type to filter on children. If
not specified all children will be listed.")
    protected boolean uid = false;

    public void run() {
    }
}
```

Let's go to next step now.

Implement Command Logic

Now I want implement the command logic - the **run** method.
But first, I need explain some things made available by the Automation Feature.

The Automation Feature is keeping an object **RemoteContext** which reflects the state of our remote session. It provides remote API and hold things such as the current document in the shell. This object is made available for injection because it was registered by the Automation Feature as a **Context Object**. So let's inject that object:

```
@Command(name = "lst", help = "List children documents filtered by type")
public class Lst implements Runnable {
    @Context
    protected RemoteContext ctx;

    ...
    public void run() {
    }
}
```

Now in the **run** method we can use the **ctx** object to access to the server and to the remote context of the shell.

Don't Like Injection?
If you don't like injection you can always lookup yourself the context objects through the shell instance.
The `@Context protected RemoteContext ctx;` construct is eqiuivalent to

```
protected RemoteContext ctx =
Shell.get().getContextObject(RemoteContext.class)
```

Now we are ready to implement the **run** method. I will omit the fields declaration to have a more readable code:

```
public void run() {
        if (doc == null) {
            // get the current document if target doc was not specified.
            doc = ctx.getDocument();
        }
        ShellConsole console = ctx.getShell().getConsole();
        try {
            if (type == null) {
                for (Document child : ctx.getDocumentService().getChildren(doc)) {
                    DocumentHelper.printName(console, child);
                }
            } else {
                for (Document doc : ctx.getDocumentService().getChildren(doc)) {
                    if (type.equals(child.getType())) {
                        DocumentHelper.printName(console, child);
                    }
                }
            }
        } catch (Exception e) {
            throw new ShellException("Failed to list document " + path, e);
        }
    }
```

You can see that the Shell instance is retrieved from the context, but you can inject it as you injected the context or use `Shell.get()` constr uct to lookup the Shell instance.

The **DocumentHelper** is a class that provide a helper method to extract the name of a document from its path and print it on the console.

You can just use `console.println(doc.getPath());` if you want to print the full path of children.

The `ctx.getDocumentService()` is returning a helper service instance to deal with remote automation calls. If you want more control on what you are doing use `ctx.getSession()` and then use the low level API of Automation Client.

Register the Command in a Namespace

Now, our command is ready to run. We need just to register it before. For this either we are directly the **remote** namespace and add it - but for many of us this is not possible since you need to modify the Nuxeo Shell JAR. In that case we will create a new feature like explained in S hell Features.

```
public class MyFeature implements ShellFeature {
    public void install(Shell shell) {
       shell.getRegistry("remote").addAnnotatedCommand(Lst.class);
    }
}
```

And then register the feature as explained in Shell Features. Build your JAR and put it on the shell application classpath.
Now you are ready to use your own command.

> ⚠ The DocRef adapter is also supporting UID references not only paths .. to specify an UID you should prepend the "doc:" string to the UID.

Exception Handling

We've seen in the previous example that the run method is wrapping all exception and re-throw them as `ShellException` which is a runtime exception. This is the pattern to use to have the shell correctly handling exceptions. If you are not catching an exception it will end-up in the console printed in a red color. If you are sending a ShellException only the exception message is printed as an error and you can see the complete stack trace of the last exception using the **trace** command. This is true in interactive mode. In batch mode exception are printed as usual.

The Command Class

Here is our final class:

```
@Command(name = "lst", help = "List children documents filtered by type")
public class Lst implements Runnable {

    @Context
    protected RemoteContext ctx;

    @Argument(name = "doc", index = 0, required = false,
completor=DocRefCompletor.class, help = "A document to list its content. If not
specified list the current document content.")
    protected DocRef doc;

    @Parameter(name = "-type", hasValue = true,
completor=SimpleDocTypeCompletor.class, help = "The type to filter on children. If
not specified all children will be listed.")
    protected boolean uid = false;

    public void run() {
        if (doc == null) {
            // get the current document if target doc was not specified.
            doc = ctx.getDocument();
        }
        ShellConsole console = ctx.getShell().getConsole();
        try {
            if (type == null) {
                for (Document child : ctx.getDocumentService().getChildren(doc)) {
                    DocumentHelper.printName(console, child);
                }
            } else {
                for (Document doc : ctx.getDocumentService().getChildren(doc)) {
                    if (type.equals(child.getType())) {
                        DocumentHelper.printName(console, child);
                    }
                }
            }
        } catch (Exception e) {
            throw new ShellException("Failed to list document " + path, e);
        }
    }
}
```

Scripting Commands

As we've seen, remote commands are using the automation client API to talk with the server. This means our remote commands are in fact wrappers around a core automation object: an automation operation. A remote command is in fact translating the user input in a server side operation and is using the automation client API to invoke that operation. All these things are hidden in Nuxeo Shell but you can always use the automation client API and directly invoke operations if you want.

This is a nice feature since automation is used also on the server side to design high level actions on the Nuxeo platform. So we reuse the code existing in Nuxeo without having to deal with Nuxeo low level API. You can, this way, even assemble administration commands using Nuxeo Studio and invoke them from the shell.

But, the problem is that using operations means to have this operation defined on server side. It may happens that Nuxeo is not always providing an operation dedicated for your needs. In this case an approach is to implement a operation (server side), deploy it on the server and then create a shell command to invoke the operation. But you cannot do this anytime you need a new command in the shell and your target server is a production server. Also there are use cases not covered by operations like listing the existing ALCs on a document (in fact all listings that doesn't involves returning from the server documents or blobs).

To solve this issue we implemented a script operation. A script operation is a server side operation that takes as input a blob (i.e. a file) containing a script (in Groovy or Mvel) and execute this script on the server in the current shell context.
Using this feature you can do anything you want not covered by operations. You can do things like from how long the server is running? or to perform garbage collection or getting monitoring information o Nuxeo Services etc.

There are two way to use scripting in the shell:

1. Use the **script** command that takes as input the script file to execute and a context execution map of parameters.
2. Creating a command that wraps a script and provides typed parameters - completion aware.

In both cases **the script must return a result string**.

The first solution is OK, when you are creating Ad-Hoc scripts - like invoking GC on the server, you can use the "trace" command to get more information in case of failure.

But if you want well types commands with completors etc. then you need to implement a real command that will invoke your script.

So, we will focus now on the second option.

Invoking Scripts from a Command

To do this, you need first to write the script - let's say a Groovy script. This script should return a string - the result of the operation. Put this script somewhere in the JAR - let's say in "scripts/myscript.groovy" in your JAR root.

Then in your shell command you can invoke this script to be executed remotely with the arguments specified by the user on the command line by invoking:

```
String result = Scripting.run("scripts/myscript.groovy", ctx);
```

where **ctx** is a String to String map containing the user options to be forwarded to the script.
The script can access these options using `Context["key"]` expression where **key** is a name of a user variable passed in the **ctx** map.

As a real example you can see the `Perms` command of the shell and the `printAcl.groovy` script it is invoking.

Of course the output of the script (a string) may be a complex object - encoded as JSON or XML - in that case your command should be able to decode it and print a human readable response on the terminal.

Security Warning
Because of potential security issues the scripting feature is available only when logged in as Administrator

Here is a complete example of a script used by the **perms** command to get the ACLs available on a document:

```
import org.nuxeo.ecm.core.api.PathRef;
import org.nuxeo.ecm.core.api.IdRef;
import org.nuxeo.ecm.core.api.security.ACP;
import org.nuxeo.ecm.core.api.security.ACE;
import org.nuxeo.ecm.core.api.security.ACL;


def doc = null;
def aclname = Context["acl"];
def ref = Context["ref"];
if (ref.startsWith("/")) {
  doc = Session.getDocument(new PathRef(ref));
} else {
  doc = Session.getDocument(new IdRef(ref));
}
def acp = doc.getACP();
def result = null;
if (aclname != null) {
  def acl = acp.getACL(aclname);
  if (acl == null) {
    result = "No Such ACL: ${aclname}. Available ACLS: ";
    for (a in acp.getACLs()) {
      result+=a.getName()+" ";
    }
    return result;
  }
  result = "{bold}${aclname}{bold}\n";
  for (ace in acl) {
    result += "\t${ace}\n";
  }
} else {
  result = "";
  for (acl in acp.getACLs()) {
    result += "{bold}${acl.name}{bold}\n";
    for (ace in acl) {
      result += "\t${ace}\n";
    }
  }
}

return result;
```

Related documentation

- Extending The Shell
- Nuxeo Shell administration documentation

Shell Namespaces

We are getting now to the end of our breakthrough about extending the shell.
We've already seen how to add new Shell Features and how to implement new commands. This last chapter is talking a bit about Shell Namespaces.

The Shell Prompt

As we've already seen, namespaces are in fact hierarchical registries of commands. When entering a namespace all other namespaces not extended by the current namespace are hidden for the shell user. But there is still a point we have not yet discussed yet - the fact that a namespace may change the shell prompt when it is activated. This is important because it is a visual feedback for the user that it switched to another context.

In this section

To explain how a namespace is changing the shell prompt I will give the example of the **local** namespcae provided by the `FileSystem` featu re. It's really easy: the FileSystem feature is registering and activating the **local** namespace which is implemented by a subclass of `Command Regsitry`. The `CommandRegistry` is providing a method that any subclass may override: **String getPrompt(Shell shell)**.

So when creating a new namespace (i.e. CommandRegistry) you can override this method to return the prompt for your namespace.
Here is the implementation of **String getPrompt(Shell shell)** of the FileSystemCommands class (which is the materialization of the **local** na mespace):

```
public String getPrompt(Shell shell) {
        return System.getProperty("user.name") + ":"
                + shell.getContextObject(FileSystem.class).pwd().getName()
                + "$ ";
    }
```

So we can see that the prompt is dynamically updated after each command execution to reflect the new context. In this case we print the local username and the name of the current directory.

The Default Namespace

The shell will activate by default the **remote** namespace if any with that name exists, otherwise it is activating the **local** namespace if any such namespace exists, otherwise the **global** namespace is activated.

To force a namespace to be activated when the shell starts you can use the **shell** Java System Property when launching the shell application. For example, the following will activate the namespace **equinox** when the shell starts:

```
java -Dshell=equinox -jar nuxeo-shell.jar
```

Of course, you should provide an "equinox" namespace to have this working - otherwise the default shell namespace will be used.

Executing Initialization Code at Startup

If your namespace need to execute some code when the shell is launched and your namespace is activated (because you set it as the default one) then you should override the method `CommandRegistry.autorun(Shell shell)` and provide your logic there. You can use this to automatically connect to a remote host if the connection details were specified on the command line when starting the shell. See the AutomationFeature "remote" namespace - this is the way it is automatically connecting when you specify the connection details on the shell command line.

Related documentation

- Extending The Shell
- Nuxeo Shell administration documentation

Shell Documentation

Commands are self documented by using annotations. These annotations are used to generate the base of the command documentation, like: short description, syntax, options, arguments etc. Using this information the shell automatically generates basic help for you.

The generated documentation is available in both text format (for terminal) and wiki format. See Nuxeo Shell Command Index for

the wiki format.

This is the minimal documentation provided by any command and it may be enough for simple commands. But for more complex commands you may want to give more in depth instructions or examples.
To do this you can create a text file that hase the same name as the short name of the command class plus the **.help** extension. This file must reside in the JAR in the same package as the command class.

Technical Detail
The **help** file class is located at runtime using the command class getResource method and the naming pattern specified above.

The help file is an ASCII text file which may contain some special tags to control the help rendering in both terminal or wiki mode.
The most important tags are:

- **{header}** - can be used to use a bold font for a title. Working in both terminal and wiki.
  Usage:

```
{header}EXAMPLES{header}
Here is an example of ...
```

- **{bold}** - same as **{header}**.
- **{code}** - can be used to escape code - working **only** in wiki, ignored (tag is removed) in terminal.

Also, you can use any confluence {tag}. It will be correctly displayed in the wiki and ignored in the terminal.
There is also a set of tags that are working only in terminal:

- underscore
- blink
- reverse
- concealed

Color tags:

- black
- red
- green
- yellow
- blue
- magenta
- cyan
- white

Background tags:

- bg.black
- bg.red
- bg.green
- bg.yellow
- bg.blue
- bg.magenta
- bg.cyan
- bg.white

In addition you can also use any numeric control code as the tag name. See JLine documentation for more details on the terminal codes.

Anyway it is **recommended** to only use **header**, **bold** and other wiki tags like **code** since terminal tags are not translated in wiki tags.

Important
When contributing new commands in the Nuxeo Shell core library (or modifying old ones) please make sure you update the wiki at Nuxeo Shell Command Index.

Here are instructions on how to synchronize the wiki page with the modifications in Nuxeo Shell:

1. Export in wiki format the namespace you modified:

```
java -jar nuxeo-shell.jar -e "connect -u Administrator -p Administrator
http://localhost:8080/nuxeo/site/automation; help -export tofile.wiki -ns
remote"
```

The connect command is needed if you want to export the **remote** namespace. For local namespaces you doesn't need to connect to a server.

2. Copy/Paste the content of the exported wiki file to the Wiki Page under Nuxeo Shell Command Index corresponding to the updated namespace.

Related documentation
• Extending The Shell • Nuxeo Shell administration documentation

Nuxeo Android Connector

What is Nuxeo Android Connector

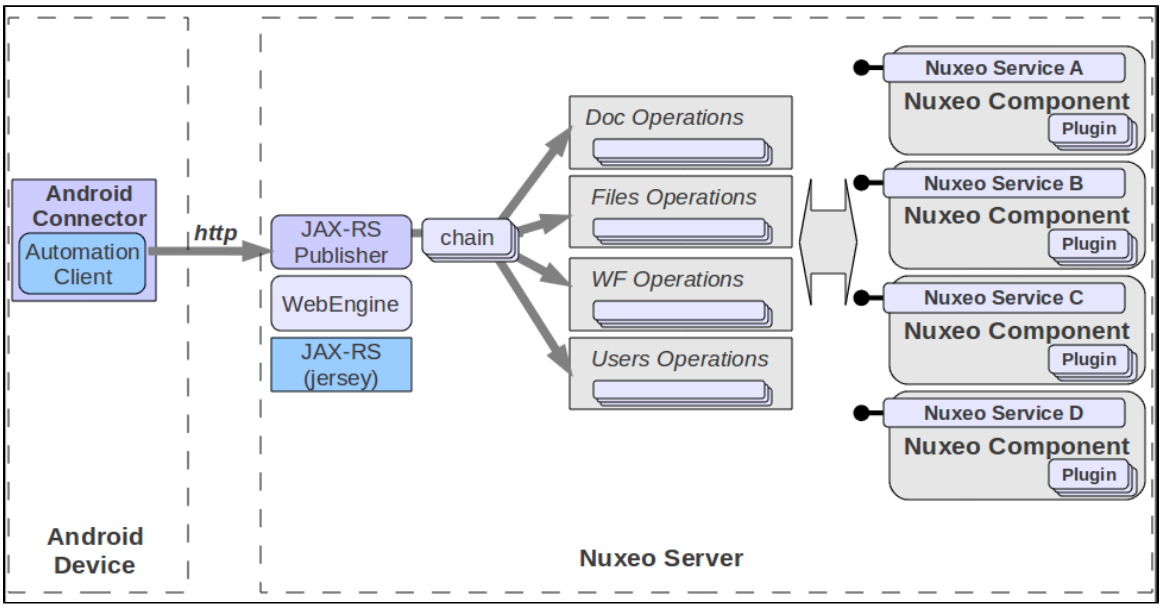
Nuxeo Android Connector is a SDK to build Android Applications that communicate with a Nuxeo Server.

Nuxeo Content Automation

Nuxeo SDK for Android is based on Nuxeo Content Automation. Basically, the Android will use Automation REST API to communicate with the Nuxeo server.

Using Automation brings several advantages :

- API calls is simple and very extensible
  (you can contribute new operations or chains via Nuxeo Studio or Nuxeo IDE)
- Calls are efficient and easily cachable (REST + JSON Marshaling)
- Thanks to Operation Chains, you can have a single call executing several operations within the same transaction



The heart of Nuxeo SDK for Android is the Nuxeo Automation Client.

The version of the Automation Client is slightly different from the standard Java Automation Client because some dependencies are not the same (using Android SDK JSON on Android and Jackson on the standard Java).
But the API and the logic remain the same, only internal implementation is a little bit different.

In addition, as it will be presented later, the Nuxeo SDK for Android provides more than just the Automation Client.

Android Connector Content Overview

The Android Connector is a single library that provides the required infrastructure to build an Android application that uses services and content from a Nuxeo server.

This connector includes :

- a Nuxeo Automation client
- caching extension to Automation Client to manage offline mode
- additional services (FileUploader, FileDownloader ...)
- a model to manage synchronizable Document lists with support for offline usage
- an integration layer to expose Nuxeo concepts the Android Way (Content Providers, BroadCastReceivers, Services ...)
- a layout system to be able to reuse Nuxeo Layout's definition to build forms and views on Android
- base classes for building an Android Application

Getting the Connector and the Source Code

Source code for Nuxeo Android Connector is available in Nuxeo's GitHub .

Sample Application

Nuxeo Automation client

In the Android Connector, the Automation Client is associated to a context that references required dependencies :

- server configuration and credentials
- network status information
- Android Context (required for Filesystem and SQL Db access)

You can access this `NuxeoContext` by using :

```
NuxeoContext.get(Context context)
```

context being the Android Application Context.

If you use the base class `SimpleNuxeoApplication` for your application, you can directly call `getNuxeoContext()` on the application.

The main advantages of using the `SimpleNuxeoApplication` base class is that :

- access to NuxeoContext is simpler
- the lifecycle of NuxeoContext will be bound to your application (instead of being a static singleton)

NB : you should use one method to access the `NuxeoContext`, but you should not mix them ...

Once you have the `NuxeoContext` you can directly access to the settings and the Automation Session :

```
ctx.getServerConfig().setLogin("jdoe");
ctx.getServerConfig().setPassword("secret");
ctx.getServerConfig().setServerBaseUrl("http://10.0.2.2:8080/nuxeo/");

if (ctx.getNetworkStatus().isNuxeoServerReachable()) {
  Document doc = (Document)
ctx.getSession().newRequest("Document.Fetch").set("value", docRef).execute();
}
```

Outside of this direct association between the Context and the Automation Client Session, the standard documentation for Nuxeo Automation Client does apply to the Automation Client embedded in Android Connector.

Android Connector and Caching

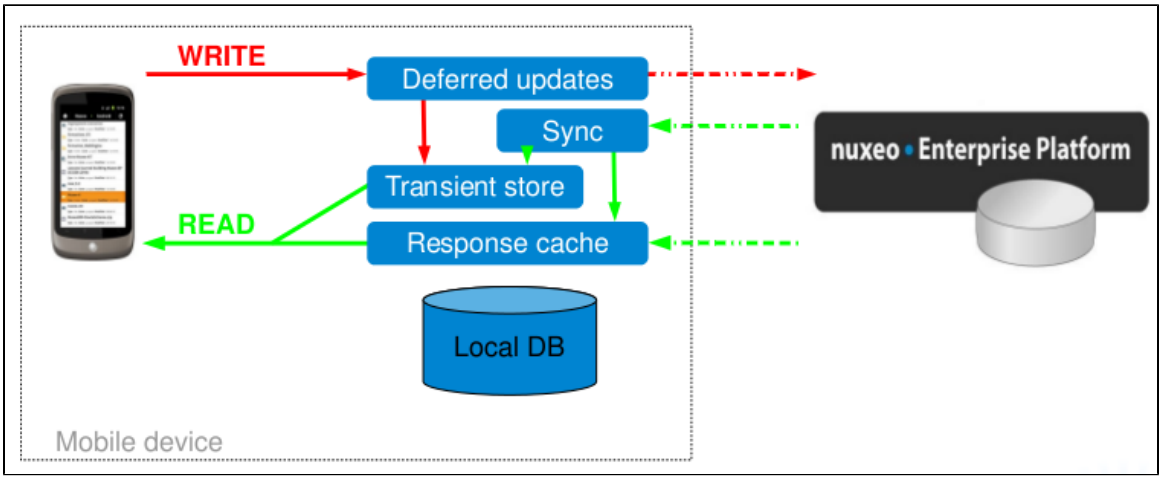
Because Android devices don't always have access to a reliable network connection, Nuxeo Android Connector manages local caching. The cache data is maintained by a SQLite DB and Filesystem storage.

Several types of caches are managed:

- `ResponseCache`: the server response is simply stored on the device so that it can be reused in case the server is unreachable.
  Typical use case is caching the result of a query so that you can continue browsing the list of documents, even if the network has gone.
- `TransientState`: because you have the ability to create or update documents from the Android device, you need to be able to store your changes locally.
  Typical use case is that you update a document and you want to keep this local change until your are able to send it back to the server and get a fresh copy.
- `DeferredUpdate`: creating or updating a document also means calling an operation on the server side. The create/update operation will be stored locally and replayed when the server is available.

In this section

- ResponseCache
- TransientState
- Deferred Updates



ResponseCache

Automation responses can be cached. This basically means that the JSON response sent by the Nuxeo server is stored on the local filesystem and is associated with a DB record that maintains metadata.
The extra metadata in the SQL DB are used to be able to match a response with the corresponding request. The DB also contains the informations to be able to reconstruct the request, so that the cache can be refreshed.

Compared to the "standard Automation Client API", when calling an operation you can specify expected caching behavior.

```
DocRef docRef = new PathRef("/");

  // Call with no CacheBehavior => default to CacheBehavior.STORE
  session.newRequest("Document.GetChildren").setInput(docRef).execute();

  // Call with CacheBehavior.STORE => will cache the response

session.newRequest("Document.GetChildren").setInput(docRef).execute(CacheBehavior.STOR
E);

  // Force refresh (try to fetch from server, store to cache)

session.newRequest("Document.GetChildren").setInput(docRef).execute(CacheBehavior.FORC
E_REFRESH);
```

TransientState

This cache stores document deltaset (changes done in the document). This includes newly created documents that do only exist in local.

The TransientState manager is mainly updated via events (`AndroidTransientStateManager` is a `BroadcastReceiver`).

This design helps making the TransientState management as transparent as possible. When a Document is created or update in local, a event is sent : TransientStateManager stores the delta. When the create/update operation has been processed by the server a new event will be fired and the TransientStateManager will delete the local storage.

To reflect the synchronization status, the Document has a `getStatusFlag()` that returns an Enum:

- `"new"` means that the document only exists in local for now,
- `"updated"` means that the document hold local modifications that have not yet been pushed to the server,
- `""` means that the document is synchronized with the server,
- `"conflict"` means that the push on the server resulted in a conflict,

`TransientStateManager` exposes an API to automatically fetch and merge deltasets on a List of Documents, but in most of the cases this is already encapsulated by the DocumentProviders.

Deferred Updates

This caches keeps track of the Create/Update/Delete operations that are pending and should be sent to the server when network is accessible. Each cached operation is indirectly linked to a set of TransientState. In addition, pending Request can have dependencies:

- dependencies between update requests,
- dependencies with pending Uploads.

Deferred Updates system is exposed via `DeferredUpdateManager` service interface. This service can be used to send an update request:

```
String execDeferredUpdate(OperationRequest request, AsyncCallback<Object> cb,
OperationType opType, boolean executeNow);
```

Android Connector additional Services

In addition of the Automation Client, Android Connector exposes several services :

Built in services

`TransientStateManager`

Manages local storage of locally modified Documents.

`DeferredUpdateManager`

Manages storage of updates operations until the network becomes available again.

`ResponseCacheManager`

Manage Storage of server responses.

`FileDownloader`

This service is also very tied to the caching system.
It manages download in a ThreadPool and caches the result on the Filesystem.

`FileUploader`

This service works side by side with the `DeferredUpdateManager`.

For Document create/update operations, the Blobs are not directly sent inside the Create/Update request.
Basically, the Blobs are uploaded via the `FileUploader` service and the Create/Update operation will contain a reference to the upload batch.
In order for this to work correctly the DeferredUpdate will have a dependency on the required uploads.

`NuxeoLayoutService`

Accessing the services

Services are accessible via simple getters on the `NuxeoContext` and `AndroidAutomationClient` objects.
You can also use the adapter system on the Session object to have direct access to all the services.

```
FileUploader uploader = getNuxeoSession().getAdapter(FileUploader.class);

FileDownloader downloader = getNuxeoSession().getAdapter(FileDownloader.class);

DeferredUpdateManager deferredUpdateManager =
getNuxeoSession().getAdapter(DeferredUpdateManager.class);

TransientStateManager transientStateManager =
getNuxeoSession().getAdapter(TransientStateManager.class);

ResponseCacheManager responseCacheManager =
getNuxeoSession().getAdapter(ResponseCacheManager.class);

NuxeoLayoutService nuxeoLayoutService =
getNuxeoSession().getAdapter(NuxeoLayoutService.class);

DocumentMessageService documentMessageService =
getNuxeoSession().getAdapter(DocumentMessageService.class);

DocumentProvider documentProvider =
getNuxeoSession().getAdapter(DocumentProvider.class);
```

DocumentProviders in Android Connector

DocumentProvider and LazyDocumentsList

The DocumentProvider system is basically an extension of the PageProvider notion that exists on the Nuxeo server side.

The `DocumentProvider` service allows to access a list of documents by it's name.

```
 LazyUpdatableDocumentsList documentsList = docProvider.getDocumentsList("MyList",
getNuxeoSession());
```

This list of documents will be fetched using an Automation Operation.
This means the content of the document list can be propulated from :

- children of a Folder
- content of a InBox (CMF use case)
- the Clipboad or the Worklist
- the result of a NXQL query
- ...

When returning lists of documents, the DocumentProvider does not provide a simple `Documents` type.
It returns a `LazyDocumentsList` type that provides additional features :

- list will be automatically fetched asynchronously page by page as needed
- the list will be cached locally to be available offline
- you can add documents to the list (even in offline mode)
- you can edit documents to the list (even in offline mode)
- list definition can be dynamically saved so that you can restore it later

The `DocumentProvider` can be accessed like any service :

```
DocumentProvider docProvider = getNuxeoSession().getAdapter(DocumentProvider.class);
```

DocumentProvider gives access to named list of documents. These lists implement the `LazyDocumentsList` interface and if they support create/update operation they also implement `LazyUpdatableDocumentsList`.

You can define your own `LazyDocumentsList` and register them to the `DocumentProvider` :

```
// register a query
String query = "select * from Document where ecm:mixinType != \"HiddenInNavigation\"
AND ecm:isCheckedInVersion = 0 AND ecm:currentLifeCycleState != \"deleted\" order by
dc:modified DESC";
docProvider.registerNamedProvider(getNuxeoSession(),"Simple select", query , 10,
false, false, null);

// register an operation
// create the fetch operation
OperationRequest getWorklistOperation =
getNuxeoSession().newRequest("Seam.FetchFromWorklist");
// define what properties are needed
getWorklistOperation.setHeader("X-NXDocumentProperties", "common,dublincore");
// register provider from OperationRequest
docProvider.registerNamedProvider("Worklist", getWorklistOperation , null, false,
false, null);

// register a documentList
String query2 = "SELECT * FROM Document WHERE dc:contributors = ?";
LazyUpdatableDocumentsList docList = new
LazyUpdatableDocumentsListImpl(getNuxeoSession(), query2, new
String[]{"Administrator"}, null, null, 10);
docList.setName("My Documents");
docProvider.registerNamedProvider(docList, false);
```

When registering a new provider, you can ask for it to be persisted in the local db. The list definition will be saved to the db and the content will be cached.
This allows the end used to define custom lists of documents that will benefit from cache and offline support.

When you need to access one of the named lists you can simply ask the `DocumentProvider` :

```
LazyUpdatableDocumentsList documentsList = docProvider.getDocumentsList(providerName,
getNuxeoSession());
```

The Nuxeo Connector provides and Android `ListAdapter` so that you can directly bind the document lists to an Android `ListView`.
(this part will be explained in more details in the next section).

Create and Update operations on `LazyDocumentsList`

You can add or edit documents :

```
// add a document
Document newDocument = ...
documentsList.createDocument(newDocument);

// update a document
Document doc2Update = documentsList.getDocument(idx);
doc2Update.set("dc:title", "Modified!");
documentsList.updateDocument(Update);
```

The actual implementation on the server side of the create/update operations will depend on your business logic.
Typically, if you list represent the content of a folder or a list of documents matching a topic, you will have different create/update/delete implementations.

List nature	Create operation	Update operation	Delete operation
Folder contents	Create document in Folder	Update document	Delete document
query on topic	Create document in personnal workspace and set topic meta-data	Update document	unset target meta-data
inbox content	Get next document from service mailbox and assign to me	Update document	distribute to next actor of the workflow

The default implementation create the document with a path that can be configured and does a simple update of the document.

```
protected OperationRequest buildUpdateOperation(Session session, Document
updatedDocument) {
 OperationRequest updateOperation =
session.newRequest(DocumentService.UpdateDocument).setInput(updatedDocument);
 updateOperation.set("properties",
updatedDocument.getDirtyPropertiesAsPropertiesString());
 updateOperation.set("save", true);
 updateOperation.set("changeToken", updatedDocument.getChangeToken()); // prevent
dirty updates !
 // add dependency if needed
 markDependencies(updateOperation, updatedDocument);
 return updateOperation;
}

protected OperationRequest buildCreateOperation(Session session, Document newDocument)
{
 PathRef parent = new PathRef(newDocument.getParentPath());
 OperationRequest createOperation =
session.newRequest(DocumentService.CreateDocument).setInput(parent);
 createOperation.set("type", newDocument.getType());
 createOperation.set("properties",
newDocument.getDirtyPropertiesAsPropertiesString());
 if (newDocument.getName()!=null) {
  createOperation.set("name", newDocument.getName());
 }
 // add dependency if needed
 markDependencies(createOperation, newDocument);
 return createOperation;
}
```

You can use your own operation definitions by inehit from `AbstractLazyUpdatebleDocumentsList` and simple implement the 2 methods bu `ildUpdateOperation` and `buildCreateOperation`.

Android SDK Integration

The Nuxeo Android SDK tries, as much as possible, to expose Nuxeo services in a natural Android way.
The idea is that the SDK should it's features via standard Android concepts and patterns.

`DocumentsListAdapter`

`DocumentsListAdapter` is a dedicated implementation of the standard Android interface `ListAdapter`.
It allows to bind an Android `ListView` to a `LazyDocumentsList`.

```
// get the document list
LazyDocumentsList documentsList = getDocumentList(data);

// define the mapping between document attributes and widgets
Map<Integer, String> mapping = new HashMap<Integer,String>();
mapping.put(R.id.title_entry, "dc:title");
mapping.put(R.id.status_entry, "status");
mapping.put(R.id.iconView, "iconUri");
mapping.put(R.id.description, "dc:description");
mapping.put(R.id.id_entry, "uuid");

// create the adapter passing it the list, the mapping and the layout
DocumentsListAdapter adapter = new DocumentsListAdapter(this, documentsList,
R.layout.list_item, mapping, R.layout.list_item_loading);

// bind to the ListView
listView.setAdapter(adapter);
```

Starting from there you can use your Nuxeo doculent list like any simple list.
The documents list will be fetched and refreshed automaticaly as needed.

If scrolling goes faster than fetching from the server, a waiting item will be displayed with a specific layout (`R.layout.list_item_loading` in the above exemple).

`ContentProvider`

Nuxeo's SDK try to expose as much as possible of the content via the Android `ContentProvider` system.

The provider authority is `nuxeo` and depending on the requested `URI` content, the call will be directed to a nuxeo service.
Basically :

URI pattern	Target Content
`content://nuxeo/documents`	returns all documents of the repository via an Android Cursor
`content://nuxeo/documents/<UUID>`	access to document with given UUID
`content://nuxeo/<providername>`	Android cursor documents in the given provider
`content://nuxeo/<providername>/UUID`	access to document with UUID in the given provider
`content://nuxeo/icons/<subPath>`	download and cache icon of the given sub path
`content://nuxeo/blobs/<UUID>`	download and cache the main blob of the doc with the given UUID
`content://nuxeo/blobs/<UUID>/<idx>`	download and cache the blob <idx> of the doc with the given UUID
`content://nuxeo/blobs/<UUID>/<subPath>`	download and cache the blob contained in the field <subpath> of the doc with the given UUID

This ContentProvider allows :

- to easily bind Nuxeo resources (like images) to Androids Views (like an `ImageView`)
- to easily use Nuxeo content from an external application
  - Interprocess marshaling is handled by the ContentProvider system
  - you don't need to depend on Nuxeo API

Event system

Android built-in event system is used by the SDK to notify for :

- Network status changes: when the Nuxeo server becomes reachable or when offline mode is required
- Configuration changes : when the settings of the NuxeoAutomationClient have been changed
- Document events : A notification is sent for Create/Update/Delete operations on `LazyDocumentsLists` (with 3 states Local, Server, Failed)

Service binding

TBD

Nuxeo Layout in Android

Nuxeo Android Connector SDK allows fetch the Documents layout definition from the server.
This allows to reuse on the Android side the layouts that are present by default in Nuxeo, or the custom ones that can be done via Nuxeo Studio.

Basically, the Layout definitions (as well as Widgets definitions and some vocabularies) are exported in JSON by the Nuxeo server.
On the Android side this definition is cached and used to build an Android View from it.

This implies to bind Nuxeo widgets to Android native Widgets.
The current SDK version provides support for basic fields (Text, TextArea, Date, File, SelectOne, SelectMany).

```
// get a ScrollView that will contains the Form generated from the Nuxeo Layout
layoutContainer = (ScrollView) findViewById(R.id.layoutContainer);

// get the service
NuxeoLayoutService nls = getNuxeoSession().getAdapter(NuxeoLayoutService.class);

// get the layout definition for the current document.
NuxeoLayout layout = nls.getLayout(this, getCurrentDocument(), layoutContainer,
getMode());
```

`LayoutMode` argument can be create/edit/view.

When you wan to change back the modifications to the document :

```
Document doc = getCurrentDocument();
layout.applyChanges(doc);
```

Because some widgets can start new `Activity`, the activity responsible to handle the display of the form will need to implment some additional callback.

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
 layout.onActivityResult(requestCode, resultCode, data);
 super.onActivityResult(requestCode, resultCode, data);
}
```

See sample code and base classes for more details.

SDK provided base classes

The SDK tries to provide as much as possible base classes that you can rely on to avoid having to do too much plumbing.

Class Name	Type	Description

SimpleNuxeoApplication	Application	Integrate `NuxeoContext` with application context
BaseNuxeoActivity	Activity	Provide skelton for async call to the Nuxeo server on activity initialization
AbstractNetworkSettingsActivity	Activity	Skeleton for managing Nuxeo network and caching settings
AbstractNuxeoSettingsActivity	Activity	Skeleton for managing Nuxeo server settings
BaseSampleListActivity	Activity	Manage display of a list of documents

Additional Modules

This chapter presents how to apprehend and customize the additional packages available on the Nuxeo Platform, typically from the Nuxeo Marketplace.

- Amazon S3 Online Storage
- Automated Document Categorization
- Digital Asset Management (DAM) — The navigation is not document-centric and organized in collaborative workspaces as in Nuxeo DM but is instead search-centric. Furthermore, the application ergonomy should feel like a rich collection browser such as iTunes or iPhoto by extensively using Ajax. Speed and responsiveness to user actions should be considered a major feature of Nuxeo DAM.
- Digital Signature — Digital Signature implements a layout on signed documents to display the signature information, which can be customized.
- Document Access Tracking
- Nuxeo Agenda
- Nuxeo - BIRT Integration
- Nuxeo Bulk Document Importer — The nuxeo-importer-core module is designed to offer support for multi-threaded import on a Nuxeo repository.
- Nuxeo CSV — If you installed the Nuxeo CSV addon from the Nuxeo Marketplace, you'll probably want to enable CSV import on the document types you defined, either in Studio or with some code. Here is how to do that.
- Nuxeo DAM PDF Export
- Nuxeo Diff
- Nuxeo Drive
- Nuxeo Groups and Rights Audit
- Nuxeo jBPM
- Nuxeo jBPM: Enable jBPM Workflow on Your Document Type — This will be useful if you want to use the jBPM workflows. Default workflows enable you to control the life cycle state through a series or a set of human validation tasks configured by the the workflow initiator himself.
- Nuxeo Jenkins Report
- Nuxeo Multi-Tenant
- Nuxeo Platform User Registration
- Nuxeo Poll
- Nuxeo Quota: Enabling Quotas on Document Types — By default, quotas are available on domains and workspaces only. It is possible to enable them on other document types using Nuxeo Studio:
- Nuxeo RSS Reader
- Nuxeo Shared Bookmarks
- Nuxeo Sites and Blogs
- Resources Compatibility — The Resources Compatibility add-on provides backward compatibility with web resources (icons, JavaScript, ...) that have been removed from the previous LTS release of Nuxeo Platform.
- Smart Search
- Unicolor Flavors Set

Amazon S3 Online Storage

The Amazon S3 Online Storage is a Nuxeo Binary Manager for S3. It stores Nuxeo's binaries (the attached documents) in an Amazon S3 bucket.

Before You Start

You should be familiar with Amazon S3 and be in possession of your credentials.

Installing the Package

Use the Update Center to install the package from the Nuxeo Marketplace.

In this section:

Configuring the Package

In order to configure the package, you will need to change a few Nuxeo templates, and provide values for the configuration variables that define your S3 credentials, bucket and encryption choices

Specifying Your Amazon S3 Parameters

In `nuxeo.conf`, add the following lines:

```
nuxeo.s3storage.bucket=your_s3_bucket_name
nuxeo.s3storage.awsid=your_AWS_ACCESS_KEY_ID
nuxeo.s3storage.awssecret=your_AWS_SECRET_ACCESS_KEY
```

If you installed the bundle JAR manually instead of using the marketplace package you will also need:

```
nuxeo.core.binarymanager=org.nuxeo.ecm.core.storage.sql.S3BinaryManager
```

The bucket name is unique across all of Amazon, you should find something original and specific.

The file `nuxeo.conf` now contains S3 secret access keys, you should protect it from prying eyes.

You can also add the following optional parameters:

```
nuxeo.s3storage.region=us-west-1
nuxeo.s3storage.cachesize=100MB
```

The region code can be:

- for us-east-1 (the default), don't specify this parameter
- for us-west-1 (Northern California), use `us-west-1`
- for eu-west-1 (Ireland), use `EU`
- for ap-southeast-1 (Singapore), use `ap-southeast-1`

Since 5.6, you can also use:

- for us-west-2 (Oregon), `us-west-2`
- for ap-southeast-2 (Tokyo), use `ap-southeast-2`
- for sa-east-1 (Sao Paulo), use `sa-east-1`

Crypto Options

With S3 you have the option of storing your data encrypted (note that the local cache will *not* be encrypted).

The S3 Binary Manager can use a keystore containing a keypair, but there are a few caveats to be aware of:

- The Sun/Oracle JDK doesn't always allow the AES256 cipher which the AWS SDK uses internally. Depending on the US export restrictions for your country, you may be able to modify your JDK to use AES256 by installing the "Java Cryptography Extension

Unlimited Strength Jurisdiction Policy Files". See the following link to download the files and installation instructions: http://www.oracle.com/technetwork/java/javase/downloads/index.html

- Don't forget to specify the key algorithm if you create your keypair with the `keytool` command, as this won't work with the default (DSA). The S3 Binary Manager has been tested with a keystore generated with this command:

```
keytool -genkeypair -keystore </path/to/keystore/file> -alias <key alias>
-storepass <keystore password> -keypass <key password> -dname <key distinguished
name> -keyalg RSA
```

If you get `keytool error: java.io.IOException: Incorrect AVA format`, then ensure that the distinguished name parameter has a form such as: `-dname "CN=AWS S3 Key , O=example, DC=com"`.

Don't forget to **make backups of the `/path/to/keystore/file` file** along with the **store password, key alias and key password** . If you lose them (for instance if the EC2 machine hosting the Nuxeo instance with the original keystore is lost) you will lose the ability to recover any encrypted blob from the S3 backet.

With all that above in mind, here are the crypto options that you can add to `nuxeo.conf` (they are all mandatory once you specify a keystore):

```
nuxeo.s3storage.crypt.keystore.file=/absolute/path/to/the/keystore/file
nuxeo.s3storage.crypt.keystore.password=the_keystore_password
nuxeo.s3storage.crypt.key.alias=the_key_alias
nuxeo.s3storage.crypt.key.password=the_key_password
```

The Nuxeo `S3BinaryManager` class is using S3 Client-Side Encryption instead of S3 Server-Side Encrytption. CSE is safer than SSE. With CSE an attacker need both access to the **AWS credentials and the key** to be able to access the unencrypted data while SSE will only require the potential attacker to provide the **AWS credentials**.

Connection Pool Options

Since Nuxeo 5.8.0-HF06 (and Nuxeo 5.9.2 and Nuxeo 5.6.0-HF30) you can configure the internal S3 connection pool. This pool has a size of 50 by default, so if you've configured Nuxeo to use more sessions than this and all the sessions are accessing S3, you may run out of connections.

The following parameters can be used to change some connection pool parameters (the defaults are shown):

```
nuxeo.s3storage.connection.max=50
nuxeo.s3storage.connection.retry=3
nuxeo.s3storage.connection.timeout=50000
nuxeo.s3storage.socket.timeout=50000
```

The timeouts are expressed in milliseconds.

You can read more about these parameters on the AWS ClientConfiguration documentation page.

Checking Your Configuration

To check that installation went well, you can check your startup logs and look for a line like:

```
INFO  [S3BinaryManager] Repository 'default' using S3BinaryManager
```

Don't forget to enable the `INFO` level for the group `org.nuxeo` in `$NUXEO_HOME/lib/log4j.xml` to see INFO level messages from Nuxeo classes.

If your configuration is incorrect, this line will be followed by some error messages describing the problems encountered.

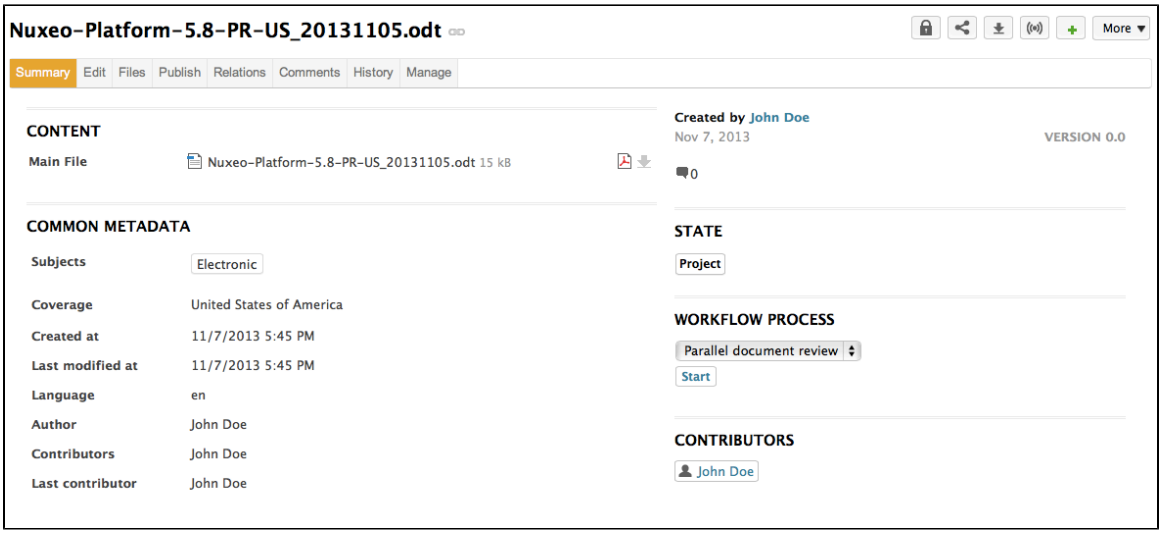
Automated Document Categorization

The Automated Document Categorization package enables the system to automatically fill in some metadata of the document when it is

created, from the document's content.

The Automated Document Categorization package requires no specific installation steps. It can be installed like any other package from the Marketplace or from the Admin Center.

After the package is installed, new documents directly have some metadata filled in.



Other documentation about this package

- Automated Document Categorization user documentation

Digital Asset Management (DAM)

Functional Overview

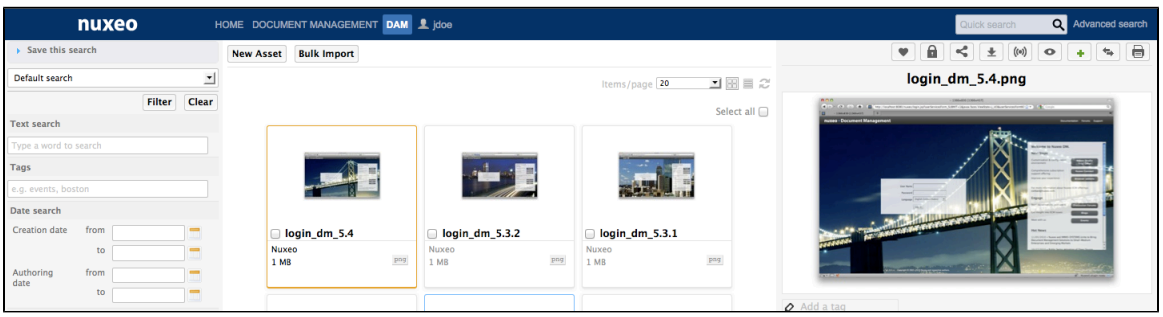
The Digital Asset Management module of the Nuxeo Platform provides organizations with an application to manage their digital assets which can be office (MS, OOo, PDF, ...) or multimedia (pictures, audio and video) files.

The main difference with Nuxeo DM (Document Management) is that DAM is not aimed at "producing" new documents with collaborative editing / review workflows but merely at browsing an existing collection that have been authored externally and imported in batch in the DAM application.

User will merely edit the properties / categories or add annotations to enrich the asset without editing the main attached file. The main edit actions would preferably take place in the Document Management UI.

The Digital Asset Management module requires no specific installation steps. It can be installed like any other package from the Marketplace or from the Admin Center. It is also possible to install it from the Startup wizard.

After the package is installed, users have a DAM tab on top of the application, next to the Document Management tab.



On this page

Ajaxified search-centric UI

The navigation is not document-centric and organized in collaborative workspaces as in Nuxeo DM but is instead search-centric. Furthermore, the application ergonomy should feel like a rich collection browser such as iTunes or iPhoto by extensively using Ajax. Speed and responsiveness to user actions should be considered a major feature of Nuxeo DAM.

Ajax features are implemented using the Rich faces components and the Ajax4JSF extensions. The online demo is available here: http://livedemo.exadel.com/richfaces-demo/richfaces/support.jsf?c=support&tab=usage

Some a4j tips are gathered on the following wiki page: Ajax4jsf Best Practices

In addition to JSFs Ajax features, we plan to also reuse existing Ajax features of Nuxeo DM such as the right click actions menu based on jQuery plugins.

Development with Nuxeo DAM

Nuxeo DAM source files can be retrieved for development using GitHub.

Nuxeo DAM Source code

The source code specific to the DAM project is located here: https://github.com/nuxeo/nuxeo-dam

To clone the Nuxeo DAM repository, use the following:

```
git clone https://github.com/nuxeo/nuxeo-dam
```

The main development branch is `master`. To switch to this branch, use:

```
git checkout master
```

You can also update on stable branches, or releases tags.

Stable branches are named from the Nuxeo DAM version: `1.0`, `1.1`, ...

Release tags are named from the release version of Nuxeo DAM: `release-1.0`, `release-1.1`

Source code layout

The project layout is as follows:

- `nuxeo-dam-api`
  Common classes used in other modules (Constants, Exceptions, ...)
- `nuxeo-dam-core`
  Core document schemas definitions and other contributions to services
- `nuxeo-dam-importer`
  Contains the core importer classes to be used during the import process and JAX-RS classes to be able to use the HTTP importer.
- `nuxeo-dam-web`
  Web resources for Nuxeo DAM

Continuous integration

The automated build / selenium test reports are hosted there: http://qa.nuxeo.org/jenkins/view/DAM/

Resources

- To help us prioritize new features, please see JIRA. You can vote on your favorite features.
- To ask questions and give feedback, please see the Nuxeo answers.

Additional Features

Core Features

The core features of Nuxeo DAM will be explained in this section:

Nuxeo DAM Core

Schema

A specific DAM schema is added: `dam_common`. It is used to store DAM related information like author of the assets, authoring date of the assets.

Document types

Existing Nuxeo DM document types

We reuse the document types defined in a standard Nuxeo, but types are overridden to use our custom schema:

- `Picture`: for image related files (jpeg, png, gif, bmp, ...)
- `File`: for Office document files (Microsoft Office documents, OpenOffice documents, PDF, text files, ...)
- `Video`: to be defined in a generic video module (like imaging one)
- `Audio`: to be defined in a generic audio module

Nuxeo DAM document types

2 new document types are defined for DAM:

- `ImportSetRoot`: the root folder where the `ImportSets` are stored. Extends the `Folder` type.
- `ImportSet`: extends the `Folder` type, contains the assets related to a given import.

One `ImportSet` is created for each import.

Facets

Nuxeo DAM defines a new facet `Asset` to specify which document types are assets.

This facet is added on the `Picture`, `File`, `Video` and `Audio` document types.

Physical document hierarchy layouting

As stated previously, an `ImportSet` document is created for each import, regardless of whether the file being imported is a composite file (eg. a zip file) or a single file (eg. a single jpg image).

They are all stored in the same `ImportSetRoot`: `/default-domain/import-sets` in the repository.

Each `ImportSet` document contains the assets imported.

Hierarchy example:

```
/default-domain/import-sets-root
                  |- folder1
                  |     |- import-set-1
                  |     |     |
                  |     |     |-file1
                  |     |     |-image1
                  |     |     |-subfolder1
                  |     |     |  |-image2
                  |     |     |  |-image3
                  |     |     `-audio_file1
                  |     `- import-set-2
                  |     |
                  |     |-file2
                  |     |-image4
                  |     `-audio_file2
                  `- folder2
                        |- import-set-3
                        |     |
                        |     |-file3
                        |     |-image5
                        |     |-subfolder2
                        |     |  |-image6
                        |     |  |-image7
                        |     `-audio_file3
                        `- import-set-4
                        |
                        |-file4
                        |-image8
                        `-audio_file4
```

The type of folder1 and folder2 is now "ImportFolder".

DAM Bulk Edit

On the same principle as the default bulk edit of the Nuxeo Platform, the DAM module offers a bulk edit form that allows to edit several assets at the same time, with a form adapted to assets relevant metadata.

Customizing the DAM Bulk Edit Form

The default bulk edit form is based on a layout called `damBulkEdit@edit`. To change it you just need to override the default one by your own `damBulkEdit@edit` layout to display your own widgets.

Default DAM bulk edit layout

```
<require>org.nuxeo.dam.layouts</require>
<extension target="org.nuxeo.ecm.platform.forms.layout.WebLayoutManager"
    point="layouts">
  <layout name="damBulkEdit@edit">
    <templates>
      <template mode="any">/layouts/layout_bulkedit_template.xhtml</template>
    </templates>
    <rows>
      <row>
        <widget>dam_edit_tags</widget>
      </row>
      <row>
        <widget>damc_author</widget>
      </row>
      <row>
        <widget>damc_authoringDate</widget>
      </row>
      <row>
        <widget>subjects</widget>
      </row>
      <row>
        <widget>coverage</widget>
      </row>
    </rows>
  </layout>
</extension>
```

⚠ The widgets used in the `damBulkEdit@edit` layout should not have the property `required` set to `true`. If the widgets you want to use have this property, redefine them in the layout without the `required` property.

Video

Nuxeo provides an addon with the following video features:

- In-browser video preview with JPEG thumbnails and QuickTime player
- Storyboard extraction and time based navigation
- Darwin Streaming Server (a.k.a. DSS) integration for seeking forward large videos (using the storyboard for instance) without first buffering all the file locally.

The source code of this addon is located here: https://github.com/nuxeo/nuxeo-platform-video/

To clone it, use the following:

```
git clone git://github.com/nuxeo/nuxeo-platform-video.git
```

Content of this section:

Core module

Video type definition

Schemas

- `video`: store the storyboarding items
- `streamable_media`: store the streamable version of a video file

Document Type

This module defined one document type `Video` which used the `video` and `streamable_media` schemas.

Core event listeners

VideoStoryboardListener

Compute the storyboard for document type that holds the `HasStoryboard` facet. The video storyboard is stored in the `vid:storyboard` field. Also update the `strm:duration` duration field.
VideoPreviewListener

Compute the a 2 thumbnails previews (same sizes as the picture previews) for documents that have the `HasVideoPreview` facet. The results is stored in the `picture` schema using the same picture adapter as the `Picture` documents. If the format is not supported by ffmpeg, a black thumbnail is generated. Also updates the `strm:duration` field.
MediaStreamingUpdaterListener

Asynchronous event listener to build the hinted streamable version of video to be used by the DSS_ integration. The results is stored in the `strm:streamable` field.

FileManagerService contribution

`VideoImporter` is contributed to the `FileManagerService` to create documents of type `Video` if the mimetype is matching when using the drag and drop plugin
or the DAM import button.

Convert module

This package holds the backend converters to compute JPEG thumbnails (preview and storyboard) and streamable version of videos for the DSS integration.

Provides contributions to the `CommandlineExecutorService`:

- get the ffmpeg output info (e.g. the duration in seconds) of a video file
- generating a single screenshot of a video file at a given time offset in seconds with ffmpeg
- generating a sequence of regularly spaces screenshots to compute the storyboard of a video file with ffmpeg
- converting a video from any format to mp4 (container format) H264 (video codec) + aac (audio codec) using handbrake (used for streaming)
- hinting mp4 files to make them suitable for streaming using DSS by using the MP4Box command.
- checking the presence of hinting tracks in a mp4 file using mp4creator to avoid recomputing them when not necessary (optim, not used yet).
- converting a video from any format to ogg (container format) theora (video codec) + vorbis (audio codec) using ffmpeg2theora (not used by default but could be use as a base for Icecast integration in the future as an alternative to DSS for instance).

All those `CommandlineExecutorService` contributions are wrapped into 3 higher level java classes that are contributed to the `ConversionService`:

- `ScreenshotConverter`: extract a single JPEG preview of the video
- `StoryboardConverter`: extract a list of JPEG files with time offset info
- `StreamableMediaConverter`: compute a streamable version of the video suitable for DSS_ integration.

JSF module

Provides basic JSF templates and backing seam components to be able to display a video player (using the quicktime plugin) that either use direct HTTP buffering or the RTSP-based URL that plays well with a darwin streaming server instance if a streamable version of the video is available.

This package also holds sample templates used in Nuxeo DAM to display a storyboard of a video that positions the Quicktime player to the right time offset when clicking on one of the thumbnails.

Dependencies

Mandatory

- ffmpeg is needed to compute JPEG previews, storyboard, and duration extraction: mandatory.

Mandatory if DSS mode enabled

- DSS: the streaming server itself.
- handbrake is used for encoding to h264/aac to compute the version streamable using darwin: only mandatory if the streaming server mode is enabled (disabled by default).
- MP4Box is used for track hinting for mp4 files: only mandatory if the streaming server mode is enabled (disabled by default).

Might be used in the future

- mp4creator will be used to avoid building streamable version of videos that are already streamable in their original version.
- ffmpeg2theora is an optional dependency used by a converter that is not used by default in either Nuxeo DAM or Nuxeo DM.

Audio

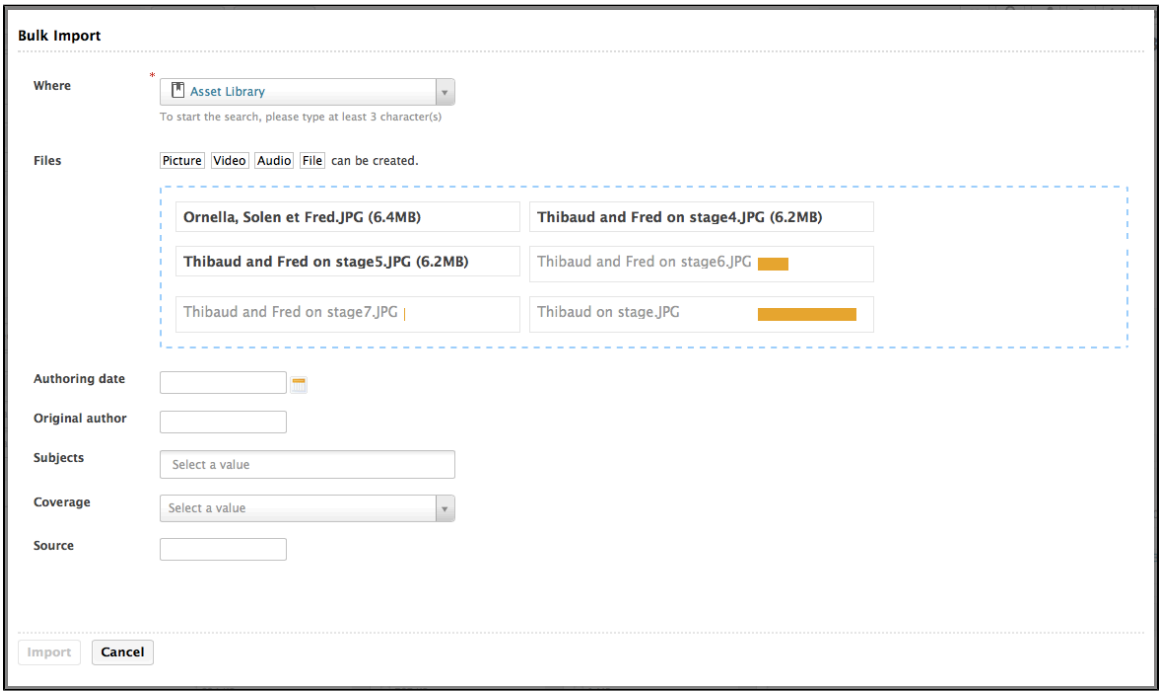
Customizing Nuxeo DAM

This chapter presents you how to configure and customize your Nuxeo DAM.

Table of Contents

Customizing the new Bulk Import UI

The DAM bulk import UI was rewritten in Nuxeo Platform 5.7.1. It is inspired from the bulk import UI that was added in the Document Management module in 5.5, but with some more finishing work, and more configuration possibilities. The bulk import UI in DM will be aligned to this new interface in one of the next Fast Track versions too.



Here is how the new bulk import works:

The import popup lists all the import actions that have been contributed in a drop down list. In 5.7.1, this drop down list is only displayed if there are at least two import actions.
To add an element in the list, you have to contribute an action in the category `DAM_IMPORT_ASSETS`. Actions in this category must define as properties a form layout and an automation chain.

Action contribution

```
<action id="dndDamBulkImportAssets" link=""
      order="10" label="label.smart.import"
      help="desc.smart.import.file">
      <category>DAM_IMPORT_ASSETS</category>
      <properties>
        <property name="chainId">Dam.ImportWithMetaDataInSeam</property>
        <property name="layout">damBulkImport@create</property>
      </properties>
    </action>
```

The selected action displays the form expressed in the contribution and an "Import" button.

The bean attached to the "Import" button creates a Map Object (`DataModelProperties` instance). It pushes the metadata from the form in the Map Object (without schema validation) and starts the automation chain id expressed into the action. The chain takes the blob list as input and puts the Map Object in the context and names it "`docMetaData`".

Operation chain contribution

```
<chain id="Dam.ImportWithMetaDataInSeam">
      <operation id="Dam.Import">
        <param type="boolean" name="overwrite">true</param>
      </operation>
      <operation id="Document.Update">
        <param type="properties" name="properties">expr:Context.get("docMetaData")
        </param>
      </operation>
      <operation id="Seam.AddMessage">
        <param type="string" name="severity">INFO</param>
        <param type="string" name="message">label.dam.assets.imported</param>
      </operation>
      <operation id="Seam.Refresh" />
    </chain>
```

`DataModelProperties` is a map object that stores each value set in the form layout with the field name as key and without validation of the schema manager (as we can have in `DocumentModel`). If you bind a widget to a field `foo:bar` that doesn't exist, the `DataModelProperties` object will store it without generating an exception and you can fetch this field as usual (`dataModelProperties.get("foo:bar")`).

Digital Signature

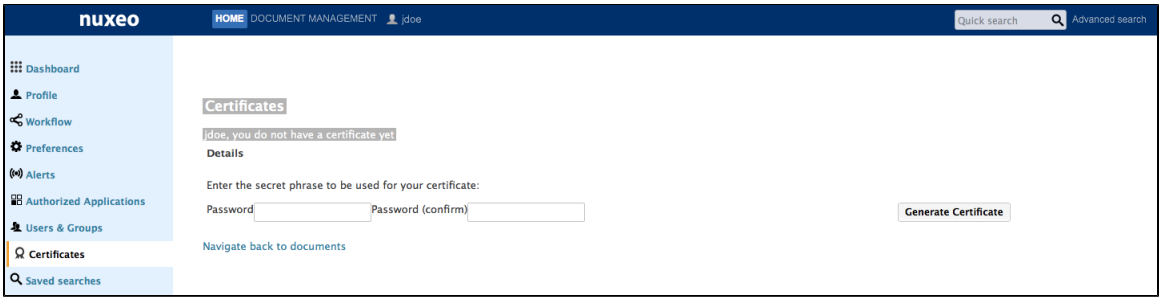
The Digital Signature addon introduces PDF signing capabilities to the Nuxeo Platform. This addon also provides generation of user certificates, which are required for document signing.
Digital Signature implements a layout on signed documents to display the signature information, which can be customized.

Installation

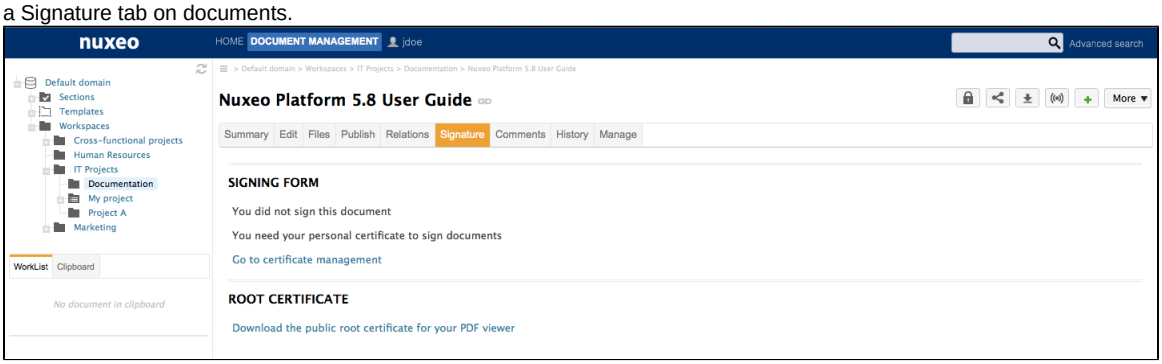
The Digital Signature package requires no specific installation steps. It can be installed like any other package from the Marketplace or from the Admin Center.

After the package is installed, users get some new tabs:

- a Certificates tab in the Home,



- a Signature tab on documents.



Configuration

The Digital Signature package provides a sample root certificate populated with sample company's values. However you will need to configure the package so that documents are signed with your company's certificate and information instead of the sample one.

Setting up the Local Root Certificate

To disambiguate, the term "root certificate" in this section - and in the configuration of this plugin - relates to the Local Certificate Authority (CA) of your company which is the root of all user certificates. This is not to be confused with the global root Certificate Authority, that is one of the top-most entities of the global "chain of trust".

This plugin's root certificate helps establishing a simple method of user certificate verification, as it can be installed in a PDF reader. The best approach, however, is to have your local Certificate Authority's certificate signed by a higher level CA whose ancestor has been signed by one of the actual root Certificate Authorities. This incurs some setup overhead in the initial stages of the project. This method guarantees, though, a more secure approach to document verification, and frees the end users from having to install certificates in their PDF readers. PDF readers capable of handling security are updated automatically with the global root Certificate Authority information.

As the keystore configured in the installable package is a sample keystore containing a test configuration, it is required that it be replaced with the client keystore containing the keypair and the certificate to be used for signing user certificates. As of now the certificate+keypair need to be stored in a .jks formatted keystore and configured via the extension mechanism.

The user certificate generation step requires a Certificate Authority certificate (CA) to be set up inside the Nuxeo Platform system as all user certificates have to be signed by a CA with a recognizable identity — a company rather than a single user. The term local CA can be understood here as "company Certificate Authority" or "system-wide Certificate Authority". Note that there is only one CA certificate per system but each user can have his own certificate.

Setting up a CA certificate from a 3rd party authority

For this exercise you will need the following software:

keytool: the keytool comes with your JDK (Java Development Kit) installation.

openssl: Open SSL

1. Create a keypair (with alias `pdfcakey` in this example).

```
keytool -genkey -keyalg RSA -alias pdfcakey -keypass password -validity 365
-keysize 1024 -dname "cn=PDF-CA, ou=Headquarters, o=Example Organization,
c=US" -keystore pdfca-keystore.jks
```

This creates a keypair (private and public key), and self-signs it automatically.

If you don't wish to use a 3rd party Certificate Authority to sign your key, you can stop here.

2. Create a certificate signing request (CSR).

```
keytool -keystore pdfca-keystore.jks -storepass aaaaaa -alias alternatekey
-keypass password -certreq -file pdfca.csr
```

3. Submit the CSR to a well-known 3rd party Certificate Authority of your choice to sign it.

   You can find examples of 3rd party CAs here and here.
4. When you receive the signed certificate pdfca.crt, import it into your keystore using a new new alias (`pdfcacert` in this example).

```
keytool -import -trustcacerts -alias pdfcacert -file pdfca.crt -keystore
pdfca-keystore.jks
```

Setting up a local CA certificate

An alternative method would be to set up a local signing CA and use it for signing certificates.

Though it could work for small-scale deployments, this approach is not recommended for production purposes.

Step 1: Create a Certificate Authority (CA)

1. Create a CA key.

```
openssl genrsa -out ca.key 2048
```

2. Create a self signed CA certificate.

```
openssl req -new -x509 -days 356 -key ca.key -out ca-self-signed.crt
```

Step 2: Create Subordinate Certificate Authority (SUBCA)

1. Create the key for the subordinate CA.

```
openssl genrsa -out subca.key 2048
```

2. Create a certificate signing request (CSR) for the subordinate CA.

```
openssl req -new -key subca.key -out subca.csr
```

3. Sign the CSR of the subordinate CA.

```
openssl x509 -req -days 730 -in subca.csr -CA ca-self-signed.crt -CAkey ca.key
-set_serial 01 -out subca.crt
```

4. Import a certificate created from your CSR into a JKS keystore.

```
keytool -import -alias certalias -file subca.crt -keystore keystore.jks
```

5. Convert the x509-certificate and the key to pkcs12 format to make it importable into the java keystore.

```
openssl pkcs12 -export -in subca.crt -inkey subca.key -name keyalias -CAfile
ca.crt -caname root -out subca.p12
```

(use "export" as password when prompted)

6. Convert the pkcs12 file to JKS format.

```
keytool -importkeystore -deststorepass storepass -destkeypass keypass
-destkeystore keystore.jks -srckeystore subca.p12 -srcstoretype PKCS12
-srcstorepass export -alias keyalias
```

Now you will need to replace the sample certificate with your own that you just created. You can use the configuration information below which explains how to override the sample certificate with your company certificate.

Step 3: Replace the sample root certificate

1. Create a `***-config.xml` (e.g.`rootcert-digitalsignature-config.xml`) file with the content below:

```
<component name="my.signature.rootservice.config">
  <require>org.nuxeo.signature.config.default</require>
  <extension target="org.nuxeo.ecm.platform.signature.api.pki.RootService"
point="rootconfig">
    <configuration>
      <rootKeystoreFilePath>test-config/keystore.jks</rootKeystoreFilePath>
      <rootKeystorePassword>abc</rootKeystorePassword>
      <rootCertificateAlias>pdfcacert</rootCertificateAlias>
      <rootKeyAlias>pdfcakey</rootKeyAlias>
      <rootKeyPassword>abc</rootKeyPassword>
    </configuration>
  </extension>
</component>
```

2. Put the extension in the `config` directory of your server:
   - `$NUXEO/nxserver/config` for a Tomcat distribution,
   - `$NUXEO/server/default/deploy/nuxeo.ear/config` for a JBoss distribution.

Setting up the Company Information for New Certificates

Another extension provides general company information used in all certificates, like Country, Locale, Organization Name and Organizational Unit.

To add your company's information for users certificates:

1. Create another XML file called `***-config.xml` (e.g.`companyinfo-digitalsignature-config.xml`) with the content below:

```
<?xml version="1.0"?>
<component name="my.signature.userservice.config">
  <require>org.nuxeo.signature.config.default</require>
  <extension target="org.nuxeo.ecm.platform.signature.api.user.CUserService"
point="cuserdescriptor">
    <userDescriptor>
      <countryCode>MX</countryCode>
      <organization>Sigma Alimentos</organization>
      <organizationalUnit>Marketing</organizationalUnit>
    </userDescriptor>
  </extension>
</component>
```

2. Put the extension in the `config` directory of your server:
   - `$NUXEO/nxserver/config` for a Tomcat distribution,
   - `$NUXEO/server/default/deploy/nuxeo.ear/config` for a JBoss distribution.

Use

The layout is defined by:

- Lines and Columns number,
- Starting point cell (Left to right - Top to bottom),
- Signature text size.

By default the configuration sets lines number to 5, columns number to 3, starting cell to 1 and text size to 10px.

```
<extension
          target="org.nuxeo.ecm.platform.signature.api.sign.SignatureService"
          point="signature">
    <configuration>
      <reason>This document signed as an example.
      </reason>
      <layout id="defaultConfig" lines="5" columns="3" startLine="1" startColumn="1"
textSize="10"/>
    </configuration>
  </extension>
```

It means signatures display begins in the first cell, at the top left of the PDF document and so on, on 5 lines and 3 columns maximum.

Here is an example of how to override this default configuration:

```
<require>org.nuxeo.signature.config.default</require>
<extension
          target="org.nuxeo.ecm.platform.signature.api.sign.SignatureService"
          point="signature">
    <configuration>
      <reason>This document signed as an example.
      </reason>
      <layout id="customConfig" lines="5" columns="3" startLine="3" startColumn="3"
textSize="10"/>
    </configuration>
  </extension>
```

Signatures display will begin in the third cell (top right) and so on, on 5 lines and 3 columns maximum.

If signatures number is higher than available cells number, the document will be signed digitally but extra signatures won't be displayed into it.

In this section

- Installation
- Configuration
  - Setting up the Local Root Certificate
    - Setting up a CA certificate from a 3rd party authority
    - Setting up a local CA certificate
  - Setting up the Company Information for New Certificates
- Use

Related documentation

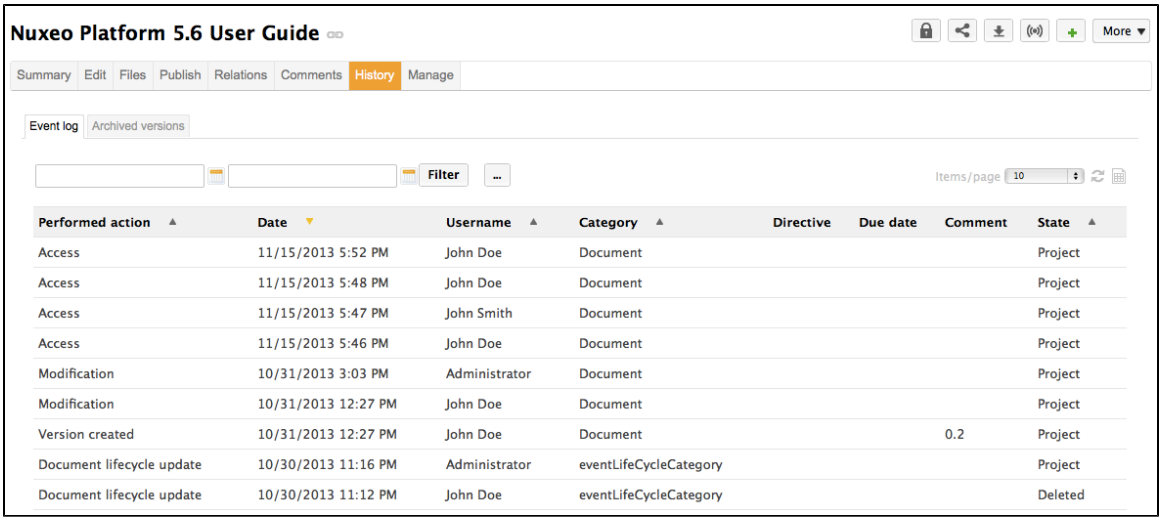
- Digital Signature user documentation

Document Access Tracking

The Document access tracking package is used to register in the document's history the fact that users have accessed the document, and so have probably read it.

The Document access tracking requires no specific installation steps. It can be installed like any other package from the Marketplace or from the Admin Center.

After you installed the package, an "Access" action is logged in the History tab of documents every time a user click on it to consult it.



Related Documentation

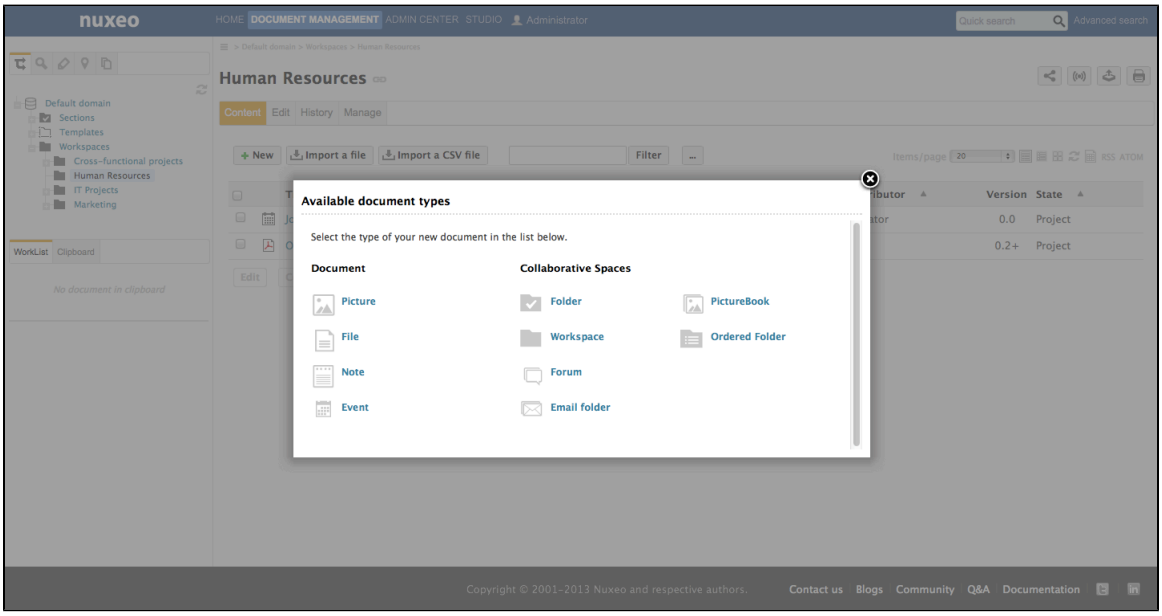
- Document Access Tracking user documentation

Nuxeo Agenda

The Nuxeo Agenda package provides users with a new documents type "Event" that enables them to manage their list of meetings and other events with a calendar view.

The Nuxeo Agenda package requires no specific installation steps. It can be installed like any other package from the Marketplace or from the Admin Center.

After the package is installed, a new document type is available from workspaces, called "Event" and users can add an Agenda gadget on their dashboard.



Related Documentation

- Nuxeo Agenda user documentation

Nuxeo - BIRT Integration

The Nuxeo - BIRT Integration package leverages the reporting features of Eclipse BIRT, enabling users to create reports on the application's activity, directly from the Nuxeo Platform.

Installation

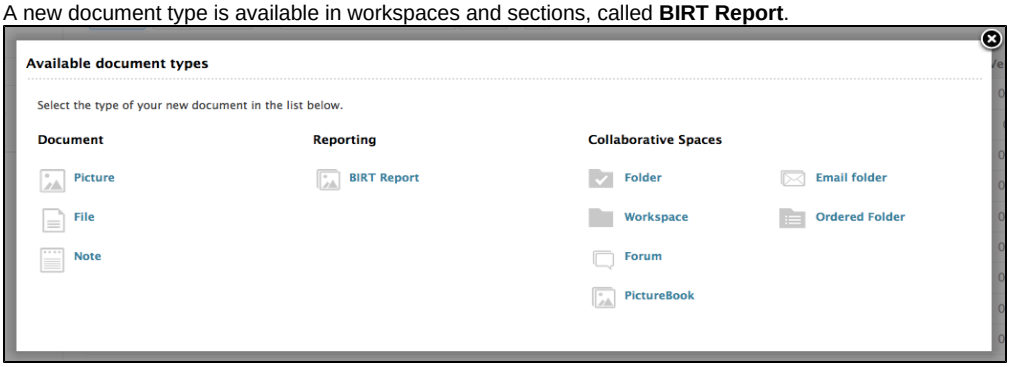
The Nuxeo - BIRT Integration addon requires no specific installation steps. It can be installed like any other package from the Marketplace or from the Admin Center. Note however that it requires the Document Management module.

After you installed the Nuxeo - BIRT Integration package, here are the changes you get in the Nuxeo Platform:

In this section

- the Admin Center has a new vertical tab, called **Reporting**;

- A new document type is available in workspaces and sections, called **BIRT Report**.



Configuration

The Nuxeo - BIRT Integration addon requires that your Nuxeo application runs with PostgreSQL. See the Connecting Nuxeo to the Database page.

Related Documentation

- Nuxeo - BIRT Integration user documentation

Nuxeo Bulk Document Importer

The `nuxeo-importer-core` module is designed to offer support for multi-threaded import on a Nuxeo repository.

Usage

The file importer comes as a Java library (with the Nuxeo Runtime Service) and a sample JAX-RS interface to launch, monitor and abort

import jobs.

Quick Start

To import the folder '/path/to/import' into the workspace '/default-domain/workspaces/some-workspace' while monitoring the import logs from a REST client, use the following HTTP GET queries:

- `GET http://localhost:8080/nuxeo/site/fileImporter/logActivate`
- `GET http://localhost:8080/nuxeo/site/fileImporter/run?targetPath=/default-domain/workspaces/some-workspace&inputPath=/path/to/import&batchSize=10&interactive=false&nbThreads=`
- `GET http://localhost:8080/nuxeo/site/fileImporter/log`

To execute these HTTP queries you can either use a browser with an active Nuxeo session (JSESSIONID cookie) or use a third party stateless HTTP client with HTTP Basic Authentication, eg: with the curl command line client:

```
$ curl --basic -u 'Administrator:Administrator'
"http://localhost:8080/nuxeo/site/fileImporter/log"
```

Don't forget put the URL in quotes if it includes special shell characters such as '&'. You can also use the generic HTTP GUI client from the rest-client Java project: http://code.google.com/p/rest-client/
Be sure to fill in the 'Auth' tab with your user credentials.

Memory

The importer requires a lot of memory. Make sure your maximum heap size is set as high as possible for your environment. Maximum heap size can be set in nuxeo.conf in the JAVA_OPTS variable; for example, argument **-Xmx4g** will set maximum heap size to 4 gigabytes. See Configuration Parameters Index (nuxeo.conf) for more details.

REST API

Resource URL	Description	Output
`GET nuxeo/site/randomImporter/run`	Random text generator for load testing	text/plain; charset=UTF-8
`GET nuxeo/site/fileImporter/run`	Default file importer	text/plain; charset=UTF-8
`GET nuxeo/site/fileImporter/log`	Get current log buffer content	text/plain; charset=UTF-8
`GET nuxeo/site/fileImporter/logActivate`	Activate logging	text/plain; charset=UTF-8
`GET nuxeo/site/fileImporter/logDesactivate`	Deactivate logging	text/plain; charset=UTF-8
`GET nuxeo/site/fileImporter/status`	Get importer thread status	text/plain; charset=UTF-8 "Running" or "Not Running"
`GET nuxeo/site/fileImporter/kill`	Stop the importer thread if running	text/plain; charset=UTF-8

fileImporter/run

Parameter	Default value	Description
`leafType`	null	Leaf type used by the `documentModelFactory` for the import.
`folderishType`	null	Folderish type used by the `documentModelFactory` for the import.
`inputPath`	N/A	Root path to import (local to the server).
`targetPath`	N/A	Target path in Nuxeo
`skipRootContainerCreation`	false	If true the root container won't be created

batchSize	5	Number of documents that will be created before doing a commit
nbThreads	5	Maximum number of importer threads that can be allocated
interactive	false	

N/A: no default value, the parameter is required.

randomImporter/run

Parameter	Default value	Description
targetPath	N/A	Target path in Nuxeo
skipRootContainerCreation		
batchSize		Number of documents that will be created before doing a commit
nbThreads		Maximum number of importer threads that can be allocated
interactive		
nbNodes	N/A	Number of nodes to create
fileSizeKB		
onlyText	true	
blockSyncPostCommitProcessing		
blockAsyncProcessing		
bulkMode	true	

N/A: no default value, the parameter is required.

Extend

You can easily write your own importer, extending the `org.nuxeo.ecm.platform.importer.base.GenericMultiThreadedImporter` class.

Using XML extension points you can also define the different building blocks of the importer:

- class for reading source nodes,
- docType used for leaf Documents,
- docType used for folderish Documents,
- `documentModelFactoryClass`.

See the developer documentation of Nuxeo Bulk Document Importer for details.

See nuxeo-platform-importer Javadoc.

Directory Tree and Threading

The default importer is targeting a simple use case: import a complete filesystem tree inside a Nuxeo repository.

On most computers you have several CPUs and several cores: this means you can import more documents per second by using several threads. However, when importing a tree, threading must be considered carefully:

- Each thread will be associated with a Transaction (remember we import several documents before doing a commit),
- Each transaction is isolated from others (MVCC mode).

This means that a new thread must be created only when a new branch will be accessible inside the source filesystem. At least, the default `ImporterThreadingPolicy` (`DefaultMultiThreadingPolicy`) does that.

As a result, if you import a big folder with a flat structure, you will only have one importer thread, even if you configure to allow more.

To be sure to be able to leverage multi-threading, you can either:

- Ensure the source filesystem is a tree with at least two levels,

- Change the importer threading policy.

Importer and Metadata

The default importer provides two classes to read the source files as well as metadata:

`FileWithMetadataSourceNode`

This is the default implementation, that was mainly targeting at importing a filesystem where file are stored by folders.

The idea is to associate a set of metadata on a per folder basis: the `metadata.properties` will be used for defining the metadata for all files inside the same folder. By default, metadata will be inherited from parent folder, but may be completed or overridden by a local `metadata.properties`.

Here is a structure:

```
TopicA
   file1.pdf
   file2.pdf
   metadata.properties
   TopicA1
      file1.pdf
      file2.pdf
      metadata.properties
   TopicA2
      file1.pdf
      file2.pdf
      metadata.properties
   TopicA3
      file1.pdf
      file2.pdf
      metadata.properties
TopicB
file1.pdf
metadata.properties
TopicB1
   file1.pdf
   file2.pdf
   metadata.properties
TopicB12
file1.pdf
file2.pdf
metadata.properties
```

The `metadata.properties` file is a simple property file in the format `xpath` = `value`. Typically:

```
dc\:description=some description
dc\:source=some source
dc\:subjects=subject4|subject5
dc\:issued=2015-30-04T09:39:43.00Z
```

Please note that:

- Date properties must be formatted using the ISO 8601 standard
- multi-valued property syntax is `dc\:subjects=subject4|subject5`, the default separator being `|`
- complex properties are not supported currently.

`FileWithIndividualMetadasSourceNode`

This second implementation will try to file a property file for each imported file. This allows to have a per file metadata set.

A sample structure would be:

```
branch1
   branch11
      hello11.pdf
      hello11.properties
   hello1.pdf
   hello1.properties
hello.pdf
hello.properties
```

To use this node type you need to redefine the importer. For that create a `importer-config.xml` in `nxserver/config`.

```
<?xml version="1.0"?>
<component name="customImporter">
<require>org.nuxeo.ecm.platform.importer.service.jaxrs.contrib</require>

<extension target="org.nuxeo.ecm.platform.importer.service.DefaultImporterComponent"
point="importerConfiguration">
   <importerConfig sourceNodeClass
="org.nuxeo.ecm.platform.importer.source.FileWithIndividualMetadasSourceNode" >
      <documentModelFactory leafType="File" folderishType="Folder"
documentModelFactoryClass="org.nuxeo.ecm.platform.importer.factories.DefaultDocument
ModelFactory" />
    </importerConfig>
</extension>
</component>
```

Instanciating the importer

It has a configurable framework which has as the main part, the `org.nuxeo.ecm.platform.importer.base.GenericMultiThreaded Importer` class. This 'importer' is responsible, depending on the way it is configured, for performing the import. The configuration of an 'importer' can be established starting with the instantiation of such an 'importer'.

You need to provide a source node of the import, which should contain:

- the entry point of what will be imported,
- a path to where the import should be made on the current repository,
- parameters that will control the maximum number of threads that will be created during the import,
- a logger that will be used during the import (a default one, which is provided by the module, can be used).

In case you need to have an audit support for the import, you can obtain one by providing a 'jobName', which will be used to represent the workflow of the import that will be started in audit. The audit support can be used to avoid later imports (in case the import finished with success).

In this section

Here is an example of how such an importer can be instantiated:

```
TestSourceNode sourceNode = new TestSourceNode(...);
GenericMultiThreadedImporter importer = new GenericMultiThreadedImporter(
    sourceNode, "/", 10, 5, super.getLogger());
```

Configuring the importer

Next, an 'importer' can be configured after instantiation, by providing it 'tools' that are used during the import.

factory

One of these 'tools' is so called the 'factory', and it is used when performing the import of a document. Usually such a 'factory' is supposed to treat both cases, when importing a folderish or a leaf document (an interface is provided for this scope `org.nuxeo.ecm.platform.importer.factories.ImporterDocumentModelFactory`).

filter

Another 'tool' that is used is the 'filter'. More than one 'filter' can be provided to a 'factory' and their scope is to handle the events that are raised during the import. Usually it is better to block all the events that are raised during and after the import of a document (the import of a document can be translated in creating a Nuxeo document model and saving properties on it, which often causes the raise of events), in order to increase the performance of the import.

Thread policy

The last 'tool' that can be provided to an 'importer' is the thread policy that should be used. In case no thread policy is specified, then the default multi thread one is used (this is provided by `org.nuxeo.ecm.platform.importer.threading.DefaultMultiThreadingPolicy` class).

Here is an example of how such tools can be provided to an instantiated importer.

```
TestDocumentModelFactory documentModelFactory = new TestDocumentModelFactory(...);
importer.setFactory(documentModelFactory);
if (useMultiThread) {
    importer.setThreadPolicy(super.getThreadPolicy());
} else {
    importer.setThreadPolicy(new MonoThreadPolicy());
}
ImporterFilter filter = new TestImporterFilter(true,
    true, true, false);
importer.addFilter(filter);
```

Usually such an 'importer' should be instantiated and configured in an instance method of a class that extends the `org.nuxeo.ecm.platform.importer.executor.AbstractImporterExecutor` class. In this instance method, after the importer is instantiated and configured, a call to a superclass method should be made, which will start the import.

```
super.doRun(importer, Boolean.TRUE);
```

The second parameter specifies whether the import should start synchronous or asynchronous.

This class will be the base class for the import and the method that instantiates, configure and start the import, should be called.

Download

To download `nuxeo-importer-core`, check the Nuxeo Marketplace or, if needed, download a more recent version of the JAR (to be installed by hand) from the Nuxeo Maven repository.

Other import tools

You can also have a look at the https://github.com/nuxeo/nuxeo-platform-replication which is a Nuxeo replication tool that uses internally the `nuxeo-importer-core` module. For more details about Nuxeo replication have a look at How to replicate the Nuxeo repository and http://doc.nuxeo.org/5.1/books/nuxeo-book/html/admin-replication.html.

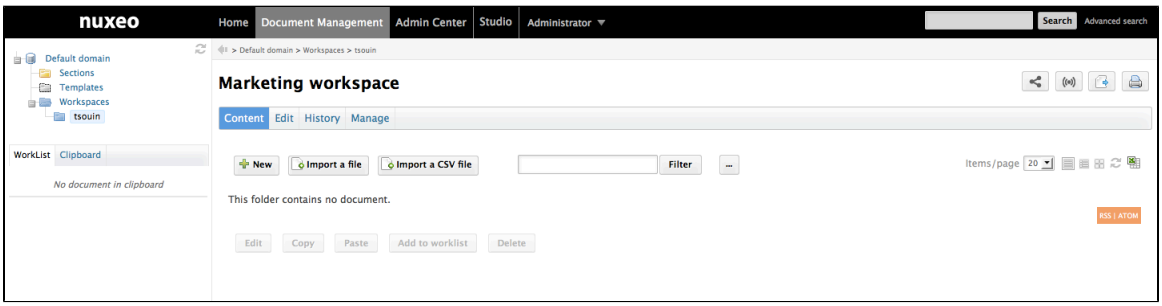
Nuxeo CSV

The Nuxeo CSV addon enables users to proceed to a bulk import of documents in the Nuxeo Platform using a CSV file. This addons enables to create documents with their metadata filled in, to import files with their main attachment, to create a tree structure.

Installation

The Nuxeo CSV package requires no specific installation steps. It can be installed like any other package from the Marketplace or from the Admin Center.

After the package is installed, users have a **Import a CSV file** button available in workspaces, folders and in any document where they can import files.



Configuration: Enabling File Upload

The Nuxeo CSV addon enables users to create file documents and upload their main attachment at the same time. This requires to configure where the server will take the attachments. This is done adding the parameter `nuxeo.csv.blobs.folder` in the server nuxeo.conf and giving him a value that is the path to a folder that can be accessed by the server.

Using Nuxeo CSV

If you installed the Nuxeo CSV addon from the Nuxeo Marketplace, you'll probably want to enable CSV import on the document types you defined, either in Studio or with some code. Here is how to do that.

1. In Nuxeo Online Services, in the **Advanced Settings** menu, click on **XML Extensions**.
2. Click on **New** to create a new extension.
3. Give it an ID and click on **Next**.

   You may want to check the Studio naming conventions.
4. Paste the following content in the text area and fill in the `<type>` tag with the ID of the document type on which you want to enable CSV import.

```
<require>org.nuxeo.ecm.platform.actions</require>
<extension target="org.nuxeo.ecm.platform.actions.ActionService"
  point="filters">
  <filter id="importFile" append="true">
    <rule grant="true">
      <permission>AddChildren</permission>
      <type>YourCustomTypeID</type>
    </rule>
  </filter>
</extension>
```

5. In the XML extension, put as many `<type>` tags as documents types on which CSV import should be enabled.
6. Click on **Save**.
   After you update your Nuxeo Platform instance, your document type(s) will have the **Import a CSV file** button.

If you don't want to use Studio and prefer using your IDE, you can just add a contribution with the XML above.

In this section

- Installation
- Configuration: Enabling File Upload
- Using Nuxeo CSV

Other pages about Nuxeo CSV

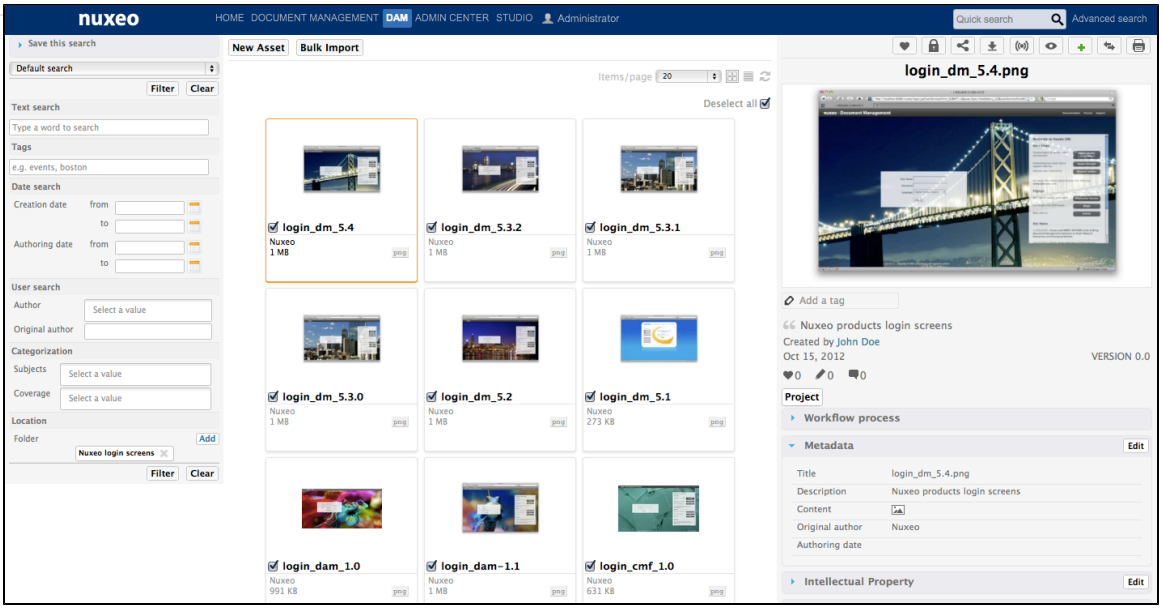
- Nuxeo CSV user documentation

Nuxeo DAM PDF Export

The Nuxeo DAM PDF Export package enables users to export a selection of pictures in a PDF document.

The Nuxeo DAM PDF Export package requires no specific installation steps. It can be installed like any other package from the Marketplace or from the Admin Center.

After the package is installed, a new **Export as PDF** button is displayed at the bottom of the documents lists in the DAM main tab.



Nuxeo Diff

Source code: https://github.com/nuxeo/nuxeo-diff/tree/release-5.8

Bundles:

- http://explorer.nuxeo.org/nuxeo/site/distribution/Nuxeo%20Platform-5.8/viewBundle/org.nuxeo.diff.content
- http://explorer.nuxeo.org/nuxeo/site/distribution/Nuxeo%20Platform-5.8/viewBundle/org.nuxeo.diff.core
- http://explorer.nuxeo.org/nuxeo/site/distribution/Nuxeo%20Platform-5.8/viewBundle/org.nuxeo.diff.jsf

Services:

- http://explorer.nuxeo.org/nuxeo/site/distribution/Nuxeo%20Platform-5.8/viewService/org.nuxeo.ecm.diff.content.adapter.ContentDiffAdapterManager
- http://explorer.nuxeo.org/nuxeo/site/distribution/Nuxeo%20Platform-5.8/viewService/org.nuxeo.ecm.diff.service.DiffDisplayService
- http://explorer.nuxeo.org/nuxeo/site/distribution/Nuxeo%20Platform-5.8/viewService/org.nuxeo.ecm.diff.service.DocumentDiffService

Other Nuxeo Diff documentation
• Nuxeo Diff user documentation

Nuxeo Drive

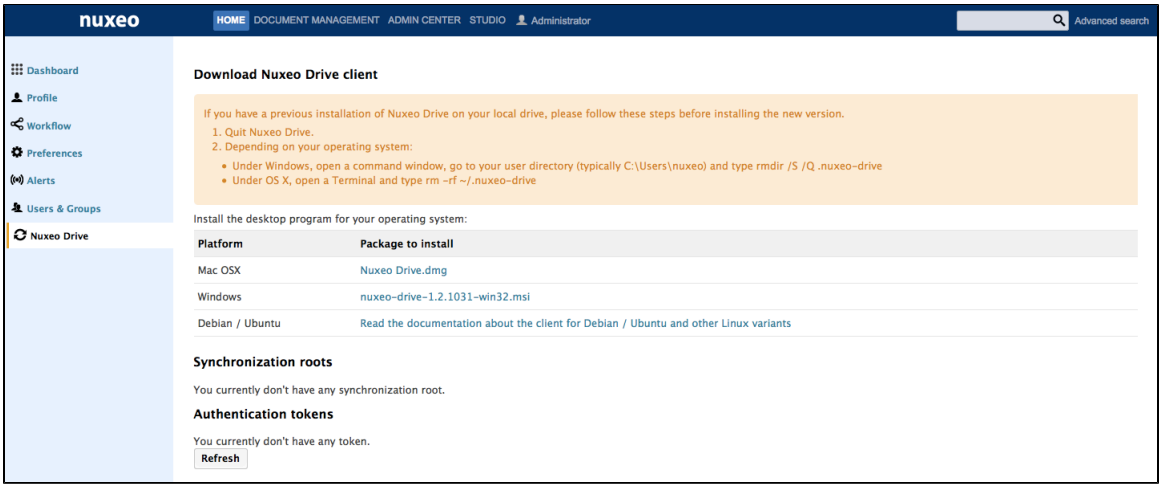
This page is about Nuxeo Drive 2. For versions 1.x see the page Nuxeo Drive 1.x Dev Documentation.

Developer guide now lives in the nuxeo-drive GitHub repository.

Installation

he Nuxeo Drive package requires no specific installation steps. It can be installed like any other package from the Marketplace or from the Admin Center.

After Nuxeo Drive has been installed on the server, users have a Nuxeo Drive tab on their Home.



Configuration

Setting up Nuxeo Drive Beta Channel

To set up the beta channel for Nuxeo Drive update and benefit from the latest versions of Nuxeo Drive, add the following line to your nuxeo.conf file:

```
org.nuxeo.drive.beta.update.site.url=http://community.nuxeo.com/static/drive-tests/
```

Related Documentation

- How to Manually Initialize or Deploy a Nuxeo Drive Instance
- Nuxeo Drive Update Site
- Nuxeo Drive user documentation

How to Customize Nuxeo Drive Versioning Policy

When you edit a document, either from your Nuxeo Drive folder or using the online editing, a new version is automatically created on the Platform and the version number is updated:

- If you are not the last contributor of the document
- Or if your last edit is more than an hour ago

How to Configure the Versioning Delay or Version Increment

You can configure two parameters of Nuxeo Drive versioning policy thanks to the extension point fileSystemItemFactory: the last edit delay until a new version is created and which version increment will occur (minor or major).

For example, to create a major version if the document is modified 30 minutes after the last change, use this contribution:

```
<require>org.nuxeo.drive.adapters</require>

  <extension target="org.nuxeo.drive.service.FileSystemItemAdapterService"
    point="fileSystemItemFactory">

    <fileSystemItemFactory
class="org.nuxeo.drive.service.impl.DefaultFileSystemItemFactory"
name="defaultFileSystemItemFactory" order="40">
      <parameters>
        <parameter name="versioningDelay">1800</parameter>
        <parameter name="versioningOption">MAJOR</parameter>
      </parameters>
    </fileSystemItemFactory>

  </extension>
```

How to Change Nuxeo Drive Versioning Policy

If you need to make more changes on the versioning mechanism in Nuxeo Drive:

1. Write your own implementation of VersioningFileSystemItemFactory interface, and in particular write the expected behavior in the needsVersioning method.
2. Contribute to the `fileSystemItemFactory` extension point to use your new class:

```
<extension target="org.nuxeo.drive.service.FileSystemItemAdapterService"
    point="fileSystemItemFactory">

    <fileSystemItemFactory class="com.sample.drive.CustomFileSystemItemFactory"
name="customFileSystemItemFactory" order="20"/>

  </extension>
```

In this section

Related Documentation

- Nuxeo Drive user documentation
- Nuxeo Drive developer documentation
- How to Manually Initialize or Deploy a Nuxeo Drive Instance

How to Manually Initialize or Deploy a Nuxeo Drive Instance

This page is about Nuxeo Drive 2 and still needs to be completed. For versions 1.x see the page Nuxeo Drive 1.x Admin Documentation.

Usually Nuxeo Drive is initialized when the user completes the Settings panel and successfully logs in. But for auto deployment you might want use the command line or MSI.

Command Line

You can bind a new engine on Nuxeo Drive by calling the `ndrive` executable with the following arguments. It is recommended to have your Drive not running while executing this command.

```
ndrive.exe bind-server [--password PASSWORD] [--local-folder LOCALFOLDER] USERNAME
URL
```

More information about the parameters:

- `USERNAME`: The username of the user who will be using Nuxeo Drive. Mandatory.
- `URL`: The URL of the Nuxeo server. Mandatory.
- `PASSWORD`: The password of the user who will be using Nuxeo Drive. If you don't specify the `PASSWORD` then it will be asked to the user when Nuxeo Drive is started.
- `LOCALFOLDER`: The path to the Nuxeo Drive folder that will be created. Path must include the Nuxeo Drive folder. If `LOCALFOLDER` is not specified then the default location will be picked.

MSI

On Windows you can automatically call the `bind-server` command on install if you set up the MSI variables:

- `TARGETURL`. The URL of the Nuxeo server. Mandatory.
- `TARGETUSERNAME`: The username of the user who will be using Nuxeo Drive. Mandatory.
- `TARGETPASSWORD`: The password of the user who will be using Nuxeo Drive. If you don't specify it then it will be asked to the user when Nuxeo Drive is started.
- `TARGETDRIVEFOLDER`: The path to the Nuxeo Drive folder that will be created. Path must include the Nuxeo Drive folder. Mandatory.

To set up the MSI variables:

1. In a command prompt, type the following command and replace the variable values with your own values.

   ```
   msiexec /i nuxeo-drive.msi TARGETURL=http://localhost:8080/nuxeo
   TARGETUSERNAME=username TARGETPASSWORD=password
   TARGETDRIVEFOLDER=\Path\to\Nuxeo\Drive\Folder
   ```

   Note: If the path to the Nuxeo Drive folder holds spaces, use quotes at the beginning and the end of the path. For instance: `"C:\Users\John Doe\Documents\Nuxeo Drive"`.
   The Nuxeo Drive installer opens.
2. Follow the installer steps.
3. At the end of the installer, start Nuxeo Drive.
   The **Settings** window is displayed, with account settings prefilled with the variables values you filled in. If you haven't specified the password, you now need to fill it in to use Nuxeo Drive.

Other documentation about Nuxeo Drive

- Nuxeo Drive Update Site
- Nuxeo Drive developer documentation
- Nuxeo Drive user documentation

Nuxeo Drive 1.x Admin Documentation

Usually Nuxeo Drive is automatically initialized at first startup. This includes:

- The creation of the configuration folder: `<user_home>/.nuxeo-drive`
- The creation of the local folder: `<user_home>/.Nuxeo Drive`
- The initialization of the SQLite database: `<user_home>/.nuxeo-drive/nxdrive.db`

However, you might want to do this initialization manually, for example to preset the Nuxeo server URL and proxy configuration before launching Nuxeo Drive for the first time. This can be useful for the deployment of Nuxeo Drive on a large set of desktops, allowing end users to work on a preconfigured instance, only needing to provide their credentials at first startup.

Note that this process can be scripted for an automated deployment.

Prerequisite

If an existing instance of Nuxeo Drive is installed first make sure you reset it before running manual initialization:

1. Quit Nuxeo Drive.
2. Remove the `.nuxeo-drive` directory:

OS X / Linux

```
rm -rf ~/.nuxeo-drive
```

Windows

```
rmdir /S /Q "C:\users\<username>\.nuxeo-drive"
```

3. Remove the `Nuxeo Drive` directory:

OS X / Linux

```
rm -rf ~/Nuxeo\ Drive/
```

Windows

```
rmdir /S /Q "C:\users\<username>\documents\Nuxeo Drive"
```

Configuration Folder

OS X / Linux

```
mkdir ~/.nuxeo-drive
touch ~/.nuxeo-drive/nxdrive.db
```

Windows

```
mkdir "C:\users\<username>\.nuxeo-drive"
```

Device Id Generation

The `device_config` table of the SQLite database needs a unique id as a primary key of its single row (`device_id` column). You first need to generate this id, for example with Python:

```
$ python
    Python 2.7.3 (default, Sep 26 2013, 20:03:06)
    [GCC 4.6.3] on linux2
    Type "help", "copyright", "credits" or "license" for more information.
    >>> import uuid
    >>> uuid.uuid1().hex
    '1bd6686882c111e391a6c8f733c9742b'
    >>> exit()
```

SQLite Database Initialization

1. Connect to the empty SQLite database.

   OS X / Linux

   ```
   sqlite3 ~/.nuxeo-drive/nxdrive.db
   ```

   Windows

   a. Install and open SqliteBrowser.
   b. Click on **New Database** and save it under `C:\users\<username>\.nuxeo-drive\nxdrive.db`.
   c. Click Cancel in the **Edit table definition** window.

2. Create the `device_config` and `server_bindings` tables.

   OS X / Linux

   Execute the following queries in the Terminal.

```
CREATE TABLE device_config (
            device_id VARCHAR NOT NULL,
            client_version VARCHAR,
            proxy_config VARCHAR,
            proxy_type VARCHAR,
            proxy_server VARCHAR,
            proxy_port VARCHAR,
            proxy_authenticated BOOLEAN,
            proxy_username VARCHAR,
            proxy_password BLOB,
            proxy_exceptions VARCHAR,
            auto_update BOOLEAN,
            PRIMARY KEY (device_id),
            CHECK (proxy_authenticated IN (0, 1))
        );
        CREATE TABLE server_bindings (
            local_folder VARCHAR NOT NULL,
            server_url VARCHAR,
            remote_user VARCHAR,
            remote_password VARCHAR,
            remote_token VARCHAR,
            server_version VARCHAR,
            update_url VARCHAR,
            last_sync_date INTEGER,
            last_event_log_id INTEGER,
            last_filter_date INTEGER,
            last_ended_sync_date INTEGER,
            last_root_definitions VARCHAR,
            PRIMARY KEY (local_folder)
        );
```

Windows

a. Click on the **Execute SQL** tab.
b. Copy paste the previous queries and click on **Execute SQL** (blue arrow).
c. Click on **Database Structure** and check the `device_config` and `server_bindings` tables have been created.

3. Insert the single row in `device_config`.

Use the previously generated id for the `device_id` column, and set your proxy settings as in the examples below.

Manual proxy configuration

```
INSERT INTO device_config (device_id, proxy_config, proxy_type,
proxy_server, proxy_port, proxy_authenticated, auto_update) VALUES
('1bd6686882c111e391a6c8f733c9742b', 'Manual', 'http', '10.218.9.82',
'80', 0, 0);
```

System proxy configuration

```
INSERT INTO device_config (device_id, proxy_config, proxy_authenticated,
auto_update) VALUES
('1bd6686882c111e391a6c8f733c9742b', 'System', 0, 0);
```

No proxy configuration

```
INSERT INTO device_config (device_id, proxy_config, proxy_authenticated,
auto_update) VALUES
('1bd6686882c111e391a6c8f733c9742b', 'None', 0, 0);
```

4. Insert a row in `server_bindings` .
   Use your local folder path, the Nuxeo server URL, the Nuxeo server version and the Nuxeo Drive update site URL as in the example below.

   OS X / Linux

   ```
   INSERT INTO server_bindings (local_folder, server_url, server_version,
   update_url) VALUES ('/home/<username>/Nuxeo Drive',
   '<protocol>://<host>:<port>/nuxeo/', '6.0',
   'http://community.nuxeo.com/static/drive/');
   ```

   Windows

   ```
   INSERT INTO server_bindings (local_folder, server_url, server_version,
   update_url) VALUES ('C:\users\<username>\documents\Nuxeo Drive',
   '<protocol>://<host>:<port>/nuxeo/', '6.0',
   'http://community.nuxeo.com/static/drive/');
   ```

5. Quit SQLite.

   OS X / Linux

   ```
   .exit
   ```

   Windows

   Click on **Write Changes** and close SqliteBrowser.

Start Nuxeo Drive

The Settings popup should appear waiting for the user's credentials only.

In this section

Other documentation about Nuxeo Drive

Nuxeo Drive 1.x Dev Documentation

Nuxeo Drive is a desktop client that enables bidirectional synchronization between the local desktop and a Nuxeo content repository.This page aims to describe the main technical concepts used in Nuxeo Drive, on both the client and server side.

Please make sure you have read the Nuxeo Drive User Documentation before reading this page. If you are interested in building and developing on Nuxeo Drive, please take a look at the contributor guide.

> Note that this description of the technical details and internals of Nuxeo Drive should not be viewed as a commitment that they are a stable and definitive API. The details of the APIs, Operations, and the local storage in particular, are subject to change in order to improve the user experience.

Client

The Nuxeo Drive client is a full Python application. It is distributed as a complete package embedding all required software and libraries to run under Windows 32 and 64 bits (.msi) and Mac OS (.dmg). This includes Python itself and a set of Python third party libraries listed in the requirements.txt file.

You can fetch the latest release of the Nuxeo Drive client for Windows and Mac OS (also available from your Nuxeo Platform **Home** > **Nuxeo Drive** tab).

Local Storage

Drive local storage relies on SQLAlchemy, the Python SQL toolkit and Object Relational Mapper. The database file `nxdrive.db` is located in the `.nuxeo-drive` folder located in the user home directory.

The data model is described below.

- `device_config`: Single row table holding the local device configuration.
  - `device_id` (primary key): Universal unique identifier of the local device.
  - `client_version`: Nuxeo Drive version, read from `nxdrive/__init__.py` and accessible through the `ndrive -v` command. It is displayed in the About tab of the Settings windows.
  - `proxy_config`: Proxy configuration, possible values are: 'System', 'None', 'Manual'.
  - `proxy_type`: Proxy protocol, possible values are: 'http', 'https'.
  - `proxy_server`: Proxy server URL.
  - `proxy_port`: Proxy port.
  - `proxy_authenticated`: Flag to indicate if the proxy server requires authentication.
  - `proxy_username`: User name for proxy authentication if required.
  - `proxy_password`: Password for proxy authentication if required.
  - `proxy_exceptions`: Comma-separated list of proxy exceptions.
  - `auto_update`: Flag to indicate if Nuxeo Drive should update automatically when a new version is available on the update site.
- `server_bindings`: Table holding the server bindings, i.e. the configuration of the connections between Nuxeo Drive and the synchronized Nuxeo servers (can be several).
  - `local_folder` (primary key): Absolute path of the desktop folder where the synchronized content is stored. By default it is called `Nuxeo Drive` and is located in the user home directory on Mac OS and Linux, in `My Documents` on Windows.
  - `server_url`: URL of the Nuxeo server, matching the following pattern: `(http|https)://<host>[:<port>]/nuxeo`.
  - `remote_user`: User name for authentication against the Nuxeo server.
  - `remote_password`: Password for authentication against the Nuxeo server. Optional. See the Username / password based authentication section.
  - `remote_token`: Token for authentication against the Nuxeo server. Optional. See the Token based authentication section.
  - `server_version`: Version of the Nuxeo server, returned by the `NuxeoDrive.GetClientUpdateInfo` operation.
  - `update_url`: URL of the update site, returned by the `NuxeoDrive.GetClientUpdateInfo` operation. Default value is http://community.nuxeo.com/static/drive/.
  - `last_sync_date`: Date of the last remote polling as a timestamp in milliseconds.
  - `last_event_log_id`: Id of the last audit event log entry, used as the upper bound of the range clause in the change summary query.
  - `last_filter_date`: Last time a local filter was applied through the Folders tab as a timestamp in milliseconds.
  - `last_ended_sync_date`: Last time a synchronization iteration was over as a timestamp in milliseconds. Displayed in the "Last synchronized" entry of the systray menu.
  - `last_root_definitions`: Ids of the currently synchronized containers (usually called "synchronization roots").
- `last_known_states`: Table holding the state of the synchronized files and folders as a set of pair states, each pair representing the local file or folder on one side and the remote document on the other side.
  - `id` (primary key): Auto increment id.

- `local_folder`: Foreign key to `server_bindings.local_folder`.
- `last_local_updated`: Timestamp of the last local state update.
- `last_remote_updated`: Timestamp of the last remote state update.
- `local_digest`: Digest of the local file (empty for folders).
- `remote_digest`: Digest of the remote file (empty for folders).
- `local_path`: Path of the local file or folder, relative to the `local_folder`.
- `remote_ref`: Reference of the remote document, see XXX for more information.
- `local_parent_path`: Parent path of the local file or folder, relative to the `local_folder`.
- `remote_parent_ref`: Parent reference of the remote document.
- `remote_parent_path`: Parent path of the remote document, it is a concatenation of the ancestor references.
- `local_name`: Name of the local file or folder.
- `remote_name`: Name of the remote document. By default it is the value of `dc:title` for a `Folderish` document and the filename for a `BlobHolder`.
- `folderish`: Integer representation of the `Folderish` facet of the remote document.
- `local_state`: State of the local file or folder. See Pair states for the list of possible states.
- `remote_state`: State of the remote document. See Pair states for the list of possible states.
- `pair_state`: Pair state. See Pair states for the list of possible pair states.
- `remote_can_rename`: Flag to indicate if the remote document can be renamed.
- `remote_can_delete`: Flag to indicate if the remote document can be deleted.
- `remote_can_update`: Flag to indicate if the remote document can be updated.
- `remote_can_create_child`: Flag to indicate if the remote document accepts child creation.
- `last_sync_date`: Last synchronization date of the given pair as a `datetime`.
- `error_count`: Number of times the pair synchronization was in error and the pair blacklisted.
- `last_sync_error_date`: Last synchronization error date. This is used to blacklist documents in error from synchronization for a certain time (5 minutes by default).

Nuxeo Drive embeds the Alembic tool to handle migration of the data model in case it changes from one version to another, for more details see related documentation.

Pair States

Each record of `last_known_states` (called *pair*) holds the state of the local file or folder in `local_state` and the state of the remote document in `remote_state`.

The possible values for `local_state` and `remote_state` are:

- `unknown`
- `synchronized`
- `created`
- `modified`
- `deleted`

The possible values for `pair_state` resulting of the different combinations of `local_state` and `remote_state` are summed up in this table.

local_state	remote_state	pair_state
Regular cases		
unknown	unknown	unknown
synchronized	synchronized	synchronized
created	unknown	locally_created
unknown	created	remotely_created
modified	synchronized	locally_modified
synchronized	modified	remotely_modified
modified	unknown	locally_modified
unknown	modified	remotely_modified
deleted	synchronized	locally_deleted
synchronized	deleted	remotely_deleted
deleted	deleted	deleted
synchronized	unknown	synchronized

Conflicts with automatic resolution		
created	deleted	locally_created
deleted	created	remotely_created
modified	deleted	remotely_deleted
deleted	modified	remotely_created
Conflicts with non trivial resolution		
modified	modified	conflicted
created	created	conflicted
created	modified	conflicted
Inconsistent cases		
unknown	deleted	unknown_deleted
deleted	unknown	deleted_unknown

This is the key of the Synchronization process as it relies on the `pair_state` to decide which action is needed in order to synchronize a local file or folder with the associated remote document and vice versa.

Authentication

For each server binding (connection between Nuxeo Drive and a Nuxeo server) Nuxeo Drive needs to provide some credentials to get authenticated against the Nuxeo repository. There are two modes of authentication.

Token Based (Recommanded)

Is only possible if the `nuxeo-platform-login-token` addon is deployed on the Nuxeo server. When asking for a server binding, either through the authentication pop-up or the command line, the user needs to provide its username and password for Nuxeo Drive to make a request to the `TokenAuthenticationServlet` in order to acquire a token. This token is a `UUID` stored in a server-side SQL directory and allows a user to get authenticated through the `TokenAuthenticator` by passing the token in the `X-Authentication-Token` request header. This authentication plugin is configured to be used with the `Trusting_LM` `LoginModule` which implies that no password check is done, a `Principal` is created from the username associated with the token if the user exists in the user directory. The token is stored by Nuxeo Drive in `server_bindings.remote_token` and reused for each HTTP request. This has the benefit of not having to store the user's password on the local desktop.

At any time a user can revoke a token for a given device from the **Nuxeo Drive** tab in the `User Center`. In this case Nuxeo Drive isn't able to authenticate against the Nuxeo server, switches to offline mode and deletes the token from the local storage. The "Update credentials" entry from the Nuxeo Drive menu is then displayed as "required", allowing the user to acquire a new token by providing valid credentials.

Note that this addon is not specific to Nuxeo Drive and can be used by any third party application to authenticate to a Nuxeo repository. Technically, to acquire such a token, you need to send an HTTP `GET` request to the server using:

- A Basic authentication header built from valid credentials.
- The `/authentication/token` URL pattern.
- The following required parameters: `applicationName, deviceId, permission` (optionally `deviceDescription`). The parameters are URI decoded by the Servlet.

So a sample call would be:

```
curl -H 'Authorization:Basic **********************' -G
'http://<host>:<port>/nuxeo/authentication/token?applicationName=Nuxeo%20Driv
e&deviceId=Ubuntu64bits&permission=rw&deviceDescription=My%20Linux%20box'
```

The token is sent back as a string in the plain text response body.

Username / Password Based

Is used as a fallback if the `nuxeo-platform-login-token` add-on is not deployed on the Nuxeo server. In this case, at server binding time, both username and password are stored in the local database (`server_bindings.remote_user` and `server_bindings.remote_password`) and used for each HTTP request through a `Basic Authentication` header.

Synchronization Process

Synchronization is handled by a main thread that starts an infinite loop; see `Synchronizer.loop()` . The loop processes the sequence described below, putting Nuxeo Drive to sleep between each sequence execution for a maximum delay of 5 seconds if there is no more content to synchronize. This delay is configurable with the `--delay` option when using the `ndrive` command.

1. **Remote polling**: Get a summary of changes in the synchronization roots for the current user by calling the `NuxeoDrive.GetChangeSummary` operation. See `Synchronizer._get_remote_changes()` .
   - In the case of the first pass or if there are too many changes (> 1000, configurable with the `org.nuxeo.drive.document.change.limit` property), make a full scan of the synchronization roots by calling the `NuxeoDrive.GetChildren` operation recursively and update the state of each document in `last_known_states`.
   - In the other cases (i.e. most of the time), only update the state of the recently updated documents in `last_known_states`.
2. **Local scan**: Process a recursive full scan of the `Nuxeo Drive` folder and update the state of each file or folder in `last_known_states`. See `Synchronizer.scan_local()` .
3. **Synchronization**: Process synchronization for each pair that needs it. See `Synchronizer.synchronize()` .
   a. Get the list of *pending* pairs, i.e. query `last_known_states` with `pair_state != 'synchronized'`. See `Controller.list_pending()` .
   b. For each *pending* pair synchronize it, see `Synchronizer.synchronize_one()` . Basically the name of the synchronization handler is dynamically computed using `'_synchronize_' + last_known_states.pair_state`, then the handler is called. Here is the list of available synchronization handlers:
      - `_synchronize_locally_created`: Calls the `NuxeoDrive.CreateFolder` or `NuxeoDrive.CreateFile` operation.
      - `_synchronize_remotely_created`: Creates a local folder or download a file to an existing local folder.
      - `_synchronize_locally_modified`: Calls the `NuxeoDrive.UpdateFile` operation.
      - `_synchronize_remotely_modified`: Renames a local folder or renames / updates a local file.
      - `_synchronize_locally_deleted`: Calls the `NuxeoDrive.Delete` operation.
      - `_synchronize_remotely_deleted`: Deletes a local file or folder.
      - `_synchronize_deleted`: Deletes a pair from `last_known_states` (deletion on both sides).
      - `_synchronize_conflicted`: Rename the local file with a suffix including the username and modification time (the suffix is returned by the `NuxeoDrive.GenerateConflictedItemName` operation) and fetches the remote file by calling `_synchronize_remotely_created`. As a result, two files exist on both sides and the users need to manually resolve the conflict, for instance by deleting the suffixed file or by renaming it and deleting the original one.

Automation Operations

The communication between Nuxeo Drive and a Nuxeo server fully relies on Content Automation Concepts, except for:

- Acquiring the token: `GET` request to the `TokenAuthenticationServlet`;
- Downloading blobs: `GET` request to the `DownloadServlet`.

Here is the list of Automation operations used by Nuxeo Drive:

NuxeoDrive.GetChangeSummary

Gets a summary of document changes in the synchronization roots of the currently authenticated user since the last synchronization date. The change summary, of type `FileSystemChangeSummary` mainly holds the new synchronization date and a list of `FileSystemItemChange` objects.

NuxeoDrive.GetTopLevelFolder

Gets the top level `FolderItem` for the currently authenticated user.

NuxeoDrive.GetChildren

Gets the children of the `FolderItem` with the given id for the currently authenticated user.

NuxeoDrive.GetFileSystemItem

Gets the `FileSystemItem` with the given id for the currently authenticated user.

NuxeoDrive.FileSystemItemExists

Check if the `FileSystemItem` with the given id exists for the currently authenticated user.

NuxeoDrive.CreateFile

Creates a file with the given blob in the `FileSystemItem` with the given id for the currently authenticated user.

NuxeoDrive.CreateFolder

Creates a folder with the given name in the `FileSystemItem` with the given id for the currently authenticated user.

NuxeoDrive.Rename

Renames the `FileSystemItem` with the given id with the given name for the currently authenticated user.

NuxeoDrive.UpdateFile

Updates the `FileSystemItem` with the given id with the given blob for the currently authenticated user.

NuxeoDrive.Delete

Deletes the `FileSystemItem` with the given id for the currently authenticated user.

NuxeoDrive.CanMove

Checks if the `FileSystemItem` with the given source id can be moved to the `FileSystemItem` with the given destination id for the currently authenticated user.

NuxeoDrive.Move

Moves the `FileSystemItem` with the given source id to the `FileSystemItem` with the given destination id for the currently authenticated user.

NuxeoDrive.GenerateConflictedItemName

Generates a conflicted name for a `FileSystemItem` given its name, the currently authenticated user's first name and last name. Doing so as an operation makes it possible to override this part without having to fork the client codebase.

Data Exchange Format

As Nuxeo Drive relies on Automation REST calls, all the exchanged data is transfered in `JSON`. Here are some samples of the most common HTTP `POST` requests sent by Nuxeo Drive and their associated response. These are extracts from the Nuxeo Drive log file set to `TRACE` level.

Remote Polling Through `NuxeoDrive.GetChangeSummary`

Request

```
nxdrive.client.base_automation_client Calling
http://localhost:8080/nuxeo/site/automation/NuxeoDrive.GetChangeSummary with
headers {
    'X-Authentication-Token': u'b5b6b0ce-80b5-4398-aeb5-6a1dab01157d',
    'X-NXDocumentProperties': '*',
    'X-Device-Id': u'06208ada004411e388adc8f733c9742b',
    'Accept': 'application/json+nxentity, */*',
    'X-User-Id': u'joe',
    'Cache-Control': 'no-cache',
    'Content-Type': 'application/json+nxrequest',
    'X-Application-Name': 'Nuxeo Drive'
},
cookies [],
JSON payload {
    "params": {
        "lastSyncDate": 1375977907000,
        "lastSyncActiveRootDefinitions":
"default:ccbd9a3c-85d3-4589-ab82-8c28a22b50db"
    }
}
```

Response

```
nxdrive.client.base_automation_client Response for
http://localhost:8080/nuxeo/site/automation/NuxeoDrive.GetChangeSummary with JSON
payload {
    "fileSystemChanges":[{
        "repositoryId":"default",
        "eventId":"documentModified",
        "eventDate":1375977907092,
        "fileSystemItem":{
            "digestAlgorithm":"md5",
            "canUpdate":true,

"downloadURL":"nxbigfile/default/009e8ad7-feb6-4b64-9169-9947774780f9/blobholder:0/t
est.txt",
            "digest":"d41d8cd98f00b204e9800998ecf8427e",
            "creationDate":1375977901723,

"parentId":"defaultSyncRootFolderItemFactory#default#ccbd9a3c-85d3-4589-ab82-8c28a22
b50db",
            "folder":false,
            "canDelete":true,
            "lastModificationDate":1375977907086,
            "creator":"joe",
            "canRename":true,
            "name":"test.txt",

"id":"defaultFileSystemItemFactory#default#009e8ad7-feb6-4b64-9169-9947774780f9",

"path":"/org.nuxeo.drive.service.impl.DefaultTopLevelFolderItemFactory#/defaultSyncR
ootFolderItemFactory#default#ccbd9a3c-85d3-4589-ab82-8c28a22b50db/defaultFileSystemI
temFactory#default#009e8ad7-feb6-4b64-9169-9947774780f9",
            "userName":"joe"
        },
        "docUuid":"009e8ad7-feb6-4b64-9169-9947774780f9",

"fileSystemItemId":"defaultFileSystemItemFactory#default#009e8ad7-feb6-4b64-9169-994
7774780f9",
        "fileSystemItemName":"test.txt"
    }],
    "hasTooManyChanges":false,
    "syncDate":1375977912000,

"activeSynchronizationRootDefinitions":"default:ccbd9a3c-85d3-4589-ab82-8c28a22b50db
"
}
```

We can notice in particular that:

- The request contains the authentication token as a header and the last synchronization date as a timestamp in the `JSON` data.
- The response contains:
  - The synchronization date: `"syncdate"` that will be used to update `server_bindings.last_sync_date.`
  - The list of file system item changes with one element only: the `FileSystemItem` representation resulting of the adaptation of the `test.txt` document that has been remotely modified, including all the data needed to update the remote part of the pair corresponding to the retrieved `id` in `last_known_states`, such as the `name`, `digest`, `downloadURL` and `lastModi ficationDate`. In this case, the content of the document has been modified, so the `digest` is different from the current `la`

`st_known_states.remote_digest`, so `last_known_states.pair_state` will be updated to `remotely_modified` and the `_synchronize_remotely_modified` handler will take care of updating the local file by downloading the remote one calling the `downloadURL`.

Synchronization of a Locally Created File

File Upload Using the Automation Batch Upload

Request

```
nxdrive.client.base_automation_client Calling
http://localhost:8080/nuxeo/site/automation/batch/upload with
headers {
    'Content-Length': 356587,
    'X-Authentication-Token': u'b5b6b0ce-80b5-4398-aeb5-6a1dab01157d',
    'X-Device-Id': u'06208ada004411e388adc8f733c9742b',
    'X-File-Type': 'application/vnd.oasis.opendocument.text',
    'X-User-Id': u'joe',
    'X-File-Size': 356587,
    'Cache-Control': 'no-cache',
    'X-Batch-Id': '1376003340.46_956489996',
    'X-File-Idx': 0,
    'Content-Type': 'application/octet-stream',
    'X-Application-Name': 'Nuxeo Drive',
    'X-File-Name': 'NUXEO_WCM_MODULE_User%20stories.odt'
},
cookies []
for file /home/ataillefer/Nuxeo Drive/Test folder/NUXEO_WCM_MODULE_User stories.odt
```

Response

```
nxdrive.client.base_automation_client Response for
http://localhost:8080/nuxeo/site/automation/batch/upload with JSON payload {
    "uploaded":"true",
    "batchId":"1376003340.46_956489996"
}
```

We can notice in particular that:

- The request contains all required headers for the batch upload such as `X-Batch-Id, X-File-Type, X-File-Size, X-File-Name`. Of course the binary content itself is part of the request data.
- The response contains the `uploaded` marker to indicate a successful upload.

Document Creation Using the `NuxeoDrive.CreateFile` Operation

Request

```
nxdrive.client.base_automation_client Calling
http://localhost:8080/nuxeo/site/automation/batch/execute with
headers {
    'X-Authentication-Token': u'b5b6b0ce-80b5-4398-aeb5-6a1dab01157d',
    'X-NXDocumentProperties': '*',
    'X-Device-Id': u'06208ada004411e388adc8f733c9742b',
    'Accept': 'application/json+nxentity, */*',
    'X-User-Id': u'joe',
    'Cache-Control': 'no-cache',
    'Content-Type': 'application/json+nxrequest',
    'X-Application-Name': 'Nuxeo Drive'
},
cookies [],
JSON payload {
    "params": {
        "batchId": "1376003340.46_956489996",
        "operationId": "NuxeoDrive.CreateFile",
        "fileIdx": "0",
        "parentId":
"defaultSyncRootFolderItemFactory#default#ccbd9a3c-85d3-4589-ab82-8c28a22b50db"
    }
}
```

Response

```
nxdrive.client.base_automation_client Response for
http://localhost:8080/nuxeo/site/automation/batch/execute with JSON payload {
    "digestAlgorithm":"md5",
    "canUpdate":true,

"downloadURL":"nxbigfile/default/70235c0e-ad97-421e-92f6-4d71afaed9c1/blobholder:0/N
UXEO_WCM_MODULE_User%20stories.odt",
    "digest":"3a9fdd2e619a2a47678bfd0b7f3d97ac",
    "creationDate":1376003340567,

"parentId":"defaultSyncRootFolderItemFactory#default#ccbd9a3c-85d3-4589-ab82-8c28a22
b50db",
    "folder":false,
    "canDelete":true,
    "lastModificationDate":1376003340567,
    "creator":"joe",
    "canRename":true,
    "name":"NUXEO_WCM_MODULE_User stories.odt",

"id":"defaultFileSystemItemFactory#default#70235c0e-ad97-421e-92f6-4d71afaed9c1",

"path":"/org.nuxeo.drive.service.impl.DefaultTopLevelFolderItemFactory#/defaultSyncR
ootFolderItemFactory#default#ccbd9a3c-85d3-4589-ab82-8c28a22b50db/defaultFileSystemI
temFactory#default#70235c0e-ad97-421e-92f6-4d71afaed9c1",
    "userName":"joe"
}
```

We can notice in particular that:

- The request contains all required parameters for the batch execution in the `JSON` data, such as `batchId, operationId, parentId` (operation parameter).
- The response contains the representation of the `FileSystemItem` resulting of the adaptation of the `NUXEO_WCM_MODULE_User stories.odt` document that has been remotely created, including all the data needed to create a new pair in `last_known_states` and populate its remote part with `remote_ref` = `id`, `remote_name` = `name`, `remote_digest` = `digest`, etc.

Application update

Since version 1.3.0611, Nuxeo Drive is able to update itself with a newer or an older version (such a downgrade can be required if the Nuxeo server version is too old for the client version). This is very useful as it allows the user to keep the application up-to-date without having to manually install a new version.

Principle

Every hour (`update-check-delay` command line parameter) Nuxeo Drive checks the update site for a newer version compatible with the version of the Nuxeo server it is connected to. If such a version is available the systray icon changes colour and the "Update Nuxeo Drive" entry appears in the systray menu, allowing the user to trigger the update: download of the new version, installation, restart (uses `subprocess.Popen(args)`). At startup Nuxeo Drive checks if the current version stored in the database (`client_version`) is different from the code version, in which case it displays a dialog box to inform about the update to the newer version.

The user can configure Nuxeo Drive to automatically update itself in case of an available update through the General settings. This is persisted in the `auto_update` field.

Note that if Nuxeo Drive detects that the client version is not compatible with the server version it will stop the synchronization thread and display the "Upgrade/Downgrade required" entry in the systray menu.

Implementation

At startup, when the Setting dialog is accepted and every hour, Nuxeo Drives refreshes the update information by:

- Updating the `server_version` and `update_url` fields from the Nuxeo server by calling the `NuxeoDrive.GetClientUpdateInfo` operation.
- Instantiating or using the existing `AppUpdater` (singleton) to communicate with the update site using the `update_url` and `server_version` in order to get the update status among: `up_to_date`, `upgrade_needed`, `downgrade_needed` or `update_available`.
- Updating the systray menu if needed.

The main framework used for handling the update is esky, which we've wrapped inside the `AppUpdater` class to handle the custom logic related to the client and server compatibility.

Basically esky is able to:

- Fetch a given version from the update site, ie. download the ZIP file to a temporary directory of the application installation directory.
- Install a given version in the application installation directory.
- Cleanup old versions from the application installation directory.
- Do all this in a filesystem transactional way so that it keeps the app safe in the face of failed or partial updates.

Update site

The update site structure is detailed in a dedicated page of the Nuxeo Drive Installation and Administration section which also explains how to set up a custom update site in case the one provided by Nuxeo wouldn't fit your needs.

Logs

Nuxeo Drive logs are available:

- In the `.nuxeo-drive/logs/nxdrive.log` file,
- In the console by running the `ndrive console` command.

Options are available to set the log level:

- For the log file: `--log-level-file LOG_LEVEL_FILE`
- For the console log: `--log-level-console LOG_LEVEL_CONSOLE`

Server

The server-side part of Nuxeo Drive is distributed as a Marketplace package available from Nuxeo Connect (stable releases). You can also fetch the latest development version of the Marketplace package from our Jenkins continuous integration server (use at your own risk).

Java API

The server-side API mainly relies on the Java types described below.

FileSystemChangeFinder

Allows to find document changes. Default implementation is `AuditChangeFinder` , that relies on Nuxeo audit logs.

FileSystemItem

Representation of a file or folder on a file system. It is used as an adapter of `DocumentModel`.

FileItem

Representation of a file, i.e. a downloadable `FileSystemItem`. In the case of a `DocumentModel` backed implementation, the backing document holds a binary content. Typically a File, Note or Picture.

FolderItem

Representation of a folder. In the case of a `DocumentModel` backed implementation, the backing document is Folderish. Typically a Folder or a Workspace.

DocumentBackedFileItem

`DocumentModel` backed implementation of a `FileItem`.

DocumentBackedFolderItem

`DocumentModel` backed implementation of a `FolderItem`.

DefaultTopLevelFolderItem

Default implementation of the top level `FolderItem`, ie. the Nuxeo Drive local folder.

DefaultSyncRootFolderItem

Default implementation of a synchronization root `FolderItem`.

FileSystemItemAdapterService

Service for creating the right `FileSystemItem` adapter depending on `DocumentModel` type or facet. Factories can be contributed to implement a specific behavior for the `FileSystemItem` adapter creation.

FileSystemItemFactory

Interface for the classes contributed to the `fileSystemItemFactory` extension point of the `FileSystemItemAdapterService`. Allows to get a `FileSystemItem` for a given `DocumentModel` or a given `FileSystemItem` id.

Default factories are:

- `DefaultFileSystemItemFactory` : applies to Folderish and BlobHolder documents.
- `DefaultSyncRootFolderItemFactory` : applies to synchronization roots.
- `DefaultTopLevelFolderItemFactory` : appplies to the top level `FolderItem`, ie. the Nuxeo Drive local folder.

FileSystemItemManager

Provides an API to manage usual file system operations on a `FileSystemItem` given its id. Allows the following actions:

- Check existence,
- Read,
- Read children,
- Create,
- Update,
- Rename,
- Delete,
- Move.

Execution Process

The server-side API mainly allows to:

- Get the list of changed documents as `FileSystemItem` objects.

- Handle usual file system operations on a `FileSystemItem` given its id, such as read, get children, create, update, rename, delete and move.

The graph below illustrates the execution process corresponding to the Automation call samples seen earlier: remote polling and synchronization of a locally created file.

Customization

As usual in the platform, Nuxeo Drive relies on extension points and contributions to these extension points. The main entry point for customization is the `FileSystemItemAdapterService` , that exposes the `fileSystemItemFactory` and `topLevelFolderItemFactory` extension points.

You can take a look at:

- The default contributions embedded in the Marketplace package.
- The User workspace based hierarchy contributions embedded in the Marketplace package as an example but not deployed by default. For details, see **NXP-10663** - Add user workspace based file system tree contribution (Resolved) .
- The User workspace and permission based hierarchy contributions that are not embedded in the Marketplace package. For details, see **NXP-11074** - Implement permission based hierarchy contribution (Resolved) .
- The series of blog posts about Nuxeo Drive.

In this section

- Automation Operations
  - NuxeoDrive.GetChangeSummary
  - NuxeoDrive.GetTopLevelFolder
  - NuxeoDrive.GetChildren
  - NuxeoDrive.GetFileSystemItem
  - NuxeoDrive.FileSystemItemExists
  - NuxeoDrive.CreateFile
  - NuxeoDrive.CreateFolder
- NuxeoDrive.Rename

- NuxeoDrive.UpdateFile
  - NuxeoDrive.Delete
  - NuxeoDrive.CanMove
  - NuxeoDrive.Move
  - NuxeoDrive.GenerateConflictedItemName
- Data Exchange Format
  - Remote Polling Through NuxeoDrive.GetChangeSummary
  - Synchronization of a Locally Created File
-
-

Related Documentation

- How to Manually Initialize or Deploy a Nuxeo Drive Instance
- Nuxeo Drive Update Site
- Nuxeo Drive user documentation

Nuxeo Drive Update Site

Since version 1.3.0611, Nuxeo Drive is able to update itself with a newer or an older version (such a downgrade can be required if the version of the Nuxeo server Nuxeo Drive is connected to is too old for the client version). This is very useful as it allows the user to keep the application up-to-date without having to manually install a new version.

The update process relies on an update site holding the Nuxeo Drive binary packages for Windows (MSI) and OS X (DMG) as well as available updates for both platforms packaged as ZIP files. This page aims to explain how this update site is structured in case you would like to host your own one to manage the update policy instead of relying on the official Nuxeo update site.

Using a Custom Update Site

The update site URL is configured server-side by the `org.nuxeo.drive.update.site.url` Nuxeo framework property, default value being http://community.nuxeo.com/static/drive/. So you only need to override this property in `nuxeo.conf` to make the Nuxeo Drive updater point at the custom site.

Of course this update site will need to have the same structure as the one provided by Nuxeo, for the Nuxeo Drive updater to be able to communicate with it.

Update Site Structure

The Nuxeo Drive update site is currently implemented as a simple directory listing served by Apache and that holds the following items.

Binary Packages

The site must hold the Nuxeo Drive binary packages released for Windows (MSI) and OS X (DMG), for instance: `nuxeo-drive-1.3.0611-win32.msi` and `nuxeo-drive-1.3.0611-osx-10.7.dmg`.

These packages are aimed to be manually downloaded for the first Nuxeo Drive installation, though they can also be used to manually install a newer version in the place of an existing installation of Nuxeo Drive.

Esky Compliant ZIP Files

It must also hold available updates for both platforms packaged as esky compliant ZIP files (the auto-update framework for frozen Python applications). They must respect the following pattern: `<application_name>-x.y.zzzz.<platform>`, for instance: `nuxeo-drive-1.4.0125.win32.zip` and `Nuxeo Drive-1.4.0125.macosx-10_7-x86_64.zip`.

Metadata JSON Files

The update site must also contain JSON files representing the metadata about client and server versions. For instance:

- `1.3.0611.json` holds the minimum Nuxeo server version compatible with Nuxeo Drive 1.3.0611.
- `5.9.4.json` holds the minimum client version compatible with a Nuxeo Platform 5.9.4 instance.

This is how the Nuxeo Drive updater is able to compute the update status.

Let's say the Nuxeo Drive client version is 1.3.0611 and the version of the Nuxeo Platform instance it is connected to is 5.9.4. It will first read the `5.9.4.json` file to see if the client version is compatible, meaning greater or equal than the `nuxeoDriveMinVersion` element. Let's say `nuxeoDriveMinVersion` is equal to 1.3.0611, we now know that an upgrade to a newer client is not required.

An update could still be available. To figure this out, the updater will check the JSON metadata files of all client versions compatible with the 5.9.4 server version looking for one more recent than 1.3.0611. Imagine there is a `1.4.0125.json` file with 5.9.4 as a value of the `nuxeoPlatformMinVersion` element. Then, if the update site holds a ZIP file matching the 1.4.0125 version and the client platform, the update status will be `update_available`, allowing Nuxeo Drive to download the ZIP file and launch the update process.

This means that:

- **For each Nuxeo Drive version deployed on the update site a JSON metadata file with the corresponding version needs to be deployed** (for instance 1.3.0611.json). It should have the following structure:

```
{"nuxeoPlatformMinVersion": "5.6"}
```

- **For each Nuxeo Platform version to which Nuxeo Drive might get connected a JSON metadata file with the corresponding version needs to be deployed** (for instance 5.9.4.json). It should have the following structure:

```
{"nuxeoDriveMinVersion": "1.3.0414"}
```

Symbolic Links to Packages

Symbolic links to both packages (MSI and DMG) of the latest Nuxeo Drive version available for a given version of the Nuxeo Platform, for instance in http://community.nuxeo.com/static/drive/latest/5.9.4/. This is used for the first download of Nuxeo Drive from the Nuxeo Drive tab of the user's Home on the Nuxeo Platform UI. The download links are generated server-side following this pattern: `<update_site_URL>/latest/<Nuxeo_distribution_version>/nuxeo-drive.<extension>`, for instance http://community.nuxeo.com/static/drive/latest/5.9.4/nuxeo-drive.msi.

In this section

Other documentation about Nuxeo Drive
• How to Manually Initialize or Deploy a Nuxeo Drive Instance • Nuxeo Drive user documentation • Nuxeo Drive developer documentation

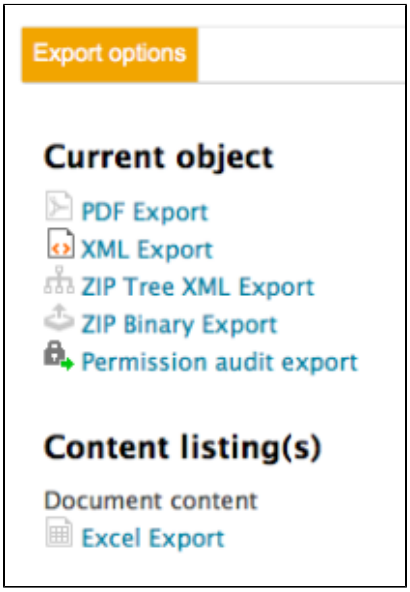
Nuxeo Groups and Rights Audit

The Nuxeo Groups and Rights Audit addon generates an Excel matrix listing every exported documents with permissions for each user.

Installation

The Nuxeo Groups and Rights Audit package requires no specific installation steps. It can be installed like any other package from the Marketplace or from the Admin Center.

After it has been installed, administrators have a new export option available, called "Permission audit export".



Configuration

Setting Up e-Mail Sending

The Nuxeo Groups and Rights Audit addon sends email to the administrator who requested the audit. So your Nuxeo server must be able to reach an e-mail server. This is the same configuration that the one required for the email alerts to work. See how to enable e-mail alerts.

Setting Up a Higher Timeout

The default timeout to process the export of rights is 1200 seconds (20 minutes). You can change this default timeout by adding the parameter `nuxeo.audit.acl.timeout` to the nuxeo.conf file and defining another value than 1200, like 3600 (1 hour) for instance.

In this section

• Installation
• Configuration
  • Setting Up e-Mail Sending
  • Setting Up a Higher Timeout

Related Documentation

- Nuxeo Groups and Rights Audit user doc

Nuxeo jBPM

The Nuxeo jBPM package was last released in Fast Track version 5.7.2. Please check its latest LTS (5.6) documentation.

Nuxeo jBPM: Enable jBPM Workflow on Your Document Type

This documentation applies to target versions of the Nuxeo Platform up to 5.5. It also applies to later versions if the jBPM add-on (nuxeo-platform-jbpm) is installed.

By default, the workflow is not enabled on your custom document types (this is done by default from Studio version 2.4).

You need to declare two distinct extensions from the **Advanced Settings** > **XML Extension** menu.
Copy the content of the extensions below and replace "CUSTOM DOCTYPE ID" by your document type ID.

- The first extension lists the workflows you want to be available for your custom document types.

```
<extension target="org.nuxeo.ecm.platform.jbpm.core.JbpmService"
    point="typeFilter">

    <type name="CUSTOM DOCTYPE ID">
      <processDefinition>review_parallel</processDefinition>
      <processDefinition>review_approbation</processDefinition>
    </type>

 </extension>
```

- The second extension says that the default workflow is available for your custom document type. Note that since Nuxeo version 5.4.3, this contribution is no more needed:

```
<extension target="org.nuxeo.ecm.platform.actions.ActionService"
    point="filters">

    <filter id="jbpm-process" append="true">
      <rule grant="true">
        <type>CUSTOM DOCTYPE ID</type>
      </rule>
    </filter>

  </extension>
```

This will be useful if you want to use the jBPM workflows. Default workflows enable you to control the life cycle state through a series or a set of human validation tasks configured by the the workflow initiator himself.

If you want to implement a workflow that is more "controlled", not built by the end user, you can leverage the "CreateTask" Operation, such as it is done in this simple tutorial. Tasks created with the CreateTask operation appear on the Summary tab.

Other pages about Nuxeo jBPM

- Nuxeo jBPM user documentation

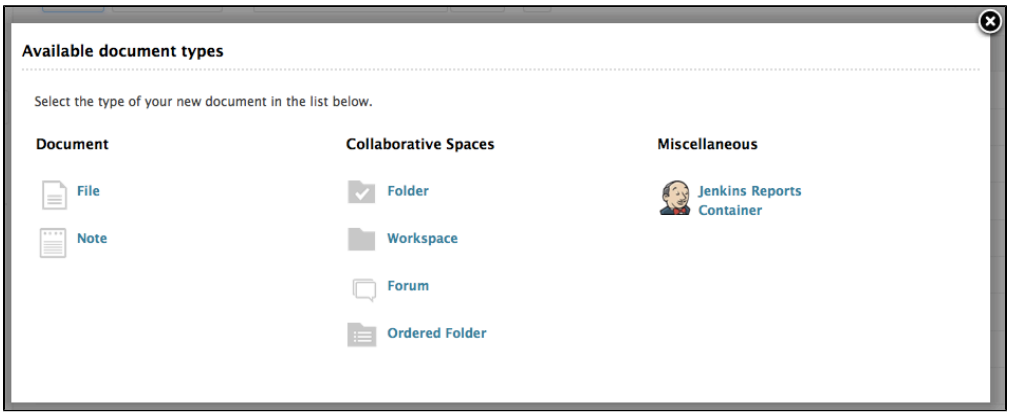
Nuxeo Jenkins Report

The Nuxeo Jenkins Report addon enables users of the Nuxeo Platform to generate and send reports on the status of the Continuous Integration on Jenkins, directly from the Nuxeo Platform. This addon is for development teams, to help them follow and share the status of their continuous integration, while leveraging the content management features of the Nuxeo Platform.

Installation

The Nuxeo Jenkins Report addons package requires no specific installation steps. It can be installed like any other package from the Marketplace or from the Admin Center.

After you installed the package, a new document type is available for creation in workspaces and reports: the Jenkins Reports Container.



Configuration

Since this addon enables users to send the report from the Nuxeo Platform, your Nuxeo server must be able to reach an e-mail server. This is the same configuration that the one required for the email alerts to work. See how to enable e-mail alerts.

In this section

Related Documentation

- Jenkins duty
- Nuxeo Jenkins Report user documentation

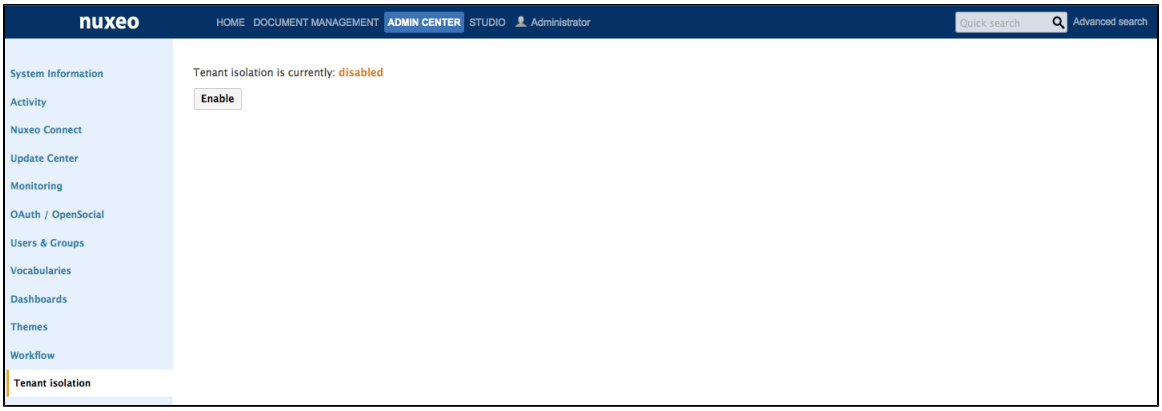
Nuxeo Multi-Tenant

The Multi-tenant addon enables to have domains, or tenants, that are independent from each other, with their own users, vocabulary values etc.

Installation

The Nuxeo Multi-tenant package requires no specific installation steps. It can be installed like any other package from the Marketplace or from the Admin Center.

After you installed it, a tab **Tenant isolation** is available in the Admin Center.



Related Documentation

- Nuxeo Multi-tenant user documentation

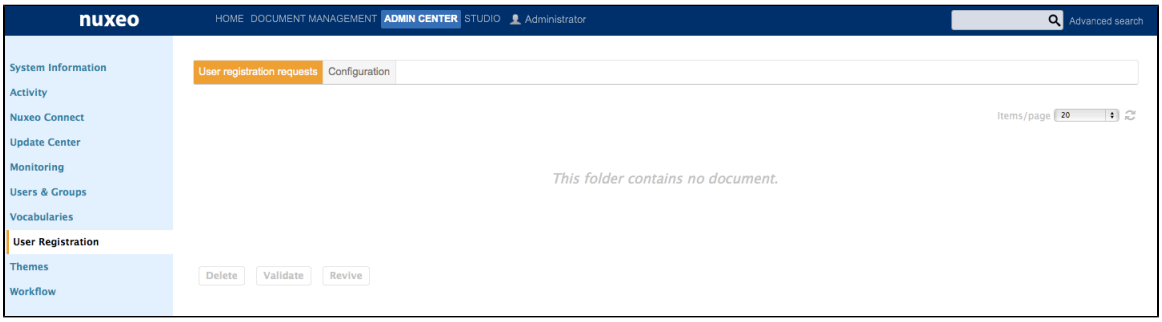
Nuxeo Platform User Registration

The Nuxeo Platform User Registration addon enables users to invite external users to access a specific space of the Platform or a limited set of spaces. The invitations must be approved by an administrator of the Platform.

Installation

The Nuxeo Platform User Registration package requires no specific installation steps. It can be installed like any other package from the Marketplace or from the Admin Center.

After the package has been installed, a new User Registration tab is available in the Admin Center.

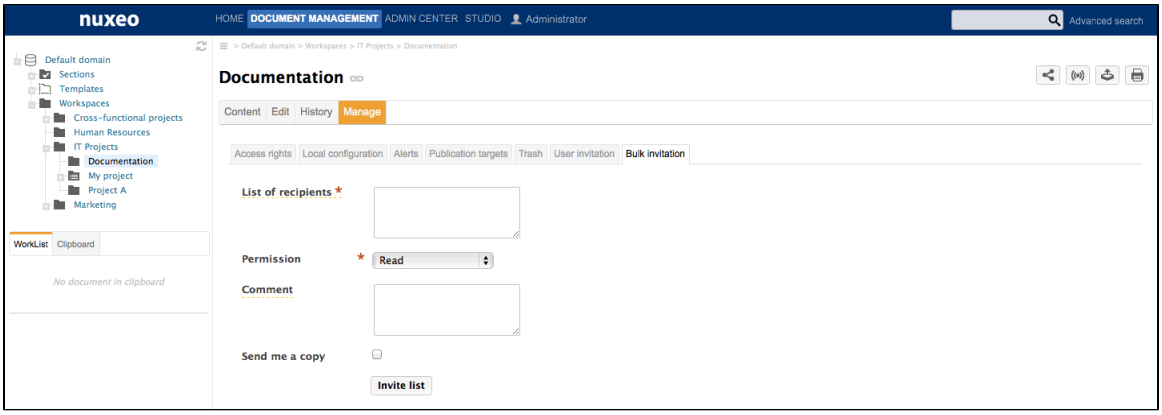


In this section

- Installation
- Configuration
  - Setting Up Email Sending
  - Global Configuration

Users with Manage rights on workspaces and section have two new subtabs in the Manage tab:

- User invitation
- Bulk invitation



Configuration

Setting Up Email Sending

The Nuxeo Platform User Registration addon sends email to the invited user with his credentials. So your Nuxeo server must be able to reach an e-mail server. This is the same configuration that the one required for the email alerts to work. See how to enable e-mail alerts.

Global Configuration

The administrators can set up some configuration directly from the **Admin Center** > **User registration** > **Configuration** tab.

Possible configuration options are:

Field	Description
Allow new user creation	Enables users to invite user that don't have an account on the Platform. A new user account is then created. The new user is not included in any group by default.
Force rights assignment	This option is useful when user is manually created or comes from another system.
Direct validation if user exists	If a user invites a user that has already been invited to another space and so who already has a account on the Platform, then the administrators don't have to approve the invitation again. It is directly approved by the system.
Local registration tab	Displays a User registration requests subtab in the local Manage tab of a space, that displays the invitations that were done from the current space and their status.

Nuxeo Poll

The Nuxeo Poll package enables Nuxeo Platform users to create surveys and have a visual overview of the results.

There are no specific requirements to install the Nuxeo Poll package. You can install it from the Marketplace or from the Admin Center.

To check that the Nuxeo Poll package was correctly installed, browse the Nuxeo Platform with the Administrator user: you can see that workspaces, sections ans templates now have an additional tab, called **Polls**.

Other documentation about this package
• Nuxeo Poll user documentation

Nuxeo Quota: Enabling Quotas on Document Types

By default, quotas are available on domains and workspaces only. It is possible to enable them on other document types using Nuxeo Studio:

1. Create a new XML extension with the following content:

```
<extension target="org.nuxeo.ecm.platform.actions.ActionService"
    point="filters">
    <filter id="QUOTA_MANAGABLE_DOCTYPES" append="true">
      <rule grant="true">
        <type>Folder</type>
      </rule>
    </filter>
</extension>
```

   Replace Folder in the <type> tag by the document type(s) on which you want to enable quotas.
2. Update your Nuxeo instance with the Studio customization.

If you don't want to use Studio and prefer using your IDE, you can just add a contribution with the XML above.

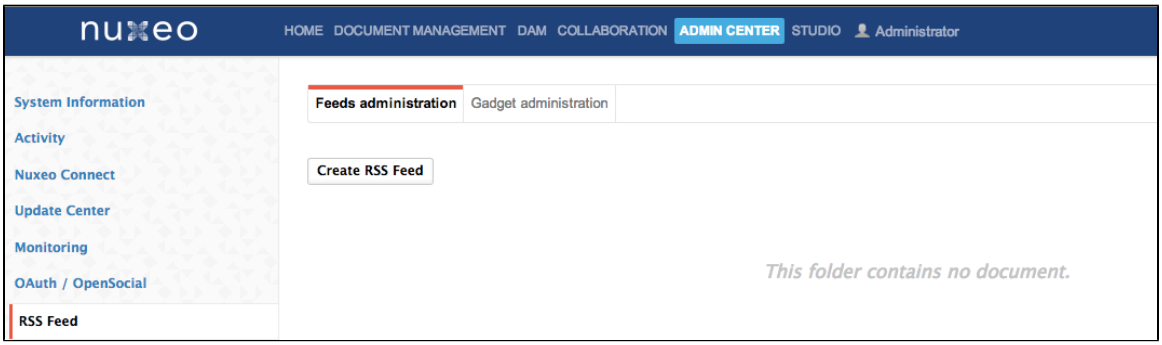
Nuxeo RSS Reader

Nuxeo RSS Reader is a Nuxeo add-on which provides an OpenSocial gadget displaying RSS feeds.

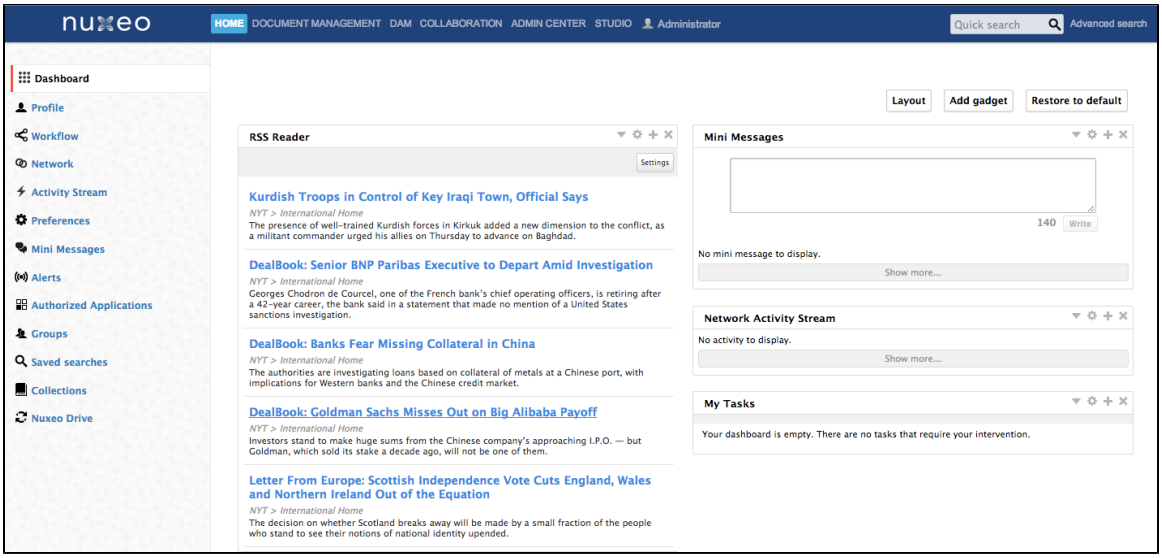
These feeds can be configured by the Administrator through the Admin Center and by the user himself.

The Nuxeo RSS Reader package requires no specific installation steps. It can be installed like any other package from the Marketplace or from the Admin Center.

After you installed the Nuxeo RSS Reader, a new vertical tab appears in the Admin Center called **RSS Feed.**



A new gadget is also available on your home page called **RSS Reader.**



Related Documentation

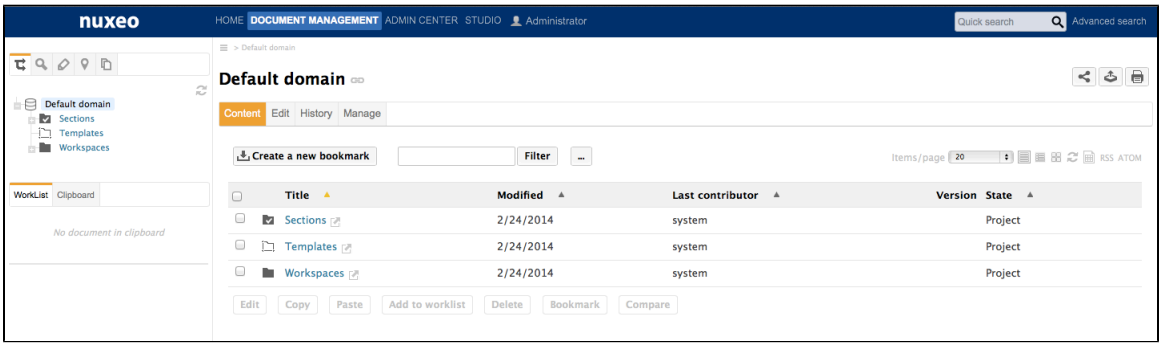
- Nuxeo RSS Reader user documentation

Nuxeo Shared Bookmarks

The Nuxeo Shared Bookmarks addon enables users to bookmark documents and organize their bookmarks in folders. They can thus organize existing documents in a new tree structure without duplicating content.

The Nuxeo Shared Bookmarks addon requires no specific installation steps. It can be installed like any other package from the Marketplace or from the Admin Center.

After the package is installed, a new button **Bookmark** button is available next to the Add to worklist, Copy, Delete buttons, and a **Create a new bookmark** button.



Related Documentation

- Nuxeo Shared Bookmarks user documentation

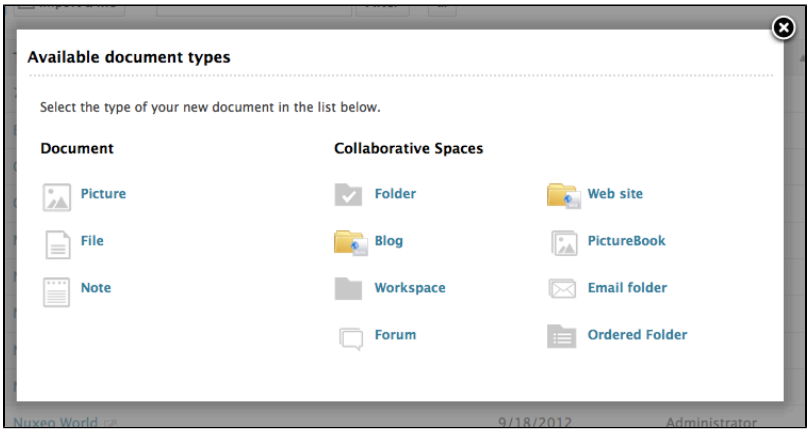
Nuxeo Sites and Blogs

The Nuxeo Sites and Blogs package provides two new documents types to the Platform: websites and blogs. Websites and Blogs are collaborative documents that are web publishing oriented. As so, they have a second interface that makes it easy to display the documents of a workspace to the public. These specific presentations are built using Nuxeo WebEngine.

The Nuxeo Sites and Blogs package requires no specific installation steps. It can be installed like any other package from the Marketplace or

from the Admin Center.

After the package is installed, two new document types are available when you click on the **New** button: Websites and Blogs.



Related Documentation

- Nuxeo Sites and Blogs user documentation

Resources Compatibility

The Resources Compatibility add-on provides backward compatibility with web resources (icons, JavaScript, ...) that have been removed from the previous LTS release of Nuxeo Platform.

It requires no specific installation steps. It can be installed like any other package from the Marketplace or from the Admin Center .

It allows you to keep using elements that we have removed from a version of the Platform to another, like icons, XHTML templates, images, etc, in your customizations. To know which resources it holds, you need to explore the nuxeo-resources-compat JAR from `$NUXEO/nxserver/bundles`.

For each LTS version, we provide a new version of the add-on that holds the resources that have been removed since the previous LTS version

Once your platform is upgraded, we highly recommend that you take the resources you need and move them into a custom bundle. The resources-compat add-on is meant to help you keep compatibility as the platform and your project evolve, but is not meant to be a permanent solution.

Smart Search

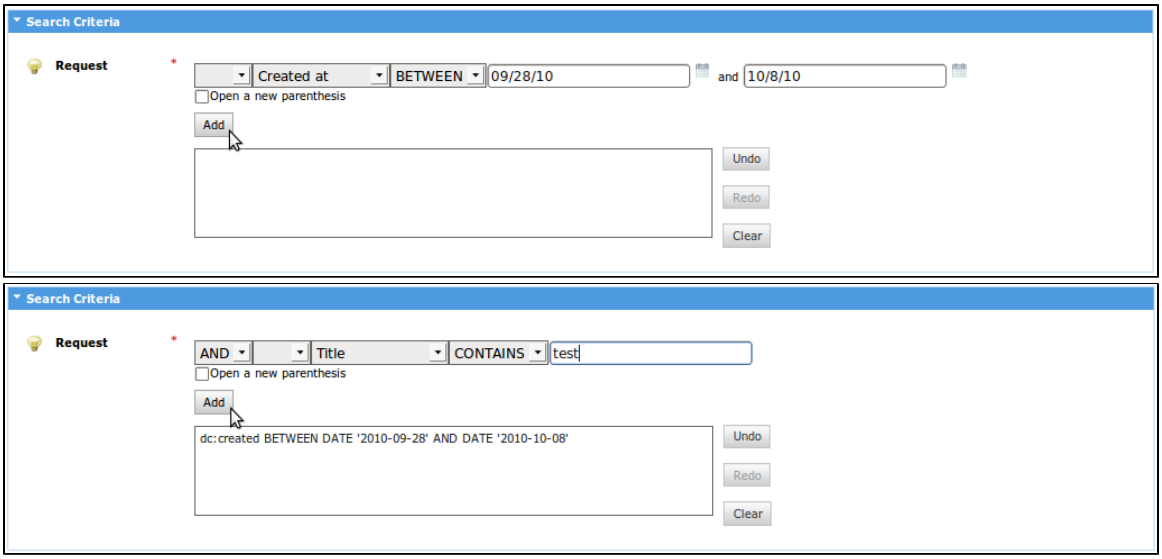
The Smart Search package is a query engine that adds a new search form in the application from which you can build your queries and save them in smart folders. It offers search criteria on content, dates, and metadata.

The Smart search package requires no specific installation steps. It can be installed like any other package from the Marketplace or from the Admin Center.

After the package is installed, a new Smart search link is available in the top right corner of the application.

Smart Query Configuration

The smart query is designed to work in conjunction with a content view.
This content view search layout displays a selector to help building a query part, and a text area with the existing query parts already aggregated:



The SmartQuery interface is very simple: it can build a query (or query part) and can check if it is in a valid state.

The IncrementalSmartQuery abstract class holds additional methods for a good interaction with UI JSF components. It is able to store an existing query part, and has getters and setters for the description of a new element to add to the query.

The IncrementalSmartNXQLQuery class implements the `org.nuxeo.ecm.platform.smart.query.SmartQuery` interface and generates a query using the NXQL syntax.

The seam component named "smartNXQLQueryActions" exposes an instance of it, given an existing query part, and is used to update it on Ajax calls.

The complete list of layouts used to generate this screen is available here: smart-query-layouts-contrib.xml.

The content view is configured to use the layout named "`nxql_incremental_smart_query`" as a search layout, and this content view is referenced both in the search form and search results templates : https://github.com/nuxeo/nuxeo-platform-smart-search/blob/release-5.8/nuxeo-platform-smart-query-jsf/src/main/resources/OSGI-INF/smart-query-contentviews-contrib.xml.

The easiest way to customize available query conditions is to override the definition of the layout named "`incremental_smart_query_selection`". This layout uses the template incremental_smart_query_selection_layout_template.xhtml that accepts one property named "hideNotOperator". This property, if set to true, will hide the selection of the 'NOT' word that can be added in front of each criterion. If you do so, operators should include negative operators.

Here is an explanation on how to define this layout widgets, that need to be of type "incremental_smart_query_condition" to ensure a good behaviour with other layouts.

As a simple example, let's have a look at the widget to add a condition on the title:

```
<widget name="nxql_smart_query_condition_title"
  type="incremental_smart_query_condition">
  <labels>
    <label mode="any">title</label>
  </labels>
  <translated>true</translated>
  <properties widgetMode="edit">
    <property name="searchField">dc:title</property>
    <propertyList name="availableOperators">
      <value>CONTAINS</value>
      <value>LIKE</value>
      <value>=</value>
    </propertyList>
  </properties>
  <subWidgets>
    <widget name="title" type="text">
      <fields>
        <field>stringValue</field>
      </fields>
    </widget>
  </subWidgets>
</widget>
```

The properties "searchField" and "availableOperators" are used to set the left expression of the condition and the operator. The subwidget is a standard widget of type "text". It is bound to the "stringValue" field so it will be stored in the smart query instance field with the same name. Other additional properties supported by the "text" widget type can be added here (for instance, the "required" or "styleClass" properties).

Here is the complete list of available field bindings:

- booleanValue
- stringValue
- stringListValue
- stringArrayValue
- datetimeValue
- otherDatetimeValue (to be used in conjunction with datetimeValue)
- dateValue
- otherDateValue (to be used in conjunction with dateValue)
- integerValue
- floatValue (to bind a Double instance)

As a more complex example, let's have a look at the widget used to add a condition on the modification date:

```
<widget name="nxql_smart_query_condition_modified"
  type="incremental_smart_query_condition">
  <labels>
    <label mode="any">label.dublincore.modified</label>
  </labels>
  <translated>true</translated>
  <properties widgetMode="edit">
    <property name="searchField">dc:modified</property>
    <propertyList name="availableOperators">
      <value>BETWEEN</value>
      <value>&lt;</value>
      <value>&gt;</value>
    </propertyList>
  </properties>
  <subWidgets>
    <widget name="modified_before" type="datetime">
      <fields>
        <field>dateValue</field>
      </fields>
      <properties widgetMode="edit">
        <property name="required">true</property>
        <property name="format">#{nxu:basicDateFormater()}</property>
      </properties>
    </widget>
    <widget name="and" type="text">
      <widgetModes>
        <mode value="any">
          #{not empty value.conditionalOperator and
          value.conditionalOperator!='BETWEEN'?'hidden':'view'}
        </mode>
      </widgetModes>
      <properties mode="any">
        <property name="value">
          &amp;nbsp;#{messages['label.and']}&amp;nbsp;
        </property>
        <property name="escape">false</property>
      </properties>
    </widget>
    <widget name="modified_after" type="datetime">
      <fields>
        <field>otherDateValue</field>
      </fields>
      <widgetModes>
        <mode value="any">
          #{not empty value.conditionalOperator and
          value.conditionalOperator!='BETWEEN'?'hidden':mode}
        </mode>
      </widgetModes>
      <properties widgetMode="edit">
        <property name="required">true</property>
        <property name="format">#{nxu:basicDateFormater()}</property>
      </properties>
    </widget>
  </subWidgets>
</widget>
```

It is more complex as some subwidgets should not be shown depending on the chosen operator: when operator "BETWEEN" is selected, all

of the three subwidgets should be displayed, whereas when other operators are selected, only the first subwidget should be shown. This is achieved by setting the widget mode according to the selected value.

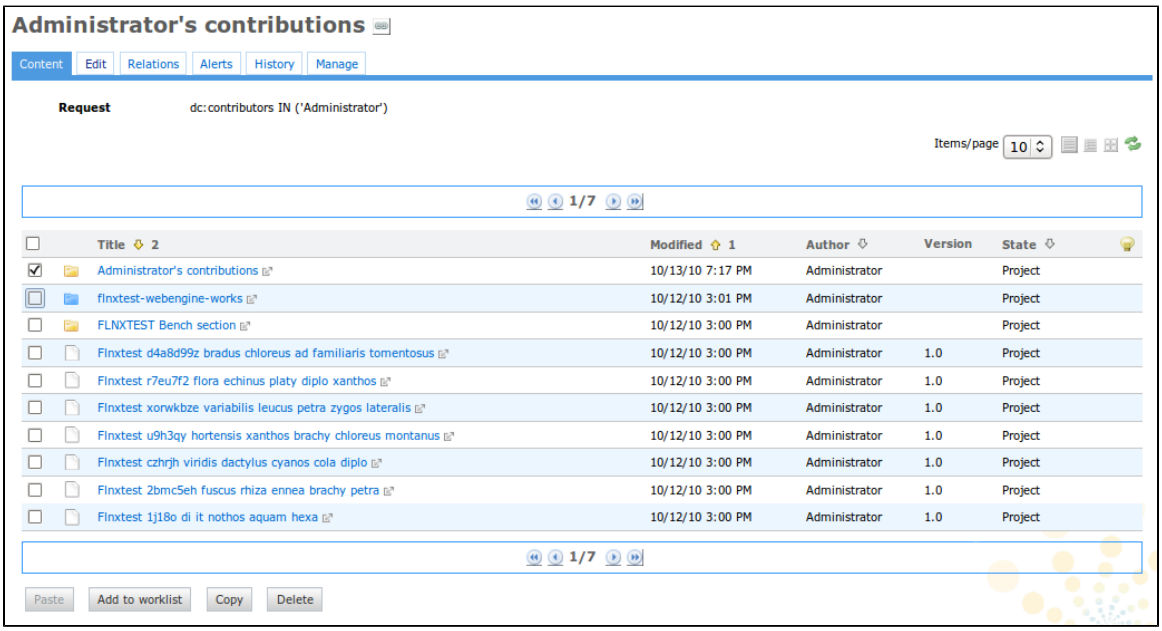
Let's have a close look at the condition *"#{not empty value.conditionalOperator and value.conditionalOperator!='BETWEEN'?'hidden':mode}"*. In this expression, "value" references the value manipulated by the widget (e.g. the smart query instance) and "mode" references the mode as passed to the layout tag. Both of these values are made available by the layout system. Here, if the conditional operator is not empty, and is different from 'BETWEEN', the widget should be hidden. Otherwise, it can be shown in the originally resolved mode. The widgets shown when the conditional operator is empty should be suitable for the first operator within the list of available operators.

An EL bug will not allow you to use a non-static value at the second position in the conditional expression: *#{condition?mode:'hidden'}* will throw an error. But placing the variable "mode" at the end of the expression is ok: *#{not condition?'hidden':mode}*.

Smart Folder Configuration

The smart folder creation and edition pages is very close to the smart search form. It reuses the same widget types, including some adjustments since the bound values are kept in its properties instead of a backing seam component. Its layout definition is here: smart-folder-layouts-contrib.xml. It also includes the definition of a widget in charge of displaying the content view results.

Note that it needs another content view to be defined (see smart-folder-contentviews-contrib.xml) so that this content view uses the query, sort information, result columns and page size as set on the document properties (note the usage of tags parameter, sortInfosBinding, resultColumns and pageSizeBinding).



Result Layout, Column and Sort Selection

The layout used to display the column selection, the sort information selection, and to display the search results, is the generic layout "`search_listing_layout`" also used in the advanced search form. If it is changed, it needs to be kept consistent between all the places referencing it:

- the smart search form
- the smart query content view result layout and result columns binding
- the smart folder layout in edit mode
- the smart folder content view result layout and result columns binding (displayed via its layout in view mode)

Please refer to the Advanced Search documentation for more information about this layout customization.

Related pages in this documentation

- Content Views
- Custom Page Providers

Related pages in other documentations

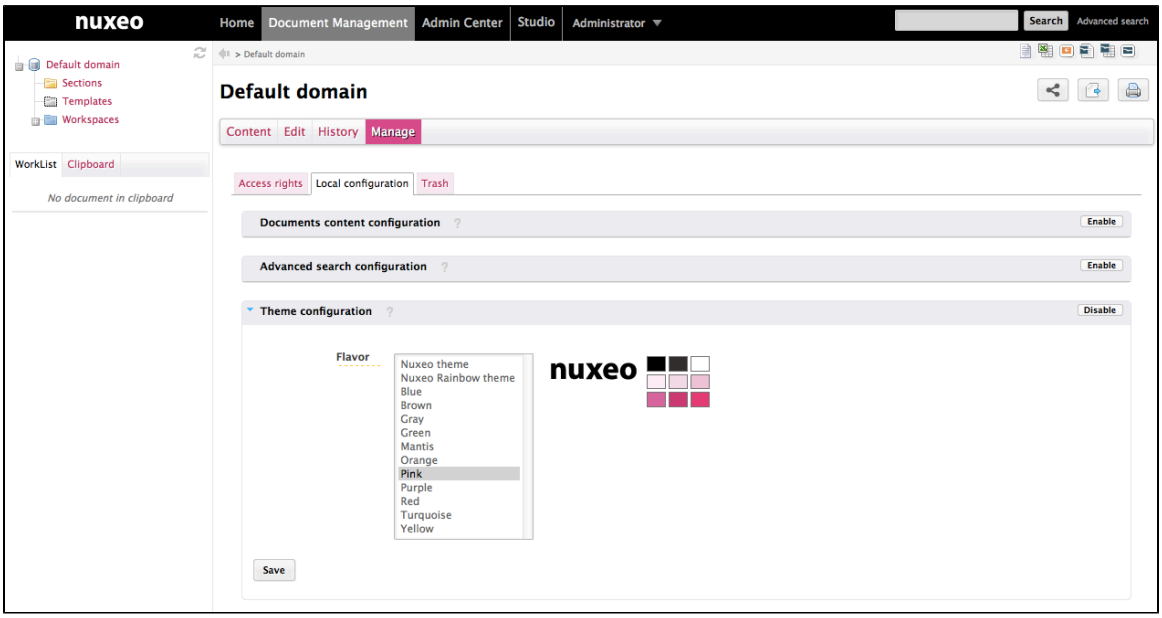
- Smart Search user documentation
- Smart search operators

Unicolor Flavors Set

This Unicolor Flavors Set package lists a set of flavors that customizes the colors of your workspaces, sections or any space on your Nuxeo application.

The Unicolor Flacors Set package requires no specific installation steps. It can be installed like any other package from the Marketplace or from the Admin Center.

After the package is installed, new flavors are available in workspaces Theme configuration.



Related Documentation

- Unicolor Flavors Set user documentation

Packaging

In the following pages, we will see how to package your customization to make them available in your Nuxeo application.

- Writing a bundle manifest — This page gives some rules for writing a bundle's manifest.
- Creating Nuxeo Packages — This section gives instructions on how to wrap a plugin and its required dependencies into a Nuxeo Package.
- Nuxeo Distributions — With nuxeo-distribution, you can build from Nuxeo sources, or from existing distribution and much more: if you need to assemble your own distribution, you will find in nuxeo-distribution resources, templates and samples on which to base your packaging.
- Nuxeo Deployment Model — The Nuxeo Platform deployment is incremental: the startup process involves different processors for different phases.

Writing a bundle manifest

This page gives some rules for writing a bundle's manifest.

You should first refer to the Component Model page for a better understanding of Nuxeo Runtime, Bundles, Components and services concepts.

Here is some practical information:

- Using the manifest you can define an unique name for your bundle (i.e. the Bundle-SymbolicName). This name is helping the framework to identify the bundle.
- Using the manifest you can define an activator class.
- Using the manifest you can declare bundle dependencies (so that the bundle can be started only when dependencies are resolved). Also, these dependencies are used to determine the visible class path of your bundle. Classes not specified in dependencies will not be visible to your bundle. Bundle dependencies **are ignored** by Nuxeo Runtime launcher.

In the Nuxeo Platform, the best way to initialize your components (without worrying about dependencies) is to use a lazy loading model - so that a service is initialized at the first call. This method also speed the startup time.

Another method is to use the **FRAMEWORK_STARTED** event for initialization. But this should be used with precaution since its use in Nuxeo doesn't respect OSGi specifications - and may change in future.

Here is an example of a minimal manifest as required by Nuxeo.

```
Manifest-Version: 1.0
Bundle-SymbolicName: org.nuxeo.ecm.core.api
Nuxeo-Component: OSGI-INF/DocumentAdapterService.xml,
  OSGI-INF/RepositoryManager.xml,
  OSGI-INF/blob-holder-service-framework.xml,
  OSGI-INF/blob-holder-adapters-contrib.xml,
  OSGI-INF/pathsegment-service.xml
```

Here is the same manifest but OSGI valid (and works in Eclipse):

```
Export-Package: org.nuxeo.ecm.core;api=split;mandatory:=api,
 org.nuxeo.ecm.core.api;api=split;mandatory:=api,
 org.nuxeo.ecm.core.api.security,
 org.nuxeo.ecm.core.api.repository,
 org.nuxeo.ecm.core.api.model.impl.primitives,
 org.nuxeo.ecm.core.api.event.impl,
 org.nuxeo.ecm.core.api.impl.converter,
 org.nuxeo.ecm.core.utils,
 org.nuxeo.ecm.core.api.security.impl,
 org.nuxeo.ecm.core.api.model.impl.osm,
 org.nuxeo.ecm.core.url,
 org.nuxeo.ecm.core.api.impl,
 org.nuxeo.ecm.core.api.operation,
 org.nuxeo.ecm.core.api.model.impl.osm.util,
 org.nuxeo.ecm.core.api.externalblob,
 org.nuxeo.ecm.core.url.nxobj,
 org.nuxeo.ecm.core.api.model,
 org.nuxeo.ecm.core.api.repository.cache,
 org.nuxeo.ecm.core.api.impl.blob,
 org.nuxeo.ecm.core.api.model.impl,
 org.nuxeo.ecm.core.api.blobholder,
 org.nuxeo.ecm.core.api.tree,
 org.nuxeo.ecm.core.api.adapter,
 org.nuxeo.ecm.core.api.local,
 org.nuxeo.ecm.core.url.nxdoc,
 org.nuxeo.ecm.core.api.facet,
 org.nuxeo.ecm.core.api.event
Bundle-ActivationPolicy: lazy
Bundle-ClassPath: .
Manifest-Version: 1.0
Bundle-Name: org.nuxeo.ecm.core.api
Created-By: 1.6.0_20 (Sun Microsystems Inc.)
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Bundle-Version: 0.0.0.SNAPSHOT
Bundle-ManifestVersion: 2
Nuxeo-Component: OSGI-INF/DocumentAdapterService.xml,
  OSGI-INF/RepositoryManager.xml,
  OSGI-INF/blob-holder-service-framework.xml,
  OSGI-INF/blob-holder-adapters-contrib.xml,
  OSGI-INF/pathsegment-service.xml
Import-Package: javax.security.auth,
 javax.security.auth.callback,
 javax.security.auth.login,
```

```
 javax.security.auth.spi,
 org.apache.commons.collections.bidimap,
 org.apache.commons.collections.map,
 org.apache.commons.logging,
 org.nuxeo.common,
 org.nuxeo.common.collections,
 org.nuxeo.common.utils,
 org.nuxeo.common.xmap.annotation,
 org.nuxeo.ecm.core.schema,
 org.nuxeo.ecm.core.schema.types,
 org.nuxeo.ecm.core.schema.types.primitives,
 org.nuxeo.runtime,
 org.nuxeo.runtime.api,
 org.nuxeo.runtime.api.login,
 org.nuxeo.runtime.model,
 org.nuxeo.runtime.services.streaming
Bundle-SymbolicName: org.nuxeo.ecm.core.api;singleton=true
```

```
Eclipse-RegisterBuddy: org.nuxeo.runtime
Eclipse-ExtensibleAPI: true
```

Nuxeo is also using two specific manifest headers:

- **Nuxeo-Component**: Which specify components declared by a bundle (as XML descriptor file paths relative to JAR root);
- **Nuxeo-WebModule**: Which specify the class name of a JAX-RS application declared by a Nuxeo bundle.

Of course these two headers are optional and should be used only when needed.

Bundle Preprocessing

Nuxeo is a very dynamic platform. When building a Nuxeo Application you will get an application template. At each startup, the application files are dynamically updated by each bundle in the application that need to modify a global configuration setting or to provide a global resource. We call this mechanism **preprocessing**.

Creating Nuxeo Packages

This section gives instructions on how to wrap a plugin and its required dependencies into a Nuxeo Package.
A package contains usually new features or patches along with installation instructions. Packages can be downloaded from a remote repository and then installed on a running Nuxeo and possibly uninstalled later. A Nuxeo package is the easiest way to distribute a plugin, as it contains in one single zip file all the bundles, libraries and runtime properties that would be required to make your new plugin work. Nuxeo uses the Nuxeo Package format for distributing all its plugins on Nuxeo Marketplace. We also encourage you to use it for delivering your customization. A Nuxeo Package can be set up either in the admin center on the Nuxeo instance, or using nuxeoctl instructions.

Some packages require the server to be restarted after the install (or uninstall). Each package provides a description of the modifications that should be done on the running platform in order to install a package. We will call "**command**" each atomic instruction of an install or uninstall process. When Commands are revertible - so that for any command execution there must be an inverse command that can be executed to rollback the modification made by the first command. When designing update packages you must ensure the installation is revertible if needed.

In this section

The rollback of an installation is done either when the installation fails (in the middle of the install process), either if the user wants to uninstall the package.

In this chapter we will discuss about the package format, package execution and rollback.

Package Format

A package is assembled as a ZIP file that contains the bundles, configuration files or libraries you want to install, along with some special files that describe the install process.

Here is a list of the special files (you should avoid to use these file names for installable resources)

- **package.xml** - The package descriptor describing package metadata, dependencies and custom handlers to be used when installing.
  See Package Manifest for more details on the file format.
- **install.xml** - A file containing the install instructions. There are two possible formats for this file: either an XML package command file, or an ant script to be used to install the package. Using ant is discouraged, you should envisage to use the package command to describe an installation rather than ant since rollback is ensured to be safe.
  See Scripting Commands for more details on the commands file format.
- **uninstall.xml** - A file containing the uninstall instructions. When using **commands** to describe the install process this file will be automatically generated (so you don't need to write it). When using ant for the install you must write the uninstall ant file too.
- **install.properties** - A Java property file containing user preferences (if any was specified during the install wizard). This file is automatically generated by the installer.
- **backup** - A directory created by the install process (when using **commands** to describe the install) to backup the existing files that were modified. The content of this directory will be used by the rollback process to revert changes.
  See Scripting Commands for more details on rollback.

- **license.txt** - A text file containing the license of the software you want to install. This file is optional.
- **content.html** - A file containing an HTML description of you package. This file can use references to resources (such as images) located in the package zip - for example you may want to display a set of screenshots for the new feature installed by the package. This file is optional.
  See Package Web Page for more details on how to write your package web page.
- **forms** - A directory containing custom wizard form definitions. This directory and all the files inside are optional.
  See Wizard Forms for more details on how to contribute wizard forms.
  There are three type of wizard forms you can contribute:
  - **install.xml** - Describe install forms (i.e. forms added to the install wizard for packages that needs user parametrization)
  - **uninstall.xml** - Uninstall forms (i.e. forms added to the uninstall wizard for packages that needs user parametrization)
  - **validation.xml** - Validation forms (i.e. forms used by the install validator if any is needed)

Apart these special files you can put anything inside a package (web resources, jars, Java or Groovy classes etc.). It is recommended to group your additional resources in sub directories to keep a clean structure for your package.

You can see that most of the files listed above are optional or generated. So for a minimal package you will only need 2 files: the **package.xml** and the **install.xml** file.

The Package Metadata

The package metadata is stored in **package.xml** file. Here is the list of properties defining a package:

- **name:** The package name. The allowed characters are the ones allowed for Java identifiers plus the dash - character.
  Example: nuxeo-automation-core
- **version:** The package version. Apart the tree digit fields separated by dots versions may contain a trailing classifier separated by a dash. Examples: 1.2.3-SNAPSHOT, 1.2, 3, 0.1.0.
- **id:** The package unique identifier. This is automatically generated from the **name** and the **version** as follows: name-version.
  Example: nuxeo-automation-core-5.3.2
- **type:** The package type. One of: *studio*, *hotfix*, or *addon*.
- **dependencies:** A list of other packages that are required by this package. The dependencies are expressed as *packageId:version_range* where version_range is a string of one or two versions separated by a colon : character.
  Example: nuxeo-core:5.3.0 - this means any version of nuxeo-core greater or equals to 5.3.0. Or: nuxeo-core:5.3.0:5.3.2 - any version of nuxeo-core greater or equals than 5.3.0 and less or equal than 5.3.2.
- **platforms:** A list of supported platform identifiers. Examples: dm-5.3.2, dam-5.3.2.
- **title:** The package title to be displayed to the user.
- **description:** A short description of the package
- **classifier:** The package classifier if any. (You can use it to put tags on the package)
- **vendor:** The identifier of the package vendor.
- **home-page:** An URL to the home page of the package (or documentation) if any.
- **installer:** A custom Install Task class that will handle the install process. If not specified the default implementation (which is using commands) will be used.
- **uninstaller:** A custom Uninstall Task that will handle the uninstall process. If not specified the default implementation (which is using commands) will be used.
- **validator:** A custom validator class. By default no validator exists. You can implement a validator to be able to test your installation form the Web Interface.
- **NuxeoValidationState:** The state of Nuxeo's validation process. One of: *none*, *inprocess*, *primary_validation*,*nuxeo_certified*.
- **ProductionState:** One of: *proto*,*testing*, *production_ready*.

For more information on the package properties and the XML format see Package Manifest.

The Install Process

Packages are fetched from a remote repository and cached locally. Once they are cached they can be installed. Fetching a package and putting it into the local cache is transparent to the user - this is done automatically when a user enters a package to see the details about the package.

When saved to the local file system the packages are unzipped - so they will be cached locally as directories. Once a package is cached locally it can be installed by the user. When installing a package the package will be first validated - to check if it can be safely installed on the user platform. If this check fails the installation is aborted. If there are warnings - the user should choose if wants to continue or not.

After validating the package an install wizard will be displayed to the user. The wizard will usually show the following pages:

1. Package license, if any is specified.
2. Custom install forms, if contributed by the package.
3. Summary of things that will be installed - this is the last step the user can abort the installation. When clicking install the install process will start.
4. An install result page. This is either a page of failure either a page of success. In case of success, if the package requires restarting the server then the user is asked whether to restart now or later the server.

Here is a pseudo-code describing how installation is drived by the wizard:

```
LocalPackage pkg = service.getPackage("package_to_install");

Task task = pkg.getInstallTask();

ValidationStatus status = task.validate();
if (status.hasErrors()) {
  // install task cannot be run. show errors to the user
} else if(status.hasWarnings()) {
  // task can be run but there are warnings. show warnings to the user and let it
decide whether or not to run the install task
} else {
  try {
    task.run(userPrefs);
  } catch (Throwabe t) {
    // if an error occurred do the rollback.
    task.rollback();
  }
}
// show install result to the user.
if (task.isRestartRequired()) {
 // ask user to restart
}
```

Using Ant to Install

When using ant you must define two ant scripts: the **install.xml** and **uninstall.xml** files.

Each of these scripts must have at least 2 ant targets. The **default** target of the install.xml script will be used to execute the installation. The target name is not important - it may have any name but should be the default target. The other required target of the script is a target named **rollback** which will be called to do the rollback if anything went wrong during the installation (i.e. during the execution of the default target).

The same rule applies for the uninstall.xml script. This ant script must have at least 2 targets: a default one which will be called to do the uninstall and another one named **rollback** which will be used to perform the rollback if anything went wrong during the uninstall execution.

There is a set of useful properties that will be injected in the ant context and thus are available in ant scripts. See below the list of these properties.

Using ant is not recommended since a safe rollback is difficult to handle.

Using Commands to Install

XML Commands are the default way to describe the installation instructions. The advantage of using commands is that the rollback and uninstall script will be automatically generated - you don't need to code it yourself. Also, you can control commands enabling or validation using EL expressions depending on the state of the target platform were the install is executed.

See Scripting Commands for more details on using commands .

As in ant scripts there is a set of properties you can use in command files to parametrize your commands. Below is the list of available properties.

Context Properties Available in Install Scripts

Here is the list of properties available to install scripts:

- all the system properties in the running JVM.
- **package.id:** The Package identifier.
- **package.name:** The Package name.
- **package.version:** The Package version.
- **package.root:** The root folder of the package (the folder containing the exploded zip).
- **env.server.home:** Since 5.5. The Nuxeo server home. ($NUXEO_HOME).
- **env.home:** The Nuxeo Runtime Environment home. ($NUXEO_HOME/server/default/data/NXRuntime on JBoss,

$NUXEO_HOME/nxserver on Tomcat).
- **env.ear:** JBoss only. The nuxeo.ear directory ($NUXEO_HOME/server/default/deploy/nuxeo.ear).
- **env.lib:** The Nuxeo lib directory (nuxeo.ear/lib on JBoss, $NUXEO_HOME/nxserver/lib on Tomcat).
- **env.syslib:** The host application lib directory ($NUXEO_HOME/lib).
- **env.bundles:** The Nuxeo bundles directory (nuxeo.ear/bundles on JBoss, $NUXEO_HOME/nxserver/bundles on Tomcat).
- **env.config:** The Nuxeo *config* directory (nuxeo.ear/config on JBoss, $NUXEO_HOME/nxserver/config on Tomcat).
- **env.templates:** Since 5.5. The configuration templates directory. ($NUXEO_HOME/templates).
- **env.hostapp.name:** The host application name (Tomcat or JBoss)
- **env.hostapp.version:** The host application version (e.g. Tomcat or JBoss version)
- **sys.timestamp:** The timestamp when the install task was created - a string in the format "yyMMddHHmmss".

Package Manifest

Let's look at a minimal example of package.xml file:

```
<package type="addon" name="nuxeo-automation" version="5.3.2">
  <title>Nuxeo Automation</title>
  <description>A service that enables building complex business logic on top of Nuxeo
services
    using scriptable operation chains</description>
  <platforms>
    <platform>dm-5.3.2</platform>
    <platform>dam-5.3.2</platform>
  </platforms>
</package>
```

This is a minimal package manifest. It is defining a package nuxeo-automation at version 5.3.2 and of type add-on.
The package can be installed on platforms dm-5.3.2 and dam-5.3.2.

TODO: replace fixed versions in platforms with range of versions.

Also, the package title and description that should be used by the UI are specified vy the title and description elements.

Note that the package names used in these examples are fictional.

Lets look at the full version of the same package manifest:

```
<package type="addon" name="nuxeo-automation" version="5.3.2">
  <title>Nuxeo Automation</title>
  <description>A service that enables building complex business logic on top of Nuxeo
services
    using scriptable operation chains</description>
  <classifier>Open Source</classifier>
  <home-page>http://some.host.com/mypage</home-page>
  <vendor>Nuxeo</vendor>
  <installer class="org.nuxeo.connect.update.impl.task.InstallTask" restart="false"/>
  <uninstaller class="org.nuxeo.connect.update.impl.task.UninstallTask"
restart="false"/>
  <validator class="org.nuxeo.MyValidator"/>
  <platforms>
    <platform>dm-5.3.2</platform>
    <platform>dam-5.3.2</platform>
  </platforms>
  <dependencies>
    <package>nuxeo-core:5.3.1:5.3.2</package>
    <package>nuxeo-runtime:5.3.1</package>
  </dependencies>
  <vendor>YourCompany</vendor>
  <supported>false</supported>
  <hotreload-support>true</hotreload-support>

<require-terms-and-conditions-acceptance>false</require-terms-and-conditions-acceptanc
e>
  <NuxeoValidationState>primary_validation</NuxeoValidationState>
  <ProductionState>production_ready</ProductionState>
  <license>LGPL</license>
  <license-url>http://www.gnu.org/licenses/lgpl.html</license-url>
</package>
```

You can see the usage of installer and uninstaller elements. These are used to specify the task implementation to be used when installing and uninstalling.
If these elements are not specified the default values will be used.
If you specify only one of the "class" or "restart" attributes, then the other attributes will get the default values.

See Creating Nuxeo Packages for an explanation of each package property.

Scripting Commands

This documentation needs to be updated

Scripting commands can be used to define the way an installation is done. Usually, when installing a new component you need to execute a limited set of commands like copy, delete, patch etc.

The Package Scripting Commands provides a easy to use format for defining the install logic of a package and more, each built-in command is providing safe rollback in case of install failures.

When writing your installation using scripting commands you don't need to write the uninstall script. This script will be automatically generated after the installation is successfully done.

Lets look at the following install.xml file:

```
<install>
  <copy file="${package.root}/myplugin.jar" tofile="${env.bundles}"
fail="tofile.isFile()" />
  <copy file="${package.root}/my.properties" tofile="${env.config}/my.properties"
    ignore="Platform.isJBoss()" />
  <copy file="${package.root}/mylib-1.2.jar"
tofile="${env.lib}/mylib-{version:.*}.jar"
    ignore="Version.isGreaterOrEqual(version, \"1.2\")" />
  <deploy file="${env.bundles}/my-plugin.jar"/>
  <reload-core/>
</install>
```

In this section

You can see the file is using contextual variables as `env.bundles`. etc. See Creating Nuxeo Packages for the complete list of context variables.

Lets take each command and see what will be executed:

- The first copy command is copying the file named `myplugin.jar` from the package root into the Nuxeo bundles directory (by preserving the file name).

  ```
  <copy file="${package.root}/myplugin.jar" tofile="${env.bundles}"
      fail="tofile.isFile()" />
  ```

  You can see a `fail` attribute was used to put a guard on this command. The guard says that the command should fail if the target file exists (i.e a JAR with the same name already exists in the Nuxeo bundles directory). See below in the Guard Attributes section for more details on using guards.
- The second copy command will copy the `my.properties` file from the package root to the Nuxeo configuration directory but only if the current platform distribution is not based on JBoss.

  ```
  <copy file="${package.root}/my.properties"
  tofile="${env.config}/my.properties"
      ignore="Platform.isJBoss()" />
  ```

  You can see here the usage of another type of guard parameter: `ignore.`
- The third copy command is a bit more complicated: This command is used to upgrade an existing library. It is checking if the version of the library is an old version and should be replaced. If it is the same or a newer version the command will be ignored.

```
<copy file="${package.root}/mylib-1.2.jar"
tofile="${env.lib}/mylib-{version:.*}.jar"
    ignore="Version.isGreaterOrEqual(version, \"1.2\")" />
```

You notice the usage of regular expression variables. The `tofile` value is using an expression of the form `{var:regex}`. This is a file pattern that allow to search for an existing file that match the given pattern. If a matching file is found the pattern portion of the file name will be extracted and inserted into the EL context under the 'var' key. If no matching file is found the command will fail.

So, in our case the first file that matches the name `mylib-*.jar` and is located in the `env.lib` directory will be selected and the value that matched the pattern will be inserted into EL context under the name `version`. That way we can use this variable in our ignore guard parameter. This will check the version of the file that matched to see if the upgrade should be done or not.

The `deploy` command will deploy (e.g. install) the specified bundle into the working Nuxeo Platform. The `deploy` command is needed only if you don't want to restart the server after the install is done. If you skip the deployment command you need to restart the server to have your new bundle deployed.

Note that the deploy won't work for all bundles. Some bundles will need the server to be restarted.

The `reload-core` is simply flushing any repository caches. This is useful if your new bundle is deploying new type of documents. In that case if you don't restart the server you need to flush the repository cache to have you new types working.

Guard Attributes

We've seen that there are two special attributes that can be used on any command:

- `fail`: this is an EL expression that can be used to force command to fail in some circumstances.
- `ignore`: this is an EL expression that can be used to avoid executing the command in some circumstances.

The variable available in EL context are:

- `Version`: a version helper. See the VersionHelper class for the list of all available methods.
  Example: `Version.isGreater(version, '1.0')`
- `Platform`: a platform helper that provides methods to check the type of the currently running Nuxeo Platform (name, version etc.).
  Examples: `Platform.matches("dm-5.3.2")`, `Platform.isTomcat()` etc.
- Pattern Variables: as we seen variable used in file pattern matching are inserted into the EL context.
- custom variables provided by each command. Each command should document which variables are provided.

Command Validation

Before running an installation the install commands are first validated, that means each command is tested in turn to see whether or not it could be successfully executed. All potential failures are recorded into a validation status and displayed to the user. If blocking failures are discovered the install will be aborted, otherwise if only *warnings* are discovered the user is asked whether or not to continue the install.

For example, a validation failure can occurs if a command is trying to upgrade a JAR that is newer than the one proposed by the command.

When validation failures occurs the installation is aborted - so nothing should be rollbacked since nothing was modified on the target platform. Of course even is the validation is successful the install process may fail. In that case an automatic rollback of all modification is done. Lets see now how the rollback is managed.

Command Rollback

Each command executed during an install is returning an opposite command if successful. The opposite command is designed to undo any modification done by the originating command. The originating command is responsible to return an exact opposite command. All built-ins commands are tested and are safe in generating the right rollback is needed to undo the command modifications. When you are contributing new commands you must ensure the rollback is done right.

As an example of describing how a command should generate its rollback command let's take the built-in copy command. To simplify let's say the copy command has a `file` parameter, a `tofile` parameter and an optional `md5` parameter.
When the copy command (`copy1`) is executed it will backup the `fileto` file if any into let say `backup_file`, generate an md5 hash of the `file` content, and then copy the `file` over the `fileto.` This command will generate a rollback command (`copy2`) that will have the following arguments:

- copy2.file = backup_file
- copy2.tofile = copy1.tofile
- copy2.md5 = md5(copy1.file)

The `md5` parameter is used (if set) to test if the target file (of the copy) has the same md5 as the one specified in the command. If not then the

command will fail - since we cannot rollback a file over another one that was modified meanwhile.

This is the approach taken by the `copy` command. You can take any approach you want but in any case the command you implement must provide a safe rollback command.

Here is a short pseudo-code of how the commands are executed (and rollback done if needed)

```
// execute each command in the install.xml file
for (Command cmd : commands) {
  Command rollbackCmd = cmd.execute(task, userPrefs);
  if (rollbackCmd != null) {
     log.addFirst(rollBackCommand);
  }
}
```

So, each time a command is executed the opposite command is logged into an command list named `log`.

If any error occurs during the execution of a command the logged commands are executed to do the rollback. If all the commands are successfully executed then the command log is persisted to a file named `uninstall.xml`. Of course this is the generated uninstall script.

The Uninstall Script

Let see now what is the uninstall script generated by the install file described above. We will show only the first copy rollback command (since the others are similar):

```
<uninstall>
  <copy file="path_to_package/backup/myplugin.jar"
tofile="path_to_bundles/myplugin.jar" md5="aaaa.." />
  ...
</uninstall>
```

You can see the uninstall script doesn't contains variables, neither guard attributes. This is normal since at install time all variables were resolved and replaced with their actual values. Also guard attributes are not useful at uninstall time since the install succeeded. Also, you can note an additional `md5` attribute that represents the md5 hash of the file that has been copied at install time. The uninstall copy will succeed only if this md5 value is the same as the target file it is being to replace. The commands ignored at install time (due to a matching `ignore` attribute) will obviously not be recorded in the uninstall file.

Also, note that in the case that the `copy` command didn't overwrite any file the `rollback` command will be a `delete` command and not a `copy` one.

The `copy` command is more complex but there are commands that are a lot more simpler to implement. For example the opposite of the `reload-core` is itself. There are cases when a command doesn't have an opposite - in that case you should return `null` as the opposite command.

Implementing a Command

The built-ins commands provided by the Nuxeo Platform may not cover all of the install use cases. In that case you must implement your own command.

To implement you own command you must extend the `AbstractCommand` class from `org.nuxeo.connect.update.task.standalone.commands:nuxeo-connect-standalone` (nuxeo-connect-standalone in https://github.com/nuxeo/nuxeo-runtime/).

Here is a simple example (the Delete command):

```
public class Delete extends AbstractCommand {

    public final static String ID = "delete";

    protected File file; // the file to restore

    protected String md5;
```

```
public Delete() {
    super(ID);
}

public Delete(File file, String md5) {
    super(ID);
    this.file = file;
    this.md5 = md5;
}

protected void doValidate(Task task, ValidationStatus status) {
    if (file == null) {
        status.addError("Invalid delete syntax: No file specified");
    }
}

protected Command doRun(Task task, Map<String, String> prefs)
        throws PackageException {
    try {
        File bak = IOUtils.backup(task.getPackage(), file);
        file.delete();
        return new Copy(bak, file, md5, false);
    } catch (Exception e) {
        throw new PackageException(
                "Failed to create backup when deleting: " + file.getName());
    }
}

public void readFrom(Element element) throws PackageException {
    String v = element.getAttribute("file");
    if (v.length() > 0) {
        FileRef ref = FileRef.newFileRef(v);
        ref.fillPatternVariables(guardVars);
        file = ref.getFile();
        guardVars.put("file", file);
        if (file.isDirectory()) {
            throw new PackageException("Cannot delete directories: "
                    + file.getName());
        }
    }
    v = element.getAttribute("md5");
    if (v.length() > 0) {
        md5 = v;
    }
}

public void writeTo(XmlWriter writer) {
    writer.start(ID);
    if (file != null) {
        writer.attr("file", file.getAbsolutePath());
    }
    if (md5 != null) {
        writer.attr("md5", md5);
    }
    writer.end();
}
```

```
}
```

You can see in that example there are four main methods to implement:

- Two are the XML serialization of the command,
- one is the command validation (should check if all required attributes are set and the command is consistent),
- and the last one is the execution itself which should return a valid rollback command.

To deploy your command you should put your class into the package (the command can be a Groovy class or a Java one). Then to invoke it just use the full class name of your command as the element name in the XML. For example if the command file is `commands/MyCommand.class` (relative to your package root) you can just use the following XML code:

```
<commands.MyCommand ... />
```

Built-in Commands

This is a list of all commands provided by Nuxeo:

Copy

Copies a file to a given destination. This command can be used to add new files or to upgrade existing files to a new version.

Usage:

```
<copy file="file_to_copy" tofile="destination"/>
```

Or

```
<copy file="file_to_copy" todir="destination_dir"/>
```

There is also a boolean `overwrite` attribute available than can be used to force command failure when overwrite is false and the destination file exists. Overwrite is by default false.

The `tofile` attribute will be injected as a File object in the EL context used by guards.

The destination can be a file pattern.

Parametrized Copy

Same as copy but the content of the copied file is generated using variable expansion based on user preferences (variables defined by the user during the install wizard).

Usage:

```
<pcopy file="file_to_copy_and_transform" tofile="destination"/>
```

Or

```
<pcopy file="file_to_copy_and_transform" todir="destination_dir"/>
```

Delete

Deletes a file.

This command takes one argument which is the absolute path of the file to delete. The argument name is `file`.

An optional parameter generated for the uninstaller is the `md5` one which will be used to avoid inconsistent uninstalls.

Directories delete are not allowed.

Usage:

```
<delete file="file_to_delete"/>
```

Deploy

Starts an OSGi bundle into Nuxeo Runtime. Needed when deploying a new bundle to Nuxeo without restarting the server. Note that not all bundles can be deployed without restarting.

This command takes one argument which is the absolute path of the bundle to deploy. The argument name is `file`.

Usage:

```
<deploy file="file_to_deploy"/>
```

Undeploy

Stops an OSGi bundle that is deployed in the Nuxeo Runtime. Needed before removing a bundle from Nuxeo without restarting the server.

This command takes one argument which is the absolute path of the bundle to undeploy. The argument name is `file`.

Usage:

```
<undeploy file="file_to_undeploy"/>
```

Reload-Core

Flushes all repository caches. Should be used when new document types are contributed and no restart is wanted.

This command takes no arguments.

The opposite command is itself.

Usage:

```
<reload-core/>
```

TODO: add more info about other existing commands

Package Web Page

Wizard Forms

Package Example

The following example is imaginary - it is not a real Nuxeo update package. It's purpose is to be an example of a complex package installable on a Nuxeo distribution.

In this example we will create a package to install Nuxeo Automation feature on a 5.3.2 version of Nuxeo DM.
What We Want to Install

Nuxeo Automation is composed of 3 Nuxeo bundles:

- nuxeo-automation-core-5.3.2.jar
- nuxeo-automation-server-5.3.2.jar
- nuxeo-automation-jsf-5.3.2.jar

and one third party library:

- mvel2-2.0.16.jar

This library is not existing on a 5.3.2 version of Nuxeo so we want to add it (not to upgrade it).

Also, for tomcat distributions we need to deploy a mail.properties file in Nuxeo configuration directory. This file contains the SMTP configuration needed by javax.mail. On JBoss we already have this configuration as a JBoss MBean. This configuration is required by the SendMail operation. The configuration file is a Java property file and contains variables that will be substituted by the values entered by the user during the install wizard.

Here is the parametrized mail.properties file we want to install:

```
mail.smtp.host=${mail.smtp.host}
mail.smtp.port=${mail.smtp.port}
mail.smtp.auth=true
mail.smtp.socketFactory.port=465
mail.smtp.socketFactory.class=javax.net.ssl.SSLSocketFactory
mail.smtp.socketFactory.fallback=false
mail.smtp.user=${mail.smtp.user}
mail.smtp.password=${mail.smtp.password}
```

The Package Structure

Here is the structure of our package:

```
nuxeo-automation-5.3.2.zip
  package.xml
  install.xml
  bundles/
    nuxeo-automation-core-5.3.2.jar
    nuxeo-automation-server-5.3.2.jar
    nuxeo-automation-jsf-5.3.2.jar
  lib/
    mvel2-2.0.16.jar
  config/
    mail.properties
  forms/
    install.xml
```

The package.xml

Here is our package manifest:

```
<package type="addon" name="nuxeo-automation" version="5.3.2">
  <title>Nuxeo Automation</title>
  <description>A service that enable building complex business logic on top of Nuxeo
services using scriptable operation chains</description>
  <vendor>Nuxeo</vendor>
  <classifier>Open Source</classifier>
  <home-page>https://doc.nuxeo.com/display/NXDOC/Content+Automation</home-page>
  <platforms>
    <platform>dm-5.3.2</platform>
    <platform>dam-5.3.2</platform>
  </platforms>
</package>
```

The Install Form

We need to define an additional page for the install wizard to ask for the properties needed to inject in the mail.properties file.

Here is the form definition we need

```
<forms>
  <form>
    <title>SMTP configuration</title>
    <description>Fill the SMTP configuration to be used by the SendMail operation. All
fields are required</description>
    <fields>
      <field name="mail.smtp.host" type="string" required="true">
        <label>Host</label>
        <value>smtp.gmail.com</value>
      </field>
      <field name="mail.smtp.port" type="integer" required="true">
        <label>Port</label>
        <value>465</value>
      </field>
      <field name="mail.smtp.user" type="string" required="true">
        <label>Username</label>
      </field>
      <field name="mail.smtp.password" type="password" required="true">
        <label>Password</label>
      </field>
    </fields>
  </form>
</forms>
```

Note that the field IDs in the form are the same as the variable keys we need to inject into the mail.properties file

The install.xml Script

Here is the content of the install.xml file

```
<install>
  <!-- copy bundles -->
  <copy file="${package.root}/bundles/nuxeo-automation-core-5.3.2.jar"
tofile="${env.bundles}"/>
  <copy file="${package.root}/bundles/nuxeo-automation-jsf-5.3.2.jar"
tofile="${env.bundles}"/>
  <copy file="${package.root}/bundles/nuxeo-automation-server-5.3.2.jar"
tofile="${env.bundles}"/>
  <!-- copy libs -->
  <copy file="${package.root}/lib/mvel2-2.0.16.jar" tofile="${env.lib}"/>
  <!-- copy the parametrized mail.properties file -->
  <pcopy file="${package.root}/config/mail.properties" tofile="${env.config}"
ignore="Platform.isJBoss()" />
  <!-- now deploy copied bundle: we don't require a server restart -->
  <deploy file="${env.bundles}/nuxeo-automation-core-5.3.2.jar" />
  <deploy file="${env.bundles}/nuxeo-automation-server-5.3.2.jar" />
  <deploy file="${env.bundles}/nuxeo-automation-jsf-5.3.2.jar" />
</install>
```

You can see the mail.properties is not installed if we are installing the package on a JBoss based distribution.

Packaging examples

Automating the packaging process allows to setup continuous integration on the Nuxeo Package build, its install and the features it provides.

There are multiple ways to build a Nuxeo Package. Focusing on those using Maven and Ant Assembly Maven Plugin, here are three different methods depending on the constraints and objectives.

From the better to the quicker method:

- Recommended method
  The recommended method is to build an NXR corresponding to the wanted result after the package install. Then we operate a comparison ("diff") between that product and a reference one (usually the Nuxeo CAP) and generate the Nuxeo Package which will perform the wanted install. That method is the better one since it will always be up-to-date in regards to the dependencies and requirements (other bundles and third-party libraries). The drawback is it takes some build time and has a dependency on a whole Nuxeo distribution.
- No-NXR method.
  That method is using the same principle for building the Nuxeo Package as for building an NXR. It is as much reliable regarding at the dependencies as the above recommended method. The drawback is since the solution is empiric, it will likely embed useless files and generate a bigger archive.
- Explicit method.
  That latest method is explicitly listing everything that must be packaged. Easy to write and very quick at build time, it requires more maintenance than the two above since you have to manually update the package assembly every time the dependencies change. You also risk not to see that an indirect dependency has changed and requires some changes on the third-party libraries. That method is not recommended except for specific cases or for a proof of concept.

See https://github.com/nuxeo/nuxeo-marketplace-sample/ for downloading a project with sample architectures, implementing the three above build methods plus the required modules for testing those Nuxeo packages with Selenium, WebDriver and Funkload.

Applied to the sample project, here are the results from those three methods.

- Recommended method

4 directories, 6 files, 128KB.

```
recommended/
    |-- install
    |   |-- artifacts-sample.properties
    |   |-- bundles
    |   |   `-- The sample project bundle.
    |   |-- templates
    |   |   `-- sample
    |   `-- test-artifacts-sample.properties
    |-- install.xml
    `-- package.xml
```

The `lib` directory is empty because all required third-parties are already included in the basic Nuxeo distribution. The `bundles` directory only contains the sample project bundle because all its dependencies are also already included in the basic distribution.

- No-NXR method.

5 directories, 150 files, 33MB.

```
nonxr/
    |-- install
    |   |-- artifacts-sample.properties
    |   |-- bundles
    |   |   `-- 46 bundles.
    |   |-- lib
    |   |   `-- 99 third-party libraries.
    |   |-- templates
    |   |   `-- sample
    |   `-- test-artifacts-sample.properties
    |-- install.xml
    `-- package.xml
```

Here, we are embedding a lot of bundles and libraries which are useless because already included in the basic Nuxeo distribution but that cannot be detected by the build process.

- Explicit method.

5 directories, 8 files, 1,6MB.

```
explicit/
    |-- install
    |   |-- bundles
    |   |   `-- The sample project bundle, explicitly listed.
    |   |-- lib
    |   |   `-- 4 third-party libraries, explicitly listed.
    |   `-- templates
    |       `-- sample
    |-- install.xml
    `-- package.xml
```

That solution builds a lighter package than the No-NXR method but we don't know if it will be missing some dependencies or not. The embedded bundles and libraries list must be manually maintained.

Nuxeo Distributions

The `nuxeo-distribution` module is used for packaging of the Nuxeo products: Nuxeo EP/DM /Jetty/Tomcat, Nuxeo Shell, Nuxeo Core Server, ... It is a way to specify declaratively which bundles and static resources are part of your application.
With `nuxeo-distribution`, you can build from Nuxeo sources, or from existing distribution and much more: if you need to assemble your own distribution, you will find in `nuxeo-distribution resources`, templates and samples on which to base your packaging.

You don't need to use the nuxeo-distribution module if:

- You want a standard Nuxeo distribution:
  => Download it from http://www.nuxeo.com/downloads/ (manual download only);
  => Download it from http://maven.nuxeo.org (manually via online interface or automatically using Maven);
- You want to customize configuration files:
  => Use the template configuration system;
- You want to build your own distribution:
  => Rely on the same tools and principles as nuxeo-distribution does but do it from your own project, with your own assembly.

You have to use `nuxeo-distribution` module if:

- You want to reproduce the Nuxeo build process,
- You want to build Nuxeo offline,
  => Being unable to download artifacts from internet, you will need a lot of other Nuxeo sources and some third-party artifacts.
- You work on Nuxeo source code and need quick feedback on your changes, you don't want to wait for our continuous integration system building nuxeo-distribution.

Read to the Nuxeo Core Developer Guide for more information on how to package from sources using nuxeo-distribution.

Available installers

By default, Nuxeo distributions are packaged as ZIP files.

Some Nuxeo distributions are also packaged with automated installers, in order to ease installation and follow the targeted OS standards, by respecting the usual directory organization for instance, creating desktop shortcuts and menu items, tweaking environment properties, help installing optional third-parties, ...
The available installers can be:

- multi-OS installer (.jar, .exe, .app, .jnlp — beta),
- Windows installer (.exe),
- Linux Debian installer (.deb),
- Mac OS X disk image/application/package (.dmg/.app/.pkg — deprecated).

Any help about OS-specific cases, recommendations, contributions or feedbacks is very welcome. See https://jira.nuxeo.com/browse/NXBT for issue management.

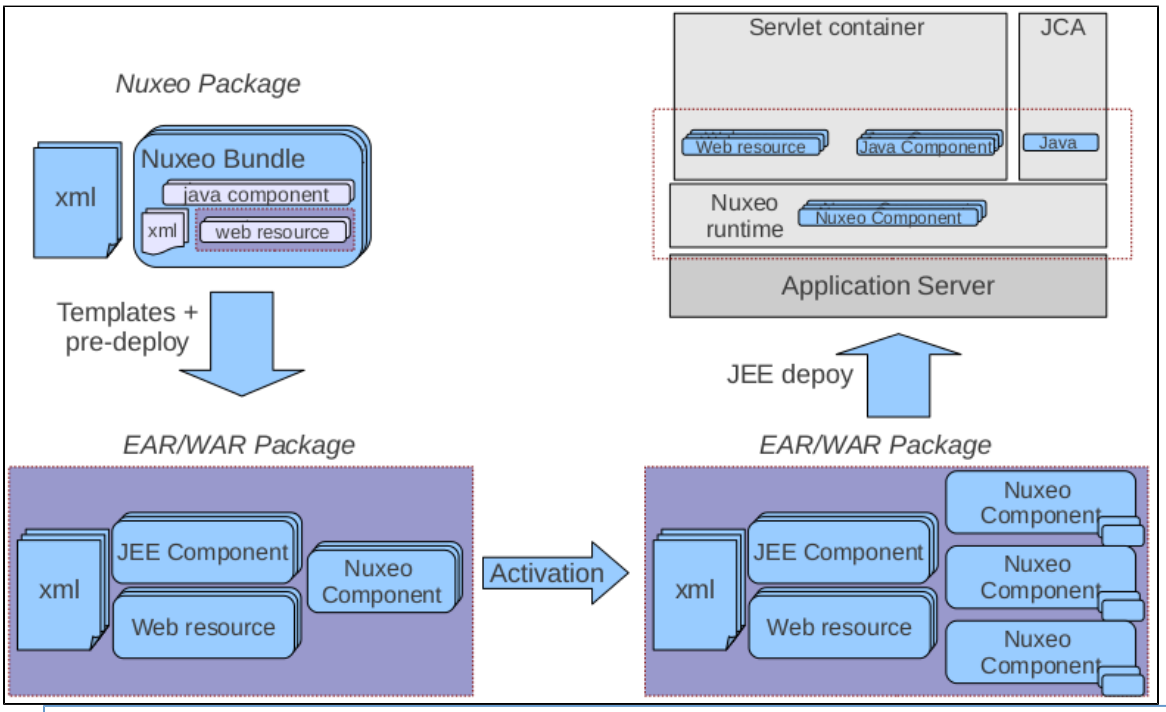
Nuxeo Deployment Model

Deployment Phases

The Nuxeo Platform deployment is incremental: the startup process involves different processors for different phases.

1. Template processor for configuration
2. Deployment-fragment pre-processor
3. Bundle activation and deployment

4. WAR/EAR deployment



In this section

Template Processor

The template system allows to use template for generating configuration files:

- data source declaration
- JCA connector declaration
- SMTP Gateway
- monitoring extensions
- misc extension point contributions
  (LDAP, SMTP, OpenOffice.org)

The template processor system uses Java property files to read the variable and do the replacement in the template to generate the actual configuration files.

The template processor system contains a profile system so that a given server can quickly be reconfigured for a target environment:

- Dev profile
- Integration profile
- Production profile
- …

The template system uses Freemarker so that template can contain simple conditional processing.

```
...
<extension target="org.nuxeo.ecm.core.repository.RepositoryService"
   point="repository">
    <repository name="default"
    factory="org.nuxeo.ecm.core.storage.sql.ra.PoolingRepositoryFactory">
      <repository name="default">
        <pool minPoolSize="${nuxeo.vcs["min-pool-size"]}"
maxPoolSize="${nuxeo.vcs["max-pool-size"]}"
          blockingTimeoutMillis="100" idleTimeoutMinutes="10" />
<#if "${nuxeo.core.binarymanager}" != "" >
        <binaryManager class="${nuxeo.core.binarymanager}" />
</#if>
        <clustering enabled="${repository.clustering.enabled}"
delay="${repository.clustering.delay}" />
        <binaryStore path="${repository.binary.store}" />
...
```

Deployment Fragment Preprocessor

In Nuxeo, the target web application is in fact created from a lot of separated bundles.

For that each bundle can contribute :

- resources to the WAR
- declaration in the web.xml
- declaration in the faces-config.xml
- Java property files for i18n
- …

Because in JEE5 there is no standard way to do that, we use a pre-deployment processor that will process the bundles for deployment-fragment.xml files.

The deployment fragment contains ANT like commands that will be executed in order to contribute bundle resources to the JEE WAR Archive.

```
<extension target="pages#PAGES">
    <!-- Bind url to start the download -->
    <page view-id="/nxconnectDownload.xhtml"
      action="#{externalLinkManager.startDownload()}" />
  </extension>

  <extension target="faces-config#NAVIGATION">
    <navigation-case>
      <from-outcome>view_admin</from-outcome>
      <to-view-id>/view_admin.xhtml</to-view-id>
      <redirect />
    </navigation-case>
  </extension>

  <extension target="web#STD-AUTH-FILTER">
    <filter-mapping>
      <filter-name>NuxeoAuthenticationFilter</filter-name>
      <url-pattern>/nxadmin/*</url-pattern>
      <dispatcher>REQUEST</dispatcher>
      <dispatcher>FORWARD</dispatcher>
    </filter-mapping>
  </extension>
```

Bundle Deployment

This phase is the real deployment “à la OSGi” :

- activate bundles
- declare components, services and extension points
- resolve Extension Point contributions

Standard WAR/EAR Deployment

The standard WAR deployment is managed by the host application server that will handle:

- Web resource declaration
  (using the aggregated descriptor generated by the pre-deployment)
- JSF initialization
- Seam Init

NuxeoCtl

NuxeoCtl is not really part of the deployment, but it's a central tool that helps managing Nuxeo Startup.

NuxeoCtl provides

- a Nuxeo Bootstrap
  - runs template system
  - starts the target Application Server
- some administration tools
  - Marketplace package administration and installation
  - start/stop/restart/configure …
- a simple command GUI

NuxeoCtl, like the Templating System, is not really needed to be able to run Nuxeo. It just helps having a simple and efficient configuration.

It will be more and more true as we continue integrating features inside NuxeoCtl :

- multi-node commands (like update package on each node)
- cloud commands

In a sense, NuxeoCtl is close to what is provided in several “Cloud packaged tomcats” (TcServer, CloudFoundry …).

Existing Deployment Targets

Nuxeo Platform currently supports several deployment targets.

	Testing (Junit)	**Custom Tomcat/JBoss (dynamic mode)**	**Tomcat WAR (static mode)**	**Jboss EAR (static mode)**
NuxeoCtl	Not used	Yes	No	No
Config templating	Not used	Yes	Run once (before creating the WAR)	Run once (before creating the EAR)
Pre-deployment	Not used	Started by custom deployer	Run once (before creating the WAR)	Run once (before creating the EAR)
Bundle activation	Yes via Junit	Started by custom deployer	Started by Servlet listener	Started by Jboss EAR listener
Standard deployment	Not used	Yes	Yes	Yes
Full deployment	No JSF / WAR	Yes	Yes	Yes
Marketplace feature	N/A	Yes	No	No

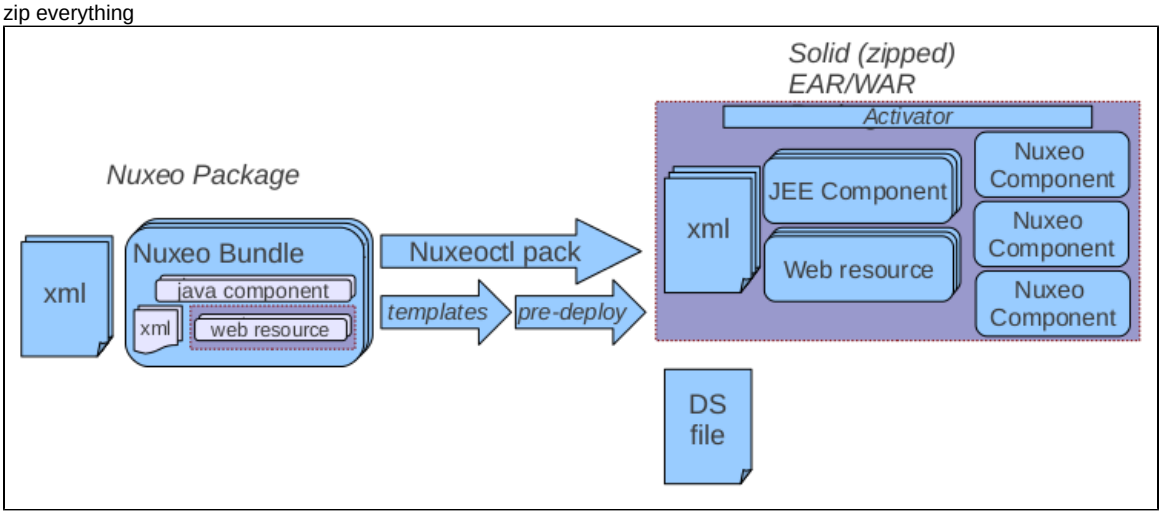
Depending on the target platform:

- all deployment phases may not be run
- platform features may change

The static deployment model was added initially for JBoss and was then extended to Tomcat too.

In the static deployment model NuxeoCtl pack command is run to:

- run the template system
- run the pre-processing
- reorganize the WAR/EAR structure
- add a activator to start the Bundle deployment
- zip everything



Advanced topics

Table of Contents:

Integrating with JPA

The following paragraphs explain the specific part of integrating a nuxeo service with JPA.

Let say we want to put in place a document rating service that persist information using JPA.
For this, we need these three modules :

- nuxeo-samples-persistence-api
- nuxeo-samples-persistence-core
- nuxeo-samples-persistence-facade

The API would be something like

```
interface DocumentRating {
DocumentRating createRating(DocumentModel doc, String name, short maxRating);
DocumentRating getRating(DocumentModel doc, String name);
DocumentRating saveRating(DocumentMOdel doc, String name, int rating);
}
```

with an entity bean

```
...
@Entity(name="DocRating"
class DocumentRating {

   @Id
   @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="DOCRATING_SEQ")
   @SequenceGenerator(name="DOCRATING_SEQ", sequenceName="DOCRATING_SEQ",
allocationSize=100)
   @Column(name = "RATING_ID", nullable = false, columnDefinition = "integer")
   private long id;

   String docUUID;

   short rating;

   short maxRating;

   long created;

   long modified;
}
...
```

Nuxeo's data providers should support deployment inside or outside an EJB container. The following describe what should be achieved for each use case.

Outside an EJB container

In that case, the framework resolves the service as the document rating provider.
A persistence provider is lazy obtained using nuxeo persistence core services and used for getting access to the entity manager.

```
class DocumentRatingProvider implements DocumentRating {

protected PersistenceProvider persistenceProvider;

public PersistenceProvider persistenceProvider() {
if (persistenceProvider == null) {
     persistenceProvider =
Framework.getService(PersistenceProviderFactory.class).newProvider("doc-rating");
    }
return persistenceProvider;
}

...
public DocumentRating saveRating(DocumentModel doc, String name, short rating) {
persistenceProvider.run(true, new RunCallback<DocumentRating>() {
public DocumentRating runWith(EntityManager em) {
         return this.saveRating(em, doc, name, rating);
      }
});
}

public DocumentRating saveRating(EntityManager em, DocumentModel doc, String name,
short rating) {
    ...
  }
...
}
```

A minimal configuration should named the persistence unit and specify the data source to be used.

```
...
  <extension target="org.nuxeo.ecm.core.persistence.PersistenceComponent"
        point="hibernate">
        <hibernateConfiguration name="doc-rating">
            <datasource>doc-rating</datasource>
            <properties>
               <property name="hibernate.hbm2ddl.auto">update</property>
            </properties>
        </hibernateConfiguration>
  </extension
...
```

Inside an EJB container

In that case, the framework is resolving the service using the document rating bean. The persistence unit is initialized by the container himself, and an entity manager is automatically injected into the bean. The entity manager is transmitted by the bean to the core provider using a call parameter (thread safe).

```
@Stateless
@Local(DocumentRatingLocal)
@Remote(DocumentRating)
class DocumentRatingBean implements DocumentRating {

@PersistenceContext(unitName = "doc-rating")
private EntityManager em;

...

public DocumentRating saveRating(DocumentModel doc, String name, short rating) {
    DocumentRatingProvider provider =
Framework.getLocalService(DocumentRatingProvider)
    return provider.saveRating(em, doc, name, rating);
  }

...
```

The persistence unit should be registered in the JPA container by providing a `META-INF/persistence.xml` descriptor in the core module.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
        http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
 version="1.0">

 <persistence-unit name="doc-rating">
  <jta-data-source>java:/doc-rating</jta-data-source>
  <class>...DocumentRating</class>
  <properties>
   <property name="hibernate.hbm2ddl.auto" value="update" />
  </properties>
 </persistence-unit>
</persistence>
```

Adding an Antivirus

This is a recurrent demand from our customers. So here is a quick guide to add an antivirus scanner when uploading blobs in Nuxeo documents.

In this section

Main Guidelines

One way to implement an antivirus scan for uploaded documents without any significant performance hit at creation time would be to:

1. implement a new `QuarantineBinaryManager` that would wrap a two instances of `BinaryManager` configured to use two distinct repositories, e.g. `repo-cleared` and `repo-quarantine;`
2. introduce a new dynamic facet with a schema that can store antivirus status and metadata of all the blobs stored on the document.

Whenever a new blob is uploaded and attached to a Nuxeo document the `QuarantineBinaryManager` would first delegate the insertion to the `repo-quarantine` instance of `BinaryManager`.

A new Nuxeo synchronous core event listener would also react to the `aboutToCreate` or `beforeDocumentModification` event and introspect whether one of the blob fields is dirty. If so, the dynamic facet of the document would be updated to mark the new blob as being quarantined for antivirus analysis and a new asynchronous task would be scheduled using the `WorkManager` that would delegate a call to the antivirus service out of transaction and then collect the outcome of the antivirus as follows:

- If the antivirus outcome is negative (no virus detected): the `WorkManager` task would call a new public method of `QuarantineBinaryManager`, for instance named `QuarantineBinaryManager#giveClearance(String blobDigest)`. This method would physically move the blob from the `repo-quarantine` bucket to the `repo-cleared`. The `WorkManager` task would also update the document dynamic facet to inform the user (e.g. with a dedicated blob widget) that the document does not contain a suspect blob.
- If the antivirus outcome is positive (a virus is detected in the attached file): the `WorkManager` task would not call the `giveClearance` method and instead just update the metadata fields of the dynamic facet schema to inform the user of the outcome of the analysis. The user could then decide to delete the contaminated blob attachment (or the system could be configured to do it automatically).

Permissions

Furthermore it would be very useful to make the event listener manage a new local ACL that would render documents with blobs in quarantine only visible to the user who uploaded the last blob until it is moved out of quarantine or deleted. This feature would have the following purposes:

- Never propagate a contaminated blob to other users by denying access to the documents that contain contaminated files.
- Do not disrupt too much any existing Nuxeo components (e.g. Nuxeo Drive [1]) that usually expect any uploaded blob in a document to be immediately available.
- Make it possible for the uploader to introspect the state of the virus analysis by making a custom blob widget.

The management of the dynamic facet, the ACL and the call to the `giveClearance` method should be wrapped in a single `AntivirusVirusAware` document adapter to abstract away all those operations in a simple and clean public API.

Implementing such extensions to the Nuxeo platform is possible but might not be easy for non-core Nuxeo developer.

1: Such an ACL might still make updated document temporarily look as if deleted to other Nuxeo Drive users while the antivirus analysis is taking place.

Dev Cookbook

Welcome to the Nuxeo Developer cookbook!

This cookbook is intended to Java developers who are starting developing on the Nuxeo platform. It provides simple development recipes to do these customizations to help them understanding the technical basis to customize a Nuxeo application.

Developers are expected to be familiar with Maven.

Recipes are intended to be independent: you can try any recipe at any time.

Our recipes:

- Quick Start Dev Guide
- How to create an empty bundle
- How to implement an Action
- How to Contribute a Simple Configuration in Nuxeo
- How to configure document types, actions and automation chains
- How to setup a test SMTP server
- Implementing local groups or roles using computed groups
- How to track the performances of your platform
- Workflow How-Tos
- Layouts and Widgets How-tos
- Localization and Translation How-Tos
- Actions and Filters How-tos
- Theme and Style How-Tos
- Special Pages Customization How-Tos
- How to Change the Default Document Type When Importing a File in the Nuxeo Platform?

Our blog tutorials:

Difficulty	Title
Intermediate	Creating a Video Export Workflow: End Workflow Automatically
Intermediate	Creating a Video Export Workflow: Display the Exported Videos

Intermediate	Creating a Video Export Workflow: Writing Export Code
Beginner	Filling a Suggestion Widget Using a WebService, A New Example
Intermediate	Creating a Video Export Workflow – Part 1
Beginner	How to Search for Documents That Have an Empty List Property
Beginner	Build A Suggestion Widget Using a Custom Directory Filled By a Web Service
Beginner	Using QR Codes with the Nuxeo Platform
Intermediate	Export Your Nuxeo Tree Structure to Your File System
Beginner	5 Ways to Get a Nuxeo Platform CoreSession
Advanced	A Spreadsheet Editor for Lists of Documents in the Nuxeo Platform
Intermediate	Monitoring Nuxeo Docker Container Logs with Logstash, Elasticsearch and Kibana
Beginner	How to Set Up Notifications When Video Conversions Are Finished
Beginner	A Content App in JavaScript Using Mustache, Bootstrap and nuxeo.js
Beginner	[Q&A Thursday] Configuring Automatic Video Conversions on The Nuxeo Platform
Beginner	CoreOS Monitoring with Diamond and Graphite
Beginner	How to Manage Dates in Automation Chains
Beginner	[Q&A Friday] How to Write a JSF Validator for a Nuxeo Studio Widget Field
Beginner	Export Data with Content Automation
Beginner	Authenticating to the Nuxeo Platform with OAuth.io
Beginner	[Q&A Friday] How to Manage Users with the REST API
Beginner	[Q&A Friday] How to Download Files Attached to Documents Using the REST API
Beginner	[Q&A Friday] How to Upload Files and Bind Them to Documents Using the REST API
Beginner	Using Docker Containers – Part 1 – Build a Full Fledged Nuxeo Image
Beginner	An Event Introspection with Nuxeo Studio
Beginner	[Q&A Friday] How to Remove Tabs, Buttons, Links Under a Specific Path
Beginner	[Q&A Friday] How to Create a Nuxeo Studio Defined Document on File Drag and Drop
Beginner	[Q&A Friday] How to get logger from within a MVEL script in Nuxeo Studio?
Beginner	Creating a Thumbnail Browsing Module with AngularJS and the REST API – Part 3
Beginner	Creating a Thumbnail Browsing Module with AngularJS and the REST API – Part 2
Beginner	Creating a Thumbnail Browsing Module with AngularJS and the REST API – Part 1
Beginner	[Q&A Friday] How to Add JavaScript or CSS Resources to Your Pages

Beginner	[Q&A Friday] How to Retrieve a Video Storyboard Through the REST API
Beginner	[Q&A Friday] How to Create a Group on Domain Creation
Intermediate	[Monday Dev Heaven] Pick a Random Picture as Background for the Login Page
Beginner	Nuxeo Studio Rocks – Part 1 – Queues (aka Smart Folders)
Beginner	Write an Operation to Search through a Directory
Beginner	[Q&A Friday] How to add two doubles with Content Automation
Beginner	[Q&A Friday] How to Attach Files to Documents with REST API
Intermediate	First Steps with AngularJS and Nuxeo Content Automation
Intermediate	Exploring Audit Tables and Nuxeo VCS
Advanced	Extend Nuxeo Drive Series #4 – Customize the Nuxeo Drive hierarchy
Intermediate	Extend Nuxeo Drive Series #3 – How to synchronize a document without attached binaries
Beginner	Extend Nuxeo Drive Series #2 – Override existing adapter parameters
Beginner	Extend Nuxeo Drive Series #1 – Override Operations
Intermediate	[Q&A Friday] How to add tagging capability during drag'n'drop
Beginner	[Monday Dev Heaven] Nuxeo and Atlassian HipChat Integration
Beginner	[Q&A Friday] How to Access a Resource Bundle with MVEL in Content Automation
Beginner	[Q&A Friday] Remotely Searching for a Document Using Tags
Beginner	[Q&A Friday] How to Retrieve the List of All Possible Subject Values Using Content Automation
Beginner	[Q&A Friday] How to Retrieve Custom Metadata with Content Automation
Intermediate	[Q&A Friday] Automatic Creation of User Workspaces
Intermediate	[Q&A Friday] How Do We Search for Accented Characters in Note Content?
Intermediate	[Q&A Friday] Can an external web service call be included in an automation chain?
Beginner	[Q&A Friday] Relative date usage in Nuxeo Studio
Beginner	[Monday Dev Heaven] Trying IntelliJ IDEA and Nuxeo Platform
Intermediate	[Q&A Friday] How to deploy a contribution on a per method basis?
Beginner	[Q&A Friday] How to add extra files to a document using Content Automation
Intermediate	[Monday Dev Heaven] Creating DeckJS Slides in Markdown
Beginner	[Q&A Friday] Document Expiration Notification
Advanced	[Monday Dev Heaven] Extract Metadata from Content File Attachments #2
Intermediate	[Q&A Friday] Enabling Preview For Document Attachments
Intermediate	[Q&A Friday] Get Parent ID from Child Label of a Nuxeo Vocabulary

Advanced	[Monday Dev Heaven] Extract Metadata from Content File Attachments
Advanced	[Q&A Friday] How to view emails stored in Nuxeo via WebDAV
Advanced	[Q&A Friday] How can you know which document properties have been modified?
Intermediate	[Monday Dev Heaven] How to search for PDF files, or any other type for that matter
Beginner	[Q&A Friday] Does Nuxeo Studio support definition of new document types?
Intermediate	[Q&A Friday] How do we upgrade existing domains with new changes to the structural template?
Intermediate	[Monday Dev Heaven] Unleash Nuxeo Shell
Intermediate	[Q&A Friday] SELECT Clause in Nuxeo NXQL Queries – Part 2
Intermediate	[Q&A Friday] SELECT clause in Nuxeo NXQL queries
Intermediate	[Q&A Friday] Is it possible to call a Seam bean from JAX-RS/WebEngine?
Intermediate	[Q&A Friday] How can I have a dashboard that cannot be modifed by users?
Advanced	[Monday Dev Heaven] Nuxeo-lang-ext-assistant, a WebEngine site here to help you translate Nuxeo
Beginner	[Q&A Friday] Choose the encoding charset for zip filenames
Beginner	[Q&A Friday] How to run Nuxeo integration tests
Advanced	[Monday Dev Heaven] Highlight searched keyword in document preview
Beginner	[Q&A Friday] Overriding templates or extension point in Nuxeo
Advanced	[Monday Dev Heaven] How to manage external assets in Nuxeo, first try with oEmbed
Beginner	[Q&A Friday] How can I replace a column in a document listing?
Beginner	[Q&A Friday] Translating Nuxeo
Advanced	[Monday Dev Heaven] Playing with OpenLayers to add GeoLocation to Nuxeo
Beginner	[Q&A Friday] Is the Tomcat Manager Supported?
Advanced	[Q&A Friday] How to add a ‘Loading, Please Wait…” message on the preview panel?
Advanced	[Monday Dev Heaven] Adding thumbnail preview of document content in Nuxeo
Advanced	[Q&A Friday] How to get thumbnails for PDF or PSD documents?
Advanced	[Monday Dev Heaven] Navigating in Nuxeo documents through a graph
Advanced	[Q&A Friday] How binaries are physically stored in the filesystem
Advanced	[Monday Dev Heaven] Add a Forgotten Password Functionality to Nuxeo, Part 2/2
Intermediate	[Q&A Friday] How do I import a document in Nuxeo with version 12 as initial version?
Advanced	[Monday Dev Heaven] Add a Forgotten Password Functionality to Nuxeo, Part 1/2

Beginner	[Q&A Friday] How to Clear UI Selected Documents in Studio
Advanced	[Monday Dev Heaven] Create Your Own Documents on File Drag and Drop
Beginner	[Q&A Friday] How does the Seam.PushDocument operation work?
Beginner	[Q&A Friday] Will Nuxeo upgrade its technical Java Stack?
Intermediate	[Monday Dev Heaven] How to add an HTML preview for iWork Pages files
Beginner	[Q&A Friday] Nuxeo WCM, blogs, wikis.
Advanced	[Monday Dev Heaven] Automatic document creation in Nuxeo, Part 2
Intermediate	[Q&A Friday] Is it possible to change Nuxeo Platform footer links with Studio?
Advanced	[Monday Dev Heaven] Automatic document creation in Nuxeo
Intermediate	[Q&A Friday] Dive into Nuxeo IDE and SDK internals
Intermediate	[Monday Dev Heaven] Use Your Nuxeo OpenSocial Gadgets in a Liferay Portal
Beginner	[Q&A Friday] How to manage local environment properties in Nuxeo
Advanced	[Monday Dev Heaven] How to unit test Nuxeo WebEngine modules
Intermediate	[Q&A Friday] Notify users about content and document deletion
Advanced	[Monday Dev Heaven] Multi-threaded, transactional bulk import with Nuxeo
Intermediate	[Q&A Friday] How to set up multi-level virtual navigation in Nuxeo
Advanced	[Monday dev heaven] Sharing with the outside world #2: getting your readers' feedback
Intermediate	[Q&A Friday] How to Test Operations?
Advanced	[Monday Dev Heaven] Share your content with the outside world
Advanced	[Q&A Friday] Embedding Nuxeo?
Beginner	[Monday Dev Heaven] Continuous Integration at Nuxeo
Advanced	[Q&A Friday] XMP Metadata Support in Nuxeo?
Advanced	[Monday Dev Heaven] Connecting Nuxeo to OAuth Authenticated APIs: Yammer as Example
Intermediate	[Q&A Friday] How do I authenticate my external application with Nuxeo?

Quick Start Dev Guide

Welcome to the Nuxeo Developer Quick Start Guide!

This guide aims at showing you how to start a Nuxeo project with a simple example using our tools Nuxeo Studio and Nuxeo IDE. The most common process when people want to customize the Nuxeo Platform consists in doing the configuration in Nuxeo Studio and creating new features (customizations that go beyond XML configuration and require actual coding) in Nuxeo IDE. This Quick Start Guide will guide you through the different steps of this process.

Our example will use a template in Nuxeo Studio, to quickly have a working set of customization in Nuxeo Studio (Studio customization is not the purpose here). This template, called Custom Doc ID Generation, provides a new document type ("Sample") for which an automation chain is run when the document is created in order to generate an incremental UID. We will add a new metadata that will be filled in with a random value upon creation as well. To generate this random string, we will use an operation that will be coded in Nuxeo IDE. And finally we will integrate this new operation in our Studio project.

The purpose of this guide is not to provide an overview of all the features of Nuxeo Studio or Nuxeo IDE but to show you how to use these two tools to make your own Nuxeo project. However, you can take a look at the Studio documentation or IDE documentation at anytime if you want more information about a feature.

In order to follow / complete this example, you will need:

- a Nuxeo Studio account (trial or regular),
- Eclipse, of course,
- Nuxeo IDE (see Getting Started with Nuxeo IDE).

Now let's start your project with Nuxeo Studio.

Starting your project with Nuxeo Studio

Prerequisites
To complete the steps described in this section, you need: • a Nuxeo Studio account (trial or regular), • an empty Studio project, • a Nuxeo Platform 5.6 instance, that has been registered on Nuxeo Connect.

In this section
1 Quick start your Nuxeo project using a template 2 Edit the template 2.1 Add a new metadata 2.2 Edit the automation chain

In this section, we will start customizing the Nuxeo Platform. The first customizations occur in Nuxeo Studio, where you can easily create new doc ument types, content views, etc. Here, we will use an application template to quickly have a working set of customizations. Then we will add some more customization to this basis.

Quick start your Nuxeo project using a template

We will use the Custom Doc ID Generation template.

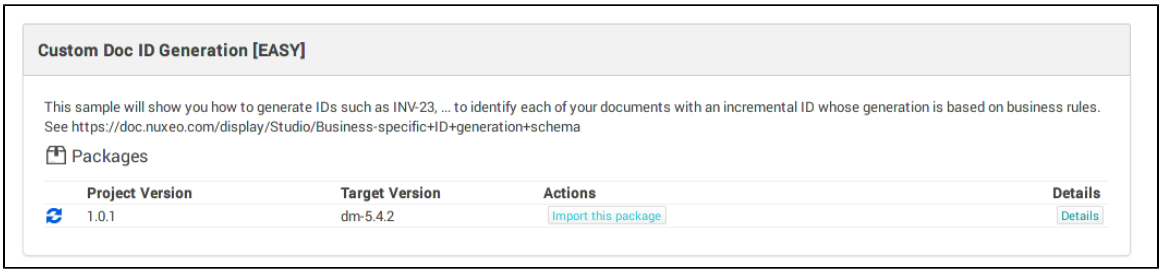
Customization set of the application template

- a new document type called "Sample", with:
  - an associated schema holding one metadata "type",
  - a new view, creation and header layouts
- an automation chain called "UIDUpdateChain"
- an event listener called "UIDUpdateListener"
- two vocabularies, called "TypeLabelVocabulary" and "TypePrefixVocabulary".

More information about this application template in the Studio documentation.

To import the Custom Doc ID Generation template:

1. In Nuxeo Studio, in the **Settings and versioning** menu, click on **Application templates**.
2. Click on the **Import this package** link of the template "Custom Doc ID Generation".
3. On the confirmation popup window, click on OK to reload your project with the application template.
   You now have the elements of the template in your project.



You should now test that the template customization works nicely on your Nuxeo instance.

1. Update your Nuxeo instance with your Studio configuration.
2. Create a "Sample" document.
3. Check that it has the defined layouts and that it has a UID generated.

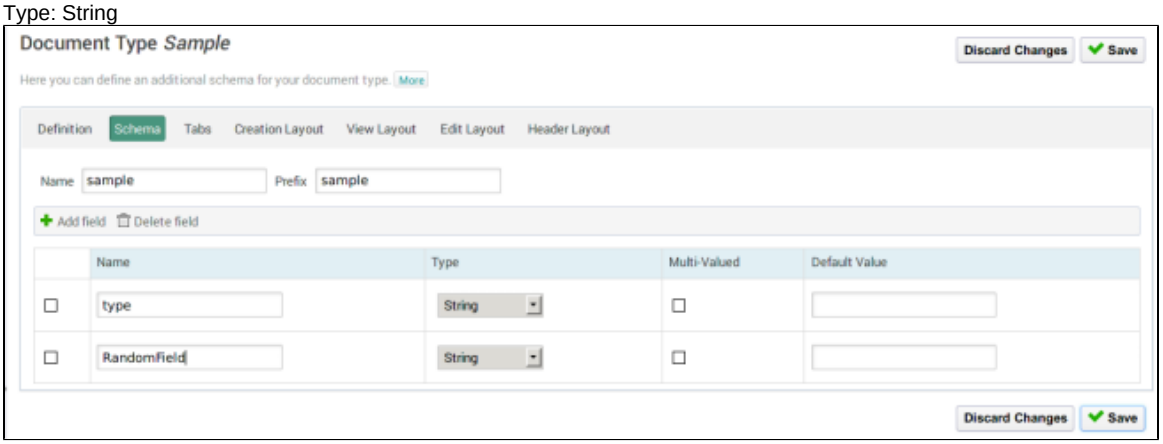
Edit the template

Add a new metadata

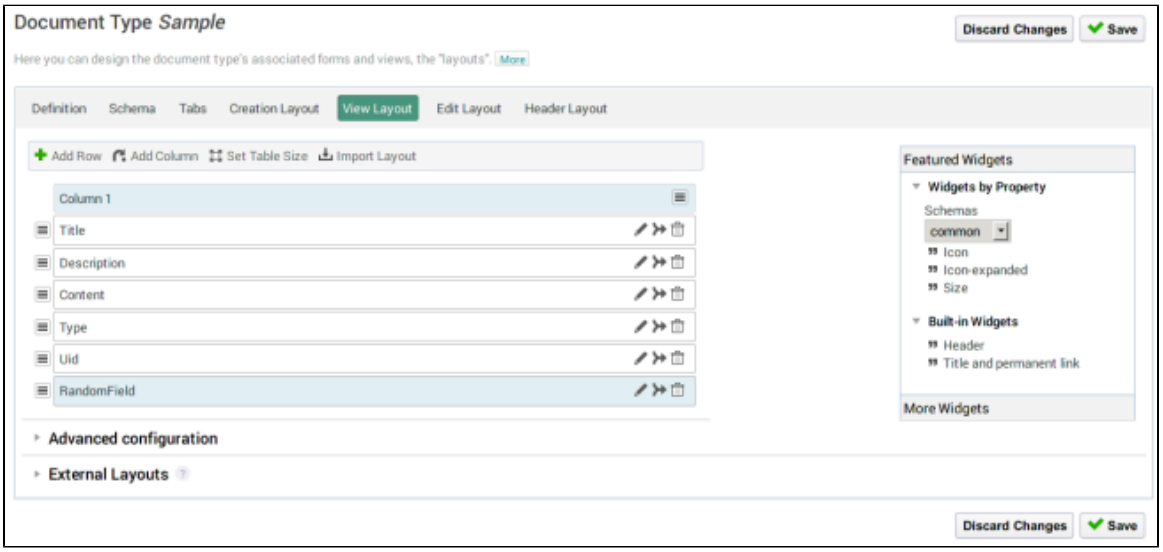
First, we'll add a new metadata for the "Sample" document type.

To do that, let's edit the "Sample" document type:

1. In the Schema tab, add a new field with the properties below:
   - Name: `RandomField`
   - Type: String

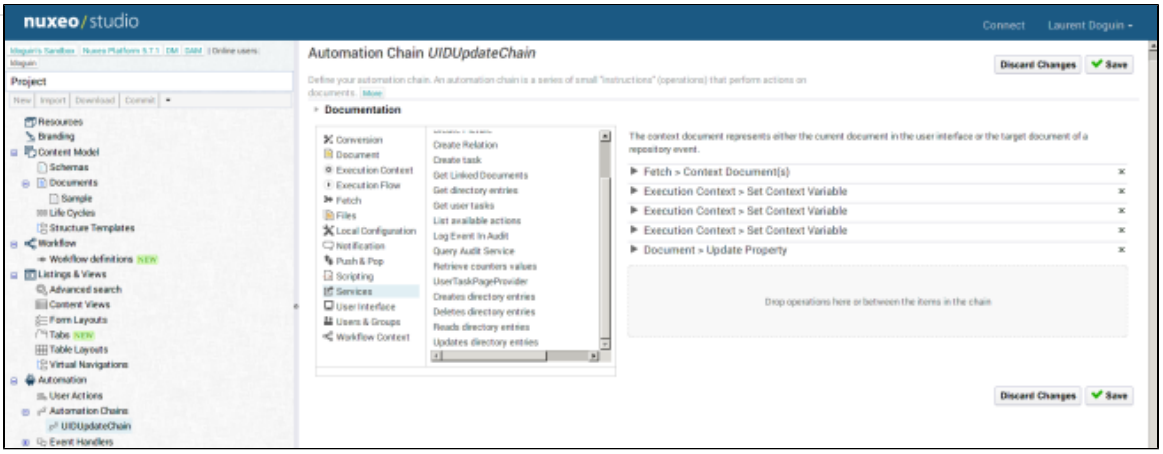


2. Edit the View layout to add the `RandomField` field in a new row at the end of the layout. Leave the default properties.



Edit the automation chain

Now we want to fill the `RandomField` metadata with a random alphanumeric generated string. Unfortunately there is no operation that will let us do that. But fear not, we're going to make our own. This will require using Nuxeo IDE.



Installing Nuxeo IDE

Nuxeo IDE, for Integrated Development Environment, is an Eclipse plugin that enables Java developers to customize Nuxeo applications. It provides an environment and features adapted to Nuxeo's projects. Java developers can work on Nuxeo applications in their familiar environment and they have Nuxeo-related features directly available. So both Nuxeo experts and developers new to Nuxeo can use it to customize Nuxeo applications easily.

In this section

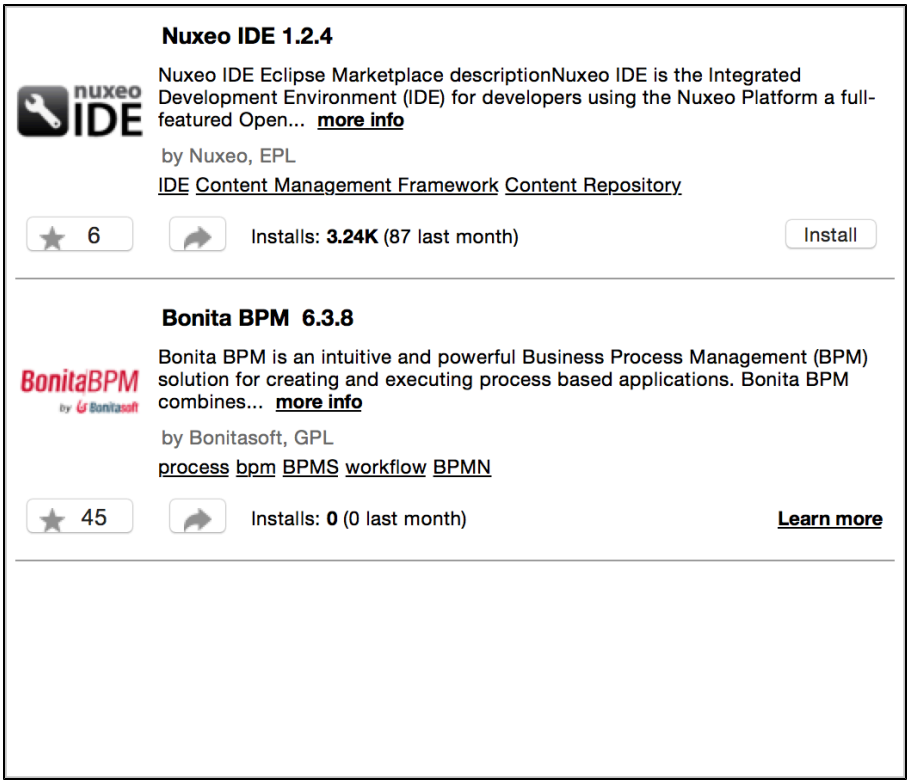
Installing Nuxeo IDE

Prerequisites

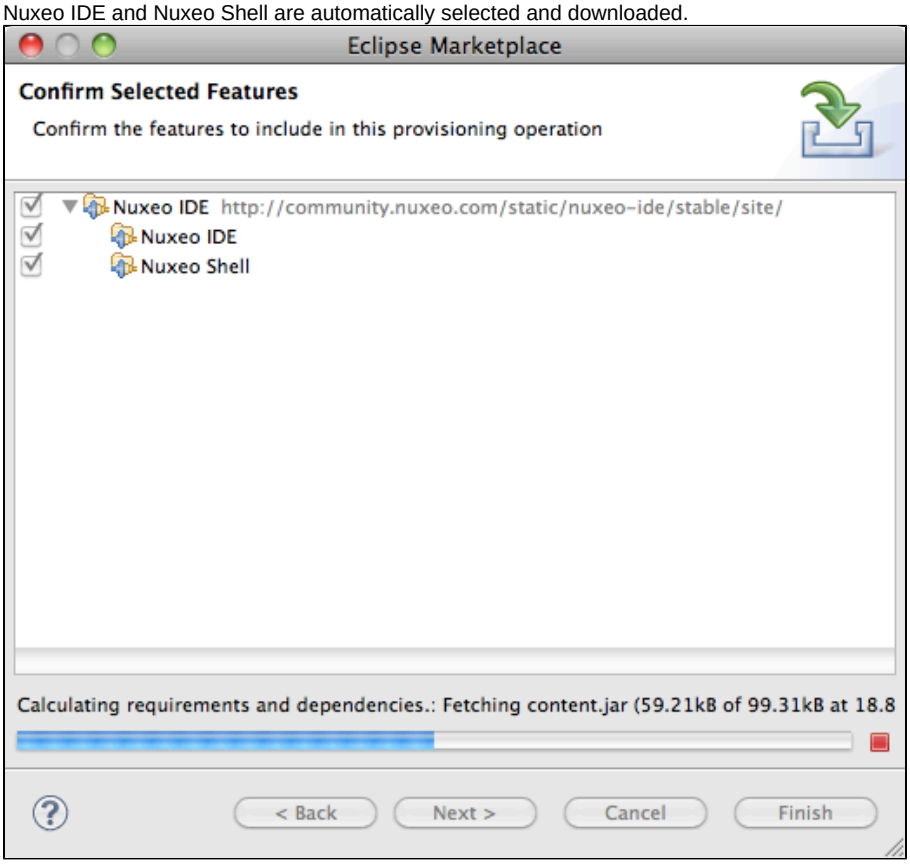
- Eclipse 4.3 (Kepler) ,
- Java 7,
- A Tomcat based distribution of Nuxeo Platform 5.8 or later.

This installation guide assumes that you have already installed some flavor of a supported version of Eclipse. If not, then you can download Eclipse from http://www.eclipse.org/downloads.

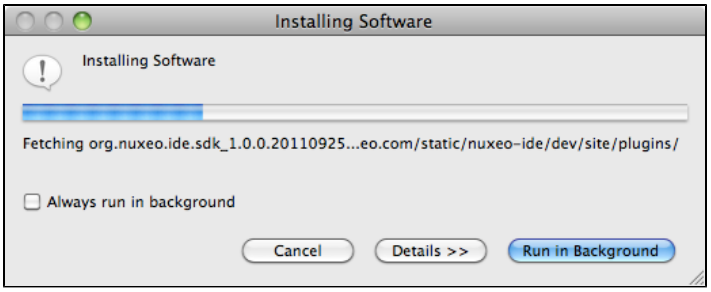
1. In Eclipse, go into the **Help**, **Eclipse Marketplace** menu.
2. The Eclipse Marketplace window opens.
3. Search for **Nuxeo**, select **Nuxeo IDE** and click on the **Install** button.



4. Nuxeo IDE and Nuxeo Shell are automatically selected and downloaded.



5. When Nuxeo IDE and Nuxeo Shell are downloaded, click on the **Next** button.
6. Accept license when prompted.
   Installation begins. After a few seconds, a security warning is prompted.
7. On the security warning window, click on **OK**.
   Installation continues.



8. Restart Eclipse when prompted.
   Nuxeo IDE is installed. A new button is available in the Eclipse toolbar:

Alternative
You can also install Nuxeo IDE from Eclipse directly.

Installing a Nuxeo DSK

A Nuxeo SDK is a regular Tomcat distribution of the Nuxeo Platform that is used as the development server, where your projects will be deployed. The Nuxeo SDK is providing the classpath for your Nuxeo projects. All the JARs on the server will be visible in your project and you will be able to reference them from your code. Only Nuxeo Platform 5.5 versions and later are supported.

To install a Nuxeo SDK:

1. UNzip the downloaded Nuxeo Platform distribution.
2. In the **Nuxeo SDK** preference page (Eclipse Menu -> Window -> Preferences -> Nuxeo -> Nuxeo SDK), click on **Add...**.
3. Select the directory where you unzipped your Nuxeo distribution.
   Be sure to select the right directory: the one containing the `bin` and `nxserver` directories.
4. **Check** the added Nuxeo SDK and click on the **OK** button to finish.
   Checking the box at the left of the SDK is required to mark it as the current Nuxeo SDK used in the Eclipse Workspace.

Alternative
You can also checkout Nuxeo source code and build a Tomcat distribution from the Nuxeo trunk. In that case, you must activate the `sdk` Maven profile: `mvn package -Ptomcat,sdk`.

Getting familiar with Nuxeo IDE

Pre-requisites
• Having installed Nuxeo IDE • Having installed a Nuxeo SDK • Being familiar with Eclipse

In this section

• The Nuxeo Perspective
  • Nuxeo Server
  • Nuxeo Components
  • Nuxeo Studio
  • Nuxeo Shell

Nuxeo IDE is an Eclipse plugin that adds some productivity development features. Here's a small tour of the basic feature. At the end of this tutorial, you should now everything there is to know about Nuxeo IDE.

The Nuxeo Perspective

We've added a new perspective simply called Nuxeo. When you open this perspective, you have access to different new tabs.

Nuxeo Server

Let's start by looking at the Nuxeo Server tab . It gives you the ability to start , stop , start in debug mode . As you can see when

starting Nuxeo, the tab starts displaying the server log. There are two associated buttons to Lock the scrolling and to Clear the console. Once you see the "Server started" line in the logs, you can click on the Open Nuxeo In Web Browser button. We'll talk about the other two (hot reload and deployment profile) later.

Nuxeo Components

The Components tab gives you an overview of the different Components available in the SDK. What's a component you might ask? It's an XML file declaring a Service, an Extension Point (XP) or a contribution to an XP. This is what makes Nuxeo easily extensible. Basically a Service will provide some business logic that can be modified or extend using XPs. The service knows how to handle and register contribution to XP. Here's an example. In Nuxeo there is a service that handles Document Type. It knows how to handle several XPs. One of them is used to register new Document Type for Nuxeo. You can find out more on our component model in our documentation.

Nuxeo Studio

The Studio tab lists the different Studio project you have access to. You can browse their content to see what configuration has been added to the project. Click on a feature and you'll be sent directly to the corresponding Studio tab. Notice the two icons on the upper right corner. One is used to refresh the list and the other is used to export the operations you develop in the IDE into your Studio project.

We'll talk more about that in the next steps.

Nuxeo Shell

Nuxeo Shell is, as its name suggest, a shell. You can use it to log in to Nuxeo and realize different actions. You can use it to connect and do maintenance work on any remote Nuxeo server.

It's a powerful tool that won't be cover in this guide. But I invite you to read its documentation.

Creating your project in Nuxeo IDE

Pre-requisites
• Having a Nuxeo Studio project • Step Installing Nuxeo IDE • Step Getting familiar with Nuxeo IDE

In this section
1 Creating a Nuxeo project (not a Webengine project) 2 Configuring Studio in IDE

Creating a Nuxeo project (not a Webengine project)

Now that you are familiar with managing your server instance, we can start coding. First thing to do is to create a new Nuxeo project.

1. Click on the **New Nuxeo Artifact** wizard button ( ).
   It displays a list of wizards used to automatically generate some code.
2. Select the first one, **Nuxeo Plugin Project**, and click on **Next**.
3. You are first asked to choose an ID for your project. Let's name it `random-string-generator`. On the same screen you can choose the Java Root package (default is `org.nuxeo.sample`), the source code location and a working set. Here we'll leave the default values.
4. Click on **Next**.
   You are now on the **Maven Settings** page. The whole Nuxeo Platform is built using Maven. You can choose to customize those Maven metadata, but it's not necessary.
5. Leave the default one and click on the **Finish** button.
   Your project is created and displayed in the **Project Explorer** panel.

Configuring Studio in IDE

We now have a brand new Nuxeo bundle. One of the cool feature of Nuxeo IDE is that it lets you bind your Eclipse project to your Nuxeo Studio project. Since you are using the Nuxeo perspective, you have two new tabs on the left Eclipse panel: one named **Nuxeo Components** and one named **Nuxeo Studio**. Take a look at the latest. You should see your Studio project if your Eclipse is properly configured. You can browse your Studio project from here. Time to make sure you have deployed the ID generation template :)

Now to actually bind the two projects together:

1. Right-click on your Eclipse project and select **Properties**.

You should have a Nuxeo entry, which contains a Nuxeo Studio sub-entry.
2. Select your Studio project and click on **OK**.
3. You have now binded your Eclipse project to your Studio project. This might seems like nothing, but we'll see how cool this actually is in the next step.

Coding your first operation

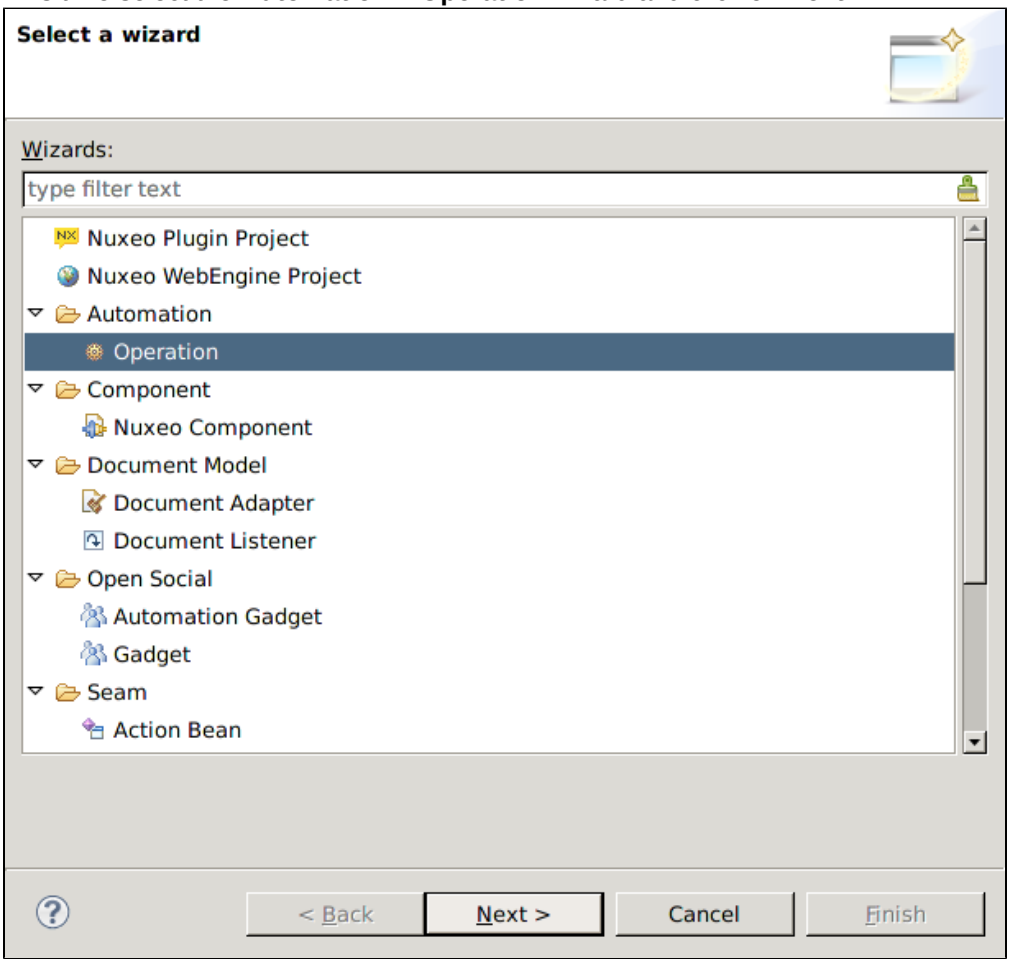
Pre-requisites
• Having installed Nuxeo IDE • Having installed a Nuxeo SDK • Being familiar with Eclipse • Creating Your Project in Nuxeo IDE

In this section
1 Creating an operation using the Operation Wizard 2 Finish chain in Studio

Creating an operation using the Operation Wizard

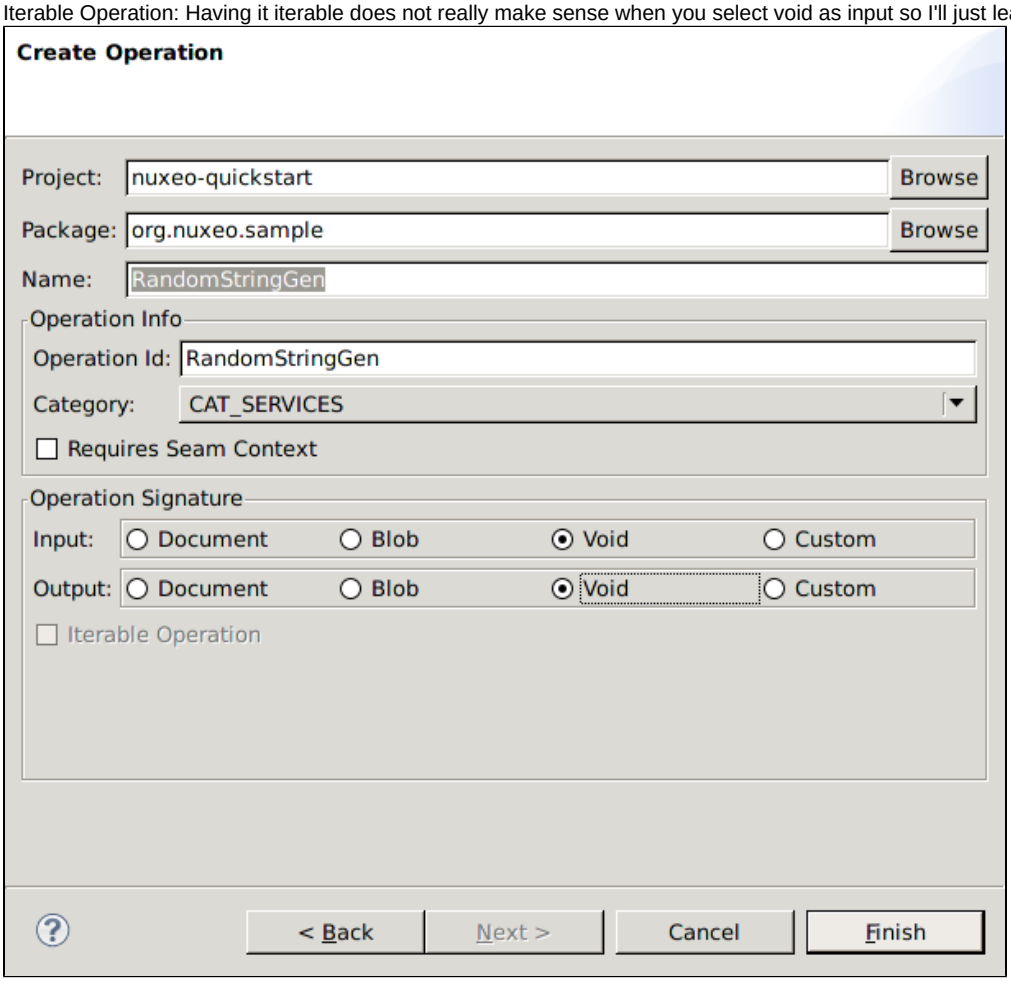
Our Eclipse project won't do much as is. We're going to add a new operation that will generate a random alphanumeric string. To do so:

1. Click on the **New Nuxeo artifact wizard** button again.
2. This time select the **Automation** > **Operation** wizard and click on **Next**.



3. Fill in the Create operation wizard:
   - Projet: It should select your current Eclipse project automatically.
   - Package: choose the Java package of your operation (because an operation is actually a Java class).
   - Name: the name of your operation. I will call it `RandomStringGenerator`.
   - Operation Id: I will use `RandomStringGenerator` too.
   - Category: This category is used in Nuxeo Studio (because yes, the ultimate goal of this is to use our new operation in Studio). I will select `CAT_SERVICES`.
   - Requires Seam Content: My operation does not need the Seam context so I will leave this box unticked.

- Operation Signature: Each operation accepts and returns something, or nothing in our case. So I will select void for both input and output.
- Iterable Operation: Having it iterable does not really make sense when you select void as input so I'll just leave it like that.



4. Click on **Finish** .
   You should now have end up with something like this:

Random String Generator Operation

```
/**
* @author ldoguin
*/
@Operation(id=RandomStringGen.ID, category=Constants.CAT_SERVICES,
label="RandomStringGen", description="")
public class RandomStringGen {
    public static final String ID = "RandomStringGen";

    @OperationMethod
    public void run() {
    }
}
```

Time to fill this skeleton and start coding. We have to generate a random alphanumeric string.

1. This can be done easily using the java.util UUID class.

```
String randomString = UUID.randomUUID().toString();
```

2. Now that the String is generated, we have to put it somewhere. When you're running an automation chain, you can access a Context object shared between every operation of your chain. So we can put it in the `OperationContext`. You can inject it in your class easily using the **`@Context`** annotation.

```
@Context
public OperationContext context;
```

3. The operation needs a name, that will be a parameter. This way we let the user decide the variable name. To declare a parameter, we use the **`@Param`** annotation.

```
@Param(name= "name", required=true)
public String name;
```

4. Of course, you need to put the variable in the operation context.

```
context.put(name, randomString);
```

Here's the final result:

Random String Generator Operation

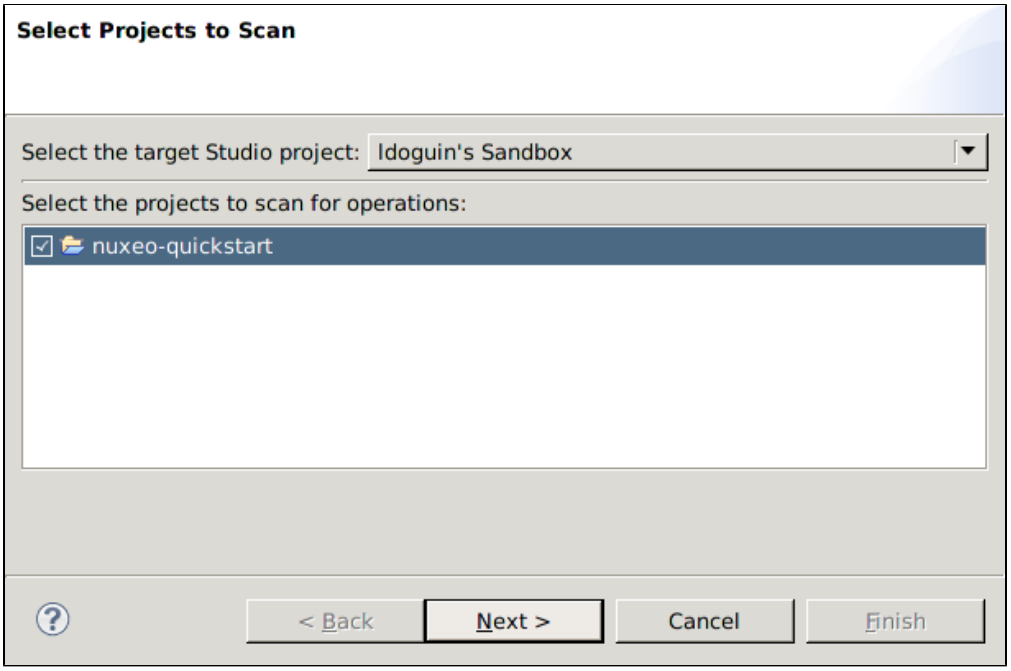
```
package org.nuxeo.sample;

import java.util.UUID;

import org.nuxeo.ecm.automation.OperationContext;
import org.nuxeo.ecm.automation.core.Constants;
import org.nuxeo.ecm.automation.core.annotations.Context;
import org.nuxeo.ecm.automation.core.annotations.Operation;
import org.nuxeo.ecm.automation.core.annotations.OperationMethod;
import org.nuxeo.ecm.automation.core.annotations.Param;

/**
* @author ldoguin
*/
@Operation(id=RandomStringGen.ID, category=Constants.CAT_SERVICES,
label="RandomStringGen", description="")
public class RandomStringGen {

    public static final String ID = "RandomStringGen"

    @Context
    public OperationContext context;

    @Param(name= "name", required=true)
    public String name;

    @OperationMethod
    public void run() {
        String randomString = UUID.randomUUID().toString();
        context.put(name, randomString);
    }

}
```

The **`@Context`** annotation lets you inject the operation context as well as any Nuxeo services, the current user etc...

Send the Operation in Studio

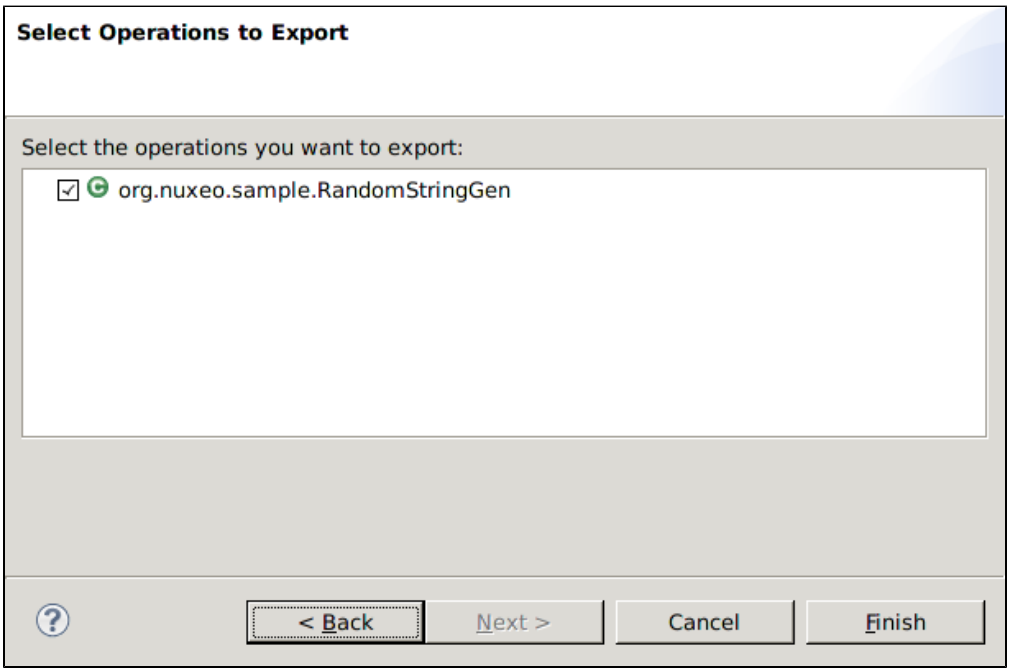
Very basic operation but you get the idea. Now it's nice to have this operation, but we still need to use it in Studio. Let's make it available there.

1. On the Nuxeo Studio tab, there's an Export Operation  buttons in the right upper corner. Click on it.
2. Select the Studio project in the drop down menu, and the Eclipse project to look for operation. Then click Next.



It should show you your operation already selected.

3. You can click on Finish and go to your Studio project.



You should now see your operation under the service category in automation. Which means we can now finish our automation chain.

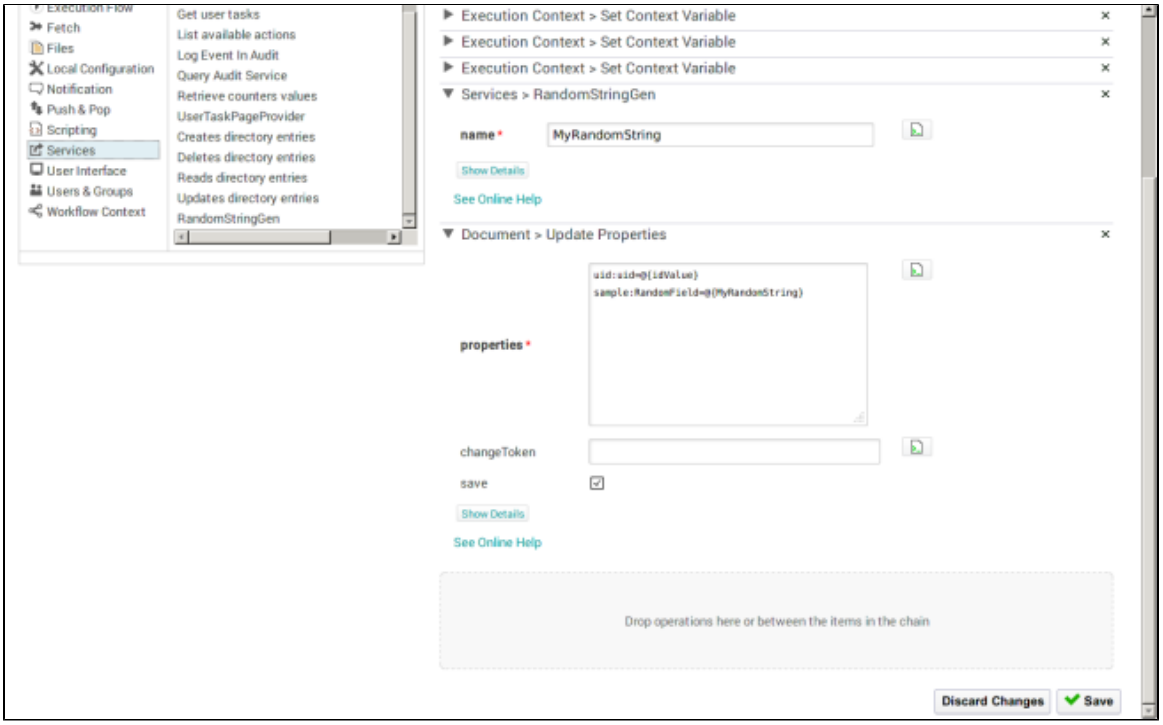
Finish chain in Studio

Let's go back in Studio to edit and finish the automation chain:

1. Click on Automation > Automation Chains > UIDUpdateChain.
2. Removed the **Document** > **Update Property** operation.
   This operation allows only one document property to be updated. Now we need to update two, the `uid` and the `RandomField` we have added. That's why we need to use the Update Properties operation instead.
3. Add the **Services** > **RandomStringGen** operation.
   As you can see we have a required **`name`** parameter. This will be the name of the random string we generate in the context. Because I am very inventive, I've picked `MyRandomString`.
4. Add the **Document** > **Update Properties** operation at the end of the chain.
   Here's the content of the **`properties`** parameter:

```
uid:uid=@{idValue}
sample:RandomField=@{MyRandomString}
```

Here's what it s supposed to look like:



Now that we know our automation chain works, let's try it on the server

Deploying your project on the server

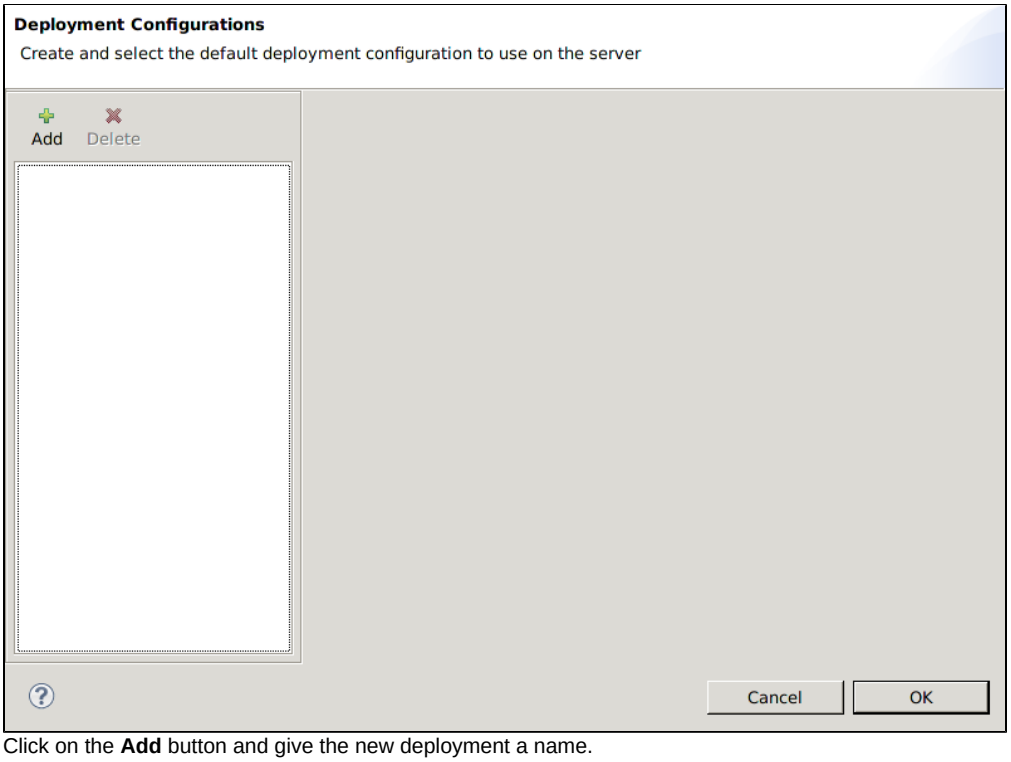
Pre-requisites
• Having installed Nuxeo IDE • Having installed a Nuxeo SDK • Being familiar with Eclipse • Creating your project in Nuxeo IDE • Coding your first operation

In this section
1 Create a deployment profile 2 Hot Reload the Project 3 Verify the result on the server

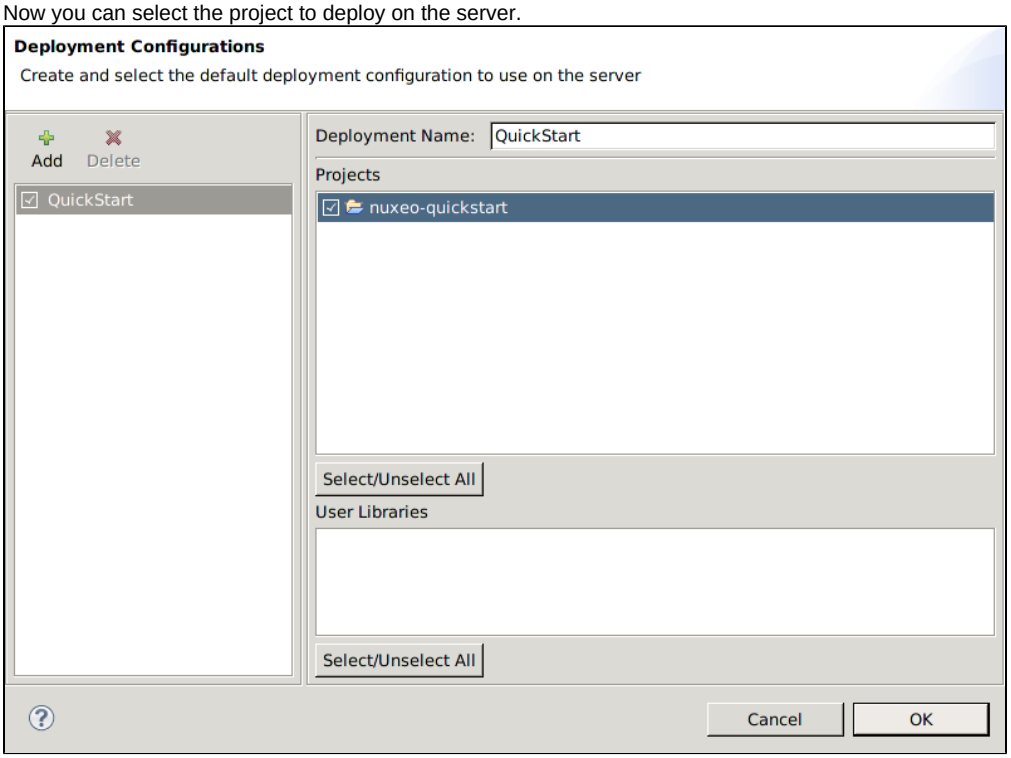
Create a deployment profile

If you deploy your Studio project as is, you will have an error saying the operation does not exist. We need to deploy it on the server. To do that:

1. Click on the 'select projects to deploy on server' button , in the Nuxeo Server view.
   You should see a new panel opening called Deployment Configurations.



2. Click on the **Add** button and give the new deployment a name.
   Now you can select the project to deploy on the server.



3. Click on **OK**.
   You deployment profile is now created. You can now load it on your server.

Hot Reload the Project

Click on the 'Reload projects on server' button to You should see something like this in the logs:

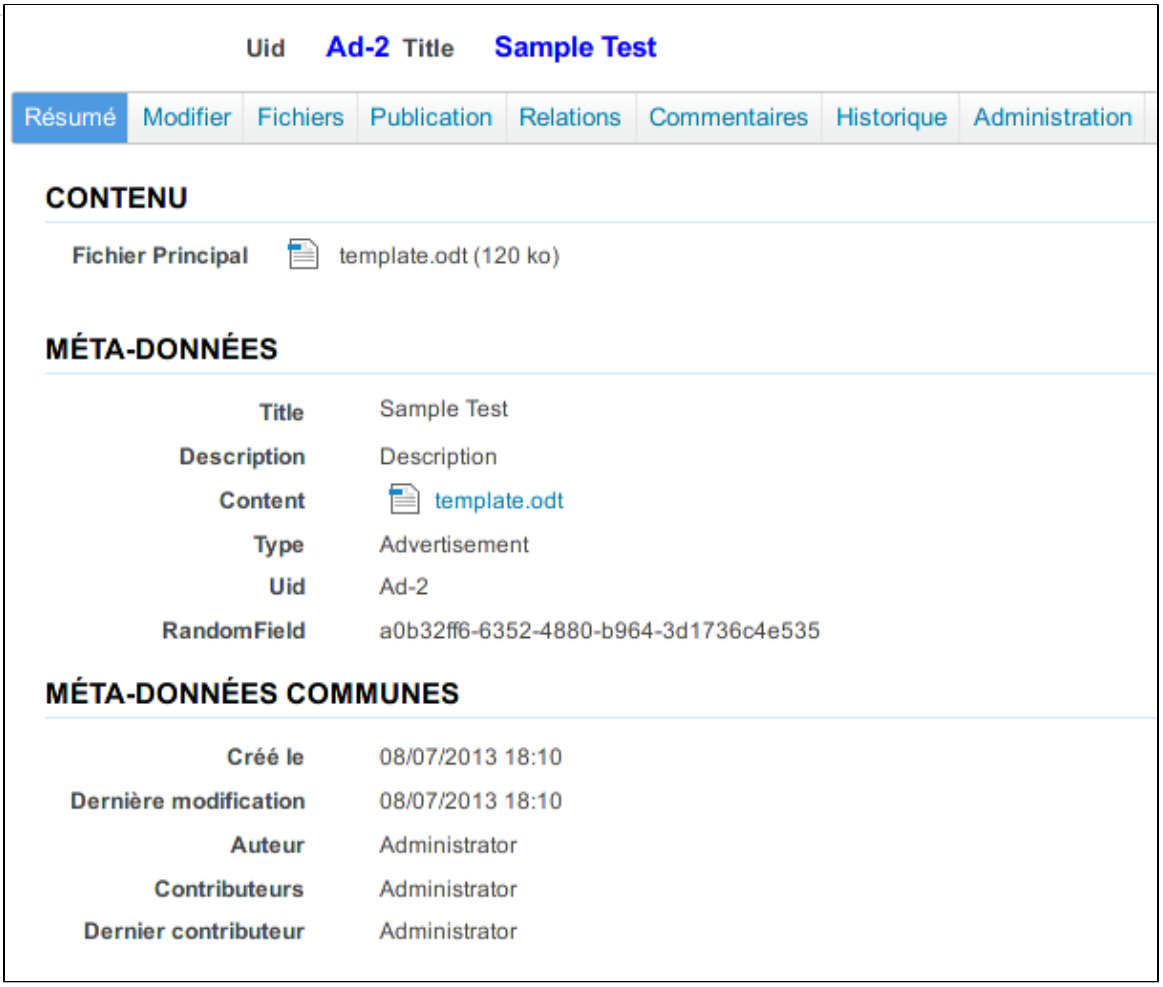
Reload Logs

```
=== Reloaded Projects on Target Server ===
= Project: nuxeo-quickstart
========================================
2013-07-08 17:05:42,151 INFO [org.nuxeo.runtime.reload.ReloadComponent] Before
undeploy bundle with name 'nuxeo-quickstart'.
=================================================================
= Component Loading Status: Pending: 0 / Unstarted: 0 / Total: 595
=================================================================
2013-07-08 17:05:42,152 INFO [org.nuxeo.runtime.reload.ReloadComponent] Undeploy done.
=================================================================
= Component Loading Status: Pending: 0 / Unstarted: 0 / Total: 594
=================================================================
2013-07-08 17:05:42,366 INFO [org.nuxeo.runtime.reload.ReloadComponent] Flush the JAAS
cache
2013-07-08 17:05:42,368 INFO [org.nuxeo.runtime.reload.ReloadComponent] Before deploy
bundle for file at '/home/ldoguin/workspaces/sample/nuxeo-quickstart/pojo-bin/main'
=================================================================
= Component Loading Status: Pending: 0 / Unstarted: 0 / Total: 594
=================================================================
2013-07-08 17:05:42,373 INFO [org.nuxeo.runtime.reload.ReloadComponent] Deploy done
for bundle with name 'nuxeo-quickstart'.
=================================================================
= Component Loading Status: Pending: 0 / Unstarted: 0 / Total: 595
=================================================================
2013-07-08 17:05:42,373 INFO [org.nuxeo.runtime.reload.ReloadComponent] Install web
resources
2013-07-08 17:05:42,374 INFO [org.nuxeo.runtime.reload.ReloadComponent] Reload runtime
properties
```

It means your project has been successfully loaded to your server.

Verify the result on the server

Now try to create a Book document again. Cool huh? This is of course a simple example, but now you know how to interact between your Eclipse project and Nuxeo Studio. You can go wild and add as many operations as you need for your operation chains and workflows.



How to create an empty bundle

This recipe describes the steps to create the bare structure of a Nuxeo add-on project (aka a bundle). It takes the Nuxeo Document Management (DM) distribution as a example but can be done with any other Nuxeo distribution, such as Nuxeo Document Asset Management (DAM) or Nuxeo Case Management Framework (CMF).

This is the very first recipe of this cookbook and it will be the basis for the development of new bundles, of new features, even of new UI elements all along this cookbook. All the other recipes will assume that this recipe has been done.

General remarks

- This recipe is not specific to a system or an IDE. You will have to adapt it to your needs. The sole obligation is to use Maven in a console. But, even this part, with experience, could be fitted to your IDE habits if you have any.
- You'll find the most frequent and common errors and problems detailed and resolved in the FAQ.
- For any remark about this recipe or about this cookbook, don't hesitate to leave us a comment on this page.

This recipe is composed of the major steps below:

- Step 1: Create the basic project skeleton using Maven
- Step 2: Complete the folder structure
- Step 3: Adapt or create Files
- Step 4: Install and check the deployment of your Bundle

What you need

Tool	Version
Java	jdk 1.6
Maven	2.2.1

Packaged Nuxeo DM distribution	5.4.1-Tomcat

Create the basic project skeleton using Maven

To create a basic folder structure, we use Maven and its default archetype. There is no required location to create your project.
To create your project structure, follow the steps below.

1. In a console, type: `mvn archetype:generate`.
   The available archetypes are listed.
2. Check that in the logs displayed, you have the two lines below:

```
[INFO] No archetype defined. Using maven-archetype-quickstart
(org.apache.maven.archetypes:maven-archetype-quickstart:1.0)
Choose archetype:
[...]
104: remote -> maven-archetype-quickstart (An archetype which contains a sample
Maven project.)
[...]
```

   If not, you may have the wrong version of Maven.

> **Warning**
> As the number of archetype is based on archetype contributions, the reference is not automatically "104". But the default proposition should still be "org.apache.maven.archetypes:maven-archetype-quickstart" whatever the version is.

3. You are prompted a series of choices to create your project. Accept the default propositions (by pressing Enter) except for the `groupId` and `artifactId`of your project which must be:

`groupId`	`org.nuxeo.cookbook`
`artifactId`	`bareproject`

4. Confirm the defined settings.
   The logs indicate that the build was successful.

Here is an example of the log you should have for the whole project creation (some lines have been skipped using "[…]"):

```
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'archetype'.
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Default Project
[INFO]    task-segment: [archetype:generate] (aggregator-style)
[INFO] ------------------------------------------------------------------------
[INFO] Preparing archetype:generate
[INFO] No goals needed for project - skipping
[INFO] [archetype:generate {execution: default-cli}]
[INFO] Generating project in Interactive mode
[INFO] No archetype defined. Using maven-archetype-quickstart
(org.apache.maven.archetypes:maven-archetype-quickstart:1.0)
Choose archetype:
1: remote -> docbkx-quickstart-archetype (-)
2: remote -> multi (-)
[...]
103: remote -> maven-archetype-profiles (-)
104: remote -> maven-archetype-quickstart (An archetype which contains a sample Maven
project.)
105: remote -> maven-archetype-site (An archetype which contains a sample Maven site
which demonstrates some of the supported document types like
    APT, XDoc, and FML and demonstrates how to i18n your site. This archetype can be
layered
```

```
    upon an existing Maven project.)
[...]
385: remote -> javg-minimal-archetype (-)
Choose a number: 104:
Choose version:
1: 1.0-alpha-1
2: 1.0-alpha-2
3: 1.0-alpha-3
4: 1.0-alpha-4
5: 1.0
6: 1.1
Choose a number: 6:
Define value for property 'groupId': : org.nuxeo.cookbook
Define value for property 'artifactId': : bareproject
Define value for property 'version': 1.0-SNAPSHOT:
Define value for property 'package': org.nuxeo.cookbook:
Confirm properties configuration:
groupId: org.nuxeo.cookbook
artifactId: bareproject
version: 1.0-SNAPSHOT
package: org.nuxeo.cookbook
Y:
[INFO] ----------------------------------------------------------------------------
[INFO] Using following parameters for creating project from Old (1.x) Archetype:
maven-archetype-quickstart:1.1
[INFO] ----------------------------------------------------------------------------
[INFO] Parameter: groupId, Value: org.nuxeo.cookbook
[INFO] Parameter: packageName, Value: org.nuxeo.cookbook
[INFO] Parameter: package, Value: org.nuxeo.cookbook
[INFO] Parameter: artifactId, Value: bareproject
[INFO] Parameter: basedir, Value: /home/user1/Workspaces/CookbookTest
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] ********************* End of debug info from resources from generated POM
************************
[INFO] project created from Old (1.x) Archetype in dir:
/home/user1/Workspaces/CookbookTest/bareproject
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 13 minutes 51 seconds
[INFO] Finished at: Thu Apr 21 10:51:01 CEST 2011
```

```
[INFO] Final Memory: 14M/213M
[INFO] ------------------------------------------------------------------------
```

Complete the folder structure

After you completed the project creation, you get this folder structure:

```
bareproject
|
`-- src
    |-- main
    |   `-- java
    |       `-- org
    |           `-- nuxeo
    |               `-- cookbook
    `-- test
        `-- java
            `-- org
                `-- nuxeo
                    `-- cookbook
```

To fit to the classical structure of a Nuxeo add-on project, you need to create new folders in `src/main` and `src/test` using your favorite means. At the end, you need to get a folder structure as shown below.

```
bareproject
|
`-- src
    |-- main
    |   |-- java
    |   |   `-- org
    |   |       `-- nuxeo
    |   |           `-- cookbook
    |   `-- resources
    |       |-- META-INF
    |       |-- OSGI-INF
    |       |-- schemas
    |       `-- web
    |           `-- nuxeo.war
    `-- test
        |-- java
        |   `-- org
        |       `-- nuxeo
        |           `-- cookbook
        `-- resources
            `-- META-INF
```

The `.war` ending of `nuxeo.war` may be deceiptive, but it is actually a folder and not a file.

Adapt or create files

Adapt the pom.xml file

We need to customize the pom.xml file provided by the archetype at the root folder of the project.

1. Change the parent entry.

```
<parent>
 <groupId>org.nuxeo.ecm.platform</groupId>
 <artifactId>nuxeo-features-parent</artifactId>
 <version>5.4.1</version>
</parent>
```

2. In the dependencies, delete the JUnit entry.
3. Add repositories.

```
<repositories>
 <repository>
  <id>public</id>
  <url>http://maven.nuxeo.org/public</url>
  <snapshots>
   <enabled>false</enabled>
  </snapshots>
  <releases>
   <enabled>true</enabled>
  </releases>
 </repository>
 <repository>
  <id>snapshots</id>
  <url>http://maven.nuxeo.org/public-snapshot</url>
  <snapshots>
   <enabled>true</enabled>
  </snapshots>
  <releases>
   <enabled>false</enabled>
  </releases>
 </repository>
</repositories>
```

Your "pom.xml" file should at the end to look like this:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>

 <groupId>org.nuxeo.cookbook</groupId>
 <artifactId>bareproject</artifactId>
 <version>1.0-SNAPSHOT</version>
 <packaging>jar</packaging>

 <name>bareproject</name>
 <url>http://maven.apache.org</url>

 <properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
 </properties>

 <parent>
  <groupId>org.nuxeo.ecm.platform</groupId>
  <artifactId>nuxeo-features-parent</artifactId>
  <version>5.4.1</version>
 </parent>

 <!-- nuxeo repos have copies of everything needed -->
 <repositories>
  <repository>
   <id>public</id>
   <url>http://maven.nuxeo.org/public</url>
   <snapshots>
    <enabled>false</enabled>
   </snapshots>
   <releases>
    <enabled>true</enabled>
   </releases>
  </repository>
  <repository>
   <id>snapshots</id>
   <url>http://maven.nuxeo.org/public-snapshot</url>
   <snapshots>
    <enabled>true</enabled>
   </snapshots>
   <releases>
    <enabled>false</enabled>
   </releases>
  </repository>
 </repositories>

 <dependencies>

 </dependencies>
</project>
```

Create a "deployment-fragment.xml" file

In order to deploy your Nuxeo add-on project in the Nuxeo server, you need to add a new file called "deployment-fragment.xml" in the "/src/main/resources/OSGI-INF" folder. This file tells the deployment mechanism which files must be copied and where. This file is not mandatory

at this stage, but it is needed to have your bundle displayed in the log at start up.

For now, the content of the file "deployment-fragment.xml" should be:

```
<?xml version="1.0"?>
<fragment version="1">
<!-- will contains some stuff -->
  <install>
<!-- useful later -->
  </install>
</fragment>
```

The content of this file will be completed in a coming recipe.

Remark:

- The given version 1 into the fragment item is important because before Nuxeo Runtime 5.4.2, the bundle dependency management was managed into the MANIFEST.MF. You have from 5.4.2 version of Nuxeo Runtime new items (require, required-by)
- If you want your bundle deployed after all other bundles/contributions, you can add a <require>all</require>
- If you have this message "*Please update the deployment-fragment.xml in myBundle.jar to use new dependency management*", this is because you didn't specify the fragment version (and maybe let dependency informations into the manifest file.
- the deployment-fragment.xml file is not required since 5.4.2 if you have no dependency information to transmit to the runtime or pre-deployment actions to execute.

Create a "MANIFEST.MF" file

As Nuxeo add-ons are OSGi modules, you need to create a "MANIFEST.MF" file in the "/src/main/resources/META-INF" folder. This file can be customized as shown in this lesson.

Here are the minimal properties the "MANIFEST.MF" file must hold:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 1
Bundle-Name: cookbook-basic-bundle
Bundle-SymbolicName: org.nuxeo.cookbook.basic;singleton:=true
Bundle-Version: 0.0.1
Bundle-Vendor: Nuxeo
```

Some of the values above are mandatory, some should be changed to adapt to your needs.
The following properties are more the less "constants" and their values must be always the same:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 1
```

The other properties should be customized to your needs.
The two principals are:

```
Bundle-Name: cookbook-basic-bundle
Bundle-SymbolicName: org.nuxeo.cookbook.basic;singleton:=true
```

- "Bundle-Name" corresponds to the human-readable name of the bundle;
- "Bundle-SymbolicName" is the reference computed by the OSGi container and looked-up by the other bundles. This declaration is immediately followed, on the same line, by ";singleton:=true" which declares to the OSGi container that the bundle can't cohabit with an other version of the bundle at runtime. The semi-colon is of course mandatory.

The other properties are:

- "Bundle-Version": This property is all on your responsibility. The Nuxeo convention is three digits separated by a dot such as "0.0.1";
- "Bundle-Vendor": This is the name of the add-on owner.

Although not used in this recipe, there is one more property you should know of: "Nuxeo-Component:".
It contains a list of files used to define various elements of your component. Its use is detailed in this lesson.

Formatting
The trickiest and most important part of a "MANIFEST.MF" file is its formatting. One mistake and the OSGi context can't be correctly started, leading to unexpected issues and an unreachable bundle. Here are the three formatting rules to respect:

1. Each property name:
   - begins at the first character of the line;
   - ends with a colon without space between the name of the property and the colon itself.
2. Each value:
   - must be preceded by a space;
   - ends with a "end of line" with eventually a comma before it.
3. There MUST be an EMPTY LINE at the END OF THE FILE.

Create files for the tests

"log4j.properties"

As the tests will run in a sandbox, it could be useful to define a file named "log4j.properties" file. It must be placed in the "/src/test/resources" folder.

Here is the content of such a file:

```
log4j.rootLogger=WARN, CONSOLE
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout
log4j.appender.CONSOLE.layout.ConversionPattern=%d{HH:mm:ss,SSS} %-5p [%C{1}] %m%n
```

To make the log more or less verbose, just change the first value of the "log4j.rootlogger" property.
In this example, the level is "WARN". If you want more details, downgrade it to "DEBUG". You will have more entries displayed in the console about Nuxeo classes involved in the running tests.

"MANIFEST.MF"

Create a new "MANIFEST.MF" file, in the "/src/test/resources/META-INF" folder this time.
The content of this file should be:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 1
Bundle-Name: cookbook-basic-bundle-test
Bundle-SymbolicName: org.nuxeo.cookbook.basic.test;singleton:=true
Bundle-Version: 0.0.1
Bundle-Vendor: Nuxeo
```

The most important difference between this content and the one declared in the "/src/main/resources/META-INF/MANIFEST.MF" file, is the value of the "Bundle-SymbolicName:" property. Those two have to be different to avoid bundle Symbolic Name collision.

Install and check the deployment of your bundle

1. Build your bundle, using the following Command Line Interface (CLI):

   ```
   $ mvn install
   ```

   In the "/target" folder of your project, you get a JAR file whose name if formed like that: artifactId-1.0-SNAPSHOT.jar.
2. Copy your brand new jar into the sub-folder "nxserver/plugins/" of your nuxeo application's root folder:
   - under Windows, assuming that the nuxeo-distribution is installed at the location "C:\Nuxeo\", copy the jar in "C:\Nuxeo\nxserver\plugins\";
   - under Linux, assuming that the nuxeo-distribution is installed at the location "/opt/nuxeo", copy the jar in

"/opt/nuxeo/nxserver/plugins".

3. Start your server using the "./nuxeoctl console" command

> You can check the dedicated Start and stop page of the technical documentation for more information about the different ways to start your server).

4. Check that your bundle is correctly deployed: check if its SymbolicName (as configured in the "/src/main/resources/META-INF") appears in the logs. The logs are displayed:
   - in the console if you started your server using the "./nuxeoctl console"
   - in the file "server.log" located in the "log" folder of your Nuxeo server root folder.
     This name is found in the list of the bundles deployed by Nuxeo in the very first lines of the logs, just after the line ended by " Preprocessing order:".

In the following example, the name of your bundle could be found at the line n°8 of the following print (some lines of the logs have been skip using "[…]"):

```
2011-04-18 10:37:02,384 INFO
[org.nuxeo.runtime.deployment.preprocessor.DeploymentPreprocessor] Preprocessing
order:
 org.nuxeo.ecm.webengine.core
 org.nuxeo.ecm.platform.ui
 org.nuxeo.ecm.platform.types.core
 org.nuxeo.ecm.platform.uidgen.core
 [...]
 org.nuxeo.ecm.platform.oauth
 org.nuxeo.cookbook.book
 org.nuxeo.ecm.platform.syndication
 org.nuxeo.ecm.platform.audit.ws
 [...]
 org.nuxeo.ecm.relations.jena
```

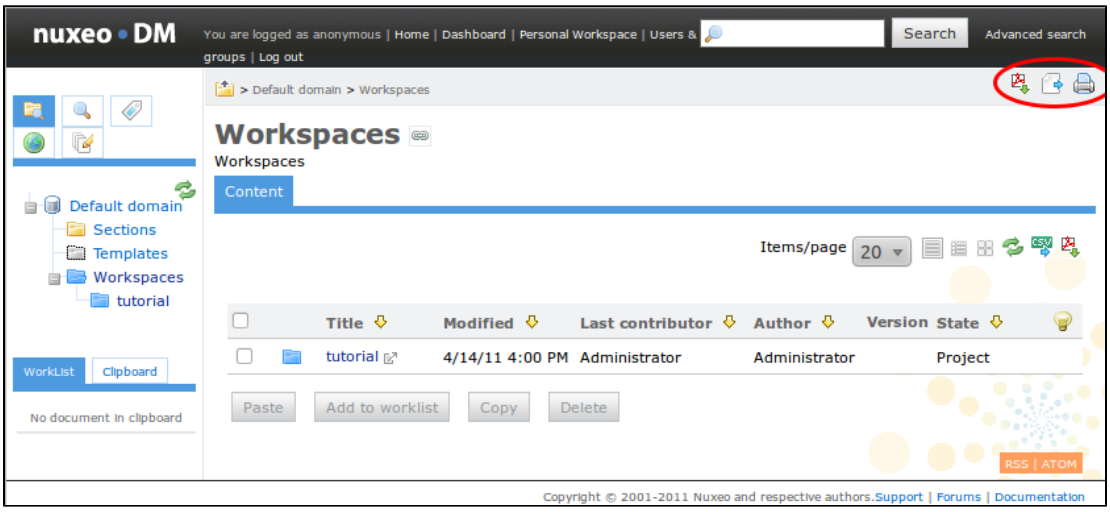
Now you've got a bundle ready for customization. You can propose your contribution to configuration and use it to improve your Nuxeo instance. Let's move to another recipe to discover how this is possible!

As said in the beginning of this recipe, if you have unexpected errors or Nuxeo Application behavior don't forget to check the FAQ and don't forget to leave us a comment about this recipe or about the cookbook!

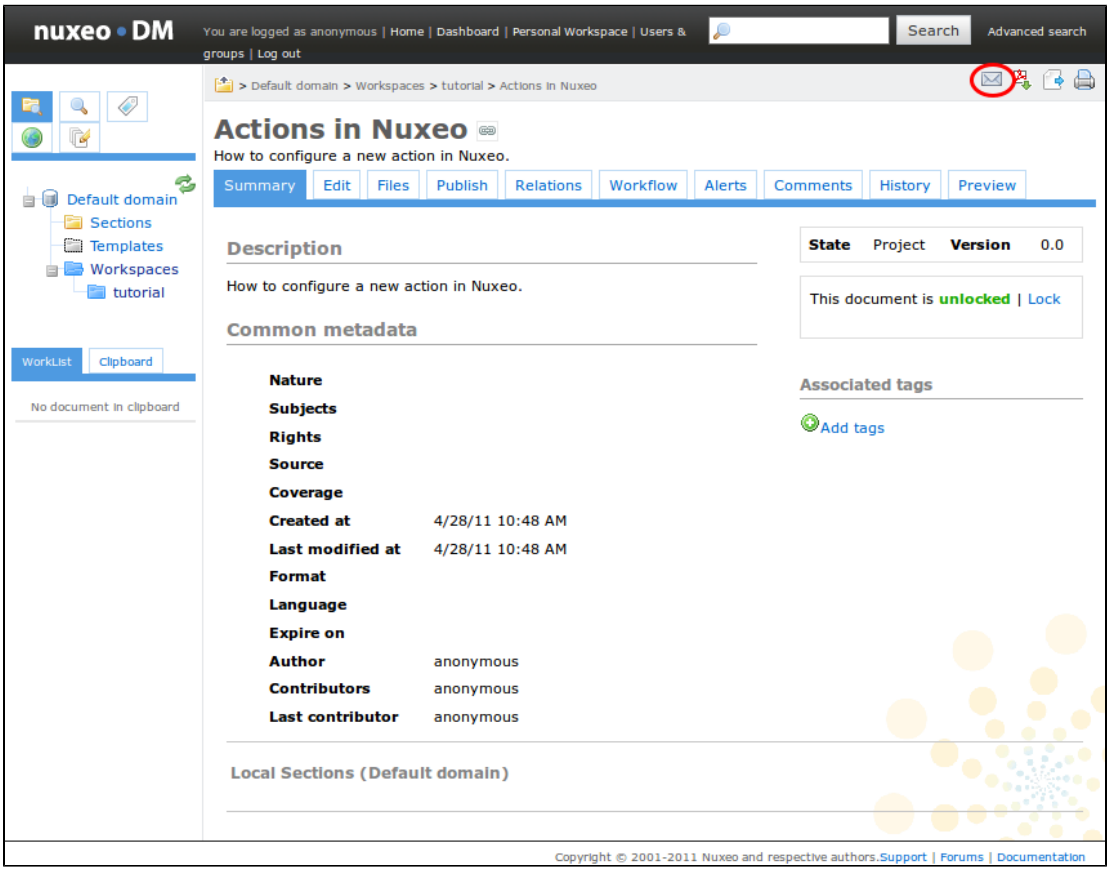
How to implement an Action

This recipe shows you how to configure an action in a bundle for a Nuxeo application.

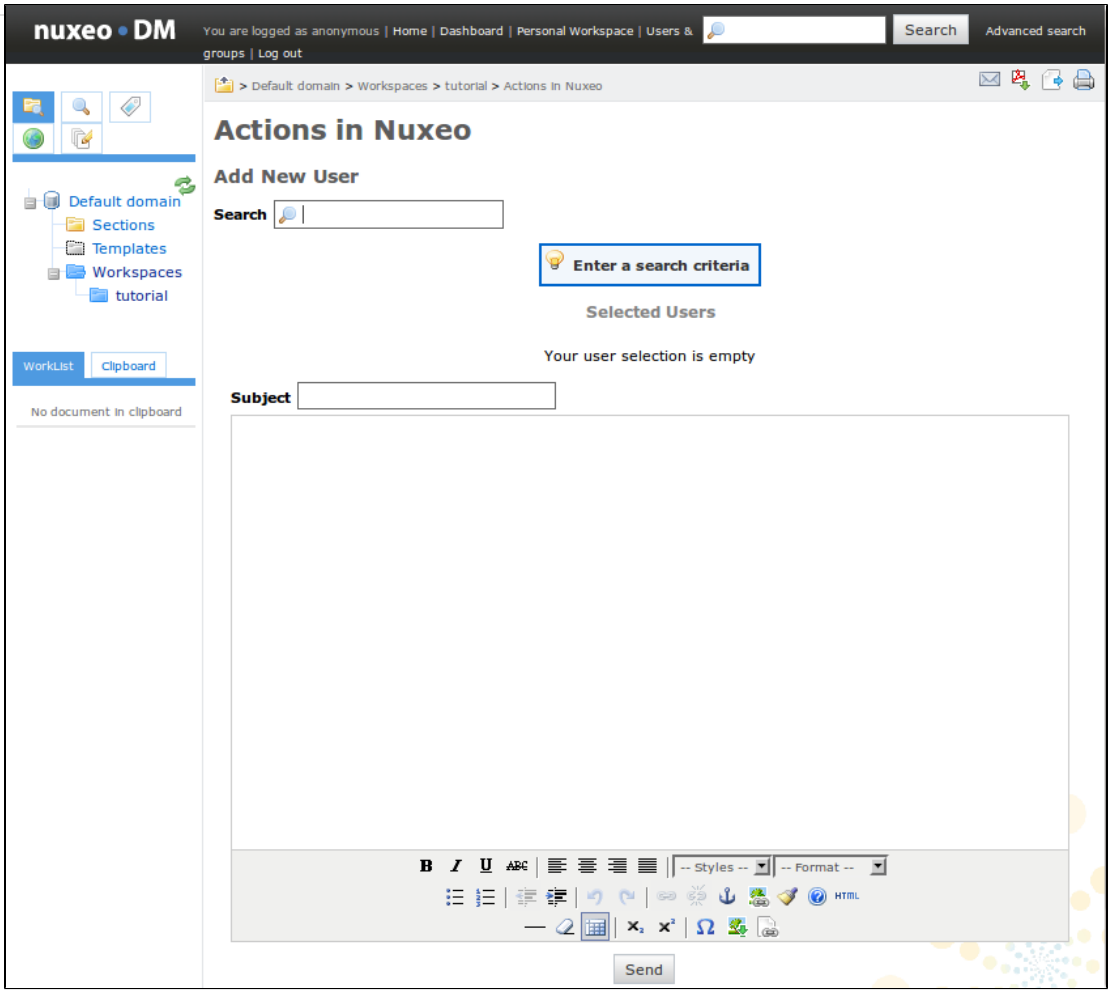
This recipe describes how to define a new clickable icon in the webpage GUI of your Nuxeo application. This icon will redirect users to the "Sending Email" page.
To keep things simple and focused on the action configuration, this recipe is based on the action called "send_email". The "send_email" action is triggered by an icon displayed in the "Context Tool area" (i.e. the red ellipse area).



This icon (circled in red) is calling the action and is displayed only when a document is exposed in the main area as shown below. We will add another icon that does the same.



When the user clicks this icon, he is redirected to the following page, from which he will be able to send the email.



For this recipe, you don't need a working email server: the call to the emailing page is sufficient. However, if you want to send the email, you can configure your Nuxeo application instance by following the steps described on the recommended configuration of a Nuxeo server.
The page How-to create an empty bundle does not exist.

This recipe is composed of the steps below:

- Create the project
- Prepare the resources needed
- Define your action in a XML file
- Edit the MANIFEST.MF file
- Install and check the deployment of your action

Create the project

If you followed the "How to create an empty bundle" recipe, you can use the project created for this recipe and jump to the configuration steps.

The very first step to implement an action is to create a new project, here called "nuxeo-action-project" (see the "How to create an empty bundle" recipe for details on the project creation).
After you completed the project's structure, you have to configure the new action and define it as an OSGi component. The project structure is generally completed manually. However, you can also get it from this zipped empty project.

Prepare the resources needed

The purpose of this recipe is to add a new icon with a label, on which users will click to send an email from the document.

1. Create a new "icons" folder in the "src/main/resources/web/nuxeo.war" folder of your project.
2. Save this icon () in the newly created "icons" folder. Of course you can use any other icon of your choice. Just be careful of the icon's name in the next steps.

3. Modify your "deployment-fragment.xml" file to incorporate this icon in your Nuxeo application.
   The "deployment-fragment.xml" file content should be like this:

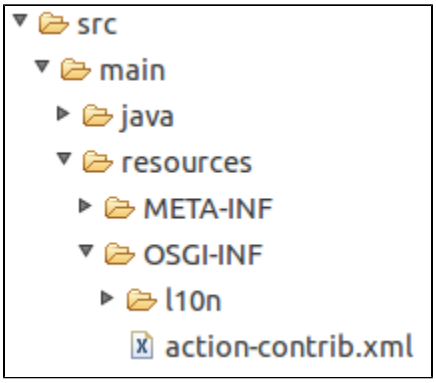
```
<?xml version="1.0"?>
<fragment version="1">

 <install>

  <!-- unzip the war template -->
  <unzip from="${bundle.fileName}" to="/" prefix="web">
   <include>/web/nuxeo.war/**</include>
  </unzip>

 </install>
</fragment>
```

Resources

The image proposed here ( ) is part of the silk icon set of FAMFAMFAM page release under Creative Commons Attribution 2.5 License.

Define your action in a XML file

Now you have to define your action in a file named "action-contrib.xml", located in the "src/main/resources/OSGI-INF" folder.



The content of the file is the following:

```
<?xml version="1.0"?>
<component name="org.nuxeo.cookbook.basic.action">
 <extension target="org.nuxeo.ecm.platform.actions.ActionService"
  point="actions">
  <action icon="/icons/asterisk_orange.png" id="myAction"
   label="An other way to send an email" link="send_email" order="17">
   <category>DOCUMENT_UPPER_ACTION</category>
   <filter id="cookbook_action_filter">
    <rule grant="false">
     <facet>Folderish</facet>
    </rule>
   </filter>

  </action>
 </extension>
</component>
```

Component
The name of your component is the the name of your action.
Make sure that this name is unique otherwise your Nuxeo application instance could behave unexpectedly.
There should be only one component called "org.nuxeo.cookbook.basic.action" for the whole application.

Extension

- The "target" of the extension is the Java class which implements your action.
- The "point" of the extension is its type. Here, you want to create an action, so the extension point is "actions" (don't forget the ending 's').

Action
The <action> tag defines:

- which "icon" will be displayed to the user. The path to the icon file is relative to the root of the "nuxeo.war" folder of your Nuxeo application;
- which "label" of the action, i.e. the text displayed as a tooltip. You can use a localizable property but to do so see the recipe "How-to localize a bundle";
- the action's "id" in Nuxeo application instance;
- which action referenced in Nuxeo should be triggered when the user clicks on the icon (parameter "link"). Here the referenced action is a simple string (send_email). It could be the path to a page or a key known by your Nuxeo application instance.

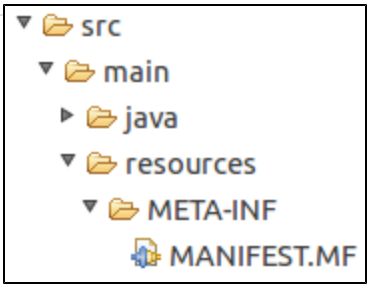
Category
The category value specifies the position of the button in the page.
Nuxeo UI is divided in several parts, called categories (cf the page Actions Display). An action can belong to several categories and lists them in the "category" attribute. For this example, it's the "Contextual tool area", whose ID is "DOCUMENT_UPPER_ACTION".

Filter
The filter value configures the conditions that must be met for the button to be displayed.
Here, the filter defines that the document should be displayed only if it's **not** decorated with the facet "folderish" (i.e. is not "folder" like). Then, the icon is displayed.
For more advanced customization, you can see the following pages:

- the filter section of the actionService page,
- the Extension Point actions page,
- the Extension Point filters page.

Edit the MANIFEST.MF file

To be detected by Nuxeo, you have to declare your new XML file ("action-contrib.xml") in the "MANIFEST.MF" file of the "src/main/resource/META-INF" folder. Thus your new action is a real and genuine Nuxeo Component.



To do so, edit the MANIFEST.MF file and add a new line with: `Nuxeo-Component: OSGI-INF/action-contrib.xml`.
In the end, and providing that you didn't add other contributions, your "MANIFEST.MF" file should contain:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 1
Bundle-Name: cookbook-action-bundle
Bundle-SymbolicName: org.nuxeo.cookbook.basic.action;singleton:=true
Bundle-Version: 0.0.1
Bundle-Vendor: Nuxeo
Nuxeo-Component: OSGI-INF/action-contrib.xml
```

The page How-to create an empty bundle does not exist.

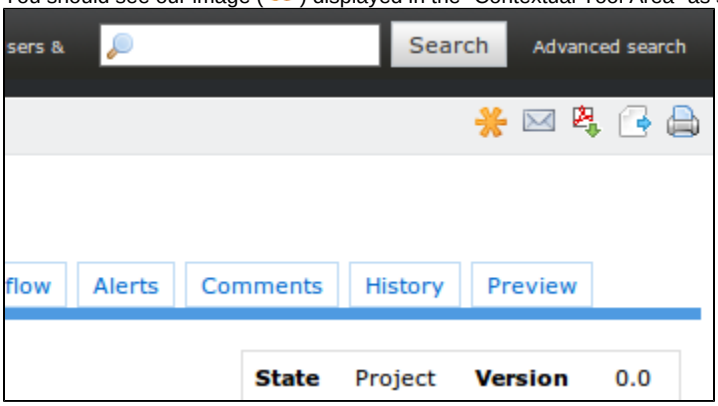
Install and check the deployment of your action

To be sure that all of your tedious work is fruitful, you have to deploy your brand new bundle in a working Nuxeo application instance.
The page How-to create an empty bundle does not exist.

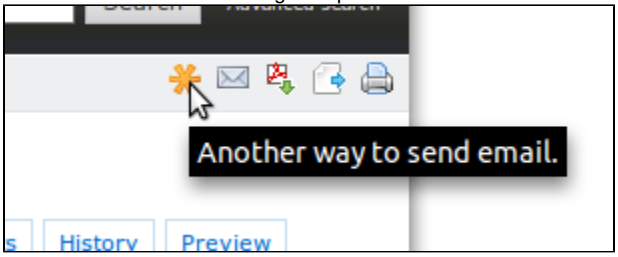
Now you know that your bundle is correctly deployed and installed. You also need to check that it works.

1. In a browser, go to the URL http://localhost:8080/nuxeo.
2. Connect to your Nuxeo application with an existing user (default credentials are Administrator/Administrator (login/password).
3. Go to a workspace.
4. Open or create a document.

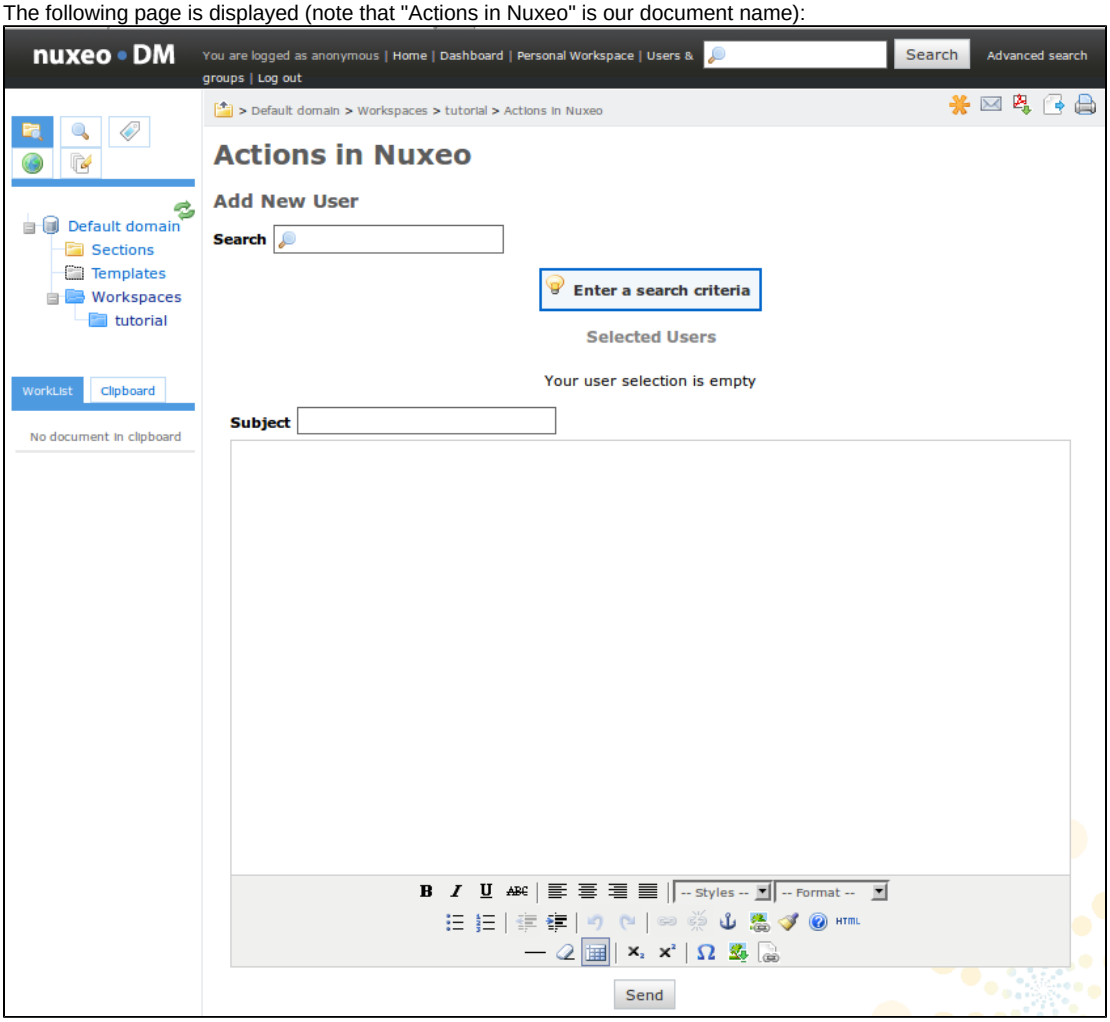
   You should see our image ( ) displayed in the "Contextual Tool Area" as shown below.



5. Move the pointer over the icon.
   You should have the following tooltip:



Click on your brand new and beautiful action icon.
The following page is displayed (note that "Actions in Nuxeo" is our document name):



Et voilà!

As said in the beginning of this recipe, if you have unexpected errors or Nuxeo Application behavior don't forget to check the FAQ and post a comment about this recipe or about the cookbook if you want to!

How to Contribute a Simple Configuration in Nuxeo

Here we will explain how you can contribute a simple configuration with Nuxeo IDE.

This "How to" does not cover contribution of a new behavior (with Java Code). Most of configurations possibilities are possible with:

- Nuxeo Studio through a non-developper interface
- or Nuxeo IDE with some templates (see other not yet written resources in this section :)

But some tricky configurations are not possible with Nuxeo Studio. For instance if you want to contribute a new JavaScript resource integrated in each JSF page. We will explain how to do that in this page.

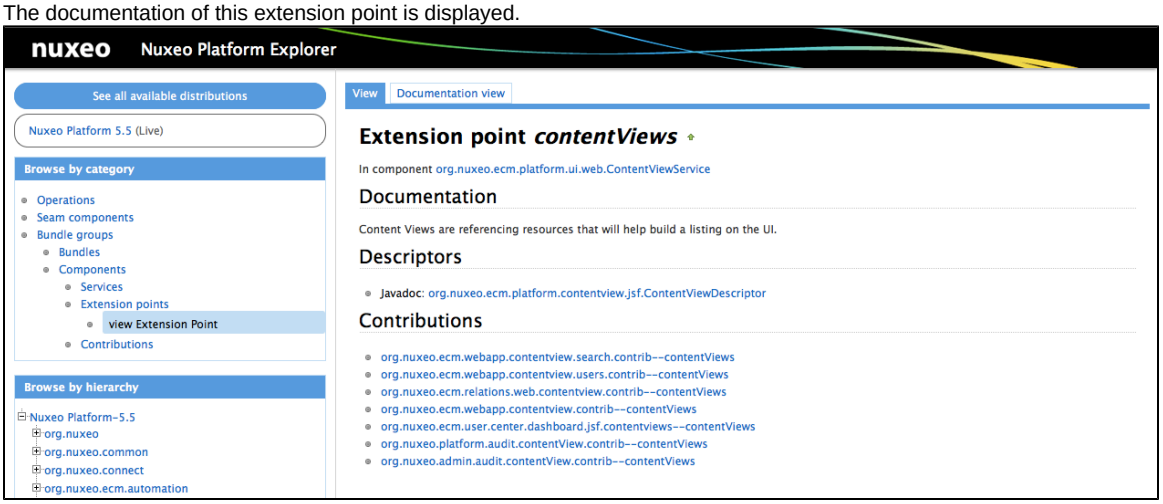
Recipe steps

Find the Extension Point Where to Contribute

Your first step is to find the open door configuration where you want to contribute. We call these open doors **Extension points**. Nuxeo lists all extension points for a given version in the Nuxeo Explorer.

1. Click on the **Explore** button of the given version you work with.

2. In the **Browse by category** panel, click on **Bundle groups > Components > Extension points**.
3. In the **Extension Point** column, click on the extension point you're interested in.
   The documentation of this extension point is displayed.



4. Then, if you click on any link in the **Contributions** section, you will see all the default contributions implemented into your Nuxeo instance.
   There are hundreds of configuration possibilities!

Create Your Contribution

Once you have found the extension point you want to contribute to, you can just contribute really easily with Nuxeo Studio (think of pioneers that did that without Nuxeo IDE and Nuxeo Explorer :).
Here we assume you that you have installed Nuxeo IDE and follow the Getting Started guide or the How-to create an empty bundle.

1. Create a file `myproject-servicewhereIcontribute-contribution.xml` into the directory `src/main/resources/OSGI-INF/` of your project.
2. Declare an empty component into this file, like that:

```
<?xml version="1.0"?>
<component
name="org.mycompany.myproject.extention.point.where.we.contribute.contribution"
version="1.0">


</component>
```

3. You must give a **unique name** for your component. If the name of your package is not unique it will **not be deployed**.

> In Nuxeo, we follow this naming way **org.mycompany.myproject.extention.point.where.we.contribute.contribution**.
> You can follow your way but be careful to avoid conflicts.

4. Add your contribution that express the configuration you want in the component XML fragment. You get something like:

```
<?xml version="1.0"?>
<component
name="org.mycompany.myproject.extention.point.where.we.contribute.contribution"
version="1.0">

       <!-- target and point value is given by the extension point definition -->
        <extension target="name.of.the.component.where.the.service.isdeclared"
point="pointNameIntoThisComponent">
          <!-- here you put your configuration XML fragment
          ...
        </extension>
</component>
```

Declare Your Contribution into Your Bundle

In the previous section you have created your configuration. But if you build the JAR of your project and put it into the Nuxeo server, your component will not be deployed as it is not declared into your bundle. You must notify the existence of your component in your JAR for the Runtime to ask him to deploy it.

This declaration is made through the `src/main/resources/META-INF/MANIFEST.MF` file:
Create a new parameter, if it does not exist.

```
Manifest-Version: 1.0
Bundle-Vendor: Nuxeo
Bundle-ActivationPolicy: lazy
Bundle-ClassPath: .
Bundle-Version: 5.5
Bundle-Name: jalon-dm-bundle
Nuxeo-Component: OSGI-INF/extensions/me.jalon.dm.bundle.importer.FileS
 ystemFetcher.xml,OSGI-INF/extensions/com.mycomapny.test.FillIDDocumen
 t.xml,OSGI-INF/extensions/com.mycomapny.test.asda.xml
Bundle-ManifestVersion: 2
Bundle-SymbolicName: jalon-dm-bundle
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
```

```
Manifest-Version: 1.0
... all the existing element already set ...
Nuxeo-Component: OSGI-INF/myproject-servicewhereIcontribute-contribution.xml
```

If the Nuxeo-Component already exists with another component declaration, separate them by commas.
The page How-to create an empty bundle does not exist.

Override the Nuxeo Default Configuration

Most of the time you will want to override an existing Nuxeo Component. Each extension point has its own logic (even if most of the time you will just have to contribute the same item with the same name). So look into the extension point definition for how to override an existing configuration.

But you have to take care of another thing. In fact components deployment is linear, so if you want to override an existing configuration, it must be deployed AFTER the existing component.

1. First you must identify this component: using Nuxeo Explorer, go to the extension point definition (see the first section).
2. Click on the contribution you want to override.
3. Copy the name of the component (value after **In component**).
4. And paste it in your component into a `<require>` item.
   You will have something like that:

```
<?xml version="1.0"?>
<component
name="org.mycompany.myproject.extention.point.where.we.contribute.contribution"
version="1.0">
   <require>name.of.the.component.you.want.to.override</require>

   <!-- target and point value is given by the extension point definition -->
   <extension target="name.of.the.component.where.the.service.isdeclared"
point="pointNameIntoThisComponent">
     <!-- here you put your configuration XML fragment
        ...
   </extension>
</component>
```

How to setup a test SMTP server

For testing purpose, for instance with notifications, you often need to setup an SMTP server. There is an easy way to setup a local smtp server by using this project : http://nilhcem.github.com/FakeSMTP/. It binds a dummy SMTP server that listens on whichever port you want (that also mean that those mails won't be really sent) and on which you can setup Nuxeo so send outgoing mails : when the server sends a mail, it shows up into the list of received EMails.

To get this up and running, you only need to adjust your $NUXEO_HOME/bin/nuxeo.conf file to reflect this setup.

$NUXEO_HOME/bin/nuxeo.conf

```
...

# Mail settings (for notifications)
#nuxeo.notification.eMailSubjectPrefix="[Nuxeo]"
mail.transport.host=localhost
mail.transport.port=25000   # Adjust with FakeSMTP configuration
#mail.transport.auth=
#mail.transport.user=
#mail.transport.password=
#mail.from=

...
```

You can download a precompile version of FakeSMTP here : fakeSMTP.jar. To launch the software, just launch the command :

```
java -jar fakeSMTP.jar
```

For Windows users, you can check this software that seems to do the same job : http://smtp4dev.codeplex.com/

Implementing local groups or roles using computed groups

In this how to, you will learn how to let managers of a workspace determine who is part of locally defined groups (local to the workspace). It is like implementing a "role" notion. If you are familiar with Social Collaboration module, you know there are "members" of a social

workspace, and "administrators". And that users with the administrator role can define who is a member and who is an administrator. Thanks to this piece of documentation, you will be able to implement the same behavior, but for any "role" you want, and even think of more subtle use cases.

The Nuxeo security system gives you all the tools you need to define security from giving a simple right to a specific user on a document to defining complex use cases. You can basically play with ACLs, granting and denying permissions to users and groups. Groups in Nuxeo are defined by users part of the "powerusers" or "administrators" groups, in the Admin Center. But it is also possible to define another category of groups, whose content definition is not "manual": the computed groups.

Computed groups let you define a list of groups to which users will be affected using Java code. There are multiple use cases where you will need this feature. Implementation will require Java development knowledge and if you are familiar with Nuxeo Core Development (CoreSession, DocumentModel, UnrestrictedRunner, ...), it's better.

Development environment requirements:

- a Nuxeo Studio project (for the Workspace modification and User action definition),
- a Nuxeo SDK instance ready for test,
- Nuxeo IDE (for bundle creation and computed group definition).

Examples of uses cases for which you will need computed groups:

- delegation management
- local groups on a workspace,
- group resolved by complex logic more generally.

In this section

In the coming example we will implement the notion of "local groups" thanks to computed groups. Users with management permission on the workspace will be able to decide who is part of the "validators" group of the workspace by editing one of the metadata of this workspace. Some more complex examples can be thought of. Here is the global strategy to implement this use case:

- I want to create a virtual group named `$idWorkspace_validator` for each workspace, where `$idWorkspace` is the id of the workspace. Virtual means that the group is not referenced in the group directory. Affectation of users to the group will be resolved by a piece of Java code, just after the user gets authenticated to the system.
- Users affected to the `$idWorkspace_validator` group are the ones listed in the `wks:validator` property of the Workspace `$idWorkspace`. `wks:validators` must be added as a property containing the list of user names, on the Workspace type.
- I want to create a button on each document whose state is `inProgress`. Clicking on this button starts a simple validation workflow with one validation task assigned to the *`$idWorkspace`*`_validator` group where `$idWorkspace` is the id of the closest parent workspace.

This use case is simple but from this example you can easily implement the delegation feature.

Preparatory step: updating the Workspace document type

Here, we will just add the `wks:validators` field to the Workspace document type definition and improve the form to let the manager of the workspace set this value.

1. In Studio, create a schema with the name `workspace` and prefix `wks`.
2. In this schema, define a field `validators` as a list of string.
3. In Studio, create a `Workspace` document type (to override the default Workspace document type set by Nuxeo).
4. In the Workspace definition, add the schema `workspace` as an extra schema.
5. In the Creation, View and Edition layouts, replace the *Warning...* widget by the `wks:validators` field and set the widget `Multiple Users/Groups suggestion` and force only users suggestions.

Now, you can specify a list of validators into each workspace. Let's make users of this list members of the `$idWorkspace_validator` group.

Java-based membership definition

This part is the most interesting part of this presentation, where we see how Computed Group Service is leveraged to have dynamical group definitions.
We can resume the Computed Group Service like that:

- You can register a class that implements a method that will be called just after each user connection.
- The list of strings returned by your method will be considered as the list of virtual groups the user belongs to.

> You should first have a quick first look to the Java doc of the ComputedGroupService and the explorer.nuxeo.org documentation around this service and get back to it after having played this tutorial!

Preparing the project

This part assumes you have Nuxeo IDE configured with the Nuxeo SDK associated and Nuxeo Connect account referenced. Please look this documentation if you don't.

1. Create a new Nuxeo Plugin Project.
2. Add the following component:

src/main/resources/OSGI-INF/test-computed-group-contrib.xml

```
<?xml version="1.0"?>
<component
 name="org.nuxeo.mail.management.security.computer.group.contribution"
 version="1.0">
 <extension point="computer"
  target="org.nuxeo.ecm.platform.computedgroups.ComputedGroupsServiceImpl">
  <groupComputer name="ValidatorsGroupComputer">
   <computer>org.nuxeo.project.computed.group.ValidatorsGroupComputer
   </computer>
  </groupComputer>
 </extension>
 <extension point="computerChain"
  target="org.nuxeo.ecm.platform.computedgroups.ComputedGroupsServiceImpl">
  <groupComputerChain append="true">
   <computers>
    <computer>ValidatorsGroupComputer</computer>
   </computers>
  </groupComputerChain>
 </extension>
</component>
```

   - The first contribution `computer` defines the class that will implement the logic that will return the list of virtual groups the user belongs to.
   - The second contribution `computerChain` enables to contribute and chain multiple resolution logics.
3. Don't forget to reference the XML contribution in the `src/main/resources/META-INF/MANIFEST.MF`. The file must be like that:

MANIFEST.MF

```
Bundle-ActivationPolicy: lazy
Bundle-ClassPath: .
Manifest-Version: 1.0
Bundle-Name: test-computed-group
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Nuxeo-Component: OSGI-INF/test-computed-group-contrib.xml
Bundle-Version: 5.7.1
Bundle-ManifestVersion: 2
Bundle-SymbolicName: test-computed-group
Bundle-Vendor: Nuxeo
```

Don't forget to let the last line empty, without any character.

Now, let's see how to implement the memberhsip logic based on the `wks:validators` property value.

Coding your Computer Group

In the previous section we asked Nuxeo Runtime to register our new computer group. We named the class `org.nuxeo.project.computed.group.ValidatorsGroupComputer.`

- So we must first create a class in `src/main/java`, defined in the package `org.nuxeo.project.computed.group` and named *`ValidatorsGroupComputer.`*
- This class must implement the `GroupComputer` interface. The Nuxeo Platform delivers an abstraction of this class with main class implemented named `AbstractGroupComputer`. We suggest to extend this class.
- The main method to implement is the `getGroupsForUser` that returns the list of virtual groups to which the user belongs given as parameter.
- The difficulty is that when `getGroupsForUser` is called the user is not yet connected. So you must play with the Unrestricted Runner object.

`ValidatorGroupComputer` class creation

1. As usual, simply create a class in the `src/main/java`.
2. Mark it as extending the `AbstractGroupComputer` class.
   You must have something like that:

Simple Static Computer Group

```
package org.nuxeo.project.computed.group;

import java.util.ArrayList;
import java.util.List;
import org.nuxeo.ecm.platform.computedgroups.AbstractGroupComputer;
import org.nuxeo.ecm.platform.usermanager.NuxeoPrincipalImpl;
/**
 * @since 5.7.2
 *
 */
public class ValidatorsGroupComputer extends AbstractGroupComputer {
    @Override
    public List<String> getGroupsForUser(NuxeoPrincipalImpl nuxeoPrincipal)
            throws Exception {
        List<String> result = new ArrayList<String>();
        result.add("myTestGroup");
        return result;
    }
    @Override
    public List<String> getAllGroupIds() throws Exception {
        // TODO Auto-generated method stub
        return null;
    }
    @Override
    public List<String> getGroupMembers(String groupName) throws Exception {
        // TODO Auto-generated method stub
        return null;
    }
    @Override
    public List<String> getParentsGroupNames(String groupName) throws Exception {
        // TODO Auto-generated method stub
        return null;
    }
    @Override
    public List<String> getSubGroupsNames(String groupName) throws Exception {
        // TODO Auto-generated method stub
        return null;
    }
}
```

TEST

As you can see in this example, the computer group statically returns `myTestGroup. Let's test your test environment:`

1. Start your SDK instance from the Nuxeo IDE interface.
2. Add your project into the deployment configuration.
3. Refresh the deployment server.
4. Connect as Administrator into your Nuxeo instance.
5. Go to **Home** > **Profile**.
6. You must see a section virtual user into the main view with the `myTestGroup` referenced.

If you don't have this please look errors message into your Java project and into the server console.

If you refresh several times your project in the SDK server, you will see `myTestGroup` as many times as you did refresh. This is because the extension point registering your Computer Group adds your contribution with each refresh. But if you stop and restart the server, your contribution will be deployed once, and `myTestGroup` will be displayed once.

Now, we need to replace this static result by a dynamic one that will be the list of `$idWorkspace_validator` where the user is referenced. But when the `getGroupsForUser` method is called, no Session on the Core Repository is available as the user is not yet connected. Here comes the `UnrestrictedRunner` object.

Here you can find the project ready to use.

UnrestrictedRunner object

Extending the `UnrestrictedRunner` object helps you executing code with a session without security constraint, even if you don't have session available.

How it works:

1. Define a constructor where you will initialize the parameters needed for your code unrestricted execution.
2. Implement a run method where a `CoreSession` will be available without restriction.
3. Execute the `runUnrestricted` method that will execute your run implementation without restriction.

Why do we need of this? Because in our example, we would like to fetch all workspaces where the user about to connect is referenced into the `wks:validators` field.

In other words, we would like to make the following query `SELECT * FROM Workspace WHERE wks:validators = 'theUsername'`, to get the id of each workspace to create the dynamic virtual groups list. Here is the code result:

UnrestrictedRunner Example implementation: get Workspace Ids

```
protected class GetWorkspaceIds extends UnrestrictedSessionRunner {


        private static final String QUERY_GET_WORKSPACE_IDS = "SELECT ecm:uuid "
                + "FROM WORKSPACE WHERE wks:validators = '%s'";


        public IterableQueryResult ids = null;


        private String username;

        protected GetWorkspaceIds(String repositoryName, String username)
                throws Exception {
            super(repositoryName);
            this.username = username;
        }


        @Override
        public void run() throws ClientException {
            String query = String.format(QUERY_GET_WORKSPACE_IDS, username);
            ids = session.queryAndFetch(query, "NXQL");
        }
    }
```

You can create this class as a public class, but we suggest to create it directly into the Computer Group.

You will need this pattern several times to implement you Computer Group. So lets move to the next section and first replace the `getGroupsForUser` method with the dynamic resolution one.

`ValidatorGroupComputer` class with dynamic group list resolution

In this section we will just merge information from the two previous ones and test it.

Here is the final version of the `ValidatorGroupComputer` class:

ValidatorGroupComputer with dynamic groups

```
package org.nuxeo.project.computed.group;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.nuxeo.ecm.core.api.ClientException;
import org.nuxeo.ecm.core.api.IterableQueryResult;
import org.nuxeo.ecm.core.api.UnrestrictedSessionRunner;
import org.nuxeo.ecm.core.api.repository.RepositoryManager;
import org.nuxeo.ecm.platform.computedgroups.AbstractGroupComputer;
import org.nuxeo.ecm.platform.usermanager.NuxeoPrincipalImpl;
import org.nuxeo.runtime.api.Framework;
/**
 * @since 5.7.2
 *
 */
public class ValidatorsGroupComputer extends AbstractGroupComputer {
    private static final Log log = LogFactory.getLog(ValidatorsGroupComputer.class);
    @Override
    public List<String> getGroupsForUser(NuxeoPrincipalImpl nuxeoPrincipal)
            throws Exception {
        String username = nuxeoPrincipal.getName();
        GetWorkspaceIds runner = new GetWorkspaceIds(getRepository(), username);
        runner.runUnrestricted();
        List<String> groupIds = new ArrayList<String>();
        String groupId = null;
        for (Map<String, Serializable> id : runner.ids) {
            groupId = ((String) id.get("ecm:uuid")) + "_validator";
            log.debug("Virtual Group Id found: " + groupId);
            groupIds.add(groupId);
        }
        return groupIds;
    }
    @Override
    public List<String> getAllGroupIds() throws Exception {
        // TODO Auto-generated method stub
        return null;
    }
    @Override
    public List<String> getGroupMembers(String groupName) throws Exception {
        // TODO Auto-generated method stub
        return null;
    }
    @Override
    public List<String> getParentsGroupNames(String groupName) throws Exception {
        // TODO Auto-generated method stub
        return null;
    }
    @Override
    public List<String> getSubGroupsNames(String groupName) throws Exception {
        // TODO Auto-generated method stub
        return null;
    }
```

```
protected class GetWorkspaceIds extends UnrestrictedSessionRunner {
    private static final String QUERY_GET_WORKSPACE_IDS = "SELECT ecm:uuid "
            + "FROM WORKSPACE WHERE wks:validators = '%s'";
    public IterableQueryResult ids = null;
    private String username;
    protected GetWorkspaceIds(String repositoryName, String username)
            throws Exception {
        super(repositoryName);
        this.username = username;
    }
    @Override
    public void run() throws ClientException {
        String query = String.format(QUERY_GET_WORKSPACE_IDS, username);
        ids = session.queryAndFetch(query, "NXQL");
    }
}
private String getRepository() {
    return
```

```
Framework.getLocalService(RepositoryManager.class).getDefaultRepository().getName();
    }
}
```

TEST 1

As you can see in this example, the computer group statically returns `myTestGroup. Let's test your test environment:`

- Stop your server from the Nuxeo IDE interface (if you didn't do it).
- Start it again from the Nuxeo IDE interface.
- Add your project into the deployment configuration (if you removed it).
- Refresh the deployment server.
- Connect as Administrator into your Nuxeo instance.
- Go to **Home** > **Profile**.
- You must see a section virtual user into the main view with no group referenced.

TEST 2

- Right-click on your Java project into the Nuxeo IDE.
- Go to **Nuxeo** > **Nuxeo Studio**.
- Check the Nuxeo Studio Project where you defined the Workspace with the workspace schema and validate.
- Refresh.
- Connect as Administrator into your Nuxeo instance.
- Create a workspace and add Administrator as validator.
- Log out.
- Log in as Administrator.
- Go to **Home** > **Profile**.
- You must see a section virtual user into the main view with one group referenced.

Why does Administrator log out? Because the resolution of groups are **only during the connection.**

Button creation

And now we have to create the button that starts the workflow and assign the validation task to the virtual group.

This part is a pure Studio demonstration, just a way of making sure our newly defined group do work :

1. Create a user action:
    a. Choose the Contextual Tool category.
    b. Add filter to limit to users that have Write permission.
2. Create an automation chain and attach it to this action.
   In the Automation Chain definition:
    a. Fetch > Context Document(s).
    b. Execution Context > Set Context Variable From Input: `name = "documentToValidate"`.
    c. Document > Get Parent : `type = Workspace`.
    d. Execution Context > Set Context Variable: `name = "validatorGroup", value = "group:@{Document.id}_validator"`.
    e. Execution Context > Restore Document Input : `name = "documentToValidate"`.
    f. Service > Create Task : task name = "Vallidation", directive = "Please Validate the document", variable name for actors prefixed = "validatorGroup", create one task per actor = unchecked

TEST

1. Stop your server from the Nuxeo IDE interface (if you didn't do it).
2. Start it again from the Nuxeo IDE interface.
3. Refresh the Nuxeo Studio Panel in Nuxeo IDE interface.
4. Add your project into the deployment configuration (if you removed it).
5. Refresh the deployment server.
6. Connect as Administrator into your Nuxeo instance.
7. Create two users: user1 and user2.
8. Create a workspace and set user1 as validator.
9. Create a File and click on your button.
10. Connect as user1.
    On his Home Dashboard, you will see a task assigned on the File document.
11. Log out and connect as Administrator.
12. Modify the Workspace and add user2 into the validators list.
13. Log out and connect as user1.
    On his Home Dashboard, you will see a task assigned on the File document.

Conclusion

Next steps could be:

- Add a listener on the Workspace to add an ACE on it to grant read or write right on the workspace for the validator group.
- Implement a delegation document type that stores a missing user and a list of delegated users and add a computer group that resolves indirect group assignment through this object.

How to track the performances of your platform

Managing sizing and performance of any ECM application is a tricky job, because each application is different and many factors must be taken into account.
The Nuxeo Platform is designed to optimize performance. As a consequence, continuous performance testing is part of the Nuxeo quality assurance process. Results are based on metrics that focus on user experience, such as application response time. The outcome of this continuous, measured improvement is that the Nuxeo Platform gives rapid response times even under heavy loads, with thousands of concurrent users accessing a repository that stores millions of documents.

Performance of the Nuxeo Platform

The first step is to identify which factors do impact performance and which factors do not impact performance.

Impacting Factors

Security Policies

The typical behavior of an ECM system is that you can only view a Document if you are allowed to. The same principle applies to creating or modifying documents. However, the "Access Check" is the most factor that impacts most significantly because the system may need to check for read access on a very large number of documents.

The default security policy in Nuxeo uses ACLs (Access Control Lists). Depending on the target use cases, you may have very few ACLs (when ACLs are defined only on top containers) or a lot of ACLs (when they are defined on almost every documents). To be able to deal with both cases, Nuxeo provides several optimizations in the way ACLs are processed: for example, ACL inheritance may be pre-computed. But depending on the target use-case, the best solution is not always the same one.

In the Nuxeo Platform we allow to define custom security policies that can be based on business rules. We also provide ways to convert these business rules into queries so that checks can be done quickly on huge documents repositories.

As a security policy is clearly an impacting factor, the Nuxeo Platform provides a lot of different optimizations. You can then choose the one that fits your needs.

Presentation Layer

The presentation layer is very often the bottleneck of an ECM web application.

It is easy to make mistakes in the display logic (adding costful tests, fetching too much data ...) that can slow down the application. This is particularly true when using JSF, but even when you use another presentation technology, it is possible to impact performance by wrongly modifying some templates.

The good news is that Nuxeo's default templates are well tested. However, when modifying Nuxeo's template or add a new custom one, web developers must be aware of performance issues:

- you don't want to have a round trip to database inside a display loop (that's what prefetch is done for),
- you don't want a costful business test to be done 20 times per page (that's what Seam context is made for),
- you don't want a single page listing 100 000 documents (because there is no user able to use it and that the browser won't be happy),
- ...

This may seem obvious, but in most cases you can solve performance issues just by profiling and slightly modifying a few display templates.

On this page

Document Types

A very common task in an ECM project is to define your own Document Types. In most cases it will have little or no impact on performance.

However, if you define documents with a lot of meta-data (some people have several hundred meta-data elements) or if you define very complex schema (like nesting complex types on 4 levels), this can have impact on:

- the database : because queries will be more complex,
- the display layer : because correctly configuring prefetch will be very important.

Number of Documents

As expected, the number of documents in the repository has an impact on performance:

- impact on database size, and as a consequence on the database performance,
- impact on ACLs management,
- possible impacts on UI listings.

This is a natural impact and you cannot exclude this factor when doing capacity planning.

The good news is that Nuxeo's document repository has been tested successfully with several millions of documents with a single server.

Concurrent Requests

The raw performance of the platform is not tied to a number of users but to a number of concurrent requests: 10 hyperactive users may load the platform more than 100 inactive users.

In terms of modeling the users activity, think in terms of Transaction/s or Request/s: concurrent users is usually too vague.

Factors That Have Little or no Impact

Size of the Files

When using Nuxeo's repository, the actual size of the binary files you store does not directly impact the performance of the repository. Since the binary files are stored in a Binary Store on the file system and not in the Database, impact will be limited to Disk I/O and upload/download time.

Regarding binary file size, the only impacting factor is the size of the full-text content because it will impact the size of the full-text index. But in most cases, big files (images, video, archives ...) don't have a big full-text content.

Average Number of Documents per Folder

A common question is about the number of documents that can be stored in a Folder node. When you use Nuxeo's VCS repository, this has no impact on the performance: you can have folders with several thousands of child documents.

When designing your main filing plan, the key question should be more about security management, because your hierarchy will have an impact on how ACLs are inherited.

Some Generic Tips for Tuning the Nuxeo Platform

Independent from use cases, some technical factors have an impact on performance:

Application Server

The Nuxeo Platform is available on Tomcat and JBoss servers. Tomcat tends to have better raw performance than JBoss.

Tomcat HTTP and AJP connector configuration impact the behavior of the server on load, limiting the `maxThread` value to prevent the server from being overloaded and to keep constant throughput.

Under load the JBoss JTA object store can generate lots of write operations even for read-only access. A simple workaround can be to use a ramdisk for the `server/default/data/tx-object-store` folder.

Note also that the default maximum pool size for the AJP connector on JBoss is only 40, which can quickly become a bottleneck if there is no static cache on the frontal HTTP server.

JVM Tuning

Always use the latest 1.6 JDKs, they contain performance optimizations.

Log Level

Log level must be set to INFO or WARN to reduce CPU and disk writes.

Database

Database choice has a large impact on performance.

PostgreSQL has more Nuxeo optimizations than other databases. It is the preferred database platform.

Tuning is not optional, as Nuxeo does not provide default database configurations for production.

Network

The network between the application and the database has an impact on performance.

Especially on a page that manipulates many documents and that generates lots of micro JDBC round trips.

Our advice is to use a Gigabit Ethernet connection and check that any router/firewall or IDS don't penalize the traffic.

Here are some example of the command *ping -s PACKETSIZE* in the same network (MTU 1500) that can give you an idea of the latency added to each JDBC round trip:

Ping packet size	Fast Ethernet (ms)	Gigabit Ethernet (ms)	ratio
default	0.310	0.167	1.8562874
4096	1.216	0.271	4.4870849
8192	1.895	0.313	6.0543131

While the database will process a simple request in less than 0.05ms most of the JDBC time will be spend on the network

from 0.3ms on Gigabit Ethernet to 1.9ms on Fast Ethernet (6 times more).

Note that you can check your network configuration using the ethtool command line.

If you have a firewall or your database don't reply to ICMP ping, you can test the network latency using a tool like jdbctester.

Also knowing your JDBC driver configuration may help, for instance Oracle by default do a round trip every 10 rows, this can be changed using the following JAVA_OPTS

```
-Doracle.jdbc.defaultRowPrefetch=50
```

How we Manage the Nuxeo Platform Performance

Now, that we have seen that managing performance involves many factors, let's see how we manage this at Nuxeo for the Platform and its modules.

A Toolbox for Benchmarking the Nuxeo Platform

We provide several tools to load test and benchmark the Platform: see the Tool chapter later in this document.

Continuous Performance Testing via CI

Benchmarking once is great, but the real challenge is to be sure to detect when performances are impacted by a modification (in the UI, in the Document Types, ...).

To do so, we use small benchmark tests that are automatically run every night by our CI chain. The test is configured to fail if the performance results are below the performance results of the previous build.

This fast bench enables to check core and UI regressions on a simple case.

- Hudson benching job
- Daily bench report
- Daily bench monitoring report
- Benching script sources

This allows us, for example, to quickly detect when a template has been wrongly modified and lets us quickly correct it before the faulty changeset becomes hidden by hundreds of other modifications.

Periodic Benchmark Campaigns

Every 2 or 3 months, we run major benchmarking campaigns to tests the platform on the limits.

This is a great opportunity to do careful profiling and eventually introduce new database and Java optimizations.

Sizing your Nuxeo Platform-Based ECM Application

In order to correctly size your Nuxeo Platform-based ECM application, you should:

Define your Requirements

You have to define your needs and hypotheses for any factor that can impact the platform performance:

- target number of documents in the repository,
- target security policy,
- target filing plan and ACLs inheritance logic,
- target request/s.

Setup Performance Testing From the Beginning

Performance benchmarking is not something you should postpone to a pre-production phase.

It's far more efficient (and cheaper) to setup performance tests from the beginning.

Start with simple benchmark tests (based on the ones provided by Nuxeo) on a raw prototype and improve them incrementally as you improve your customization.

Using this approach will help you:

- detect a performance issue as soon as possible,
- correct small problems when they are still small,
- avoid having a lot of mistakes to correct just before going to production.

You can leverage all the standard tests we provide and also the Hudson integration if you want to use Hudson as CI chain provider.

Use Interpolation When Needed

Nuxeo provides standard benchmarks for both small and big documents repositories.

When needed, you can use these results to interpolate results from your tests.

Performance Toolbox Provided by the Nuxeo Platform

Benchmarking Tools

We use FunkLoad for performance testing. This tools enables us to produce quickly new scenarios.
Here are the main advantages:

- An http proxy recorder generates the initial bench script.
- FunkLoad comes equipped and ready with "batteries included":
    - helpers to make assertions,

    - library to generate random content,
    - library to share user credentials between threads,
    - basic monitoring.
- Scripts are done in Python which enables complex scenario implementation.
- Benches are easily automated using simple Makefile.
- FunkLoad produces a detailed report and differential report to compare two bench results.
- Nuxeo DM has a Python library to write tests with a "fluent interface pattern" like:

```
(LoginPage(self).view()
.login('Administrator', 'Administrator')
.getRootWorkspaces()
.createWorkspace('My workspace', 'Test ws')
.rights().grant('ReadWrite',  'members')
.view()
 .logout())
```

  This makes it easy to create new scenarios.

We also use Nuxeo DM addon tools like `nuxeo-platform-importer` to populate the document base.

Metrics to Monitor During a Bench

- CPU: The iowait or percent of time that CPU is idle during which the system has outstanding disk I/O request can be useful to identify an I/O bottleneck. On multi CPUs, if only one of the CPU is used at 100%, it may be the cause of an overloaded garbage collector.
- JVM Garbage Collector throughput: this is the percentage of total time of the JVM not spent in garbage collection.
- Disk utilization: to check for device saturation.
- JBoss JCA connection pool.
- SQL queries that took up most time.

Monitoring Tools

- sysstat sar for monitoring the system activity (cpu, disk, network, memory ...). Using kSar it can produce nice pdf reports.
- The JBoss LoggingMonitor service can monitor specific attributes of a MBean periodically and log its value to the filename specified.
- JVM garbage collector logging using a JAVA_OPTS.
- PosgreSQL log_min_duration to log SQL queries.
- logchart to produce miscellaneous charts from the sar output, JBoss logs, GC logs and dabatase logs.
- pgfouine the PostgreSQL log analyzer wich is used by logchart.

Example of a logchart monitoring report

More info on the Monitoring Nuxeo DM FAQ.

Nuxeo Metrics Monitoring Tools with Mbeans

In `nuxeo-runtime-management-metric`, Nuxeo provides the infrastructure that can be used to monitor use of services or class through mbeans. The mbean displays access counts on methods and the time spent on it. It can also serialize its results in XML.

As an example, we will first see how to configure and monitor access to the Nuxeo repository backend class.

Monitor Nuxeo Core Backend Access

The idea is to plug our monitor class as a proxy of the real Repository class. When a method gets through the proxy, metrics are automatically added and named with interface and method names. All metrics have an operation "Sample" that provides the metrics you are looking for.

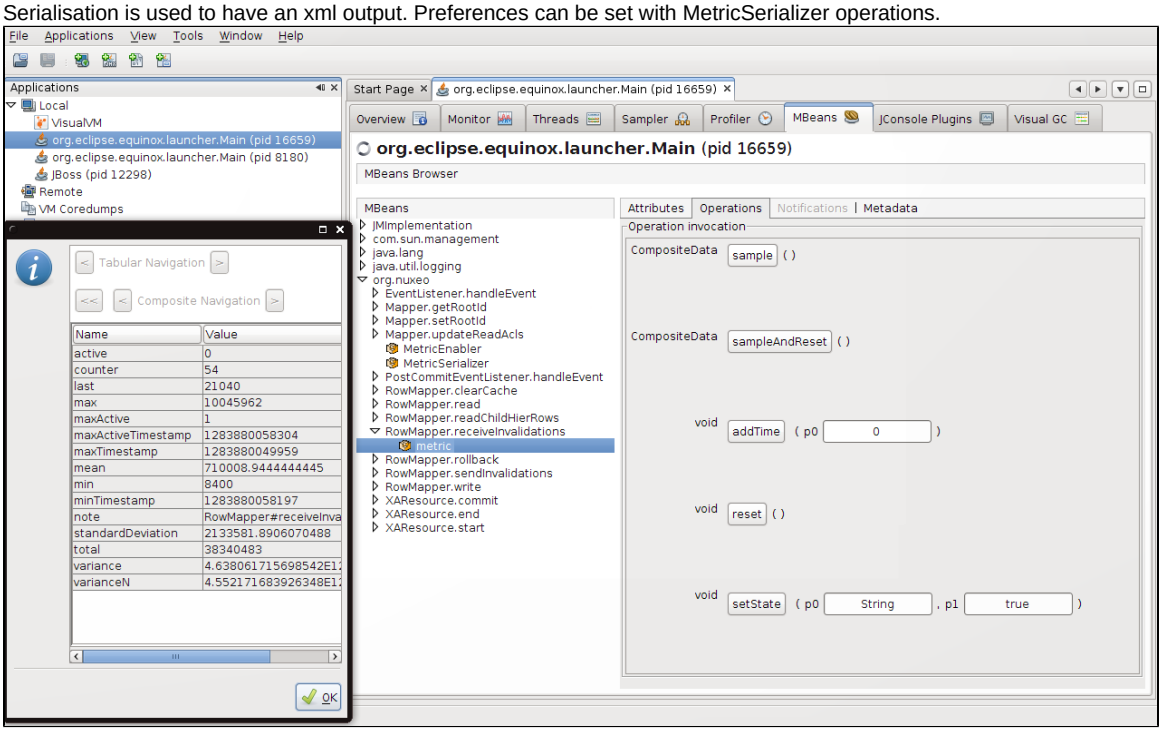
1. Modify the file `config/default-repository-config.xml`(be careful to modify the right file if you are using templates configuration system) and add this line:

```
<backendClass>org.nuxeo.ecm.core.storage.sql.management.MonitoredJDBCBackend</bac
kendClass>
```

   This class is a proxy to the real backend class. Nuxeo VCS core storage will behave exactly like before. The proxy just counts and records time spent on each method of the interface, and make it available to the mbean.

   When using VCS remote on a deported client, the class to used is `MonitoredNetBackend`.

2. To view the result, run jconsole or Visualvm.
3. Connect to your running Nuxeo repository Java process.
4. Go to the mbean tab
5. In the mbeans "org.nuxeo" you will find all the metrics. MetricEnable contains operations to enable/disable logging and serialisation. Serialisation is used to have an xml output. Preferences can be set with MetricSerializer operations.



Create your Own monitored Proxy

The previous example had its proxy class available in the Nuxeo Platform and the backend class could easily be replaced by modifying an extension point. However, creating a new proxy class is still easy. Let's try adding a monitor proxy to all the listener to monitor Listener access:

Listener objects are created in `EventListenerDescriptor: initListener`.

The idea is to create the proxy with `MetricInvocationHandler.newProxy` and provide the instance to proxy and the Interface class to monitor.

The proxy will replace the original instance:

```
public void initListener() throws Exception {
        if (clazz != null) {
            if (EventListener.class.isAssignableFrom(clazz)) {
                inLineListener = (EventListener) clazz.newInstance();
                inLineListener = MetricInvocationHandler.newProxy(
                        inLineListener, EventListener.class);
                isPostCommit = false;
            } else if (PostCommitEventListener.class.isAssignableFrom(clazz)) {
                postCommitEventListener = (PostCommitEventListener)
clazz.newInstance();
                postCommitEventListener = MetricInvocationHandler.newProxy(
                        postCommitEventListener, PostCommitEventListener.class);
                isPostCommit = true;
            }
```

Restarting the repository and accessing to the proxy will make the class monitored in the monitoring tool.

Some Example Benchmark Results

Goals

Demonstrate adequate response times for various document retrieval and insertion operations on a large storage of 10 million documents.

Steps

1. Tune the database following tips in the Nuxeo PostgreSQL FAQ.
2. Tune Nuxeo DM: for mass import, we disable the fulltext indexing (as described in the "Mass import specific tuning" section of PostgreSQL configuration page) and disable the ACL optimization (NXP-4524).
3. Import content: mass import is done using a multi-threaded importer to create File document with an attached text file randomly generated using a French dictionary. Only a percentage of the text file will be indexed for the full text, this ratio simulate the proportion of text in a binary format.
   Sources of the nuxeo-platform-importer
4. Rebuild fulltext as described in the "Mass import specific tuning" FAQ.
5. Generate random ACLs on documents. This can be done with a simple scripts that generate SQL inserts into the ACL table.
6. Enable the read ACLs optimization, performing the SQL command:

```
SELECT nx_rebuild_read_acls();
```

7. Enable the ACL optimization (NXP-4524).
8. Bench using the same scripts as in continuous integration for writer and reader. In addition we have a navigation bench that randomly browses folders and documents.

Results Overview

The base was successfully loaded with:

- 10 million of documents,
- 1TB of data.

Below are some average times:

- Accessing a random document using the **Nuxeo DM** web interface under load of 250 concurrent users accessing the system with 10 seconds pause between requests: 0.6s.
- Accessing a document that has already been accessed, under load: 0.2s.
- Accessing a random document or download attached file using a simple **WebEngine** application: 0.1s.
  It can handle up to 100 req/s which can be projected to at least 1000 concurrent users.
- Creating a new document using the **Nuxeo DM** web interface under load: 0.8s.

This bench showed no sign of being impaired by the data volume once the data was loaded from disk.

http://public.dev.nuxeo.com/~ben/bench-10m/

Customizing Bench

The bench procedure can be customized to validate customer installation:

- The mass importer tool can be used as a template to inject a customized document type instead of File documents.
- Scripts can be modified to have realistic scenarios.
- Scripts can be combined to create realistic loads.

Workflow How-Tos

Main how-tos are in the Nuxeo Studio section. Here we lists some programmation tips, when you want to use the Java API, which you should seldom have to do.

How to Query Workflow Objects

Workflows are stored in Nuxeo as documents of type DocumentRoute. A workflow instance is created by copy from a workflow model. The life cycle state for a workflow model is "validated", while the instances can be "running", "done" or "canceled".

A node in the workflow is stored as a document of type RouteNode. A node has an unique id generated by Studio, set on the `rnode:nodeId` property. Nodes can be automatic steps or nodes of type task. If a node is of type task, that means a task (persisted by default as a document of type TaskDoc) is created when the workflow is executing that node. The workflow will be resumed only when this task is completed.
Since workflows, nodes and task are all documents, the NXQL query language can be used, like on any other document type.

On this page

Querying Workflows

Using Life Cycle State

Query all running workflows:

```
Select * from DocumentRoute where ecm:currentLifeCycleState = 'running'
```

Query all available workflow models:

```
Select * from DocumentRoute where ecm:currentLifeCycleState = 'validated'
```

Using Workflow and Node Variables

Both workflow and node variables are persisted on the workflow and node documents. These are metadata stored on a dynamic facet, in a schema named as following:

- for workflow variables, the name of the schema (and its prefix too) is `var_$WorkflowModelName`,
- for node variables, is `var_$NodeId`.

The name of this facet is stored as the property:

- `docri:variablesFacet` on the workflow documents
- `rnode:variablesFacet` for nodes.

So both workflow and node variables can be queried as any other Nuxeo property.

Query all running default serial workflows, having the global variable "initialComment" set to "test"

```
Select * from DocumentRoute where var_SerialDocumentReview:initiatorComment = 'test'
```

Querry all running workflows for a given document

```
Select * from DocumentRoute where docri:participatingDocuments IN ('$docId') AND
ecm:currentLifeCycleState = 'running'
```

Querying Workflows Suspended at a given Step

NXQL queries can reference any metadata. Using the CoreSession#queryAndFetch API we can look for workflows suspended on a given step. This will return in an IterableQueryResult the id of the document representing the workflow document.

```
Select ecm:parentId from RouteNode  where rnode:nodeId = 'Task5237' and
ecm:currentLifeCycleState = 'suspended'
```

where 'Task5237' is the unique id of the node.

Querying Workflow Tasks

Query for all opened tasks

```
Select * from TaskDoc where ecm:currentLifeCycleState = 'opened'
```

How to modify a workflow variable outside of workflow context

A workflow (route instance) is stored in Nuxeo as a document of type "DocumentRoute". A node in the workflow is stored as a document of type "RouteNode". Both workflow and node variables are persisted on the workflow and node documents.
If you want to modify these variables outside of the workflow context (from a listener for example), you have to fetch the workflow instance document and you can use the available methods on their adapters, GraphRoute and GraphNode:

```
void setVariables(Map<String, Serializable> map);
Map<String, Serializable> getVariables();
```

Eg.

```
GraphRoute route = doc.getAdapter(GraphRoute.class);
GraphNode node = route.getNode(nodeId);
```

You can either listen to events triggered on the document following the workflow, or on workflow events.
In the first case, in the event handler you have the document following the workflow and you have to get the workflow instance document. Use the following method on the DocumentRoutingService service:

```
List<DocumentRoute> getDocumentRoutesForAttachedDocument(CoreSession session, String
attachedDocId)
```

For the second case, the `workflowTaskCompleted` event is triggered during a workflow every time a task is completed. On this event, the id of the workflow (route) instance documents is directly added into the map holding the properties of this event.
Here is some sample data fetched with my debugger on a break point on `TaskEventNotificationHelper notifyEvent` method:

```
event: workflowTaskCompleted
eventProperties:
{category=eventDocumentCategory, sessionId=default-6778825317969559609,
recipients=[Administrator, Administrator, test],
comment=szss, repositoryName=default,
taskInstance=org.nuxeo.ecm.platform.task.TaskImpl@65fff289, documentLifeCycle=project}

((org.nuxeo.ecm.platform.task.TaskImpl)eventProperties.get("taskInstance")).getVariabl
es()
taskVariables: {createdFromTaskService=true, taskNotificationTemplate=myTemplate,
document.routing.step=0be590a5-8d03-48ef-9649-a82f06d8001a,
nodeId=Taska2e, documentRepositoryName=default,
routeInstanceDocId=d05b14e4-8d60-41be-bea2-0d4063196c0b, directive=Aknowledgement,
validated=false,
documentId=5b09103d-2fe1-40de-8737-a64b49425a6e}
```

So if you want to set a workflow variable from a listener listening to `workflowTaskCompleted` event:

1. Get the `taskInstance` from the `eventProperties` map.
2. Get the workflow document (using his id = "`routeInstanceDocId`") from the `taskVariables`.
3. Adapt the document to `GraphRoute`.
4. Use `setVariables` like in the first example.

How to Complete a Workflow Task Programmatically

You might need this to implement some sort of automatic processing instead of having the user completing the task from the UI.

Basically, you have to call:

```
void endTask(CoreSession session, Task task, Map<String, Object> data,  String status)
throws ClientException;
```

on the DocumentRoutingService , where:

- task is the Task to end;
- data is a map of variables. If a variable called "comment" is contained in this map, its value will be logged by the audit service under the "workflowTaskCompleted" event.
  When a user submits the task form (configured via Studio), this map contains all the variables in the form;
  If you want to set an existing Workflow or a Node variable when completing the task you can add them into this map.
- status is the id of the button the user would have clicked to submit the task form (if the outgoing transitions of the workflow node that created the task have conditions depending on it).
  This id is the button id you specified when you configured the task form in Studio.

In your graph, you have nodes and transitions between these nodes. When the workflow enters a node, if that node is of type Task (that means is not an automatic one, like Start and Stop for example), a task is created at that step. This task is persisted as a document model, that can be adapted to the Task object. When a task is ended, the workflow is resumed. The task is holding information about the node and the workflow instance it was created from, so when the task is ended using the API above, the related workflow instance is resumed.

To fetch all the open tasks assigned to a given user, on a document use:

```
List<Task> getTaskInstances(DocumentModel dm, NuxeoPrincipal user, CoreSession
coreSssion) throws ClientException;
```

on the TaskService.

To fetch all the open tasks originating from the same node in the workflow use:

```
List<Task> getAllTaskInstances(String processId, String nodeId, CoreSession session)
throws ClientException;
```

where:

- processId is the id of the document representing the workflow instance.
- node id, is a unique identifier of that node in the workflow, generated by Studio. It's listed on the General tab, on the node popup-up when editing a workflow node in Studio.

For some detailed examples on how to use this API, you can check the JUnit tests in the GraphRouteTest.

Layouts and Widgets How-tos

The following how-tos list simple recipes to achieve layouts and widgets customizations.

- How to Add a New Widget to the Default Summary Layout — This how-to explains how to insert a new widget to the default Summary layout so that's displayed on all document Summary pages.

How to Add a New Widget to the Default Summary Layout

This how-to explains how to insert a new widget to the default Summary layout so that's displayed on all document Summary pages.

> This how-to defines how things should be done from version 5.6. For earlier versions, the whole summary layout needs to be redefined.

Since version 5.6, a new widget type has been added to display actions. It takes advantage of the fact that actions needing different kinds of rendering can now be mixed up even if they're using the same category. This widget type makes it possible to display a list of actions, but also to include some widget types rendering.

Since the default summary layout contains four widgets displaying actions, it is possible to pile up widgets in them. The available action categories are:

- `SUMMARY_PANEL_TOP` to add widgets on top of default summary (takes the whole panel width, empty by default),
- `SUMMARY_PANEL_LEFT` for left zone,
- `SUMMARY_PANEL_RIGHT` for right zone,
- `SUMMARY_PANEL_BOTTOM` for bottom zone (takes the whole panel width, empty by default).

Here is the definition of the widget referencing actions for the `SUMMARY_TOP_LEFT` category:

```
<extension target="org.nuxeo.ecm.platform.forms.layout.WebLayoutManager"
  point="widgets">

  <widget name="summary_panel_left" type="summary_current_document_custom_actions">
    <handlingLabels>true</handlingLabels>
    <labels>
      <label mode="any"></label>
    </labels>
    <properties widgetMode="any">
      <property name="category">SUMMARY_PANEL_LEFT</property>
      <property name="subStyleClass">summaryActions</property>
    </properties>
  </widget>

</extension>
```

Default widgets (presenting the document relations, status, publications, etc...) are contributed to one of these zones, with orders separated by 100 (check out the explorer to get a complete overview, beware that addons may contribute to these zones already).

Here is a sample contribution to add a widget to the left widget panel:

```
<extension target="org.nuxeo.ecm.platform.forms.layout.WebLayoutManager"
  point="widgets">

  <widget name="summary_note_text" type="richtext_with_mimetype">
    <fields>
      <field>note:note</field>
      <field>note:mime_type</field>
    </fields>
    <properties mode="view">
      <property name="translatedHtml">
        #{noteActions.translateImageLinks(field_0)}
      </property>
      <property name="cssClass">note_content_block</property>
    </properties>
  </widget>

</extension>

<extension target="org.nuxeo.ecm.platform.actions.ActionService"
  point="actions">

  <action id="summary_note_text" type="widget" order="100">
    <category>SUMMARY_PANEL_LEFT</category>
    <properties>
      <property name="widgetName">summary_note_text</property>
    </properties>
    <filter-id>hasNote</filter-id>
  </action>

</extension>
```

This contribution will add the widget named `summary_note_text` to the summary layout when current document is a note (see filter named `hasNote`).
The action order will make it possible to change the order of appearance of this new widget in comparison to other "action widgets" defined in the same category.

Localization and Translation How-Tos

Work in progress!

This sections provides how-tos to customize the localization and translation behavior of the Platform.

For the different ways to contribute translations, see the page How to translate the Nuxeo Platform.

- How to Force Locale — You can force the locale by removing other locales support. This is done in the deployment-fragment.xml file of your plugin.

How to Force Locale

You can force the locale by removing other locales support. This is done in the `deployment-fragment.xml` file of your plugin.

"Sample extract of deployment-fragment.xml replacing other bundles' contributions to faces-config.xml"

```
<require>org.nuxeo.ecm.platform.lang.ext</require>
<extension target="faces-config#APPLICATION_LOCALE" mode="replace">
  <locale-config>
    <default-locale>en_US</default-locale>
    <supported-locale>en_US</supported-locale>
  </locale-config>
  <message-bundle>messages</message-bundle>
</extension>
```

The `mode="replace"` attribute will replace all the previously contributed `<extension target="faces-config#APPLICATION_LOCALE">`.

So you have to carefully make your plugin being deployed after any other bundle contributing a `locale-config`. In the above sample, this is ensured by the `require` parameter.

This configuration applies on JSF pages and will not change the default locale on the login page which only depends on the browser configuration.

Actions and Filters How-tos

- How to Let User Set Rights on Non Folderish Documents — If you want the user to be able to set rights on non folderish document, you need to overwrite the TAB_RIGTS action contribution to use a filter that doesn't filter on Folderish facet.
- How to Remove Tabs — In Nuxeo, tabs are actions. Actions are associated with categories. For instance the top links belong to the USER_SERVICES category. The document tabs are associated with the VIEW_ACTION_LIST category.

How to Let User Set Rights on Non Folderish Documents

TODO: need to update given examples as of 5.8 (at least)

User uses tabs to interact with document. One of this tab, is the "Access Rights" tab, uner the "Manage" tab. This tab appears, by default, only for folderish document.

It is defined in `actions-contrib.xml` in `nuxeo-platform-webapp-core`:

```
<action id="TAB_RIGHTS" link="/incl/tabs/document_rights.xhtml" order="50"
  label="action.view.rights" icon="/icons/file.gif">
    <category>TAB_MANAGE_sub_tab</category>
    <filter-id>rights</filter-id>
</action>
```

and the definition of the filter is:

```
<filter id="rights">
  <rule grant="true">
    <permission>WriteSecurity</permission>
    <facet>Folderish</facet>
  </rule>
</filter>
```

If you want the user to be able to set rights on non folderish document, you need to overwrite the `TAB_RIGTS` action contribution to use a filter that doesn't filter on `Folderish` facet.

You need to create a contribution referenced in your plugin's `MANIFEST.MF` or you can add a file in `nuxeo.ear/config` folder.

An example contribution follow, you can copy it in a `actions-config.xml` file and put this file in `nuxeo.ear/config` to test:

```
<component name="com.myexample.actions.contrib">
  <!-- We want the original contribution to be read first so we can overwrite it-->
  <require>org.nuxeo.ecm.platform.actions</require>

  <extension target="org.nuxeo.ecm.platform.actions.ActionService"
    point="actions">
    <!-- remove the TAB_RIGHTS tab -->
    <action id="TAB_RIGHTS" enabled="false" />
    <!-- Our new contribution, we are not using the same filter -->
    <action id="FOLDERISH_TAB_RIGHTS" link="/incl/tabs/document_rights.xhtml"
order="50"
      label="action.view.rights" icon="/icons/file.gif">
      <category>TAB_MANAGE_sub_tab</category>
      <filter-id>non-folderish-rights</filter-id>
    </action>

  </extension>

  <extension target="org.nuxeo.ecm.platform.actions.ActionService"
    point="filters">

    <!-- Our new filter, not filtering on Folderish facet-->
    <filter id="non-folderish-rights">
      <rule grant="true">
        <permission>WriteSecurity</permission>
      </rule>
    </filter>

  </extension>

</component>
```

How to Remove Tabs

TODO: need to update given examples as of 5.8 (at least)

In Nuxeo, tabs are actions. Actions are associated with categories. For instance the top links belong to the `USER_SERVICES` category. The document tabs are associated with the `VIEW_ACTION_LIST` category.

- To remove a tab you first need to find which action is used to show the tab, finding first the category, then the tab (For the `VIEW_ACTION_LIST` tabs, it is also shown in the URL as tabId).
- then to remove it, you need to provide a contribution that lists the tab where you can disable it.

For example, if I want to remove the workflow tab:

- click on the workflow tab (in the default Nuxeo DM distribution you can see it in document types like File or Note). If you look at the URL you will see the following parameter `tabId=TAB_CONTENT_JBPM`, and so the `TAB_CONTENT_JBPM` will be the action name that you want to remove
- search the source which should result in you finding out that the `org.nuxeo.ecm.platform.jbpm.web.actions` component provides this action in the (nuxeo-platform-jbpm-web jar).

Now, to override it you can use the following contribution:

```
<component name="org.nuxeo.ecm.platform.actionsContrib">

  <require>org.nuxeo.ecm.platform.jbpm.web.actions</require>
  <extension target="org.nuxeo.ecm.platform.actions.ActionService"
    point="actions">

    <action id="TAB_CONTENT_JBPM" link="/incl/tabs/document_process.xhtml"
      enabled="false" label="action.view.review" icon="/icons/file.gif"
      order="60"/>
  </extension>

</component>
```

Note the `enable="false"` entry above.

Now it is enough to create a `actions-config.xml` file in `<NUXEO_SERVER>/nxserver/config` with this contribution and ... the workflow tab should not be showing anymore.

Theme and Style How-Tos

- How to Override a Default Style — If you'd like to override a default dynamic style, here's an how-to, taking as example the override of the "feedback message" styling. Find out how it's styled by default, in case the style is using flavors and you would like to keep this feature (as flavor variables are resolved at display time, you may not be aware that some variables are used).
- How to Add a New Style to Default Pages
- How to Declare the CSS and Javascript Resources Used in Your Templates — The CSS and JavaScript resources needed by your JSF pages can be added directly from inside your templates.
- How to Show Theme Fragment Conditionaly — A theme describes what you show and how you show it in your application. It is a set of pages, sections, cells and fragments. It is possible to show some fragment depending on some condition using perspective. A perspective is an attribute of the fragment element. If there is no perspective attribute, the fragment will be shown in all perspective. In a default Nuxeo Platform instance, there is only one perspective named default.

How to Override a Default Style

Most of the default application styling is done using "dynamic" CSS files, sometimes referring to flavor variables. These styles can be browsed on GitHub.
If you'd like to override a default dynamic style, here's an how-to, taking as example the override of the "feedback message" styling. Find out how it's styled by default, in case the style is using flavors and you would like to keep this feature (as flavor variables are resolved at display time, you may not be aware that some variables are used).

For instance, here's the default styling for the class `facesStatusMessage` defined in `messages_and_tooltips.css`:

```
.facesStatusMessage { top: 1em; right: 1em; position: fixed; z-index: 10000 }
```

Here's another example, showing a style using flavor variables:

```
.errorMessage, .errorFeedback { background-color:"warning (__FLAVOR__ background)";
border-color:"error (__FLAVOR__ border)" }
```

To override the "feedback message" styling:

1. Create a local copy of this style so that you can change it.
2. Create an XML extension point to declare it and attach it to theme pages.
   Here's a sample structure:

```
pom.xml
src
    main
        resources
            META-INF
               MANIFEST.MF
            OSGI-INF
               theme-contrib.xml
            themes
                css
                    my_project_css.css
```

The MANIFEST.MF file must reference the `theme-contrib.xml` file for it to be taken into account at deployment.
Here's an excerpt:

```
Nuxeo-Component: OSGI-INF/theme-contrib.xml
```


Do not forget to add a new line at the end of this file, otherwise manifest parsing may fail.

3. Fill the theme contribution to declare your own CSS file.
Here's an example:

```
<?xml version="1.0"?>

<component name="com.my.application.theme">

  <!-- require this contribution as it's the one declaring the original styles to
override -->
  <require>org.nuxeo.theme.nuxeo.default</require>

  <extension target="org.nuxeo.theme.styling.service" point="pages">
    <themePage name="galaxy/default">
      <styles append="true">
        <style>my_project_css</style>
      </styles>
    </themePage>
  </extension>

  <extension target="org.nuxeo.theme.styling.service" point="styles">
    <style name="my_project_css">
      <src>themes/css/my_project_css.css</src>
    </style>
  </extension>

</component>
```

Here you can see that styling is added to the "galaxy/default" theme page, which is the legacy name for the default Nuxeo theme page.

Depending on what modules or addons you install, you may want to add the given styles to other pages like "userCenter/default" (theme page for the Home tab) and "admin/default" (theme page for the Admin Center). You might want to add it to other pages of the default theme too, like "galaxy/print" or "galaxy/popup".

So you can add as many `themePage` tags as needed taking example on this one.

4. Fill the CSS file.

   Here's an example contribution to put the feedback message at the center of the screen (instead of top right, as done by the original default style):

   ```
   .facesStatusMessage {
     left: 40%;
     position: absolute;
     right: 40%;
     text-align: center;
     top: 40%;
     z-index: 1000
   }
   ```

   This file can also reuse existing flavor variables.
5. Make sure your Nuxeo plugin is correctly deployed, and your new styles will take effect!

How to Add a New Style to Default Pages

Work still in progress

You basically have two options:

- Add these classes to "static" CSS files: they won't be able to reference flavors, but can use CSS3 properties (as of Nuxeo 5.5 and 5.6, CSS3 properties management is not working well for dynamic CSS files).
- Add theses classes to "dynamic" CSS files (see above how to declare new dynamic CSS styles and bind them to the theme page).

How to Declare the CSS and Javascript Resources Used in Your Templates

The CSS and JavaScript resources needed by your JSF pages can be added directly from inside your templates.

Let us consider the following JSF page:

```
<nxthemes:composition xmlns="http://www.w3.org/1999/xhtml"
  xmlns:nxthemes="http://nuxeo.org/nxthemes"
  xmlns:ui="http://java.sun.com/jsf/facelets">

  <ui:define name="body">
    <div class="myWidget">A special widget that requires custom CSS and
Javascript.</div>
  </ui:define>

</nxthemes:composition>
```

The simplest way to include resources to the page is to add a `<nxthemes:require>` tag to your JSF template:

```
<nxthemes:composition xmlns="http://www.w3.org/1999/xhtml"
  xmlns:nxthemes="http://nuxeo.org/nxthemes"
  xmlns:ui="http://java.sun.com/jsf/facelets">

  <nxthemes:require resource="myCss.css" />
  <nxthemes:require resource="myScript.js" />

  <ui:define name="body">
    <div>A special widget that requires custom CSS and Javascript</div>
  </ui:define>

</nxthemes:composition>
```

This will load the `myCss.css` style and the `myScript.js` script and all their dependencies automatically whenever the JSF page is being displayed.

If you are using FreeMarker templates under WebEngine, the syntax is:

```
<@nxthemes_require>myCss.css</@nxthemes_require>
<@nxthemes_require>myScript.js</@nxthemes_require>
```

Resources are declared as theme contributions as usual:

```
<extension target="org.nuxeo.theme.services.ThemeService" point="resources">

    <resource name="myCss.css">
      <path>path/to/myCss.css</path>
    </resource>

    <resource name="myScript.js">
      <path>path/to/myScript.js</path>
    </resource>

</extension>
```

How to Show Theme Fragment Conditionaly

A theme describes what you show and how you show it in your application. It is a set of pages, sections, cells and fragments. It is possible to show some fragment depending on some condition using perspective. A perspective is an attribute of the fragment element. If there is no perspective attribute, the fragment will be shown in all perspective. In a default Nuxeo Platform instance, there is only one perspective named `default`.

1. First you need to register the perspective with:

   ```
   <extension target="org.nuxeo.theme.services.ThemeService"
       point="perspectives">
     <perspective name="myperspective">
       <title>perspective not to be shown all the time</title>
     </perspective>
   </extension>
   ```

   A perspective is just a name.
2. then you add a perspective attribute to your fragment in the theme configuration:

```
<!--this shows the fragment only in myperspective perspective.-->
<fragment perspectives="myperspective" type="generic fragment"/>
<!--this shows the fragment in myperspective and the default perspective. -->
<fragment perspectives="myperspective,default" type="generic fragment"/>
```

3. Finally, you add your own scheme to the perspective negociator to resolve the perspective:

```
<extension target="org.nuxeo.theme.services.ThemeService" point="negotiations">
  <negotiation object="perspective" strategy="web"> <!--strategy is web for
webengine and default for jsf-->
    <scheme>
      com.mycompany.mMnegociator  <!--here is you class-->
    </scheme>
  <!--don't forget to add the other perspective negociators -->
</extension>
```

Your class should implement the `Scheme` interface. The `getOutcome` method is passed a `WebContext` or a `FacesContext`. If you can't decide which perspective it is, return null, the query will be passed to the next negociator.

Special Pages Customization How-Tos

This chapter describes customization of special pages.

- How to Customize the Error Page — You have to override this nuxeo_error.jsp page in a custom project.
- How to Customize the Login Page — If you want to change the welcome page of Nuxeo, you just have to add the following code to the deployment-fragment.xml of your plugin's JAR. In this example, it will the redirect the user to the default Nuxeo dashboard.
- How to Define Public Pages (Viewable by Anonymous Users) — To make some pages or parts of the repository visible to people without requiring them to be authenticated to the Platform, follow the following steps.

How to Customize the Error Page

Work in progress, this page needs to be updated.

The Error Page when the server is deployed is located at `server/default/deploy/nuxeo.ear/nuxeo.war/nuxeo_error.jsp`.

In the source code, this page is located at `nuxeo-platform-webapp/src/main/resources/nuxeo.war/nuxeo_error.jsp`.
You have to override this `nuxeo_error.jsp` page in a custom project.

1. Copy this page inside your custom project in the same relative location as it is in the original Nuxeo sources, that is in a comparable `nuxeo.war` directory.
2. And then remove the following part:

```
<div class="stacktrace">
  <a href="#"
     onclick="javascript:toggleError('stackTrace'); return false;">show
stacktrace</a>
</div>
<div id="stackTrace" style="display: none;">
  <h2><%=exception_message %></h2>
  <inputTextarea rows="20" cols="100" readonly="true">
    <%=stackTrace%>
  </inputTextarea>
</div>
```

3. The `build.xml` of your custom project should contain the following target to nicely deploy your customized version of `nuxeo_error.jsp`:

```
<target name="web" description="Copy web files to a live JBoss">
  <copy todir="${deploy.dir}/${nuxeo.ear}/nuxeo.war" overwrite="true">
    <fileset dir="${basedir}/src/main/resources/nuxeo.war/" />
  </copy>
</target>
```

How to Customize the Login Page

Content is outdated, work is still in progress!

If you want to change the welcome page of Nuxeo, you just have to add the following code to the `deployment-fragment.xml` of your plugin's JAR. In this example, it will the redirect the user to the default Nuxeo dashboard.

Actually, this is not enough to redirect to the dashboard as the second parameter of the `initDomainAndFindStartupPage` method will be used only in some special cases, but declaring the page with view id `/nxstartup.xhtml` and binding it to your custom action is the way to proceed to override the default behaviour.

```
<extension target="pages#PAGES">
  <page view-id="/nxstartup.xhtml"
      action="#{startupHelper.initDomainAndFindStartupPage('Default domain',
'user_dashboard')}" />
</extension>
```

Note that the first argument of the `startupHelper.initDomainAndFindStartupPage()` method is currently not used by the default implementation and is kept only for compatibility. The second argument is a normal JSF view (actually, outcome), for instance the standard one is defined as:

```
<extension target="faces-config#NAVIGATION">
  <navigation-case>
    <from-outcome>user_dashboard</from-outcome>
    <to-view-id>/user_dashboard.xhtml</to-view-id>
    <redirect />
  </navigation-case>
</extension>
```

And the `/user_dashboard.xhtml` resource contains the JSF code to use.

How to Define Public Pages (Viewable by Anonymous Users)

To make some pages or parts of the repository visible to people without requiring them to be authenticated to the Platform, follow the following steps.

Activating the Anonymous User

The first step you need to take is to enable the anonymous user.

1. Anonymous user needs to be registered through a configuration files, that should look like this:

```
<?xml version="1.0"?>
<component name="org.nuxeo.ecm.platform.login.anonymous.config">
  <!-- Add an Anonymous user -->
  <extension target="org.nuxeo.ecm.platform.usermanager.UserService"
    point="userManager">
    <userManager>
      <users>
        <anonymousUser id="Guest">
          <property name="firstName">Guest</property>
          <property name="lastName">User</property>
        </anonymousUser>
      </users>
    </userManager>
  </extension>
</component>
```

   - This file needs to be named `what-you-want-config.xml` and to be deployed under `nxserver/config/`.
   - Or you can use template system and copy it into `template/common/config/`.
2. Restart the application server.
   After the server is up and running, when accessing Nuxeo again, the anonymous user will be used and logged into the application.

Managing Rights of the Anonymous User

The anonymous user after the configuration above is considered as a simple user with a given rights, a user workspace, etc. Managing his rights and visible actions is the same than other users.

So be careful, if you give Write access to the repository to this user, you will let anyone have access to the server to create content.

Defining the Login Page for the Anonymous User

- If you give at least Read Access to the Default Domain for the Anonymous User case, he will be redirected at the root of the Default Domain into the Nuxeo DM view.
- If you give no Right Access to the Default Domain for the Anonymous User, he will be redirected to the Home View.
  Anonymous User will have the possibility to access to his content from the Dashboard (gadgets "My Workspace", "My Documents", ...)

How to Change the Default Document Type When Importing a File in the Nuxeo Platform?

In this cookbook, importing a file corresponds to these actions:

- Using the Drag'n Drop,
- Using the "Import a file" button,
- Adding a file from a WebDAV drive.

The mechanism to create a Nuxeo document with an import is tight to the extension point plugins from the FileManager service.

According to the mimetype of the file you try to import, a specific plugin will be called. And most of the time, it's the `DefaultFileImporterp` lugin that will be used.

So, to create a document of your own type that, you have to set the `docType` attribute when overwriting the default contribution:

```
<require>org.nuxeo.ecm.platform.filemanager.service.FileManagerService.Plugins</requ
ire>

<extension target="org.nuxeo.ecm.platform.filemanager.service.FileManagerService"
point="plugins">
    <plugin name="DefaultFileImporter" merge="true" docType="MyCustomFileType" />
</extension>
```

Related topics in developer documentation

- Importing Data in Nuxeo
- Nuxeo Core Import / Export API
- Nuxeo CSV
- Drag and Drop Service for Content Capture (HTML5-Based)

Contributing to Nuxeo

Founded on the principles of open source, Nuxeo is passionate about community: the ecosystem of our community users, customers and partners who run their critical content-centric applications on our platform. Open source ensures that these external stakeholders have full visibility into not only the source code but the roadmap and ongoing commitment to standards and platforms. Customers, integrators, contributors and our own developers come together to share ideas, integrations and code to deliver value to the larger user base.

Nuxeo development is fully transparent:

- Important development choices can be followed on through the tech reports. Comments are more than welcome!
- Any commit in the code can be followed on Nuxeo GitHub repositories and on ecm-checkins @lists.nuxeo.com,
- Any evolution and bug fixing is tracked on JIRA ,
- Quality of the product development can be monitored on our Jenkins Continuous Integration site and SonarQube Quality Assurance site.

Nuxeo is always happy when someone offers to help in the improvement of the product, whether it is for documentation, testing, fixing a bug, suggesting functional improvement or contributing totally new modules. To maintain the quality of such an open development process, Nuxeo has set up a few strict rules that a Nuxeo community member should follow to be able to contribute.

There will be a "contributions portal" in the future; until then, this page explains the process that should be respected.

On this page

Before describing this process, here are a few points that are the basis of the Nuxeo development process and that should always be kept in mind.

- Any evolution in Nuxeo sources should be matched with a JIRA issue (http://jira.nuxeo.com/browse/NXP, or corresponding product NXMOB, NXIDE, NXBT...).
- Any code evolution must be documented, in the English language.
- Any new feature, even a low-level one, must be unit-tested.
- Any new feature must be implemented respecting usual Nuxeo software design, leveraging services, not putting business logic in Seam components. A bad design code could be rejected.

Translations

Nuxeo labels are stored in ASCII files. We use the UTF-8 encoding for non ASCII files (like \u00e9 for é). You have two different options. See How to translate the Nuxeo Platform for more details.

With Crowdin

Steps taken from the well written Flattr translation guide:

1. Join the Nuxeo translation group of your choice at crowdin.net/project/nuxeo. Pick a language you want to translate and start by clicking “translate”.
2. In the Crowdin translation view you will find all the phrases to translate to the left. (To view only the ones that still need translation, use the “missing translations” filter.)
3. Click on a phrase you want to translate. You see the original phrase in the top, and a box to fill out your translation beneath.
4. Enter the translation and by clicking “commit translation” commit it for approval/rating.

Without Crowdin

1. Fork nuxeo-features/nuxeo-platform-lang (French and English) and nuxeo-platform-lang-ext (all other languages) projects. See How to translate the Nuxeo Platform for more details.
2. Use any standard Java i18n tool to edit the files (postfix it with the i18n code of the language you are translating).
3. Create a contribution type ticket via JIRA NXP ticket.
4. Submit your path file attached to the JIRA issue or make a pull-request on GitHub.
5. Optionally, send an email to the nuxeo-dev mailing list with the subject "Labels contribution proposal" or a post on Nuxeo Answers or the Nuxeo Google+ community.

Documentation

Contribution is welcome both for technical (books and guides, FAQ, tutorials) and functional documentation. Ask a contributor account for http://doc.nuxeo.com on Nuxeo Answers, the nuxeo-dev mailing list or on the Nuxeo Google+ community.

Testing

Testing is always welcome, particularly when Nuxeo submits a new Fast Track version of its products. As our products are easily downloadable, it doesn't require any specific development skill.

1. Download the version you want to test, set it up.
2. Get and read the user guide for the selected distribution and modules (Nuxeo DM, Nuxeo DAM, Nuxeo CMF).
3. For any bug you detect, ask for a confirmation on Nuxeo Answers, create a JIRA ticket, specifying the version of the product, the environment (OS, browser, ...), the conditions and the reproduction steps. Before each release every ticket is read and depending on its severity, fixed before the release or postponed.

Improvements and Bug Fixes

Improving a module is always welcome and is carefully managed by Nuxeo developers. Process is through a JIRA "Contribution" ticket and GitHub. Depending on the nature of your changes, you might be asked to sign and return the Contributor Agreement. This is mandatory for everything that isn't minor improvement or bugfix. You may get credentials to commit directly when you get used to submitting pull requests and that those one respect the framework logic and quality rules.

1. Create a JIRA "Contribution" ticket that will hold a description of your improvements, functionally and technically.
2. Send an email to the nuxeo-dev mailing list, or post on the Nuxeo Google+ community, to notify the community as well as Nuxeo developers.
3. Nuxeo will approve your specifications (or ask you some more information/change) and will give you recommendations. The JIRA issue will be in "specApproved" state.
4. Read the Coding and design guidelines.
5. Fork the project on GitHub.
6. Do your modifications in a new branch named "FEATURE-the_Jira_issue-a_short_description", respecting the coding and design guidelines. Be sure it doesn't break existing unit tests.
7. Send a pull-request.
8. In JIRA, set the ticket to "devReview" state and give a link to your pull request.
9. Finally, we can ask for some changes, putting comments on your code, then your branch will be merged by a Nuxeo developer.

New Modules

Nuxeo is highly modularized and as a consequence, it is totally possible to develop a new feature that will be deeply mixed with existing interface. Our main recommendation, among respecting coding rules and design, is to respect the usual code layout: core, API, facade, web, … If you have

such a project, Nuxeo will be glad to help you designing your module, and to provide a GitHub repository, aside a web page (Wiki) and a JIRA project for the visibility of your development.

1. Start by an introductory email in the mailing list, explaining purpose of the new module you want to develop (BEFORE developing it) and how you think of doing it or how you did it (although it is always better to contact the list before).
2. After a few exchanges in the mailing list, return the Contributor Agreement signed. Nuxeo will then add you to the GitHub organization and give you rights to commit in a new GitHub repository.
3. Read and respect the Coding and design guidelines.
4. Commit your development regularly (meaning don't wait to finish everything: on the contrary commit each of your developments on a very atomic mode, mentioning purpose of your commit in JIRA (take it as an advice more than a rule).
5. Unit tests are mandatory and Test Driven Development is strongly encouraged. Functional tests could also be integrated. We'll put your module under continuous integration, if the quality of the code respects Nuxeo criteria.
6. You can ask for a code review in the nuxeo-dev mailing list.
7. Package your plugin as a Nuxeo Package, if you want it to be on Nuxeo Marketplace Plus it will be much easier for people to install it.

In addition to code conventions and development good practices above-mentioned, wen creating a new module you should also take the following recommendations into considerations:

- Align your code on a recent released version or on the latest development version.
- Provide a **clean POM** (well indented, no duplication, inheriting nuxeo-ecm POM, ...).
- If needed, provide a list of the artifacts (libraries) or Public Maven repositories that should be added to the Nuxeo Maven repository to be able to build.
- Avoid embedded libraries.
- Avoid introducing new libraries if the equivalent already exists in Nuxeo.

Allergic to GitHub?

If you are allergic to GitHub, you can still contribute patches:

1. Create a JIRA ticket of type "Contribution" describing the problem and what you plan to do (or what you did, if it comes after).
2. Send an email to the nuxeo-dev mailing list, or post on the Nuxeo Google+ community, to notify the community.
3. Read the Coding and design guidelines.
4. Fork the "master" branch of the sub-project you want to patch.
5. Make your modifications, respecting the coding and design guidelines, and check that they don't break existing unit tests.
6. Create a patch file and attach it to the JIRA ticket you created.
7. Send an email to nuxeo-dev mailing list to notify the community of your contribution.
8. The patch will either be validated or you will receive feedback with guidance to complete it.
9. The patch will be committed by a Nuxeo developer.

Contributor Agreement

Click here to download the Nuxeo Contributor Agreement (PDF).

For small patches and minimal changes, one doesn't need a contributor agreement, as it cannot be considered original work with a separate license and copyright. The contributor agreement is for folks who contribute non-trivial amounts of code (at least one new file for instance).

Is source code needed?

Getting the Nuxeo source code is not needed to create applications on top of it. You should be able to create your own customizations of the Nuxeo Platform or default applications (DM, DAM, etc.) by creating extensions (also known as plugins), without changing the Nuxeo code.

The advantages of this approach are substantial:

- by running stock Nuxeo EP code + your own customization, it's much easier to pinpoint issues when then happen, and facilitates greatly the support process,
- upgrading to a newer version of Nuxeo EP is much easier,
- you don't need to understand the detailed internal and low level architecture of the Nuxeo Platform, only the big picture + the API you will need for your project,
- if forces the Nuxeo developers to think about extensibility, leading to a cleaner architecture.

Of course, if you are an advanced developer, you may want to join the Nuxeo community as a contributor, in which case you will want to look at the Nuxeo Core Developer Guide, in particular its Getting the Nuxeo Source Code chapter.

You may also want to have a look at this guide if, as a technology evaluator, you want to get a clearer picture of how Nuxeo EP is developed and what processes are in place to ensure the highest quality level.

How to translate the Nuxeo Platform

With Crowdin

There is a Nuxeo project on Crowdin. To join the project, create an account on Crowdin and follow this link: http://crowdin.net/project/nuxeo/invite.

If you are not familiar with Crowdin, take a look at their quick start guide.

Basically here are the steps to contribute:

1. Join the Nuxeo translation group of your choice at crowdin.net/project/nuxeo. Pick a language you want to translate and start by clicking “translate”.
2. In the Crowdin translation view you will find all the phrases to translate to the left. (To view only the ones that still need translation, use the “missing translations” filter.)
3. Click on a phrase you want to translate. You see the original phrase in the top, and a box to fill out your translation beneath.
4. Enter the translation and by clicking “commit translation” commit it for approval/rating.

By default, new users have the translator status. It means you can propose new translations, but can't validate them. To gain the proofreader status and have the validation permission, please ask for it on Nuxeo Answers or on the dev list.

Now here's a couple of tips taken from the Flattr translation guide.

- **Some phrases have placeholders** in them (like %s). These are placeholders for values that are added later. E.g. a name: “Hello, %s”, could be “Hello, Peter” on the website. They must remain in the translation as well, although placement may vary.
- **If the context for the phrase is unclear**, you can request the context from the Nuxeo team. These comments are also visible for anyone else that translates or approves translations.

Before every release, the translations are exported from Crowdin to the nuxeo-platform-lang-ext addon by Nuxeo.

Without Crowdin

Update or Create Localized Messages

You will find French and English language resources files in `nuxeo-platform-lang/src/main/resources/nuxeo.war/WEB-INF/classes` living in https://github.com/nuxeo/nuxeo-features GitHub repository.

Other language translations live in https://github.com/nuxeo/nuxeo-platform-lang-ext

To edit a language file, we recommend using either:

- a standalone resource editor: http://resourcebundleeditor.dev.java.net/
- an Eclipse plugin: http://sourceforge.net/projects/eclipse-rbe

Properties files should only contains ASCII characters. Use the code point directly for non ASCII char. For instance the char 'é' should be '\u00e9'. Take a look at this table for the details.

You should create your translations from `messages_en.properties` or `messages_fr.properties` since only those are maintained by Nuxeo developers. Make sure your new file uses a four letter code. Two letter code are only used as fallback for dialects ad will be handled directly by Nuxeo. The switch to a four letter code has been initiated since we started using Crowdin to manage community translation.

Once you have added your new properties file, you need to modify the `deployment-fragment.xml` file adding a new entry with your locale:

```
<extension target="faces-config#APPLICATION_LOCALE">
  <locale-config>
    <default-locale>en</default-locale>
    <supported-locale>en_GB</supported-locale>
    <supported-locale>en_US</supported-locale>
    <supported-locale>fr</supported-locale>
    <supported-locale>fr_FR</supported-locale>
    <supported-locale>ar_SA</supported-locale>    <!-- Arabic - Saudi Arabia -->
    <supported-locale>ca_ES</supported-locale>    <!-- Catalan - Spain -->
    <supported-locale>cs_CZ</supported-locale>    <!-- Czech - Czech Republic -->
    <supported-locale>zh_CN</supported-locale>    <!-- Chinese (Simplified) - China
-->
    <supported-locale>de</supported-locale>       <!-- German - Germany -->
    <supported-locale>de_DE</supported-locale>    <!-- German - Germany -->
    <supported-locale>el_GR</supported-locale>    <!-- Greek - Greece -->
    <supported-locale>es</supported-locale>       <!-- Spanish - Spain -->
    <supported-locale>es_ES</supported-locale>    <!-- Spanish - Spain -->
    <supported-locale>eu_ES</supported-locale>    <!-- Basque -->
    <supported-locale>fr_CA</supported-locale>    <!-- French - Canada -->
    <supported-locale>gl_ES</supported-locale>    <!-- Galician -->
    <supported-locale>it_IT</supported-locale>    <!-- Italian - Italy -->
    <supported-locale>ja_JP</supported-locale>    <!-- Japanese (Gregorian calendar) -
Japan -->
    <supported-locale>nl</supported-locale>       <!-- Dutch - Netherlands -->
    <supported-locale>nl_NL</supported-locale>    <!-- Dutch - Netherlands -->
    <supported-locale>pl</supported-locale>       <!-- Polish - Poland -->
    <supported-locale>pl_PL</supported-locale>    <!-- Polish - Poland -->
    <supported-locale>pt</supported-locale>       <!-- Portuguese - Portugal -->
    <supported-locale>pt_PT</supported-locale>    <!-- Portuguese - Portugal -->
    <supported-locale>pt_BR</supported-locale>    <!-- Portuguese - Brazil -->
    <supported-locale>ru</supported-locale>       <!-- Russian - Russia -->
    <supported-locale>ru_RU</supported-locale>    <!-- Russian - Russia -->
    <supported-locale>sr_RS</supported-locale>    <!-- Serbian(Cyrillic) - Serbia -->
    <supported-locale>vi_VN</supported-locale>    <!-- Vietnamese - Vietnam -->
  </locale-config>
</extension>
```

For Nuxeo versions before 5.4.3, extension target is "faces-config#APPLICATION"

Build and Test

To test, build with Maven the modified `nuxeo-platform-lang` or(and) `nuxeo-platform-lang-ext` module(s) and copy it(them) into the bundles directory (`$NUXEO_HOME/server/default/deploy/nuxeo.ear/bundles/` for JBoss or `$NUXEO_HOME/nxserver/bundles/` for Tomcat).

Contribute

Please contribute your translation work back to Nuxeo so that other users can benefit from, and improve upon, your work.

Login Page

By default, language is defined following the OS or browser locale settings. However, the language used after login may be forced using the language selector.

If you contribute a new language to Nuxeo, we'll update the login page.

If you need to temporarily patch it for testing purpose, the `login.jsp` page is located in `nuxeo-platform-webapp/src/main/resources/web/nuxeo.war/` living in https://github.com/nuxeo/nuxeo-dm repository.

If you want to permanently overwrite the default login.jsp page with your own, the good practice is to create a plugin for Nuxeo rather than editing Nuxeo source code.