Nuxeo CMF 1.x

Developer Documentation

Table of Contents

1. Nuxeo CMF developer documentation . . . 3
  - 1.1 Introduction . . . 3
  - 1.2 Case Management . . . 3
    - 1.2.1 Content Model . . . 3
    - 1.2.2 Interface Components . . . 5
    - 1.2.3 Document and manipulation . . . 7
    - 1.2.4 Service interface . . . 9
    - 1.2.5 Application Packaging . . . 9
  - 1.3 Content Routing . . . 10
    - 1.3.1 Content Routing documents . . . 10
    - 1.3.2 Route processing and lifecycle . . . 12
    - 1.3.3 Creating Tasks . . . 15
    - 1.3.4 Advanced Features . . . 17
  - 1.4 Additional Functionalities . . . 18
    - 1.4.1 Mailbox synchronization . . . 18
    - 1.4.2 Classification (aka filing) . . . 20
    - 1.4.3 Email capture . . . 20
    - 1.4.4 File system import . . . 21
      - 1.4.4.1 Simple importer . . . 21
      - 1.4.4.2 Case Importer . . . 21

Nuxeo CMF developer documentation

This is the developer manual for the Nuxeo Case Management Framework. It is intended for developers who want to create applications using this framework. Knowledge of the Nuxeo Platform will help to understand this book, but is not mandatory.

A Case Management application uses 4 key objects:

- A **Case Item** is a document - it can be a file, a picture or a folder containing other documents.
- A **Case** is a set of Case Items.
- A **Mailbox** (aka **Case Folder**) contains Cases.
- A **Case Link** is a symlink from a Case Folder to a Case.

Download
Download this documentation in PDF.

If we were to use claim management nomenclature as an example:

- A claim (Case) is filled with documents: expert reports (Case Item), pictures (Case Item) expenses reports (Case Item)...
- The claim can be processed by the financial services department, the legal department and a claim processing manager. Each group has a Case Folder in which they can see the list of claims they are processing.
- A claim (Case), contained in the Case Folder of the legal departement, the financial department and of the manager, points to the same Case Item. However, the actions and fields visible/changeable on each Case Item depend on the Case Folder it is in.

This document shows how Nuxeo CMF works and also how to create a custom application from Nuxeo CMF.

Introduction

Glossary

Here's a list of all terms used in this document.

- **Case Management**: case management, including creation of documents, receiving and sending
- **Case**: a document used for grouping of other documents. Cases are shared by all users receiving it. Any change applied to it will impact all receivers.
- **Case Item**: a document to be sent, holding the relevant metadata. This information is shared by all users receiving it. Any change appliled to it will impact all receivers. A CaseItem can not be distributed without a Case.
- **Mailbox**: the main container for case links received and sent. Also, it holds additional metadata: profiles having access to a given mailbox, the owner of the mailbox, the type of mailbox, a list of favourite recipients and other settings.
- **CaseLink**: an entry in a case folder, representing sending or receiving information. Also, it holds a reference to the shared cases or case items that are sent.

Main problematics

Nuxeo Case Management is a high-level application providing an array of generic features related to Case Management. Some of these features will be used as is by client projects, but others will need to be customized.

The application should be designed so that it's easy to customize some parts of the application. It should also state clearly what is inherent to it and cannot be changed.
The main class entry can be found here and here.

Case Management

In this section:

- Content Model
- Interface Components
- Document and manipulation
- Service interface
- Application Packaging

Content Model

Persistence Policy

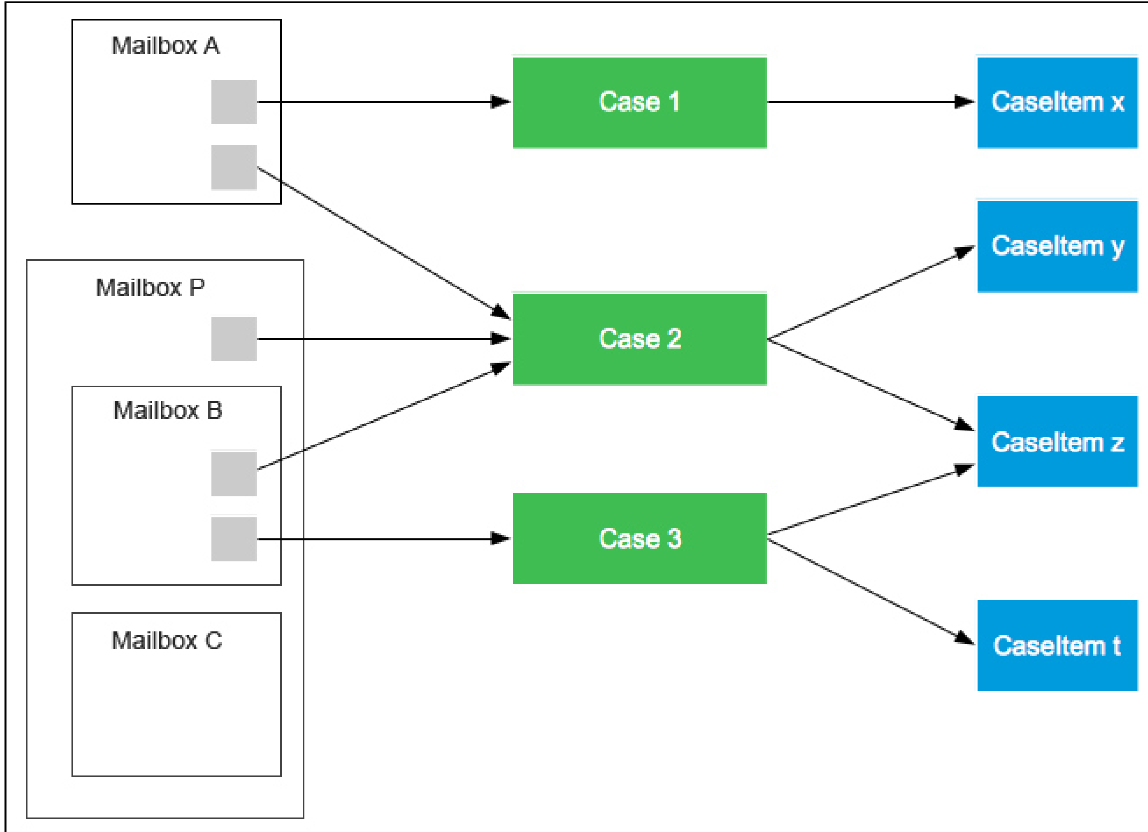
How a case and its case items are persisted is configurable. The framework offers 2 policies:

- The `hierachicalCase` policy persists all the items of a case under the case. It makes the creation of the user interface simpler as it can use Nuxeo's default way to show documents and to browse the item hierarchy. We recommend this policy for case management applications where case items have to be hierarchichal and where one case item can be found in only one case. This policy is used in the

default application shipped with Nuxeo CMF.
- The `SingleCaseItem` policy persists all the items outside the case. It allows to have the same items in different cases. We recommend this policy when users can create a case from existing items. This policy is used in the Correspondence application.

SingleCaseItem policy

In Nuxeo CMF, everything is a Nuxeo document. Nuxeo CMF uses the following document types: Case (green rectangle), CaseItem (blue rectangle), Mailbox (alias a CaseFolder) (white rectangle) and CaseLink (grey rectangle).



In this diagram, there are 3 Cases:

- Case 1 has 1 case item (CaseItem x) and is in MailboxA .
- Case 2 has 2 case items (CaseItem y and z) and is in Mailbox A, B and P.
- Case 3 has 2 case items (CaseItem z and t) and is in Mailbox B.

This shows some importants concepts about Nuxeo Case Management Framework:

- Mailboxes are folderish and can contain CaseLinks.
- The object in a Mailbox is a simple document (CaseLink). Sending a Case to a Mailbox only creates a CaseLink document: we do not copy the Case and its CaseItems in the Mailbox.
- Cases hold only references to CaseItems (not shown in this diagram is the fact that CaseItems are ordered inside a case) - we do not copy CaseItems inside Cases.
- Cases and CaseItems are not children of a Mailbox. This will have an impact on security.

Document hierarchy

Nuxeo CMF uses two hierarchies of documents to store content: Mailboxes are stored in one, and Cases and CaseItems in the other. Those two hierarchies can be placed side by side with the other usual roots (Workspaces, Sections and Templates).

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

```
-- CaseRoot
|    `-- 2009
|        `-- 10
|            |-- 18
|            |   |-- CaseItem x
|            |   `-- Case 1
|            `-- 19
|                |-- CaseItem y
|                |-- CaseItem z
|                |-- CaseItem t
|                |-- Case 2
|                |-- Case 3
|-- MailboxRoot
|   |-- Mailbox A
|   |       |-- CaseLink
|   |       |-- CaseLink
|   |-- Mailbox B
|   |       |-- CaseLink|   |       |-- CaseLink
`-- Nuxeo's additional needed roots (Worspaces, Template, Sections ...)
```

The diagram above provides a hierarchical representation of the items listed in the previous diagram (the one with rectangles). Case 1 and CaseItem x were created on the 18th of October 2009. The other cases and case items on the 19th.

The `CaseRoot` is used to store CaseItems and Cases documents. They are created in a “yyyy/mm/dd” hierarchy corresponding to their creation date, so that not too many documents are stored in the same folder. This hierarchy does not have any other functional purpose. By default, only administrators can access it. Rights on CaseItems and Cases will be set automatically by the distribution service.

The `MailboxRoot` is used to store all Mailboxes (personal, generic)... By default, only administrators have access to it. Rights on Mailboxes are handled automatically by the system (to give access rights to the delegates of the Mailbox for instance).

HierarchichalCase policy

This policy persists Cases the same way that the `SingleCaseItem` policy. It persists CaseItems under Cases in the repository.

Documents and envelopes

Documents can be accessed outside the context of a case, but they need to be sent/received through a case, which is the main entry when clicking on a received document.

Cases reference documents stored in them by storing target document IDs in a string list in its schema.

If other metadata need to be involved (a comment stating who added a document to the envelope, for instance), it would have to be duplicated in a custom complex schema, and synchronization between both fields should be possible through event listeners.

Document Security

CaseItems and Cases live in a different hierarchy than Mailboxes. The standard security principle are still applied too each document, however we need to be able to set the security depending on the folder the Case/CaseItem is in.

This is done in 2 steps:

- When a Case/a Case Item is created and distributed, a new ACE is added to the document with a group name of "mailbox_<MAILBOX_NAME>" for each mailbox the document is in. When the document is added/removed from a Mailbox, this ACE is added/removed.
- When a user logs in, the set of Mailboxes he can access is computed. Each one is added to the list of groups the user belongs to with the "mailbox_" prefix.

The setting of ACE is done using a listener. The setting of groups is done using the computed group extension point.

The normal Nuxeo security still applies to documents, and administrators can access the right management tab and give/remove access to the document to users and groups. The *mailbox specific* rights are not shown in that tab.

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

Interface Components

CaseItem

A CaseItem is a document that can be sent and received. Its creation is done via an adapter. The CaseItem implementation may hold additional information stored in the case complex schemas (a new Case implementation would then be used in custom projects):

- body: the body of the document
- senders: a list of userId
- sending date: the date the document was first sent
- receive date: the date the document was first received
- confidentiality: the confidentiality level of the document (by default from 1 to 4)
- origin: the origin of the document (from a mail, a OCR ...)

All the methods available via the CaseItem interface can be found here.

Case

The Case is the object that's being sent. Its creation is done via an adapter. When a Case is sent, a CaseLink is created in the sender's Mailbox a nd in each Mailbox of the receiver. It holds the following information:

All the methods available via the Case interface can be found here.

CaseLink

A CaseLink holds information about the case sent or received. Its creation is done via an adapter, by the system in Mailbox documents only, and holds the current information:

- ID
- postid, shared by all posts created from a given sent post
- subject
- comment
- sending date
- sender (sending mailbox title)
- sender mailbox ID
- internal recipients (map with 'type' as key (action, information, refusal, etc) are possible values and 'recipient' as another key (with mailbox id as value)).
- external recipients (map with 'type' as key (action, information, refusal, etc...) and other keys 'name', 'email' to store external user information)
- type (sent, action, information, refusal, etc...) may be null in case of multiple types
- read (boolean)

This metadata should be mapped to getters in the CaseLink interface:

```
String getSubject();
boolean isRead();
Case getCase();
String getCaseId();
String getComment();
CaseFolder getSender();
String getSenderId();
String getCaseLinkType();
boolean isRead();
Case getCase();
DocumentModel getDocument();
```

When sending a document, a current Mailbox needs to be in the context. A document of type “CaseLink” with `incoming=false` is created in the Mailbox folder.

The sent CaseLinks are stored in the sending Mailbox as documents so that it's possible to get the history of sent documents, allowing filtering, sorting, and the possibility to remove them (among other features).

When a Case is sent, a new CaseLink with `incoming=true` is created in all resolved participant mailboxes. The type (action, information...) is also updated according to the given participant. The other information is kept as is.

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

The received CaseLink objects are stored in participants' Mailboxes as documents so it's possible to get the history of sent documents, allow filtering, sorting, and remove them, or mark them as read (among other features).

Mailbox

Definition

A Mailbox folder is a container for CaseLink. Its creation is done via an adapter. It holds the following information:

- id
- title
- description
- type (personal or generic)
- owner
- users (aka Manager)
- groups (aka group Manager)
- affiliated Mailbox id
- notified users
- favorites
- mailing lists
- profiles

All the methods available via the Mailbox are listed here.

The Mailbox is a folderish container created in a special branch of the document hierarchy and it provides the usual features we get in Nuxeo when navigating to a mailbox folder, handling access rights, managing contained documents.

Users who have access to the mailbox (delegates) will inherit rights to the mailbox folder, as well as the rights to the documents created inside it (CaseLinks as well as other documents like classification folders). The "right management" privilege should be given to all users who have access to the mailbox, but the Mailbox interface should filter information that can only be changed by an administrator (for instance, type and profile).

Automatic Mailbox creation

The application contributes a listener to the loginSuccess event. It allows for personal Mailbox creation upon successful user login, and possibly creation of other Mailboxes, eg: Mailboxes for the groups to which the user belongs to. An extension point allows to contribute an implementation for the Mailbox creation. The default implementation will create personal Mailboxes depending on the presence of a boolean system property.

It is assumed that once the user is logged in, all of the necessary Mailboxes are created.

As client projects may want to define specific ways to initialize Mailboxes, the code triggering automatic creation of personal Mailbox at user login, and the code code triggering update/or automatic creation of all Mailboxes is pluggable.

Default Mailbox tabs

The default Mailbox view has tabs that follow the standard Nuxeo tab system (using the VIEW_MAILBOX_ACTION_LIST category). Standard tab navigation can be used. 5 tabs are available in the *View* view:

1. Inbox: the list of CaseLinks received in this Mailbox
2. Service (only in generic Mailboxes): the list of CaseLinks received in this Mailbox and all the posts received in the personal Mailbox of affiliated user.
3. Draft: the list of CaseLinks that were not yet sent
4. Sent: the list of CaseLinks sent.
5. Manage: the administration view of the mailbox

Mailbox administration

The mailbox administration view shows 4 sub-tabs in the MANAGE_MAILBOX_ACTION_LIST category: Modification, Delegation, Classification Folder, Mailing List.

Mailbox editing

The Edit sub-tab is available under the Mailbox management tab. It provides the standard editing functionality along with filtering for information such as type or profile.

Mailbox removal

When a Mailbox is deleted, an `onMailboxDeleted` event is sent. A synchronous listener is contributed to perform the Mailbox deletion.

The default implementation of Mailbox removal will delete the Mailbox and its content (CaseLinks). To address issues such as how to transfer information when a generic or personal Mailbox is deleted, it is possible for an application to modify this behavior by contributing a listener to the `onMailboxDeleted` event.

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

Document and manipulation

Document creation

By default, Case and CaseItem documents are stored in a tree hierarchy inside the CaseRoot folder. The hierarchy is based on the date:

1. parent folder with the year (4digits)
2. child folder with the month (2 digits)
3. grand-child folder with the day (2digits)
4. the document

The hierarchy is created as needed.

When a Case is distributed, a CaseLink is created inside all participant Mailboxes (sender and receiver). Each CaseLink will have the participants information populated.

CaseItem, Case and CaseLink all have a participants property:

- CaseItem participants property defines the list of all the participants of the CaseItem, with the CaseItem possibly contained in several different Cases.
- Case participants property defines the list of all the participants of the Case, it is possibly that this Case has been distributed several times.
- CaseLink participants property defines all the participants of the Case corresponding to the CaseLink, this property is immutable.

Adapter creation

The Case, CaseItem, Mailbox and CaseLink are created through adapters. A document adapter factory is contributed for each one. A Case can be created using:

```
Case case = documentModel.getAdapter(Case.class);
```

This allows for overriding the implementation of the interface by overriding the contribution to the adapter. All the schema manipulations should be restricted to the implementation and not leak to other objects or services. The document model remains available via methods on the interfaces.

```
DocumentModel doc = case.getDocument();
```

It is possible to pass data to the adapter factory via the context map of the document model.

Document manipulation

Manipulation of Mailboxes, CaseItems and Cases is done via services. However, it would be too cumbersome to persist those objects via a service as only those objects know their structure and object tree. So each object (mail, post, message and mail item) offers a save method that allows to persist it.

Document manipulation should rely on the facet of a document, not on its type. Any document type can be a Case if it has the "Distributable" facet . A CaseItem is a document type that has both "Distributable" and "CaseGroupable" facets.

Document	Facet	Schemas
CaseItem	Distributable CaseGroupable	• distribution • case_item
Case	Distributable	• distribution
Mailbox	Mailbox	• mailbox • distribution
CaseLink	CaseLink	• distribution

Developers can extend the application by contributing their document types as long as they provide the proper adapter that would return an object implementing the corresponding interface and have the proper facet.

The distribution schema is the schema used to keep track of the recipient list ordered by type (action, information ...)

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

Case and CaseItem

It is possible to create an empty Case and to add CaseItems to it. But is not possible to create a CaseItem without being in a context of a Case. A CaseItem is always seen inside a Case.

The CaseItem interface has 2 methods (setDefaultCaseId, getDefaultCaseId method) to set and get the default Case in which the document will be viewed. The default implementation will set a Case id in its schema, it is however possible to override the getDefaultCaseId method to have a runtime computation.

CaseLink and Task

A CaseLink is the view of a Case inside a Mailbox. It carries the following information:

- creation date
- is read

A CaseLink can also carry the information of weather an action needs to be taken on the case. This is a task management feature. When an actionnable CaseLink is in a Mailbox, this means than anyone subscribed to this mailbox can perform the task. A CaseLink has the following task management information:

- isActionnable (this is a task or a simple CaseLink)
- the due date
- the operation chains to run in case of validation
- the operation chains to run in case of refusal
- a list of label/value set by the action creating the task. The Case document and CaseLink document will be passed to the operation chains
- automaticValidation, default is false, if true the action is validated on the date it is due
- label is the label of the task to be performed

Document deletion

Listeners are contributed to listen to the deletion of documents. They call classes contributed via extension points to implement the behavior needed when a document is deleted. The default behavior is:

- for CaseItem: When a Case item is deleted:
  - if a Case contains only this item, then the Case is deleted
  - if a Case contains this item with another document, the document entry is removed.
- for a Case: When a Case is deleted, the corresponding CaseLinks are deleted, and all CaseItems that are not referenced elsewhere are also deleted. If the deleted case was the default Case of a CaseItem, a new default Case is set on the item.

Service interface

The CaseDistributionService is used to retrieve sending information. It is also used to manage sending of documents and cases.

Main API

Distribution management

```
CaseLink sendCase(CoreSession session, CaseLink postRequest,
            boolean isInitial);
public CaseLink sendCase(CoreSession session, CaseLink postRequest,
            boolean isInitial, boolean actionable)
```

Application Packaging

The core application is based on the following modules:

- nuxeo-case-managmenent-**api**: The API of the application. It should be deployed on all servers in case of multi-machine deployment.
- nuxeo-case-managmenent-**core**: The core implementation. It should be deployed on the stateful server in case of multi-machine deployment.
- nuxeo-case-managmenent-**distribution**: The module that builds the application.
- nuxeo-case-managmenente-**facade**: The module to allow remote access to the correspondence service.
- nuxeo-case-managmenent-**lang**: The language pack. By default the supported languages are French and English.
- nuxeo-case-managmenent-**test**: This module is to be used by other modules for testing. It provides extensible test cases.
- nuxeo-case-managmenent-**web**: This module should be on the stateless server in case of multi-machine deployment. It provides the

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

xhtml, seam and jsf components.

Content Routing

The **content routing** add-on allows users to create, update, manage and start workflow on documents (and cases, for case management applications). This add-on is provided by default in Nuxeo CMF. However, it can be deployed independently from Nuxeo CMF and can be installed in any other Nuxeo distribution.

A **route** is a workflow, i.e. **a set of steps** that documents can go through.

A user starts by creating an empty route model, then adds steps to it. Steps can be processed sequentially or in parallel. After the route model is built, a manager will validate it. Users can then apply this route to documents by selecting it from the available validated routes and clicking a button to start the route.

In this section:

- Content Routing documents
- Route processing and lifecycle
- Creating Tasks
- Advanced Features

Content Routing documents

A typical use case of content routing is that a user creates a route by giving it a name and a description. Then he can add steps or groups of steps (step folders). A step is associated with a chain of operations. A step folder is a container that can be either serial (each step runs one after the other) or parallel (all steps in it are run at the same time).

Routes, steps and steps folders are persisted in Nuxeo using documents. In this section, we will see how to create steps:

- Step document type creation
  - What steps to create?
  - How to create a step?
    - Creating the document type
    - Adding the layout
- Adding actions to a step
  - Creating an automation chain
  - Binding automation chain and Step document type

Step document type creation

What steps to create?

When creating an application using routing, the first question to answer is: what kind of step should a user be able to create ? In other words, what actions can be run through the route. For the following section, we will take an example to illustrate the configuration necessary. Here, we allow users to choose from three different types of steps:

- A step that assigns a task to a user: the user will be able to validate or refuse the set of documents attached to the route.
- A step to publish the documents: this step will publish the set of documents to a section chosen by the user in a remote Nuxeo instance.
- A step that sends a mail with all documents in the route attached.

From this example, we note two important things:

- The first step creates a task for a user. However, the content routing module is **not** a task management module. A task manager is not provided as part of the module. It is the responsibility of the user of the module to create tasks where necessary.
- When a step is run, it can either return as completed or open. A publishing step would return as completed automatically as there is no user interaction. A task step would remain open until the task is completed. This creates a waiting state when the route is running, where we wait for outside input (user input).

How to create a step?

Once we know what steps we want, the next questions to answer are: what part of the step should be configured by the user running/creating the route? What part of the step should be configured by the admin running the server ? Of course, this depends entirely on your use case and steps.

For our example, we choose:

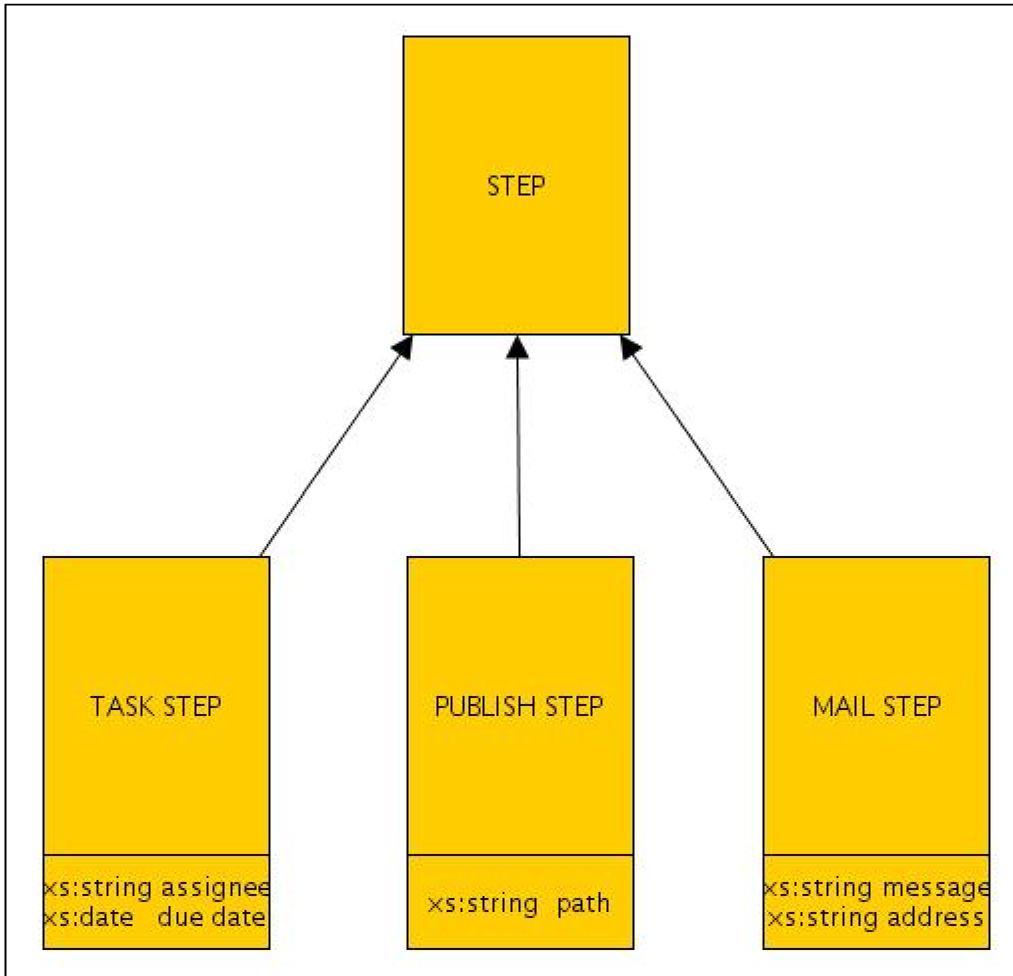
Step type	User configurable	Admin configurable
Task Step	• assignees • due date	• what happens when task is validated • what happens when task is refused
Publish Step	• In which section to publish the documents	• address of the remote Nuxeo instance
Mail Step	• email address of the recipients • message to add with the documents	

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

We will create a document type for each type of step and add a schema with fields for each user configurable data. The type needs to extend the `DocumentRouteStep` type.

Creating the document type

We create three document types with their own schema to add their specific metadata. The three new types extend the `DocumentRouteStep` document type. The `DocumentRouteStep` type is a very simple step that becomes done as soon as it is run.



You can look at Nuxeo EP documentation on how to create schema and document type. You can also have a look at the routing components and CMF components: the creation of step type in CMF shows the definition of 4 types of steps:

- `DistributionStep`
- `DistributionTask`
- `GenericDistributionTask`
- `PersonalDistributionTask`

They're all related and so use the same schema to persist their metadata.

Adding the layout

We add layouts to each type of step. As for any Nuxeo document type, we add Create, Edit and View layouts. When creating the layout, we should keep in mind that the person creating the steps and the person editing the steps often have very different functional roles. The Layout contribution for CMF shows an example of layout.

Adding actions to a step

Creating an automation chain

In the previous section we saw how to create a type of `DocumentRouteStep` to be able to persist data specific to our step. In this section we will see how to run a step. What we want to do depends on the step:

- For the task step, we want to get the assignee name, the due date and create a validation task for the assignee on the attached

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

document.
- For the publication step, we want to get the path of the section and connect to the remote Nuxeo application to publish the attached documents.
- For the mail step, we will get the message, the mail address, create a mail, attach the documents to it and send the mail.

For each of these steps, we will create an automation chain that will do the work. This automation chain will be provided, in its context, with the document attached to the route and the step document that is running. So, for example, the mail step automation chain will do:

1. get the address and message from the step document
2. get the attached document from the context
3. send the mail with the value collected

The Content Automation section will show you how to create operations and operation chains. You need to read it first if you plan to create operation manually. You can also use Nuxeo Studio.The routing module adds to the context the step document with key `document.routing.step`. It allows any operation run to access the `DocumentRouteStep` and then the route and the attached document. This the duty of the automation chain to notify the route if the step is done or not (in other word if we are in a waiting state or not). We provide the `setDone` method on the step to do it. We also provide ResumeStepOperation you can use in you chain. Its only function is to call the `setDone` method on the step attached to the chain.

Binding automation chain and Step document type

Once you created your automation chain to run the step, you finally bind together the document type for the step you created and the corresponding automation chain. This is done using the `chainsToType` contribution.
For each step document type, you need to contribute the operation chain that is called when the step is run. You also need to contribute an operation chain that will undo the step when it has already been done, and a third one that will undo the step when it is running.

RELATED TOPICS
Document types
Content Automation

Route processing and lifecycle

We have seen in the previous section how to create steps and define what will be done when the step is run. In this section we will see how a user creates a route and runs it.

Basically, when users create a route, they actually create a model of route, that will need to be validated in order to be used on documents. What is run on the documents is actually an instance of the route, that can be canceled. All these steps of the route evolution are explained below.

- Creating a route
    - Where can I create a route?
    - How can I validate the route?
    - What routes are available?
- Running a route
    - What does it mean to start a route?
    - What happens when we start a route on a set of documents?
    - Running the route
    - Note about lifecycle transition

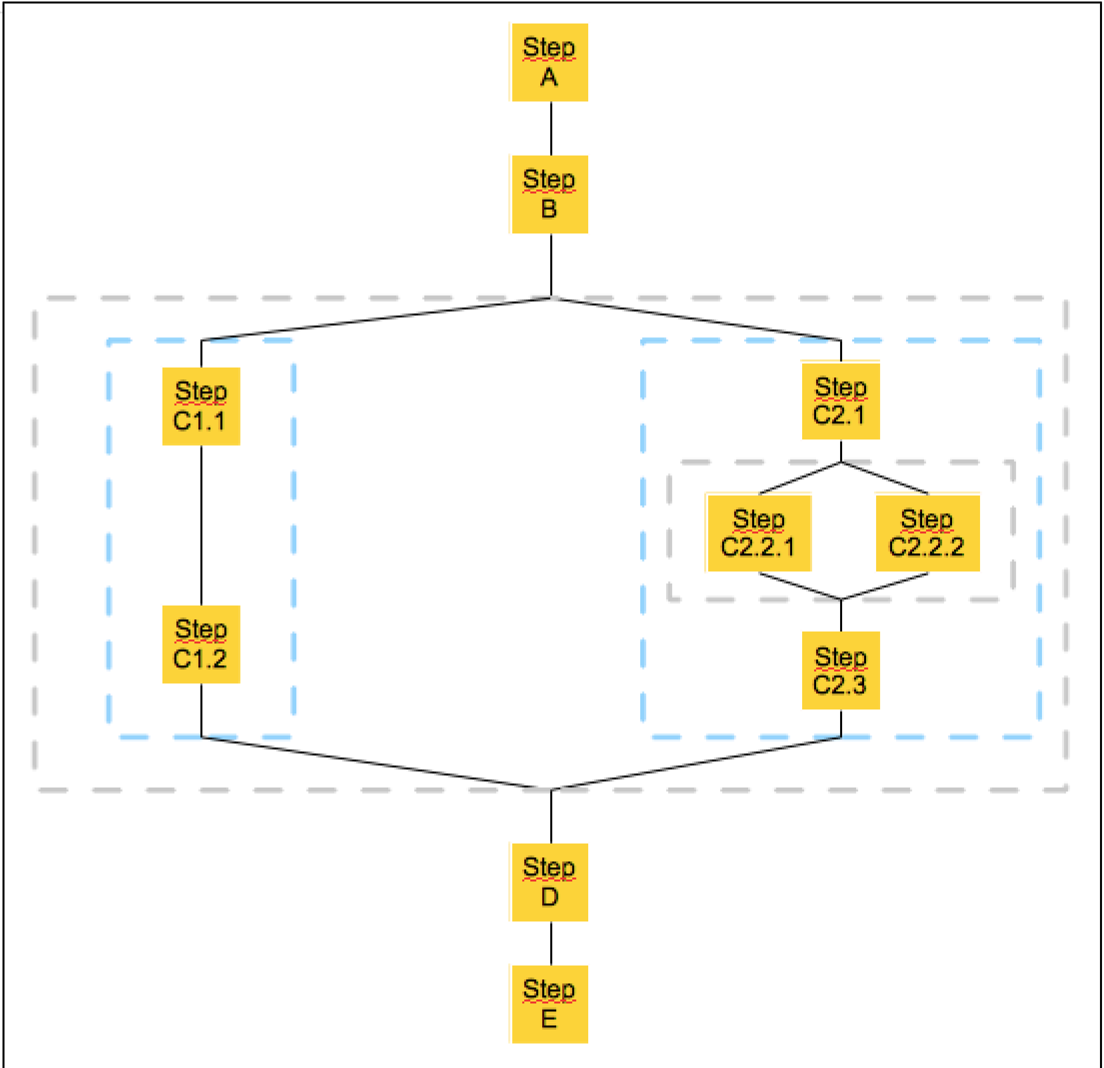
Creating a route

Where can I create a route?

A route is a document, called `DocumentRoute`. Via the user interface, the user will click on the on the "New" button. He will be able to select the `DocumentRoute` document type. He chooses if the route is serial or parallel. Default value is serial. He can then add folder (choosing again if they are serial or parallel) and steps. All the contributed step types are available to choose from. When creating the step document, he fills in the value using the layout contributed in the previous section.

In the default Nuxeo CMF implementation, this is the type service that decides where a user can create some types of document. The default routing configuration allows anyone to create a route inside a workspace or a folder. As users can only access workspaces and folders from their personal workspace in CMF, it is actually the only place where users can create routes.

When the user is done, he gets a hierarchy of Route elements like this workflow for instance:



In this example workflow, the route is composed of steps that can be grouped in parallel step folders (in grey) or serial step folders (in blue). This route is currently a model of route that users will be able to use on their documents. All route elements are at the "draft" state.

How can I validate the route?

The route we created is a model of route. This means that we didn't attach any document to it for processing. Before the route is made available to users so they can run it with attached documents, it needs to be validated. To validate the route, we call the method `validateRouteModel` on the `DocumentRoutingService`.

The default UI makes the "validate model" button only available to user who belongs to the `routeManagers` group. You need to create this group and add users to it. In the CMF, if you are not a route manager, you need to:

1. Go to to your personal space (this is the only place in CMF where a user can create non Case/CaseItem types of document).
2. Create the route.
3. Add to the permission read/write to a `routeManagers` on the route or one of its parent folder.
4. Inform route managers that the route needs validation.

Once a route is validated, it is immutable and all the elements of the route are at the 'validated' state. It also becomes readable by everyone.

What routes are available?

The method `getAvailableRouteModel` on the `DocumentRoutingService` returns a list of route models. This is a simple query done with the session of the user. If you need to make only some routes available to some users, you can override the query `DOC_ROUTING_SEARCH_ALL_ROUTE_MODELS` in the query model contribution. This query will return the list of route names in UI suggestion box for route.

Running a route

What does it mean to start a route?

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

When a user wants a set of documents to go through all the steps of a route, he starts the route on the document. In the UI, he can do it either from:

- the "Summary" tab of a document, selecting a route from the ones available and starting it;
- from the "Summary" tab of the route, selecting the document and starting the route.

From the Summary tab of the route, the suggestion widget offers documents found with the `CURRENT_DOC_ROUTING_SEARCH_ATTACHED_DOC` query model, that can be found in the query model contribution. This contribution can be overridden to fine graine the documents on which users can start a route.

What happens when we start a route on a set of documents?

When we start a route on a set of documents, the route model that we chose stays completely unmodified. The route that the user created and validated is unchanged. When we start a route:

1. A copy of the route hierarchy is created in a hidden folder at the root. We will call this copy an instance of the route. Administrators have access to this hierarchy: this is useful for debugging and finding problems.
2. The IDs of the attached documents are set on the route instance.
3. All the elements of the route are set to the ready state.

So it is actually the instance of the route that is run, and that can be modified. An instance always has attached documents. Modifications on the instance only affect this instance. A user might change an instance of a route by adding/editing/deleting steps. The method `saveRouteAsNewModel` in the `DocumentRoutingService` will enable users to save the modified instance in a new route model. It will copy the instance route in the user personal workspace, in the "draft" state only changing its name (adding `(COPY)` to its name).

The set of methods `createNewInstance` in the `DocumentRoutingService` allows to create an instance of a route with attached documents and start this instance.

Running the route

When a route starts, it calls the method `run` on each of its children (folder or steps). All elements do:

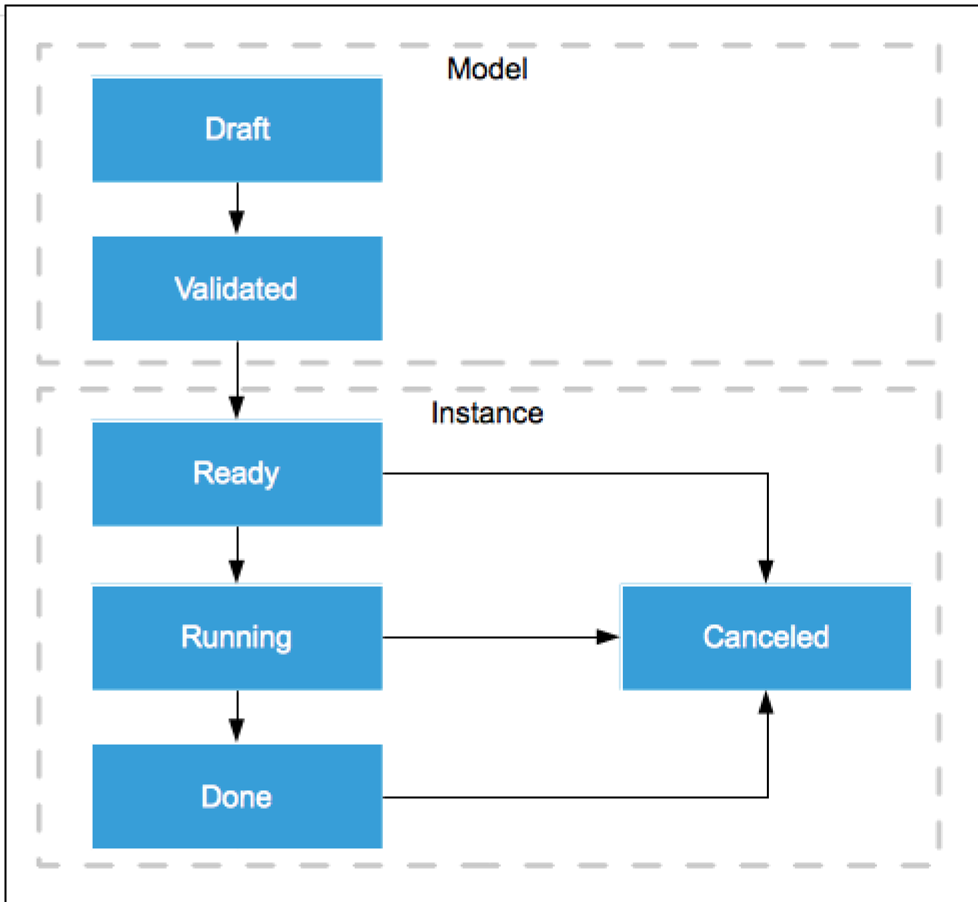
1. set itself to 'running',
2. run,
3. set itself to 'done' if needed.

The 'run' actions depend on the element it is applied to:

- steps will call the automation chains bound to the type of the step,
- folders will call the run method on their children and set themselves to done if their children are done.

We see that the lifecycle states of the elements of the route tell us how advanced we are in the processing of the route. The lifecycle is the same for all the elements (route, folder and step):

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



- **to Validated:** The document route is being validated. All the elements of the route become validated at the same time.
- **to Ready:** A new instance of the route is created in the `DocumentRouteInstances` folder. All the elements of this new route instance are at the ready state.
- **to Running:** The element is being run. Each element of the route reaches this state one after the other. The step is at the running state when the operation chain bound to it is being run.
- **to Done:** The element is done.
- **to Canceled:** When a route is canceled, all its element are canceled. The `undoFromRunning` and `undoFromDone` operation chain are run depending of the state of the canceled element.

For each transition taken, there is an event sent that you can use to plug some functionalities. Note the `stepWaiting` event, that is sent when the step runs the operation chain and returns still at the 'running' state. This means that the step is waiting for an outside event to resume. In this case, this is the responsibility of the outside event to resume the route.

Note about lifecycle transition

The `BulkLifeCycleChangeListener` in Nuxeo is a listener that recurses lifecycle change from a folder to its children. This is especially useful when deleting a document: when a folder is deleted (ie: lifecycle goes to deleted), we want all of its children to be deleted.

However, this is problematic for `DocumentRoute` documents as we don't want, when a route is set to 'Running', that all of its children are set to 'Running'. This is also problematic because the children is a post commit listener. This means that when the first step of a route goes from 'Validated' to 'Ready' to 'Running' in one go, it will complain that it is in 'Validated' state if we rely on the post commit listener to take the transition. To avoid this problem, this transition uses the non recursive attribute in their declaration.

RELATED TOPICS
Events and listeners
Nuxeo EP platform overview
Content Routing Documents

Creating Tasks

We have seen in Content Routing documents what a step document is and how to bind it to an operation chain. In this section, we will create a step that creates a task for a user.
How a task is persisted depends on the application. Here are two possible solutions:

- A Nuxeo DM application might choose to use jBPM to create and manage task.
- A Nuxeo DM application could create some Task document and persist it somewhere hidden in the repository.

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

CMF implementation of tasks is based on this second solution and uses a particular type of CaseLink to create a Task (ActionableCaseLink). We will walk through the steps required for task implementation in CMF.

- What is a task in Nuxeo CMF ?
- Creating the task step
  - Information on the step
  - Information on the task
  - Mapping between task and step
- Creating the operation chain that creates the task
- Running the step

What is a task in Nuxeo CMF ?

CMF is a framework that manages cases. A task in CMF is something to do on a case. A user sees a case in a mailbox via a case link that points to the case. We use this case link to materialize the task. A task is just an extended case link that will persist, on top of the case link metadata, the metadata related to the task. This particular type of case link is called an `ActionableCaseLink`.

To implement the task, we need at least to know:

- who should do the task,
- what should be done,
- what is the due date,
- what should happen when the due date is passed.

In CMF, the 'who' is a mailbox. A task is given to a mailbox, anyone who can access the mailbox can do the task.
The 'what should be done' value is selected from a drop-down list on the task (the values of the list being defined in cm_routing_task_type vocabulary).
The user will select the due date using a calendar widget.
To define what should happen when the due date is passed, we add an automatic validation field. If the due date is passed and the automatic validation is set to true, then the task is automatically validated. The `AutomaticActionCaseLinkValidatorListener` will find such tasks and validate them.

We will use a step to get the information from the user and an operation chain to automatically create the task.

Creating the task step

Information on the step

Now that we know what information are needed for the task, we need to get them from the user. This is done by enabling the user to create a task step, on which he will fill in the needed information.

1. We add the routing_task schema.
2. We create the GenericDistributionTask document type. It allows a user to create a task in a generic mailbox.

CMF includes three other types of step documents by default:

- `DistributionTask`, that allows a user to create a task for any mailbox;
- `PersonalDistributionTask`, that allows a user to create a task for a personal mailbox;
- `DistributionStep`, that distributes a case to a mailbox but without creating a task.

We add the ecm type for GenericDistributionTask and the generic_mailboxes_routing_task layout. This is 'classic' Nuxeo configuration. The distribution_mailbox widget is used for a user to select one mailbox. It is an interesting example on passing property (`mailboxSuggestionSearchType`) to the xhtml of the widget to allow better reuse of the same file.

Information on the task

The task, and therefore the `ActionableCaseLink`, will hold the same information as the step. It also need to keep the ID of the step so it can resume the route when the task is done. The `actionable_case_link` schema provides this information.

The task is created by an automation chain, when the step is run.

Mapping between task and step

ActionableCaseLink	Value taken from
dueDate	Step due date
label	Step label
documentId	Context: the ID of the document being processed. It is in the context of the operation.

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

stepId	Step: The id of the step, it will be used for the validation/refusal operation chain to resume the documentRoute
actionnable	Configuration: The value true/false is passed in the configuration of this operation
validate chain	Configuration: The value is passed in the configuration of this operation
refuse chain	Configuration: The value is passed in the configuration of this operation

Creating the operation chain that creates the task

When a route reaches a step, the operation chain of the step is run and the document is passed in the context. To create a task in CMF, we need to:

1. create an actionable case link,
2. set the value on the case link using the value on the step,
3. set the step ID value on the case link,
4. do the distribution using the newly created case link.

The chain is contributed using the chains contribution.

In the three task operation chains implemented by default, we use 3 operations. These operations pass values between each user by putting and object in the context of the chain. The key for those objects can be found in the `CaseConstants` class or in the `DocumentRoutingConstants`.

- The `CreateCaseLink` operation creates a new case link and puts it in the context.
- The Mapping operation sets the values of the case link using step information and operation parameter.
- The distribution operation distribute the case using the created case link.

Running the step

When the step is run, the task operation chain is run to create the task. The user can then see the task in his/her mailbox, and can validate or refuse the task. Validation and refusal can each have a different operation chain.
To help you implement this validate/refusal behavior, we provide some helper classes:

- `ActionableObject` enables to implement an interface for objects that can be validated or refused.
- `ActionableValidator` is a helper class that can validate or refuse an object and restart the route. It calls the appropriate operation chain, gets the step ID and sets the step to "Done", then resumes the route.

RELATED TOPICS
Document types
Content Automation
Content Routing documents
Route processing and lifecycle

Advanced Features

Adding metadata to the route

When running a route, we might want to use the result of an action of a step in another step. We can persist some metadata in the route so they can be used by all steps. A typical example would be:

1. first step is a validation step on a document by a user,
2. second step will publish this document in a section, if the document was approved.

The result of whether the document was validated or not needs to be persisted in the route document so it can be used by the second step. To add metadata to a route we need to:

1. add a schema with the added metadata,
2. create a new type of document that has the `DocumentRoute` facet and the `document_route_instance` and `document_route_model` model schemas.

When creating a new type of `DocumentRoute`, you should not forget to update the ecm type contribution so the new type of document is available to the user for creation.
You can have a look at the `DocumentRoute` contribution and `document_route_instance` and `document_route_model` model schemas as an example.

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

Content Routing security

When a user validates a Route, an instance is created in a restricted part of the repository. It is created with an unrestricted session (as Administrator). So afterward, if we want a user to validate or refuse a step or modify the route, we need to give him the rights to do it.

All steps of the route are always running using an unrestricted session, with the possible exception of setting the step to "Done state". If the step doesn't need user interaction, "Done" state will be set with an unrestricted session. If it needs the user to do an action, we use the user's sessions. So a user only needs Validate rights on the step to validate/refuse a task, even if the validation of the step leads to other steps to be validated.
Each step of the route will throw events that will allow to modify the security on the route and steps. This implementation provides for highly customizable security: you can disable a listener and create your own listener according to your security needs.

Because the Route is not aware of what happens when a step is run (it doesn't know what is inside the operation chain), it is the responsibility of the operation chains to modify the step if needed. Typically, if an operation chain assigns a task to a user, the operation chain should also give the correct validating rights to this user for the step.

The security of a route is based on the security of the documents. Rights on Route and Step are mapped to the rights on the underlying document:

Security of Document Routing	Security implementation	comment
Can create a Route Model	user can create a DocumentRoute document type	everyone should be able to do it. If you need to customize it, you can override the `typesTool` seam component. Look at CMF code for an example where only routeMangers can create a DocumentRoute
Validating a Route Model	users belonging to the group routeManagers	you can add filters to the action to modify who can access the button
Creating a Route instance	If a user can access a document and a route, he can run an instance	the group routeManagers can modify the rights on DocumentRoute to give rights only to people who can start instances of it
Validating a step	user can follow transition on the Step document	
Updating a step	user has write permissions on the Step document	
Deleting a step	user has delete permissions on the Step document	
Adding a step	user can add children on the parent document	

RELATED TOPICS
Document types
Events and Listeners
Content Automation
Security management in a Nuxeo Repository

Additional Functionalities

The CMF default platform will be using several modules to address different features. The main feature is to be able to send and receive documents. Other features are provided via addons and integrated in the default application.

In this section:

- Mailbox synchronization
- Classification (aka filing)
- Email capture
- File system import
    - Simple importer
    - Case Importer

Mailbox synchronization

Nuxeo CMF offers a simple way to synchronize a `Mailbox` (generic or personal) with a directory.

The following meta data are set on a `Mailbox` to assist in the synchronization process:

- `lastSyncUpdate`: the last time this `Mailbox` was synchronized
- `origin`: the name of the directory with which this `Mailbox` was last synchronized (default is the empty string for manual creation of

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

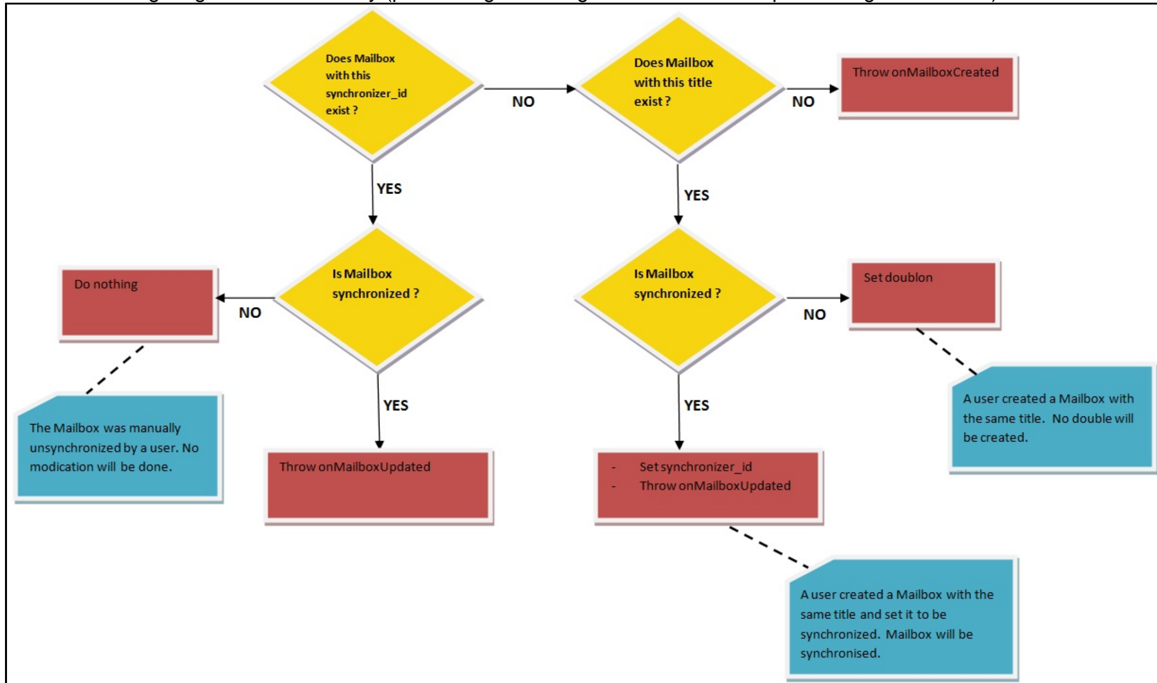
Mailbox)

- synchronizerId: the id given by the synchronization. The mailbox_id and the synchronizerId are different. For example, if a Mailbox is created by a synchronization process and then moved, the next synchronization process will create another Mailbox. Both will have the same synchronizerId because they were created from the same entry in the directory, however they will have different values for mailbox_id.
- synchronizedState: the state of this Mailbox, the state can be:
  - synchronized: this Mailbox is synchronized with a directory (see origin to find the directory name)
  - unsynchronized: this Mailbox is not synchronized. This is the default value (it may still have an value for origin if it was created from a synchronization process)
  - doublon: an entry in the directory exists for this Mailbox but, as this Mailbox was created manually in the application, no synchronization will occur.

The synchronization service provides a listener to the syncMailbox event. This listener will walk the directory and throw an event when a Mailbox is found to have been created, updated or deleted. It is your responsibility to provide the contribution that will throw the syncMailbox event. We recommend doing so during nightly processing or downtime.

You can can provide to the service a list of directories for synchronization. The default implementation will choose to synchronize the **user** and **group** directories. You will also need to provide a class that will create a Mailbox title from the directory entry. This is important as the title of a Mailbox is used to know if a user intended to create a Mailbox that is meant to be synchronized (see the "Does Mailbox with this title exists" node in the diagram).

The syncMailbox listener will:

1. Set a variable $dateT = now
2. For each directory (default group directory and user directory) :
   a. get all the entries at this level, add them to a queue Q
   b. run the following diagram for each entry (processing all siblings in the tree before processing the children)
   c. 

   d. get an item from the queue and run 2. for each of its children. Compare the child lastSyncUpdate property with $dateT. If they are equal, it means that this child has already been processed. This can happen for instance when processing the group directory: a group can be the subgroup of many groups.
3. Select all synchronized Mailboxes belonging to the directory and with lastSyncUpdate < $dateT, and for each throw a onCaseFolderDeleted event

The lastSyncUpdate and origin metadata are always updated before throwing an event.

When a directory is synchronized, 3 events are thrown:

- onMailboxUpdated event thrown in the context, the core session, the entry and the mailbox.
- onMailboxCreated event thrown in the context, the core session and the entry.
- onMailboxDeleted event thrown in the context, the core session and the entry.

CMF provides 3 listeners:

- an onMailboxUpdated listener that will update the title of the Mailbox when the origin of the mailbox equals the name of the directory. For the group directory, it checks the property org.nuxeo.nxcm.onGroupDirectoryUpdate in the file nxcm.properties in nuxeo.ear/config and if the value is:

    - `merge` (default value): then users field becomes the existing users + the users of the entry
    - `override`: then the new value is the list of users of the entry
    - `doNothing`: well, it does ... nothing
- an `onMailboxDeleted` listener that will put the `Mailbox` in the `deleted` state
- an `onMailboxCreated` listener that will create a `Mailbox` with:
    - `mailbox_id`: the ID of the entry
    - `type`: generic
    - `title`: if this the user directory `<family name> <name> (<organization>)`, if this is the group directory `<ID of the entry>`
    - `owner`: the owner entry of the entry if this is the user directory, empty if this is the group directory
    - `users`: empty for the user directory, member of the group if this is the group directory

This default implementation assumes that a `Mailbox/Entry` represents a group or a user. It will be added as a group of the `Mailbox` so every member of this group will be able to see this `Mailbox`. The group of the `Mailbox` will be added to the existing groups. You can provide your own listener if you need more customization. You can extend all these listeners in your own project.

Classification (aka filing)

Classification is available through addons. It allows to file documents in folders so that user can easily find them in their own classification folders.

Core types

Classification folders are required to have the “classification” schema (no need of another criterion like a facet to identify them).

Two types of classification folders are available by default: `ClassificationRoot` and `ClassificationFolder`. Any number of `ClassificationFolder` can be created under a `ClassificationRoot` or `ClassificationFolder`. Rights management is only available on the `ClassificationRoot`.

Classified documents are stored by referencing their document id in the property `classification:targets`. The document id is of the form `reponame:docid`. Versions of documents cannot be classified.

As these folders may need to hold specific metadata, only the code related to filing should be hardcoded. The rest of the configuration should be done using xml extension points.

Permissions

A new `Classify` permission should be available on `ClassificationFolder` document types. It should be checked to allow users to classify or unclassify the documents. As classification is done through writing to the document metadata, `Classify` should include the `WriteProperties` permission.

User interface

The list of `ClassificationRoot` documents the user has access to should be listed in a selector on the left side of the screen.

When classifying a document, the list of `ClassificationRoot` the user has the `Classify` permission on should be available.

A `ClassificationRoot` of the `ClassificationFolder` document has to list classified documents before its usual content. Users can only `Unclassify` these documents (no other action is allowed).

A classifiable document may hold a new tab presenting the list of classification folders or roots it is classified in, filtered by the `Read` access right.

Email capture

What is email injection ?

Email injection is the fetching of mail from an external mailbox into Nuxeo. It is based on `nuxeo-platform-mail`. The action contributed to the mail action pipe uses a service to allow easy contribution of elements to:

- parse the email message
- retrieve recipients
- inform

Having an action pipe configured, the email fetcher will automatically fetch emails from your email server and make it available to the pipe.

Configuration

The configuration is done in the nuxeo.conf file (default values will be used if a property is not set). The default configuration disables the fetching.

property name	default value

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

cm.mail.import.enable	false
cm.mail.import.server.user	nuxeo-correspondence@test.nuxeo.com
cm.mail.import.server.password	changeme
cm.mail.import.server.mail.store.protocol	imap
cm.mail.import.server.mail.imap.host	imap.gmail.com
cm.mail.import.server.mail.imap.starttls.enable	true
cm.mail.import.server.mail.imap.ssl.protocols	SSL
cm.mail.import.server.mail.imap.socketFactory.class	javax.net.ssl.SSLSocketFactory
cm.mail.import.server.mail.imap.port	993
cm.mail.import.server.mail.imap.socketFactory.port	993
cm.mail.import.server.mail.imap.socketFactory.fallback	false

File system import

In CMF, the user can create `Case` and `CaseItem` manually. The file system import allows for automatic import into NXCM files and documents. There are 2 ways to import:

- The simple importer will scan a directory. It will create a `Case` and a `CaseItem` for each file in a directory. This `Case` will be put in a configured `CaseFolder`.
- The case importer will scan a directory and read each xml file. The xml file will contain the necessary information to create and distribute a `Case` and `CaseItems`.

Simple importer

The simple importer uses the `nuxeo-platform-importer-core` to import files into NXCM. A FAQ is written on how to use it (How to use nuxeo-platform-importer). The different variables are read from the contribution to the `SimpleImporter service`. They are:

- path in the repository of the `CaseFolder` into which we import the files
- maximum number of threads to be used
- the factory to be used to create `Case` and `CaseItem`
- the filesystem path specifying the directory containing the file to import

The default contribution, `CaseItemDocumentFactory` only process files. It creates a `CaseItem` for each file and puts it in its own `Case`. You can extends this class if you need to a specific implementation. Once a file is imported, the suffix `_imported` is added to the filename. It is the responsibility of the administrator to remove those files from the directory.

This importer is started each time a `startSimpleImport` event is thrown. If an important volume of documents are to be imported, we recommend to throw this event during downtime.

Case Importer

`nuxeo-case-management-importer` is an application to import cases from the file system to Nuxeo. It uses RESTlet for the import. This application expects a folder in which it can find xml files corresponding to the metadata of documents to import. This xml file can reference other files within the same directory that will be the blobs of the document.

A default schema of the xml file is provided with a default class to read it. The developer can override this class to read different schemas. The default schema is:

```
<document>
  <schema name="dublincore"></schema>
  <schema name="correspondence_distribution"></schema>
  <mailEnvelopeItems>
    <path></path>
    <path></path>
    <path></path>
  </mailEnvelopeItems>
</document>
```

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

The `mailEnvelopItems` node references a list of paths. Each path entry points to a file that is an item of this envelope. It is not allowed to have envelope with no items inside it.

The schema nodes have a name attribute referencing the name of the schema. Each node under it will use the schema prefix and field. It has to be the same prefix declared in Nuxeo.

Any schema used in the Document returned by the adapter when calling for a `CorrespondenceEnvelope` can be used.

The import is rejected if:

- a file pointed to in `mailEnvelopeitem` is not found
- No recipients are filled
- There is no title field (default is `dc:title`)

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