



Developer Documentation

Nuxeo Platform 5.5

Table of Contents

1. Technical Documentation Center	4
1.1 Overview and Architecture	5
1.2 Overview	6
1.3 Architecture	9
1.3.1 Architecture overview	9
1.3.2 About the content repository	12
1.3.2.1 VCS Architecture	21
1.3.3 Platform features quick overview	30
1.3.4 Component model overview	32
1.3.5 API and connectors	36
1.3.6 UI frameworks	39
1.3.7 Deployment options	42
1.3.8 Performance management for the Nuxeo Platform	47
1.4 Customization and Development	53
1.4.1 Learning to customize Nuxeo EP	54
1.4.2 Document types	55
1.4.3 Document, form and listing views	62
1.4.3.1 Layouts (forms and views)	62
1.4.3.1.1 Manage layouts	63
1.4.3.1.2 Document layouts	70
1.4.3.1.3 Layout display	71
1.4.3.1.4 Standard widget types	72
1.4.3.1.5 Custom templates	73
1.4.3.1.6 Custom widget types	85
1.4.3.1.7 Generic layout usage	93
1.4.3.2 Content views	93
1.4.3.2.1 Custom Page Providers	100
1.4.3.2.2 Page Providers without Content Views	101
1.4.3.3 Views on documents	102
1.4.4 Versioning	104
1.4.5 User Actions (links, buttons, icons, tabs)	105
1.4.6 Events and Listeners	108
1.4.6.1 Scheduling periodic events	113
1.4.7 Tagging	115
1.4.8 Directories and Vocabularies	116
1.4.9 Adding custom LDAP fields to the UI	120
1.4.10 Authentication	123
1.4.11 User Management	132
1.4.12 Publisher service	145
1.4.13 Querying and Searching	149
1.4.13.1 NXQL	149
1.4.13.2 Fulltext queries	154
1.4.13.3 Advanced search	155
1.4.13.4 Faceted Search	160
1.4.13.5 Smart search	169
1.4.14 Security Policy Service	172
1.4.15 Theme	175
1.4.15.1 Migrating my customized theme	181
1.4.16 Nuxeo UI Frameworks	185
1.4.16.1 Seam and JSF webapp overview	185
1.4.16.1.1 JSF tips and howtos	186
1.4.16.2 GWT Integration	190
1.4.16.3 Extending The Shell	198
1.4.16.3.1 Shell Features	198
1.4.16.3.2 Shell Commands	201
1.4.16.3.3 Shell Namespaces	207
1.4.16.3.4 Shell Documentation	208
1.4.16.4 WebEngine (JAX-RS)	210
1.4.16.4.1 Session And Transaction Management	223
1.4.16.4.2 WebEngine Tutorials	226
1.4.16.5 Nuxeo Android Connector	262
1.4.16.5.1 Nuxeo Automation client	263
1.4.16.5.2 Android Connector and Caching	264
1.4.16.5.3 Android Connector additional Services	265
1.4.16.5.4 DocumentProviders in Android Connector	266
1.4.16.5.5 Android SDK Integration	269
1.4.16.5.6 Nuxeo Layout in Android	271

1.4.16.5.7 SDK provided base classes	271
1.4.16.6 Nuxeo Flex Connector	272
1.4.16.6.1 AMF Mapping in Nuxeo	273
1.4.16.6.2 Build and deploy Nuxeo Flex Connect	275
1.4.16.6.3 Using Flex Connector	278
1.4.17 Extending Nuxeo	281
1.4.17.1 The Nuxeo Plugin model	282
1.4.17.2 Adding a new UI block in Nuxeo DM	282
1.4.17.3 Making a connector	282
1.4.17.4 Integrating with JPA	282
1.4.18 Updating your application using the Admin Center (web interface)	284
1.4.18.1 Register your Nuxeo instance	288
1.4.18.2 Install a new package on your instance	291
1.4.18.2.1 Semantic Entities Installation and Configuration	294
1.4.18.3 Uninstall a package	296
1.4.18.4 Update your instance with Studio configuration	297
1.4.19 Using Nuxeo as a service platform	298
1.4.19.1 ECM Services in Nuxeo EP	300
1.4.19.2 WebServices	301
1.4.19.2.1 Building a SOAP-based WebService client in Nuxeo	301
1.4.19.2.2 Building a SOAP based WebService in Nuxeo	303
1.4.19.2.3 Trust Store and Key Store Configuration	306
1.4.19.3 Content Automation	309
1.4.19.3.1 Operations Index	312
1.4.19.3.2 Contributing an Operation	342
1.4.19.3.3 Contributing an Operation Chain	347
1.4.19.3.4 REST API	348
1.4.19.3.5 REST Operation Filters	357
1.4.19.3.6 Using the REST API - Examples	358
1.4.19.4 OpenSocial, OAuth and Nuxeo EP	372
1.4.19.4.1 OpenSocial configuration	379
1.4.19.5 Repository access	379
1.4.19.5.1 CMIS for Nuxeo	380
1.4.19.5.2 WebDAV	383
1.4.19.5.3 WSS before Nuxeo 5.4.2	384
1.4.20 Packaging	387
1.4.20.1 Packaging from sources	387
1.4.20.2 Packaging a Nuxeo plugin	388
1.4.21 Navigation URLs	392
1.4.21.1 URLs for files	395
1.4.22 Drag and Drop Service for Content Capture (HTML5-based)	396
1.5 Dev Cookbook	399
1.5.1 How-to create an empty bundle	399
1.5.2 How-to implement an Action	408
1.5.3 How-to contribute a simple configuration in Nuxeo	415
1.5.4 How-to configure document types, actions and operation chains	418
1.5.5 How to remove the Language Selector	437
1.6 Contributing to Nuxeo	437
1.6.1 Is source code needed?	440
1.6.2 How to translate Nuxeo DM	440
1.6.3 Creating Packages for the Marketplace	441
1.6.3.1 Package Manifest	444
1.6.3.2 Scripting Commands	445
1.6.3.3 Package Web Page	451
1.6.3.4 Wizard Forms	451
1.6.3.5 Package Example	451
1.7 Nuxeo Distributions	453
1.7.1 Available installers	454

Technical Documentation Center

Technical documentation for Nuxeo Platform 5.5

Welcome to the Technical documentation

Center of Nuxeo applications

Overview and Architecture

Documentation presenting Nuxeo Enterprise Platform architecture.

Using Nuxeo as a service platform

Documentation about the services provided by Nuxeo Platform

Extending Nuxeo

Documentation about the extension capabilities offered by Nuxeo Enterprise Platform.

Developer Guide

Resources for developers that want to use or extend Nuxeo.

Installation and Administration Guide

Documentation about installing and administering a Nuxeo Server.

Nuxeo UI Frameworks

Resources about Nuxeo UI technologies

Nuxeo Distributions

Documentation about Nuxeo distributions and associated tools.

Download



[Download this documentation in PDF.](#)



License

This documentation is copyrighted by Nuxeo and published under the Creative Common BY-SA license. More details on the [Nuxeo documentation license](#) page.

Recently updated

Recently Updated

-  [Technical Documentation Center](#)
 - updated about 6 hours ago • [view change](#)
-  [How-to contribute a simple configuration in Nuxeo](#)
 - updated Dec 07, 2015 • [view change](#)
-  [How-to contribute a simple configuration in Nuxeo](#)
 - updated Jan 30, 2015 • [view change](#)
-  [How-to implement an Action](#)
 - updated Jan 26, 2015 • [view change](#)
-  [Contributing an Operation](#)
 - updated Jan 26, 2015 • [view change](#)
-  [CMIS for Nuxeo](#)
 - updated Jul 28, 2014 • [view change](#)
-  [Ajax forms and actions](#)
 - updated Jan 07, 2014 • [view change](#)
-  [Contributing to Nuxeo](#)
 - updated Oct 16, 2013 • [view change](#)
-  [Ajax forms and actions](#)
 - updated Jul 15, 2013 • [view change](#)
-  [Content Automation](#)
 - updated Jul 01, 2013 • [view change](#)

Overview and Architecture

This chapter presents an overview of Nuxeo Enterprise Platform. This overview also applies to the applications based on Nuxeo EP, such as [Nuxeo Document Management \(DM\)](#), [Nuxeo Digital Asset Management \(DAM\)](#) or [Nuxeo Case Management Framework \(CMF\)](#).

You should read this if:

- you want to understand Nuxeo application's architecture,
- you want to extend Nuxeo EP or a Nuxeo EP-based application,
- you want to integrate Nuxeo EP or a Nuxeo EP-based application into your existing applications.

This chapter presents the topics below:

Nuxeo EP - a platform for ECM

A quick overview of Nuxeo EP concepts

Architecture overview

Main concepts on Nuxeo architecture

Platform features quick overview

Presentation of the main features available in Nuxeo EP

About the content repository

Quick overview of Nuxeo document repository and associated features

UI frameworks

Overview of UI frameworks integrated inside Nuxeo EP

Component model overview

Introduction to the Nuxeo Component and Service model

API and connectors

Presents how you can integrate Nuxeo with external applications.

Deployment options

Presents the different configurations that are available for Nuxeo EP deployment.

Performance management for the Nuxeo Platform

Presents how we management performance tuning and testing for Nuxeo EP.

Overview

Nuxeo Enterprise Platform (Nuxeo EP) is a platform for building ECM applications. It provides the technical infrastructure and the high level services needed to build customized ECM applications.

In order to be flexible and easily adaptable, Nuxeo EP is composed of small software parts that can be assembled together to build an application. These software parts (called "Bundles" in Nuxeo) contain:

- **Services:** to provide the features needed to manage your information;
- **Components:** to implement, configure and extend the services;
- **Presentation frameworks and UI building blocks:** to provide the user interface on top of the service stack.

You can get more details by reading the [Component model overview](#) page.

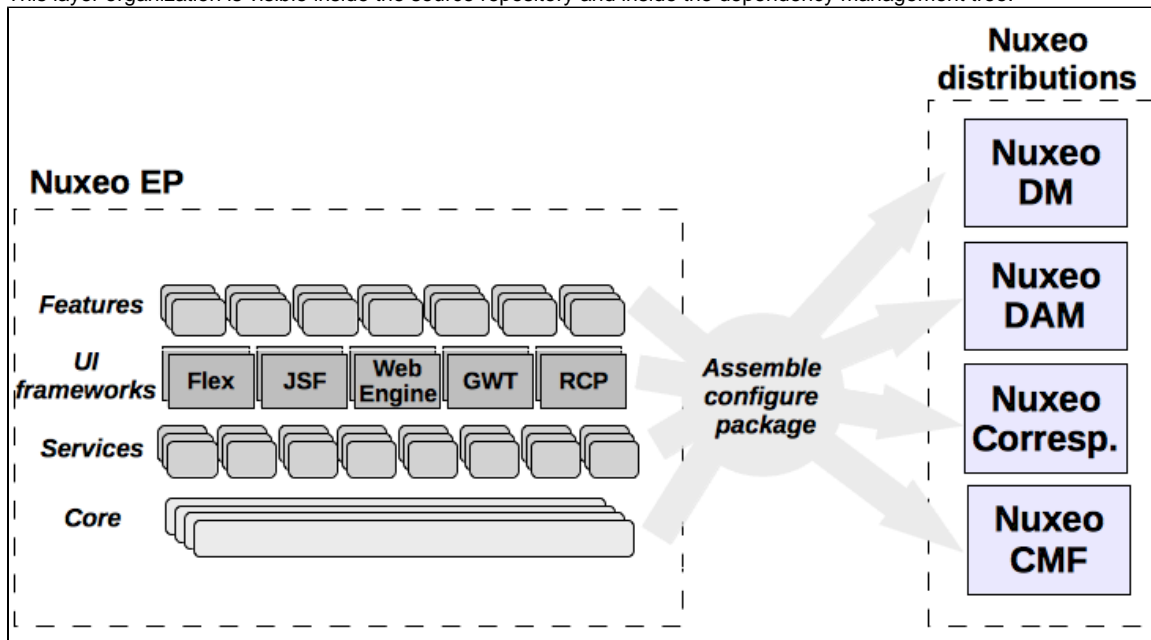
One of the key features of our platform is that most components are configurable and extensible. This gives you full control to extend and configure the existing services in order to fulfill the specific needs of your projects.

Assembling building blocks from the platform

Inside the platform, we organize the software parts into several families, or layers:

- the **Core** layer defines all low level components and services that deal with Document storage; (Document repository, Security service, Life Cycle management, ...)
- the **Services** layer includes components and services that provide additional services on top of the core layer; (User management, directory management, mime-types detection, ...)
- the **UI framework** provides an integrated framework for the several presentation technologies Nuxeo EP supports; (JSF/Seam, JAX-RS/WebEngine, GWT, Flex/AMF, ...)
- the **Features** layer contains high level ECM services that also provide some default UI (Workflow engine, Subscriptions, Picture management, ...).

This layer organization is visible inside the source repository and inside the dependency management tree.



Nuxeo Distribution is the assembly tool we use to extract and packages components from Nuxeo EP into a runnable application. This tool, based on maven, takes care of the dependencies management and adds the needed default configuration so that the assembled application can easily be run.

Nuxeo Distribution provides an automated and safe way to accomplish the following tasks:

- select bundles from the platform according to your need,
- select all the required (dependent) bundles,
- associate default configuration (document types, workflows, life-cycles, ...),
- generate WAR or EAR,

- associate storage configuration (storage type, database type, ...),
- if needed generate a ready-to run package including the application server (Tomcat, JBoss ...).

Nuxeo provides several default distributions:

- **Nuxeo CAP:** Nuxeo CAP stands for Content Application Platform. Nuxeo CAP basically includes all the basic features to manage Documents and navigate inside the content repository. This distribution is used as a basis for most distributions: for example CAP contains the infrastructure used by Nuxeo DM, Nuxeo CMF and Nuxeo DAM (see below).
- **Nuxeo Core Server:** Nuxeo Core sever is a light distribution of Nuxeo EP. It mainly includes the Document Repository, the Audit service, Security and User management services, CMIS connector. This means Core Server is basically a CMIS repository.
- **Nuxeo DM:** this distribution is the historical distribution of Nuxeo EP and all features for managing Documents: navigation and search, workflows, preview and annotations, picture management, etc.
Use the [Document Management - User Guide](#) to learn more about this distribution.
- **Nuxeo DAM:** this distribution focuses on Digital Asset Management
You can consult the [Digital Asset Management User Guide](#) for more information.
- **Nuxeo Case Management Framework:** this framework enables to build applications dedicated to the manipulation and distribution of cases.
You can consult the [Nuxeo CMF User Guide](#) for more information.
- **Nuxeo Correspondence Management:** this distribution manages inbound and outbound correspondence

You can get more information about Nuxeo-Distribution by reading the [dedicated documentation](#).

Using Nuxeo Enterprise Platform

As a platform, Nuxeo EP can be used in several different ways:

- Use a pre-built application that you customize,
- Create your own distribution from Nuxeo EP,
- Integrate your application with Nuxeo EP.

Turn-Key pre-built ECM applications

We provide several pre-built ECM applications on top of Nuxeo EP: Nuxeo DM, Nuxeo DAM, Nuxeo Correspondence...

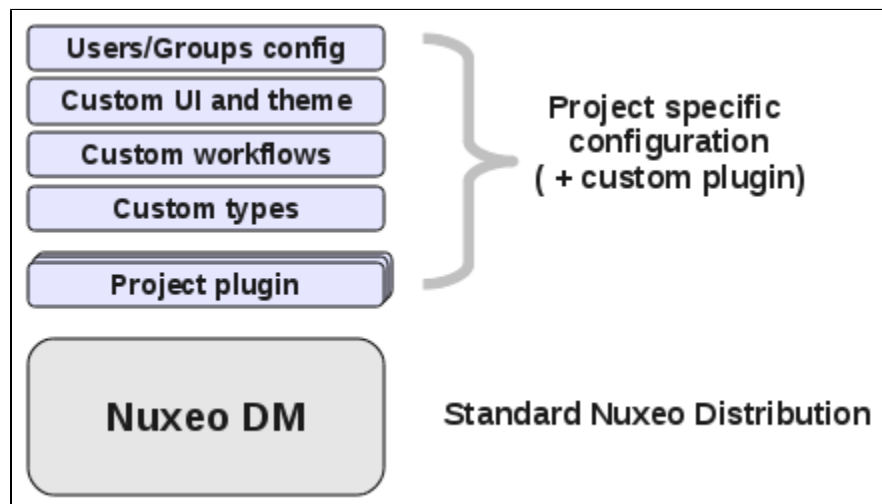
These are distributions of the Nuxeo platform that include all needed services, UI and configuration to provide a ready-to-run web application to do Document Management, Digital Asset Management, Correspondence Management...

These distributions are the best solution to quick-start testing our ECM platform.

Even if these distributions are ready to run and include a lot of default configuration, they can still be customized.

In a lot of cases, a project's implementation is built on top of one of the existing default distributions and just modifies what is needed:

- define custom Document types,
- customize workflows,
- customize security policies,
- adapt UI themes,
- add or remove buttons,
- remove unneeded features and services,
- add additional services and features.



Create a custom Nuxeo distribution

A lot of default distributions, and in particular Nuxeo DM, come with a lot of features. We choose to do so in order to make platform testing and evaluation easier. But in a lot of projects, part of the features included in DM are not needed.

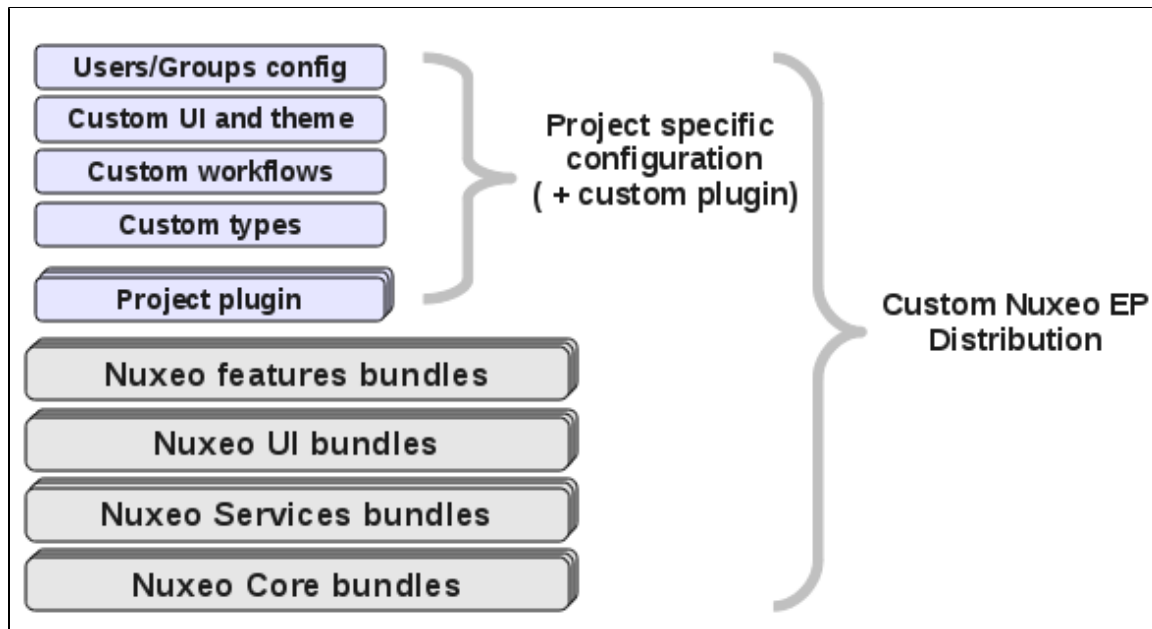
Of course, you can disable the unwanted features with some XML configuration. But in some cases, it is more efficient to build your own distribution.

Nuxeo Distribution is an open system and you can use it to build your own distribution. This allows you to precisely select the bundles you need for your project:

- don't include bundles that you don't need,
- add bundles that are not included in any default distribution,
- add project and environment specific configurations.

Building your own distribution has an other advantage: since you can assemble your ECM application server with one command line, you can easily:

- integrate it into an Integration Chain,
- have several profiles (dev, pre-prod, prod).



Integrate your application with Nuxeo EP

The third way of using Nuxeo EP is to integrate it with an existing application. Depending on your requirements, you may choose different approaches.

Deploy your application inside Nuxeo Platform

The first solution is to deploy your existing Web Application inside the Nuxeo server. Basically, this means that you simply have to package your Web Application in Nuxeo's way and your application will have full access to the entire Nuxeo EP framework (components and services).

Use Nuxeo EP as a service platform

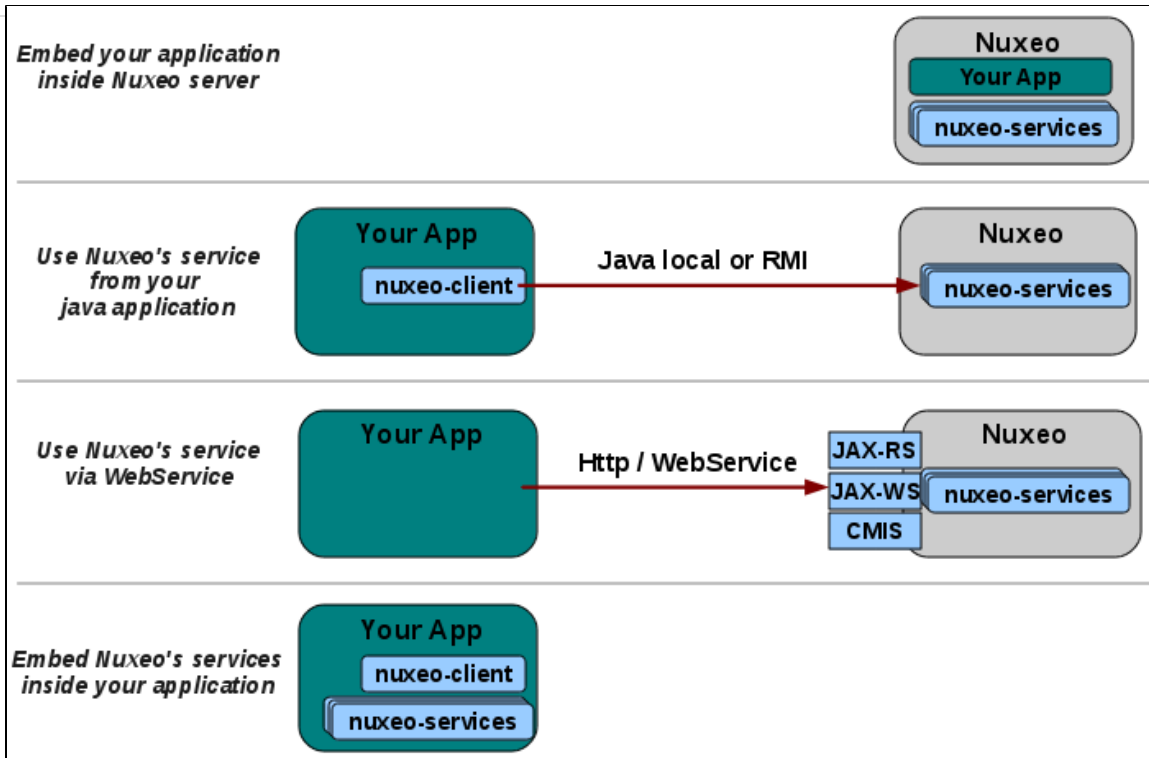
The second solution is to use Nuxeo EP as a service provider: your application uses Nuxeo's ECM services remotely. Depending on the technology you use in your application, you may use Nuxeo's services

- via Java remoting (RMI)
- or via webservices (JAX-RS or JAX-WS).

Embed Nuxeo services into your application

The last solution is to embed a part of Nuxeo EP inside your application and use Nuxeo's services as a Java Library.

For that you can use **nuxeo-client** that can deploy a minimal Nuxeo environment so that Nuxeo's services can be run. Because Nuxeo EP is very modular, you can have a very small Nuxeo stack and embed Nuxeo's services even in a very small java application.



Architecture

- [Architecture overview](#)
- [About the content repository](#)
- [Platform features quick overview](#)
- [Component model overview](#)
- [API and connectors](#)
- [UI frameworks](#)
- [Deployment options](#)
- [Performance management for the Nuxeo Platform](#)

Architecture overview

Technologies in Nuxeo EP

100% Java Based

Nuxeo EP is 100% built on top of Java: as long as a JVM (Java Virtual Machine) is available in your target environment, you should be able to run Nuxeo EP.

Nuxeo EP requires at least Java 5 (we use Generics and Annotation) and is validated against Sun JDK version 5 and 6.

POJO and JEE

The Nuxeo platform is designed so that you can run services and components in various environments:

- on bare Java,
- in a servlet container,
- in a JEE container (EAR including EJB3 bindings).

Nuxeo components and services will leverage the Java EE infrastructure if available: JTA transaction management, JAAS security, EJB3 resource pooling...

OSGi inspired component model

Nuxeo components and services are packaged like an OSGi Bundle.

Each software part of Nuxeo EP is packaged in a jar that declares:

- its dependencies,
- the components provided,
- the services provided.

Nuxeo extends the OSGi model to include needed features like service binding and extension points.

Leverage existing open source frameworks

Nuxeo integrates many existing open source frameworks. When something that works exists in open source we prefer to integrate it rather than rewrite it.

We integrate the following open source frameworks:

- JBPM: the main process engine,
- Drools: for rules management,
- Seam: used for the Web Component model in the JSF UI toolkit,
- Shindig: used as the infrastructure provider for OpenSocial,
- Jena RDF: the RDF provider.

This one is deprecated

- JackRabbit: since Nuxeo has introduced [VCS](#).

Main design goals

The main design goals of the Nuxeo EP architecture are to provide:

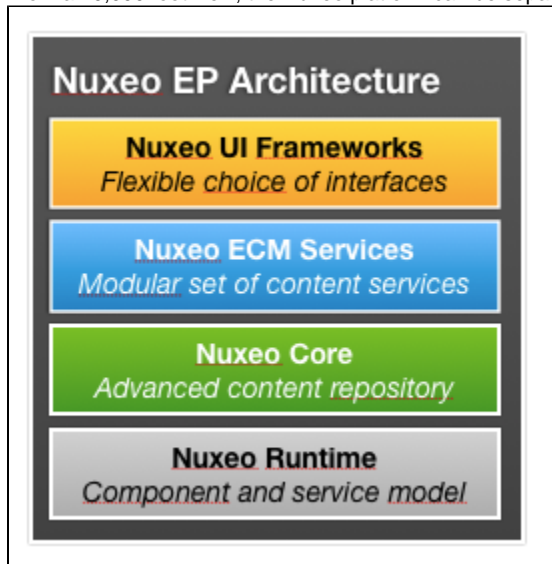
- powerful and clean solutions to configure and extend the platform,
- agility to deploy and integrate it in a complex environment.

These 2 points are the main reasons for the Bundles/Components/Service model we use.

The next paragraph will present how we achieve these goals.

10,000 foot view

From a 10,000 foot view, the Nuxeo platform can be separated in 4 layers:



Nuxeo Runtime

Nuxeo Runtime provides the component, service and deployment model used to make the platform run.

Nuxeo Runtime is based on the OSGi component model but adds some features like:

- a service lookup and binding system,
- an extension point system,
- a deployment system.

Nuxeo components and services run on top of Nuxeo Runtime. As a consequence, Nuxeo EP can run in any environment that is supported by Nuxeo Runtime.

This basically means that Nuxeo Runtime is also an abstraction layer that helps Nuxeo EP run in different environments.

Nuxeo Core

The Nuxeo Core layer contains all the services and components required to manage document storage.

Nuxeo Core can be used independently as an embeddable Document Repository.

Nuxeo Services

Nuxeo Services contains content management services and components:

- workflows,
- audit service,
- comment service,
- ...

All these services and components have some common points:

- they almost all use Nuxeo Core (because they provide services bound to Documents),
- they are designed as generic and as configurable as possible,
- they don't depend on any UI layer.

If you don't want to use the UIs provided by Nuxeo, but simply do service calls from your application, you will use the Nuxeo services layer directly.

The gadget below lists the services deployed in the DM distribution :

Error rendering macro 'gadget' : Error rendering gadget [<http://explorer.nuxeo.org/nuxeo/site/gadgets/services/services.xml>]

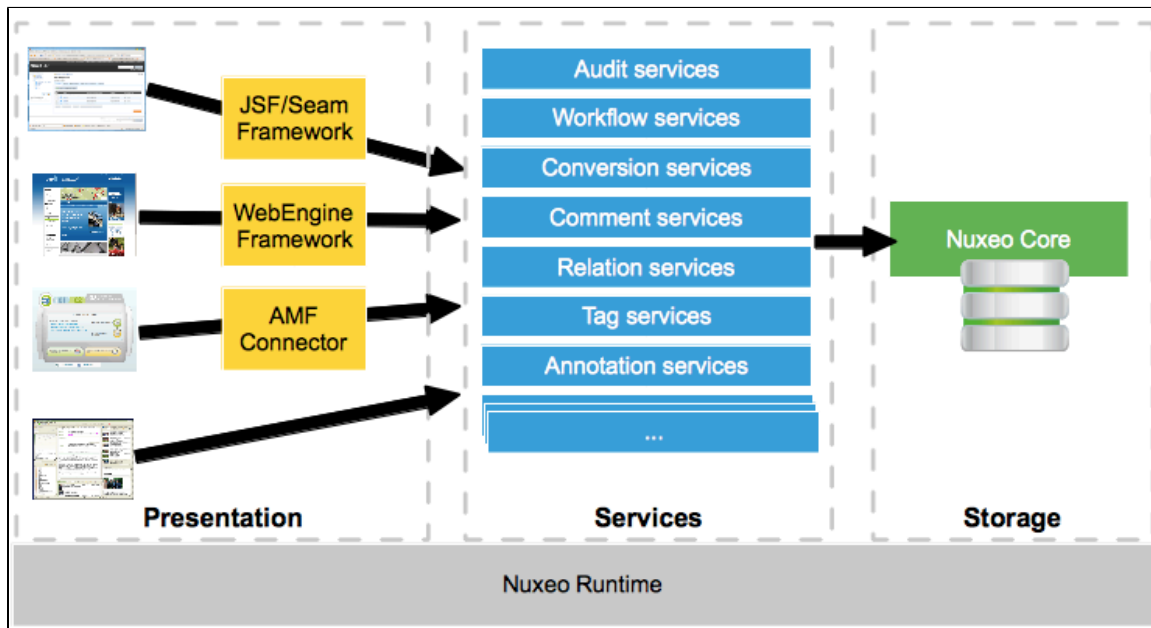
Nuxeo client technologies

On top of the service layer, Nuxeo EP provides:

- several client technology frameworks (Web 2.0, RichClient, RIA ...),
- reusable building blocks.

For example, when using JSF/Seam web technology, Nuxeo EP provides ready to use screens and buttons for most of the features and services of the platform.

This allows you to easily assemble a Web Application without having to rebuild everything from scratch or deconstruct a pre-built monolithic application.



Easy customization and integration

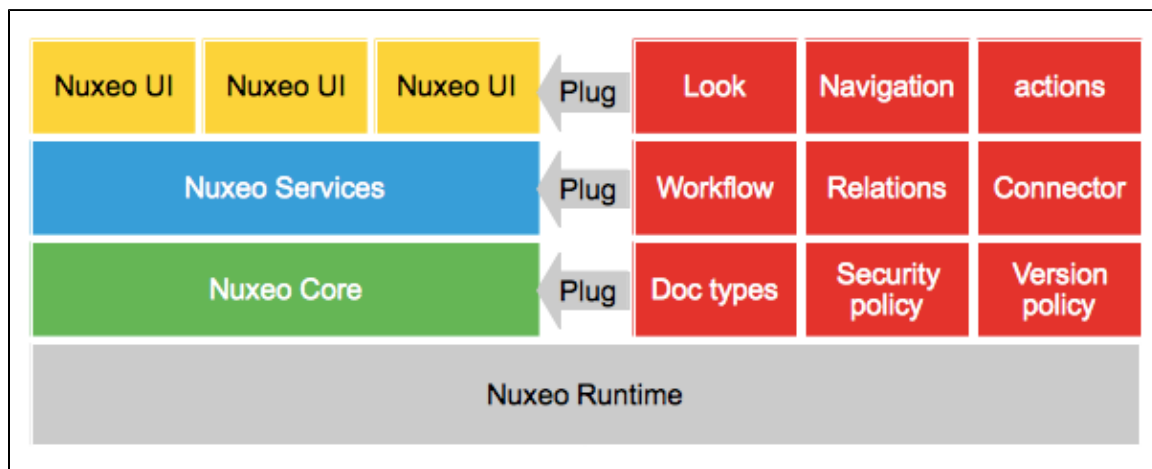
One of the main goals of Nuxeo EP is to provide an easy and clean way to customize the platform for your application needs.

For that, Nuxeo Runtime provides an *Extension Point* system that can be used to:

- configure services and components (XML configuration),
- extend existing services and components (Java code or scripting).

The extension point model is used everywhere inside Nuxeo EP. This means there is only one model for all components and services.

You use the same model to configure storage layers as to configure buttons in the Web UI.



In the current version of Nuxeo EP, there are about 200 extension points available.

This means:

- you can really do a lot with simple configuration;
- you can do customization that can be upgraded (extension points are maintained with the platform).

To leverage the Extension Points capabilities, you can either:

- write XML configuration,
- use Nuxeo Studio to generate the XML for you.

In terms of customization, the most common tasks include:

- 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 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.

If you need to extend Nuxeo EP or integrate it with existing applications, our platform also provides a lot of advantages:

- Nuxeo EP provides several APIs to access services (Java, RMI, SOAP, JAX-RS, ...),
- Nuxeo EP is based on standards (OSGi, JTA, JCA, JAAS, EJB3, JAX-RS, JAX-WS ...),
- You can use extension points to inject your code into the platform.

The last point is probably the most important.

Thanks to the extension point system and easy-to-use tools around - [Nuxeo Studio](#) and [Nuxeo IDE](#) - you can write your business code - without large technical knowledge - and inject it cleanly in the right component of Nuxeo EP:

- no need to hack to make it run,
- your custom code will be based on maintained extension points and interfaces and will be able to be easily upgraded.

About the content repository

In this section:

- [Document in Nuxeo](#)
 - [Document vs File](#)
 - [Schemas](#)
 - [Document Types](#)
 - [Life Cycle](#)
- [Security management](#)
 - [ACL model](#)
 - [Security Policies](#)
- [Indexing and Query](#)
 - [Indexing](#)
 - [Query support](#)

- Other repository features
 - Versioning
 - Proxies
 - Event systems
- Repository Storage
 - Nuxeo Visible Content Store (VCS)
 - Apache Jackrabbit Backend
- Advanced features
 - Lazy Loading and binary files streaming
 - Transaction management
 - DocumentModel Adapter

Document in Nuxeo

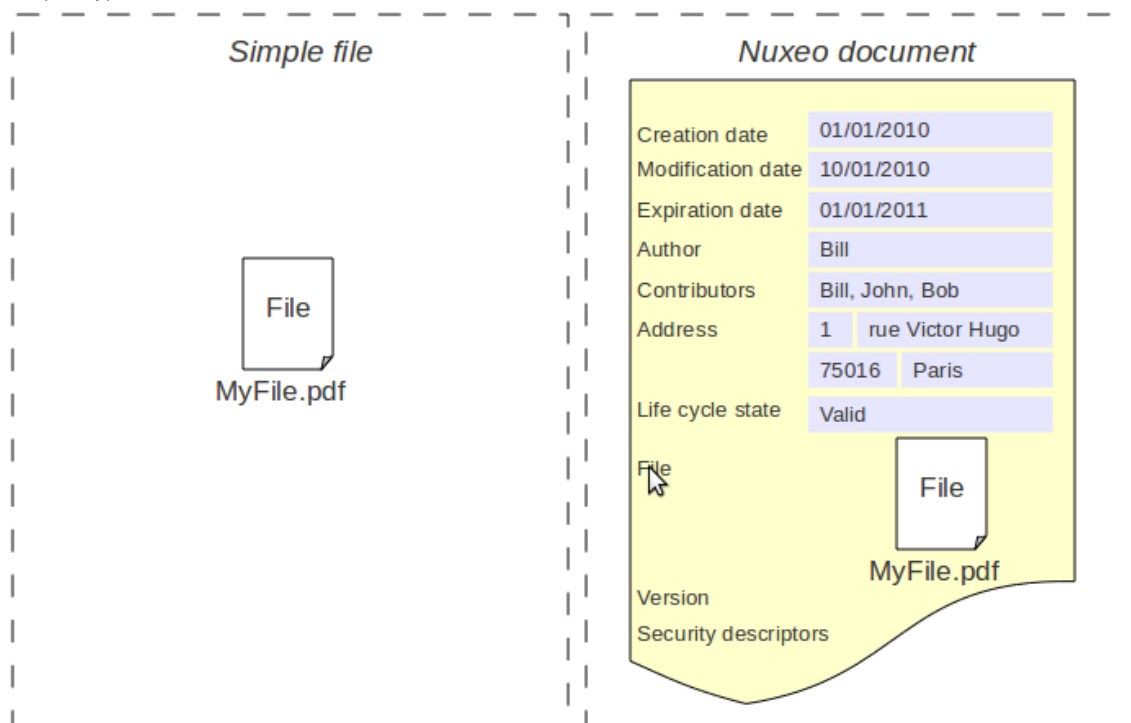
Document vs File

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 type.



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 0, 1 or several files.

In fact, inside the Nuxeo repository, even a Folder is seen as a document because it holds meta-data (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 meta-data 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 meta-data blocks.

At pure storage level, the Facets are simple declarative markers. These marker are used by the repository and other Nuxeo EP 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>

```

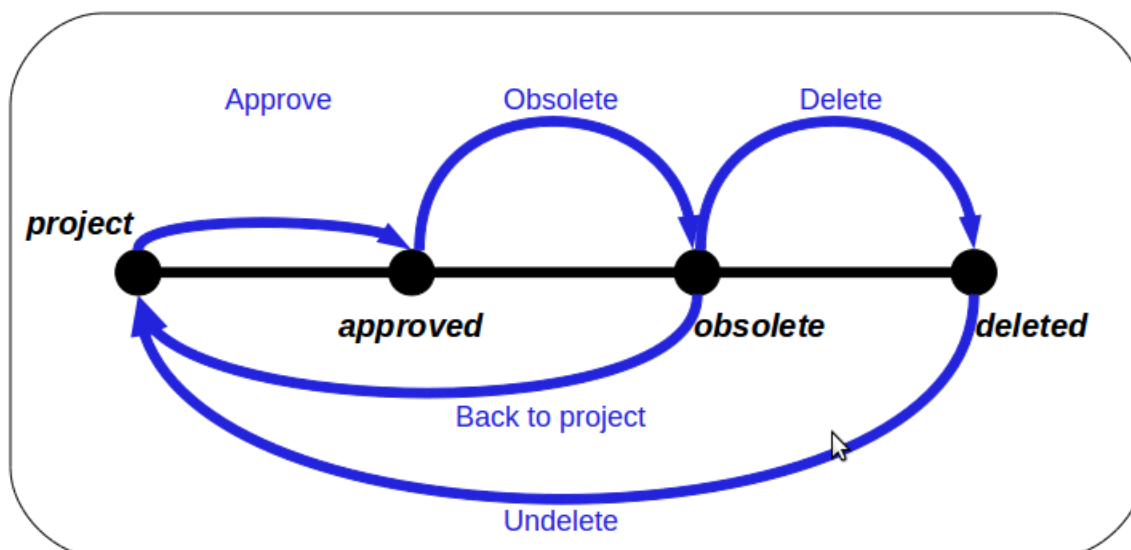
Life Cycle

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.

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.

ACP 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:

- document has a default ACL: the one that can be managed via back-office UI,
- 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 in 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 meta-data, if you use the ACL system, you have to compute a given ACL each time this meta-data is set. You will also have to compute a dedicated user group for each confidentiality level.

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

Such policies 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 meta-data (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 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 meta-data indexing, the Nuxeo Repository will maintain a fulltext index that will be composed of: all meta-data text content + all text extracted from files.

Configuration options depend on the storage backend, but you can define what should be put into the fulltext index and even define several separated fulltext 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 2 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 (meta-data) 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 Nuxeo DM 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:

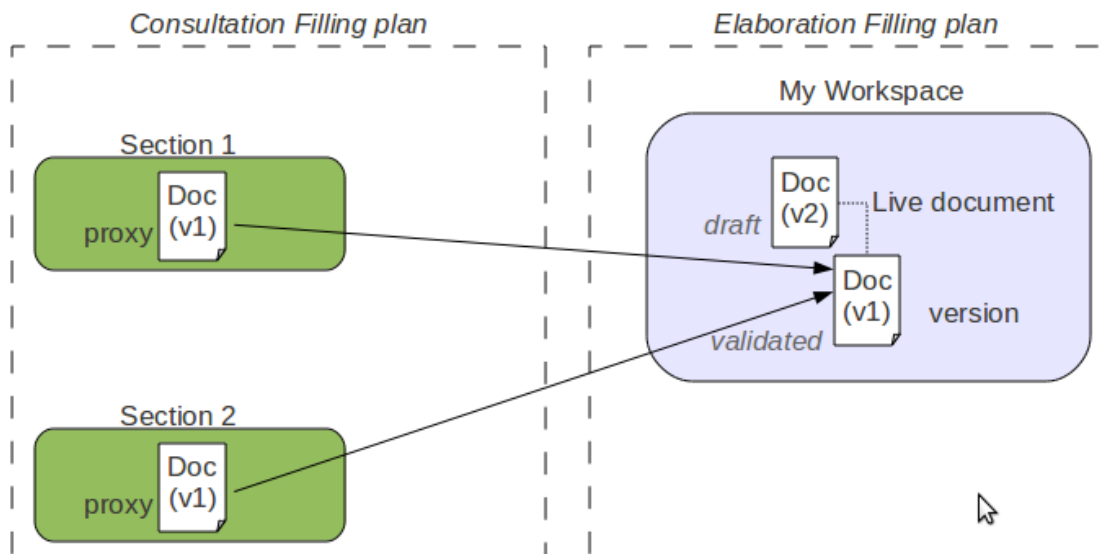
- the proxy will have the same meta-data as the target document,
- the proxy will hold the same files as the target documents (since file is a special kind of meta-data).

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 to duplicate any data.

The initial use case for proxies in Nuxeo DM is local publishing: when you are happy with a document (and eventually 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 can not 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 meta-data, 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 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,
- fulltext extraction is managed by a listener too,
- ...

Using the event listener system for these features offer 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

The Nuxeo Repository consists of several services.

One of them is responsible for actually managing persistence of Documents. This service is pluggable. Nuxeo Repository can have two different persistence backends:

- Nuxeo VCS.
- Apache Jackrabbit (only up to Nuxeo 5.3.2),

Choosing between the two backends depends on your constraints and requirements, but from the application point of view it is transparent:

- The API remains the same,
- Documents are the same.

The only impact is that VCS has additional features that are not supported by the Jackrabbit backend.

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 has several advantages:

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

VCS being now the default Nuxeo backend, it also provides some features that are not available when using the 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 2 (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.

Apache Jackrabbit Backend



This backend is not present in new Nuxeo versions.

The Jackrabbit backend is compliant with the JSR-170 standard (JCR).

This is the "historical" backend, since first versions of Nuxeo were using this backend by default (it was the only one available).

Jackrabbit provides a storage abstraction layer and can be configured:

- to store everything on the file system (meta-data + files),
- to store everything in a SQL DataBase (meta-data + files),
- to store meta-data in the SQL DataBase and store files on the filesystem (recommended solution).

Advantages of this backend:

- it is JSR-170 compliant so you can use any compliant browser to access your Nuxeo Documents, even without Nuxeo code,
- it can run on a pure filesystem (not recommended for production).

Drawbacks of this backend:

- SQL storage is cryptic (Database stores serialized java objects),
- JackRabbit uses a Lucene index on filesystem (so clustering and hot-backup are complicated),
- doing reporting on JackRabbit data is complex.

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 chunk 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 backend) 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:

- provide helper methods that compute or update some fields,
- add integrity checks based on business rules,
- add 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, Nuxeo EP 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,
- ...

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.

In this section:

- [Mapping Nuxeo to nodes and properties](#)
 - [Nodes, properties, children](#)
 - [Children](#)
 - [Fragment tables](#)
 - [Fields mapping](#)
 - [Security](#)
- [Fragments tables](#)
 - [Hierarchy table](#)
 - [Type information](#)
 - [Simple fragment tables](#)
 - [Collection fragment tables](#)
 - [Files and binaries](#)
 - [Relations](#)
 - [Versioning](#)
 - [Proxies](#)
 - [Locking](#)
 - [Security](#)
 - [Miscellaneous values](#)
 - [Fulltext](#)
- [Other system tables](#)
 - [Repositories](#)
 - [Clustering](#)
 - [Path optimizations](#)
 - [ACL optimizations](#)

Mapping Nuxeo to nodes and properties

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 node. 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` column provides ordering of the values.

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

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.

Fragments 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 others 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-601e0653f906 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 since Nuxeo EP 5.3.2 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|`.

Note that the `mixintypes` column is new since Nuxeo EP 5.4.1.

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 hash, 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

Since Nuxeo 5.3.2, 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` before Nuxeo 5.5:

id	source	target
1843	5670	5700

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).

Since Nuxeo 5.5 the relation type holds more information fit all relational needs:

Table `relation`:

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

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`,
- the column `isversion` is new since Nuxeo EP 5.4.

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	2.1		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's benefit (before Nuxeo EP 5.4 it was just an integer incremented for each new version),
- the columns `islatest` and `islatestmajor` are new since Nuxeo EP 5.4.

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:

- the `targetid` is the id of a version node and is a `FOREIGN KEY` reference to `hierarchy.id`.
- the `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.

Before Nuxeo EP 5.4.1, the locking table was simpler and only contained one `lock` column with values like `Administrator:Aug 20, 2008`, or `NULL` when not locked. This was changed to the format described above by [NXP-6054](#).

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 lifecycle information (lifecycle policy and lifecycle 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		

Fulltext

The fulltext 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.

Since Nuxeo EP 5.4.1, 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 fulltext 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 since Nuxeo EP 5.3.2 (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 **read_acls**:

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 **read_acls_cache**:

users_md5	acl_id
f4bb42d8	1
f4bb42d8	1234
f4bb42d8	5678
c5ad3c99	1
c5ad3c99	1234

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.

Nuxeo Platform 5.5 Technical Documentation

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

Table hierarchy_modified_acl:

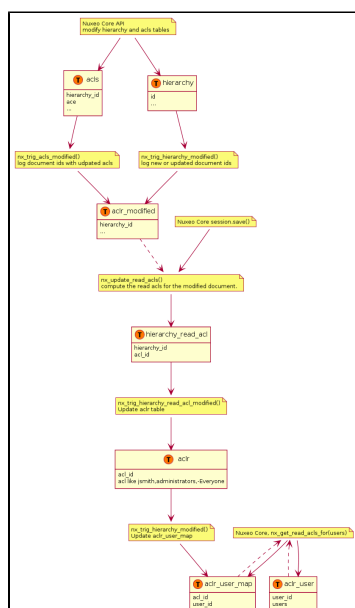
id	is_new
5678	FALSE
5690	TRUE

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_REAL_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:



Platform features quick overview

This page presents a quick overview of the main features available in Nuxeo EP:

- Document Management
- Indexing
- Renditions, preview and annotations
- Process Management and content automation
- eMails
- Digital Asset Management
- Portal and Web views

It is not supposed to be exhaustive, the main target is to help you see what Nuxeo EP can do for you.

Document Management

Document Management features are the core of Nuxeo EP.

Basically, using the Nuxeo Repository you can define your own document types. For this, you can define XSD schemas that will define the structure of your documents:

- you can share schemas between document types, to have common blocks of meta-data,
- you can use inheritance between document types, to create a hierarchy.

The fields in schemas (and then in documents) can be:

- simple types (String, Date, Integer, Boolean, Double ...),

- list of simple types (multi-valued properties),
- complex types:
 - file (binary stream, filename, mime-type, size),
 - custom complex types (like an postal address).

The Nuxeo Repository can manage versioning on documents (including numbering policies), manage customized ID generation. It can also manage security on documents (see [the page on Repository overview](#) for more details).

Nuxeo EP lets you associate documents with relations and tags. It also provides UI building blocks to help you use the documents inside the repository:

- The form (Layout and Widget) system enables you to easily define View/Edit/Create screens,
- Navigation can be based on several physical filing plans:
 - physical plan,
 - meta-data (virtual navigation),
 - tags,
 - ...

Indexing

All documents stored in the Nuxeo Repository are indexed. *Indexing* is configurable and by default manages:

- all meta-data keyword indexing,
- fulltext indexing on all extracted from meta-data fields and files.

All standard files types are *fulltext indexed*. Fulltext extraction is pluggable: you can add a custom extractor if you have very specific file types.

The Repository provides a *query system* to let you:

- search on meta-data,
- search on fulltext.

Nuxeo Search can be done in [NXQL](#) (SQL like) or CMISQL.

Search results take security into account: a user can not find a document that he can not access.

Renditions, preview and annotations

Nuxeo EP includes a *conversion service* that can be used to convert binary files to text, html, pdf, etc. This `ConversionService` is pluggable and you can define your own conversion plugins and your own conversion chains.

Nuxeo EP also includes a *preview service* that provides HTML preview UI coupled with an *annotation service*. This lets user read and annotate a MSOffice file without needing to run MSOffice locally. The Annotation server is based on the Annotea W3C standard and provides annotation capabilities for both text and images.

Process Management and content automation

Documents are associated with a *lifecycle*. The lifecycle defines the possible states of a document and the associated transitions. Many simple review process can be managed with a simple lifecycle and some custom listeners (java code or scripting).

You can also use *Content Automation* to simply define operation chains triggered by events using [Nuxeo Studio](#) (no need to code).

Of course, if you need real *Business Process management*, Nuxeo EP integrates the jBPM engine and a set of generic handlers that can be used to manipulate documents from within the business process context.

eMails

Nuxeo provides features to:

- send a document via email,
- manage notifications via email,
- fetch eMails from a mailbox and transform them into Nuxeo Documents,
- inject eMails in Nuxeo Repository.

Digital Asset Management

Nuxeo EP provides several features to deal with *pictures*:

- extract or set meta-data associated with picture formats (EXIF, IPTC ...),
- resize and rotate images,
- convert images between formats,
- generate thumbnails and picture book views,
- provide browsing and tiling web UI (allows to view and zoom on a very large image even in a web browser).

For *video* assets, Nuxeo EP provides services for:

- video conversion,
- video thumbnails extraction,
- integration with a streaming server (Darwin).

Portal and Web views

The main target of Nuxeo EP is to provide tools to create and manage content. But Nuxeo EP also provides tools to generate *Web views* on your content.

For that, [WebEngine](#) provides a simple template-based rendering system that is completely integrated with Nuxeo documents and services.

With WebEngine, it's easy to provide a custom web view on top of a document, workspace, folder, ...

Nuxeo EP also provides pre-built WebEngine module for WebSites, Blogs and Wikis.

If you want to aggregate content or small web applications you can also use Nuxeo as a *portal*.

For that Nuxeo EP integrates:

- a UWA widgets container,
- a complete OpenSocial Server (OAuth, Google Gadgets, Social APIs...).

Component model overview

In this section:

- [Nuxeo Bundles](#)
- [Components and Services](#)
- [Extension Points](#)
 - [Declare an extension point](#)
 - [Contribute to an Extension Point](#)
 - [Extension Points everywhere](#)
- [Packaging and deployment](#)

Nuxeo Bundles

Inside Nuxeo EP, software parts are packaged as Bundles.

A Nuxeo Bundle is a jar archive containing:

- 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 (ie: other bundles that should be present for this bundle to run),
- list XML components that are part of the bundle.

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:

- Bundle is named `org.nuxeo.ecm.core.convert`.
- Bundle depends on 2 other bundles: `core.api` and `convert.api`.
- Bundle contains one XML component: `convert-service-framework.xml`.

Components and Services

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 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. Once declared the Service will be available via a simple Nuxeo Runtime call:

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

Extension Points

One of the corner stone 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 (= contribute XML configuration),
- extend the behavior of components (= contribute Java code or scripting).

Basically, inside Nuxeo EP, 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:

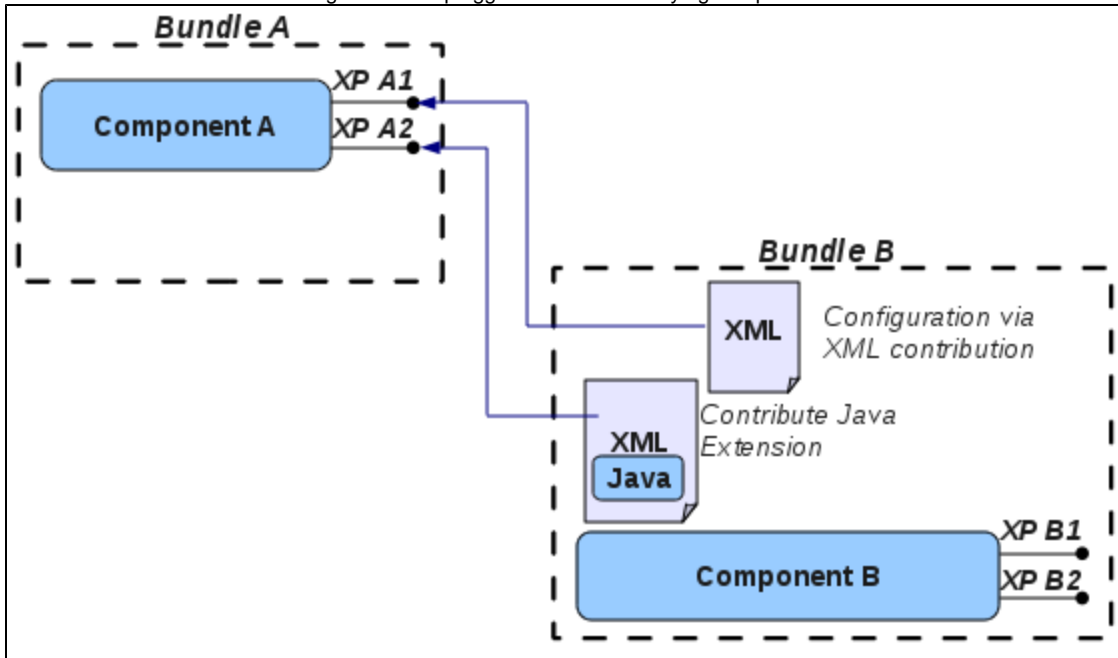
- inside Nuxeo Runtime itself,
- inside Nuxeo Core (configure and extend Document storage),
- inside Nuxeo Service layer (configure and extend ECM services),
- inside UI layer (assemble building blocks, contribute new buttons or views, configure navigation, ...).

Each Java Component can declare one or several extension points.

These Extension Points can be used:

- to provide configuration,
- to provide additionnal code (i.e. : plugin system).

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



Declare 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:

- the declaration of a Java Component (via the component tag) with a unique id (into the name attribute),

- 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 2 extension points:
 - one to contribute configuration,
 - one to contribute java code (new converter plugins).

Each extension point have his own xml structure descriptor, to specify the xml fragment is waiting for into this extension point:

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

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

Each Nuxeo extension points use this pattern to declare configuration possibilities, service integration, behavior extension, etc...

You understand this pattern, you will understand all extension points into Nuxeo. And you can use this infrastructure to declare your own business services.

Contribute to an Extension Point

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

For that, the XML component needs:

- to be referenced in a MANIFEST bundle,
- to specify a target extension point,
- to 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 exemple 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

Nuxeo Platform uses extension Points extensively, to let you extend and configure most of the features provided by the platform.

You can find all extension Points available in the DM distribution [here](#).

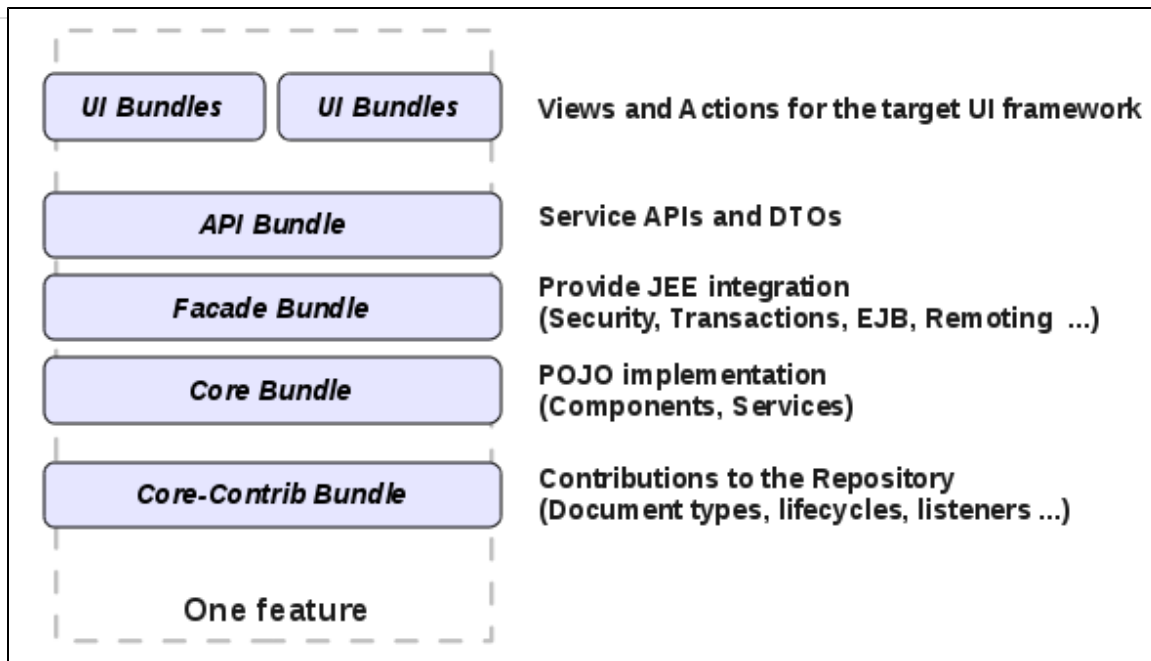
Packaging and deployment

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

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.

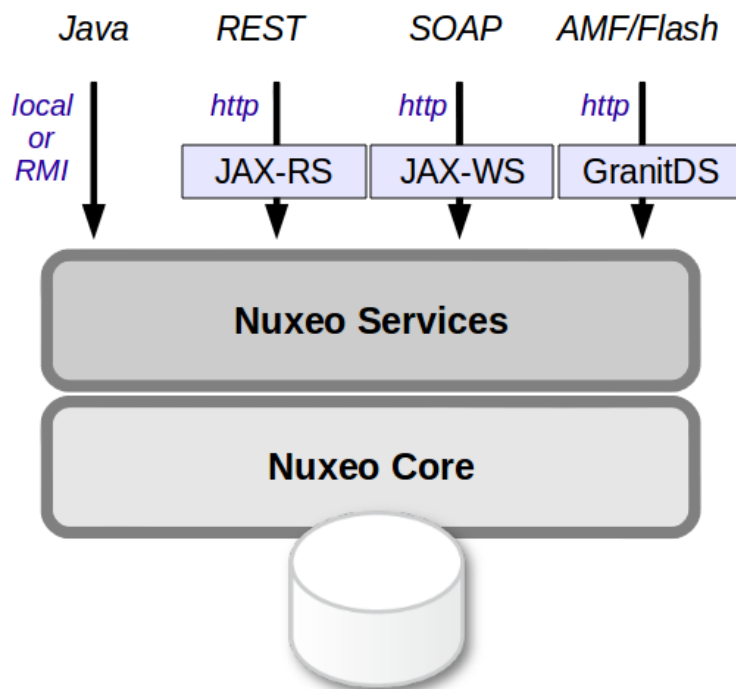
This is basically a bundle group for a given feature.

API and connectors

Nuxeo APIs

Nuxeo EP provides several types of API :

- Java local API
- Java remote API
- WebService (JAX-WS) API
- REST (JAX-RS) API
- AMF (Flash) API



Since Nuxeo EP is built using the java language, only the Java local API can be used to extend Nuxeo.

When running in the same JVM as Nuxeo you have access to all Nuxeo API (components and services).

For more information about using Nuxeo internal API, please see the [Customization and Development](#) and [Extending Nuxeo](#) sections.

All other APIs provide remote access and allow you to call some Nuxeo Services from an external application. These are typically the APIs you may use to integrate Nuxeo with third party application. For more information about this topic, please see : [Using Nuxeo as a service platform](#).

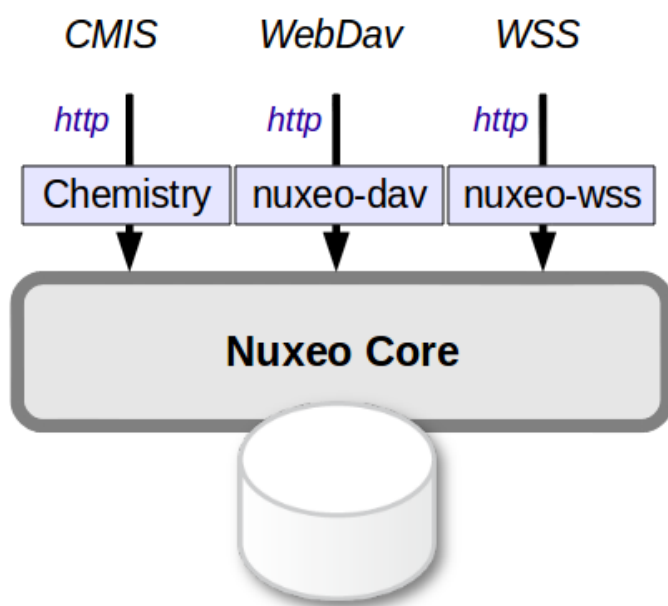
Content oriented Protocols

You can also access Nuxeo EP content via a "standard" protocol.

Nuxeo EP currently supports:

- CMIS,
- WebDav,
- Windows Sharepoint Services protocol (only the subset used by MS Office and MS Explorer).

Compared to the APIs, these protocols won't give you access to services but directly to the content.



Depending on the chosen protocol, your access to Nuxeo Content will be more or less powerful:

- CMIS gives you access to a big subset of the data managed by Nuxeo.
- WebDav and WSS mainly map 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.

For more information, please see the section dedicated to [Repository access](#).

Connectors

Directories

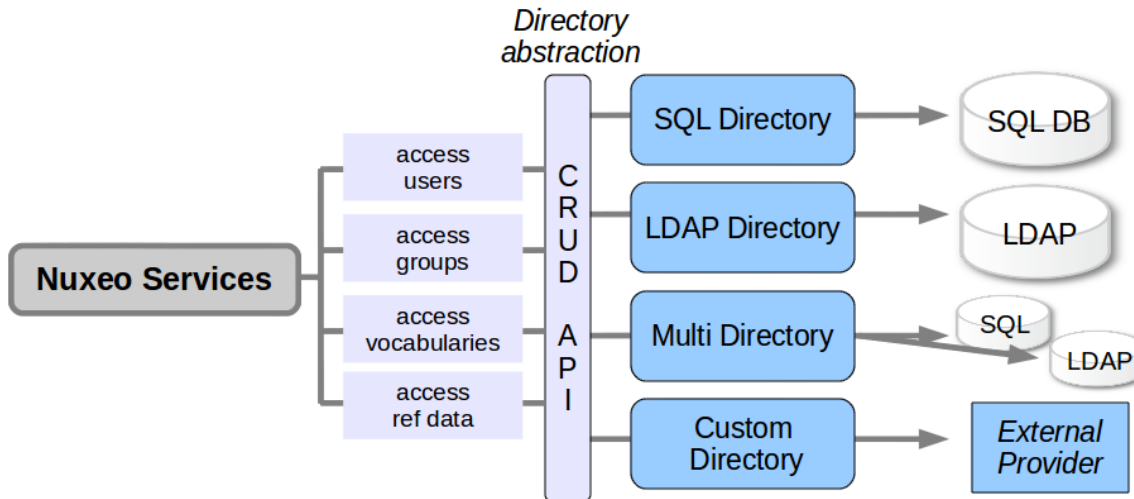
Inside Nuxeo EP, directories are used to provide a abstraction on all referencial data that can be manipulated inside the application.

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.

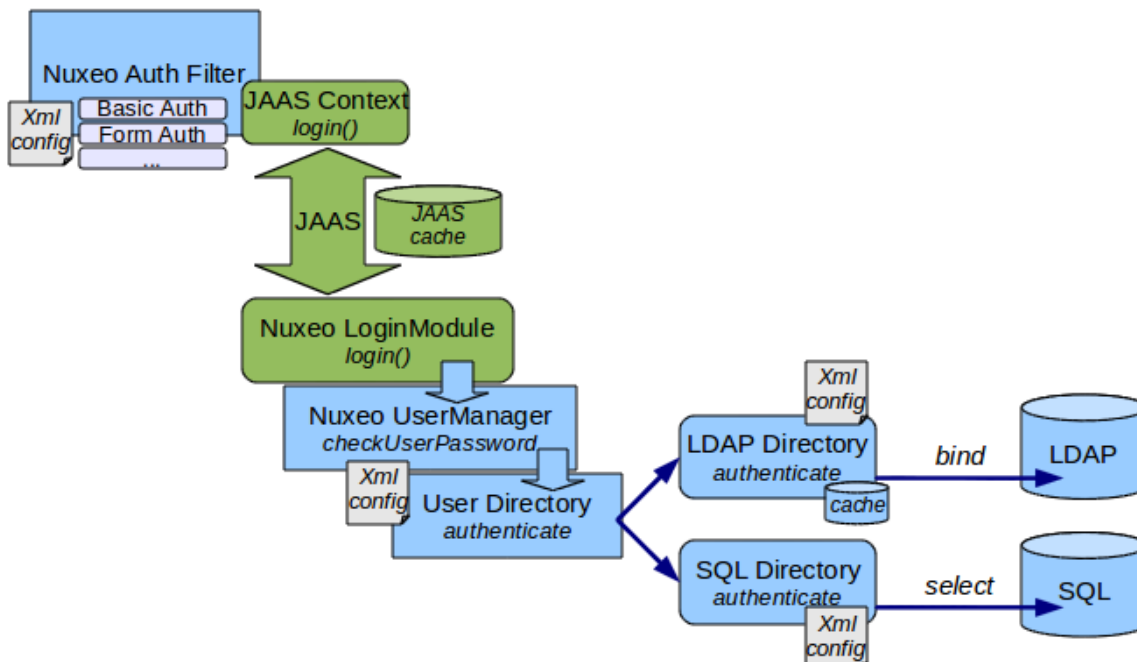
A frequent use case is also to use the directory abstraction to map a given webservice or application that manages centralized data. (We provide a connector sample for that.)

Authentication and User management

Authentication and user management is also a typical use case for integration between Nuxeo EP and existing infrastructures:

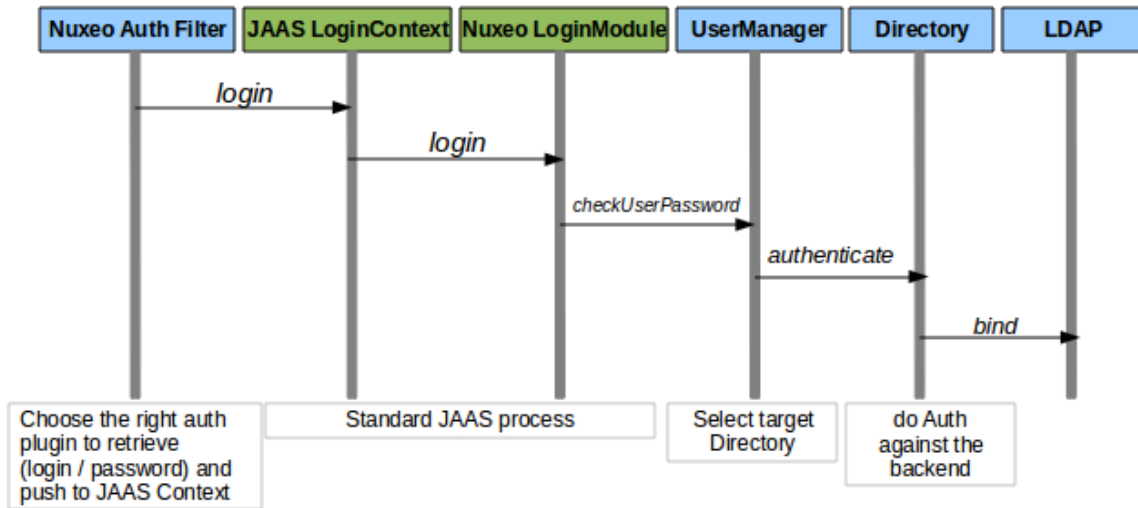
- integrating a Single Sign On system,
- integrating an application that manage users profiles,
- integrating an application that manage groups.

As seen above, directories provide part of the solution. But in order to be able to integrate a lot of different authentication use cases, Nuxeo EP authentication system was designed to be very pluggable.



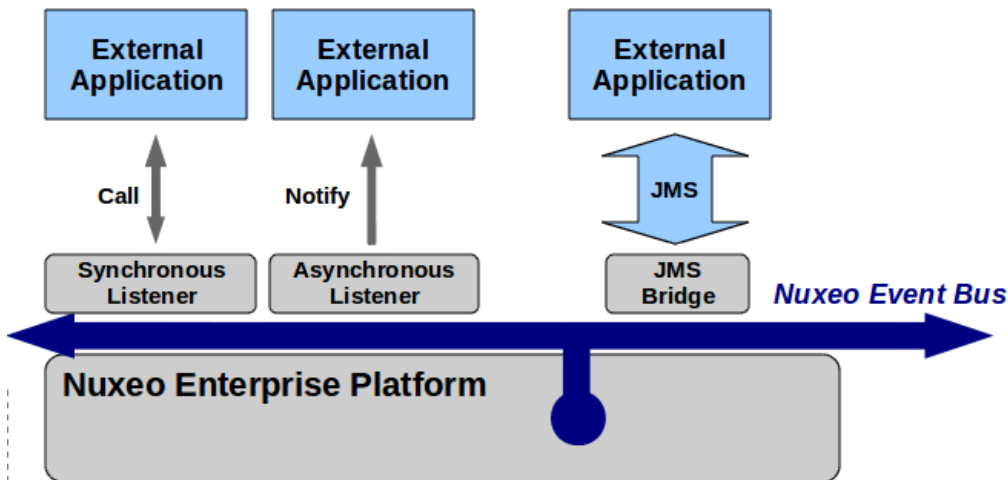
The initial identification can be done at Java level via JAAS or at Http level via a dedicated filter. The filter is pluggable so that the way of retrieving

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.



Integration via Nuxeo Event system

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



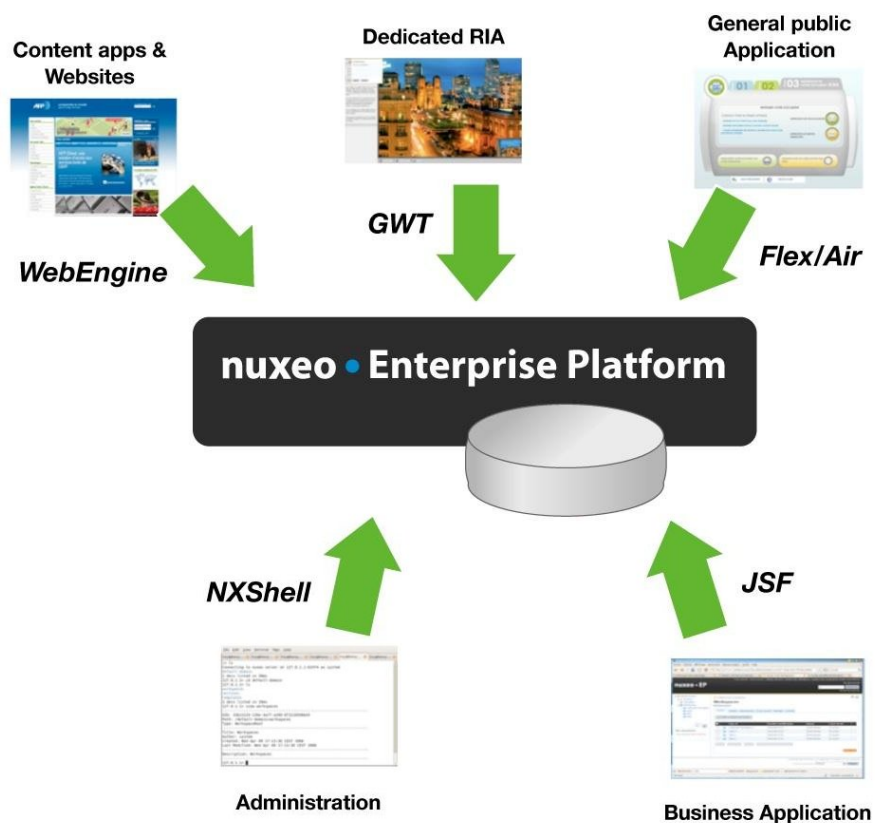
UI frameworks

Nuxeo EP proposes different technologies for the client side of the application.

The choice of one technology vs. the other depends on both the project's stakes and its context.

The different technologies available are:

- JSF/Seam
- WebEngine
- Flex client
- GWT Client
- Shell Client



JSF/Seam

The Nuxeo EP default web UI is based on JSF (a Java EE standard) for the UI component model, and Seam for the navigation and context management.

Technologies:

- Sun JSF 1.2,
- Facelets,
- JBoss RichFaces 3.2,
- JBoss Seam 2.

Key points:

Nuxeo EP's JSF interface is fully modular and very easy to customize:

- integrated theme manager,
- XML configuration for buttons, tabs, actions, etc.
- Form and Widget layout engine.

Typical use case:

- Business application,
- Document management back-office.

WebEngine

JSF technology is not best suited to create websites with content managed by Nuxeo EP, due to JSF's stateful model, full abstraction of the HTML/JS code, etc. Thus, Nuxeo has developed a simple yet powerful rendering engine based on Freemarker and JAX-RS: [Nuxeo WebEngine](#).

Technologies:

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

Key points:

Nuxeo WebEngine enables web developers to easily create a customized web interface on top of Nuxeo EP:

- Simple template engine,

- Direct access to HTML,
- Java scripting support,
- Lightweight development environment based on Jetty.

Typical use case:

Nuxeo WebEngine is designed to expose Nuxeo EP 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 Nuxeo EP.

Flex client

Nuxeo EP provides a Flex/AMF connector allowing an Air/Flex client to connect.

Technologies:

- Air/Flex,
- AMF remoting integrated via GraniteDS.

Key points:

The Flex technology can be easily deployed, as the equipment rate in flash VM is quite high. The Flash technology allows rapid development of advanced clients with a rich and user-friendly interface.

Typical use case:

It would be a small application requiring rich media support and a plain user interface for a large audience. For example, this technology has been used for eLearning applications based on Nuxeo EP.

GWT Client

GWT (Google Web Toolkit) allows the Java development of applications that will be deployed under HTML/JavaScript format.

Nuxeo has integrated the GWT technology:

- in the build environment (via Maven),
- with the platform via dedicated REST APIs,
- with the extension points model (to allow modular development with GWT as it is available within the rest of Nuxeo EP).

Technologies:

- Google Web Toolkit 1.5,
- JAX-RS to communicate with server.

Key Points:

The GWT technology allows the development of user-friendly and reactive applications with no deployment needed.

From the development side, it is highly productive to be able to code in Java (Java IDE, Type Safety, unit tests) without bothering with classical RIA related problems (JavaScript debug, multi-browsers support, etc.).

Typical use case:

GWT allows the development of complex interfaces that are difficult to create rapidly with standard web technologies:

- Text and image annotation interface,
- Tiling client to display and navigate in large images.

Shell Client

The Shell client is based on the client same library than the one embedded in Apogee.

With Nuxeo Shell, the client is presented with a command-line shell and a set of commands to directly access Nuxeo Services and Content Repository.

Technologies:

- Java OSGi,
- Groovy Scripting,
- Jline for the command line.

Key Points:

The Nuxeo Shell may be used in 2 modes:

- the interactive mode (commands line use),
- the non-interactive mode (scripts & batches).

The available commands are defined by an extension point and Java classes or by simple Groovy scripts. It is therefore very easy to add customized commands for each project.

Typical use cases:

The Nuxeo Shell may prove useful in several cases:

- administration access:
 - command line use,
 - scripting implementation of customized commands.
- Exec environment for scheduled commands,
- Data recovery tool,
- Low level performance test tool.

Deployment options

Here are the different deployment possibilities:

- [Agile deployment](#)
 - [Simple deployment](#)
 - [Cluster deployment](#)
 - [Multi-VM deployment](#)
 - [Service externalization](#)
 - [Isolating the web layer](#)
- [Sample deployments](#)
 - [HA deployment and DRP](#)
 - [Offline client](#)
 - [Multi-Instances](#)

Agile deployment

Thanks to Nuxeo Runtime and to the bundle system, Nuxeo EP deployment can be adapted to your needs:

- deploy only the bundles you really need,
- deploy on multiple servers if needed,
- deploy on multiple infrastructure:
 - server side: JBoss, Tomcat, Jerry or POJO,
 - client side: Equinox, POJO.

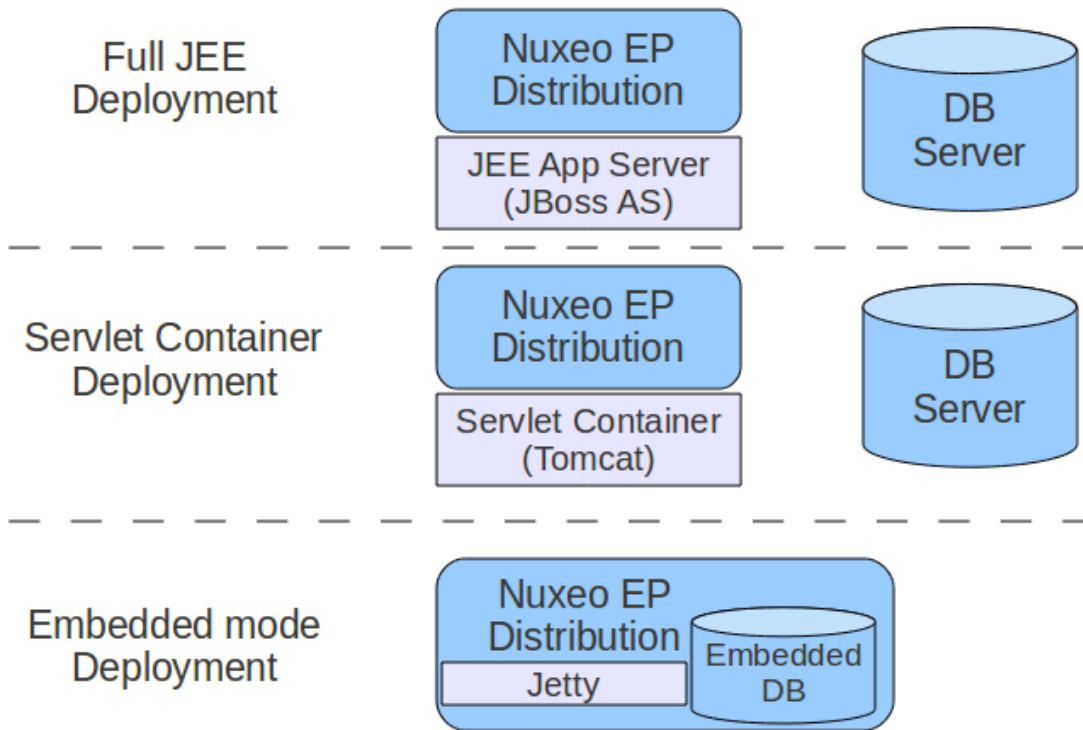
Simple deployment

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:
 - full JEE server: JBoss AS,
 - servlet container: Tomcat,
 - embeded mode: Jetty.

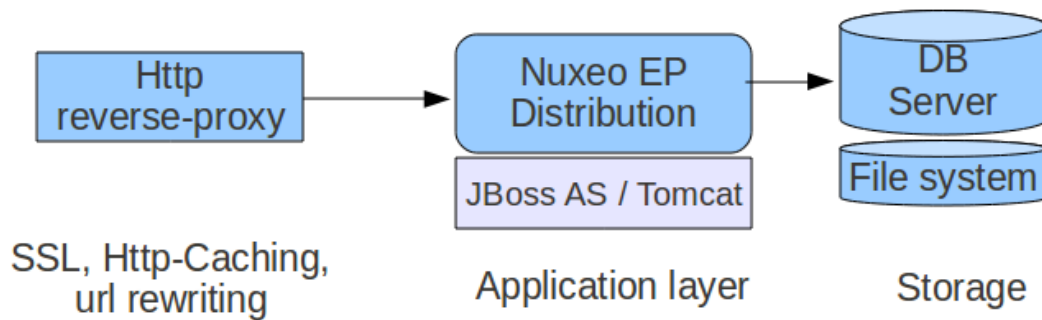
The Tomcat packaging comes in 2 flavors:

- bare Tomcat packaging,
- JCA/JTA Tomcat packaging (default): it adds to Tomcat the required infrastructure to manage transaction and JCA polling.



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

- https/ssl encryption,
- http caching,
- url rewriting.

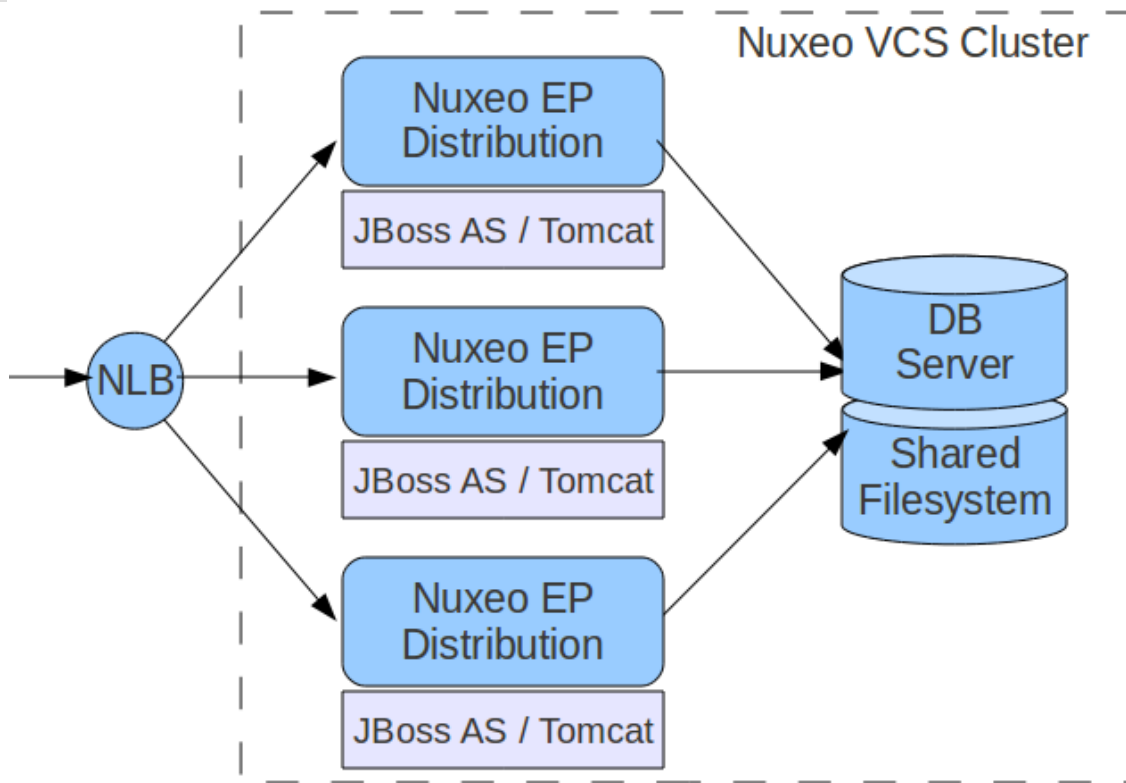


Cluster deployment

In order to manage scale out, Nuxeo EP provides a simple clustering solution.

When using VCS repository storage mode, it is possible to activate the cluster mode. When cluster mode is enabled, you can have several Nuxeo EP 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.

Depending on the UI framework used for presentation layer, the network load balancing can be statefull (JSF webapp) or stateless (WebEngine).



Multi-VM deployment

Nuxeo components interact between each others via service interfaces. Since most services can be remotely called via RMI, you can split a Nuxeo distribution (like DM) in several parts on several JVM.

Of course, this kind of multi-VM deployment is more complex than simple mono-vm deployment:

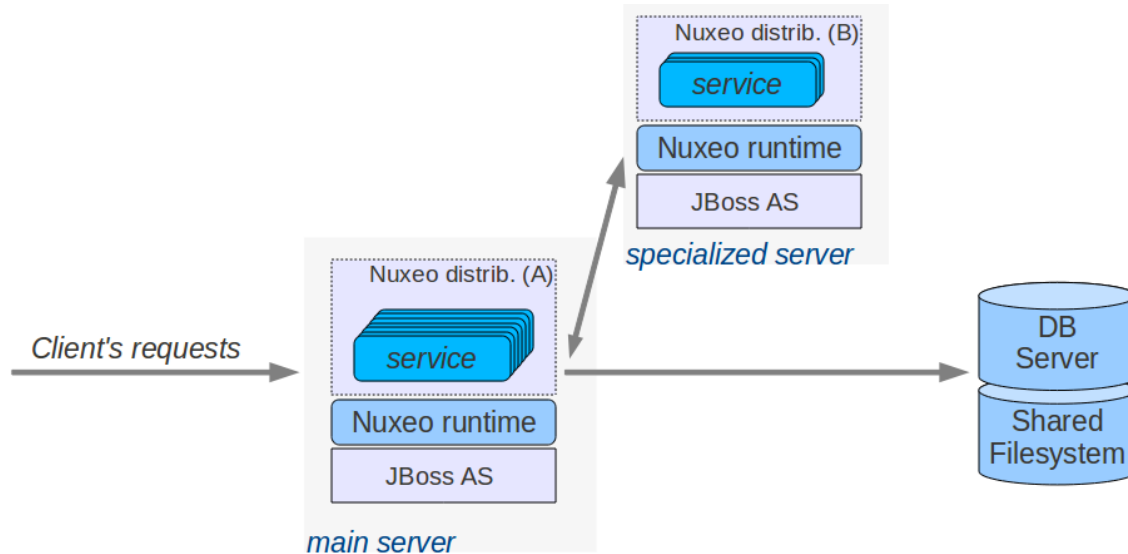
- you need to configure service lookup layout (service groups),
- all additional custom plugins must be cleanly packaged.

Nevertheless, this kind of deployment can be interesting to solve specific requirement.

Service externalization

Depending on your use cases, some services may consume more resources than others. For example, if the application needs to do heavy conversion work or needs to resize a lot of pictures, it may be interesting to isolate theses services:

- to prevent batch processing from slowing down interactive processing,
- to spread services on more CPUs.



Isolating the web layer

An other use case is to separate Nuxeo bundles into two categories:

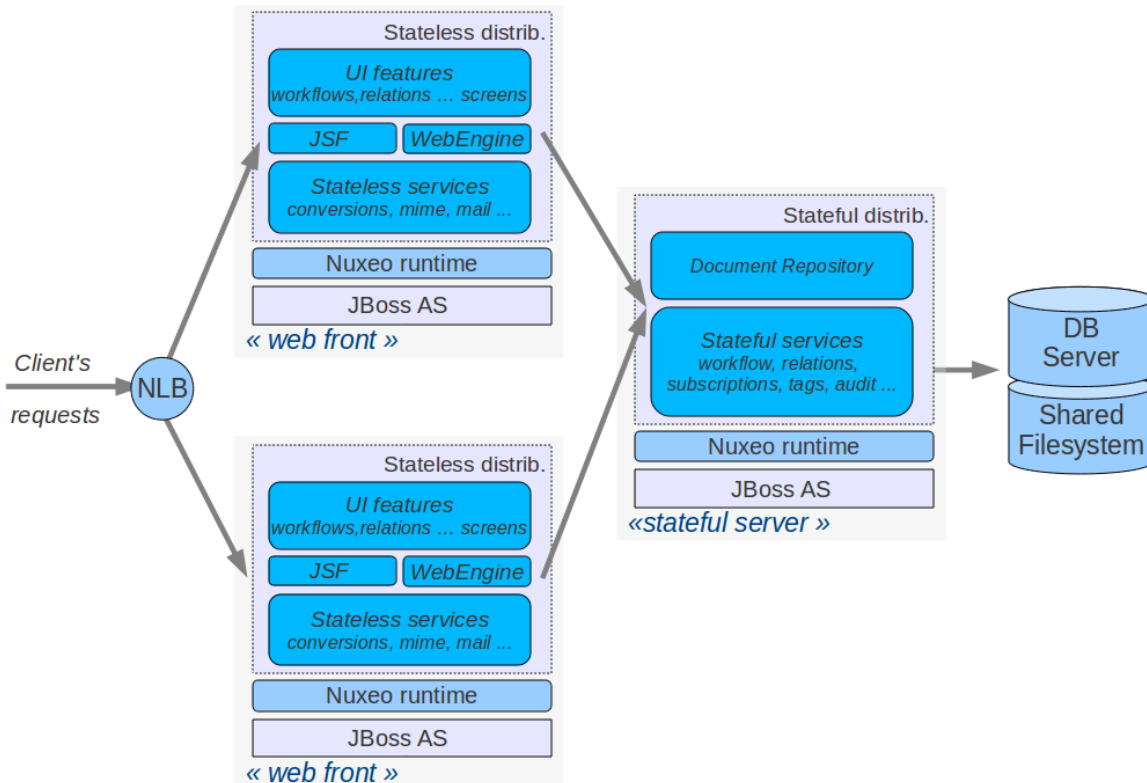
- bundles that manage persistent transactional data,
- bundles that don't.

By doing this split, we have:

- A "Stateless server" that runs:
 - all services without state,
 - the UI layer (framework and screens associated to features).
- A "Stateful server" that runs all persistent services including the Document repository.

This 2 parts packaging provides some advantages:

- it allows scaling out of the web layers,
- it provides full 3 layers decoupling (Nuxeo can be hosted in a 3 layers DMZs architecture).



i These two packaging are somehow badly named since the Stateless server is not really stateless (JSF is statefull for example), but it does not manage any persistent state.

Sample deployments

HA deployment and 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.

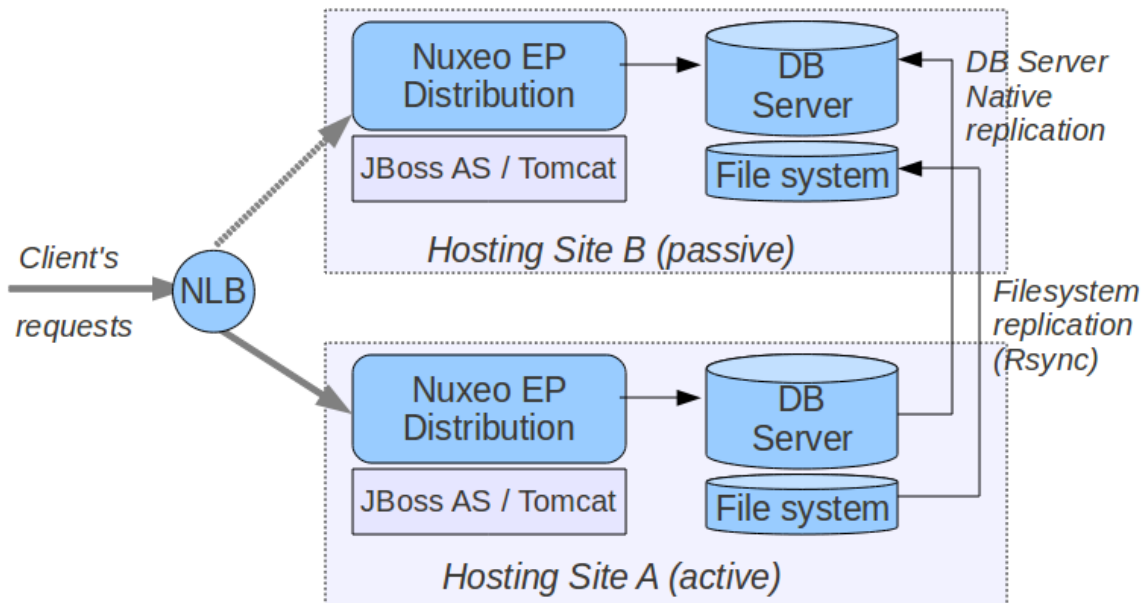
Because Nuxeo storage VCS+Filesystem is safe, you can use a replication system between the two sites. Basically, you can use the replication/standby solution provided by the Database server you choose. This replication tool just has to be transactional.

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:

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



Offline client

Multi-Instances

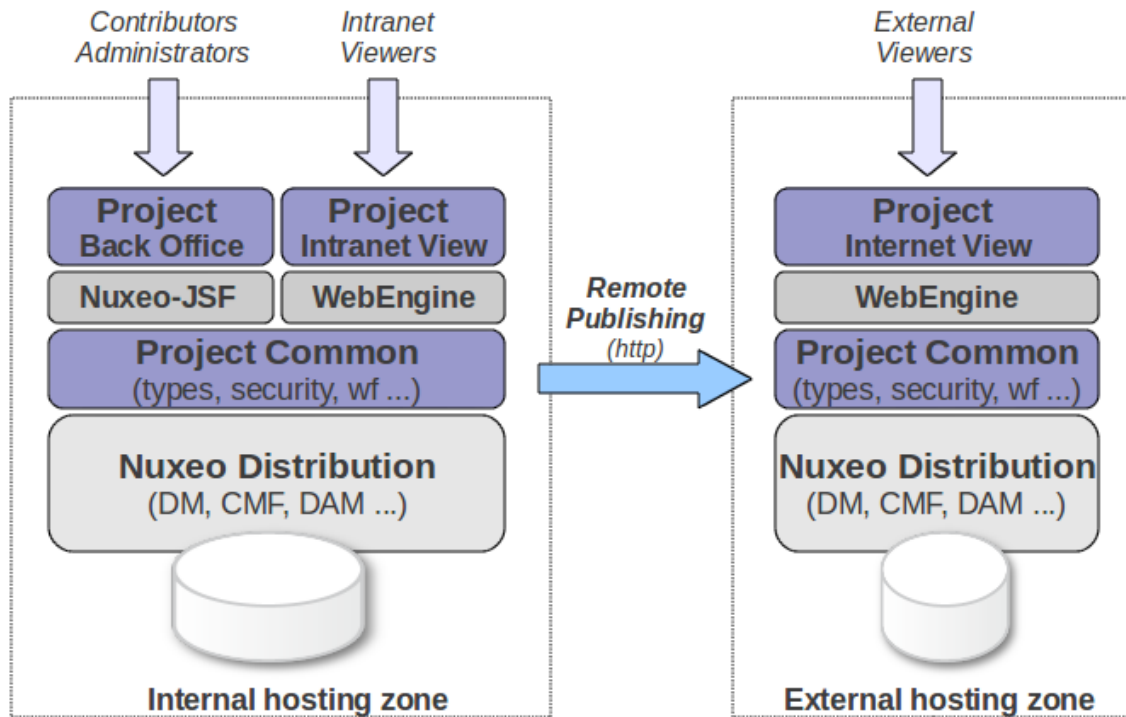
In some cases, you may want to have several separated instances:

- because you have several sub-applications,
- because you want to use different kinds of hardware depending on the sub-application,
- because you have constraints on hosting in separated DMZ.

A typical use case is when you have an internal application for contribution and viewing, but you also need to let external (internet) users access a small part of the content. Technically, you could have the same application serving the content depending on the user profile or several applications sharing the same repository. But in many cases, hosting and security constraints may lead to have two separated hosting platforms: one for internal and one for external.

In order to achieve that you can leverage the built-in feature for remote publishing between two Nuxeo instances.

See Remote Publisher for more details)



Performance management for the Nuxeo 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.

On this page

- Performance of the Nuxeo Platform
 - Impacting factors
 - Factors that have little or no impact
 - Some generic tips for tuning the Nuxeo Platform
- How we manage the Nuxeo Platform performance
 - A Toolbox for benchmarking the Nuxeo Platform
 - Continuous performance testing via CI
 - Periodic Benchmark campaigns
- Sizing your Nuxeo Platform-based ECM application
 - Define your requirements
 - Setup performance testing from the beginning
 - Use interpolation when needed
- Performance toolbox provided by the Nuxeo Platform
 - Benchmarking tools
 - Metrics to monitor during a bench
 - Monitoring tools
 - Nuxeo Metrics Monitoring tools with mbeans
- Some example Benchmark results
 - Goals
 - Steps
 - Results overview
 - Customizing bench

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.

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.

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 usefull 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.
- PostgreSQL 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 which 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:

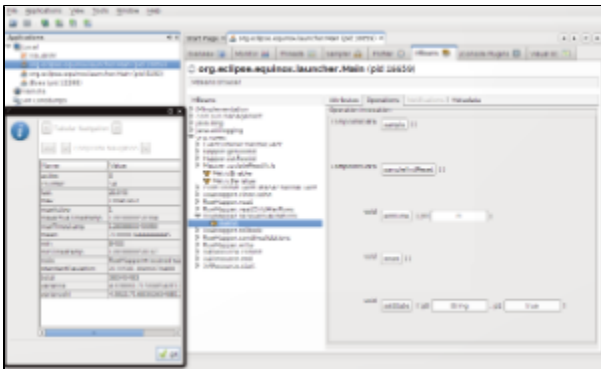
```
<backendClass>org.nuxeo.ecm.core.storage.sql.management.MonitoredJDBCBackend</backendClass>
```

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.
- 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.

Customization and Development

Nuxeo Platform provides several solutions to let you customize and extend the platform:

- use Nuxeo Studio to do your configurations and extensions via a Web UI,
- write XML files to configure Nuxeo and deploy new plugins,
- develop your own extensions and plugins.

Choosing the right solution depends on:

- your requirements:
if you want to customize your Nuxeo application, Nuxeo Studio should do all the work for you. But if you want to develop a very specific service or component, you will have to write code;
- your profile:
in order to develop extensions to Nuxeo you need to be able to write some Java code.

Using Nuxeo Studio

If you are not used to customizing Nuxeo, you should give a try to [Nuxeo Studio](#), a visual environment to configure your Nuxeo DM, DAM or CMF instance.

Doing XML configuration

Inside the Nuxeo Platform, you can configure a lot of stuff via simple XML files.

We advise you to give first a glance at all the Nuxeo EP wiki domain, so that you get a global idea of the architecture of the product. In particular, you should have a look at the [Component model overview](#) that will explain you how the platform is built.

Then, you will need to know:

- what can be configured inside the platform (what extension points exist);
- what contributions are already deployed in the Nuxeo distribution you use.

If you want to go further and configure other aspects of Nuxeo, you don't need to get the source code, you can simply browse the [Platform Explorer Site](#) that let you browse:

- [Nuxeo Distributions](#),
- [Services](#),
- [Extension Points](#).

For each item, you can have access to description, XML definition and samples.

For an example, you can have a look at this [Sample Link](#).

When you are not creating a real Nuxeo Plugin (i.e. a JAR), XML configuration files should:

- be copied in the "config" directory (`nuxeo.ear/config` or `nxserver/config`),
- have a filename ending with `-config.xml`,
- have a unique component identifier.

By default, XML files contributed in the "config" directory are loaded only when the server starts, so you need to restart the server to see your changes.

Java plugins

If you want to go further (or just prefer coding), you can of course use Java to build a new Nuxeo component.

One of the key points is that you don't need Nuxeo source code to do that:

- you don't need to have Nuxeo source code to be able to write a plugin,
- you don't need to rebuild Nuxeo to deploy your plugin.

Nuxeo Java components are deployed the same way as XML components are deployed, you just have to package the JAR correctly, copy it in the right location and restart the server.

In order to start coding you can read the [Dev Cookbook](#).

Learning to customize Nuxeo EP

Inside Nuxeo EP, pretty much everything is about Extension Point.

Extension points are used to let you contribute XML files to the Nuxeo components.

This means you can use the extension point system :

- to define a new Document Type,
- to hide a button from the default UI that you want to remove,
- to change the condition that make a particular view available,
- to add a new navigation axis,
- to change the way the Documents listings are displayed,
- ...

So before going further, you may want to take a look at the [Component model overview](#) 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 [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.

RELATED TOPICS

[Component model overview](#)

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 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.

Table of contents:

- [Schemas](#)
- [Facets](#)
- [Structural document types](#)
- [UI document types](#)
 - [General information](#)
 - [Facelet views](#)
 - [Layout](#)
 - [Containment rules](#)

- [Summary](#)

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 be 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 is 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 be 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.

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>
```

Deprecated layout configuration



This is the old and deprecated layout configuration that has been replaced by the one described above. If present, it is used instead of the new configuration for compatibility purposes.

This **layout** (without a final **s**) configuration section defines a series of widgets, that describe what the standard layout of this document type will be.

Here we define four widgets, displayed as simple input fields or as a text area.

```
<type id="Sample">
  ...
  <layout>
    <widget jsfcomponent="h:inputText"
      schemaname="dublincore" fieldname="title"
      required="true" />
    <widget jsfcomponent="h:inputTextarea"
      schemaname="dublincore" fieldname="description" />
    <widget jsfcomponent="h:inputText"
      schemaname="sample" fieldname="sample1" />
    <widget jsfcomponent="h:inputText"
      schemaname="sample" fieldname="sample2" />
  </layout>
  ...
</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.

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">
  <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>
      <layouts mode="any">
        <layout>heading</layout>
        <layout>note</layout>
      </layouts>
    </type>
    <type id="Folder">
      <subtypes>
        <type>Sample</type>
      </subtypes>
    </type>
    <type id="Workspace">
      <subtypes>
        <type>Sample</type>
      </subtypes>
    </type>
  </extension>
</component>

```

Document, form and listing views

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 EP-based application use layouts and content views.

In this section:

- [Layouts \(forms and views\)](#)
 - [Manage layouts](#)
 - [Document layouts](#)
 - [Layout display](#)
 - [Standard widget types](#)
 - [Custom templates](#)
 - [Custom widget types](#)
 - [Generic layout usage](#)
- [Content views](#)
 - [Custom Page Providers](#)
 - [Page Providers without Content Views](#)
- [Views on documents](#)

Layouts (forms and views)

Let our artists go wild on imaginative page layouts.

– Grant Morrison

Layouts are used to generate pages rendering from an xml configuration.

In a document oriented perspective, layouts are mostly used to display a document metadata in different use cases: present a form to set its schemas 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.

In this chapter we will see how to define a layout, link it to a document type, and use it in XHTML pages.

Layouts

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

It's all the same machine, right? The Pentagon, multinational corporations, the police! You do one little job, you build a widget in Saskatoon and the next thing you know it's two miles under the desert, the essential component of a death machine!

– Holloway, Cube

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.

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 that will be used to render it in a given mode.

Modes

Both layouts and widgets have modes.

The layout modes can be anything although some default modes are included in the application: create, edit, view, listing and search. Since 5.4.2, some new default modes are included: bulkEdit, header, csv, pdf and plain.

The widget modes are more restricted and widget types will usually only handle two modes: edit and view. The widget mode is computed from the layout mode following this rule: if the layout is in mode create, edit or search, the widget will be in edit mode. Otherwise the widget will be in view mode. Since 5.4.2, new widget modes have been added: pdf, csv and plain. 'plain' is the new default mode, as it is very close to the view mode except it's not supposed to include HTML tags.



Since Nuxeo 5.4, the mapping between the layout mode and the widget more is more loose: if the layout is in mode create, edit, bulkEdit or search, **or if its mode starts with one of these mode names**, the widget will be in edit mode. Otherwise the widget will be in the default mode ('view' before 5.4.1, and 'plain' after).

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 so that it only displays 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.

The following pages explain how to work with layouts:

- [Manage layouts](#)
- [Document layouts](#)
- [Layout display](#)
- [Standard widget types](#)
- [Custom templates](#)
- [Custom widget types](#)
- [Generic layout usage](#)

Manage layouts

Custom layouts can be contributed to the web layout service, using its extension point. The layout definition is then available through the service to control how it will be displayed in a given mode.

Some JSF tags have been added to the Nuxeo ECM layout tag library to make them easily available from an xhtml page.

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>
```

Layout definition

The above layout definition is used to display the title and the description of a document. Here are its properties:

- **name:** string used as an identifier. In the example, the layout name is "heading".
- **templates:** 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:** definition about what 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. 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. Since 5.1.7 and 5.2.0, it will be searched in the global widget registry before being ignored. This new feature is a convenient way to share widget definitions between layouts. Please refer the widget definition section.

Widget definition

Two widget definitions are presented on the above example. Let's look into the "title" widget and present its properties:

- name: 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 here.
- labels: 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: 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.dublicore.title" message will be translated at rendering time. Default is true.
- fields: 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 "dublicore" 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. Side note: 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}`.
- properties: 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 properties can be set on a widget:

- helpLabels: list that follows the same pattern as labels, but used to set help labels.
- widgetModes: list of local modes used to override the local mode (from the layout).
- subWidgets: list of widget definitions, as the widget list, used to describe sub widgets use to help the configuration of some complex widget types.

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>
```

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>
  </extension>
</component>
```

```

</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:basicDateAndTimeFormatter()}</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>

```



```
        </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 <https://github.com/nuxeo/nuxeo-jsf/blob/release-5.5/nuxeo-platform-webapp-base/src/main/resources/OSGI-INF/layouts-listing-contrib.xml>.

More information about how to write a listing layout template can be read in chapter about [Custom 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.

EL expressions in layouts and widgets

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: value (and equivalent document) + levels and changing "value" context
 - layoutValue: represents the value (evaluated) passed in a "nxl:layout" or "nxl:documentLayout" tag attributes.
 - layoutMode: represents the mode (evaluated) passed in a "nxl:layout" or "nxl:documentLayout" tag attributes.
 - value: represents the current value as manipulated by the tag: in a "nxl:layout" tag, it will represent the value resolved from the "value" tag attribute ; in a "nxl:widget" tag, 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, 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 its possible to access the generated layout object.
- Inside a "nxl:layoutRow", or equivalent "nxl:layoutColumn" 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.
- Inside a "nxl:layoutRowWidget", or equivalent "nxl:layoutColumn" widget, the variables "widget" and "widgetIndex" are available to access the generated current widget, and its index in the row or column. The variables added the level information are also available: widget_0, widget_1, ... and widgetIndex_0, widgetIndex_1... This is useful when needed to use the widget as defined in a higher level.
- Inside a widget template, some "field_n" variables are available: "field_0" represents the resolved first field value, "field_1" the second value, etc... Since 5.3.1, the variable "field" is available as an alias to "field_0". Since 5.3.2, 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 complete reference is available at <http://community.nuxeo.com/api/nuxeo/release-5.5/tlddoc/nxl/tld-summary.html>.

Document layouts

Layouts can be linked to a document type definition by specifying the layout names:

```
<layouts mode="any">
  <layout>heading</layout>
  <layout>note</layout>
</layouts>
```

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.

Since 5.2.GA, 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>
```

Since 5.3.1, 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](#).

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.

Layout display

Layouts can be displayed thanks to a series of JSF tags that will query the web layout service to get the layout definition and build it for a given mode.

For instance, we can use the `documentLayout` tag 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>
```

Since 5.4.2, 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" />
```

We can also display a specific layout for a document, even if it is not specified in the document type definition:

```
<div xmlns="http://www.w3.org/1999/xhtml"
      xmlns:nxl="http://nuxeo.org/nxforms/layout">
  <nxl:layout name="heading" mode="view" value="#{currentDocument}" />
</div>
```



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 `nxu:repeat` tag and handle all the `<table>` works 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>
```

Please refer to the tag library documentation available at <http://community.nuxeo.com/api/nuxeo/release-5.5/tlddoc/nxl/tld-summary.html>.

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/release-5.5/tlddoc/> for nuxeo jsf tags.

text

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. 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 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 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>.

textarea

The textarea widget displays a textarea in create or edit mode, with 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 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>.

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.

file

The file widget displays a file uploader/editor in create or edit mode, with 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 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 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 multi selection of directory entries in create or edit mode, with 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>.

list

The `list` widget displays an editable list of items in create or edit mode, with additional message tag for errors, and the same list of items in other modes. Items are defined using sub widgets 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.

View online demo: <http://layout.demo.nuxeo.org/nuxeo/layoutDemo/listWidget>

checkbox

The `checkbox` widget displays a checkbox in create, edit and any other mode, with 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>.

Custom templates

Some templating feature 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.

```
<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"
  id="#{layout.id}">

<c:if test="#{layout.mode != 'edit' and layout.mode != 'create' and layout.mode !=
'bulkEdit'}">

  <table class="dataInput">
    <tbody>

      <nxl:layoutRow>
```

```

<tr>
  <nxl:layoutRowWidget>
    <c:choose>
      <c:when test="#{widget.translated}">
        <td class="iconColumn" style="vertical-align:top;">
          <c:if test="#{!empty widget.helpLabel}">
            <h:graphicImage value="/icons/lightbulb.png"
              onmouseover="tooltip.show('{#{messages[widget.helpLabel]}}',
500);"
              onmouseout="tooltip.hide();" />
          </c:if>
        </td>
        <td class="labelColumn">
          <h:outputText value="#{messages[widget.label]}" />
        </td>
      </c:when>
      <c:otherwise>
        <td class="iconColumn" style="vertical-align:top;">
          <c:if test="#{!empty widget.helpLabel}">
            <h:graphicImage value="/icons/lightbulb.png"
              onmouseover="tooltip.show('{#{widget.helpLabel}}', 500);"
              onmouseout="tooltip.hide();" />
          </c:if>
        </td>
        <td class="labelColumn">
          <h:outputText value="#{widget.label}" />
        </td>
      </c:otherwise>
    </c:choose>
    <td class="fieldColumn" colspan="#{nxu:test(layoutRow.size==1,
2*layout.columns-1, 1)}">
      <nxl:widget widget="#{widget}" value="#{value}" />
    </td>
  </nxl:layoutRowWidget>
</tr>
</nxl:layoutRow>

</tbody>
</table>

</c:if>

<c:if test="#{layout.mode == 'edit' or layout.mode == 'create' or layout.mode ==
'bulkEdit'}">

  <table class="dataInput">
    <tbody>

      <nxl:layoutRow>
        <tr>
          <nxl:layoutRowWidget>
            <c:choose>
              <c:when test="#{widget.translated}">
                <td class="iconColumn" style="vertical-align:top;">
                  <c:if test="#{!empty widget.helpLabel}">
                    <h:graphicImage value="/icons/lightbulb.png"
                      onmouseover="tooltip.show('{#{messages[widget.helpLabel]}}',
500);"
                      onmouseout="tooltip.hide();" />

```

```

        </c:if>
      </td>
      <td class="labelColumn">
        <h:outputText value="#{messages[widget.label]}"
          styleClass="#{nxu:test(widget.required, 'required', '')}" />
      </td>
    </c:when>
    <c:otherwise>
      <td class="iconColumn" style="vertical-align:top;">
        <c:if test="#{!empty widget.helpLabel}">
          <h:graphicImage value="/icons/lightbulb.png"
            onmouseover="tooltip.show('#{widget.helpLabel}', 500);"
            onmouseout="tooltip.hide();" />
        </c:if>
      </td>
      <td class="labelColumn">
        <h:outputText value="#{widget.label}"
          styleClass="#{nxu:test(widget.required, 'required', '')}" />
      </td>
    </c:otherwise>
  </c:choose>
  <td class="fieldColumn" colspan="#{nxu:test(layoutRow.size==1,
2*layout.columns-1, 1)}">
    <nxl:widget widget="#{widget}" value="#{value}" />
  </td>
</nxl:layoutRowWidget>
</tr>
</nxl:layoutRow>

</tbody>
</table>

</c:if>

```

```
</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" or "create" 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.

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<DocumentModel>
    named "documents"

    Other needed parameters are:
    - provider: instance of a PageProvider<DocumentModel> to handle sort
    - layoutListingStatus: iteration status, used to print table header
      matching widget label.
  </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>
            <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('#{contentView.name}',
this.checked)"
        reRender="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('#{contentView.name}',
this.checked)"

action="#{documentListingActions.checkCurrentDocAndProcessSelectPage(contentView.name,
contentView.selectionListName, documents.selected, currentDocument.ref)}"
        reRender="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">
            <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>
            <f:verbatim>&nbsp;</f:verbatim>
            <c:if test="#{ascIndex != -1}">
                <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_down.png" />
            <c:if test="#{hasSeveralSorts}">
                #{ascIndex + 1}
            </c:if>
            </h:commandLink>
        </c:if>
        <c:if test="#{descIndex != -1}">
            <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_up.png" />
        <c:if test="#{hasSeveralSorts}">
            #{descIndex + 1}
        </c:if>
        </h:commandLink>
    </c:if>
    <c:if test="#{ascIndex == -1 and descIndex == -1}">
        <h:commandLink immediate="true"

action="#{provider.addSortInfo(layoutColumn.properties.sortPropertyName, true)}"
        id="#{layoutColumn.widgets[0].name}_header_sort_add">
        <h:graphicImage value="/icons/sort_down.png" />
        </h:commandLink>
    </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>
</th>
</nxl:layoutColumn>
<c:if test="#{provider.sortable}">
    <th>
        <h:graphicImage value="/icons/lightbulb.png"
            onmouseover="tooltip.show('#{messages['contentview.sort.help']}', 200,
'topleft');"
            onmouseout="tooltip.hide();" />
    </th>
</c:if>
</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}">
            <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>

```

```
<c:if test="#{provider.sortable}">
  <td class="iconColumn">
    </td>
  </c:if>
</tr>
```

```
</f:subview>
```

As you can see, this layout make 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,...

Any custom template can be defined following this example to handle additional properties to display on the final table header and columns.

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.

Since 5.4.2, even 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 Nuxeo.

```
<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 [E L 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.

With nuxeo version <= 5.3.0, to handle a list of complex properties (each entry of the list is a map with keys 'name' and 'email' for instance), you can use the definition:

```
<widget name="employees" type="list">
  <fields>
    <field>company:employees</field>
  </fields>
  <subWidgets>
    <widget name="employee" type="template">
      <labels>
        <label mode="any"></label>
      </labels>
      <fields>
        <field></field>
      </fields>
      <properties mode="any">
        <property name="template">
          /widgets/complex_widget_template.xhtml
        </property>
      </properties>
      <!-- subwidgets for complex -->
      <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>
  </subWidgets>
</widget>
```

With nuxeo version > 5.3.0, to handle a list of complex properties (each entry of the list is a map with keys 'name' and 'email' for instance), you can use the 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="name" type="text">
          <fields>
            <field>name</field>
          </fields>
        </widget>
        <widget name="email" type="text">
          <fields>
            <field>email</field>
          </fields>
        </widget>
      </subWidgets>
    </widget>
  </subWidgets>
</widget>
```

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 since nuxeo version 5.2 GA.

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>
```

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:

```
<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

Layout Widget Editor

Label [\[?\]](#)

title

Help label [\[?\]](#)

Translated [\[?\]](#)

☐

Read only [\[?\]](#)

☐

► Advanced mode configuration [\[?\]](#)

Widget Type

Text

[\[?\] Online demo](#)

Required

☐


Max length

Size

Style

Style class

Cancel

 Save

- documentation of available layouts and widget types on a given Nuxeo instance (see on your Nuxeo instance: <http://localhost:nuxeo/site/layout-manager/>, <http://localhost:nuxeo/site/layout/> before 5.5)

dev
JSON definitions 5.4.0 5.4.1 5.4.2
Layout Template
document
JSON definitions 5.4.0 5.4.1 5.4.2
Checkbox Complex Datetime Decimal number Duration File HTML text Hidden Integer List Multiple vocabulary Secret Text Textarea Vocabulary

TEXT

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.

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.

General Information

Categories: document
Widget type name: text

Links

[JSON definition](#)

Sample JSON export URLs:

Before 5.5:

http://localhost:8080/nuxeo/site/layout/widgets/widgetTypes/document	all widget types with category "document"
http://localhost:8080/nuxeo/site/layout/widgets/widgetTypes/document?version=5.4.0	all widget types with category "document", filtering widget types with a version strictly higher than 5.4.0

Since 5.5:

http://localhost:8080/nuxeo/site/layout-manager/widgets/widget-types/document	all widget types with category "document"
http://localhost:8080/nuxeo/site/layout-manager/widgets/widget-types/document?version=5.4.0	all widget types with category "document", filtering widget types with a version strictly higher than 5.4.0
http://localhost:8080/nuxeo/site/layout-manager/widgets/widget-types?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://layout.demo.nuxeo.org>)

Text

Overview
Reference
Preview

View mode
Edit mode

Properties

Preview

Change the properties in the form below to preview the generated widget.

Label
My widget label

Help label
My widget help label

Translated
☐

Style

Style class

Escape
☐

Custom properties
Add

Submit

My widget label
Some sample text

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
        <h:inputText /> tag in create or edit mode, and properties
        accepted on a <h:outputText /> 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">

```

Copyright © 2010-2016 Nuxeo.
This documentation is published under Creative Common BY-SA license. More details on the [Nuxeo Documentation License page](#).

90

```

<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: 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.

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.

Content views

 Content views are available in Nuxeo since version 5.4.

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:

- define the NXQL query that will be used to retrieve the documents, filtering some of them (documents in the trash for instance)
- pass on contextual parameters to the query (the current container identifier)
- define a filtering form to refine the query
- define what columns will be used for the rendering of the list, and how to display their content
- handle selection of documents, and actions available when selecting them (copy, paste, delete...)
- handle sorting and pagination
- handle caching, and refresh of this cache when a document is created, deleted, modified...

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.png" />
    </resultLayouts>

    <selectionList>CURRENT_SELECTION</selectionList>
    <actions category="CURRENT_SELECTION_LIST" />

  </contentView>

</extension>
```

The content view query

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.

This 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.

This element 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.

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 infos 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).

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" and is available since version 5.4.2. 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.

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_filter">

    <coreQueryPageProvider>
      <property name="coreSession">#{documentManager}</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>

```

This 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 be added to the query string. This fixed part can also take parameters using the "?" character and "parameter" elements.

The "searchLayout" element defines what layout 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'll 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.

Since 5.4.2, the "showFilterForm" element makes it possible to show this form above the content view results.



Since 5.4.2, the "searchDocument" variable can be used in EL expressions to bind the page size, the sort infos 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 content view cache

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 fixed cache key can be used to cache only one instance.

Caching only one instance is useful as several features may need to retrieve information from a given content view. When caching only one instance, setting the "cacheSize" element to more than "1" is useless: 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.

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.



The selection actions rely on the cached content view to retrieve documents from the list, so using a cache of size "0" will make these actions fail: a cache of at least 1 is required. If caching options are not enough for you to force refresh, it is also possible to force the refresh at rendering, for instance by calling `contentView#refreshPageProvider` in the xhtml template.

The "refresh" cache makes it possible to refresh 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)`.

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.

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

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 watch 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>
```

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 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" is available since version 5.4.2. It 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 available since version 5.4.2. They are used to display the message stating that there are no elements in the content view.

The elements "showPageSizeSelector" and "showRefreshCommand" are available since version 5.4.2. They 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 caching will take effect when using the "contentViewActions" bean. Although the cache key, cache size and events allow to perform the most common use cases, it is sometimes useful to call this bean methods directly when forcing a refresh.

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.

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.Con`

tentViewActions) and "documentContentViewActions" (org.nuxeo.ecm.webapp.contentbrowser.DocumentContentViewActions) 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-5.5/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>
```

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

 Page providers are available in Nuxeo since version 5.4.

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.

Views on documents

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 Nuxeo EP (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 definitions:

```
<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 EP 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. As mentioned before, 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>
```

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.

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 Nuxeo EP, the Check In operation that creates a new version removes the Working Copy, whose role has been

fulfilled. This is not the case in Nuxeo EP, 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 Nuxeo EP 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.

User Actions (links, buttons, icons, tabs)

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).

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.

Register a new action

An action can be registered using the following example extension:

```
<extension target="org.nuxeo.ecm.platform.actions.ActionService"
  point="actions">

  <action id="logout" link="#{loginLogoutAction.logout}"
    label="command.logout">
    <category>USER_SERVICES</category>
  </action>

</extension>
```

The above action will be used to display a "logout" link on the site. Here are its properties:

- **id**: string identifying the action. In the example, the action id is "logout".
- **label**: the action name that will be used when displaying the link. In the example, the label is "command.logout". This label is a message that will be translated at display.
- **link**: string representing the command the action will trigger. This string may represent a different action given the template that will display the action. In the example, a JSF command link will be used, so it represents an action method expression. The seam component called "loginLogoutAction" holds a method named "logout" that will perform the logout and return a string for navigation.
- **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 "USER_SERVICES". It is designed to group all the actions that will be displayed on the right top corner of any page of the site.

Other properties can be used to define an action. They are listed here but you can have a look at the main actions contributions file for more examples: `nuxeo-platform-webapp-core/srs/main/resources/OSGI-INF/actions-contrib.xml`.

- **filter-ids**: id of a filter that will be used to control the action visibility. An action can have several filters: it is visible if all its filters grant the access.
- **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.
- **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. This attribute may be deprecated in the future.
- **enabled**: boolean indicating whether the action is currently active. This can be used to hide existing actions when customizing the site

behavior.

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.

Manage category to display an action at the right place

Actions in the same category are supposed to be displayed in the same area of a page. Here are listed the main default categories if you want to add an action there:

- **USER_SERVICES**: used to display actions in the top right corner of every page. The link attribute should look like a JSF action command link. See the example above.
- **VIEW_ACTION_LIST**: used for tabs displayed on every document. As each tab is not displayed in a different page, but just includes a specific template content in the middle of the page, the link attribute has to be a template path. For instance:

```
<action id="TAB_VIEW" link="/incl/tabs/document_view.xhtml" enabled="true"
  order="0" label="action.view.summary">
  <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">
  <category>VIEW_ACTION_LIST</category>
  <filter-id>view_content</filter-id>
</action>
```

- **SUBVIEW_UPPER_LIST**: used to display actions just below a document tabs listing. As **USER_SERVICES**, these actions will be displayed using a command link, so the link attribute has to be an action method expression. For instance:

```
<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>

<action id="newDocument" link="select_document_type" enabled="true"
  label="action.new.document" icon="/icons/action_add.gif">
  <category>SUBVIEW_UPPER_LIST</category>
  <filter-id>create</filter-id>
</action>
```

Manage 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.

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: boolean indicating whether this is a granting rule or a denying rule.
- permission: 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: 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: 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: 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: 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: 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.



In EL expressions representing conditions, Seam component names can only be used when placed at the beginning of the expression. As a result, it is not possible to use several seam components in a single expression.

Adapt templates to display an action

It is important to understand that an action does **not** define the way it will be rendered: This is left to pages, templates and other components displaying it. Most of the time, actions will be rendered as command links or command buttons.

For instance, actions using the `USER_SERVICES` category will be rendered as action links:

```
<nxu:methodResult name="actions"
  value="#{webActions.getActionsList('USER_SERVICES')}">
  <nxu:dataList layout="simple" var="action" value="#{actions}"
    rowIndexVar="row" rowCountVar="rowCount">
    <h:outputText value=" | " rendered="#{row!=(rowCount-1)}" />
    <nxx:commandLink action="#{action.getLink()}">
      <t:htmlTag value="br" rendered="#{row==(rowCount-1)}" />
      <h:outputText value="#{messages[action.label]}" />
    </nxx:commandLink>
  </nxu:dataList>
</nxu:methodResult>
```

The `nxu:methodResult` tag is only used to retrieve the list of actions declared for the `USER_SERVICES` category. The `nxx:commandLink` is used instead of a simple `h:commandLink` so that it executes commands that where described as action expression methods.

Another use case is the document tabs: actions using the `VIEW_ACTION_LIST` category will be rendered as action links too, but actions are managed by a specific seam component that will hold the information about the selected tab. When clicking on an action, this selected tab will be changed and the link it points to will be displayed.

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

- Concepts
- Registering an Event Listener
- Processing an Event using an Event Listener
- Sending an Event
- Handling errors
- Common Events
 - Basic Events
 - Copy/Move Events
 - Versioning Events
 - Publishing Events
- Asynchronous vs Synchronous listeners
 - Performances and monitoring

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="dcllistener" 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
    }
    ...
```

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

- **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.
- **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.
- **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).
- **documentModified**: a document has been saved after a modification.
- **beforeDocumentSecurityModification**: a document's ACLs are about to change.
- **documentSecurityUpdated**: a document's ACLs have changed.
- **aboutToRemove** / **aboutToRemoveVersion**: a document or a version are about to be removed.
- **documentRemoved** / **versionRemoved**: a document or a version have been removed.
- **documentLocked** / **documentUnlocked**: a document has been locked or unlocked.
- **sessionSaved**: the session has been saved (all saved documents are written to database).
- **lifecycle_transition_event**: a transition has been followed on a document.

Copy/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

- **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.
- **beforeRestoringDocument**: a document is about to be restored.
- **documentRestored**: a document has been restored.

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:

- **documentPublished**: a document has been published (event sent for the base document being published and for the published document).
- **documentWaitingPublication**: a document has been published but is not approved yet (event sent for the base document being published and for the published document).
- **documentPublicationApproved**: a document waiting for approval has been approved.

- **documentPublicationRejected**: a document waiting for approval has been rejected.
- **documentUnpublished**:

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

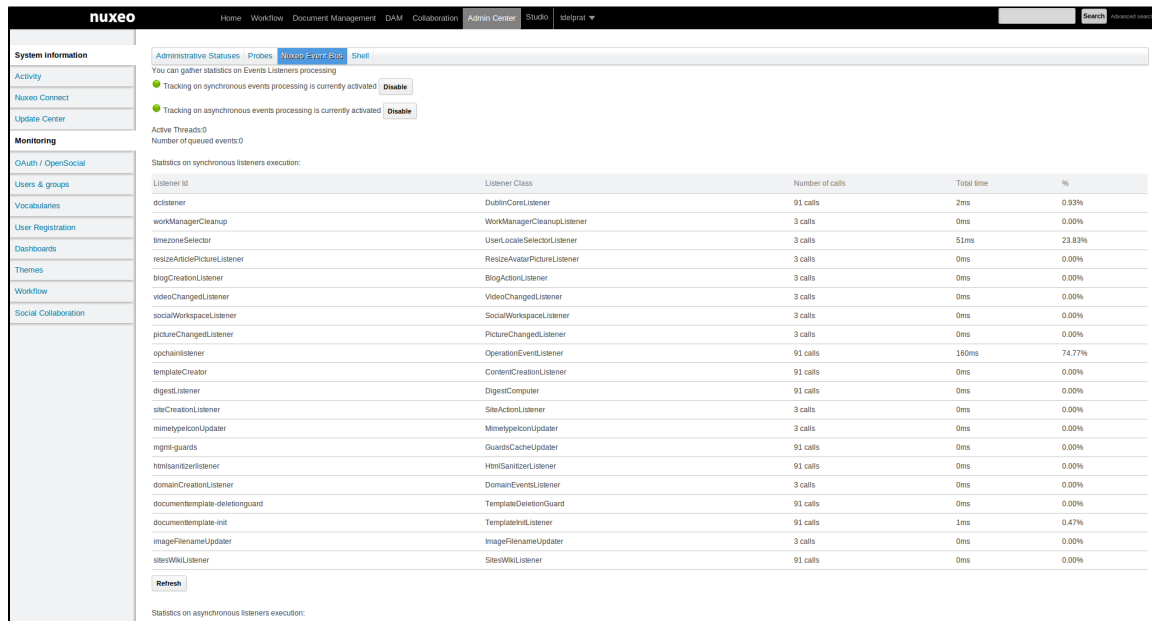
However, it is easy to have a single listener class that exposes both interfaces and can be use both synchronously and asynchronously.

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 screenshot shows the Nuxeo Admin Center interface. The left sidebar contains navigation links: System information, Activity, Nuxeo Connect, Update Center, Monitoring, OAuth / OpenSocial, Users & groups, Vocabularies, User Registration, Dashboards, Themes, Workflow, and Social Collaboration. The main content area is titled 'Monitoring' and shows 'Statistics on synchronous listeners execution:'. It includes a table with columns: Listener Id, Listener Class, Number of calls, Total time, and %. Below the table is a 'Refresh' button and a section for 'Statistics on asynchronous listeners execution:'.

Listener Id	Listener Class	Number of calls	Total time	%
dcListener	DublinCoreListener	91 calls	2ms	0.93%
workManagerCleanup	WorkManagerCleanupListener	3 calls	0ms	0.00%
timezoneSelector	UserLocaleSelectorListener	3 calls	51ms	23.83%
resizeArticlePictureListener	ResizeAvatarPictureListener	3 calls	0ms	0.00%
blogCreationListener	BlogActionListener	3 calls	0ms	0.00%
videoChangedListener	VideoChangedListener	3 calls	0ms	0.00%
socialWorkspaceListener	SocialWorkspaceListener	3 calls	0ms	0.00%
pictureChangedListener	PictureChangedListener	3 calls	0ms	0.00%
opchainListener	OperationEventListener	91 calls	160ms	74.77%
templateCreator	ContentCreationListener	91 calls	0ms	0.00%
digestListener	DigestComputer	91 calls	0ms	0.00%
siteCreationListener	SiteActionListener	3 calls	0ms	0.00%
minetypeconUpdater	MimetypeconUpdater	3 calls	0ms	0.00%
mgmt-guards	GuardsCacheUpdater	91 calls	0ms	0.00%
htmlsanitizerListener	HtmlSanitizerListener	91 calls	0ms	0.00%
domainCreationListener	DomainEventsListener	3 calls	0ms	0.00%
documentTemplate-deletionGuard	TemplateDeletionGuard	91 calls	0ms	0.00%
documentTemplate-init	TemplateInitListener	91 calls	1ms	0.47%
imageFilenameUpdater	ImageFilenameUpdater	3 calls	0ms	0.00%
sitesWikiListener	SitesWikiListener	91 calls	0ms	0.00%

- 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.

Scheduling periodic events

Overview

The Scheduler Service is a Nuxeo EP service to schedule events at periodic times. This is the best way in Nuxeo EP 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.platform.scheduler.core.service.SchedulerRegistryService` component.

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.platform.scheduler.core.service.SchedulerRegistryService"
    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 0,15,30,35 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](#). A tutorial is also available [here](#).

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 * ?
```

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.

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 dublicore schema, 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** which 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.

Directories and Vocabularies

In Nuxeo EP, 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 Nuxeo EP itself (therefore allowing shared management and usage).

A **vocabulary** is a specialized **directory** with only a few important columns that are used by Nuxeo EP to display things like menus and selection lists.

Table of contents:

- [SQL directories](#)
- [LDAP directories](#)
- [Multi-directories](#)
- [References between directories](#)
 - [Static reference as a dn-valued LDAP attribute](#)
 - [Dynamic reference as a ldapUrl-valued LDAP attribute](#)
 - [LDAP tree reference](#)
 - [Defining inverse references](#)
 - [References defined by a many-to-many SQL table](#)

SQL directories

SQL directories read and store their information in a SQL database. They are defined through the `directories` 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`, `subinitial` or `subfinal`; this is used in most UI searches,

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 Nuxeo EP 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 who 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 behaviour.
- **readOnly**: boolean value. 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. This is useless to provided if **readOnly** attribute is set to false.
- **creationClass**: use as many tag as needed to specify which class are used to defined new people entries in 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>(&!(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 from the children entries. It will be stored in a list, even if there can be only 0 or 1 parent.

WARNING: 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

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>
    </rows>
  </layout>

```

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>
```

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).

Nuxeo authentication framework is pluggable so that you can contribute new plugin and don't have to rewrite and reconfigure a complete JAAS infrastructure.

On this page

- [Pluggable JAAS Login Module](#)
- [NuxeoLoginModule](#)
- [NuxeoLoginModule Plugins](#)
- [Pluggable Web Authentication Filter](#)
 - [NuxeoAuthenticationFilter](#)
 - [Built-in Authentication Plugins](#)
 - [Additional Authentication Plugins](#)
 - [CAS2 Authentication](#)
 - [PROXY_AUTH: Proxy based Authentication](#)
 - [NTLM_AUTH: NTLM and IE challenge/response authentication](#)
 - [PORTAL_AUTH: SSO implementation for portal clients](#)
 - [ANONYMOUS_AUTH: Anonymous authentication plugin](#)
- [Related pages](#)

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:

- 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 than 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 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 informations 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 an 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 `UserIdentificationInfo`, 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.defaultConfig</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.

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.defaultConfig</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.java: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(NuxeoAuthenticationFilter.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 do 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 `$TOMCAT_HOME/nxserver/bundles/` or `$JBASS_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.defaultConfig</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.defaultConfig</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&nbsp];
            <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. See the Nuxeo-Http-client-library for more information.

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.defaultConfig</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 `$TOMCAT_HOME/nxserver/bundles` or `$JBOSSE_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>
```

Related pages

- Configuring User & Group Storage and Authentication (Nuxeo Installation and Administration - 5.8)
- Configure User & Group storage and Authentication (Nuxeo Installation and Administration - 5.5)
- Authentication, users and groups (Nuxeo Installation and Administration - 5.5)
- Configure User & Group storage and Authentication (Nuxeo Installation and Administration - 5.6)
- Configuring User & Group Storage and Authentication (Nuxeo Installation and Administration — 6.0)
- Authentication, users and groups (Nuxeo Installation and Administration - 5.5)
- Authentication, users and groups (Nuxeo Installation and Administration — 6.0)
- Using Shibboleth (Nuxeo Installation and Administration - 5.8)
- Using CAS2 authentication (SQL case) (Nuxeo Installation and Administration - 5.5)
- Using OAuth (Nuxeo Installation and Administration - 5.5)
- Using a LDAP directory (Nuxeo Installation and Administration - 5.5)
- Using Shibboleth (Nuxeo Installation and Administration - 5.5)
- Authentication, users and groups (Nuxeo Installation and Administration - 5.6)
- Using OAuth (Nuxeo Installation and Administration - 5.6)
- Using OAuth (Nuxeo Installation and Administration — 6.0)

Showing first 15 of 41 results
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,

User Management

In Nuxeo Platform, the concept of a user i

- users are needed for authentication and e
- users have associated information that ca

An abstraction, the
userManager
, centralizes the way a Nuxeo Platform ap
LoginModule
when someone attempts to authenticate e

On this page

- Users and Groups configuration
- Example of User Manager configuration
- Schema definition
- User Manager definition
- Directory definition
- Simple case
- LDAP case
- UserManager
- Simple case
- Configuring the User Manager with anonymous user and other virtual users
- User and Group display
- User Layout definition
- User Layout definition
- Related pages

Users and Groups configuration

The data about users (login, password, name, personal information, etc.) and the

- a part of user can be stored into an LDAP server and into SQL, and SQL can store groups,
- ...

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 schemas, 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 definition into explorer.nuxeo.com, [here](#). 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 schemas can now be used when we define a new directory, called "MyUserDirectory". Here, you will have a SQL directory:

```
<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.

```
<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>
```

Here is a simple LDAP directory 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>
```

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 schemas 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>
      </rows>
```

```

<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">
\#{userManagerActions.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>
```

User Layout definition

Before 5.2.GA, you need to redefine the deprecated layout configuration of two standard document types, User and UserCreate (which are used in the default user management screens and backing beans) to add your new field:

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

<type id="User" coretype="User">
<label>User</label>
<icon>/icons/user.gif</icon>
<default-view>view_user</default-view>
<layout>
<widget schemaname="myuser" fieldname="username"
jsfcomponent="h:inputTextReadOnly" />
<widget schemaname="myuser" fieldname="firstName"
jsfcomponent="h:inputText" />
<widget schemaname="myuser" fieldname="lastName"
jsfcomponent="h:inputText" />
<widget schemaname="myuser" fieldname="email"
jsfcomponent="h:inputText" />
<widget schemaname="myuser" fieldname="company"
jsfcomponent="h:inputText" />
<widget schemaname="myuser" fieldname="petName"
jsfcomponent="h:inputText" />
</layout>
</type>

<type id="UserCreate" coretype="UserCreate">
<label>UserCreate</label>
<icon>/icons/user.gif</icon>
<default-view>create_user</default-view>
<layout>
<widget schemaname="myuser" fieldname="username"
jsfcomponent="h:inputText" required="true" />
<widget schemaname="myuser" fieldname="password"
jsfcomponent="h:inputSecret" required="true" />
<widget schemaname="myuser" fieldname="firstName"
jsfcomponent="h:inputText" />
<widget schemaname="myuser" fieldname="lastName"
jsfcomponent="h:inputText" />
<widget schemaname="myuser" fieldname="email"
jsfcomponent="h:inputText" required="true" />
<widget schemaname="myuser" fieldname="company"
jsfcomponent="h:inputText" />
<widget schemaname="myuser" fieldname="petName"
jsfcomponent="h:inputText" />
</layout>
</type>

</extension>
```

Related pages



- Layouts and Widgets (Forms, Listings, Grids) (Nuxeo Platform Developer Documentation)
- Authentication, Users and Groups (Nuxeo Platform Developer Documentation)
- Shibboleth Authentication (Nuxeo Platform Developer Documentation)
- Using CAS2 Authentication (Nuxeo Platform Developer Documentation)
- Using a LDAP Directory (Nuxeo Platform Developer Documentation)
- Authentication and User Management (Nuxeo Platform Developer Documentation)

• Publisher service

Since Nuxeo DM 5.3GA, there are three ways to publish a document:

- on local sections, ie the sections created in your Nuxeo DM instance,
- on remote sections, ie the sections of a remote Nuxeo server,
- on the file system.

Publication is configured using the `PublisherService`.

On this page
<ul style="list-style-type: none"> • About the PublisherService • Configuring local sections publishing • Configuring remote sections publishing <ul style="list-style-type: none"> • Server configuration • Client configuration • Configuring file system publishing

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 [javadoc of the PublicationTree](#).

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 `RelativeRootPath` parameters.
- `RelativeRootPath`: used when you just want to define a relative path (without specifying the domain path). A `PublicationTree` instance 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">localhost</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.

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](#).

NXQL

NXQL syntax

The general syntax of a NXQL expression is:

```
SELECT (*|[DISTINCT] <select-clause>) FROM <from-clause> [WHERE <where-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>`.

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, cf [NXP-4339](#))
- `(<predicate>)`
- `NOT <predicate>`
- `<expression>`

An `<operator>` can be:

- AND
- OR
- =
- <> (or != for Java compatibility)
- <
- <=
- >
- >=

- [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 ("). Inside a string, special characters are escaped by a backslash (}), including for the quotes or double quotes contrary to the standard SQL syntax which would double them. The special { \n, \r, \t and { } 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, 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).

Complex property references

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 3rd 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/length` for the length of the 3rd file,
- `files/*/length` for any length
- `files/*1/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/name LIKE '%.txt' AND files/*1/length = 0
```

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

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

For simple lists, there is a slight difference in doing these two requests:

```
SELECT * FROM Document WHERE dc:subjects LIKE 'test%'
SELECT * FROM Document WHERE dc:subjects/* LIKE 'test%'
```

The first one internally uses a SQL EXISTS and a subquery, and also worked before Nuxeo 5.5. 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.

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>`.

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 for ordering.

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

ecm:isCheckedInVersion: 1 for versions and 0 for non-version (`DocumentModel.isVersion()`). This can only be compared to 1 or 0.

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

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

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 [Fulltext queries](#) for more.

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 filename = 'testfile.txt'
SELECT * FROM Document WHERE uid = 'isbn1234'
SELECT * FROM Document WHERE filename = 'testfile.txt' OR dc:title = 'testfile3_Title'
SELECT * FROM Document WHERE 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%'
```

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'
```

Fulltext 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.

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 `univers`es 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'`.

Advanced search

i This page gives information on how to configure the advanced search since version 5.4

On version 5.4, the advanced search is configured to work in conjunction with a [content view](#).

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](#)).

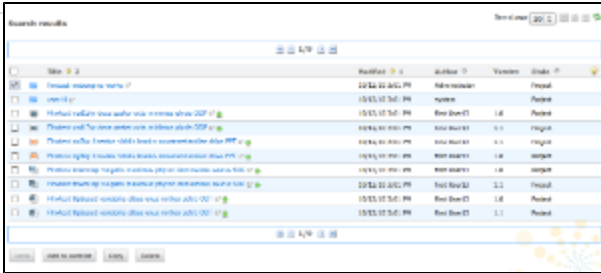
It also references the search layout (defined here: [search-layouts-contrib.xml](#)) and the result layouts (defined here: [layouts-listing-contrib.xml](#)).

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 infos 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

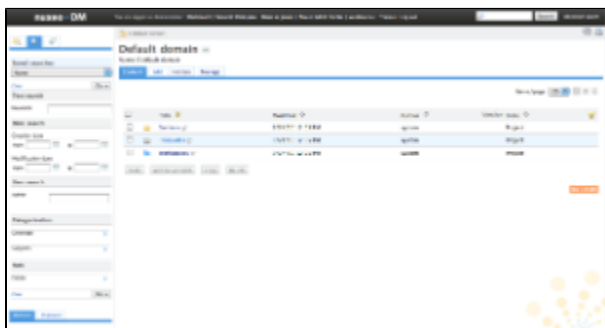


The Faceted Search is available in Nuxeo since version 5.4.

Overview

The faceted search adds a new way to browse the whole document repository using visual filters on metadata. The new tab appears on the left, just after the navigation tree.

Here is a screenshot of the default faceted search:



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 do that 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`: define which Document type to use to populate the values in the query
- `searchLayout`: define 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" />
        </predicate>
      </whereClause>
    </coreQueryPageProvider>
  </contentView>
</extension>
```

```

        <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 sub widgets. It displays the widget label, and list the sub widgets below according to the wrapperMode. The sub widgets can use 3 wrapperMode (to be defined in the sub widget properties):

- row: the sub widget label is displayed on one row, and the sub widget content on another row.
- column: the sub widget label and content are displayed on the same row (default mode if not specified)
- noLabel: the sub widget 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 2 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!

Smart search

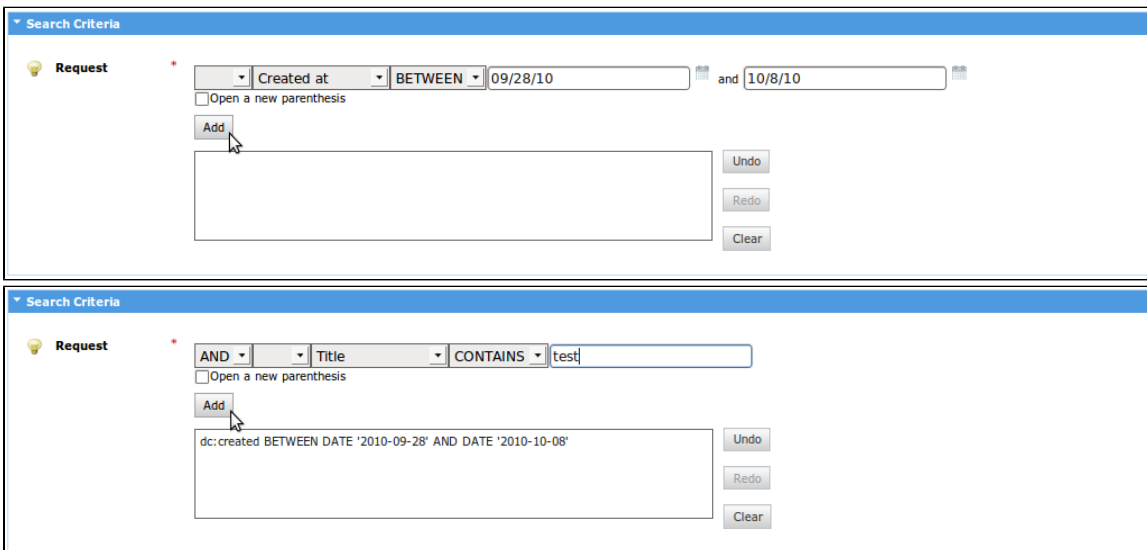
 The Smart Search addon is available in Nuxeo since version 5.4.

This addon adds UI features to build a query and save it in a document.

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 fo 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-5.5/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">
          &nbsp;#{messages['label.and']}&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'} w` 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.

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 to 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 recommended 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 two different methods of the interface:

- how security is checked on a given document (method `checkPermission`),
- how security is applied to NXQL searches (method `getQueryTransformer`).

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 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.

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` 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.

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);
        }
    }
}

```

```
}
```

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.

A Nuxeo Theme includes



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

- [Page layout](#)
- [Branding : use flavors and styles!](#)
 - [The pages extension point](#)
 - [The styles extension point](#)
 - [The flavor extension point](#)
- [Theme static resources](#)

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 explicitly 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` extension 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.

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.

On this page

- [Theme XML file](#)
 - [Page layouts](#)
 - [Page, section, cell and fragment formats](#)
 - [Styles](#)
- [Theme contributions: pages, flavors and styles](#)
 - [Pages](#)
 - [Flavors](#)
 - [Styles](#)
 - [Old palette contributions](#)

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> element.

```
<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>
```

Nuxeo UI Frameworks

Nuxeo uses several UI frameworks:

- [Seam and JSF webapp overview](#)
 - [JSF tips and howtos](#)
- [GWT Integration](#)
- [Extending The Shell](#)
 - [Shell Features](#)
 - [Shell Commands](#)
 - [Shell Namespaces](#)
 - [Shell Documentation](#)
- [WebEngine \(JAX-RS\)](#)
 - [Session And Transaction Management](#)
 - [WebEngine Tutorials](#)
- [Nuxeo Android Connector](#)
 - [Nuxeo Automation client](#)
 - [Android Connector and Caching](#)
 - [Android Connector additional Services](#)
 - [DocumentProviders in Android Connector](#)
 - [Android SDK Integration](#)
 - [Nuxeo Layout in Android](#)
 - [SDK provided base classes](#)
- [Nuxeo Flex Connector](#)
 - [AMF Mapping in Nuxeo](#)
 - [Build and deploy Nuxeo Flex Connect](#)
 - [Using Flex Connector](#)

Seam and JSF webapp overview

Nuxeo EP provides a web framework to build business applications for thin clients.

This framework is based on the standard JEE view technology: Java Server Faces.

Nuxeo JSF: technical stack

Nuxeo JSF framework integrates several technologies in order to make the development of Web Application fast and efficient.

Nuxeo JSF stack includes (as of Nuxeo 5.4) :

- 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

Inside Nuxeo EP, Seam Framework is used only for the JSF (client) layer.

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)

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 : Nuxeo 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 Nuxeo code for that, neither do you need to change the default Nuxeo 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 its 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 DM, and with simple configuration have a custom UI.

Nuxeo JSF : key concepts

JSF tips and howtos

Topics in this section:

- [Ajax forms and actions](#)
- [Back & next buttons paradigm and JSF in Nuxeo](#)
- [Error page customization](#)
- [JSF and Javascript](#)
- [JSF tag library registration](#)
- [JSF troubleshoot](#)

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">
  <nx1: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>
```

Note that 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 generate 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.

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).

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](#)).

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" />
```

Back & next buttons paradigm and JSF in Nuxeo

Nuxeo DM 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 Nuxeo 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.

Error page customization

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 page in a custom project. 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.

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>
```

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>
```

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 your custom taglib file `mylibrary.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

Although 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(Html
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+I. You'll get ajax debug logs in a popup window and hopefully will understand what is the problem.

Also make sure 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 recommendations 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", "iParam", "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.

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](#)
 - [Creating a Hello World Application](#)
- [Deploying the GWT Application on a Nuxeo Server](#)
- [Accessing your GWT module from the client](#)
 - [Using GWT RPC mechanism in a Nuxeo GWT module.](#)
 - [Using Other Server / Client Communication Mechanisms.](#)
 - [How is this working?](#)
- [Example of a pom.xml](#)

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:

- [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 `install()` 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.

Even 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](#)
- [Shell Commands](#)
- [Shell Namespaces](#)
- [Shell Documentation](#)

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:

```
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 pages



[Built-in Commands](#) (Nuxeo Installation and Administration - 5.5)



[Nuxeo Automation Commands](#) (Nuxeo Installation and Administration - 5.5)



[File System Commands](#) (Nuxeo Installation and Administration - 5.5)



[Nuxeo Server Commands](#) (Nuxeo Installation and Administration - 5.5)



[Shell Batch Mode](#) (Nuxeo Installation and Administration - 5.5)



[Shell Commands](#) (Nuxeo Enterprise Platform (EP) - 5.5)



[Shell Features](#) (Nuxeo Enterprise Platform (EP) - 5.5)



[Extending The Shell](#) (Nuxeo Enterprise Platform (EP) - 5.5)



[Shell Documentation](#) (Nuxeo Enterprise Platform (EP) - 5.5)



• 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 th child document type.

Define the command syntax

Shell Namespaces (Nuxeo Enterprise Platform (EP) - 5.5)



Configuration Commands (Nuxeo Installation and Administration - 5.5)

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.

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 required 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 specifies that it 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 activates 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 equivalent 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()` construct 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 [Shell 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.

The first solution is OK, when you are creating Ad-Hoc scripts - like invoking GC on the server. 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 pages



[Built-in Commands](#) (Nuxeo Installation and Administration - 5.5)



[Nuxeo Automation Commands](#) (Nuxeo Installation and Administration - 5.5)



[File System Commands](#) (Nuxeo Installation and Administration - 5.5)



[Nuxeo Server Commands](#) (Nuxeo Installation and Administration - 5.5)



[Shell Batch Mode](#) (Nuxeo Installation and Administration - 5.5)



[Shell Commands](#) (Nuxeo Enterprise Platform (EP) - 5.5)

• 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



[Shell Features](#) (Nuxeo Enterprise Platform (EP) - 5.5)



[Extending The Shell](#) (Nuxeo Enterprise Platform (EP) - 5.5)



[Shell Documentation](#) (Nuxeo Enterprise Platform (EP) - 5.5)



[Shell Namespaces](#) (Nuxeo Enterprise Platform (EP) - 5.5)



[Configuration Commands](#) (Nuxeo Installation and Administration - 5.5)

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.

To explain how a namespace is changing the shell prompt I will give the example of the **local** namespace provided by the `FileSystem` feature. It's really easy: the `FileSystem` feature is registering and activating the **local** namespace which is implemented by a subclass of `CommandRegistry`. 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** namespace):

```
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 pages



[Built-in Commands](#) (Nuxeo Installation and Administration - 5.5)



[Nuxeo Automation Commands](#) (Nuxeo Installation and Administration - 5.5)



[File System Commands](#) (Nuxeo Installation and Administration - 5.5)



[Nuxeo Server Commands](#) (Nuxeo Installation and Administration - 5.5)



• Shell Documentation

As we've discussed in the Overview section, 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 generate basic help for you.



The generated documentation is available in both text format (for terminal) and wiki format. See [Shell Command Index](#) for the wiki format.

Shell Features (Nuxeo Enterprise Platform (EP) - 5.5)

Extending The Shell (Nuxeo Enterprise Platform (EP) - 5.5)

Shell Documentation (Nuxeo Enterprise Platform (EP) - 5.5)

Shell Namespaces (Nuxeo Enterprise Platform (EP) - 5.5)

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 has 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 [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 [Shell Command Index](#) corresponding to the updated namespace.

Related pages

-  [Built-in Commands](#) (Nuxeo Installation and Administration - 5.5)
-  [Nuxeo Automation Commands](#) (Nuxeo Installation and Administration - 5.5)
-  [File System Commands](#) (Nuxeo Installation and Administration - 5.5)
-  [Nuxeo Server Commands](#) (Nuxeo Installation and Administration - 5.5)
-  [Shell Batch Mode](#) (Nuxeo Installation and Administration - 5.5)
-  [Shell Commands](#) (Nuxeo Enterprise Platform (EP) - 5.5)
-  [Shell Features](#) (Nuxeo Enterprise Platform (EP) - 5.5)
-  [Extending The Shell](#) (Nuxeo Enterprise Platform (EP) - 5.5)
-  [Shell Documentation](#) (Nuxeo Enterprise Platform (EP) - 5.5)
-  [Shell Namespaces](#) (Nuxeo Enterprise Platform (EP) - 5.5)
-  [Configuration Commands](#) (Nuxeo Installation and Administration - 5.5)

• 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
- Declaring a JAX-RS Application in Nuxeo
 - Example
 - Automatic discovery of JAX-RS resources at runtime
- Declaring a WebEngine Application in Nuxeo
 - Upgrading your old-style WebEngine module to work on 5.4.2
 - Example
- What is WebEngine good for?
- JAX-RS Resource Templating
 - Example
 - WebEngine Template Variables
 - Custom Template Variables
- WebEngine Modules
 - WebEngine Objects
 - WebEngine Adapters
 - @Path and HTTP method annotations.
 - Dispatching requests to WebObject sub-resources.
 - Module deployment
 - Module structure
 - Web Object Views
 - Extending Web Objects
 - Extending Modules
 - Template Model
 - Static resources
 - Headless modules
 - Groovy Modules
- Building WebEngine projects

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/mysite>

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 5.4.1 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 declare 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 manifests 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`. This 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 5.4.1 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 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 that 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 exist 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 follow 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 extends 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 exist.
- The `/skin/resources` directory 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 necessary 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 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 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 put 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:

```

```

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.

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 start when requesting a static resource). The static resources locations is `*/skin/*` - this will match any path that target a resource inside a **skin** directory in a JAX-RS application.

If you want to globally disable the automatic transaction management in WebEngine you can define the following runtime property:

```
org.nuxeo.webengine.tx.auto=false
```

in a `.properties` file inside the Nuxeo **config** directory.

Session Management

Also, WebEngine is providing a managed Core Session to any JAX-RS resource that want to connect to the repository. WebEngine will close the managed Core Sessions when no more needed so you should not worry about leaking Core Sessions. These sessions can be used either in JAX-RS resource methods, either in JAX-RS MessageBodyReaders or MessageBodyWriters.

By default the managed session is closed after the request ends (so it is stateless - a new request will use a new Core Session). You also have the option to change this behavior and reuse the same session in all requests belonging to the same HttpSession. In that case the session will be automatically closed when the HttpSession is invalidated. We call this **stateful** Core Session management.

If you want to globally turn on the stateful session management you can define the following runtime property:

```
org.nuxeo.webengine.session.stateful=true
```

in a `.properties` file inside the Nuxeo **config** directory.

To get the managed session 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 either get the current principal either get a Core Session:

```
UserSession us = WebEngine.getActiveContext().getUserSession();
Principal principal = us.getPrincipal();
CoreSession session1 = us.getCoreSession();
CoreSession session2 = us.getCoreSession("myrepo");
```

When calling the **getCoreSession()** method and no managed session was still created for the target repository then a new Core Session is created and returned. If the Core Session was already created (by a previous `getCoreSession()` call) then the existing session is returned.

You can see that, there are two flavors of `getCoreSession()` method:

- `CoreSession getCoreSession()`
- `CoreSession getCoreSession(String repositoryName)`

The first one is returning a session for the default repository. The second one will return a session for the named 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 the **getCoreSession()** 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 client side which will be the repository used to create a managed Core Session. To do this you can either use an HTTP request header:

```
X-NXRepository: myrepo
```

either a request parameter:

```
nxrepository=myrepo
```

If not specified the default repository defined by the Nuxeo server will be used.

So that, by specifying in the HTTP request a repository name, when calling `UserSession.getCoreSession()` - the return session will target the repository you specified.

Doing cleanup 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 `MessageBodyWriters`. 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() {
    cleanup(HttpServletRequest httpRequest) {
        ....
    }
});
```

The **cleanup** method will be invoked after the request was processed and the response created.

Configuring session and transaction management on a path basis

You can also configure how session and 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="" point="">
  <path value="/mymodule1" autoTx="false" stateful="true" />
  <path value="/mymodule2/resources" autoTx="false" />
  <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 and stateful session management is on.

The second one says that for any resource which path begins with `/mymodule2/resources` then automatic transaction is off and the default session management will be used.

The third rule says that for any `.gif` file inside `/mymodule3` automatic transaction is off and default session management will be used.

By default the **value** attribute represent a path prefix. If you want to use a regular expression you should 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 lexicographically order.

Regular expression rules will be always put after the prefix based rules (i.e. prefix based rules are *stronger*).

So that to find the best matching 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 following:

```
/a/b/c
/b/c
/a
/b
/a/b/d/.*\.gif
```

Thus, for the path: `/a/b` the first match will be `/a` - and will be used to define the session and 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:
 - *
 - * <code>GET</code> to get a view on the DocumentObject
 - * <code>POST</code> to create a DocumentObject
 - * <code>PUT</code> to update a document object
 - * <code>DELETE</code> to delete a DocumentObject.
 - *
- * 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 functionalities 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 functionality 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 functionality 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)
 - *
 - * 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].
 - *

 - * 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 `root` and a `target` 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>`.
- * `
`
- * `Note` 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 a special 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</code>`. 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</code>` directory. Views may live in two places:
- * ``
- * `` 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.
- * `` 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.
- * ``
- * 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>Derived</code>` too.
- * You may override these views by redefining them on type `<code>Derived</code>`
- * `
`
- * 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 explicitely 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);
    }
}

```

```

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 explicitly 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 explicitly
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 uncaught exception is thrown during the request. From that method you should
return a suitable response to render the error.
* To ensure exceptions are correctly 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 resoval 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 resoval 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 <code>info</code> 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 il8n messages - but in this example
we are displaying them directly without using il8n 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>

```

Nuxeo Android Connector

What is Nuxeo Android Connector

Nuxeo Android Connector is a SDK to build Android Applications that communicate with a Nuxeo Server.

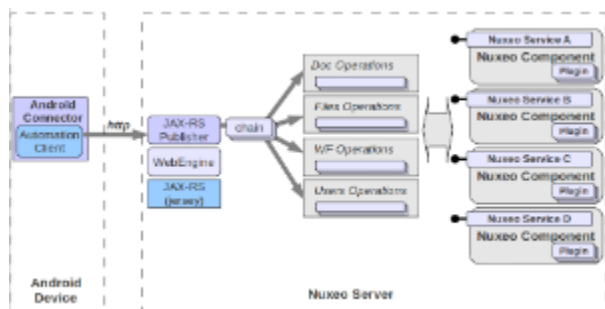
Nuxeo Content Automation

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();
}
```

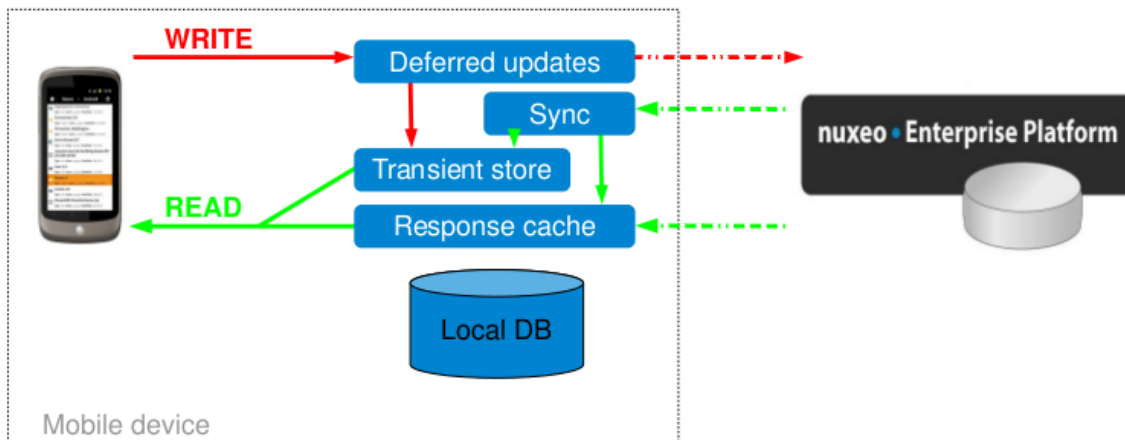
Outside of this direct association between the Context and the Automation Client Session, the [standard documentation for Nuxeo Automation Client](#) does apply to the Automation Client embedded in Android Connector.

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 you 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.



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 meta-data 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 a 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.STORE);

// Force refresh (try to fetch from server, store to cache)
session.newRequest("Document.GetChildren").setInput(docRef).execute(CacheBehavior.FORCE_REFRESH);
```

TransientState

This cache stores Document deltas (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 synchrozed with the server
- "conflict" means that the push on the server resulted in a conflict

TransientStateManager exposes an API to automatically fetch and merge deltas 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 exeuteNow);
```

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 its 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 populated from :

- children of a Folder
- content of a InBox (CMF use case)
- the Clipboard 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 an `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(doc2Update);
```

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 personal 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 inherit from `AbstractLazyUpdateableDocumentsList` and simple implement the 2 methods `buildUpdateOperation` 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 expose its 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 docuent list like any simple list.
The documents list will be fetched and refreshed automatically 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
<code>content://nuxeo/documents</code>	returns all documents of the repository via an Android Cursor
<code>content://nuxeo/documents/<UUID></code>	access to document with given UUID
<code>content://nuxeo/<providername></code>	Android cursor documents in the given provider
<code>content://nuxeo/<providername>/UUID</code>	access to document with UUID in the given provider
<code>content://nuxeo/icons/<subPath></code>	download and cache icon of the given sub path
<code>content://nuxeo/blobs/<UUID></code>	download and cache the main blob of the doc with the given UUID
<code>content://nuxeo/blobs/<UUID>/<idx></code>	download and cache the blob <idx> of the doc with the given UUID
<code>content://nuxeo/blobs/<UUID>/<subPath></code>	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 want 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 implement 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.

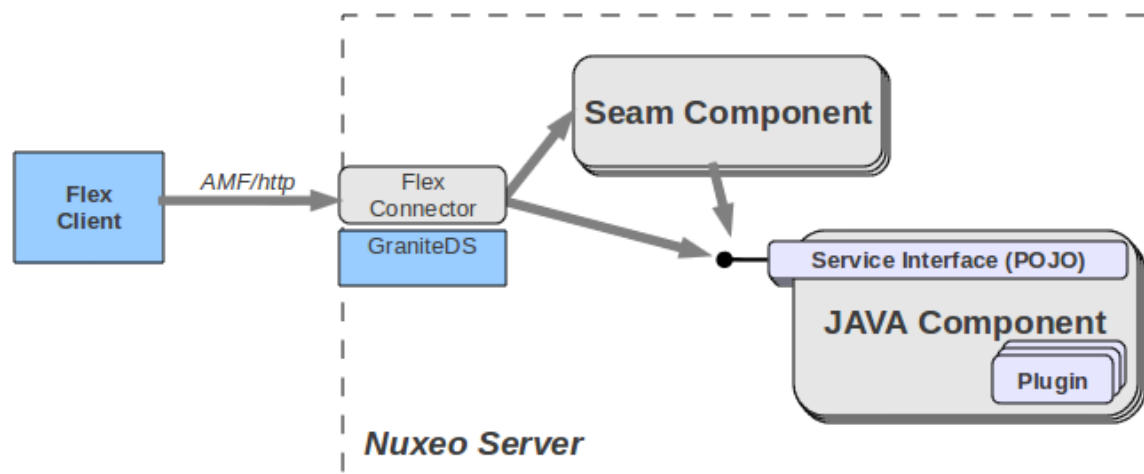
Class Name	Type	Description
------------	------	-------------

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

Nuxeo Flex Connector

What is Nuxeo Flex Connector

The Flex Connector is an AMF (Action Message Format) bridge that allows Flash clients to use services exposed by the Nuxeo platform.



This means you can use the Flex Connector to have your Flex application:

- Read, search, create, update documents stored in Nuxeo,
- Manage workflows and tasks,
- Use conversion services,
- Manage users and send notifications,
- ...

How does it work

The Nuxeo Flex Connector is based on [Granite Data Services](#) which is a free, open source (LGPL) alternative to Adobe® LiveCycle® (Flex™ 2+) Data Services. GraniteDS provides a full support for AMF3/Remote Object for EJB3/Seam/Spring/Guice/Pojo technologies.

Nuxeo uses Granite to expose Seam Beans and Nuxeo Runtime services as remotable services to Flex clients.

Support for Nuxeo Automation services will be added in the next release (2.0) of the connector.

For security reasons, all Nuxeo services and Seam beans are not exposed by default and you can use [Nuxeo extension point system](#) to define what services you want to expose to the Flex clients.

In addition, since ActionScript and Java are different languages, the connector does some type mapping and you have the possibility to configure your own specific mapping. See [AMF Mapping in Nuxeo](#) for more details.

Getting the Flex Connector

You can use [Nuxeo Marketplace](#) ([Connector](#) and [Samples](#)) or [Nuxeo Update Center](#) to install the Nuxeo Flex Connector and associated samples.

You can also get the JARs from our [Maven repository](#) .

Since the Nuxeo Flex Connector is LGPL and fully open source as the rest of the Nuxeo Platform, you can also check out the sources and built it. See [Build and deploy Nuxeo Flex Connect](#) section.

Using the Flex Connector

The Nuxeo Flex Connector comes with a set of samples to show you how to use the Nuxeo services from a Flex client. We also provide a set of reusable Flex components and a build system integrated with Maven.

Read more about [Using Flex Connector](#)

AMF Mapping in Nuxeo

Services and AMF

GraniteDS allows to expose different types of services, and you can use the GraniteDS configuration files to do so. However, the Nuxeo Flex Connector contains a dedicated Component that is used to configure GraniteDS via the extension point system so that you don't have to update the GraniteDS configuration file.

You can use [Nuxeo Platform Explorer](#) to get more details about:

- the [NxGraniteConfigService component](#),
- the [services Extension Point](#).

Using Nuxeo Runtime services

All the features of the Nuxeo platform are exposed as Services. You can access these services via the Flex Connector. For that you need to declare what services need to be available via AMF remoting.

Here is a simple example for exposing the SchemaManager service as a remotable:

```
<component name="org.nuxeo.ecm.platform.ui.granite.config.facetExplorer">
  <extension
    target="org.nuxeo.ecm.platform.ui.granite.config.NxGraniteConfigService"
    point="services">
    <runtime id="schemaManager" class="org.nuxeo.ecm.core.schema.SchemaManager" />
  </extension>
</component>
```

- `class` defines the service interface to be exposed,
- `id` defines the name that will be used on the client side.

On the client side, you will use the `services-config.xml` to declare the remote service:

```
<services>
  <service
    id="schemaManager"
    class="flex.messaging.services.RemotingService"
    messageTypes="flex.messaging.messages.RemotingMessage">
    <destination id="schemaManager">
      <channels>
        <channel ref="nx-amf" />
      </channels>
      <properties>
        <factory>nxruntimeFactory</factory>
        <class>org.nuxeo.ecm.core.schema.SchemaManager</class>
      </properties>
    </destination>
  </service>
</services>
```

Using Seam beans

Although you can access to Nuxeo Runtime services, it make sense to use Seam beans:

- because you may want to manage some states on the server side: Seam Context management is easy;
- because you want Nuxeo to expose custom services that contains logic specific for your UI (having Seam beans as Controllers for your Flex UI);
- because you want to reuse existing Nuxeo Seam Beans.

In order to make a Seam Bean accessible remotely from the Flex side you need to:

1. declare the bean in the services extension point of `NxGraniteConfigService`,
2. annotate the methods you want to call with Seam `@WebRemote` annotation.

Here is a simple example for exposing the 2 Seam beans as a remotable:

```
<extension point="services"
target="org.nuxeo.ecm.platform.ui.granite.config.NxGraniteConfigService">
    <seam id="flexDocumentManager" source="flexRepositoryService" />
    <seam id="flexActionService" />
</extension>
```

- `id` is the name of the service on the client side;
- `source` is the name of the Seam Component (defaults to `id` if unset).

On the client side, you will use the `services-config.xml` to declare the remote service:

```
<service
    id="flexDocumentManager"
    class="flex.messaging.services.RemotingService"
    messageTypes="flex.messaging.messages.RemotingMessage">
    <destination id="flexDocumentManager">
    <channels>
        <channel ref="nx-amf" />
    </channels>
    <properties>
        <factory>seamFactory</factory>
        <source>flexRepositoryService</source>
    </properties>
    </destination>
</service>

<service
    id="flexActionService"
    class="flex.messaging.services.RemotingService"
    messageTypes="flex.messaging.messages.RemotingMessage">
    <destination id="flexActionService">
    <channels>
        <channel ref="nx-amf" />
    </channels>
    <properties>
        <factory>seamFactory</factory>
        <source>flexActionService</source>
    </properties>
    </destination>
</service>
</services>
```

Type Mapping and Marshaling

DocumentModel vs FlexDocumentModel

GraniteDS provides a simple and transparent mapping between Java Object and ActionScript objects. But this basic mapping only transfers the data and uses a simple one to one property mapping.

In some cases, this is not enough because you need a better control on the Marshaling.

For accessing Nuxeo services (Seam Bean or Runtime Service), we have this requirement for transmitting Documents (`DocumentModel`) over the wire between Flex and Java sides:

- because a `DocumentModel` is a big object;
- because you want to be able to update the Document on the client side: so you need methods;
- because you want to be able to detect on the server side what part of the Document have been changed;
- because you don't want Blobs to be sent as part of the Documents.

For this we have used GraniteDS `Externalizer` and `InvocationListener` to provide a mapping between the Java `DocumentModel` and an ActionScript `FlexDocumentModel`.

We also added support so that `DocumentRef` are serialized as Action Scripts strings.

```
IdRef ( "0f9d03ed-ba00-4c54-b5b2-2f6b4be4a239" ) =>
"IdRef:0f9d03ed-ba00-4c54-b5b2-2f6b4be4a239"
PathRef ( "/default-domain/workspaces/ws1" )      =>
"pathRef:/default-domain/workspaces/ws1"
```

Thanks to this mapping you can very easily call the existing Nuxeo services on most of the methods.

So for example:

Java signatures

```
DocumentModel methodA(DocumentRef docRef);

DocumentRef methodB(DocumentModel doc);
```

ActionScript signatures

```
function methodA(docRef:String):FlexDocumentModel;

function methodB(doc:FlexDocumentModel):String;
```

Advanced Marshaling

If for some custom objects you need more than what is provided by default in the Nuxeo Flex Connector, you can leverage GraniteDS to define some custom Marshaling.

- **Externalizer**
If you want to use Nuxeo API, you will need a mapping between Java Object and Action Script Object. GraniteDS provides a pluggable externalizer for your different Object. It aims to (de)serialize the different fields of your objects. For more information, [see GraniteDS documentation](#).
- **InvocationListener**
You might need more control on mapping. For instance, the `DocumentModel` object in Nuxeo is rather complicated. So we have a `FlexDocumentModel` object which is a simplified version of `DocumentModel`. The mapping between those two Java objects is done in the `NuxeoInvocationListener`. It listens to each service invocation method call. Then we can switch from `FlexDocumentModel` to `DocumentModel` or the other way around. For more information on `InvocationListener`, [see GraniteDS documentation](#).

Build and deploy Nuxeo Flex Connect

Getting the source code

The sources are hosted in our Mercurial repository.

To get a copy:

```
hg clone http://hg.nuxeo.org/addons/nuxeo-platform-flex
cd nuxeo-platform-flex
hg update 1.1-5.4.2
```

NB: This example assumes you want to use version 1.1 of the Flex Connector for Nuxeo Platform 5.4.2

This source tree contains:

- sources of the Flex Connector,
- sources of the SWC reusable components,
- sources of the Samples (client side and server side),
- distribution of Nuxeo CAP + Flex Connector + Samples.


```

|-- nuxeo-flex-components          (SWC for FlexDocumentModel and Login)
|-- nuxeo-flex-components-extra    (SWC for additional UI components)
|-- nuxeo-flex-connector           (Nuxeo GraniteDS plugin : the real server side
connector)
|-- nuxeo-flex-distribution        (build runnable server : Nuxeo CAP + Connector +
Samples)
`-- samples                       (samples)
    |-- actions
    |   |-- client
    |   `-- server
    |-- browser
    |   |-- client
    |   `-- server
    |-- documentAPI
    |   |-- client
    |   `-- server
    |-- document-browser
    |   |-- client
    |   `-- server
    |-- facet-explorer
    |   |-- client
    |   `-- server
    |-- flex-login
    |   |-- client
    |   `-- server
    |-- samples-container-tab      (server side plugin that is used to have the samples
displayed in Nuxeo)
    |-- simple
    |   |-- client
    |   `-- server
    |-- state-management
    |   |-- client-edit
    |   |-- client-save
    |   |-- client-select
    |   `-- server
    |-- tree
    |   |-- client
    |   `-- server
    |-- users
    |   |-- client
    |   `-- server
    `-- vocabularies
        |-- client
        `-- server

```

Building and deploying

As all other builds in Nuxeo, this build is driven by Maven.

Building and deploying the Flex Connector

```
mvn clean install -Pconnector
```

The result of the build is a nuxeo-flex-connector JAR that should be copied in the `bundles` or `plugins` directory of your Nuxeo server.

Building and deploying the SWC and the Samples

```
mvn clean install -Psamples
```

The result of the build is a set of JARs that should be copied in the `bundles` or `plugins` directory of your Nuxeo server.

Building the distribution

```
mvn clean install -Pall
```

This command runs the steps below:

1. download and unzip a Nuxeo CAP server,
2. download the needed GraniteDS libraries,
3. build the JAR, SWC and SWF files,
4. put these files and the GraniteDS libraries in the downloaded Nuxeo CAP server.

✓ The README.txt contains more informations about the build.

The result is a prebuilt Nuxeo CAP server containing the Flex Connector and the samples. To run it:

- for Posix users:

```
cd target/stage/nuxeo-flex-tomcat
chmod +x bin/nuxeoctl
bin/nuxeoctl console
```

- for MS users:

```
cd target\stage\nuxeo-flex-tomcat
bin\nuxeoctl.bat
```

Using Flex Connector

Nuxeo Flex sample code

Content

The Flex Connector comes with a set of sample codes that can be used to understand:

- how to use Nuxeo services,
- how to use Documents provided by Nuxeo.

For that we provide 11 samples:

Basic remoting samples	Shows how to call remote services and how Java Objects and DocumentModels Marshaling is handled.
Document API browser sample	Shows how to use the repository API to browse the content and display documents properties.
DocumentAPI sample	Shows how to modify a document.

Actions sample	Shows how to use Nuxeo Action service to fetch from server what actions are available in a given context (Document + user).
Nuxeo Runtime sample	Shows how to call directly a Nuxeo Runtime Service.
Flex Login sample	Shows how you can provide a Flex UI for Login/Logout in Nuxeo.
Vocabularies sample	Shows how to use Nuxeo's vocabularies from Flex.
User management sample	Searches and modifies a user.
Tree sample	Simple demo of the provided Navigation Tree SWC component.
Document Explorer	Sample application to show how to build an Explorer using Nuxeo service and Nuxeo reusable SWC components.
Document API and State Management	This sample shows how you can manage state on the server side to share some data between several SWF applications.

When deploying the samples on a Nuxeo instance you can access the samples by 2 ways:

- use the "Flex Samples" tab from within the main UI,
- use the direct link <http://server/nuxeo/flexsamples/>.



If you don't already have a valid authentication session, using the second option will allow you to login using the Flex login screen.

As you will see, these samples are really simple examples, not a showcase. There are three main reasons for that:

- we are not Flex developers;
- we use bare ActionScript 3 to be sure this will work in any environments. This means:
 - no additional framework or components,
 - no specific IDE : text editor + command line build;
- we wanted to keep the code as simple as possible.

Getting the samples

If you installed the Samples via the Marketplace to do your tests, and you want to go further and look at the code, you probably want to [check out the sources](#).

Once done you can generate the Eclipse project by running:

```
mvn eclipse:eclipse
```

Alternatively, you can also get the source JARs from our [Maven repository](#).

Building your Flex client

Nuxeo SWC Libs

The Nuxeo Flex Connector comes with two SWC components:

- nuxeo-platform-ui-flex-client-components.swc that contains the basic ActionScript "tools":
 - FlexDocumentModel ActionScript Object,
 - Login helpers,
 - DocumentTree simple UI component.
- nuxeo-platform-ui-flex-client-components-extra.swc that contains UI components used by the Document Explorer sample.

If you want to call services that input or output DocumentModels, you will need to use the FlexDocumentModel and then use

nuxeo-platform-ui-flex-client-components.swc as a dependency.

If you use Maven, you will get the SWC files automatically (see below), otherwise, you can download them from our [maven repository](#).

Building with Maven

Building with Maven is clearly not the only possible solution, but since we are Java developers, this is the solution we choose 😊

You can use the samples as examples, but basically, in your POM file you should:

- add the dependency on Nuxeo SWC component:

```
<dependencies>
  <dependency>
    <groupId>org.nuxeo.ecm.platform</groupId>
    <artifactId>nuxeo-platform-ui-flex-client-components</artifactId>
    <type>swc</type>
  </dependency>
</dependencies>
```

- declare the FlexMojo plugin to build the SWF:

```
<build>
  <sourceDirectory>src/main/flex/</sourceDirectory>
  <plugins>
    <plugin>
      <groupId>org.sonatype.flexmojos</groupId>
      <artifactId>flexmojos-maven-plugin</artifactId>
      <version>3.0.0</version>
      <extensions>true</extensions>
    </plugin>
  </plugins>
</build>
```

Using Flex Builder

From Flex Builder you can add Nuxeo SWC:

1. Click "Project" "Properties".
2. Choose "Flex Builder Path" in left.
3. Choose Library in right.
4. Click "Add SWC".
5. Choose nuxeo-platform-ui-flex-client-components-1.1-5.4.2.swc.

Once done, you can use FlexDocumentModel, Login and DocumentTree in your project.

Data Services Configuration

GraniteDS requires services configuration on both client and server sides. To avoid painful duplication of code, the connector provides two default factories and a default channel.

Default Nuxeo factories

There are two different factories: one to get Seam component and one to get Nuxeo services. These are the two factories you should have to use in your service-config.xml file.

```
<factories>
  <factory id="seamFactory"
class="org.nuxeo.ecm.platform.ui.granite.factory.NuxeoSeamServiceFactory" / >
  <factory id="nxRuntimeFactory"
class="org.nuxeo.ecm.platform.ui.granite.factory.NuxeoRuntimeServiceFactory" / >
</factories >
```

Default channel

This is the default channel you have to use in your `service-config.xml` file.

```
<channels >
  <channel-definition id="nx-amf" class="mx.messaging.channels.AMFChannel" >
    <endpoint
      uri="http://{server.name}:{server.port}/nuxeo/nuxeo-amf/amf"
      class="flex.messaging.endpoints.AMFEndpoint"/ >
    </channel-definition >
    <!--server.port and server.name are resolved dynamically at run time-- >
  </channels >
```

Service declaration

See [AMF Mapping in Nuxeo](#)

Provided Seam components

The FlexConnector comes with a set of Seam components that are dedicated to Flex remoting usage. The samples will show you how to use them.

You can list the available Seam Bean by using [Nuxeo Platform Explorer](#) .

The main Seam components are:

Seam component name	Usage	Remotable
flexActionService	High level interface on top of Action Service	Yes
flexDocumentManager	Manages an even scoped CoreSession	No, Server side only
flexNavigationContext	Manages server side state	No, Server side only
flexRepositoryService	High level interface on the CoreSession	Yes (called flexDocumentManager on the client side)
flexUserService	High level interface on the UserManager	Yes
flexVocabularyService	High level interface for the Vocabularies	Yes

Roadmap of the Flex Connector

You can keep an eye on the development by using [NXFLEX Jira](#).

The next major release will be 2.0 and will include:

- alignment on GraniteDS 2.2.x,
- support for Automation Services call,
- support for push.

Extending Nuxeo

Table of Contents:

- [The Nuxeo Plugin model](#)
- [Adding a new UI block in Nuxeo DM](#)
- [Making a connector](#)
- [Integrating with JPA](#)

The Nuxeo Plugin model

Adding a new UI block in Nuxeo DM

Making a connector

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>
```

Updating your application using the Admin Center (web interface)

The Nuxeo Admin Center is a space within the Nuxeo Platform that provides administrative services, such as server and application usage summary information, as well as access to upgrades, patches, the Nuxeo Marketplace, and Nuxeo Studio projects.

The Admin Center offers access to different kinds of information about your Nuxeo instance. Depending the modules and additional packages you have installed on the Platform, you may have more or less information.

The default tabs available in the Admin Center are listed below.

- **System information:** this section of the Admin Center provides information about the server the Nuxeo Platform is installed on, about your instance configuration.

System information

Host: Nuxeo distribution Setup Repository statistics Repository binaries

Nuxeo application name	Nuxeo Platform														
Nuxeo application version	5.5														
Distribution name	cap														
Distribution version	5.5														
Target application server	tomcat														
Distribution date	201112131441														
Warnings on startup	None														
Deployed bundles	<table border="1"> <tr><td>org.nuxeo.admin.center</td><td>5.5.0-HF04</td></tr> <tr><td>org.nuxeo.admin.center.monitoring</td><td>5.5</td></tr> <tr><td>org.nuxeo.connect.client.wrapper</td><td>5.5</td></tr> <tr><td>org.nuxeo.connect.update</td><td>5.5.0-HF03</td></tr> <tr><td>org.nuxeo.ecm.actions</td><td>5.5.0-HF03</td></tr> <tr><td>org.nuxeo.ecm.audit.io</td><td>5.5</td></tr> <tr><td>org.nuxeo.ecm.automation.core</td><td>5.5.0-HF06</td></tr> </table>	org.nuxeo.admin.center	5.5.0-HF04	org.nuxeo.admin.center.monitoring	5.5	org.nuxeo.connect.client.wrapper	5.5	org.nuxeo.connect.update	5.5.0-HF03	org.nuxeo.ecm.actions	5.5.0-HF03	org.nuxeo.ecm.audit.io	5.5	org.nuxeo.ecm.automation.core	5.5.0-HF06
org.nuxeo.admin.center	5.5.0-HF04														
org.nuxeo.admin.center.monitoring	5.5														
org.nuxeo.connect.client.wrapper	5.5														
org.nuxeo.connect.update	5.5.0-HF03														
org.nuxeo.ecm.actions	5.5.0-HF03														
org.nuxeo.ecm.audit.io	5.5														
org.nuxeo.ecm.automation.core	5.5.0-HF06														

- **Activity:** enables administrators to have information and statistics on the application.

Activity

Users sessions Events

Total number of active HTTP Sessions 1
Total number of HTTP requests 7
List active sessions within last : 30 minutes

Login	Inactivity time	Accessed pages	Session duration	Last accessed url	Last activity
Administrator	0s	7	22m 5s	/nuxeo/view_admin.faces	Apr 12, 2012 3:23:11 PM

- **Nuxeo Connect:** you can connect your Nuxeo instance with your [Nuxeo Connect](#) account. This enables you to manage your Support tickets from your Nuxeo application and to connect your Nuxeo instance to the Nuxeo Marketplace.

Nuxeo Connect

Nuxeo Connect registration status Nuxeo Connect tickets

Your Nuxeo instance is registered

Instance registration summary

Contract Status: Connect registration OK
Contract end date: 01/07/2020
Description:
Instance type: Development
Last refresh date: 4/12/2012 3:28 PM Refresh
Logical instance identifier (CLID): e7aa37f0-018e-47ae-0364-df58b02fb229--6b524e6c-hdyr-9364-87a3-4eaa7c9b6010
Technical instance identifier (CTID): Mac OS X-h2u2adW0p2AA8np2gOURG===+pvlDhBE7zRkAdsgYBrEig==
Change your connect registration Unregister

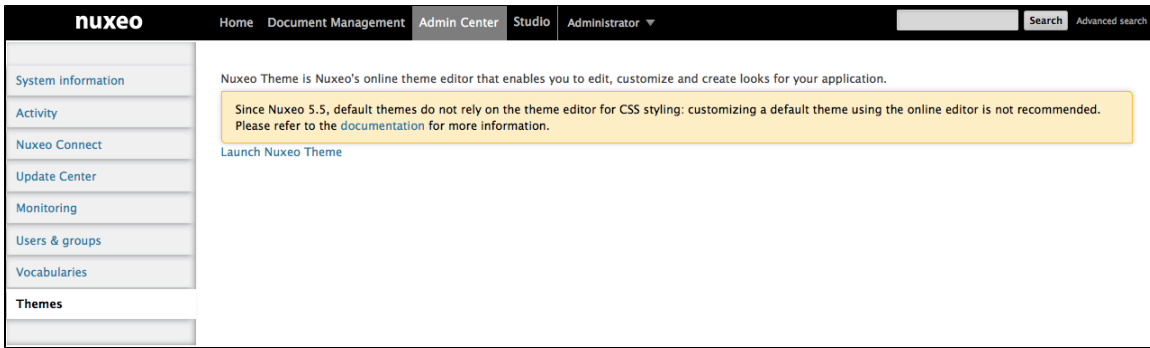
- **Update Center:** this section of the Admin Center provides all the updates you can need (updates and patches, Nuxeo Marketplace packages, direct access to your Nuxeo Studio packages, local packages)

- **Monitoring:** enables administrators to monitor some technical information and display messages to users.

- **Users & Groups:** enables administrators to create, edit, delete users and groups of users.

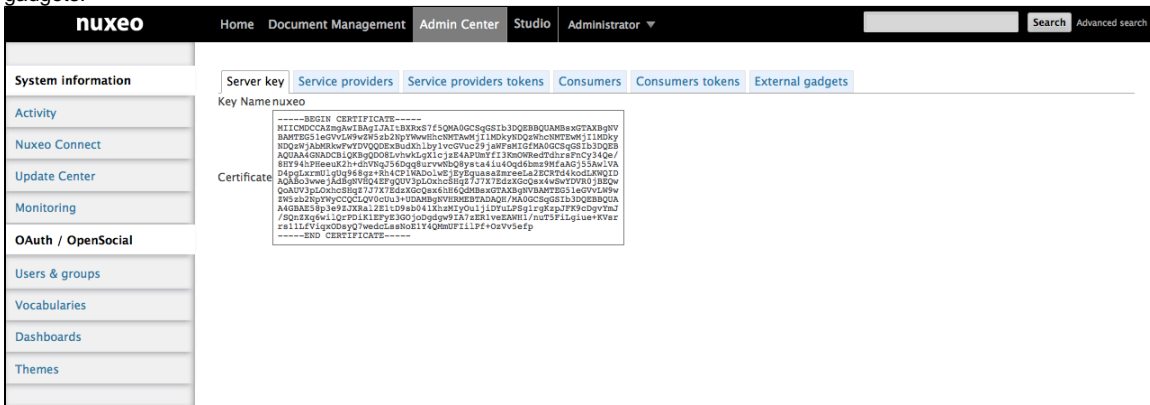
- **Vocabularies:** enables administrators to customize and adapt the values displayed in lists in the application.

- **Themes:** gives administrators access to Nuxeo Themes, that enables them to create new themes for the application.

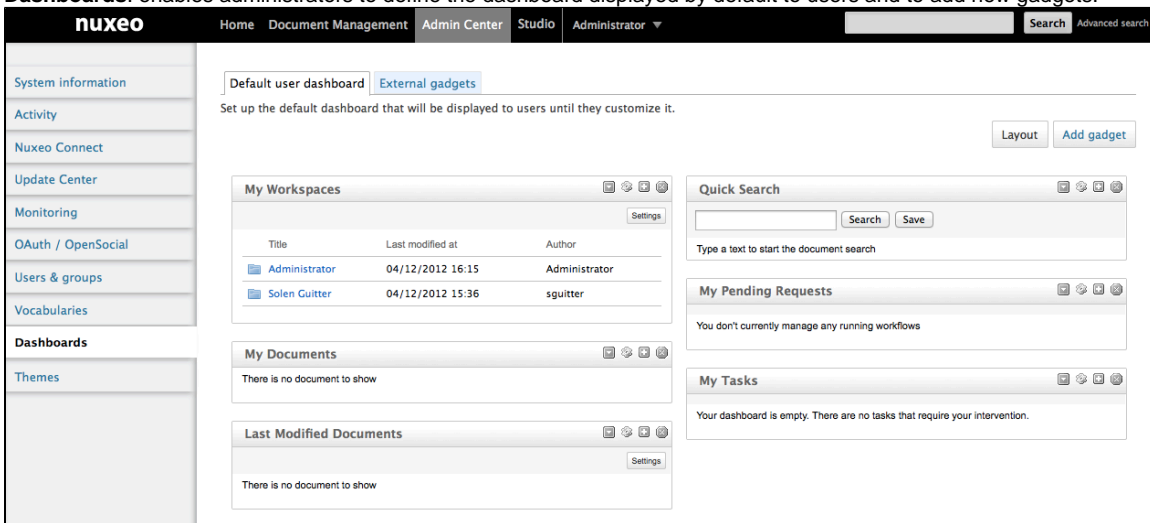


The Document Management module adds the tabs below:

- **OAuth/OpenSocial**: enables administrators to manage the authentication with other applications using OAuth protocol and to add new gadgets.

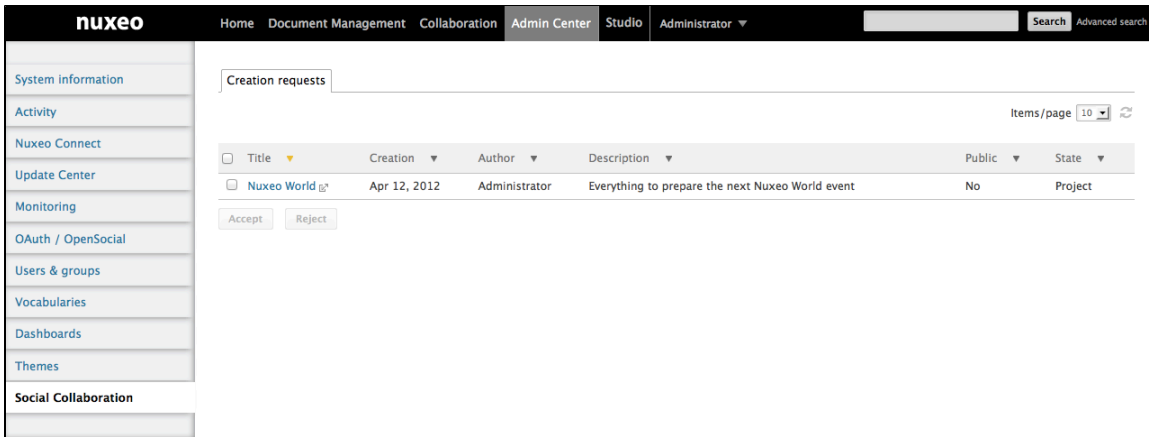



- **Dashboards**: enables administrators to define the dashboard displayed by default to users and to add new gadgets.



The Social Collaboration module adds the tabs below:

- **Social Collaboration**: enables administrators to manage the creation of new social workspaces.



 Installing Marketplace packages can add new tabs in the Admin Center.

Register your Nuxeo instance

Registering your Nuxeo application with Nuxeo Connect will give you access to a wide range of services, such as:

- the Update Center, so that you can easily install patches and bug fixes,
- a global view of your open support tickets,
- [Nuxeo Marketplace](#), the app store for the Nuxeo community, that provides an easy and powerful way to add features and plugins to your Nuxeo application,
- an automatic update of your Nuxeo Studio configuration, including hot redeploy.

The registration process only copies a file on your file system. This enables the Nuxeo Connect portal to identify the instance among all the registered instances. You can register multiple instances.

On this page

- [How to register](#)
 - [Creating your Nuxeo Connect account](#)
 - [Registering online](#)
 - [Registering offline](#)
- [Re-registering your Nuxeo instance](#)

How to register

To be able to register, you need to have a [Nuxeo Connect account](#).

Registration can be done during the installation steps, using the [configuration wizard](#) or at anytime later, through the Admin Center. Registration doesn't require an Internet access. If your server cannot connect to the internet, follow the [offline registration steps](#). Otherwise, follow the [online registration steps](#).

For development instances on which you may need to remove your data, you may need to [re-register your instance](#). In that case, use the offline registration form.

Creating your Nuxeo Connect account

If you already have an account on Nuxeo Connect, either because you are a Nuxeo customer, or because you created a trial, you can continue to the next step. If not, follow those steps to get credentials to the Nuxeo Connect portal.

To subscribe to a Connect trial:

1. Go to the [Connect trial registration form](#).
2. Fill the form. Provide a valid email address or else registration will not be completed.
3. Confirm registration by clicking on the link sent to the email address in the previous step.
4. If it is not already done, download last published version of [Nuxeo DM](#).

Registering online

1. Start your Nuxeo instance and connect with Administrator/Administrator (or your administrator password if it is already customized).
2. Click on the **Nuxeo Admin Center** link, on the top of the user interface to go to the Administration Center.
3. Click on the **Nuxeo Connect** link, on the left side.
4. Authenticate to Nuxeo Connect portal by giving your credentials.
5. Choose which application is concerned among the ones your recorded in Nuxeo Connect.

6. Give a description of your registration, like "Jolene's instance" and indicate if it is a development, qualification or production instance (just for our information).
7. The registration process will end automatically, you can now browse the various tabs of the Update Center area, and set up plugins from the [Nuxeo Marketplace](#), see the [how to](#).

Registering offline

Offline registration can be used for first registration when the server doesn't connect to the internet. It is also used for registration.

To register your instance for the first time:

1. Start your Nuxeo instance and connect with Administrator/Administrator (or your administrator password if it is already customized).
2. Click on the **Nuxeo Admin Center** link, on the top of the user interface.
3. Click on the **Nuxeo Connect** link, on the left side.
4. On the right part of the registration screen, get the **instance technical identifier** called CTID (ex: *Mac OS X-EbMKUsirT9WQszM5mDkaKAp=-BhnJsMDaabDHAQ0A300d6Q==*) and store it in a file so that you can connect to internet (or just copy it if the machine from which you have access to the Admin Center has internet).

Offline instance registration

Use this if you already have a Nuxeo Connect Account but your Nuxeo instance is not connected to the Internet.

First, get the CLID number on the [Nuxeo Connect Web Site](#)

You will be asked to provide your instance technical identifier :

Mac OS X-dBoOyQOXNxaa7RS1wE818A==8eJrb/GjQoDK20aWVmFc+w==

Then, paste here the CLID number provided by Nuxeo Connect and register!

Description of your Nuxeo instance

CLID provided by Nuxeo Connect Web Site

[Register this instance](#)

5. Go to [Nuxeo Connect portal](#).
You will arrive on your application page, and will see all the information regarding the application that was declared when you created your account.
6. On the "Instances" area, click on the **Add a new instance** link.

Associated instances

This application is not associated with any Nuxeo instance(s).
[Click here](#) to add one.

Target environment

Operating System	Linux 32bits
Nuxeo Product	5.3.0
Repository Store	JCR Repository

7. Fill in the registration form and submit it.
The instance is registered. You are given an identifier (CLID) to validate registration from Nuxeo Admin Center.
8. Copy this identifier.
9. On the Admin Center, fill in the instance description, paste the CLID from Nuxeo Connect and click on the **Register this instance** button.

Offline instance registration

Use this if you already have a Nuxeo Connect Account but your Nuxeo instance is not connected to the Internet.

First, get the CLID number on the [Nuxeo Connect Web Site](#)

You will be asked to provide your instance technical identifier :

Mac OS X-dBoOyQOXNxaa7RS1wE818A==8eJrb/GjQoDK20aWVmFc+w==

Then, paste here the CLID number provided by Nuxeo Connect and register!


Description of your Nuxeo instance

CLID provided by Nuxeo Connect Web Site

[Register this instance](#)

The registration is approved and the registration summary is displayed.

Nuxeo Connect registration status
Nuxeo Connect Tickets

 Your Nuxeo instance is registered

Instance registration summary

Contract Status :	Connect subscription is expired.
Contract end date :	01/07/2010
Description :	my instance
Instance type	Development
Logical instance identifier (CLID)	5cc14d5a-a240-4f36-99ce-a359637eb983--e296b69c-110b-46f3-85e8-0708b25a8bcd
Technical instance identifier (CTID)	Mac OS X-dBoOyQOXNxo7RS1wE818A==8ejrb/CjQoDK20aWVmFc+w==

Re-registering your Nuxeo instance

If you have removed your data from your Nuxeo application, in case of a development instance, for example, you will need to register your instance again.

To re-register your instance:

1. Log in to Nuxeo Connect.
2. In the associated instances, click on your Nuxeo instances link.
The page listing the associated instances for your project opens.
3. Copy the Identifier of the instance you want to register (Identifier is of the form "fdedaf21-be00-412e-b0ab-f2394479d5f8--885dcf60-5a4a-46e8-a904-0123456789ab").
4. In Nuxeo Admin Center, paste this identifier in the CLID field, fill in the instance description and click on the **Register this instance** button.

Offline instance registration

Use this if you already have a Nuxeo Connect Account but your Nuxeo instance is not connected to the Internet.

First, get the CLID number on the [Nuxeo Connect Web Site](#)

You will be asked to provide your instance technical identifier :
Mac OS X-dBoOyQOXNxo7RS1wE818A==8eJrb/GjQoDK20aWVmFc+w==


Then, paste here the CLID number provided by Nuxeo Connect and register!

Description of your Nuxeo instance

CLID provided by Nuxeo Connect Web Site

Your instance is registered again and the registration summary is displayed.

Nuxeo Connect registration status
Nuxeo Connect Tickets

 Your Nuxeo instance is registered

Instance registration summary

Contract Status :	Connect subscription is expired.
Contract end date :	01/07/2010
Description :	my instance
Instance type	Development
Logical instance identifier (CLID)	5cc14d5a-a240-4f36-99ce-a359637eb983--e296b69c-110b-46f3-85e8-0708b25a8bcd
Technical instance identifier (CTID)	Mac OS X-dBoOyQOXNxo7RS1wE818A==8eJrb/GjQoDK20aWVmFc+w==

Install a new package on your instance

This will guide you in the steps to install a new Nuxeo package from the Marketplace.

Packages can be installed [from the Marketplace](#) or directly [from the Admin Center](#).



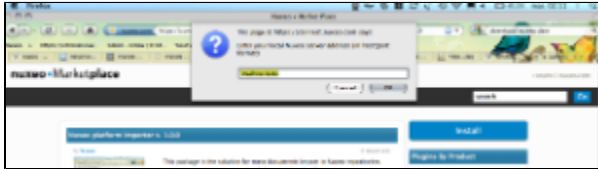
You need to have [registered your Nuxeo instance](#) to be able to install packages.

To install a new package from Nuxeo Marketplace:

- Go on the package you want to install in the Marketplace and click on the **Install** button on the right.



- Choose the URL of the Nuxeo Server on which you want to set up the package (and on which you have Administrator credentials). For instance, `localhost:8080`.



Your Nuxeo application opens in a new window.

- Log in as an administrator, if you are not already. You are directed in the Admin Center.
- Confirm that you want to download the package.



- Confirm downloading a second time, from the Admin Center of the application where you want to install the package.

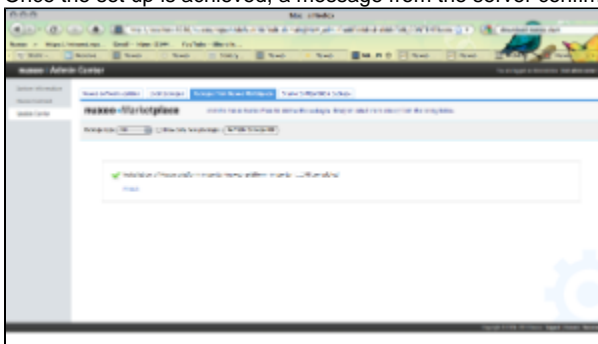


Once the package has been downloaded, it has a green **Install** link.

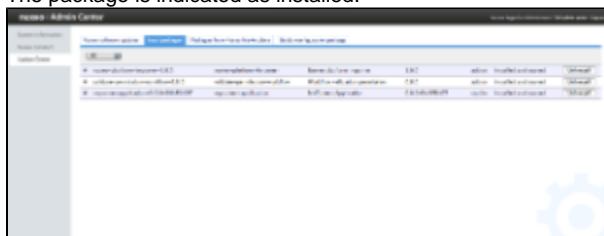
- Install the package by clicking on the **Install** link. Click on **Start** to confirm installation.



Once the set up is achieved, a message from the server confirms that the installation was performed correctly.



7. Click on the **local packages** tab.
The package is indicated as installed.



You may need to restart your server for the package to be fully functional. If your package is indicated as supporting hotreload, you don't need to reboot your server.

✓ You can uninstall the package from here if needed.

To install a new package from Nuxeo Admin Center:

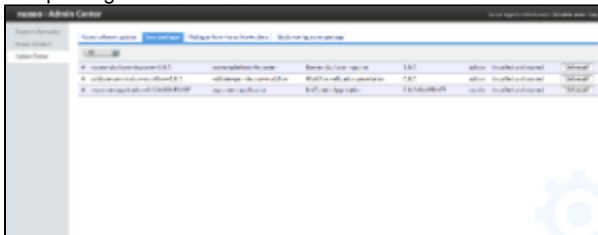
1. Go on the Admin Center in your application.
2. In the **Update Center**, go on the **Packages from Nuxeo Marketplace**.
3. Click on the **Download** link of the package you want to download.
A download in progress page is displayed while the package is being downloaded. When download is finished, the list of Marketplace packages is displayed again and the downloaded package has a green **Install** link.
4. Install the package by clicking on the **Install** link. Click on **Start** to confirm installation.



Once the set up is achieved, a message from the server confirms that the installation was performed correctly.



5. Click on the **local packages** tab.
The package is indicated as installed.



You may need to restart your server for the package to be fully functional. If your package is indicated as supporting hot reload, you don't need to reboot your server.



You can uninstall the package from here if needed.

Semantic Entities Installation and Configuration

The Semantic Entities addon available on the marketplace will require some specific setup to work properly.

<https://connect.nuxeo.com/nuxeo/site/marketplace/package/semantic-entities-1.0.0>

Requirements

- The Semantic Entities addon requires an internet access from the Nuxeo DM server to the <http://dbpedia.org> and the <https://stanbol.dem.o.nuxeo.com> web-services. The Stanbol access is optional: you are advised to setup your own Apache Stanbol instance. The access to <http://dbpedia.org> is mandatory but future versions of Apache Stanbol will make this requirement optional too.

- The addon further require a database server that handles concurrency at write time (e.g. using MVCC) to avoid blocking the rest of the application while syncing and linking new entities definitions. Hence the default H2 database of Nuxeo DM will not work properly: you should configure a proper database server such as PostgreSQL

Installing

After registering your Nuxeo DM instance to Nuxeo Connect, got to the Update Center section of the Administration Center, install DM-5.4.0.1-HF01 and semantic-entities package. Once installed, restart the Nuxeo DM instance. Upon restart you should see a new Entities container in the default domain.

Database configuration

To install with PostgreSQL, follow the usual instructions (in particular update the file `bin/nuxeo.conf` to register the templates default and postgresql) and then:

- On Nuxeo DM 5.4.0.1 only (already fixed in the future Nuxeo DM 5.4.1): configure the fulltext index for the field `dc:title` in the file `NUXEO_HOME/templates/postgresql/nxserver/config/default-repository-config.xml`, replace the indexing block by:

```
<indexing>
  <fulltext analyzer="english">
    <index name="default">
      <!-- all props implied -->
    </index>
    <index name="title">
      <field>dc:title</field>
    </index>
    <index name="description">
      <field>dc:description</field>
    </index>
  </fulltext>
</indexing>
```

- Drop a constraint that prevents the ACL optimizations store procedure to work concurrently (see [NXP-6038](#)), e.g. using `psql`:

```
\c nuxeo
nuxeo=# ALTER table hierarchy_read_acl DROP CONSTRAINT hierarchy_read_acl_pkey;
ALTER TABLE
nuxeo=# CREATE INDEX hierarchy_read_acl_id_idx ON hierarchy_read_acl (id);
CREATE INDEX
```

Apache Stanbol configuration

This addon requires an access to an external semantic engine web-service. By default the addon will use <https://stanbol.demo.nuxeo.com>. Your are however strongly advised to setup your own instance to avoid any issue pertaining to the confidentiality of your documents and the availability of the demo server.

Here are the instructions to setup your own Apache Stanbol server:

- Download the following jar (fise is the former name of Apache Stanbol): <http://dl.dropbox.com/u/5743203/IKS/snapshots/eu.iksproject.fise.launchers.lite-0.9-20101022.jar>
- Run with:

```
java -jar eu.iksproject.fise.launchers.lite-0.9-20101022.jar -p 9090
```

- Configure the file at:

```
NUXEO_HOME/nxserver/config/semanticentities.properties
```

to update the engine URL with:

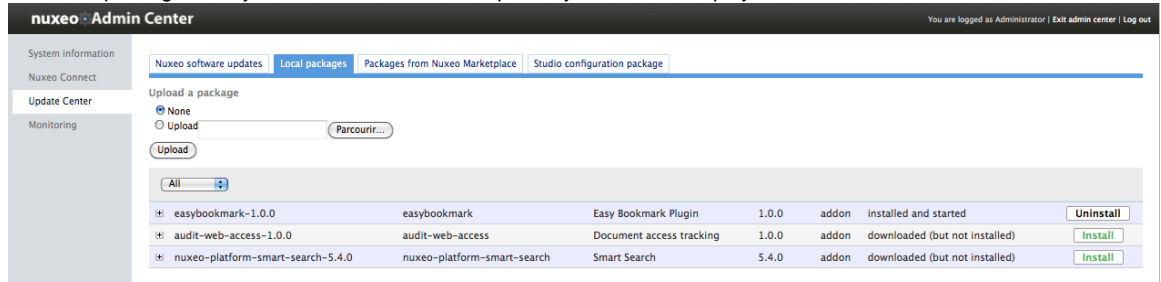
```
org.nuxeo.ecm.platform.semanticentities.stanbolUrl=http://localhost:9090/engines/
```

Uninstall a package

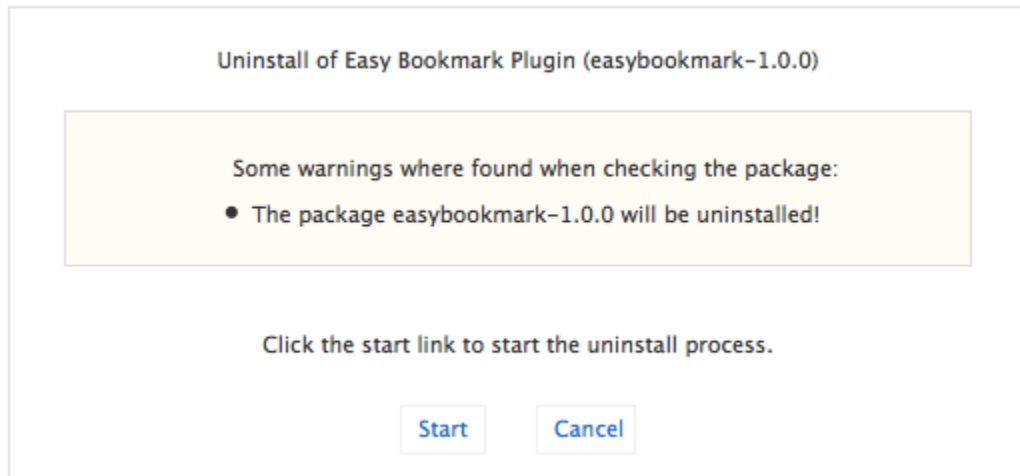
Uninstalling a package can be done from the Admin Center only.

To uninstall a package:

1. In the Admin Center, go on the **Local packages** tab of the **Update Center**.
The list of packages that you have downloaded and possibly installed is displayed.



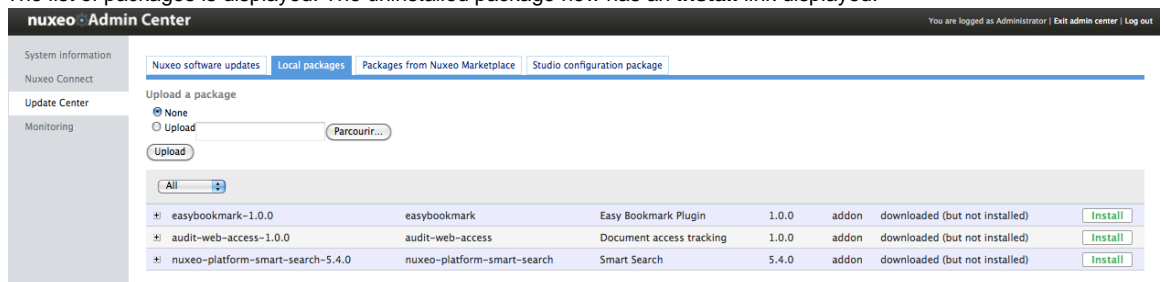
2. Click on the **Uninstall** link of the package you want to uninstall from your application.
A confirmation message is displayed.
3. Click on the **Start** button to confirm you want to uninstall the package.



4. When uninstallation is done, click on the **Finish** button.



The list of packages is displayed. The uninstalled package now has an **Install** link displayed.



✔ If the package doesn't support hotreload, you need to reboot the server so the unistallation is completed.

Update your instance with Studio configuration

Just like you can [install Nuxeo Marketplace packages](#) on your Nuxeo instance from the [Admin Center](#), you can also update it with your customization done in Nuxeo Studio. The **Studio configuration package** tab lists all the tagged versions of your Studio projects. But it also enables you to load your current Nuxeo Studio configuration. Hot reload of your Nuxeo Studio configuration enables you to test directly

your changes in your Nuxeo application, without having to restart the server or logout.

To apply your Studio changes to your application:

1. Make sure you have [registered your instance](#).
2. In your application, go on **Nuxeo Admin Center > Update Center > Nuxeo Studio**.
3. Click on the button **Update**.
The configuration is reloaded: if you configured a new button for instance, bound to a content automation chain, you can just test it directly.

Using Nuxeo as a service platform

Nuxeo EP exposes services in several ways:

- Java services (both local and remote)
- Web services (SOAP and JAX-RS)
- Content Automation

This chapter will give you a brief overview of how and when to use these services.

Java Services

Nuxeo EP 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:

```
RelationManager rm = Framework.getService(RelationManager.class)
```

Web Services (JAX-WS and JAX-RS)

Nuxeo EP includes 2 Web Service stacks:

- one for SOAP Web service (JAX-WS) based on SUN-JAX-WS or JBossWS depending on the application server
- one for REST Web service (JAX-RS) based on Jersey

Unlike the Remote Java service binding, Nuxeo EP does not provide a one to one mapping via Web Services.

There are at least 2 reasons for that:

- serialization cost
- granularity of the API

About the first point, Nuxeo Java service API exposes some complexe objects, the first one of them being the DocumentModel:

- that can contain a lot of data
- that supports lazy loading and streaming

This DocumentModel object is used as input or output of most services (and that makes sense for an ECM platform).

Completely serializing the DocumentModel object in XML is possible (we provide helper for that), but it's clearly not very efficient since it generate big XML objects.

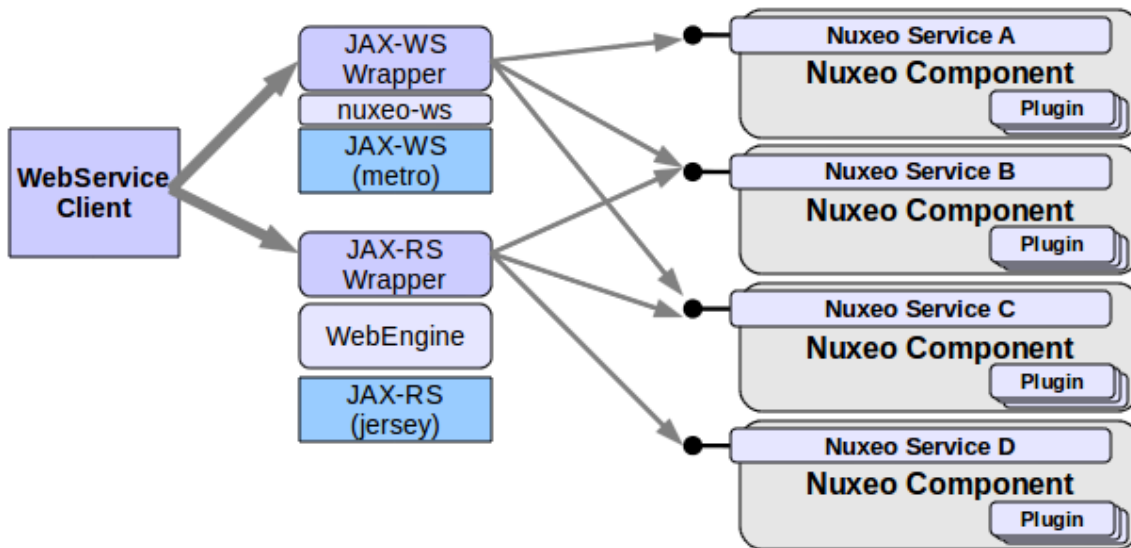
For the second point, the java API is finely grained and a direct WebService mapping would imply 2 problems :

- too many network calls
- difficult transaction management

Given these limitations, we choose :

- to expose the main features via specific Web Service
- to provide a framework to easily expose dedicated WebService API

Our approach is to provide the needed tools so that you can easily expose the exacte API needed by your remote client.



So depending of the needs of your application, the idea is to select the needed java services and create a java wrapper that provides a high level API and use Nuxeo framework to expose the wrapper via Web Service.

For more informations about declaring custom webservices on top of Nuxep EP, please see the [JAX-WS and JAX-RS binding](#) sections.

Content Automation

Writing a custom wrapper allows to expose a very optimized and very precise Web Service.

But on the other hand, it implies to write and deploy custom code.

In a lot of the cases, Content Automation provides an alternative to this approach with a very minimal performance overhead.

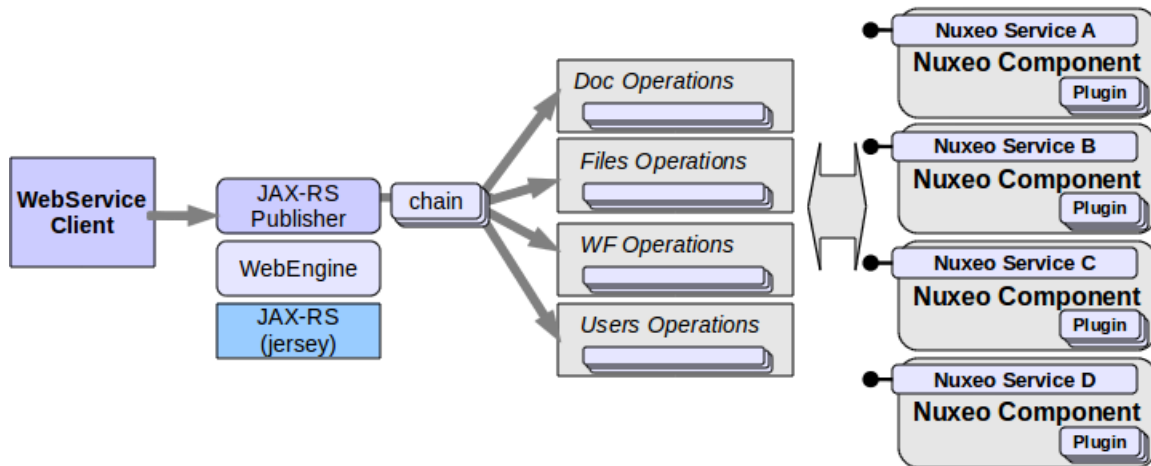
The idea is to define a simple and high level API on top of the java services : we call this Operations.

In order to avoid back and forth exchanges and costly serialization we want to chain the operations on the server side.

Using Content Automation, you can define you own Web Service API without writing code, you simply have to assemble the needed operations (via XML or via Nuxeo Studio).

Of course, if needed, you can also contribute your own custom java operations.

Operations and Operation chains are automatically exposed via a REST API using JSON marshaling.



For more information about Content Automation, please see the [Content Automation](#) chapter.

OpenSocial

Since 5.4.2, Nuxeo EP is an OAuth service provider.

This means you can consume Nuxeo Automation REST Services from any OpenSocial compliant application.

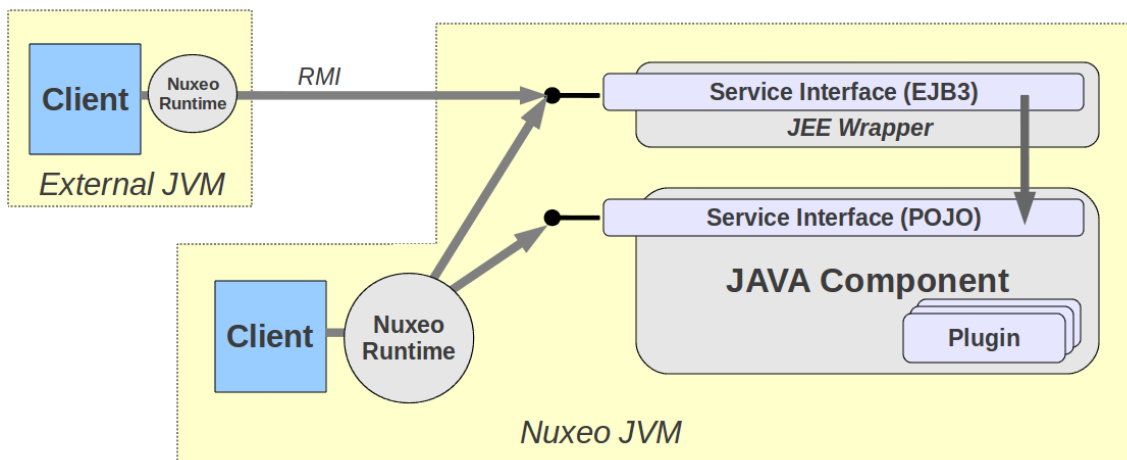
Please see the [OpenSocial Page](#) for more informations.

ECM Services in Nuxeo EP

Services in Nuxeo

All ECM features in Nuxeo EP are exposed as Java Services.

These services can be local (inside the same JVM) and most of them can also be called remotely via RMI if your target Application Server supports EJB3 remoting.



All services are defined by a Java Interface and are accessed via this interface.

Nuxeo Runtime provides a lookup system to access a Service:

```
MyEcmService service = Framework.getService(MyEcmService.class);
```

This lookup call encapsulate the potential JNDI lookup to access a remote service if your application server and configuration require it.

If you want explicitly to access a local service you can also use:

```
MyEcmService service = Framework.getLocalService(MyEcmService.class);
```

Inside the Seam WebLayer, most common Nuxeo Services are also wrapped as Seam components so you can directly get them via injection:

```
@In(create = true, required = false)
protected CoreSession documentManager;

@In(create = true)
protected RelationManager relationManager;
```

How to know what services are available

Nuxeo EP is a big platform and it provides a lot of different services.

You can use [Platform Explorer WebSite](#) or [platform explorer addon](#) to see the services included in a given distribution of Nuxeo EP.

Listing ECM Services

Error rendering macro 'gadget' : Error rendering gadget [<http://explorer.nuxeo.org/nuxeo/site/gadgets/services/services.xml>]

WebServices

JAX-WS bindings

Nuxeo EP includes a JAX-WS compliant implementation to expose SOAP based webservices.

- Sun JAX-WS (Metro) is used on Tomcat and JBoss 4.x distributions
- JBossWS is used on JBoss 5.x distributions

As explained earlier in these pages, JAX-WS is not the preferred way to expose WebService on top of Nuxeo Platform, this is the reason why, the default distributions don't expose a lot of SOAP based WebServices.

By default, the exposed SOAP WebServices 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

If you want to access the SOAP Webservice :

- To list all the deployed endpoints
 - <http://server:port/nuxeo/websevice/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 WebServices 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 WebServices 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 [here](#).

You might also want to build a **WebService client** in Nuxeo to be able to query a remote WebService. A simple example is available [here](#).

JAX-RS REST APIs

Since 5.2 Nuxeo EP includes a JAX-RS framework to allow easy publishing of Java classes via REST APIs.

Nuxeo WebEngine provides a nice integration model so that you can very simply write Java classes that use Nuxeo Services and Documents and expose it via REST with JSON or XML marshaling.

JAX-RS / WebEngine is the current preferred solution for exposing custom WebService on top of the platform and it is also the base of Content Automation WebService APIs.

You can get more informations about WebEngine and JAX-RS [here](#).

RESTLETs in Nuxeo

The Restlet framework is integrated inside Nuxeo since 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, Restlet 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%20DM-5.5/viewExtensionPoint/org.nuxeo.ecm.platform.ui.web.restAPI.service.PluggableRestletService--restlets>

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](#)
- [Getting an instance of the WebService client](#)
- [Calling a WebService method](#)

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/wsdl`s 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/wsdl</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/wsdl`s 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/wsdl/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.restfulwebservice.net/wcf/WeatherForecastService.svc?wsdl"),
    new QName("http://www.restfulwebservice.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.restfulwebservice.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.

Building a SOAP based WebService in Nuxeo

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 [here](#).

On this page

- [WebService interface](#)
- [Webservice implementation](#)
- [WebService end point and URL mapping](#)
- [About code factorization](#)

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 either the `jaxws#ENDPOINT` or the `web#JBOSWS` extension point in the `deployment-fragment.xml` of your Nuxeo project, depending on the distribution you are using.

Tomcat and JBoss 4.x distributions

```
<extension target="jaxws#ENDPOINT">
  <endpoint name="nuxeosample"
    implementation="org.nuxeo.ecm.samples.ws.server.NuxeoSampleWSImpl"
    url-pattern="/webservices/nuxeosample" />
</extension>
```

JBoss 5.x distributions

```
<extension target="web#JBOSWS">
  <servlet>
    <description>NuxeoSampleWebService EndPoint</description>
    <display-name>NuxeoSampleWebService EndPoint</display-name>
    <servlet-name>NuxeoSampleWebServiceEndPoint</servlet-name>
    <servlet-class>org.nuxeo.ecm.samples.ws.server.NuxeoSampleWSImpl
    </servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>NuxeoSampleWebServiceEndPoint</servlet-name>
    <url-pattern>/webservices/nuxeosample</url-pattern>
  </servlet-mapping>
</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>.



Note for Nuxeo IDE users

For the moment, contributions to either the `jaxws#ENDPOINT` or the `web#JBOSWS` extension point in the `deployment-fragment.xml` of a Nuxeo project are not taken into account when using Nuxeo IDE.

A workaround is to uncheck the project from the *Deployment configurations* of Nuxeo IDE, build it and deploy the JAR file manually to the `nxserver/plugins` directory of the Nuxeo SDK.

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 [here](#) how to build a client-side SOAP based WebService 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

- Introduction
- Static Trust Store and Key Store
- Dynamic Trust Store
- Adding your Certificates into the default Trust Store
- Troubles



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)
```

Content Automation

Content Automation is a Nuxeo service that exposes commons 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 operation chains (or macro operations). These operations are also exposed remotely through a [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 operation 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. If you need to define dynamic operation parameters (which 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.



This documentation is oriented toward developers looking for extending and/or using the REST API of Content Automation. [You can also easily and graphically create new chains and actions using Nuxeo Studio.](#)

Outline of this document:

- [What is an Operation?](#)
- [What is an Operation Chain?](#)
- [Working With Operations](#)
 - [The Operation Input](#)
 - [The Operation Parameters](#)
 - [The Operation Output](#)
 - [Contributing new Input/Output types \(since Nuxeo 5.4.2\)](#)
 - [Using scripting expression in operations](#)

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 Operation 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 operation chain that creates a document, then attaches a blob into 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.

This is called in Nuxeo Automation an **operation chain**. Operation 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 operation 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 Operation Chain](#) page to see how you can define and contribute a new operation chain using Nuxeo extension points.

Working With Operations

In this section I will discuss more about how operations work and how they can be chained to obtain working operation 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 **Execute Operation** 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 view as a finite operation 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 cause and effects you want - you can create your ECM universe using operations.

Now, lets discuss about the bricks that compose an operation:

1. An operation has an input (provided by the cause)
2. An operation may have zero or more parameters (used to parametrize the way an operation is behaving)
3. 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.

So we can say that an operation is working on a Document or a Blob.

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 is 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" Nuxeo Component.
- an expression - this represents a MVEL expression (which is compatible with basic EL expressions) that can output dynamic values. Whe

using expressions you *must* prepend it with the `_expr:_` prefix.
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 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 Nuxeo 5.4.2)

If needed, 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 put data extracted from the POJO object into the JSON object
- a reading method that get data from the JSON object and builds a POJO object from
- a reference builder that extracts the server reference from a POJO object
- a reference resolver that provides access to 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 an implementation of `org.nuxeo.ecm.automation.server.jaxrs.io.JsonMarshaller` interface. The implementation class is to be contributed to the automation server component using the `marshallers` extension point.

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.JsonMarshalling.addMarshaller(JsonMarshaller<?>)`.

Here is some snip-sets extracted from the [Automation Server Rest Test Suite](#).

```

...
public class MyObject {
    String message;

    String getMessage() {
        ....
    }
    ...
}
...
public class MyObjectMarshaller extends
org.nuxeo.ecm.automation.client.jaxrs.spi.marshallers.BeanMarshaller<MyObject> {
    public MyObjectClientMarshaller() {
        super(MyObject.class);
    }
}
...
org.nuxeo.ecm.automation.client.jaxrs.spi.JsonMarshalling.addMarshaller(new
MyObjectMarshaller());
...
public class MyObjectMarshaller extends
org.nuxeo.ecm.automation.server.jaxrs.io.marshallers.BeanMarshaller<MyObject> {
    public MyObjectClientMarshaller() {
        super(MyObject.class);
    }
}
...
<extension target="org.nuxeo.ecm.automation.server.AutomationServer"
    point="marshallers">

    <marshaller class="org.nuxeo.ecm.automation.server.test.MyObjectServerMarshaller"
/>

</extension>
...

```

Using scripting expression in operations

Operations can be parametrized using MVEL scripting expressions. For more details about scripting you can look at [Use of MVEL in Automation Chains](#).

Operations Index

- User Interface
 - [Add Error Message](#)
 - [Add Info Message](#)
 - [Change Tab](#)
 - [Clear Clipboard](#)
 - [Clear Worklist](#)
 - [Download file](#)
 - [Navigate to Document](#)
 - [Push to Clipboard](#)
 - [Push to Seam Context](#)
 - [Push to Worklist](#)
 - [Refresh](#)
 - [Show Create Document Page](#)
- Files
 - [Attach File](#)
 - [Export to File](#)
 - [Get Document File](#)

- Get Document Files
- HTTP Post
- Remove File
- Set File Name
- Zip
- Document
 - Check In
 - Check Out
 - Copy
 - Create
 - Delete
 - Filter List
 - Follow Life Cycle Transition
 - Get Child
 - Get Children
 - Get Parent
 - Lock
 - Move
 - Multi-Publish
 - Publish Document
 - Remove ACL
 - Remove Property
 - Reset
 - Save
 - Set ACL
 - Set File
 - Snapshot Version
 - Unlock
 - Update Properties
 - Update Property
- Fetch
 - Context Document
 - Document
 - Fetch By Property
 - File From URL
 - PageProvider
 - Query
 - UI Clipboard
 - UI Current Document
 - UI Current Domain
 - UI Current Workspace
 - UI Document From Seam
 - UI Selected documents
 - UI Worklist
- Conversion
 - Convert To PDF
 - Get image view
 - Render Document
 - Render Document Feed
 - Resize a picture
- Services
 - Create Document from file
 - Create Document from file in User's workspace
 - Create Picture
 - Create Relation
 - Create task
 - Get Linked Documents
 - Get directory entries
 - Get user tasks
 - List available actions
 - Log Event In Audit
 - Query Audit Service
 - Retrieve counters values
 - UserProcessPageProvider
 - UserTaskPageProvider
- Users & Groups
 - Get Principal Emails
 - Get User Setting
 - Get User Settings
 - Get Users and Groups
 - Login As
 - Logout

- Push & Pop
 - Pop Document
 - Pop Document List
 - Pop File
 - Pop File List
 - Pull Document
 - Pull Document List
 - Pull File
 - Pull File List
 - Push Document
 - Push Document List
 - Push File
 - Push File List
- Execution Context
 - Restore Document Input
 - Restore Documents Input
 - Restore File Input
 - Restore Files Input
 - Set Context Variable
 - Set Context Variable From Input
- Execution Flow
 - Run Document Chain
 - Run Operation
 - Save Session
- Scripting
 - Run Input Script
 - Run Script
- Notification
 - Send E-Mail
 - Send Event

User Interface

Add Error Message

Add a message to be displayed in case the chain execution fails. This is a void operation - the input will be returned back as output General Information

Category: User Interface

Operation Id: Seam.AddErrorMessage

Parameters

Name	Type	Required	Default Value
message	string	true	

Signature

Inputs: void

Outputs: void

Add Info Message

Add a message to be displayed after the chain is successfully executed. This is a void operation - the input will be returned back as output General Information

Category: User Interface

Operation Id: Seam.AddInfoMessage

Parameters

Name	Type	Required	Default Value
message	string	true	

Signature

Inputs: void

Outputs: void

Change Tab

Change the selected tab for the current document. Preserve the current input. General Information

Category: User Interface
Operation Id: Seam.ChangeTab
Parameters

Name	Type	Required	Default Value
tab	string	true	

Signature

Inputs: void
Outputs: void

Clear Clipboard

Clear the clipboard content. General Information

Category: User Interface
Operation Id: Seam.ClearClipboard
Parameters

Name	Type	Required	Default Value
------	------	----------	---------------

Signature

Inputs: void
Outputs: void

Clear Worklist

Clear the worklist content. General Information

Category: User Interface
Operation Id: Seam.ClearWorklist
Parameters

Name	Type	Required	Default Value
------	------	----------	---------------

Signature

Inputs: void
Outputs: void

Download file

Download a file General Information

Category: User Interface
Operation Id: Seam.DownloadFile
Parameters

Name	Type	Required	Default Value
------	------	----------	---------------

Signature

Inputs: blob
Outputs: void

Navigate to Document

Navigate to the input document. The outcome of the UI action will be stored in the operation chain context as the 'Outcome' variable. Returns back the document. General Information

Category: User Interface
Operation Id: Seam.NavigateTo
Parameters

Name	Type	Required	Default Value
view	string	false	

Signature

Inputs: document
Outputs: document

Push to Clipboard

Add a input document(s) to clipboard. Returns back the document(s) General Information

Category: User Interface
Operation Id: Seam.AddToClipboard
Parameters

Name	Type	Required	Default Value
------	------	----------	---------------

Signature

Inputs: document, documents
Outputs: document, documents

Push to Seam Context

Push the current input document into Seam context. Returns back the document. General Information

Category: User Interface
Operation Id: Seam.PushDocument
Parameters

Name	Type	Required	Default Value
name	string	true	
scope	string	true	session, conversation, page, event

Signature

Inputs: document
Outputs: document

Push to Worklist

Add the input document(s) to worklist. Returns back the document(s) General Information

Category: User Interface
Operation Id: Seam.AddToWorklist
Parameters

Name	Type	Required	Default Value
------	------	----------	---------------

Signature

Inputs: document, documents
Outputs: document, documents

Refresh

Refresh the UI cache. This is a void operation - the input object is returned back as the oputput General Information

Category: User Interface
Operation Id: Seam.Refresh
Parameters

Name	Type	Required	Default Value
------	------	----------	---------------

Signature

Inputs: void
Outputs: void

Show Create Document Page

Show the document creation form given a type. This is a void operation - the input object is returned back as the oputput General Information

Category: User Interface

Operation Id: Seam.CreateDocumentForm Parameters

Name	Type	Required	Default Value
type	string	true	

Signature

Inputs: void

Outputs: void

Files

Attach File

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. General Information

Category: Files

Operation Id: Blob.Attach
Parameters

Name	Type	Required	Default Value
document	document	true	
save	boolean	false	true
xpath	string	false	[file:content]

Signature

Inputs: blob, bloblist

Outputs: blob, bloblist

Export to File

Save the input blob(s) as a file(s) into the given target directory. The blob(s) filename is used as the file name. You can specify an optional **prefix** string to prepend to the file name. Return back the blob(s). General Information

Category: Files

Operation Id: Blob.ToFile
Parameters

Name	Type	Required	Default Value
directory	string	true	
prefix	string	false	

Signature

Inputs: blob, bloblist

Outputs: blob, bloblist

Get Document File

Gets a file attached to the input document. The file location is specified using an xpath to the blob property of the document. Returns the file. General Information

Category: Files

Operation Id: Blob.Get
Parameters

Name	Type	Required	Default Value
xpath	string	false	[file:content]

Signature

Inputs: document, documents

Outputs: blob, bloblist

Get Document Files

Gets a list of files that are attached on the input document. The files location should be specified using the blob list property xpath. Returns a list of files. General Information

Category: Files

Operation Id: Blob.GetList

Parameters

Name	Type	Required	Default Value
xpath	string	false	files:files

Signature

Inputs: document, documents

Outputs: bloblist, bloblist

HTTP Post

Post the input file to a target HTTP URL. Returns back the input file. General Information

Category: Files

Operation Id: Blob.Post

Parameters

Name	Type	Required	Default Value
url	string	true	

Signature

Inputs: blob, bloblist

Outputs: blob, bloblist

Remove File

Remove the file attached to the input document as specified by the 'xpath' parameter. If the 'xpath' point to a blob list then the list will be cleared. If the file to remove is part of a list it will be removed from the list otherwise the 'xpath' should point to a blob property that will be removed. If the save parameter is set the document modification will be automatically saved. Return the document. General Information

Category: Files

Operation Id: Blob.Remove

Parameters

Name	Type	Required	Default Value
save	boolean	false	true
xpath	string	false	[file:content]

Signature

Inputs: document, documents

Outputs: document, documents

Set File Name

Modify the filename of a file stored in the input document. The file is found in the input document given its xpath specified through the 'xpath' parameter. Return back the input document. General Information

Category: Files

Operation Id: Blob.SetFilename

Parameters

Name	Type	Required	Default Value
name	string	true	
save	boolean	false	true

xpath	string	false	[file:content]
-------	--------	-------	----------------

Signature

Inputs: document, documents

Outputs: document, documents

Zip

Creates a zip file from the input file(s). If no file name is given, the first file name in the input will be used. Returns the zip file. General Information

Category: Files

Operation Id: Blob.CreateZip

Parameters

Name	Type	Required	Default Value
filename	string	false	

Signature

Inputs: blob, bloblist

Outputs: blob, blob

Document

Check In

Checks in the input document. Returns back the document. General Information

Category: Document

Operation Id: Document.CheckIn

Parameters

Name	Type	Required	Default Value
version	string	true	minor, major
comment	string	false	
versionVarName	string	false	

Signature

Inputs: document, documents

Outputs: document, documents

Check Out

Checks out the input document. Returns back the document. General Information

Category: Document

Operation Id: Document.CheckOut

Parameters

Name	Type	Required	Default Value
------	------	----------	---------------

Signature

Inputs: document, documents

Outputs: document, documents

Copy

Copy the input document into the given folder. The name parameter will be used as the copy name otherwise if not specified the original name will be preserved. The target folder can be specified as an absolute or relative path (relative to the input document) as an UID or by using an EL expression. Return the newly created document (the copy). General Information

Category: Document

Operation Id: Document.Copy

Parameters

Name	Type	Required	Default Value
target	document	true	
name	string	false	

Signature

Inputs: document, documents

Outputs: document, documents

Create

Create a new document in the input folder. You can initialize the document properties using the 'properties' parameter. The properties are specified as *key=value* pairs separated by a new line. The key used for a property is the property xpath. To specify multi-line values you can use a \ charcater followed by a new line.

Example:

```
dc:title=The Document Title
```

```
dc:description=foo bar
```

. Returns the created document. General Information

Category: Document

Operation Id: Document.Create

Parameters

Name	Type	Required	Default Value
type	string	true	
name	string	false	
properties	properties	false	

Signature

Inputs: document, documents

Outputs: document, documents

Delete

Delete the input document. The previous context input will be restored for the next operation. General Information

Category: Document

Operation Id: Document.Delete

Parameters

Name	Type	Required	Default Value
------	------	----------	---------------

Signature

Inputs: document, documents

Outputs: document, documents

Filter List

Filter the input list of documents given a condition. The condition can be expressed using 4 parameters: types, facets, lifecycle and condition. If more than one parameter is specified an AND will be used to group conditions.

The 'types' paramter can take a comma separated list of document type: File,Note.

The 'facet' parameter can take a single facet name.

The 'life cycle' parameter takes a name of a life cycle state the document should have.

The 'condition' parameter can take any EL expression.

Returns the list of documents that match the filter condition. General Information

Category: Document

Operation Id: Document.Filter

Parameters

Name	Type	Required	Default Value
------	------	----------	---------------

class	string	false	Any, Regular Document, Document Link, Published Document, Document Proxy, Document Version, Immutable Document, Mutable Document
condition	string	false	
facet	string	false	
lifecycle	string	false	
pathStartsWith	string	false	
types	string	false	

Signature

Inputs: documents

Outputs: documents

Follow Life Cycle Transition

Follow the given transition on the input document life cycle state General Information

Category: Document

Operation Id: Document.SetLifeCycle
Parameters

Name	Type	Required	Default Value
value	string	true	

Signature

Inputs: document, documents

Outputs: document, documents

Get Child

Get a child document given its name. Take as input the parent document and return the child document. General Information

Category: Document

Operation Id: Document.GetChild
Parameters

Name	Type	Required	Default Value
name	string	true	

Signature

Inputs: document, documents

Outputs: document, documents

Get Children

Get the children of a document. The list of children will become the input for the next operation General Information

Category: Document

Operation Id: Document.GetChildren
Parameters

Name	Type	Required	Default Value
------	------	----------	---------------

Signature

Inputs: document, documents

Outputs: documents, documents

Get Parent

Get the parent document of the input document. The parent document will become the input for the next operation. You can use the 'type'

parameter to specify which parent to select from the document ancestors General Information

Category: Document
Operation Id: Document.GetParent
Parameters

Name	Type	Required	Default Value
type	string	false	

Signature

Inputs: document, documents
Outputs: document, documents

Lock

Lock the input document for the current user. Returns back the locked document. General Information

Category: Document
Operation Id: Document.Lock
Parameters

Name	Type	Required	Default Value
owner	string	false	

Signature

Inputs: document, documents
Outputs: document, documents

Move

Move the input document into the target folder. General Information

Category: Document
Operation Id: Document.Move
Parameters

Name	Type	Required	Default Value
target	document	true	
name	string	false	

Signature

Inputs: document, documents
Outputs: document, documents

Multi-Publish

Publish the input document(s) into several target sections. The target is evaluated to a document list (can be a path, UID or EL expression). Existing proxy is overridden if the override attribute is set. Returns a list with the created proxies. General Information

Category: Document
Operation Id: Document.MultiPublish
Parameters

Name	Type	Required	Default Value
target	documents	true	
override	boolean	false	true

Signature

Inputs: document, documents
Outputs: documents, documents

Publish Document

Publish the input document into the target section. Existing proxy is overridden if the override attribute is set. Return the created proxy. General Information

Category: Document

Operation Id: Document.Publish
Parameters

Name	Type	Required	Default Value
target	document	true	
override	boolean	false	true

Signature

Inputs: document, documents

Outputs: document, documents

Remove ACL

Remove a named Acces Control List from the input document(s). Returns the document(s). General Information

Category: Document

Operation Id: Document.RemoveACL
Parameters

Name	Type	Required	Default Value
acl	string	true	

Signature

Inputs: document, documents

Outputs: document, documents

Remove Property

Remove the given property of the input document(s) as specified by the 'xpath' parameter. If the property points to a list then clear the list. Removing a property means setting it to null. Return the document(s). General Information

Category: Document

Operation Id: Document.RemoveProperty
Parameters

Name	Type	Required	Default Value
xpath	string	true	
save	boolean	false	true

Signature

Inputs: document, documents

Outputs: document, documents

Reset

Reload the input document from the repository. Any previous modification made by the chain on this document will be lost if these modifications were not saved. Return the reloaded document. General Information

Category: Document

Operation Id: Document.Reload
Parameters

Name	Type	Required	Default Value
------	------	----------	---------------

Signature

Inputs: document, documents

Outputs: document, documents

Save

Save in the repository any modification that was done on the input document. Returns the saved document. General Information

Category: Document

Operation Id: Document.Save

Parameters

Name	Type	Required	Default Value
------	------	----------	---------------

Signature

Inputs: document, documents

Outputs: document, documents

Set ACL

Set Acces Control Entry on the input document(s). Returns the document(s). General Information

Category: Document

Operation Id: Document.SetACE

Parameters

Name	Type	Required	Default Value
permission	string	true	
user	string	true	
acl	string	false	local
grant	boolean	false	true
overwrite	boolean	false	true

Signature

Inputs: document, documents

Outputs: document, documents

Set File

Set the input file to the given property on the input document. 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 document. General Information

Category: Document

Operation Id: Blob.Set

Parameters

Name	Type	Required	Default Value
file	blob	true	
save	boolean	false	true
xpath	string	false	[file:content]

Signature

Inputs: document, documents

Outputs: document, documents

Snapshot Version

Create a new version for the input document. Any modification made on the document by the chain will be automatically saved. Increment version if this was specified through the 'snapshot' parameter. Returns the live document (not the version). General Information

Category: Document

Operation Id: Document.CreateVersion

Parameters

Name	Type	Required	Default Value
increment	string	false	None, Minor, Major

Signature

Inputs: document, documents

Outputs: document, documents

Unlock

Unlock the input document. The unlock will be executed in the name of the current user. An user can unlock a document only if has the UNLOCK permission granted on the document or if it the same user as the one that locked the document. Return the unlocked document General Information

Category: Document

Operation Id: Document.Unlock
Parameters

Name	Type	Required	Default Value
------	------	----------	---------------

Signature

Inputs: document, documents

Outputs: document, documents

Update Properties

Set multiple properties on the input document. The properties are specified as *key=value* pairs separated by a new line. The key used for a property is the property xpath. To specify multi-line values you can use a \ charcater followed by a new line.

Example:

dc:title=The Document Title

dc:description=foo bar

. Returns back the updated document. General Information

Category: Document

Operation Id: Document.Update
Parameters

Name	Type	Required	Default Value
properties	properties	true	
save	boolean	false	true

Signature

Inputs: document, documents

Outputs: document, documents

Update Property

Set a single property value on the input document. The property is specified using its xpath. The document is automatically saved if 'save' parameter is true. If you unset the 'save' you need to save it later using Save Document operation. Return the modified document. General Information

Category: Document

Operation Id: Document.SetProperty
Parameters

Name	Type	Required	Default Value
value	serializable	true	
xpath	string	true	
save	boolean	false	true

Signature

Inputs: document, documents

Outputs: document, documents

Fetch

Context Document

Fetch the input of the context as a document. The document will become the input for the next operation. General Information

Category: Fetch

Operation Id: Context.FetchDocument

Parameters

Name	Type	Required	Default Value
------	------	----------	---------------

Signature

Inputs: void

Outputs: document

Document

Fetch a document from the repository given its reference (path or UID). The document will become the input of the next operation. General Information

Category: Fetch

Operation Id: Document.Fetch

Parameters

Name	Type	Required	Default Value
value	document	true	

Signature

Inputs: void

Outputs: document

Fetch By Property

For each specified string property value, fetch all documents that match the property and the optional where clause. Matching documents are collected into a list and the returned to the next operation. The operation has no input. General Information

Category: Fetch

Operation Id: Document.FetchByProperty

Parameters

Name	Type	Required	Default Value
property	string	true	
values	stringlist	true	
query	string	false	

Signature

Inputs: void

Outputs: documents

File From URL

Creates a file from a given URL. The file parameter specifies how to retrieve the file content. It should be an URL to the file you want to use as the source. You can also use an expression to get an URL from the context. Returns the created file. General Information

Category: Fetch

Operation Id: Blob.Create

Parameters

Name	Type	Required	Default Value
file	resource	true	
encoding	string	false	
filename	string	false	
mime-type	string	false	

Signature

Inputs: void
Outputs: blob

PageProvider

Perform a query or a named provider query on the repository. Result is paginated. The query result will become the input for the next operation. If no query or provider name is given, a query returning all the documents that the user has access to will be executed. General Information

Category: Fetch

Operation Id: Document.PageProvider
Parameters

Name	Type	Required	Default Value
language	string	false	NXQL
page	integer	false	
pageSize	integer	false	
providerName	string	false	
query	string	false	
queryParams	stringlist	false	
sortInfo	stringlist	false	

Signature

Inputs: void
Outputs: documents

Query

Perform a query on the repository. The query result will become the input for the next operation. General Information

Category: Fetch

Operation Id: Document.Query
Parameters

Name	Type	Required	Default Value
query	string	true	
language	string	false	NXQL

Signature

Inputs: void
Outputs: documents

UI Clipboard

Get clipboard content from the UI context. General Information

Category: Fetch

Operation Id: Seam.FetchFromClipboard
Parameters

Name	Type	Required	Default Value
------	------	----------	---------------

Signature

Inputs: void
Outputs: documents

UI Current Document

Get the current Document from the UI context. General Information

Category: Fetch

Operation Id: Seam.GetCurrentDocument
Parameters

Name	Type	Required	Default Value
------	------	----------	---------------

Signature

Inputs: void
Outputs: document

UI Current Domain

Get the current Domain from the UI context. General Information

Category: Fetch
Operation Id: Seam.GetCurrentDomain
Parameters

Name	Type	Required	Default Value
------	------	----------	---------------

Signature

Inputs: void
Outputs: document

UI Current Workspace

Get the current Workspace from the UI context. General Information

Category: Fetch
Operation Id: Seam.GetCurrentWorkspace
Parameters

Name	Type	Required	Default Value
------	------	----------	---------------

Signature

Inputs: void
Outputs: document

UI Document From Seam

Fetch a document from the Seam context given its Seam name. General Information

Category: Fetch
Operation Id: Seam.FetchDocument
Parameters

Name	Type	Required	Default Value
name	string	true	

Signature

Inputs: void
Outputs: document

UI Selected documents

Fetch the documents selected in the current folder listing General Information

Category: Fetch
Operation Id: Seam.GetSelectedDocuments
Parameters

Name	Type	Required	Default Value
------	------	----------	---------------

Signature

Inputs: void
Outputs: documents

UI Worklist

Get worklist content from the UI context. General Information

Category: Fetch

Operation Id: Seam.FetchFromWorklist
Parameters

Name	Type	Required	Default Value
------	------	----------	---------------

Signature

Inputs: void

Outputs: documents

Conversion

Convert To PDF

Convert the input file to a PDF and return the new file. General Information

Category: Conversion

Operation Id: Blob.ToPDF
Parameters

Name	Type	Required	Default Value
------	------	----------	---------------

Signature

Inputs: document, blob, bloblist

Outputs: blob, blob, bloblist

Get image view

Get an image from a Picture document. General Information

Category: Conversion

Operation Id: Picture.getView
Parameters

Name	Type	Required	Default Value
viewName	string	false	

Signature

Inputs: document

Outputs: blob

Render Document

Get a document or a list of document in input and outputs one or more blobs that contain a rendered view for each input document given a rendering template. The template attribute may contain either the template content either a template URI. Template URIs are strings in the form 'template:template_name' and will be located using the runtime resource service. Return the rendered file(s) as blob(s) General Information

Category: Conversion

Operation Id: Render.Document
Parameters

Name	Type	Required	Default Value
template	string	true	
filename	string	false	output.ftl
mimetype	string	false	text/xml
type	string	false	ftl, mvel

Signature

Inputs: document, documents

Outputs: blob, bloblist

Render Document Feed

Get a list of documents as input and outputs a single blob containing the rendering of the document list. The template attribute may contain either the template content either a template URI. Template URIs are strings in the form 'template:template_name' and will be located using the runtime resource service. Return the rendered blob General Information

Category: Conversion

Operation Id: Render.DocumentFeed

Parameters

Name	Type	Required	Default Value
template	string	true	
filename	string	false	output.ftl
mimetype	string	false	text/xml
type	string	false	ftl, mvel

Signature

Inputs: documents

Outputs: blob

Resize a picture

Use conversion service to resize a picture contained in a Document or a Blob General Information

Category: Conversion

Operation Id: Picture.resize

Parameters

Name	Type	Required	Default Value
maxHeight	int	true	
maxWidth	int	true	

Signature

Inputs: document, blob

Outputs: blob, blob

Services

Create Document from file

Create Document(s) from Blob(s) using the FileManagerService. General Information

Category: Services

Operation Id: FileManager.Import

Parameters

Name	Type	Required	Default Value
overwrite	boolean	false	

Signature

Inputs: bloblist, blob

Outputs: documents, document

Create Document from file in User's workspace

Create Document(s) in the user's workspace from Blob(s) using the FileManagerService. General Information

Category: Services

Operation Id: UserWorkspace.CreateDocumentFromBlob

Parameters

Name	Type	Required	Default Value
------	------	----------	---------------

Signature

Inputs: bloblist, blob
Outputs: documents, document

Create Picture

Create a Picture document in the input folder. You can initialize the document properties using the 'properties' parameter. The properties are specified as key=value pairs separated by a new line. The key originalPicture is used to reference the JSON representation of the Blob for the original picture. The pictureTemplates parameter can be used to define the size of the different views to be generated, each line must be a JSONObject { title="title", description="description", maxsize=maxsize}. Returns the created document. General Information

Category: Services
Operation Id: Picture.Create
Parameters

Name	Type	Required	Default Value
name	string	false	
pictureTemplates	properties	false	
properties	properties	false	

Signature

Inputs: document, documents
Outputs: document, documents

Create Relation

Create a relation between 2 documents. The subject of the relation will be the input of the operation and the object of the relation will be retrieved from the context using the 'object' field. The 'predicate' field specify the relation predicate. The 'outgoing' flag indicates the direction of the relation - the default is false which means the relation will go from the input object to the object specified as 'object' parameter. Return back the subject document. General Information

Category: Services
Operation Id: Relations.CreateRelation
Parameters

Name	Type	Required	Default Value
object	document	true	
predicate	string	true	
outgoing	boolean	false	false

Signature

Inputs: document, documents
Outputs: document, documents

Create task

Enable to create a task bound to the document.

Directive, **comment** and **due date** will be displayed in the task list of the user. In **accept operation chain** and **reject operation chain** fields, you can put the operation chain ID of your choice among the one you contributed. Those operations will be executed when the user validates the task, depending on whether he accepts or rejects the task. You have to specify a variable name (the **key for ...** parameter) to resolve target users and groups to which the task will be assigned. You can use Get Users and Groups to update a context variable with some users and groups. If you check **create one task per actor**, each of the actors will have a task to achieve, versus "the first who achieve the task makes it disappear for the others".

General Information

Category: Services
Operation Id: Workflow.CreateTask
Parameters

Name	Type	Required	Default Value
task name	string	true	

due date	date	false	
directive	string	false	
comment	string	false	
accept operation chain	string	false	
reject operation chain	string	false	
variable name for actors prefixed ids	string	false	
additional list of actors prefixed ids	stringlist	false	
create one task per actor	boolean	false	true

Signature

Inputs: document, documents

Outputs: document, documents

Get Linked Documents

Get the relations for the input document. The 'outgoing' parameter can be used to specify whether outgoing or incoming relations should be returned. Returns a document list. General Information

Category: Services

Operation Id: Relations.GetRelations

Parameters

Name	Type	Required	Default Value
predicate	string	true	
outgoing	boolean	false	

Signature

Inputs: document

Outputs: documents

Get directory entries

Get the entries of a directory. This is returning a blob containing a serialized JSON array. The input document, if specified, is used as a context for a potential local configuration of the directory. General Information

Category: Services

Operation Id: Directory.Entries

Parameters

Name	Type	Required	Default Value
directoryName	string	true	
lang	string	false	
translateLabels	boolean	false	

Signature

Inputs: bloblist, blob

Outputs: documents, document

Get user tasks

List tasks assigned to this user or one of its group. Task properties are serialized using JSON and returned in a Blob. General Information

Category: Services

Operation Id: Workflow.GetTask

Parameters

Name	Type	Required	Default Value
------	------	----------	---------------

Signature

Inputs: void
Outputs: blob

List available actions

Retrieve list of available actions for a given category. Action context is built based on the Operation context (currentDocument will be fetched from Context if not provided as input). If this operation is executed in a chain that initialized the Seam context, it will be used for Action context. General Information

Category: Services
Operation Id: Actions.GET
Parameters

Name	Type	Required	Default Value
category	string	true	
lang	string	false	

Signature

Inputs: void, document
Outputs: blob, blob

Log Event In Audit

Log events into audit for each of the input document. The operation accept as input one ore more documents that are returned back as the output. General Information

Category: Services
Operation Id: Audit.Log
Parameters

Name	Type	Required	Default Value
event	string	true	
category	string	false	Automation
comment	string	false	

Signature

Inputs: document, documents
Outputs: document, documents

Query Audit Service

Execute a JPA query against the Audit Service. This is returning a blob with the query result. The result is a serialized JSON array. You can use the context to set query variables but you must prefix using 'audit.query.' the context variable keys that match the ones in the query. General Information

Category: Services
Operation Id: Audit.Query
Parameters

Name	Type	Required	Default Value
query	string	true	
maxResults	int	false	
pageNo	int	false	

Signature

Inputs: void
Outputs: blob

Retrieve counters values

Retrieve data collected by one or more Counters. General Information

Category: Services

Operation Id: Counters.GET

Parameters

Name	Type	Required	Default Value
counterNames	stringlist	true	

Signature

Inputs: void

Outputs: blob

UserProcessPageProvider

Returns the current user's processes. General Information

Category: Services

Operation Id: Workflow.UserProcessPageProvider

Parameters

Name	Type	Required	Default Value
language	string	false	
page	integer	false	
pageSize	integer	false	

Signature

Inputs: void

Outputs: blob

UserTaskPageProvider

Returns the current user's processes. General Information

Category: Services

Operation Id: Workflow.UserTaskPageProvider

Parameters

Name	Type	Required	Default Value
language	string	false	
page	integer	false	
pageSize	integer	false	

Signature

Inputs: void

Outputs: blob

Users & Groups

Get Principal Emails

Fetch the principal emails that have a given permission on the input document and then set them in the context under the given key variable name. The operation returns the input document. You can later use the list of principals set by this operation on the context from another operation. The 'key' argument represents the variable name and the 'permission' argument the permission to check. If the 'ignore groups' argument is false then groups are recursively resolved, extracting user members of these groups. Be **warned** that this may be a very consuming operation.

Note that

- groups are not included
- the list pushed into the context is a string list of emails.

General Information

Category: Users & Groups

Operation Id: Document.GetPrincipalEmails

Parameters

Name	Type	Required	Default Value
permission	string	true	
variable name	string	true	
ignore groups	boolean	false	false

Signature

Inputs: document

Outputs: document

Get User Setting

Get user setting from given type General Information

Category: Users & Groups

Operation Id: Document.getSetting

Parameters

Name	Type	Required	Default Value
type	string	true	

Signature

Inputs: void

Outputs: document

Get User Settings

Get user settings from given category General Information

Category: Users & Groups

Operation Id: Document.getSettings

Parameters

Name	Type	Required	Default Value
category	string	true	

Signature

Inputs: void

Outputs: documents

Get Users and Groups

Fetch the users and groups that have a given permission on the input document and then set them in the context under the given key variable name. The operation returns the input document. You can later use the list of identifiers set by this operation on the context from another operation. The 'key' argument represents the variable name and the 'permission' argument the permission to check. If the 'ignore groups' argument is false then groups will be part of the result. If the 'resolve groups' argument is true then groups are recursively resolved, adding user members of these groups in place of them. Be **warned** that this may be a very consuming operation. If the 'prefix identifiers' argument is true, then user identifiers are prefixed by 'user:' and groups identifiers are prefixed by 'group:'. General Information

Category: Users & Groups

Operation Id: Document.GetUsersAndGroups

Parameters

Name	Type	Required	Default Value
permission	string	true	
variable name	string	true	
ignore groups	boolean	false	false
prefix identifiers	boolean	false	false

resolve groups	boolean	false	false
----------------	---------	-------	-------

Signature

Inputs: document

Outputs: document

Login As

Login As the given user. If no user is given a system login is performed. This is a void operations - the input will be returned back as the output. General Information

Category: Users & Groups

Operation Id: Auth.LoginAs

Parameters

Name	Type	Required	Default Value
name	string	false	

Signature

Inputs: void

Outputs: void

Logout

Perform a logout. This should be used only after using the Login As operation to restore original login. This is a void operations - the input will be returned back as the output. General Information

Category: Users & Groups

Operation Id: Auth.Logout

Parameters

Name	Type	Required	Default Value
------	------	----------	---------------

Signature

Inputs: void

Outputs: void

Push & Pop

Pop Document

Restore the last saved input document in the context input stack. This operation must be used only if a PUSH operation was previously made. Return the last *pushed* document. General Information

Category: Push & Pop

Operation Id: Document.Pop

Parameters

Name	Type	Required	Default Value
------	------	----------	---------------

Signature

Inputs: void

Outputs: document

Pop Document List

Restore the last saved input document list in the context input stack General Information

Category: Push & Pop

Operation Id: Document.PopList

Parameters

Name	Type	Required	Default Value
------	------	----------	---------------

Signature

Inputs: void

— **Outputs:** documents

Pop File

Restore the last saved input file in the context input stack. This operation must be used only if a PUSH operation was previously made. Return the last *pushed* file. General Information

Category: Push & Pop

Operation Id: Blob.Pop

Parameters

Name	Type	Required	Default Value
------	------	----------	---------------

Signature

Inputs: void

Outputs: blob

Pop File List

Restore the last saved input file list in the context input stack General Information

Category: Push & Pop

Operation Id: Blob.PopList

Parameters

Name	Type	Required	Default Value
------	------	----------	---------------

Signature

Inputs: void

Outputs: bloblist

Pull Document

Restore the first saved input document in the context input stack. This operation must be used only if a PUSH operation was previously made. Return the first *pushed* document. General Information

Category: Push & Pop

Operation Id: Document.Pull

Parameters

Name	Type	Required	Default Value
------	------	----------	---------------

Signature

Inputs: void

Outputs: document

Pull Document List

Restore the first saved input document list in the context input stack General Information

Category: Push & Pop

Operation Id: Document.PullList

Parameters

Name	Type	Required	Default Value
------	------	----------	---------------

Signature

Inputs: void

Outputs: documents

Pull File

Restore the last saved input file in the context input stack. This operation must be used only if a PUSH operation was previously made. Return the first *pushed* file. General Information

Category: Push & Pop

Operation Id: Blob.Pull

Parameters

Name	Type	Required	Default Value
------	------	----------	---------------

Signature

Inputs: void
Outputs: blob

Pull File List

Restore the first saved input file list in the context input stack General Information

Category: Push & Pop
Operation Id: Blob.PullList
Parameters

Name	Type	Required	Default Value
------	------	----------	---------------

Signature

Inputs: void
Outputs: bloblist

Push Document

Push the input document on the context stack. The document can be restored later as the input using the corresponding pop operation. Returns the input document. General Information

Category: Push & Pop
Operation Id: Document.Push
Parameters

Name	Type	Required	Default Value
------	------	----------	---------------

Signature

Inputs: document
Outputs: document

Push Document List

Push the input document list on the context stack. The document list can be restored later as the input using the corresponding pop operation. Returns the input document list. General Information

Category: Push & Pop
Operation Id: Document.PushList
Parameters

Name	Type	Required	Default Value
------	------	----------	---------------

Signature

Inputs: documents
Outputs: documents

Push File

Push the input file on the context stack. The file can be restored later as the input using the corresponding pop operation. Returns the input file. General Information

Category: Push & Pop
Operation Id: Blob.Push
Parameters

Name	Type	Required	Default Value
------	------	----------	---------------

Signature

Inputs: blob
Outputs: blob

Push File List

Push the input file list on the context stack. The file list can be restored later as the input using the corresponding pop operation. Returns the input file list. General Information

Category: Push & Pop
Operation Id: Blob.PushList
Parameters

Name	Type	Required	Default Value
------	------	----------	---------------

Signature

Inputs: bloblist
Outputs: bloblist

Execution Context

Restore Document Input

Restore the document input from a context variable given its name. Return the document. General Information

Category: Execution Context
Operation Id: Context.RestoreDocumentInput
Parameters

Name	Type	Required	Default Value
name	string	true	

Signature

Inputs: void
Outputs: document

Restore Documents Input

Restore the document list input from a context variable given its name. Return the document list. General Information

Category: Execution Context
Operation Id: Context.RestoreDocumentsInput
Parameters

Name	Type	Required	Default Value
name	string	true	

Signature

Inputs: void
Outputs: documents

Restore File Input

Restore the file input from a context variable given its name. Return the file. General Information

Category: Execution Context
Operation Id: Context.RestoreBlobInput
Parameters

Name	Type	Required	Default Value
name	string	true	

Signature

Inputs: void
Outputs: blob

Restore Files Input

Restore the file list input from a context variable given its name. Return the files. General Information

Category: Execution Context
Operation Id: Context.RestoreBlobsInput

Parameters

Name	Type	Required	Default Value
name	string	true	

Signature

Inputs: void

Outputs: bloblist

Set Context Variable

Set a context variable given a name and the value. To compute the value at runtime from the current context you should use an EL expression as the value. This operation works on any input type and return back the input as the output. General Information

Category: Execution Context

Operation Id: Context.SetVar

Parameters

Name	Type	Required	Default Value
name	string	true	
value	object	true	

Signature

Inputs: void

Outputs: void

Set Context Variable From Input

Set a context variable that points to the current input object. You must give a name for the variable. This operation works on any input type and return back the input as the output. General Information

Category: Execution Context

Operation Id: Context.SetInputAsVar

Parameters

Name	Type	Required	Default Value
name	string	true	

Signature

Inputs: void

Outputs: void

Execution Flow

Run Document Chain

Run an operation chain which is returning a document in the current context. The input for the chain to run is the current input of the operation. Return the output of the chain as a document. General Information

Category: Execution Flow

Operation Id: Context.RunDocumentOperation

Parameters

Name	Type	Required	Default Value
id	string	true	

Signature

Inputs: document, documents

Outputs: document, documents

Run Operation

Run an operation chain in the current context General Information

Category: Execution Flow
Operation Id: Context.RunOperation
Parameters

Name	Type	Required	Default Value
id	string	true	

Signature

Inputs: void
Outputs: void

Save Session

Commit any changes made by the operation on the documents. This can be used to explicitly commit changes. This operation can be executed on any type of input. The input of this operation will be preserved as the input for the next operation in the chain. General Information

Category: Execution Flow
Operation Id: Document.SaveSession
Parameters

Name	Type	Required	Default Value
------	------	----------	---------------

Signature

Inputs: void
Outputs: void

Scripting

Run Input Script

Run a script from the input blob. A blob containing script result is returned. General Information

Category: Scripting
Operation Id: Context.RunInputScript
Parameters

Name	Type	Required	Default Value
type	string	false	mvel, groovy

Signature

Inputs: blob
Outputs: blob

Run Script

Run a script which content is specified as text in the 'script' parameter General Information

Category: Scripting
Operation Id: Context.RunScript
Parameters

Name	Type	Required	Default Value
script	string	true	

Signature

Inputs: void
Outputs: void

Notification

Send E-Mail

Send an email using the input document to the specified recipients. You can use the HTML parameter to specify whether your message is in HTML format or in plain text. Also you can attach any blob on the current document to the message by using the comma separated list of xpath expressions 'files'. If your xpath points to a blob list all blobs in the list will be attached. Return back the input document(s). General Information

Category: Notification
Operation Id: Notification.SendMail
Parameters

Name	Type	Required	Default Value
from	string	true	
message	string	true	
subject	string	true	
to	stringlist	true	
HTML	boolean	false	false
files	stringlist	false	
viewId	string	false	view_documents

Signature

Inputs: document, documents
Outputs: document, documents

Send Event

Send a Nuxeo event. General Information

Category: Notification
Operation Id: Notification.SendEvent
Parameters

Name	Type	Required	Default Value
name	string	true	

Signature

Inputs: void
Outputs: void

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).

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 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 operation 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](#)

Operation 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 provide 3 operation methods with different signatures:

1. one that takes a Document and return a Document object.
2. one that takes a list of document objects and return a list of documents.
3. 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). Nuxeo 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 contextes 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` annotation.

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** Java keyword and the result of the operation will be the **null** Java object.

Contributing an Operation Chain

An operation chain is a pipe of parametrized atomic operations. This means the operation chain specify the parametrization of each operation in the chain and not only the list of operation to execute. Because of this when executing an operation chain you should only specify the chain 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 Nuxeo extension 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">
    <operation id="Context.FetchDocument"/>
    <operation id="Context.SetVar">
      <param type="string" name="name">aaname</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:@{aaname}.zip</param>
    </operation>
    <operation id="Seam.DownloadFile"/>
  </chain>
</extension>
```

This is defining a chain that will do the following:

- Fetch the current document in User Interface and set it as the input of the next operation.
- Set a context variable named "aaname" to the value of the input document title. The input document is returned as the input for the next operation.
- Get 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 get the attached blob (the blob property to use is specified using the *xpath* parameter) and return the list of blobs as the input of the next operation.
- Create 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 "aaname" context variable that was set before i the chain. Return the zip blob as the input of the next operation.
- Show the download file to download the input zip from the browser. (this is a UI operation)

Briefly this chain is getting the current document in the User Interface, extract all blobs from its direct children, zip these blobs and offer 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 parameter are either hard coded strings either 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.

You can find the complete list of the supported types in [Contributing an Operation](#).

REST API

The Automation HTTP / REST Service

The Nuxeo Automation Server module provides a REST API to execute operations on a Nuxeo Server.

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 appears 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 [REST Operation Filters](#).

Example - Getting the Automation Service.

Let say the service is bound to <http://localhost:8080/automation>. Then to get the service description you should do:

```
GET http://localhost:8080/automation
Accept: application/json+nxautomation
```

You will get response a JSON document like:

```
HTTP/1.1 200 OK
Content-Type: application/json+nxautomation
```

```
{
  "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 operation chain) is the relative path to use to execute the operation.

For example if the service URL is <http://localhost:8080/automation> and the `Blob.Attach` operation has the url `Blob.Attach` then the full URL to that operation will be: <http://localhost:8080/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/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 (using Basic Authentication) your request 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 4 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 /automation/Document.Fetch HTTP/1.1
Accept: application/json+nxentity, */*
Content-Type: application/json+nxrequest; charset=UTF-8
Authorization: Basic QWRtaW5pc3RyYXRvcjpBZGlpbmlzdHJhdG9y
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 as input a Blob

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 /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 QWRtaW5pc3RyYXRvcjpBZGlpbmlzdHJhdG9y
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 it 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:

1. `document` - a repository document
2. `documents` - a list of documents
3. `blob` - a blob (binary content usually attached to a document)
4. `blobs` - a list of blobs
5. `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:

1. `document` -> a JSON object describing the document is returned. The used Content-Type is "**application/json+nxentity**"
2. `documents` -> a JSON object describing the document list is returned. The used Content-Type is "**application/json+nxentity**"
3. `blob` -> The blob raw content is returned. The used Content-Type will be the same as the blob mime-type.
4. `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.
5. `void` -> HTTP **204** is returned. No content and no Content-Type is returned.
6. `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 return 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
- `lastModified` - 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 3 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, contains 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 contains 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 three 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 /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 QWRtaW5pc3RyYXRvcjpbZGlpbmlzdHJhdG9y
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":&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: dublicore, 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 '/' !) either the document UID.

Example:

```
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 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 chains

You can see operation chains as macro operations. These are chains of atomic operations that are registered on the server.

To extend the default operation set provided by Nuxeo, you can either write your own atomic operation, either define an operation chain through a Nuxeo XML extension.

When defining an operation chain on the server, it will become visible in the operation registry returned by the GET request o the automation service.

You must note that operation chains do not take parameters (only an input) because when defining such a chain, you also define all the parameters needed by each operation in the chain.

If you need parameterizable parameters (their value are computed at each execution), then use EL expressions as values.

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 [REST Operation Filters](#) for more details.

JavaScript clients

The automation service is ready to use with JavaScript clients (from a browser).

Browser support will be improved by working on `Multipart/form-data` encoding support to execute operations.

A GWT client will be provided as an example.

Java Clients

A Java client is in progress. More details later.

X-NXRepository

REST Operation Filters

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 to be 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 `disabled` parameter) and the second one is defining a security rule for the operation chain named `audit`.

You notice the usage of the `chain` attribute which must be set to `true` every time a binding refer to an operation 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** the 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 operation chain main that should be protected.
- **chain** - true if the name refer to an operation 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 either 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 matched the operation is not accessible if the connection is not secured.

Using the REST API - Examples

This section explain how to use automation REST calls using raw HTTP requests and response - this page is more theory than concrete examples. You can check concrete examples using different HTTP clients to here:

- [PHP Automation Client](#)
- [Using a Python client](#)
- [Using cURL](#)
- [Using Java API](#)
- [Using Nuxeo Automation Client](#)

Raw HTTP examples

Here is a complete example on using automation service. This example will do the following:

1. get the automation registry
 2. set a blob on an existing document (let say `/default-domain/workspaces/myws/file`) by forcing the server to avoid returning back the blob.
 3. get the same document with all the data inside (all the schemas)
 4. download the content of the blob we set at step 2. (and using the information available in the document retrieved at step 3.)
1. Get the automation registry

REQUEST:

```
GET /automation HTTP/1.1
Accept: application/json+nxautomation
Host: localhost:8080
```

RESPONSE:

```
HTTP/1.1 200 OK
Content-Type: application/json+nxautomation
```

```
{
  "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": [ ... ]
    },
    ... // other operation follows here
  ],
  "chains" : [
    // a list of operation chains (definition is identical to regular operations)
  ]
}
```

2. Upload a blob into a File document: "/default-domain/workspaces/myws/file"

See the **X-NXVoidOperation** header to avoid the blob being returned by the server.

REQUEST:

```
POST /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":{"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--

```

RESPONSE: 204

3. Get the document data where we uploaded the blob

(see X-NXDocumentProperties header used to specify that all the document data (schemas) should be returned)

REQUEST:

```

POST /automation/Document.Fetch HTTP/1.1
Accept: application/json+nxentity, */*
Content-Type: application/json+nxrequest; charset=UTF-8
Authorization: Basic QWRtaW5pc3RyYXRvcjpbZGlpbmlzdHJhdG9y
X-NXDocumentProperties: *
Host: localhost:8080

```

```

{"params":{"value":"/default-domain/workspaces/myws/file"},"context":{}}

```

RESPONSE:

```

HTTP/1.1 200 OK
Content-Type: application/json+nxentity
Content-Length: 1121

```

```
{
  "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
  }
}
```

4. Download the content of the blob we set at step 2.

You notice in the last result that the documents contains our blob and the "data" property points to a relative URL that can be used to download the blob content.

Let's download it:

REQUEST:

```
GET /automation/files/96bfb9cb-a13d-48a2-9bbd-9341fcf24801?path=%2Fcontent HTTP/1.1
Authorization: Basic QWRtaW5pc3RyYXRvcjpbZGlpbmlzdHJhdG9y
Host: localhost:8080
```

RESPONSE:

```
HTTP/1.1 200 OK
Content-Type: image/jpeg
Content-Length: 290096
Content-Disposition: attachment; filename=test.jpg
```

```
[the blob raw data here]
```

PHP Automation Client

A php automation client is made available on github (<https://github.com/nuxeo/nuxeo-automation-php-client>). 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 (<https://github.com/nuxeo/nuxeo-automation-php-client/tree/master/NuxeoAutomationClient>) and some sample use cases here (<https://github.com/nuxeo/nuxeo-automation-php-client/tree/master/tests>).

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.

```
$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 sendRequest() 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 blob, 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 it out here:

<http://blogs.nuxeo.com/fermigier/2010/08/a-sample-python-library-for-the-nuxeo-content-automation-jsonrpc-api.html>

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 screen cast that demonstrate how to deploy a custom server side operation developed in Java using the Nuxeo IDE and a client python script that calls it: <http://dev.blogs.nuxeo.com/2012/01/exploring-nuxeo-apis-content-automation.html>

Here is the same example as the [cURL example](#), this time 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
- 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
```

Using cURL

In this examples we are using the unix `curl` command line tool 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.

```
curl -H 'Content-Type:application/json+nxrequest' -X POST \
  -d '{"params":{"query":"SELECT * FROM Document"},"context":{}}' \
  -u Administrator:Administrator
http://localhost:8080/nuxeo/site/automation/Document.Query
```

You can simplify the command by removing the `"context"` attribute (since it is empty). And by removing quotes from JSON keys that don't need to be quoted:

```
curl -H 'Content-Type:application/json+nxrequest' -X POST \
  -d '{"params:{query:"SELECT * FROM Document"}}' \
  -u Administrator:Administrator
http://localhost:8080/nuxeo/site/automation/Document.Query
```

Using Java API

This example demonstrate the usage of the `java.net` API to invoke remote operations through Automation REST API.

We will do the same as the [cURL example](#) - executing the query: `SELECT * FROM Document`.

```

public static void main(String[] args) throws Exception {
    String username = "Administrator";
    String password = "Administrator";
    OutputStreamWriter wr = null;
    InputStream in = null;
    try {
        URL url = new
URL("http://localhost:8080/automation/nuxeo/site/Document.Query");
        URLConnection conn = url.openConnection();
        conn.setDoOutput(true);
        String auth = username + ":" + password;
        String encodedAuth = new sun.misc.BASE64Encoder().encode(auth.getBytes());
        conn.setRequestProperty("Authorization", "Basic " + encodedAuth);
        conn.setRequestProperty("Content-Type",
            "application/json+nxrequest");
        conn.setRequestProperty("Accept", "application/json+nxentity");
        wr = new OutputStreamWriter(conn.getOutputStream());
        wr.write("{params:{query:\"SELECT * FROM Document\"}}");
        wr.flush();

        // Get the response
        in = conn.getInputStream();
        StringBuffer buf = new StringBuffer();
        byte[] cbuf = new byte[1024 * 64];
        int r = in.read(cbuf);
        while (r > -1) {
            if (r > 0) {
                buf.append(new String(cbuf, 0, r));
            }
            r = in.read(cbuf);
        }
        System.out.println(buf.toString());
    } finally {
        if (wr != null) {
            wr.close();
        }
        if (in != null) {
            in.close();
        }
    }
}

```



You may notice the usage of `sun.misc.BASE64Encoder` - its usage is not recommended since it is a SUN internal utility class.

Of course you may create a specific API over the `java.net` to expose the REST automation mechanisms using a simplified API.

Instead of writing your own Java client API you should check the [Nuxeo Automation Client](#) since it is already providing this.

Using Nuxeo Automation Client

As we've seen in previous examples you can invoke a remote automation server using any language or HTTP tool you want. Nuxeo already provides a high level client implementation for Java programmers. Using this client API simplifies your task since it handles all the protocol level details.

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>
```

For a direct download, see <https://maven.nuxeo.org/>.



This documentation applies for nuxeo-automation-client versions greater or equal to 5.4.

The client library depends on:

- net.sf.json-lib:json-lib, net.sf.ezmorph:ezmorph - for JSON support
- org.apache.httpcomponents:httpcore, org.apache.httpcomponents:httpclient - for HTTP support
- javax.mail - for multipart content support

Query example

Here is the same example as the one in [Java API example](#) - execute the "SELECT * FROM Document" query against a remote automation server:

```
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 using the automation client is much easier than writing yourself all the protocol details as in the previous example [Using Java API](#)

You can see the code above has 3 distinctive parts:

1. Opening a connection.
2. Invoking remote operations
3. Destroying the client.

So 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 are 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.

So you create a new session by calling:

```
Session session = client.getSession("Administrator", "Administrator");
```

This will authenticates 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();
```

Using a session you can now invoke remote operations. To create a new invocation request you should pass in the right operation or chain ID:

```
OperationRequest request = session.newRequest("Document.Query");
```

and then populate the request with all the required arguments:

```
request.set("query", "SELECT * FROM Document");
```

You can see in our example you have to specify only the `query` argument. If you have more arguments you call in turn the `set` method for each of these arguments. If you need to specify execution context parameters you can use `request.setContextProperty` method. The same, if you need to specify custom HTTP headers you can use the `request.setHeader` method.

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.

Executing a request will either thrown 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();
```

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 creating one session per login. A session is thread safe.


Blob upload example

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. Finally we will download back this file.

First get the root document and create a new File document at location /myfile


```
// 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.

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.

Now get the the file document where the blob was uploaded. Then retrieve the blob remote URL from the document metadata. We can use this URL to download the blob.

```
// 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.

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();
```

The complete example

Here is the complete code of the example. For more examples, see the unit tests in `nuxeo-automation-server` project.

```
// 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();

// Upload The file
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();

// 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");

// 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();
```

OpenSocial, OAuth and Nuxeo EP

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/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 independant 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 WebServices

OpenSocial defines a set of Rest WebServices 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 : manage relationship between users. (each user has the ability to manage his own groups : friends, coworkers, ...)
- Activity service : track 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.

Manage authentication and security of data accesses

In OpenSocial there are usually 3 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 2 types of user identify:

- the viewer : the user in front of the browser
- the owner : the user owning the page that contains the gadget

If you take the example of a gadget that display inside Nuxeo your emails from Gmail :

- 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 directly Gmail (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 in OpenSocial contexte there are 3 constraints :

- 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](#).

Nuxeo EP server can be used both as an OAuth Service consumer or as an OAuth Service provider.

You can find more details about OAuth and it's integration in Nuxeo later in this page.

What can it be used for

Dashboard / Portal use case

The most direct use cases is to use OpenSocial gadgets to adress Dashboard / Portal use case.

This is currently the default usage of OpenSocial within Nuxeo CAP / DM.

For that, we have created some simple and extensible gadgets :

- My recent documents
- My Workspaces
- My Tasks
- ...

The user can then use the Nuxeo's dashboard to have a quick overview of it's activity inside Nuxeo platform.

Using OpenSocial gadgets to do so is interesting because :

Let users control their Dashboards.

Depending on the security policy, the user is able to customize it's dashboard

- add / remove gadgets
- change gadgets configurations

Leverage 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 ...
- ...

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.

Manage user identity

Because OpenSocial relies on OAuth you can have a narrow controle 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 2 applications.

For example, you can :

- display some informations coming from SAP next to the Invoice Document in Nuxeo
- display links to related Nuxeo Documents from within a confluence wiki page
- display specification documents (Stored in Nuxeo) from within the Jira Issue

More generally, if you have Enterprise wide Rest Webservice (like Contacts managements, Calendar management ...) you can expose them via OpenSocial gadgets so that these services are accessible to users of all OpenSocial aware applications.

Social network management

Because OpenSocial standardize 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 makes sense :

- manage communities
- manage activity
- manage user skills
- ...

OAuth in Nuxeo EP

Starting with 5.4.1, Nuxeo EP 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 an 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's 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 consumer is hosted by your Nuxeo instance, you will need to do some configuration to define how Nuxeo must access this external service.

For more details about OAuth support in Nuxeo, please see the [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](#) :

- provide a solution to easily have custom REST Apis without having to write code (Using Nuxeo Studio)
- common access point for all REST services (All Operation and Chains can be called the same way)
- native support for JSON marshaling (means easy integration in the Gadget JS)
- 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 these services via an OpenSocial gadget.

Gadgets in Nuxeo EP

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 includes :

- 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
- list the required features of the gadget
- contains the translation resources
- contains the authentication informations
- 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 add more flexibility and power :

Better reusability

Most of you gadgets will share some common content.

For Javascripts 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-gaget bundle (in skin/resources/scripts or skin/resources/css)

Better context management

When generating the GadgetSpec 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 gadget is inside Nuxeo
 - you want to use 3 Legged authentication when gadget is hosted inside an external application

In order to address these problem 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 gadget will be renderd inside Nuxeo or in an external application
jsContext	Automatically generated 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 gadgetSpec url will also be available at the root of the freemarker context.

Integrated internationalization

GadgetSpec 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
- 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 a full featured gadgets base on Automation API is very simple.

```
<?xml version="1.0" encoding="UTF-8"?>
<Module>
  <ModulePrefs title="Nuxeo Automation"
    description="Simple gadget that uses Nuxeo REST Automation API "
    author="tdeprat" author_email="tdeprat@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 through 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:

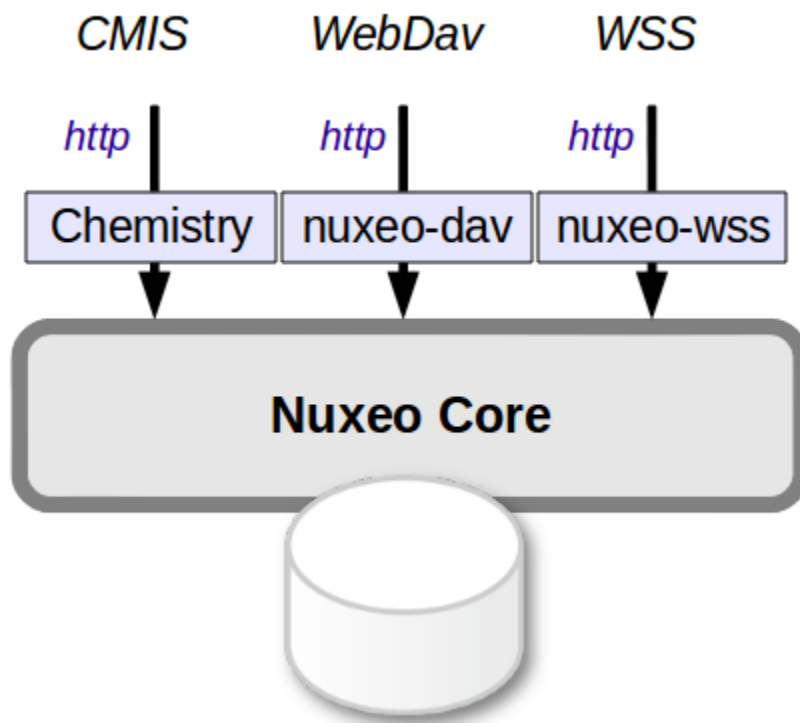
- `dndValidation`: 'true' if the container should wait the validation of the Drag'n Drop before doing the actual move, 'false' otherwise. If the parameter is not present, default to 'false'.
- `showPreferences`: 'true' if the gadget preferences need to be displayed after adding a gadget, 'false' otherwise. 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.

Repository access

Nuxeo EP currently supports 3 protocols to expose the document repository:

- [CMIS](#),
- [WebDAV](#),
- the [Windows Sharepoint Services](#) protocol (only the sub set used by MS Office and MS Explorer).

Compared to the Service APIs, these protocols won't give you access to services but directly to the content.



Depending on the chosen protocol, your access to Nuxeo Content will be more or less powerful:

- CMIS gives you access to a big subset of the data managed by Nuxeo,
- WebDav and WSS mainly map Nuxeo's content as a file-system (with all the associated limitations).

These protocols may be useful in several use cases:

- Desktop integration,
- Allow a portal or a WCM solution to access Nuxeo's content.

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 is [CMIS 1.0 Committee Specification 01](#).

Outline of this document:

- [Status](#)
- [Versions](#)
- [Online demo](#)
- [Downloads](#)
 - [Nuxeo DM release](#)
 - [Nuxeo Core Server release](#)
- [Usage](#)
 - [AtomPub](#)
 - [SOAP](#)
 - [CMIS Clients](#)
 - [From Java code within a Nuxeo component](#)
- [Documentation](#)
- [Capabilities](#)
- [Model mapping](#)
- [Nuxeo specific System Properties](#)
 - [Since Nuxeo 5.4.2](#)
 - [Since Nuxeo 5.5](#)
- [Source Code](#)
- [Additional Resources](#)

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 the full CMIS 1.0 starting with Nuxeo EP 5.4. (A previous version of the connector based on an earlier version of Apache Chemistry backend has been included by default in Nuxeo DM since Nuxeo EP 5.3.1.)

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

Nuxeo DM release

The Nuxeo DM 5.4 release includes the CMIS connector by default. You can get it from here: <http://nuxeo.com/en/downloads/download-dm-form>

Nuxeo Core Server release

We're now also providing an extra lightweight packaging of a Nuxeo repository that only includes CMIS access to the repository (no web UI at all).

You can grab it here: <http://www.nuxeo.com/en/downloads/download-ep-form>

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>.

AtomPub

You can use a CMIS 1.0 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:title+
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
```



The above AtomPub URLs are correct for Nuxeo EP 5.4, but note that previous versions of Nuxeo (5.3.1 and 5.3.2) used different URLs (`site/cmis` instead of `atom/cmis/default`).



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.x
sd">
      <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 1.0 are available.

The best one is the [CMIS Workbench](#), part of OpenCMIS.

Other older clients are available:

- <http://code.google.com/p/cmispaces/> is an Adobe AIR client (you will need to change the cmisUrl parameter in the file CMISpacesCon fig.xml to point it to your Nuxeo server),
- <http://code.google.com/p/cmis-explorer/> is another Adobe AIR client,

A command-line shell is also available:

- [CMIS Shell](#) (direct 2MB download of most recent version)

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 [howtos](#).

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 1.0 HTML version](#) or download [CMIS 1.0 \(PDF\)](#) (1.3 MB).

Capabilities

The Nuxeo OpenCMIS connector implements the following capabilities from the specification (in Nuxeo 5.4.2):

Get descendants supported	Yes
Get folder tree supported	Yes
Unfiling supported	No
Multifiling supported	No

Version-specific filing supported	No
Query	Both combined
Joins	Inner and outer
All versions searchable	Yes
PWC searchable	Yes
PWC updatable	Yes
Content stream updates	PWC only
Renditions	Read
Changes	Object IDs only
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.
- Proxy documents are not visible in CMIS (as of 5.4.1).
- Secondary content streams are not visible as renditions (as of 5.4.1).

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, Nuxeo EP 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 lifecycle 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).

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 [Document Management 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/hxserver/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|cadaver
|BitKinex).*</header>
      </headers>

      <replacementChain>
        <plugin>DIGEST_AUTH</plugin>
        <plugin>WEBDAV_BASIC_AUTH</plugin>
      </replacementChain>
    </specificAuthenticationChain>
  </extension>

</component>
```

WSS before Nuxeo 5.4.2

This page describes how WSS has been used in Nuxeo DM, since version 5.4.2 this has been replaced by the [WebDAV](#) support.

Windows SharePoint Services is a set of protocol that is used by Microsoft applications (mainly the Windows Explorer and MS Office) to access content stored inside a SharePoint Team Services site.

Target Scope of WSS implementation in Nuxeo

The target scope of this implementation is:

- Browse / Open / Save documents from MS Office 2003 / 2007
- Browse / Open / Save documents from Explorer if MS Office 2003 / 2007 is installed
- Display some meta-data informations in MS Office 2003 / 2007

Understanding the beast

WSS is not a complicated protocol, it's an aggregation of several protocols and different technologies. In a lot of cases, finding the needed information was a complicated task.

Not because there is no documentation but because there is too much documents and that information is largely spread across several pieces.

Some may suspected this is done "on purpose" 😊

The documentation mixes all aspects of SharePoint technologies:

- communicating from FrontPage to administer web sites

- communication between several SharePoint servers
- customizing SharePoint
- automating SharePoint
- ...

The point is that the really interesting part is what WSS clients (mainly Explorer and MSOffice) really use.

And on these aspects it's not that easy to find precise informations about:

- error handling for a given method
- meaning of a meta-data in weerm-rpc
- how the client interpret and use the results
- ...

Development was started based on the documentation, but we quickly realized that:

- some points were not working as inside the doc
- some parts of the protocol are not useful / needed

So, quickly we fall back to using ngrep to dump dialogs between a Windows VM and a SharePoint server. Not very fancy, but efficient.

Outlines

Here is basically a quick overview of the WSS logic:

Negotiation phase

Client ask the server for capability and supported version.

At this point, the main "browsing protocol" is chosen: WebDav or FrontPage RPC.

FrontPage for browsing

Since WebDav for WSS seems to be dropped for new versions of windows clients, we choose FrontPage RPC. This is basically a simple HTTP protocol:

- method name is sent as a request parameter
- parameters are encoded (as QueryString, or as veerm encoding in the body)

HTML and JS for MSOffice WebView dialog

Open and Save dialogs for MSOffice are generated from HTML and JS is returned by the server.

There is some mystery on what part of the HTML and JS is actually used by MSOffice to make the dialog actually work.

For example the *Open* dialog worked very simply (based on a ngrep capture), but the Web Dialog for *Save As* never worked for a reason that is still unknown.

WebServices for Office Companion

The MS Office companion tool bar displays information about the data related to the workspace of the current open document:

- users
- tasks
- links
- other documents

This panel is using some WebServices to fetch data from the server.

Strangely, one of the panel tabs always remains empty, even when communicating with a "real SharePoint server".

Please note that this is a simple view, some clients (like the WSS Mac client) do not exactly behave this way...

Implementation choices

Make a generic handler

A big part of the work is about filters, handlers and marshalers: nothing fancy or fun.

We wanted to have this code as generic as possible in order to be reusable and not bound to the Nuxeo framework.

For most protocol handling part, that mainly depends on Servlet API, this is not really an issue.

It was more painful for managing configuration and pluggability: Extension Points were missing 😊

Root binding

Surprisingly, WSS protocol needs to communicate with the root of the server.

Some part of the negotiation protocol can lead to think there was a way to limit root calls to only one, but it did not work as expected.

So, some service calls have to be handled in the root servlet context.

This is a design constraint, because you must have code in both context, and in some cases, it has to be the same code (same call on root and on Nuxeo context path).

General design

Generic protocol handler

The generic handler is mainly composed of:

- a main filter
- handlers for each protocol (HTTP, FP-RPC endpoints, WebService ...)
 - handling marshaling and unmarshaling
- a proxy system (to handle and forward calls made to the root)

SPI

The generic handler calls a simple SPI to do the actual work:

- listing content
- checkout / uncheckout
- get meta-data on a document
- download / upload a file

Test env

For testing purpose, a dummy "in memory backend" is provided.
It allows :

- direct live testing of the generic handler
- unit testing of the handler (using ngrep capture as test input)

Use of freemarker

Most WSS response handling is about HTML formatting.
For all this work FreeMarker was used since it is already bundled with Nuxeo.

Dummy WebService implementation

When testing WSS it appears that :

- very few calls are indeed needed
- most of them return a XML results encoded in a SOAP envelop

So, current implementation does not include a real WebService stack, and simply uses FreeMarker.

Using WSS in Nuxeo

Nuxeo backend for WSS

Setup

In order to test WSS against Nuxeo, you have two configurations to make:

- setup the root filter and set it in proxy mode
- deploy the nuxeo-wss-backend

Step 1 can be done in the root web.xml by just adding the org.nuxeo.wss.rootFilter param:

```
<filter>
  <display-name>WSS Filter</display-name>
  <filter-name>WSSFilter</filter-name>
  <filter-class>org.nuxeo.wss.servlet.WSSFilter</filter-class>
  <init-param>
    <param-name>org.nuxeo.wss.rootFilter</param-name>
    <param-value>/nuxeo</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>WSSFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

NB: the wss-handler jar needs to stay in \$JBoss/server/default/lib
(no need to deploy it inside Nuxeo)

Step 2

Deploy nuxeo-platform-wss-backend in nuxeo.ear/plugins/

Accessing

You should be able to create a WebFolder or to open from MSOffice using:

- <http://server:8080/nuxeo/> : to access directly Nuxeo
- <http://server:8080/> : to access directly server root

Limitations of the model

WSS model is closer to a filesystem than to an ECM System.

Basically:

- a document is a file
- document title (display name) = name (URL part)
- checkout system is very simple

In a lot of cases, Nuxeo DM offers much richer features, but they won't be accessible via WSS because the model does not permit that.

Furthermore, only very basic features of WSS are accessible via MS Office and Explorer: the simple filesystem operations.

Known bugs

- since Nuxeo separates title and names and WSS does not, renaming from WSS can sometimes not work as expected
- delete from Windows Explorer displays an error but works
(and response from Nuxeo server seems OK, but client does not like it...)

Using Generic Handler for your application

You can use Nuxeo WSS Generic Handler to provide WSS access to your application.

For that you should:

- implement SPI
(cf package org.nuxeo.wss.spi)
- override templates if needed
(templates defined in the backend override the default one)
- configure the filter to use your backend

Packaging

In the following pages, we will see how to package your customization to make them available in your Nuxeo application.

- [Packaging from sources](#)
- [Packaging a Nuxeo plugin](#)

Packaging from sources

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`.

Packaging a Nuxeo plugin

Before starting to write your first Nuxeo Plugin you need to understand how the plugin code should be packaged as a Nuxeo Bundle, how it may contribute resources (as static files or configuration) to the application and how it may contribute extensions to other components in the framework. All this wiring is done at application startup. By understanding how the application starts and plugins are started you can control how your components will be wired with other components in the application.

Overview

Let's talk first about some concepts used when developing Nuxeo Plugins.

- A **bundle** is a Java archive (JAR) packaged so that it works inside a Nuxeo Application.
- A **plugin** is a functional module that can be plugged into a Nuxeo Application to provide some new functionality.
- A **library** is a third party Java library required by some bundles to work.



A plugin may come packaged as a single or as several bundles plus some optional third party libraries.

- A **component** is a piece of software defined by a **bundle** used as an entry point by other components that wants 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.
- A **service** is a Java interface exported by a **component** to the outside world. A component is also providing an implementation for the service it exports but it is discouraged to access the implementation class from outside a component. Components **must** be accessed using the interfaces they provide and not through real implementation classes.



A component may export zero or more services, extension points or contributions to other extension points.

When designing a plug-in you should clearly separate the plugin logic (i.e. types of logic it provides) and package these logical units as separate bundles. Also, we recommend to split the public API from the implementation and package them in two different bundles.

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 - apart 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.

Nuxeo Core

The Nuxeo Core is a set of plugins that provides the Nuxeo Content Repository and some extra services closely related to the repository.

If you want to build an embedded repository **Nuxeo Runtime + Nuxeo Core** plugins are enough. You may also consider to add the CMIS plugin for remote access to the repository.

Nuxeo Services

This the third layer of plugins that includes the vast majority of Nuxeo plugins and which provide high level functionalities over the content

repository.

Plugins in this layer require (almost always) the Nuxeo Core layer.

Nuxeo UI

For the UI part you currently have multiple choices:

- either use the default Nuxeo UI infrastructure based on Seam + JSF.
- either use WebEngine.
- either use GWT (through WebEngine).
- either use a composite UI made of all of these technologies together.

You can also install optional plugins (e.g. **add-ons**) not installed by default in a Nuxeo Application.

You can easily install add-ons from Nuxeo Marketplace, or create add-ons using Nuxeo Studio.

Thus, Nuxeo provide a modular framework to create content based applications. The difficulty to create custom applications comes from the modularity itself. Almost any Nuxeo plugin (i.e. apart Nuxeo Runtime) has dependencies on other plugins. When building a custom application you should make sure you install all dependencies you need.

In next sections we will discuss more about how to package your bundles and how wiring is done at application startup.

Nuxeo Bundles

A Nuxeo Bundle is a regular Java JAR with an OSGi Manifest file. Although Nuxeo Runtime is not an OSGi framework, it is able to load OSGi bundles and wire them into the application.

But what is OSGi?

For more details about OSGi check the OSGi specifications. I will only explain in short terms some OSGi concepts:

Roughly an OSGi framework provides a:

- a lifecycle model for Java modules
- a service model.

When an OSGi framework starts it will try to load all bundles installed in the system and when all dependencies of a bundle are resolved it is starting it. Starting a bundle means invoking the declared Bundle Activator if any is declared in the Manifest file.

This way each bundle that register 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.

More 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 lifecycle mechanism helps bundles to 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.

As I said Nuxeo Runtime is not a fully OSGi framework, so not all OSGi features are supported. Here is a list of

But anyway Nuxeo use te OSGi model in loading bundles and also requires some OSGi headers in the manifest. In fact the only OSGi header required by now is Bundle-SymbolicName. Apart this you can use a regular manifest file.



Note

Even if Nuxeo Runtime launcher is not fully OSGi it is recommended to use valid OSGi Manifests. Because the core part of Nuxeo can already run on Equinox - you should provide a valid OSGi Manifest if you want to deploy your bundles in an Equinox based distribution of Nuxeo. But in the default distribution you only need to define a Bundle-SymbolicName

There are 2 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. (this will change in future)

Then what's good for the OSGi Manifest?

- Using the manifest you can define an unique name for your bundle (e.g. 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 - so for now you don't need to care about dependencies - but see the note below.



Important

In future Nuxeo is planing to replace its own launcher with a real OSGi framework - like Virgo - so your are encouraged to use real OSGi Manifests to avoid refactoring later.

So, to resume: in **Nuxeo** you don't have to declare dependencies or other OSGi Manifest headers than **Bundle-SymbolicName**. You can optionally declare a **Bundle-Activator** to receive notifications when your bundle is started or stopped.

In Nuxeo, 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 its not respecting 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
```



Backward Compatibility

Before the 5.4 release, Nuxeo used the OSGi dependency headers like **Bundle-Require** but this was removed in 5.4 release because the usage was not conform to OSGi specifications. So now you don't need to specify neither **Bundle-Require**, **Nuxeo-Require** nor **Nuxeo-RequiredBy** in your Manifest files. But anyway see the not above - you are encouraged to use real OSGi Manifests.

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**. In fact it is more a post build process that is triggered at each startup.

Why are we doing this?

Shouldn't we run this preprocessing at build-time (e.g. in the maven build cycle) rather than at runtime?

Nuxeo Components

Nuxeo Extension Points

Nuxeo Services

Nuxeo Properties and Configuration Extensions

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) into a URL, and decoding of this URL into a document view.

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. The 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/release-5.5/tiddoc/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;
}
```

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-ald9-aldcd74cc709/blobholder:0/mainfile.jpg
http://localhost:8080/nuxeo/nxfile/default/47ad14f2-c7a6-4a3f-8e4b-6c2cf1458f5a/files: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"
```

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 now 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



Depending on the DropZone you select, the import action will be different:

- ContentView: 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
- mainBlob: attach a file to the document
- otherBlobs: 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 operation 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 red 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,
- has the 'dropzone' CSS class,
- has a context attribute.

Drop zone declaration

```
<div id="myDropZone" class="dropzone" context="myDropZone"> ... </div>
```

Associating Operation Chains

Each dropzone context is associated with a set of Content Automation Operations or Operation 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>
</action>
```

Where:

- id is the Operation or the Operation Chain identifier (for Operation 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 Operation 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 Operation chain you may need to access Seam Context.

For that, new Operation where 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 operation 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.

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:

- [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 operation chains](#)
- [How to remove the Language Selector](#)

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	<code>org.nuxeo.cookbook</code>
artifactId	<code>bareproject</code>

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 il8n 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 deceptive, 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 is formed like that: artifactId-1.0-SNAPSHOT.jar.
- 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".
 - 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).

- 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 "[CORG:...]").

```
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!

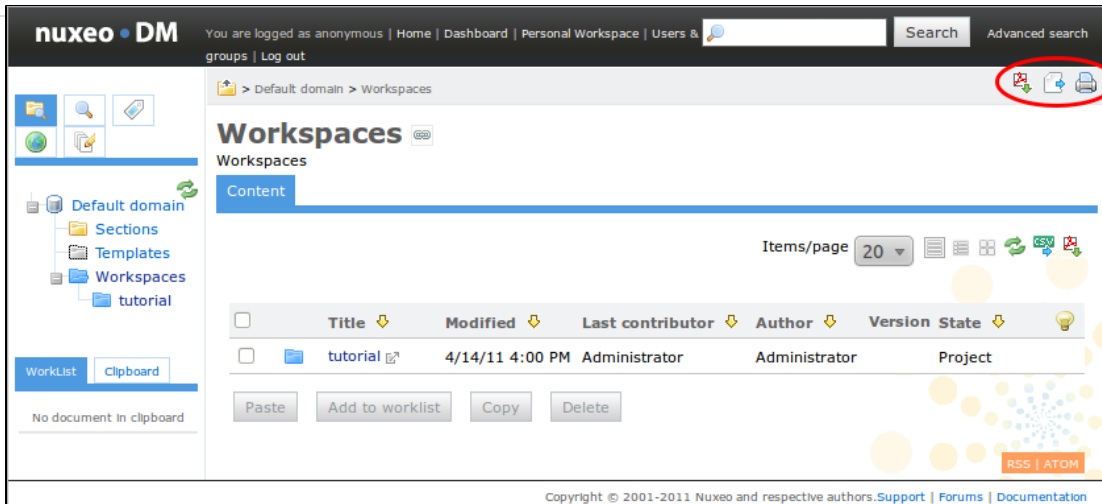
As said in the beginning of this recipe, if you have unexpected errors or Nuxeo Application behavior don't forget to check the [FAQ](#) and don't forget to leave us a comment about this recipe or about the cookbook!

How-to implement an Action

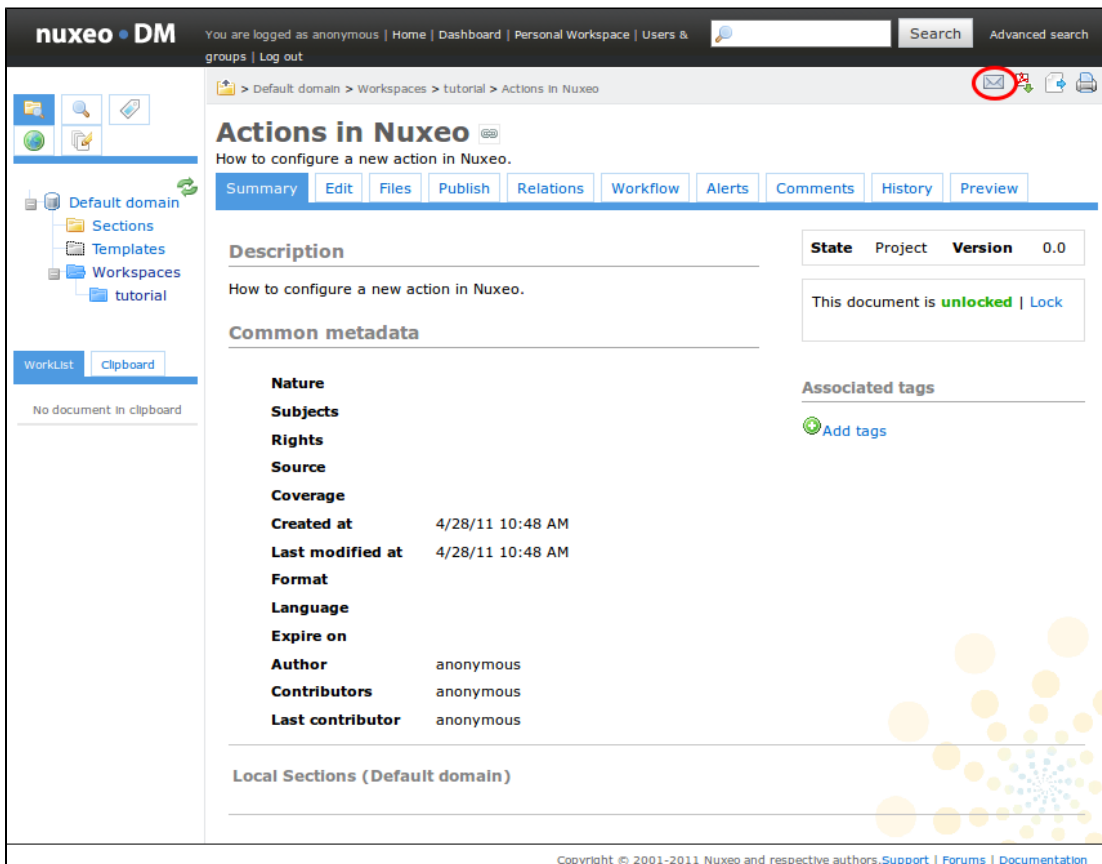
This recipe shows you how to configure an action in a bundle for a Nuxeo application.

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.

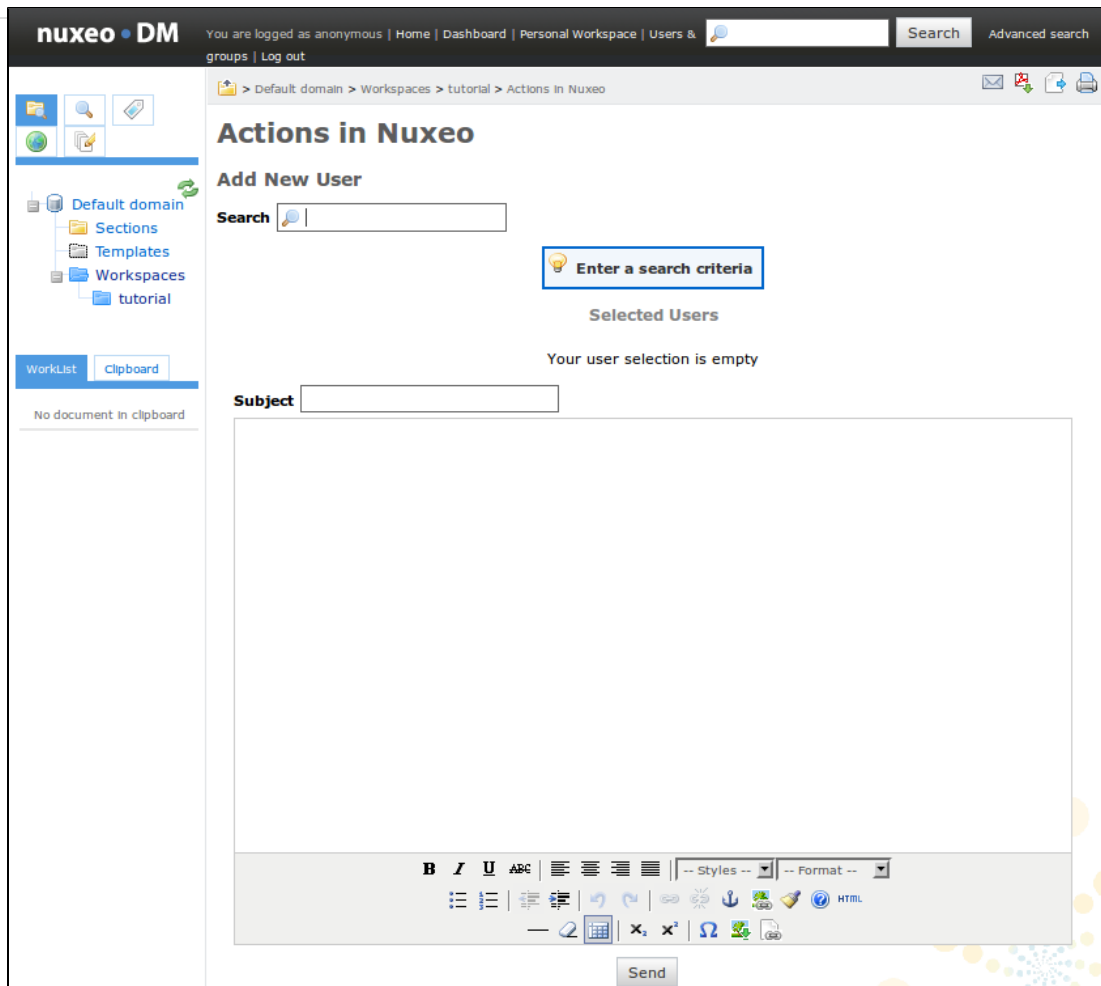
To keep things simple and focused on the action configuration, this recipe is based on the action called "send_email". The "send_email" action is triggered by an icon displayed in the "Context Tool area" (i.e. the red ellipse area).



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](#).



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 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](#).

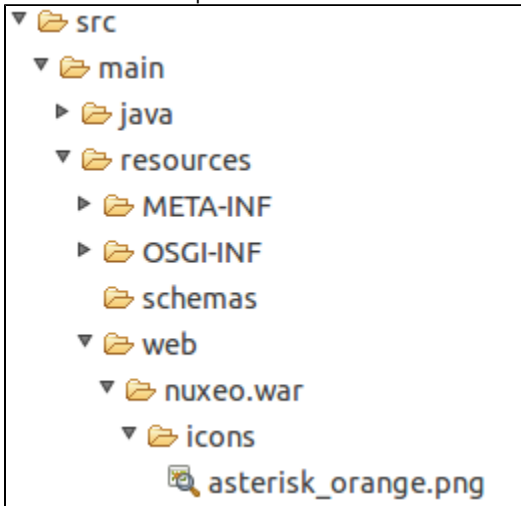
The very first step to implement an action is to create a new project, here called "nuxeo-action-project" (see the "[How-to create an empty bundle](#)" recipe for details on the project creation).

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>
```

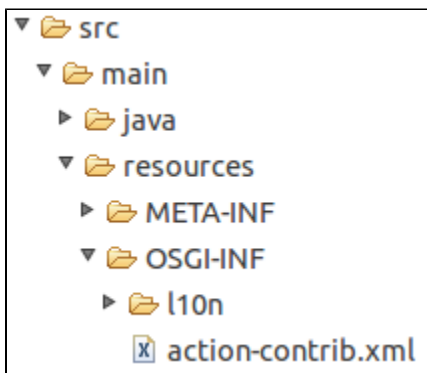


Resources

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 [User actions categories](#)). 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.

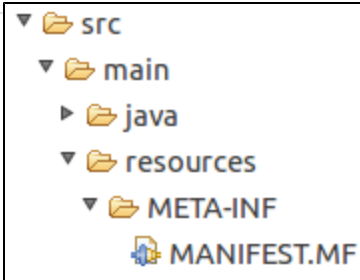
Here, the filter defines that the document should be displayed only if it's **not** decorated with the facet "folderish" (i.e. is not "folder" like). Then, the icon is displayed.

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
```



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.

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.

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 is 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).

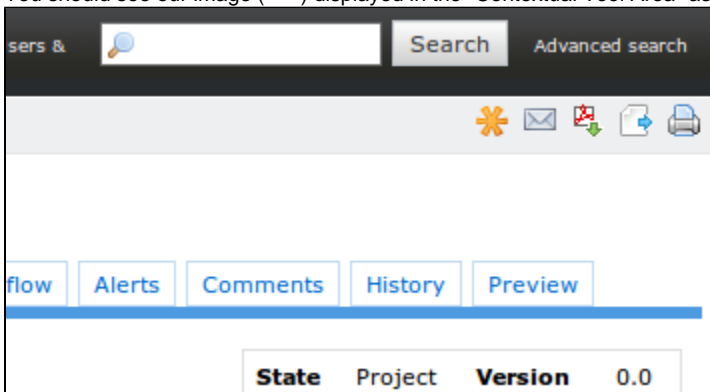
4. Check that your bundle is correctly deployed: check if its SymbolicName (as configured in the `"/src/main/resources/META-INF"`) appears in the logs. The logs are displayed:
 - in the console if you started your server using the `./nuxeoctl console`
 - in the file "server.log" located in the "log" 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:".

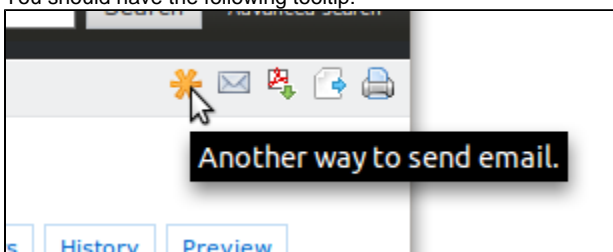
Now you know that your bundle is correctly deployed and installed. You also need to check that it works.

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.

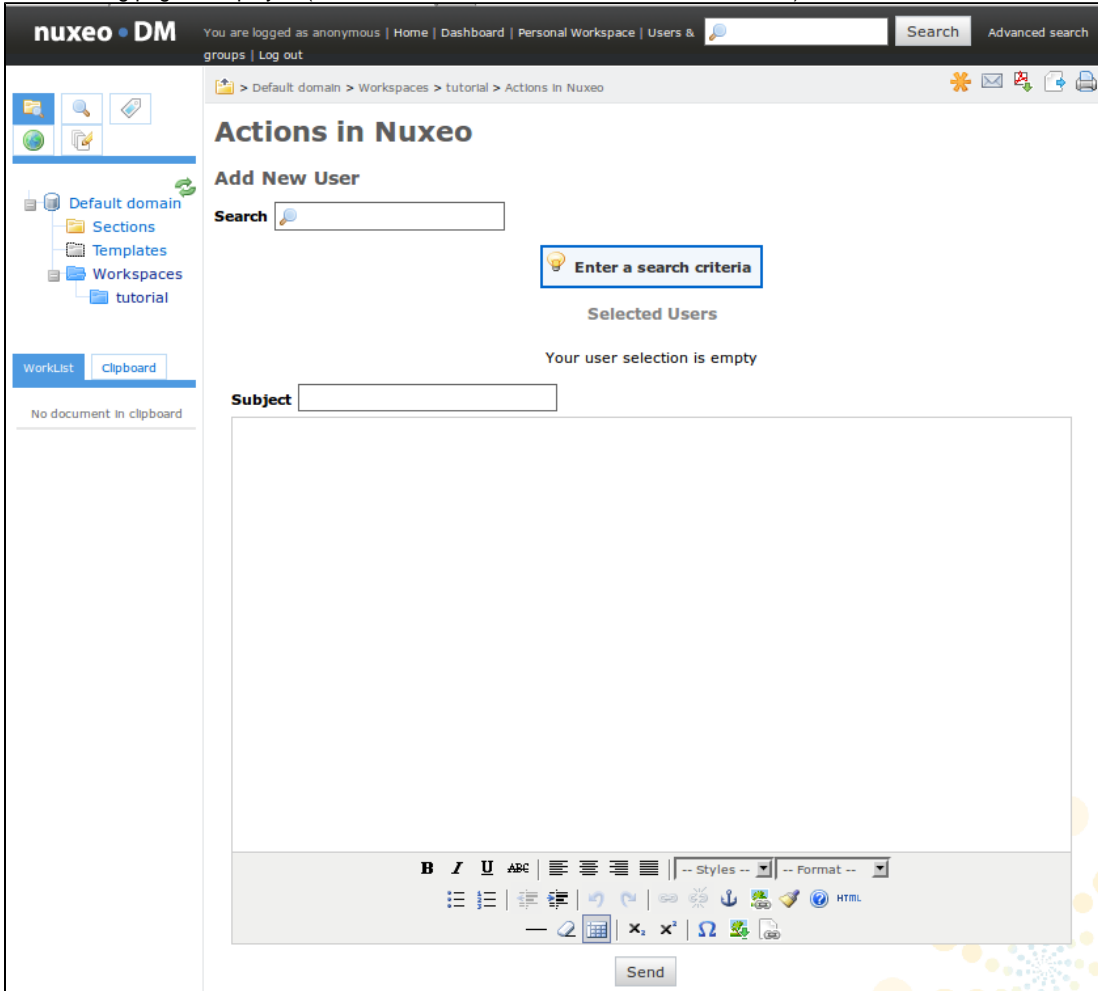


5. Move the pointer over the icon.
- You should have the following tooltip:



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

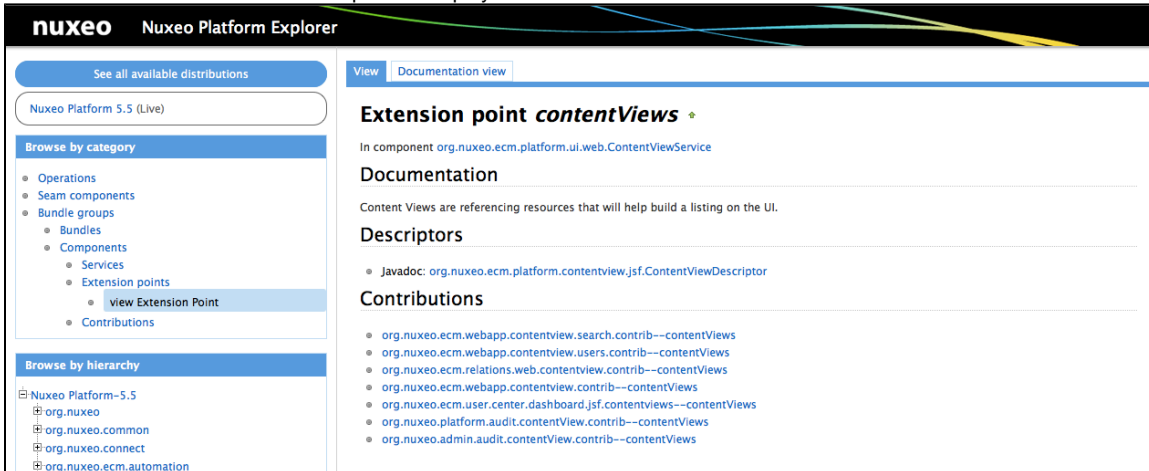
- [Find the extension point where to contribute](#)
- [Create your contribution](#)
- [Declare your contribution into your bundle](#)
- [Override the Nuxeo default configuration](#)

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.

- In the **Browse by category** panel, click on **Bundle groups > Components > Extension points**.
- In the **Extension Point** column, click on the extension point you're interested in.
The documentation of this extension point is displayed.



- Then, if you click on any link in the **Contributions** section, you will see all the default contributions implemented into your Nuxeo instance.
For instance for the Nuxeo DM 5.5 module, there are [225 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](#).

- Create a file `myproject-servicewhereIcontribute-contribution.xml` into the directory `src/main/resources/OSGI-INF/` of your project.
- 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>
```

- 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.

- 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.FilesS
&nbsp;systemFetcher.xml,OSGI-INF/extensions/com.mycomapny.test.FillIDDocumen
&nbsp;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.



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.

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 configure document types, actions and operation chains



Alternative title

A tutorial about the hard way - configure the platform via XML only.

The goal of this tutorial is to give you a fast introduction into the most important steps if you want to customize and/or test Nuxeo in process oriented environments.

The tutorial consists of:

- a description of the use case,
- a description of the filing (structure) plan,
- the real tutorial with integrated tests of your understanding.

The tutorial includes a JAR archive with all necessary stuff. Even the solutions are integrated. So you can check the solutions if you have problems 😊

File	Modified
> filing_structure_plan.jar This is the complete example.	Jun 07, 2012 by Michael Bell

Drag and drop to upload or [browse for files](#)



TODO

1. Replace Makefile with a Maven configuration.

On this page

- Introduction
 - Use Case
 - Filing (structure) plan
 - META-INF/MANIFEST
 - Makefile
- Definition of a new folder type
 - Structural configuration
 - User interface
 - Actions
 - Training lesson
- Definition of a new file type
 - Structural configuration
 - User interface
 - Training lesson
 - Actions
 - Training lesson
- Operation chains
 - Basic stuff
 - Dynamic names / MVEL
 - Training lessons
- Versioning
 - Training lesson
- Management of states (life cycle)
 - Training lesson
- Extension of meta data
 - Structural configuration
 - User interface
 - Training lesson

Introduction

Use Case

The use case is the management of the parking permissions at area 51. So this is a quite common task for the public service.

Filing (structure) plan

The general folder structure is as follows:

File system structure

```
process_name/
  description
  requests/
    request-1/
      customer_form
      approved
      rejected
```

If you work at a real public administration then you know that `process_name` would be of course a number or a cryptical combination of characters and `request-1` would be a real serial number with the `process_name` in front (e.g. 10.13.1, 10.13.2, ...).

Nevertheless this is an example for everybody and readability is preferred over correctness.

META-INF/MANIFEST

The MANIFEST is the configuration file of the bundle. It must be at the first position in the JAR archive. The explanation is a historical one and quite simple. Just imagine tapes and not random access memory. Now do you want to rewind the whole tape to know what is stored on it and where?

META-INF/MANIFEST

```
Manifest-Version: 1.0
Require-Bundle: org.nuxeo.ecm.core.api,
Bundle-Vendor: Nuxeo Community
Bundle-Category: runtime
Bundle-Localization: plugin
Bundle-Name: Nuxeo Example Public Administration Process
Created-By: 1.6.0_24 (Sun Microsystems Inc.)
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Nuxeo-Component: OSGI-INF/process-actions.xml,
    OSGI-INF/process-type.xml,
    OSGI-INF/process-ui-type.xml,
    OSGI-INF/process_description-chains.xml,
    OSGI-INF/process_description-type.xml,
    OSGI-INF/process_description-ui-type.xml,
    ...
    OSGI-INF/process_requests-type.xml,
    OSGI-INF/process_requests-ui-type.xml
Bundle-ManifestVersion: 1
Bundle-SymbolicName: org.nuxeo.dev.cookbook.example.pap;singleton:=true
```

There are four things which you should change in this manifest:

- the vendor name,
- the name of the bundle itself,
- the symbolic name of the bundle,
- the components (the content aka *.xml).

The symbolic name should be a domain controlled by your organization and it should be unique inside your organization. Additionally it is a good idea to choose a name which is some kind of self explanatory in terms of big environments where developers, maintainers and operators are disjunct.

The components are the files which are loaded by Nuxeo. If your archive includes a configuration file which is missing in the MANIFEST then it will be ignored without any error message from Nuxeo.



Want to know more?

More information about the MANIFEST are available from the [How-to create an empty bundle recipe](#).

Makefile

Why a makefile? Well, the author is an old style Unix guy which is in fact a bad excuse for my lack of knowledge in terms of Maven.

The makefile supports two important functions - make and make install. make builds the JAR archive and make install places it at the correct path and restarts the server. make install only works on Debian based systems like Ubuntu. If you use another system then please ignore it and only use make.

So what is your actual/first task? Please replace the makefile with a Maven configuration (and a description) and contribute it back 😊

Definition of a new folder type

The folders create the general structure.
A folder itself usually does not contain a document.
It contains other folders and documents.
Additionally a folder is described by some meta data.

Nuxeo clearly separates the structural definition and the user interface of a document.
So both parts must be configured separately.

The examples utilize the process.
The process is the management or better only the granting/denial of parking permissions at area 51.
PAP means public administration process. It is used to guarantee the uniqueness of the component names.

Structural configuration

First we need to define the folder type itself. This means which (meta) data is needed in the folder and which features are available for the folder itself. So let's extend `org.nuxeo.ecm.core.schema.TypeService` with the component `org.nuxeo.dev.cookbook.example.pap.process.type`.

OSGI-INF/process-type.xml

```
<?xml version="1.0"?>
<component name="org.nuxeo.dev.cookbook.example.pap.process.type">

  <extension target="org.nuxeo.ecm.core.schema.TypeService" point="doctype">
    <doctype name="DevCookbookExamplePapProcess" extends="Folder">
      <schema name="common" />
      <schema name="dublincore" />
    </doctype>
  </extension>

</component>
```

The new type is only a special folder with most basic stuff.



Meta Data

Libraries especially in research environments like universities heavily depend on meta data standards. If your content should be usable, searchable and referenceable then it is strongly recommended that you use standards like Dublin Core. Please see here for more informations:

- http://en.wikipedia.org/wiki/Dublin_Core
- <http://dublincore.org/>
- [http://www.openarchives.org/documents/FAQ.html#Why does the protocol mandate a common metadata format \(and why is that common format Dublin Core](http://www.openarchives.org/documents/FAQ.html#Why does the protocol mandate a common metadata format (and why is that common format Dublin Core)

User interface

The user interface configuration must basically include two things - the layout and the parents.

First the layout stuff defines how a type is displayed. It does not only create a label and a description.

The category defines for example in which section you will see the the document type if you click on the `add document` button.

You can find a detailed description here [Document types](#).

The parents are defined to give the user interface a hint where the the new document type should be placed.

The example process is only allowed directly at the root of a workspace.

OSGI-INF/process-ui-type.xml

```

<?xml version="1.0"?>
<component name="org.nuxeo.dev.cookbook.example.pap.process.ui-type">

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

    <type id="DevCookbookExamplePapProcess">
      <label>Public Administration Process</label>
      <description>This is an example for a basic process in the public
administration.</description>
      <category>Collaborative</category>
      <default-view>view_documents</default-view>
      <layouts mode="any">
        <layout>heading</layout>
      </layouts>
      <layouts mode="edit">
        <layout>heading</layout>
        <layout>dublincore</layout>
      </layouts>
      <icon>icons/folder.gif</icon>
      <bigIcon>icons/folder_100.png</bigIcon>
    </type>

    <!-- only allow the basic process folder in the root of a workspace -->

    <type id="Workspace">
      <subtypes>
        <type>DevCookbookExamplePapProcess</type>
      </subtypes>
    </type>

  </extension>

</component>

```

Actions

If you create a new type then you usually do this to better control the activities and the workflows inside this type. The next step is to get some button to perform some [actions](#) or [operation chains](#).

The first activity of every good paper tiger is the creation of a good description of the process. Mainly this includes a short name and a description of the content (e.g. parking permissions). Additionally requests are only accepted if there is a description. This means that the button has to do more than only one operation. Et voila, we need an [operation chain](#).

OSGI-INF/process-actions.xml

```
<?xml version="1.0"?>
<component name="org.nuxeo.dev.cookbook.example.pap.process.actions">

  <extension target="org.nuxeo.ecm.platform.actions.ActionService" point="actions">

    <!-- first you have to create the description of the process -->

    <action id="newDevCookbookExamplePapProcessDescription"

link="#{operationActionBean.doOperation('CreateDevCookbookExamplePapProcessDescription
')}}"
      enabled="true"
      order="10"
      label="Create a process description"
      icon="/icons/action_add.gif">
    <category>SUBVIEW_UPPER_LIST</category>
    <filter id="newDevCookbookExamplePapProcessDescription">
      <rule grant="true">
        <permission>AddChildren</permission>
        <type>DevCookbookExamplePapProcess</type>
      </rule>
    </filter>
    </action>

  </extension>

</component>
```

The content of the action element is simple:

Name	Description
id	Name
order	A hint where to place the button if there are several actions.
label	The displayed name.
icon	The used basic icon.
link	The function call with the correct parameters.

The operation is a speciality here because it is the operation to call a chain. The parameter is the name of the chain.

The category is the position where to place the button. Please note that this depends on the document type. If you have a folder then SUBVIEW_UPPER_LIST is a normal button directly below the name and the description of the folder.

The filter is a control that defines when to display an action. If you want to reuse a filter, you can do this by only specifying the matching id. Here we only want to see the button if we have the permission to create new documents in this folder (e.g. a description or a folder with the requests) and if the folder is of type DevCookbookExamplePapProcess.

Training lesson

Before we start with the tests please remember that you can always look into the JAR archive to check your solution.

A request folder will contain the form from the customer, the approval or denial of a parking permission.

The parent of a request folder is a DevCookbookExamplePapProcessRequests folder.

Every customer form will be in a separate request folder.

Please create and fill the following files:

1. process_request-type.xml
2. process_request-ui-type.xml

Definition of a new file type

A normal file (which is not a folder) usually contains only meta data or additionally a real file. So if you think in terms of a file system hierarchy then this is the place where to store your data.

If you want to store a text or PDF file in an ECM system then you store some raw data together with some meta data.

A file system knows such meta data too but it is very limited (e.g. ACLs, ownership, some timestamps, etc.).

If you use an ECM system then you can freely define which meta data you need and want to store.

Structural configuration

The structural definition is a little bit more complex here. Some more schemas are used to get additional features like unique identifiers and typical file meta data. The really interesting stuff are the facets.



Warnings from facets

If you look into your server log then you will usually see a warning that an unknown facet is used but this can be ignored (Nuxeo Enterprise Platform 5.5).

The facets support us with two features.

Publishable makes it possible that the description can be copied to **Sections**. This area is used only to make documents available. You cannot edit documents in sections. So if you want to publish the description (e.g. to people who work in area 51), then a place in **Sections** would be right.

Versionable allows you to maintain different versions of the same document. This could be useful if you need to know all old descriptions. The versioning of Nuxeo is described later in this tutorial.

OSGI-INF/process_description-type.xml

```
<?xml version="1.0"?>
<component name="org.nuxeo.dev.cookbook.example.pap.process-description.type">

  <extension target="org.nuxeo.ecm.core.schema.TypeService" point="doctype">
    <doctype name="DevCookbookExamplePapProcessDescription" extends="Document">
      <schema name="common" />
      <schema name="uid" />
      <schema name="dublincore" />
      <schema name="file" />
      <facet name="Versionable" />
      <facet name="Publishable" />
    </doctype>
  </extension>

</component>
```

User interface

The specification of the user interface only includes some minor interesting stuff. The special stuff for files is made available and the description is only allowed directly in the root folder of a process.

That's it. KISS.

OSGI-INF/process_description-ui-type.xml

```
<?xml version="1.0"?>
<component name="org.nuxeo.dev.cookbook.example.pap.process-description.ui-type">

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

    <type id="DevCookbookExamplePapProcessDescription">
      <label>Process description</label>
      <description>The description of the process.</description>
      <category>SimpleDocument</category>
      <default-view>view_documents</default-view>
      <layouts mode="any">
        <layout>heading</layout>
        <layout>file</layout>
      </layouts>
      <layouts mode="edit">
        <layout>heading</layout>
        <layout>file</layout>
        <layout>dublincore</layout>
      </layouts>
      <icon>icons/file.gif</icon>
      <bigIcon>icons/file_100.png</bigIcon>
    </type>

    <!-- only the process folder itself should contain a process description -->

    <type id="DevCookbookExamplePapProcess">
      <subtypes>
        <type>DevCookbookExamplePapProcessDescription</type>
      </subtypes>
    </type>

  </extension>

</component>
```

Training lesson

If you want to test your understanding you can create now the following configurations:

1. The user request form
 - a. process_request_form-type.xml
 - b. process_request_form-ui-type.xml
2. The file which documents the approval of the user request
 - a. process_request_approval-type.xml
 - b. process_request_approval-ui-type.xml
3. The file which documents the rejection of the user request
 - a. process_request_reject-type.xml
 - b. process_request_reject-ui-type.xml

Please remember you can find all files in the JAR archive.
It is not necessary to do all the stuff by yourself.

Actions

Let's assume there is a customer request but there is no decision. So we need two buttons on the request form: one to approve and one to reject the request.

If you want to configure an action button on a file then this works a little bit different than on folders.

First you need to define a tab and then you can place buttons in this tab.
The example re-use the folder panel as tab.

Let's start with the approval of a request.

The category `VIEW_ACTION_LIST` defines a new tab in the document.
The XHTML document in the parameter `link` defines which template is used.

OSGI-INF/process_request_form-actions.xml

```
<?xml version="1.0"?>
<component name="org.nuxeo.dev.cookbook.example.pap.process-request-form.actions">

  <extension target="org.nuxeo.ecm.platform.actions.ActionService" point="actions">

    <action id="tabDevCookbookExamplePapProcessRequestDecision"
      link="/incl/document_actions.xhtml"
      enabled="true"
      order="30"
      label="Decision"
      icon="/icons/file.gif">
      <category>VIEW_ACTION_LIST</category>
      <filter id="tabDevCookbookExamplePapProcessRequestDecision">
        <rule grant="true">
          <type>DevCookbookExamplePapProcessRequestForm</type>
          <!-- permission:write -->
        </rule>
      </filter>
    </action>
```

The category `SUBVIEW_UPPER_LIST` is the usual category from the folders.
This works here because the template is the default folder template.

```
<action id="approveDevCookbookExamplePapProcessRequest"

link="#{operationActionBean.doOperation('ApproveDevCookbookExamplePapProcessRequest')}"

  <extension target="org.nuxeo.ecm.platform.actions.ActionService" point="actions">
    <action id="approveDevCookbookExamplePapProcessRequest"
      link="/incl/document_actions.xhtml"
      enabled="true"
      order="20"
      label="Approve the request"
      icon="/icons/action_add.gif">
      <category>SUBVIEW_UPPER_LIST</category>
      <filter id="approveDevCookbookExamplePapProcessRequest">
        <rule grant="true">
          <type>DevCookbookExamplePapProcessRequestForm</type>
          <!-- permission:write -->
        </rule>
      </filter>
    </action>
  </extension>
</component>
```

Now there is a tab called `Decision` with a button `Approve request`.



I18N - Internationalization

Please note that you can translate the labels. You can translate in the usual `message.po` files. The problem is that there is only one central file. So you must change the master file.

Training lesson

Okay, the task to implement the reject button should be very simple. Please create the reject button (perhaps on another tab).

Operation chains

Operation chains are a very powerful tool. Every operation has an input and an output. If you want to connect them like a pipe then you just have to ensure that the input and output are compatible. This sounds complicated but it is very easy because there are not so many allowed input and output types:

- void
- document
- blob

The only problem is that even a list of documents is represented as a blob. So you cannot directly use it but below you will find a workaround.

Basic stuff

The example shows the operation chain for the creation of the description of the process. The configured action above defined the name `CreateDevCookbookExamplePapProcessDescription` for this chain. What is the job of this chain? First create a description, second create the folder for the requests and last open the description to fill it with some content.

First things first. Let's create the description. No, just a moment.

There are two very important functions - `Seam.AddErrorMessage` and `Seam.AddInfoMessage`. Please always add an error message in front of every operation or functional block to ensure that you get a meaningful error message. If the operation chain ends then please add an info message which reports the success. This costs some time but it enormous helpful if something fails and the user can give you meaningful error messages because they learned that every operation results in a message.

The following code snip is quite simple. It creates the description and give it the name description.

OSGI-INF/process_description-chains.xml

```
<?xml version="1.0"?>
<component name="org.nuxeo.dev.cookbook.example.pap.process-description.chains">

  <extension target="org.nuxeo.ecm.core.operation.OperationServiceComponent"
    point="chains">

    <chain id="CreateDevCookbookExamplePapProcessDescription">
      <!--
        Create a fresh description
        Create folder requests
        Open the description
      -->
      <operation id="Seam.AddErrorMessage">
        <param type="string" name="message">The description could not be
created.</param>
      </operation>
      <operation id="Document.Create">
        <param type="string"
name="type">DevCookbookExamplePapProcessDescription</param>
        <param type="string" name="properties">dc:title=description</param>
      </operation>
```

Next the resulting document is loaded to the user interface. So if the chain is completed, the user sees it.

```
...
    <operation id="Seam.AddErrorMessage">
      <param type="string" name="message">Cannot load the description to the
UI.</param>
    </operation>
    <operation id="Seam.NavigateTo">
    </operation>
```

Perhaps you will ask now, why you do the last step first and what happens if something fails later? The document is only available now as an input. I could store it in a variable but this is not necessary. Operation chains work as a single transaction. So if something fails, nothing happened.

After the description is done, there is still one thing missing - the folder for the requests. So let's create it.

```
...
    <operation id="Document.GetParent">
    </operation>
    <operation id="Seam.AddErrorMessage">
      <param type="string" name="message">The folder for the requests could not be
created.</param>
    </operation>
    <operation id="Document.Create">
      <param type="string" name="type">DevCookbookExamplePapProcessRequests</param>
      <param type="string" name="properties">dc:title=requests</param>
    </operation>
```

Do you see the mistake? Just remember, first things first!

Finally the success message is setup. Please note that you can configure it at any place in the chain. The functions to set an error or an information just copy the input to the output. They are so called void functions.

```
...
    <operation id="Seam.AddInfoMessage">
      <param type="string" name="message">The description was successfully
created.</param>
    </operation>
  </chain>

</extension>

</component>
```

Dynamic names / MVEL

The operation environment implements a very useful scripting engine [MVEL](#). So let's take a look at a simple example.

The example create a new customer request. First we need to calculate which request it is. Please remember this is a public administration example. So the name of the 42nd request is `request-42` and not `request-41`.

The first part includes the following steps:

- ensuring that the folder with the requests is really loaded in the context,
- caching this folder in the variable `requests_folder`,
- get the children of this folder (the requests),
- store the result in the variable `documents`
- store the number of the children plus one in the variable `request_count`

- restore the folder with the requests from the variable `requests_folder`

Finally we know which request will be the next and the parent folder is the input.
This is business as usual.

OSGI-INF/process_request-chains.xml

```
<?xml version="1.0"?>
<component name="org.nuxeo.dev.cookbook.example.pap.process-request.chains">

  <extension target="org.nuxeo.ecm.core.operation.OperationServiceComponent"
point="chains">

    <chain id="CreateDevCookbookExamplePapProcessRequest">
      <!--
        Input: Called from the folder with all requests
        Calculate a name for the new folder
        Create folder with title
        Create customer_form
        Open customer form
      -->
      <!-- get the folder as input -->
      <operation id="Seam.AddErrorMessage">
        <param type="string" name="message">Cannot load the folder with the
requests.</param>
      </operation>
      <!-- usually unnecessary -->
      <operation id="Context.FetchDocument">
      </operation>
      <!-- determine number of children -->
      <operation id="Seam.AddErrorMessage">
        <param type="string" name="message">Cannot calculate the name of the new
file.</param>
      </operation>
      <operation id="Context.SetInputAsVar">
        <param type="string" name="name">requests_folder</param>
      </operation>
      <operation id="Document.GetChildren">
      </operation>
      <operation id="Context.SetInputAsVar">
        <param type="string" name="name">documents</param>
      </operation>
      <operation id="Context.SetVar">
        <param type="string" name="name">request_count</param>
        <param type="object"
name="value">expr:@{ (Context["documents"].size()+1).toString()}</param>
      </operation>
      <operation id="Context.RestoreDocumentInput">
        <param type="string" name="name">requests_folder</param>
      </operation>
```

Okay, I agree that the determination of `request_count` is not hundred percent intuitive and the author needed some support from the community liaison too 😊👍

Nevertheless it is possible to learn a lot from this one-liner.

Second the request folder will be created. Please note the usage of the dynamically calculated request count.
If you use variable then always use the context. The other stuff is quite intuitive.

```
<!-- create structure -->
  <operation id="Seam.AddErrorMessage">
    <param type="string" name="message">Cannot create a new request
folder.</param>
  </operation>
  <operation id="Document.Create">
    <param type="string" name="type">DevCookbookExamplePapProcessRequest</param>
    <param type="string"
name="properties">expr:dc:title=request-@{Context["request_count"]}</param>
  </operation>
```

Third the customer form is created and loaded.

```
<operation id="Seam.AddErrorMessage">
  <param type="string" name="message">Cannot create a new customer form.</param>
</operation>
<operation id="Document.Create">
  <param type="string"
name="type">DevCookbookExamplePapProcessRequestForm</param>
  <param type="string" name="properties">dc:title=customer_form</param>
</operation>
<operation id="Seam.AddErrorMessage">
  <param type="string" name="message">Cannot load the new customer form.</param>
</operation>
<operation id="Seam.NavigateTo">
</operation>
<operation id="Seam.AddInfoMessage">
  <param type="string" name="message">The new request was successfully
created.</param>
</operation>
</chain>

</extension>

</component>
```

Training lessons

If you want to do some training then you can write some chains for the approval or denial of the customer requests. You can create the according documents (e.g. a ticket for the car window) or you can send an email to the entrance (or an invoice to planet Mars).

Versioning

If you use a document management system or a similar system then you expect that documents have versions. If you are a software developer then you are familiar with checkouts and commits.

If you approve a request then the customer form should get a new final version. The following chain which implements the approval creates such a new major version.

OSGI-INF/process_request_approval-chains.xml

```
<?xml version="1.0"?>
<component name="org.nuxeo.dev.cookbook.example.pap.process-request-approval.chains">

  <extension target="org.nuxeo.ecm.core.operation.OperationServiceComponent"
point="chains">

    <chain id="ApproveDevCookbookExamplePapProcessRequest">
      <!--
        Create new major version
        Create an approval
        Remove write permission on request form
        Open the approval
      -->
      <operation id="Document.CheckOut">
      </operation>
      <operation id="Document.CheckIn">
        <param type="string" name="version">major</param>
        <param type="string" name="comment">The request was approved.</param>
      </operation>
      <operation id="Seam.AddErrorMessage">
        <param type="string" name="message">The approval document could not be
created.</param>
      </operation>
      <operation id="Document.GetParent">
      </operation>
      <operation id="Document.Create">
        <param type="string"
name="type">DevCookbookExamplePapProcessRequestApproval</param>
        <param type="string" name="properties">dc:title=approval</param>
      </operation>
      <operation id="Seam.AddErrorMessage">
        <param type="string" name="message">Cannot load the approval to the
UI.</param>
      </operation>
      <operation id="Seam.NavigateTo">
      </operation>
      <operation id="Seam.AddErrorMessage">
        <param type="string" name="message">Cannot remove the write permissions from
the customer form.</param>
      </operation>
      <operation id="Document.GetParent">
      </operation>
      <!-- this does not work if you work as Administrator ;) -->
      <operation id="Document.SetACE">
        <param type="string" name="permission">WriteProperties</param>
        <param type="string" name="user"></param> <!-- this needs a real user -->
        <param type="boolean" name="grant">false</param>
      </operation>
      <operation id="Seam.AddInfoMessage">
        <param type="string" name="message">The approval was successfully
created.</param>
      </operation>
    </chain>

  </extension>

</component>
```

The initial check out is necessary to get a working copy of the document.
You can only check in documents which are a working copy.

The check in requires at minimum the parameter `version` which differs between minor or major check ins.
The final check after an approval is of course a major version.
The comment is optional which is a horror for people who uses revision control systems for software development.
One of the most important rule is: **No commits without comments.**

Training lesson

Please create the configuration `process_request_reject-chains.xml` as a test of your understanding.

Management of states (life cycle)

Perhaps you note the title of the section is not *Status management*.
The reason is simple: Nuxeo enforces a life cycle model which is some kind of a finite state machine.
If you need some kind of state for a folder or a document, then you have to define a life cycle including the allowed state transitions.

The definition is quite straight forward:

OSGI-INF/process_request-lifecycle.xml

```
<?xml version="1.0"?>
<component name="org.nuxeo.dev.cookbook.example.pap.process-request.lifecycle">

  <extension target="org.nuxeo.ecm.core.lifecycle.LifeCycleService" point="lifecycle">

    <lifecycle name="DevCookbookExamplePapProcessRequestPolicy" initial="new">

      <states>
        <state name="new">
          <transitions>
            <transition>approve</transition>
            <transition>reject</transition>
          </transitions>
        </state>
        <state name="approved">
        </state>
        <state name="rejected">
        </state>
      </states>

      <transitions>
        <transition name="approve" destinationState="approved">
          <description>Approve the request.</description>
        </transition>
        <transition name="reject" destinationState="rejected">
          <description>Reject the request.</description>
        </transition>
      </transitions>

    </lifecycle>

  </extension>
```

The states are only used by Nuxeo to define the allowed transition paths.
The states `approved` and `rejected` are only specified because it gives a better overview.
The state after a transition is only taken from `destinationState`.
If there is a typo in a state then this is a dead end without an error message.

After the definition of the life cycle, you have to apply the life cycle to one or more document types.

You can re-use such life cycles and you can use life cycles to filter actions.

The filter configuration does not support a state filter directly but there is an operation `Operation Document.Filter` which supports a state filter.

```
...
    <extension target="org.nuxeo.ecm.core.lifecycle.LifeCycleService" point="types">
        <types>
            <type
name="DevCookbookExamplePapProcessRequest">DevCookbookExamplePapProcessRequestPolicy</
type>
            </types>
        </extension>

</component>
```

The integration of the life cycle into the configuration is simple.

The initial state is predefined in the life cycle configuration.

You have only to add a state change in the chain of the approval (`Operation Document.SetLifeCycle`).

The parameter is the name of the transition.

OSGI-INF/process_request_approval-chains.xml

```
...
    <operation id="Document.GetParent">
    </operation>
    <!-- Document is now the request folder -->
    <operation id="Document.SetLifeCycle">
        <param type="string" name="value">approve</param>
    </operation>
    ...
```

Training lesson

If you want to verify your knowledge then you can create a life cycle for the request form itself.

Alternatively you can only add the state change to the reject chain.

Extension of meta data

Sometimes you need more meta informations than usual.

If you want to manage such a special place like area 51 then you need several different kinds of parking places.

You have cars, air planes, x-planes and of course ufos 😊

Structural configuration

First you need to extend the meta data of a document type.

Such structural changes are done via XML schemas.

So let's introduce a new vehicle type:

schemas/process_request_form.xsd

```
<?xml version="1.0"?>
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema"

  targetNamespace="http://project.nuxeo.org/schemas/dev/cookbook/example/pap/process-request-form/"

  xmlns:area51="http://project.nuxeo.org/schemas/dev/cookbook/example/pap/process-request-form/"
  >

  <xs:simpleType name="vehicle_type">
    <xs:restriction base="xs:string">
      <xs:enumeration value="Car"/>
      <xs:enumeration value="Airplane"/>
      <xs:enumeration value="X-Plane"/>
      <xs:enumeration value="Ufo"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:element name="parking_area" type="area51:vehicle_type"/>
</xs:schema>
```

So we have now a schema with exactly one element called `parking_area`.

Now need to make it available.

This requires a small extensions of the type definition:

OSGI-INF/process_request_form-type.xml

```
<?xml version="1.0"?>
<component name="org.nuxeo.dev.cookbook.example.pap.process-request-form.type">

  <extension target="org.nuxeo.ecm.core.schema.TypeService" point="schema">
    <schema name="area51" src="schemas/process_request_form.xsd" prefix="area51" />
  </extension>

  <extension target="org.nuxeo.ecm.core.schema.TypeService" point="doctype">
    <doctype name="DevCookbookExamplePapProcessRequestForm" extends="Document">
      <schema name="common" />
      <schema name="uid" />
      <schema name="dublincore" />
      <schema name="file" />
      <schema name="area51" />
      <facet name="Versionable" />
    </doctype>
  </extension>

</component>
```

The change is minimal. You need to import the schema via the according extension point and you must add one line to the document type definition. That's it.

User interface

Second you must modify the user interface. This is much more tricky in this case. You need in this case a new database table, a new widget and a new layout. Finally you have to extend the type for the user interface itself.

Configure a new database table

The data source is called directory in Nuxeo.

Such directories can be LDAP directories or databases.

If you want to import some list data then you must store them in a CSV file and import it.

directories/vehicle_type.csv

```
id,label,obsolete
"Car","Car","0"
"X-Plane","X-Plane","0"
"Airplane","Airplane","0"
"Ufo","Ufo","0"
```

The import is quite simple:

OSGI-INF/process_request_form-ui-type.xml

```
<extension target="org.nuxeo.ecm.directory.sql.SQLDirectoryFactory"
point="directories">
  <directory name="vehicle_type">
    <schema>vocabulary</schema>
    <dataSource>java:/nxsqldirectory</dataSource>
    <cacheTimeout>3600</cacheTimeout>
    <cacheMaxSize>1000</cacheMaxSize>
    <table>area51_vehicle_type</table>
    <idField>id</idField>
    <autoincrementIdField>false</autoincrementIdField>
    <dataFile>directories/vehicle_type.csv</dataFile>
    <createTablePolicy>on_missing_columns</createTablePolicy>
  </directory>
</extension>
```

Configure a new widget

After you have now a new directory which is filled with the data from a CSV file you need to create a widget.

A widget includes the form field or the field value depending on the situation.

OSGI-INF/process_request_form-ui-type.xml

```
<extension target="org.nuxeo.ecm.platform.forms.layout.WebLayoutManager"
point="widgets">

  <widget name="type_of_vehicle" type="selectOneDirectory">

    <labels>
      <label mode="any">Vehicle type</label>
    </labels>
    <translated>true</translated>
    <fields>
      <field>area51:parking_area</field>
    </fields>

    <properties widgetMode="any">
      <property name="directoryName">vehicle_type</property>
      <property name="localize">true</property>
      <property name="ordering">label</property>
    </properties>

  </widget>
```

The widget defines some usual stuff like name, label and meta data field id.

The interesting stuff here is the widget type.

You can find the available basic widget types at the [Standard widget types](#).

selectOneDirectory implements a select field. The options are taken from the directory called vehicle_type.

This directory was defined above. localize means that the label is translatable.

Configure a new layout

The new widget is data only. It is necessary to layout the data.

This is done with the creation of a new layout:

OSGI-INF/process_request_form-ui-type.xml

```
<require>org.nuxeo.ecm.platform.forms.layouts.webapp</require>

<extension target="org.nuxeo.ecm.platform.forms.layout.WebLayoutManager"
point="layouts">

  <layout name="area51">
    <templates>
      <template mode="any">/layouts/layout_default_template.xhtml</template>
    </templates>
    <rows>
      <row>
        <widget>type_of_vehicle</widget>
      </row>
    </rows>
  </layout>

</extension>
```

Now we have a new layout area51 which uses the widget type_of_vehicle.

Use the new layout

The new layout `area51` must be used by the user interface itself.
This is quite simple. The layout must be added to every `layouts` section.

OSGI-INF/process_request_form-ui-type.xml

```
<layouts mode="edit">
  <layout>heading</layout>
  <layout>file</layout>
  <layout>dublincore</layout>
  <layout>area51</layout>
</layouts>
```

Training lesson

If you want to test your knowledge and understanding then simply add a new field.
Just use your fantasy.



Want to see more or other features?

If you are looking for an example of another feature then please feel free to comment or change the TODO list.

How to remove the Language Selector

Your browsers already have the correct language set and you have no further use of the Language Select Box?

The easiest way is to customize the footer fragment from:

```
nuxeo-platform/nuxeo-platform-webapp/src/main/resources/nuxeo.war/incl/footer.xhtml
```

to:

```
yourProject/.../nuxeo.war/incl/footer.xhtml
```

and remove the language box:

```
<h:form>
  <h:outputText value="#{messages['label.selectLocale']}" />
  <h:selectOneMenu value="#{localeSelector.localeString}" styleClass="langSelect">
    <f:selectItems value="#{localeSelector.supportedLocales}" />
  </h:selectOneMenu>
  <h:commandButton action="#{localeSelector.select}"
    value="#{messages['command.changeLocale']}"
    class="langSubmit" />
</h:form>
```

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.

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 [N XMOB](#), [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 (non English translations)

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 "save", and optionally, if you're a proofreader, you can approve the translation.
5. Contact one or several of the Crowdin project managers to [be credited for your contribution](#).

Without Crowdin (English translations)

1. For now, English translations are managed only on GitHub. Looking at the [reference messages.properties](#) file at can help you understand in which GitHub repository or module the original translation is. For instance, look for the following sample lines:

```
## DO NOT EDIT FOLLOWING LINE
# Translations from
./nuxeo/nuxeo-features/nuxeo-platform-lang/src/main/resources/web/nuxeo.war/WEB-INF/classes/messages_en_US.properties

[...]

## DO NOT EDIT FOLLOWING LINE
# Translations from
./nuxeo/addons/nuxeo-agenda/src/main/resources/OSGI-INF/l10n/messages_en_US.properties

[...]
```

If the module is under the `addons` directory, it will be in a specific GitHub repository. Otherwise, it will be in the [main Nuxeo repository](#).

2. Use any standard Java i18n tool to edit the files.
3. Make a pull-request on [GitHub](#).

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 add-ons.
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, when 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.

Not using GitHub?

You can still contribute patches even without using GitHub:

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).

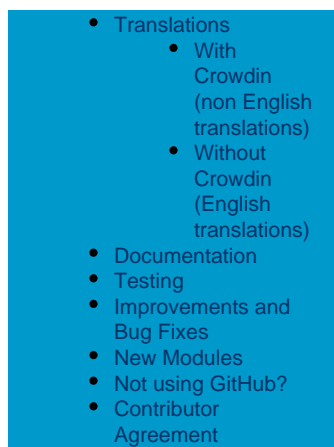
2. Send an email to [the nuxeo-dev mailing list](#), or post on [the Nuxeo Google+ community](#), to notify the community.
3. Read the [Coding and design guidelines](#).
4. Fork the "master" branch of the sub-project you want to patch.
5. Make your modifications, respecting the [coding and design guidelines](#), 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).

The signature of the Contributor Agreement is mandatory for a GitHub Pull-Request being accepted. You will be prompted for the signature in the PR comments or you can browse <https://cla-assistant.io/nuxeo/nuxeo>



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 Nuxeo DM

Language resources

Update or create localized messages

You will find French and English language resources files in `nuxeo-platform-lang/src/main/resources/web/nuxeo.war/WEB-INF/classes` living in <https://github.com/nuxeo/nuxeo-features/> GitHub repository.

Other language translations live in <http://hg.nuxeo.org/addons/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>

You should create your translations from `messages_en.properties` or `messages_fr.properties` since only those are maintained by Nuxeo developers.

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>
    <supported-locale>ar</supported-locale>
    <supported-locale>cn</supported-locale>
    <supported-locale>de</supported-locale>
    <supported-locale>es</supported-locale>
    <supported-locale>it</supported-locale>
    <supported-locale>ja</supported-locale>
    <supported-locale>ru</supported-locale>
    <supported-locale>vn</supported-locale>
  </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 <http://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.

Creating Packages for the Marketplace

The artifacts managed by the Update Center are called **packages**. 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.

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.

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 your 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**: state of Nuxeo's validation process. One of: *none*, *inprocess*, *primary_validation*, *nuxeo_certified*.
- **ProductionState**: One of: *proto*, *testing*, *production_ready*.

For more informations 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 driven 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 (Throwable 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 where 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/NXRruntime 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 by 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 Packages for the Marketplace](#) for an explanation of each package property.

Scripting Commands

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>
```

You can see the file is using contextual variables as *env.bundles*. etc. See [Creating Packages for the Marketplace](#) for the complete list of context variables.

Lets take each command and see what will be executed:

```
<copy file="${package.root}/myplugin.jar" tofile="${env.bundles}"
fail="tofile.isFile()" />
```

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). 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 **Guard Attributes** section for more details on using guards.

The second copy command

```
<copy file="${package.root}/my.properties" tofile="${env.config}/my.properties"
ignore="Platform.isJBoss()" />
```

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.

You can see here the usage of another type of guard parameter: **ignore**.

The third copy command is a bit more complicated:

```
<copy file="${package.root}/mylib-1.2.jar"
tofile="${env.lib}/mylib-{version:.*}.jar"
ignore="Version.isGreaterOrEqual(version, \"1.2\")" />
```

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.

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 *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 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 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.).
Example: 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 I will take the built-in copy command. To simplify lets say the copy command has a **file** parameter, a **tofile** parameter and an optional **md5** parameter.

When the copy command (lets name it 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 (lets name it 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 took 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 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.

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 Nuxeo 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.ecm.platform:nuxeo-connect-update` (nuxeo-admin-center/nuxeo-connect-update in <http://github.com/nuxeo/nuxeo-features/>).

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

Copy 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

Delete 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

Start 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

Stop 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

Flush 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/>
```

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 doesn'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.

Nuxeo Distributions

About Nuxeo Distributions

The `nuxeo-distribution` module is used for packaging of the Nuxeo products: Nuxeo EP/DM with JBoss/Jetty/Tomcat, Nuxeo Shell, Nuxeo Core Server, ...

With `nuxeo-distribution`, you can [build from Nuxeo sources](#) any Nuxeo distribution we have published 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.

Table of Content

- [Available installers](#)

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 — coming soon),
- [Windows installer](#) (.exe),
- [Linux Debian installer](#) (.deb),
- Red Hat Package manager (.rpm — not available for now),
- Mac OS X disk image/application/package (.dmg/.app/.pkg — not available for now).

Any help about OS-specific cases, recommendations, contributions or feedbacks is very welcome. See <https://jira.nuxeo.com/browse/NXBT> for issue management.