Nuxeo Enterprise Platform - Version 5.1

The Reference guide

5.1

Table of Contents

Part I. Introduction

Chapter 1. Getting Started

1.1. Introduction

This chapter describes how to setup a development environment for working on Nuxeo EP. It currently deals with an Eclipse environment and will be completed soon with Netbeans and IDEA.

You will see how to easily start a new project on the basis of the *sample project* you can find at org.nuxeo.project.sample. The sample project already implements a few customizations, a new document type, a new action, a new event listener, etc. Play with this example to start learning Nuxeo. If you encounter a problem don't forget you can share your experience on the ecm mailing list.

1.2. Prerequisites

We assume from now on that on your computer, you have the following ready-to-use environment:

- Java Development Kit (JDK) 1.5.x (Nuxeo EP is know to work now with Java 6 or OpenJDK, but this is currently not supported)
- Apache Ant 1.7.0 (or later)
- Maven 2.0.8 (or later, *not Maven 2.0.7*)
- Eclipse 3.3
- A Nuxeo EP installation from the last release (download the installation wizard). You can also download a nightly build installer here but keep in mind that this one may be broken, so don't erase your working setup with a nightly build: the result is not guaranteed.

Note : The Nuxeo EP installer also installs and sets up the right version of JBoss on your computer, so you don't need to install it by your own.

If some of these software are not already set up on your computer, please refer to Appendix B, *Detailed Development Software Installation Instructions*, where you will find useful help.

1.3. Starting a new project

1.3.1. About Maven

Before starting, if for you *mvn* sounds like an other more boring acronym to learn, you should read these few lines, otherwise, you can skip it and go to the next step.

Maven is an IT project helper, that structures the development process in many ways:

- it helps you using always the same code organization, so that everybody in the company works the same way
- it allows you to apply in an easy and standard way many tools to your application and your application source: tools for testing, quality, deployment, documentation site generation, etc.
- it simplifies a lot the way you manage dependencies in your projects by using the pattern of public / local repository.

Depending on how deep you will go with Nuxeo, you may use part or all of the aforementioned items. But for now, we will have special attention for *repositories*, *pom* and *dependencies*. When you start a project under eclipse, you usually begin by downloading then adding all the libraries you will need, and put them all in a subdirectory of the project. You may have copied many times, like JUnit. And if you have been programming for a long time, you may have many versions every where in your projects. Maven is really useful for that. When you start a project, you create a `pom.xml` which will contains all the information Maven needs to act. Especially, it will contain the dependencies of your project. The dependencies are the libraries you usually add with the "add external jar" command under Eclipse. In Maven terminology, it is called an *artifact*. So you specify in `pom.xml` which artifacts you need and which version. At compilation time (of course with Maven) it will automatically find the jar corresponding to the artifacts specified, so that there are no "no class found" error.

How does Maven know where to find it?

Maven uses *repositories of artifacts* (libraries) One of the goals of Maven is to gather all of the code production in the same place, in company or public scoped repositories, so that everybody shares the same libraries and knows where to find them. There are the official an public Maven repositories, automatically inspected when using Maven, but you can also set up private repository like Nuxeo's.. You declare the repositories you want your Maven installation to look up into in `$HOME/.m2/settings.xml`. Maven will download the needed dependencies from those specified repositories into the `$HOME/.m2/repository` directory, that acts as a "cache".

Then, suppose you want to compile your project with Maven. Maven will read the *POM*, see that it needs, for instance the JUnit library. It will check if this library is present in the local Maven repository `$HOME/.m2` and if not, it will check in a remote one. For sure, it will look into Maven repository, and maybe some private remote repositories you might have set up in your `$HOME/.m2/settings.xml` file. At compilation time, if your code uses a JUnit `TestCase` class, it will know it has to go to the repository to download the dependency.

If you manage all your projects with Maven, you will earn a lot of time at the beginning of the project, and your deployment procedure will be more reliable. So Nuxeo is developed through a full Maven process. And we encourage people who wants to develop over Nuxeo to also use Maven.

To achieve this, when you start a new project with Nuxeo, you will have to create your own `pom.xml` that will reference Nuxeo libraries, for document manipulation, etc. A POM represents a project, a resource, and has three main characteristics: its name, its domain and its version. Then it has a lot of XML to specify everything you want. For a Nuxeo project, you have to reference Nuxeo artifacts. So starting a new project with all this structure is tedious and always the same.

To avoid doing the Maven packaging by hand, we prepared for Nuxeo platform users a *Maven archetype*, that acts as a template system: we defined the typical structure and POM of a Nuxeo Customization project and then, typing a command as short as `mvn archetype` and a few parameters, Maven will auto-generate the structure of your project, and you are sure it will be Nuxeo compliant.

You can also read the Wikipedia entry for Maven and the Maven archetype introduction for more information about archetypes.

1.3.2. Generate the sample project with nuxeo-archetype-simple archetype

This archetype will generate the project you can find at org.nuxeo.project.sample project.

To create a project named `my-project` in the `com.company.sandbox` area:

```
mvn org.apache.maven.plugins:maven-archetype-plugin:1.0-alpha-7:create \
    -DartifactId=my-project \
    -DgroupId=com.company.sandbox  \
    -DarchetypeArtifactId=nuxeo-archetype-simple   \
    -DarchetypeGroupId=org.nuxeo.archetypes   \
    -DarchetypeVersion=5.1.3.1 \
    -DremoteRepositories=http://archiva.nuxeo.org/archiva/repository/nuxeo_release
```

You can see there are six arguments you need to supply:

- *artifactId*: the name of your project, usually with '-' to separate the words if there are many.
- *groupId*: the domain name of your project. Usually the package parent name of your classes, you should use '.' to separate the words (the '-' is not supported here).

- *archetypeArtifactId*: the maven archetype artifact id. To generate a sample project, Nuxeo provides you nuxeo-archetype-simple
- *archetypeGroupId*: unique for all nuxeo maven archetypes : org.nuxeo.archetypes
- *archetypeVersion*: the version of the archetype which is equivalent to the version of Nuxeo EP without the *GA* or *RC* part. (5.1.0, 5.1.2, 5.1.3, 5.1.3.1, ...).
- *remoteRepositories*: the repository location to download the archetype.

Note

Note that for the moment we use the old maven archetype plugin (1.0-alpha-7) as the latest 2.0-alpha-1 is not working with remote repositories.

You should have the following source layout:

```
my-project
|-- build.properties.sample
|-- build.xml
|-- pom.xml
|-- settings.xml
`-- src
    `-- main
        |-- java
        |   `-- com
        |       `-- company
        |           `-- sandbox
        |               |-- BookEventListener.java
        |               |-- BookFileManagerPlugin.java
        |               |-- BookISBNEventListener.java
        |               |-- BookIntegerConverter.java
        |               |-- BookIntegerValidator.java
        |               |-- BookManager.java
        |               |-- BookManagerBean.java
        |               |-- BookResultsProvider.java
        |               |-- BookTitleDescriptor.java
        |               |-- BookTitleService.java
        |               `-- BookTitleServiceImpl.java
        `-- resources
            |-- META-INF
            |   |-- MANIFEST.MF
            |   `-- ejb-jar.xml
            |-- OSGI-INF
            |   |-- actions-contrib.xml
            |   |-- booktitle-contrib.xml
            |   |-- booktitle-service-contrib.xml
            |   |-- content-template-contrib.xml
            |   |-- core-types-contrib.xml
            |   |-- deployment-fragment.xml
            |   |-- directories-contrib.xml
            |   |-- event-listener-contrib.xml
            |   |-- filemanager-contrib.xml
            |   |-- l10n
            |   |   |-- messages.properties
            |   |   |-- messages_en.properties
            |   |   `-- messages_fr.properties
            |   |-- lifecycle-contrib.xml
            |   |-- querymodel-contrib.xml
            |   |-- resultsprovider-contrib.xml
            |   |-- search-contrib.xml
            |   |-- theme-contrib.xml
            |   `-- ui-types-contrib.xml
            |-- directories
            |   `-- book_keywords.csv
            |-- nuxeo.war
            |   |-- bookwizard.xhtml
            |   `-- incl
            |       |-- book_listing_fragment.xhtml
            |       |-- bookwizard_page1.xhtml
            |       |-- bookwizard_page2.xhtml
            |       |-- bookwizard_page3.xhtml
            |       `-- tabs
            |           |-- book_view.xhtml
            |           `-- folder_books_view.xhtml
            |-- schemas
            |   `-- book.xsd
            |-- seam.properties
            `-- themes
```

```
                `-- theme-book.xml
```

1.3.3. Maven and Ant settings

You need to setup Maven to access Nuxeo's repositories. This is done by adding (or modifying) a Nuxeo profile in your `$HOME/.m2/settings.xml` file. Your generated project contains a `settings.xml` file example.

You should create a `build.properties` file on the base of the `build.properties.sample` so that `jboss.dir` points to your jboss path. This will be used by Ant to send the packaged file in the good directory at deployment time. Note that when using the Nuxeo EP installer, the JBOSS home directory is the same as the Nuxeo EP home directory.

1.3.4. "Eclipsize" your project

If you try to import your project into eclipse right now, you will see many errors in the source. This is because Eclipse is unaware of the classpath. Indeed when you (used to ;-)) do the "add library" operation from Eclipse, it creates into the project directory a `.classpath` file. So Maven (thanks to a plugin it will auto install) generates this file for you, providing that you execute the following commands, at the root of your newly created project:

```
mvn install
mvn -Declipse.workspace=/the/path/to/your/workspace eclipse:add-maven-repo
mvn eclipse:eclipse
```

Now you can run Eclipse, and import the project from the file system as you are used to:

Then go back to the Java perspective and import the projects you have previously checked out as eclipse projects:

```
File > Import > General > Existing Projects Into Workspace
```

You can see on the left the Nuxeo project structure (directly inspired from Maven project structure) and you can browse the classes without errors.

If you have set the `downloadSources` property in your `$HOME/.m2/settings.xml` you can navigate in the wall Nuxeo EP sources.

Note: if you still have errors, it might be that the `M2_REPO` variable has not been set up correctly. This variable should point to your home Maven repository, generally here : `$HOME/.m2/repository`. To check, create or update this variable in Eclipse, right-click on your project, choose "properties". Then in the "Java Build Path" entry, click on "Add variable" (see here for detailed instructions).

1.3.5. Running your custom project

Now that you have created your new custom project, you would want to see what your (not so tough) work produced? As easy as one line of command!

On your project root folder (provided you created the right *build.properties*, as mentioned earlier):

```
ant deploy
```

This will run Maven, install the needed dependencies if needed, compile the code, do the packaging and copy it onto the server directory.

Run your server (**.../NUXEO_HOME/bin/run.sh** or .bat), go to `http://jboss_host:8080/nuxeo/`

Try to log with Admin signature (Administrator/Administrator) and to create a new document: if your project was correctly deployed, you should now see a "Book" document type.

Do you still find **mvn** as a boring command? :-)

1.4. Using Documentation

- This *Nuxeo Book* is getting to be the most complete source of information around Nuxeo EP, both for beginners and advanced developer. It is a good start.
- The extension point documentation is also very useful: although you may find it rough, it is the best way to evaluate the Nuxeo extensibility potential, and one should always start with a quick look around all the extension points, to "think Nuxeo" before starting a new project, and not reinventing the wheel.
- The wiki: we try to reference all the documentation from the wiki welcome page, and you will find tricks, howtos, etc. If you want to have a writer account to help update the content, ask on the Nuxeo's mailing list.

1.5. Advanced tools and tips

1.5.1. Useful plugins when developing on top of Nuxeo with Eclipse

1.5.1.1. The JBoss IDE plugins

Note

The JBoss IDE has been morphed into the JBoss Tools (aka Exadel Studio, aka Red Hat Developer Studio). We're going to evaluate this new offering to understand how it can fit into the Nuxeo development process.

The JBoss IDE offers many useful features when dealing with JEE 5 projects, for creating EJB, managing Hibernate, etc. Especially, the JBoss IDE packaging contains JBPM Designer. You would want to use JBPM designer for writing new workflow procedures to leverage the default embedded JBPM implementation of Nuxeo EP workflow abstraction layer.

Here are the references for the Eclipse Update Manager:

```
Name: EMF Update Manager Site
URL: http://download.eclipse.org/modeling/emf/updates/
```

```
Name: JBoss
IDE URL: http://download.jboss.org/jbosside/updates/development
```

The second site has dependencies on the first one: use the "Select required" to sectionencies automatically when selecting JBossIDE.

1.5.2. Remote debugging in Eclipse

Use case: suppose you just deployed Nuxeo EP on your JBoss server and run into a new crash or any unexpected behavior. Instead of stopping the server and rerunning inside Eclipse to reproduce the bug in the debugger, you can setup a new remote debugging profile for Eclipse.

- Linux: edit `/opt/jboss/bin/run.conf`, add the following line at the end of the file and restart JBoss.

```
JAVA_OPTS="$JAVA_OPTS -Xdebug -Xrunjdwp:transport=dt_socket,address=8787,server=y,suspend=n"
```

- Windows: edit `JBOSS/bin/run.bat`, modify the line:

```
rem set JAVA_OPTS=-Xdebug -Xrunjdwp:transport=dt_socket,address=8787,server=y,suspend=y %JAVA_OPTS%
```

  by erasing the remark ('rem'), and disabling the suspend ('suspend=n'). You should have:

```
set JAVA_OPTS=-Xdebug -Xrunjdwp:transport=dt_socket,address=8787,server=y,suspend=n %JAVA_OPTS%
```

  Restart JBoss.

Then from Eclipse create a new debugger profile in menu Run > Debug create a new *Remote Java Application* profile and set the source to the filesystem directory that host your complete Eclipse workspace and set the port to 8787.

Then edit the offending source code file and set breakpoints and run your new remote debug profile. From your web browser, refresh the page that was crashing and look back into Eclipse: the debug perspective should open automatically at your fist breakpoint.

1.5.3. Browsing a JCR repository

Many Eclipse plugins all low you to browse JCR compliant repositories (the very back-end of Nuxeo uses, as a default storage system, the JackRabbit JCR reference implementation repository. We recommend you to use:

TODO: complete

1.6. Other IDEs: IntelliJ IDEA and NetBeans

1.6.1. IDEA

IntelliJ IDEA from Jetbrains is a very lovable IDE for Java that has many fans in the Java developers community. It is unfortunately not open source.

To start using IDEA for coding on the Nuxeo project, you just need to type **mvn idea:idea** from your top-level source directory: Maven will download all the dependencies, then generate the configuration files needed by IDEA and you will be able open `nuxeo-ecm.ipr` from IDEA.

Note

At the time of this writing, IDEA will report some (spurious) compilation failures and you won't be able to compile the whole application from IDEA. You can still use IDEA with the current configuration to write Java code ("with pleasure"), and use Ant and/or Maven to build/deploy the application.

1.6.2. NetBeans

NetBeans is an open source IDE from Sun.

If you're using NetBeans 6 or later, you will be able to download a Maven 2 support plugin from NetBean's "Update Center", and with it, start coding on Nuxeo EP very quickly.

More detailed instructions will be written soon.

Chapter 2. General Overview

2.1. Introduction

2.1.1. Architecture Goals

When we started building Nuxeo EP, we defined several goals to achieve. Because these goals have a structural impact on Nuxeo EP platform it is important to understand them: it helps understanding the logic behind the platform.

2.1.1.1. Flexible deployment on multiple targets

An ECM platform like Nuxeo EP can be used in a lot of different cases.

The deployment of Nuxeo EP must be adapted to all these different cases:

- Standard ECM web application

  This is the most standard use case. The web browser is used to navigate in content repositories.

  - All services are deployed on an Application server

  - In order to be easily hostable, the platform need to be compatible with several application servers

- Complex edition or rich media manipulation

  In this case having a rich client that seamlessly communicates with other desktop applications and offers a rich GUI is more comfortable than a simple web browser.

  - Interface is deployed on a rich client on the user desktop

  - Services and storage are handled on the server

- Offline usage

  In some cases, it is useful to be able to manipulate and contribute content without needing a network connection.

  - GUI and some services (like a local repository) need to be deployed on the client side

  - Server hosts the collaborative services (like the workflow) and the central repository

- Distributed architecture

  In order to be able to address the needs of large decentralized organizations, Nuxeo EP must provide a way to be deployed on several servers on several locations

  - Scale out on several servers

  - Dedicate servers to some specific services

  - Have one unique Web application accessing several decentralized repositories

- Use Nuxeo EP components from another application

  When building an business application, it can be useful to integrate services from Nuxeo EP in order to address all content oriented needs of the application.

  - Provide web service API to access generic ECM services (including repository)

  - Provide EJB3 remoting API to access generic ECM services (including repository)
  - Provide POJO API to generic ECM services

There are certainly a lot of other use cases, but mainly the constraints are:

- Be able to choose the deployment platform: POJO vs Java EE

  As first deployment targets we choose

  - Eclipse RCP: a rich client solution that uses a POJO (OSGi) component model
  - JBoss Application Server: a Java EE 5 compliant application server

- Be able to choose the deployment location of each component: client side vs server side

  The idea is to be able to deploy a component on the server side or on the client side without having to change its code or its packaging

2.1.1.2. Leverage CPS experience

Before building Nuxeo EP we worked during several years on the Zope platform with the CPS solution. CPS was deployed for a lot different use cases and we learned a lot of good practices and design patterns. Even if Nuxeo EP is a full rewrite of our ECM platform, we want to keep as much as possible of CPS good concepts.

- Concept of schemas and documents

  Inside CPS most of the data manipulated was represented by a document object with a structure based on schemas.

  This concept is very interesting:

  - Schemas enforce structure constraints and data integrity but also permit some flexibility.

    When defining a schema you can specify what fields are compulsory, what are their data type, but you can also define some flexible part of the schema.

  - Share API and UI components for Documents, Users, Records ...

    Because the Document/Schema model is very flexible it can be used to manipulate different types of data: like Users, Records and standards documents.

    From the developer's perspective this permit using the same API and be able to reuse some UI components

    From the user's perspective it give the application some consistency: because the same features and GUI can be used for all the data he manipulates.

- Actions and Views

  Because CPS was very pluggable, it was possible to easily define different views for each document type and also to let additional components contribute new actions or views on existing documents.

  Nuxeo EP has a similar concept of views and actions, even if technically speaking the technologies are different.

- Lazy fetching and caching

  Because ECM applications make a very intensive use of the repository and often need to fetch a lot of different documents to construct each page, the way the document retrieval is handled if very important to have a scalable application.

With CPS with worked a lot on caching and lazy fetching.

With Nuxeo EP we incorporated this requirement from the beginning:

- Distributed caching
- Lazy fetching and schemas and fields

2.1.1.3. Extensible platform based on components

CPS was constructed as a set of pluggable components relying on a common platform. This modularity has been maintained in the new platform. Deploying a new feature or component on the new platform is as simple as it was on the old one.

This requirement as a huge impact on the platform because the Java packaging model and the Java EE constraints are not directly compatible with it.

Adding a new component should be as simple as dropping a file or an archive in some directory without having to rebuild nor repackage the application.

This is important from the administrator point of view: be able to easily deploy new features.

This is also very important from the support point of view: be able to deploy customized components without taking the risk of forking the platform and maintain the possibility to upgrade the standards components.

2.1.1.4. Easily accessible development framework

The CPS framework was powerful but we know it was very complex to use. Not only because of the unusual CMF/Zope/Python programming model, but also because there was a lot of different concepts and you had to understand them all to be able to leverage the platform when building a new application on top of it.

Nuxeo EP aims at simplifying the task of the developer

- Clearly separate each layer

  The idea is to clearly separate presentation, processing and storage so that developers can concentrate on their task.

- Offer plugin API and SPI

  Nuxeo EP is constructed as a set of plugins so you can modify the behavior of the application by just contributing a new plugin. This is simpler because for common tasks we will offer a simple plugin API and the developer just has to implement the given interface without having to understand each part of the platform.

- Rely on JAVA standards

  We try to follow as much as possible all the Java standards when they are applicable. This will allow experienced Java developers to quickly contribute to the Nuxeo EP platform.

2.1.1.5. Leverage Java open source community

We know what it's like to have to build and maintain an entire framework starting from the application server. With the switch to the Java technology, we will use as much as possible existing open source components and focus on integrating them seamlessly in the ECM platform. Nuxeo EP is a complete integrated solution for building an ECM application, but Nuxeo won't write all infrastructure components. This approach will also make the platform more standards compliant.

Thus developers can optimize their Java/JEE and open source experience to use Nuxeo EP.

2.1.1.6. Make the platform ready for SI integration

Because ECM applications often need to be deeply integrated into the existing SI, Nuxeo EP will be easily integrable

- API for each reusable service or component

  Depending on the components, this API could be POJO, EJB3, or WebService, and in most cases it will be available in the three formats.

- Pluggable hooks into Nuxeo EP

  This mainly means synchronous or asynchronous events listener that are a great place to handle communication and synchronization between applications.

2.1.1.7. Future-proof design

The Nuxeo EP platform was rewritten from the ground with the switch to Java. But we don't plan to do this kind of work every couple of years, it wont be efficient neither for us, nor for the users of the platform. For that reason, we choose innovative Java technologies like OSGi, EJB3, JSF, Seam ....

2.1.2. Main concepts and design

All the design goals explained just before have a huge impact on the Nuxeo EP architecture. Before going into more details, here are the main concepts of Nuxeo EP architecture.

2.1.2.1. Layered architecture

Nuxeo EP is built of several layers, following at least the 3 tiers standard architecture

- Presentation layer

  Handles GUI interactions (in HTML, SWT ...)

- Service layer

  Service stack that offers all generic ECM services like workflow, relations, annotations, record management...

- Storage layer

  Handles all storage-oriented services like document storage, versioning, life cycle ....

Depending on the components, their complexity and the needed pluggability, there can be more that 3 layers.

This layering of all the components brings Nuxeo EP the following advantages

- Choose the deployment target for each part of a component

  By separating clearly the different parts of a feature, you can choose what part to deploy on the client and what part to deploy on a server.

- Clear API separation

  Each layer will provide its own API stack

- Components are easier to reuse

  Because the service and storage layers are not bound to a GUI, they are more generic and then more reusable

Thanks to this separation in component families you can easily extract from Nuxeo EP the components you need for your application.

If you need to include Document storage facilities into your application you can just use Nuxeo EP Core: It will offer you all the needed feature to store, version and retrieve documents (or any structured but flexible dataset). If you also need process management and workflow you can also use Nuxeo EP Workflow service. And finally, if you want to have a Web application to browse and manage your data, you can reuse the Nuxeo EP Web layer.

2.1.2.2. Deployment services

The targeted platform do not provide the same mechanism to handle all the deployment tasks:

- Packaging (Java EE vs OSGi)
- Dependency management
- Extension management

Because of these differences, Nuxeo EP provides a unified deployment service that hides the specificity of the target platform. This is also a way to add a pluggable component deployment system to some platform that don't handle this (like Java EE).

This is one of the motivation for the Nuxeo Runtime that will be quickly introduce later in this document.

2.1.2.3. Extensible component model

In Nuxeo EP, an ECM application is seen as an assembly of components.

This assembly will include:

- Existing generic Nuxeo EP Components
- Extensions or configurations contributing to generic Nuxeo EP components
- Specific components and configuration

Inside Nuxeo EP each feature is implemented by a one or several reusable components and services. A feature may be implemented completely at storage level, or may require a dedicated service and a dedicated GUI.

Nuxeo EP Web application is a default distribution of a set of ECM components. This can be used "as is" or can be the base for making a business ECM application.

- If you need to remove a feature

  Just remove the component or deploy a configuration for disabling it.
- If you need to change the default behavior of one component

  You can deploy a new configuration for the component .

  - Declare a new Schema or define a document type
  - Configure the versioning policy
  - Deploy new workflow
  - ...

  This configuration may use an extension point to contribute the new behavior.

  - Contribute a new security policy
  - Contribute a new event handler
  - Deploy a new View on a document
  - ...

- If you need to add a completely new feature you can make your own component.

  First check that there is no generic Nuxeo EP component available that could help you in your task (all components are not deployed in the default webapp).

2.1.2.4. Use of innovative Java EE technology

Here is a quick list of the Java technology we use inside Nuxeo EP platform:

- Java 5
- Java EE 5: JSF and EJB3
- OSGi component model
- A lot a innovative open source projects
  - JBoss Seam, Trinidad and Ajax4JSF on the web layer
  - JBPM for the default workflow engine implementation
  - Lucene for the default search engine implementation
  - JackRabbit JSR-170 repository for the default storage back end implementation
  - JenaRDF for the relation framework
  - ...

2.2. Nuxeo Runtime: the Nuxeo EP component model

2.2.1. The motivations for the runtime layer

Building the Nuxeo Runtime was one of the first task we started. This is one of the main infrastructure component or Nuxeo EP architecture.

This paragraph will give you a quick overview of the Nuxeo Runtime, a more detailed technical presentation can be found in an other chapter of this book.

2.2.1.1. Host platform transparency

Because most of Nuxeo EP components are shared by Nuxeo RCP (OSGI/RCP) and Nuxeo EP (Java EE), an abstraction layer is required so the components can use transparently the components services independently from the underlying infrastructure.

Nuxeo Runtime provides an abstraction layer on top of the target host platform. Depending on the target host platform, this Runtime layer may be very thin.

Nuxeo Runtime already supports Equinox (Eclipse RCP OSGi layer) and JBoss 4.x (JMX). The port of Nuxeo Runtime to other Java EE application server is in progress, we already have a part of Nuxeo EP components that can be deployed on top of SUN GlassFish application server. Technically speaking, the port of Nuxeo Runtime could be done on any JEE5 compliant platform and will be almost straightforward for any platform that supports natively the OSGi component model.

2.2.1.2. Overcome Java EE model limitations

Java EE is a great standard, but it was not designed for a component based framework: it is not modular at all.

- Java EE deployment model limitations
    - Most Java EE deployment descriptors are monolithic

      For example, the web.xml descriptor is a unique XML file. If you want to deploy an additional component that needs to declare a new Java module you are stuck. You have to make one version of the web.xml for your custom configuration. For Nuxeo EP platform, this constraint is not possible:

        - Components don't know each other

          Because there are a lot of optional components, we can't have a fixed configuration that fits all.

        - We can make a version of the web.xml for each possible distribution

          There are too many optional components to build one static web.xml for each possible combination.

      This problem with the web.xml is of course also true for a lot of standard descriptors (application.xml, faces-config.xml, persistence.xml, ejb-jar.xml ....)

    - One archive for one web application

      We have here the exact same problem than with the web.xml. additional components can contribute new web pages, new web components ... We can have a monolithic web archive.

    - No dependency declaration

      Inside Java EE there is no standard way to declare the dependency between components.

      Because Nuxeo EP is extensible and has a plugin model, we need that feature. A contribution is dependent on the component it contribute to:

        - Contribution is only activated if/when the target component is activated
        - The contribution must be deployed after the target component as it may override some configuration

- Java EE component model limitations
    - Unable to deploy a new component without rebuilding the whole package

      If you take a .ear archive and want to add a new component, you have to rebuild a new ear.

    - No support for versionned components

Nuxeo Runtime provides an extensible component model that supports all these feature. It also handles the deployment of these components on the target host platform.

2.2.2. Extensible component model

Nuxeo Runtime provides the component model for the platform.

This component model is heavily based on OSGi and provides the following features:

- Platform agnostic component model

Can be deployed on POJO and Java EE platforms

- Supports dependencies management

  Components explicitly declare their requirements and are deployed and activated by respecting the inferred dependency chain.

- Includes a plugin model

  To let you easily configure and contribute to deployed components

- A POJO test environment

  Nuxeo Runtime components can be unit tested using JUnit without the need of a specific container.

2.2.2.1. The OSGi component model

OSGi (Open Services Gateway initiative) is a great standard for components based Java architecture.

OSGi provides out of the box the following features:

- Dependencies declaration and management

  A component gets activated only when the needed requirements are fulfilled

- Modular deployment system

  - Manage bundles
  - Manage fragments (sub parts of a master bundle)
  - an OSGi bundle can define one or several services

- A system to identify and lookup for a component

For Nuxeo EP, OSGi standard provides a lot of the needed features. This is the reason why Nuxeo Runtime is based on OSGi, in fact Nuxeo Runtime component model is a subset of OSGi specification.

To ensure platform transparency, Nuxeo Runtime provides adapters for each target platform to help it support OSGi components.

- This adopter layer is very thin on Equinox (Eclipse RCP) since the underlying platform is already OSGi compliant.

- This adapter may be more complex for platform that are not aware of OSGi (JBoss 4.x or GlassFish)

  In this case, the runtime adapter will handle all OSGi logic and deploy the components as native platform components. For examples, on JBoss 4.x, Runtime components are deployed as JMX MBeans.

2.2.2.2. Extension points

OSGi does not define a plugin model, but the Eclipse implementation (Equinox) does provide an extension point system.

Because we used a lot the Eclipse Extension Point system and we liked it, Nuxeo Runtime also includes an Extension Point system.

Basically every Nuxeo Component can:

- declare its dependencies

The component will also be activated after all needed components

- declare exposed extension points

  Each components can define extension points that other components can use to contribute configuration or code.

- declare contribution to other components

These declarations are handled by the OSGi deployment descriptor (MANIFEST.MF)

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Nuxeo ECM Core
Bundle-SymbolicName: org.nuxeo.ecm.core;singleton:=true
Bundle-Version: 1.0.0
Bundle-Vendor: Nuxeo
Bundle-Localization: bundle
Bundle-Activator: org.nuxeo.ecm.core.NXCoreActivator
Bundle-ClassPath: ., lib/xsom.jar,
  lib/connector-api.jar,
  lib/java-cup-v11a.jar
Export-Package: org.nuxeo.ecm.core,
  org.nuxeo.ecm.core.api,
  org.nuxeo.ecm.core.api.local,
  org.nuxeo.ecm.core.jca,
  org.nuxeo.ecm.core.lifecycle,
  org.nuxeo.ecm.core.model
Require-Bundle: org.nuxeo.ecm.core.api,
  org.nuxeo.runtime
Nuxeo-Component: OSGI-INF/CoreService.xml,
  OSGI-INF/TypeService.xml,
  OSGI-INF/RepositoryService.xml,
  OSGI-INF/CoreExtensions.xml,
  OSGI-INF/SecurityService.xml
```

For example, this descriptor defines that

- this bundle depends on `org.nuxeo.ecm.core.api`
- this bundles contains Nuxeo components like `CoreServices.xml`

  The XML descriptor will be used to define new extension points or contribute to existing one.

An extension point is a way to declare that your component can be customized from the outside:

- Contribute configuration

  Activate or deactivate a component. Define resources for a given service.

- Contribute code and behavior

  Extension points also give you the possibility to register plugins

Extension points and contribution to extension points are defined using a XML descriptor that has to be referenced in the `MANIFEST.MF`.

Here is a simple descriptor example:

```
<component name="org.nuxeo.ecm.core.listener.CoreEventListenerService">
  <require>org.nuxeo.ecm.core.repository.RepositoryService</require>
  <implementation class="org.nuxeo.ecm.core.listener.impl.CoreEventListenerServiceImpl"/>

  <extension-point name="listener">
    <object class="org.nuxeo.ecm.core.listener.extensions.CoreEventListenerDescriptor"/>
  </extension-point>
  <extension target="org.nuxeo.ecm.core.listener.CoreEventListenerService" point="listener">
    <listener name="nxruntimelistener" class="org.nuxeo.ecm.core.listener.impl.NXRuntimeEventListener" />
  </extension>
  <extension target="org.nuxeo.ecm.core.listener.CoreEventListenerService" point="listener">
```

```
    <listener name="lifecyclelistener" class="org.nuxeo.ecm.core.lifecycle.impl.LifeCycleListener" />
  </extension>
</component>
```

- This fragment depends on the Repository Service

  This fragment won't be loaded until a Nuxeo Repository is setup

- This fragment declares an extension point named listener

  This extension point let register plugins that will be invoked when a core event occurs.

  This extension point use `CoreEventListenerDescriptor` for descriptor.

- This fragment registers two contributions to the listener extension point

The contributions have to follow the descriptor defined by the target Extension Point. The descriptor defines what tags can be used when contributing.

The descriptor is simply defined by a Java class that uses annotations to defines how the XML descriptor will be used to create an Object Descriptor instance to pass to the extension point registration.

2.2.3. Flexible deployment system

Nuxeo Runtime also provides deployment services to manage how components are deployed and contribute to each other

- Dependencies management

  The dependencies are declared in the `MANIFEST.MF` and can also be defined in XML descriptors that hold contributions.

  The Nuxeo Runtime orders the component deployment in order to be sure the dependencies are respected. Components that have unresolved dependencies are simply not deployed

- Extension point contributions

  XML descriptors are referenced in the `MANIFEST.MF`. These descriptors make contributions to existing extension points or declare new extension points.

- Each component has its own deployment-fragment

  The deployment fragment defines

  - Contribution to configuration files

    For example contribute a navigation rule to `faces-config.xml` or a module declaration to `web.xml`.

    Nuxeo Runtime let you declare template files (like `web.xml`, `persistence.xml`) and let other component contribute to these files.

  - Installation instructions

    Some resources contributions (like i18n files or web pages) need more complex installation instructions because they need archives and files manipulations. Nuxeo Runtime provide basic commands to define how your components should be deployed

Here is a simple example of a deployment-fragment.

```
<fragment>
  <extension target="application#MODULE">
    <module> <ejb>${bundle.fileName}</ejb> </module>
  </extension>
  <extension target="faces-config#VALIDATOR">
    <validator>
      <validator-id>dueDateValidator</validator-id>
      <validator-class>org.nuxeo.ecm.platform.workflow.web.ui.jsf.DueDateValidator</validator-class>
    </validator>
  </extension>
  <install>
    <!-- unzip the war template -->
    <unzip from="${bundle.fileName}" to="/">
      <include>nuxeo.war/**</include>
    </unzip>
    <!-- create a temp dir -->
    <!-- be sure no directory with that name exists -->
    <delete path="nxworkflow-client.tmp" />
    <mkdir path="nxworkflow-client.tmp" />
    <unzip from="${bundle.fileName}" to="nxworkflow-client.tmp">
      <include>OSGI-INF/l10n/**</include>
    </unzip>
    <append from="nxworkflow-client.tmp/OSGI-INF/l10n/messages.properties"
            to="nuxeo.war/WEB-INF/classes/messages.properties" addNewLine="true" />
    <append from="nxworkflow-client.tmp/OSGI-INF/l10n/messages_en.properties"
            to="nuxeo.war/WEB-INF/classes/messages_en.properties" addNewLine="true" />
    <append from="nxworkflow-client.tmp/OSGI-INF/l10n/messages_fr.properties"
            to="nuxeo.war/WEB-INF/classes/messages_fr.properties" addNewLine="true" />
    <append from="nxworkflow-client.tmp/OSGI-INF/l10n/messages_de.properties"
            to="nuxeo.war/WEB-INF/classes/messages_de.properties" addNewLine="true" />
    <append from="nxworkflow-client.tmp/OSGI-INF/l10n/messages_it.properties"
            to="nuxeo.war/WEB-INF/classes/messages_it.properties" addNewLine="true" />
    <delete path="nxworkflow-client.tmp" />
  </install>
</fragment>
```

2.2.4. Extension points and Nuxeo 5

2.2.4.1. Some examples of extension point usage

Inside Nuxeo 5, extension points are used each time a behavior or a component needs to be configurable or pluggable.

Here are some examples of extension points used inside the Nuxeo 5 platform.

- Schemas and document types

  Inside Nuxeo 5 a document structure is defined by a set of XSD schemas. Schemas and Document Types are defined using an extension point.

- Storage repository

  Nuxeo core stores documents according to their type but independently of the low level storage back-end. The default back-end is JackRabbit JCR implementation. Nuxeo Core exposes an extension point to define the storage back-end. We are working on an other repository implementation that will be pure SQL based.

- Security Management

  Nuxeo include a security manager that checks access rights on each single operation. The extension point system allow to have different implementation of the security manager depending on the project requirements:

  - Enforce data integrity: store security descriptors directly inside the document

  - Performance: store security descriptors in a externalized index

  - Corporate security policy: implement a specific security manager that will enforce business rules

- Event handlers

  Nuxeo platform let you define custom Event handler for a very large variety of events related to content or processes. The event handler extension mechanism gives a powerful way to add new behavior to an existing application

- You can modify the behavior of the application without changing it's code
- The development model is easy because you have a very simple Interface to implement and you can use Nuxeo Core API to manipulate the data
- Event handlers can be synchronous or asynchronous

Nuxeo 5 itself uses the Event Handler system for a lot of generic and configurable service

- Automatic versioning: create a new version when a document is modified according to business rules
- Meta-data automatic update: update contributor lists, last modification date ...
- Meta-data extraction / synchronization: extract Meta-Data from MS-Office file, Picture ....

2.2.4.2. Nuxeo 5 Platform development model

Nuxeo 5 development model is heavily based on the usage of extensions point. When a project requires specific features we try as much of possible to include it as an extension of the existing framework rather than writting separated specific component. This means make existing services more generic and more configurable and implement the project specific needs as a configuration or a plugin of a generic component using Extension Points.

2.3. Nuxeo EP layered architecture

2.3.1. Layers in Nuxeo EP

Nuxeo EP components are separated in 3 main layers: Core / Service / UI

From the **logical point of view** each layer is a group of components that provide the same nature of service:

- Storage oriented services: Nuxeo Core

  Nuxeo core provides all the storage services for managing documents

  - Repository service
  - Versioning service
  - Security service
  - Lifecycle service
  - Records storage (directories)
  - ...
- Content and process oriented services: Nuxeo Platform

  Nuxeo provides a stack of generic services that handles documents and provide content and process management features. Depending on the project requirement only a part of the existing services can be deployed.

  Typical Nuxeo EP platform services are:

  - Workflow management service

  - Relation management service
  - Archive management service
  - Notification service
  - ...
- Presentation service: UI Layer

  The UI layer is responsible for providing presentation services like

  - Displaying a view of a document
  - Displaying available actions according to context
  - Managing page flow on a process driven operation

  These services can be very generic (like the action manager) but can also be directly tied to a type of client (like the View generation can be bound to JSF/facelets for the web implementation)

The layer organization can also be seen as a **deployment strategy**

Thanks to the Nuxeo Runtime remoting features it is very easy to split the components on several JVM. But splitting some services can have a very bad effect on the global system performance.

Because of that, all the storage oriented services are inside the core. All components that have extensive usage of the repository and need multiple synchronous interaction with it are located in the core. This is especially true for all synchronous event handlers.

The services layer can itself been split in multiple deployment unit on multiple JVMs.

On the UI side all the services are logically deployed inside the same JVM. At least each JVM must have the minimum set of services to handle user interaction for the given application.

The components are also grouped by layers according to their **dependencies**.

Core Modules can depend on Core Internal API.

Generic ECM services can depend on Core external API and can depend on external optional library (like jBPM, Jena, OpenOffice.org ...).

UI services can rely on a client side API (like Servlet API) and share a common state associated to the user session.

Layers are also organized according to **deployment target**.

The Core layer is a POJO Layer with an optional EJB facade. The core can be embed in a client application.

The services are mostly implemented as POJO services so that they can be used as an embedded library. But some of them can be depend on typical Application Server infrastructure (like JMS or EJB).

Inside the UI Layer most service are dedicated to a target platform: web (JSF/Seam), Eclipse RCP or other.

Because the layer organization has several constraints, the implementation of a unique feature is spread across several layers.

Typically a lot of transversal services are split in several sub-components in each layer in order to comply to deployment constraint and also to provide better reusability. For example, the Audit service is made of 3 main parts:

- Core Event <=> JMS bridge (Core Layer)

  Forwards core events to JMS Bridge according to configuration.

- JMS Listener and JPA Logger (Service Layer)

  Message driven bean that write logs in DB via JPA.

- Audit View (UI Layer)

  Generates HTML fragment that display all events that occurred on a document.

2.3.2. API and Packaging impacts

The layer organization can also be seen in the API.

2.3.2.1. Core API

Most of the components forming the core are exposed via the `DocumentManager` / `CoreSession` interface. The interfaces and dependencies needed to access the Core services are packaged in a API package: even if there are several Core component, you have only one dependency and API package.

The idea is that for accessing the core, you will only need to use the `DocumentManager` to manipulate `DocumentModel`s (the document object artifact). Some core services can be directly accessed via the `DocumentModel` (like the life cycle or security data).

2.3.2.2. Service Stack API

Each service exposes it's own API and then has its own API package. Service related data (like process data, relation data) are not directly hosted by the DocumentModel object but can be associated to it via adapters and facets.

2.3.2.3. UI API

The web layer can be very specific to the target application. Nuxeo EP provides a default web application and a set of base classes, utility classes and pluggable services to handle web navigation inside the content repository.

2.3.2.4. Packaging

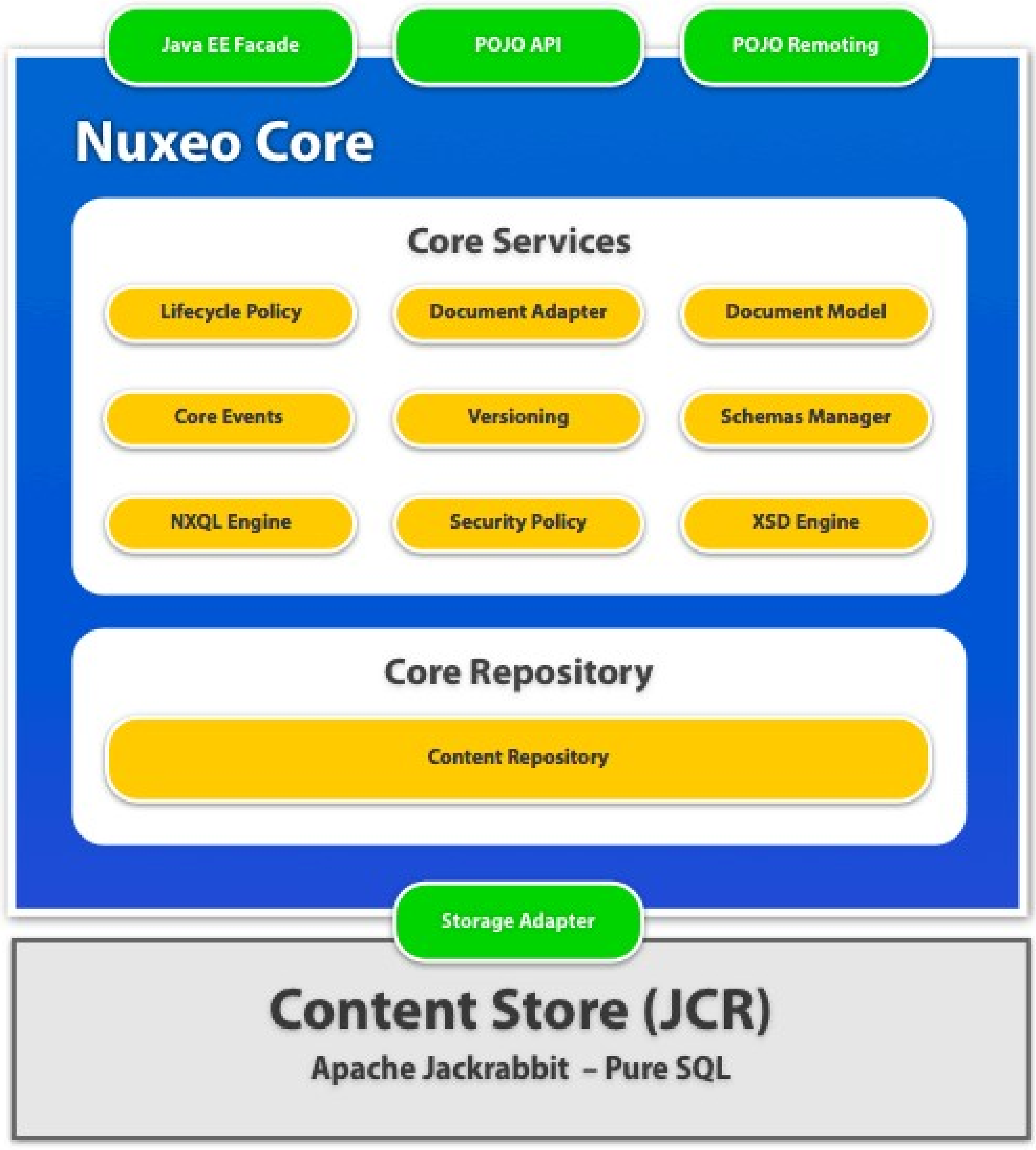
Most features are made of several Java project and generate several Maven 2 artifact.

Nuxeo packaging and deployment system (Nuxeo Runtime, Platform API, Maven ...) leverage this separation to help you distributing the needed deployment unit according to your target physical platform.

2.3.3. Illustration of the layered architecture

XXX TODO

2.4. Core Layer overview



2.4.1. Features of Nuxeo Core

Nuxeo core provides all the storage services for managing documents

- Schema service

  Let you register XSD schemas and document types based on schemas.

- Repository service

  Let you define one or more repository for storing your documents.

- Versioning service

Let you configure how to store versions.

- Security service

  Manage data level security checks

- Lifecycle service

  Manage life cycle state of each document

2.4.2. Nuxeo Core main modules

2.4.2.1. Nuxeo Repository Service

The repository service let you define new document repositories. Defining separated repositories for your documents is pretty much like defining separated Databases for your records.

Because Nuxeo Core defines a SPI on repository, you can configure how you want the repository to be implemented. For now, default implementation uses JSR-170 (Java Content Repository) reference implementation: Apache Jack Rabbit. In the future, we may provide other implementation of the Repository SPI (like native SQL DB or Object Oriented DB).

Even if for now there is only one Repository implementation available, using JCR implementation, you can configure how your data will be persisted: filesystem, xml or SQL Database. Please see "How to"s about repository configuration.

When defining a new repository, you can configure:

- The name.
- The configuration file

  For JCR, it lets you define persistence manager.

- The security manager

  Defines how security descriptors are stored in the repository (for now: org.nuxeo.ecm.core.repository.jcr.JCRSecurityManager)

- The repository factory

  Defines how the repository is created (for now: org.nuxeo.ecm.core.repository.jcr.JCRRepositoryFactory)

2.4.3. Schemas and document types

The repository enforces data integrity and consistency based on Document types definition.

Each document type is defined by:

- A name.
- An optional super document type (inheritance)
- A list of XSD schemas

  Defines storage structure

- A list of facets

  Simple markers used to define document behavior.

Here is a simple DocumentType declaration:

```
<extension target="org.nuxeo.ecm.core.schema.TypeService"
    point="doctype">
    <documentation>The core document types</documentation>
    <doctype name="Folder" extends="Document">
      <schema name="common" />
      <schema name="dublincore" />
      <facet name="Folderish" />
    </doctype>
    </extension>
```

For further explanation on Schemas and Document types, please see the dedicated section in this document.

2.4.4. Life cycle associated to documents

Inside Nuxeo repository each document may be associated with a life-cycle. The life-cycle defines the states a document may have and the possible transitions between these states. Here we are not talking about workflow or process, we just define the possibles states of a document inside the system.

The Nuxeo Core contains a LifeCycleManager service that exposes several extension points:

- one for contribution Life-Cycle management engine

  (default one is called JCRLifeCycleManager and stores life-cycle related information directly inside the JSR 170 repository)

- one for contributin life-cycle definition

  This includes states and transitions.

```
    <lifecycle name="default" lifecyclemanager="jcrlifecyclemanager"
      initial="project">
      <transitions>
        <transition name="approve" destinationState="approved">
          <description>Approve the content</description>
        </transition>
        <transition name="obsolete" destinationState="obsolete">
          <description>Content becomes obsolete</description>
        </transition>
        <transition name="delete" destinationState="deleted">
          <description>Move document to trash (temporary delete)</description>
        </transition>
        <transition name="undelete" destinationState="project">
          <description>Recover the document from trash</description>
        </transition>
        <transition name="backToProject" destinationState="project">
          <description>Recover the document from trash</description>
        </transition>
      </transitions>
      <states>
        <state name="project" description="Default state">
          <transitions>
            <transition>approve</transition>
            <transition>obsolete</transition>
            <transition>delete</transition>
          </transitions>
        </state>
        <state name="approved" description="Content has been validated">
          <transitions>
            <transition>delete</transition>
            <transition>backToProject</transition>
          </transitions>
        </state>
        <state name="obsolete" description="Content is obsolete">
          <transitions>
            <transition>delete</transition>
            <transition>backToProject</transition>
          </transitions>
        </state>
        <state name="deleted" description="Document is deleted">
          <transitions>
            <transition>undelete</transition>
          </transitions>
        </state>
      </states>
    </lifecycle>
  </extension>
```

* one for binding life-cycle to document-types

  Here is an example

```
    <lifecycle name="default" lifecyclemanager="jcrlifecyclemanager"
      initial="project">
      <transitions>
        <transition name="approve" destinationState="approved">
          <description>Approve the content</description>
        </transition>
        <transition name="obsolete" destinationState="obsolete">
          <description>Content becomes obsolete</description>
        </transition>
        <transition name="delete" destinationState="deleted">
          <description>Move document to trash (temporary delete)</description>
        </transition>
        <transition name="undelete" destinationState="project">
          <description>Recover the document from trash</description>
        </transition>
        <transition name="backToProject" destinationState="project">
          <description>Recover the document from trash</description>
        </transition>
      </transitions>
      <states>
        <state name="project" description="Default state">
          <transitions>
            <transition>approve</transition>
            <transition>obsolete</transition>
            <transition>delete</transition>
          </transitions>
        </state>
        <state name="approved" description="Content has been validated">
          <transitions>
            <transition>delete</transition>
            <transition>backToProject</transition>
          </transitions>
        </state>
        <state name="obsolete" description="Content is obsolete">
          <transitions>
            <transition>delete</transition>
            <transition>backToProject</transition>
          </transitions>
        </state>
        <state name="deleted" description="Document is deleted">
          <transitions>
            <transition>undelete</transition>
          </transitions>
        </state>
      </states>
    </lifecycle>
  </extension>
```

Life-Cycle service is detailed later in this document.

2.4.5. Security model

Inside Nuxeo Repository security is always checked when accessing a document.

Nuxeo security model includes :

* Permissions

  (Read, Write, AddChildren, ...).

  Permissions management is hierarchical (there are groups of permissions)

* ACE: Access Control Entry

  An ACE grants or denies a permission to a user or a group of users.

* ACL: Access Control List

An ACL is a list of ACE.

- ACP: Access Control Policy

  An ACP is a stack of ACL. We use ACP because security can be bound to multiples rules: there can be a static ACL, an ACL that is driven by the workflow, and another one that is driven by business rules.

  Separating ACLs allows to easily reset the ACP when a process or a rules does not apply any more.

Inside the repository each single document can have an ACP. By default security descriptors are inherited from parent, but inheritance can be blocked when needed.

Security engine also let you contribute custom policy services so that security management can include business rules.

Security model and policy service are described in details later in this document.

2.4.6. Core events system

When an event happens inside the repository (document creation, document modification, etc...), an event is sent to the event service that dispatches the notification to its listeners. Listeners can perform whatever action when receiving an event, this includes modifying the document on the fly.

As an example, part of the dublin-core management logic is implemented as a CoreEvent listener: whenever a document is created or modified, creation date, modification date, author and contributors fields are automatically updated by a CoreEvent Listener.

Core Events system is explained in more details later in this document.

2.4.7. Query system

The Repository support a Query API to extract Documents using a SQL like query.

NXQL (the associated Query Language) is presented later in this document.

2.4.8. Versioning system

The documents in the repository can be versionned.

Nuxeo Core provides:

- A pluggable version storage manager

  This let you define how versions and stored and what operations can be done on versions

- A pluggable versionning policy

  This let you define rules and logic that drives when new versions must be created and how versions numbers are incremented.

The versionning system is explained in details later in this document.

2.4.9. Repository and SPI Model

Nuxeo Core exposes a repository API on top of JackRabbit JSR170 repository.

Nuxeo repository is implemented using a SPI and extension point model: this basically means that a non JCR based repository plugin can be contributed. In fact, we have already started a native SQL repository implementation (that is not yet finished because we have no direct requirement for such a repository).

Nuxeo core can server several repository: it provides a extension point to declare additional repository: this means a single web application can use several document repository.

2.4.10. DocumentModel

Inside Nuxeo EP and especially inside the Core API the main data object is a Document.

Inside Nuxeo Core API, the object artifact used to represent a Document is called a DocumentModel.

The DocumentModel artifact encapsulates several useful features:

- Data Access over the network

  the DocumentModel encapsulate all access to Document internal fields, the DocumentModel can be sent over the network

- DocumentModel support lazy loading

  When fetched from the Core, a DocumentModel does not carries all document related information. Some data (called prefetch data) are always present, other data will be loaded (locally or remotely) from the core when needed.

  This feature is very important to reduce network and disk I/O when manipulating Document that contains a lot of big blob files (like video, music, images ...) .

- DocumentModel uses Core Streaming Service

  For files above 1 MB the DocumentModel uses the Core Streaming service.

- DocumentModel carries the security descriptors

  ACE/ACL/ACP are embedded inside the DocumentModel

- DocumentModels supports a adapter service

  In addition of the data oriented interface, a DocumentModel can be associated with one or several Adapters that will expose a business oriented interface.

- DocumentModels embeds lifeCycle service access

- DocumentModels can have facets

  Facets are used to declare a behavior (Versionnable, HiddenInNavigation, Commentable ...)

A DocumentModel can be located inside the repository using a DocumentRef. DocumentRef can be an IdRef (UUID in the case of the JCR Repository Implementation) or PathRef (absolute path).

DocumentModels also holds information about the Type of the Document and a set of flags to define some useful characteristics of the Document :

- isFolder

  Defines if the targeted document is a container

- isVersion

  Defines if the targeted document is an historic version

- isProxy

Defines if the targeted document is a Proxy (see below)

2.4.11. Proxies

A Proxy is a DocumentModel that points to a another one: very much like a symbolic link between 2 files.

Proxies are used when the same document must be accessible from several locations (paths) in the repository. This is typically the case when doing publishing: the same document can be visible in several sections. In order to avoid duplicating the data, we use proxies that point to same document.

A proxy can point to a checked out document (not yet associated to a version label) or to a versionned version (typical use-case of the publishing).

The proxy does not store document data: all data access are forwarded to the source document. But the Proxy does holds :

- it's own security descriptors
- it's own lifecycle information
- it's own DocumentRef

2.4.12. Core API

2.5. Service Layer overview

2.5.1. Role of services in Nuxeo EP architecture

The service layer is an ECM services stack on top of the Nuxeo Repository. In a sense, the Repository itself is very much like any service of this layer, it just plays a central role.

This service layer is used for :

- adding new generic ECM services (Workflow, Relations, Audit ...)
- adding connectors to existing external services
- adding dedicated projects specific components when the requirements can not be integrated into a generic component

This service layer provides the API used by client applications (Webapp or RCP based application) to do their work.

This means that in this layer, services don't care about UI, navigation or pageflows: they simply explode an API to achieve document oriented tasks.

2.5.2. Services implementation patterns

Nuxeo platform provides a lot of different services, but they all fellow the same implementation pattern. This basically means that once you understand how works one service, you almost know how they all work.

As everything in the Nuxeo Platform the services use the Nuxeo Runtime component model.

A generic service will be composed of the following packages :

- A API package (usually called nuxeo-platform-XXX-api)

  Contains all interfaces and remotable objects.

  This package is required to be able to call the service from the same JVM or from a remote JVM.

- A POJO Runtime component (usually called nuxeo-platform-XXX-core)

  The Runtime component will implement the service business logic (ie: implement the API) and also expose Extensions Points.

  All the extensibility and pluggability is handled at runtime component level. This for example means that the API can be partially implemented via plugins.

- A EJB3 Facade (usually called nuxeo-platform-XXX-facade)

  The facade exposes the same API as the POJO component.

  The target of this facade is :

  - provide EJB3 remoting access to the API
  - integrate the service into JEE managed environment (JTA and JAAS)
  - leverage some additional features of the application server like JMS and Message Driven Bean
  - provide state management via Stateful Session Beans when needed

Typically, the POJO module will be a Nuxeo Runtime Component that inherit from DefaultComponent, provide extension points and implement a Service Interface.

```
public class RelationService extends DefaultComponent implements
        RelationManager { ...}
```

The deployment descriptor associated to the component will register the component, declare it as service provider and it may also declare extension points

```
<?xml version="1.0"?>
<component name="org.nuxeo.ecm.platform.relations.services.RelationService">
  <implementation class="org.nuxeo.ecm.platform.relations.services.RelationService" />
  <service>
    <provide interface="org.nuxeo.ecm.platform.relations.api.RelationManager" />
  </service>
  <!-- declare here extension points -->
</component>
```

The facade will declare a EJB that implement the same service interface. In simple cases, the implementation simply delegates calls to the POJO component.

The facade package will also contain a contribution to the Runtime Service management to declare the service implementation.

```
<?xml version="1.0" encoding="UTF-8"?>
<component name="org.nuxeo.ecm.platform.relation.service.binding.contrib">
  <extension target="org.nuxeo.runtime.api.ServiceManagement" point="services">
      <documentation> Define the Relation bean as a platform service. </documentation>
      <service class="org.nuxeo.ecm.platform.relations.api.RelationManager" group="platform/relations">
        <locator>%RelationManagerBean</locator>
      </service>
    </extension>
</component>
```

Thanks to this declaration the POJO and the EJB Facade can now be used for providing the same interface based on a configuration of the framework and not on the client code.

This configuration is used when deploying Nuxeo components on several servers: platform configuration provides a way to define service groups and to bind them on physical servers.

2.5.3. Platform API

Each service provides it's own API composed of a main interface and of the other interfaces and types that can be accessed.

The API package is unique for a given service, access to a remote EJB3 based service is the same as accessing the POJO service.

From the client point of view, accessing a service is very simple and independent from service location and implementation: this means not manual JNDI call. Everything is encapsulated in the Framework.getService runtime API.

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

The `framework.getService` will return the interface of the required service:

- This can be the POJO service (ie: Runtime Component based Service)
- This can be the local interface of the EJB3 service (using call by ref in JBoss)
- This can be the remote interface of the EJB3 service (using full network marshaling)

The choice of the implementation to return is left to the Nuxeo Runtime that will take the decision based on the platform configuration.

The client can explicitly ask for the POJO service via the Framework.getLocalService() API: this is typically used in the EJB Facade to delegate calls to the POJO implementation.

2.5.4. Adapters

DocumentModel adapters are a way to adapt the DocumentModel interface (that is purely data oriented) to a more business logic oriented interface.

In the pure Service logic, adding a comment to a document would look like this:

```
CommentManager cm = Framework.getService(CommentManager.class);
cm.createComment(doc,"my comment");
List<DocumentModel> comments = cm.getComments(doc);
```

DocumentModel adapter give the possibility to have a more object oriented API:

```
CommentableDoc cDoc = doc.getAdapter(CommentableDoc);
cDoc.addComment("my comment");
List<DocumentModel> comments = cDoc.getComments();
```

The difference may seem small, but documentModel adapter can be very handy

- to have a more clean and clear code
- to handle to caching at DocumentModel level
- to implement behavior and logic associating a Document and a Service

DocumenModelAdapters are declared using an extension point that defines the interface provided by the adapter and the factory class. DocumentModelAdapters can be associated to a Facet of the DocumentModel.

2.5.5. Some examples of Nuxeo EP services

2.6. Web presentation layer overview

2.6.1. Technology choices

2.6.1.1. Requirements

The requirements for the Nuxeo Web Framework are :

- A Powerful templating system that supports composition
- A modern MVC model that provides Widgets, Validators and Controllers
- A standard framework
- A set of Widgets libraries that allow reusing existing components
- Support for AJAX integration

2.6.1.2. The JSF/Facelets/Seam choice

Nuxeo Web Layer uses JSF (SUN RI 1.2) and Facelets as presentation layer: JSF is standard and very pluggable, Facelets is much more flexible and adapted to JSF than JSP.

NXThemes provides a flexible Theme and composition engine based on JSF and Facelets.

In the 5.1 version of the platform, Nuxeo Web Layer uses Apache Tomahawk and trinidad as components library and AJAX4JSF for Ajax integration. In the 5.2 version we will move to Rich Faces.

Nuxeo Web Layer also uses Seam Web Framework to handle all the ActionListeners.

Using Seam provides :

- Simplifications and helpers on JSF usage
- A context management framework
- Dependency injection
- Remoting to access ActionsListeners from JS
- A built-in event system

2.6.2. Componentized web application

2.6.2.1. Requirements

Nuxeo Web Layer comes on top of a set of pluggable service.

Because this stack of services is very modular, so must be the web layer.

This basically mean that depending on the set of deployed services and on the configuration the web framework must provide a way

- to add, remove or customize views

for example, if you don't need relations, you may want to remove the relations tab that is by default available on document

- to add or remove a action button or link

  the typical use case is removing actions that are bound to non deployed services or add new actions that are specific to your project

- to override an action listener

  you may want to change how some actions are handled by just overriding Nuxeo defaults

- to add or customize forms

  Adding fields or customizing forms used to display document is very usefull

In order to fullfill these requirements, the key points of Nuxeo Web Framework are

- Context management to let components share some state
- Event system and dependency injection to let loosely coupled component collaborate
- A deployment system to let several components make one unique WebApp
- A set of pluggable services to configure the web application

2.6.2.2. Context management

Inside the web framework, each component will need to know at least

- what is the current navigation context

  This includes current document, current container, current Workspace, current Domain.

  This information is necessary because most of the service will display a view on the current document, and can fetch some configuration from the content hierarchy.

- who is the current user

  This includes identity and roles, but also it's preferences and the set of documents he choose to work on

In some cases, this context information may be huge, and it's time consuming to recompute all this information at each request.

Inside Nuxeo Web Framework, Seam context management is used to store these data. Depending on the lifecycle of the data Session, Conversation or Event context are used.

2.6.2.3. Loosely coupled component

At some point the components of the web framework need to interact with each other. But because components can be present or not depending on the deployment scenario, they can't call each other directly.

For that matter, the Nuxeo Web Framework uses a lot of Seam features:

- Seam's context is used to share some state between the components
- Seam's event system is used to let components notify each other
- Seam's dependency injection and Factory system is used to let component pull some data from each other without having to know each other

In order to facilitate Nuxeo Services integration into the web framework, we use the Seam Unwrap pattern to wrap Nuxeo Service into Seam Components that can be injected and manipulated as a standard Seam component.

2.6.2.4. Deployment

The Web Layer is composed of several components.

The main components are webapp-core (default webapp base) and ui-web (web framework). On top of these base components dedicated web components are deployed for each specific service.

For example, the workflow-service has it's own web components package, so do relation-service, audit-service, comment-service and so on.

Each package contains a set of views, actions, and ActionsListeners that are dedicated to one service and integrate this service into the base webapp.

Because JEE standards require the webapp to be mono-bloc, we use the Nuxeo Runtime deployment service to assemble the target webapp at deployment time.

This deployment framework let you: override resources, contribute XML descriptors like `web.xml` from several components and manage deployment order.

2.6.2.5. Key web framework services

Chapter 3. Schemas and Documents

3.1. Introduction

This chapter presents the concepts of schemas and document types, which are used to define new documents in Nuxeo EP 5.

3.1.1. Concepts

In Nuxeo EP 5, 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 the grouping of 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.

To create a new document type, we start by creating one ore more schemas that the document type will use.

The schema is defined in a `.xsd` file (which obeys the standard XML Schema syntax) and is registered by a contribution to a `schema` extension point.

The document type is registered through a contribution to another `doctype` extension point, and will specify which schemas it uses. We also register the document type as a high-level type used by the EJB and rendering layers.
“Core” document types and “ECM” component types should not be confused. The former live in the core Nuxeo EP packages, the latter belong to the high level components. Apart from their definition, most of the uses of "document types" refer to the “ECM” kind.

3.2. 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.

First, we create a schema in the `resources/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,
- two elements and their type.

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

This schema has to be referenced by a configuration file so that the system knows it has to be read. The configuration file will be placed in `OSGI-INF/core-types-contrib.xml` (the name is just a convention):

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

We name our schema sample, and the `.xsd` schema was placed under `resources/` which means that at runtime, it will be available directly from the Jar, therefore we reference it through the path `schemas/sample.xsd`. The schema is registered through an extension point of the Nuxeo component `org.nuxeo.ecm.core.schema.TypeService` named `schema`. Our own extension component is given a name, `org.nuxeo.project.sample.coreTypes`, which is not very important as we only contribute to existing extension points and don't define new ones.

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

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 1
Bundle-Name: NXSample project
Bundle-SymbolicName: org.nuxeo.project.sample;singleton:=true
Bundle-Version: 1.0.0
Bundle-Vendor: Nuxeo
Require-Bundle: org.nuxeo.runtime,
 org.nuxeo.ecm.core.api,
 org.nuxeo.ecm.core
Provide-Package: org.nuxeo.project.sample
Nuxeo-Component: OSGI-INF/core-types-contrib.xml
```

3.3. Core Document Types

By itself, the schemas is not very useful. We associate it with a new “core” document type that we will be creating. In the same `OSGI-INF/core-types-contrib.xml` as above, we add the following:

```
<?xml version="1.0"?>
<component name="org.nuxeo.project.sample.coreTypes">
  <extension target="org.nuxeo.ecm.core.schema.TypeService" point="schema">
    <schema name="sample" src="schemas/sample.xsd" prefix="smp"/>
  </extension>
  <extension target="org.nuxeo.ecm.core.schema.TypeService" point="doctype">
    <doctype name="Sample" extends="Document">
      <schema name="common"/>
      <schema name="dublincore"/>
      <schema name="sample"/>
    </doctype>
  </extension>
</component>
```

The document type is defined through a contribution to an other extension point, *doctypes*, of the same extension component as before, `org.nuxeo.ecm.core.schema.TypeService`. 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.

The Dublin Core schema that we use already contains standard metadata fields, like a title, a description, the modification date, the document contributors, etc. Adding it to our document type ensures that a minimal level of functionality will be present.

3.4. ECM Document Types

After the “core” document type, we need to create the “ECM” document type. This is done through a contribution to another extension point, that we place in the `OSGI-INF/ecm-types-contrib.xml`, the basic structure of this file is:

```
<?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" coretype="Sample">
      ...
    </type>
  </extension>
</component>
```

As of this writing, the document type id has to be the same as the underlying core type; this restriction may be lifted in the future. The type element will contain all the information for this type, this is described below.

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

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

Inside the type element, we provide various information, described below.

3.4.1. Label and Icon

```
<label>Sample document</label>
<icon>/icons/file.gif</icon>
```

The label and icon are used by the user interface, for instance in the creation page when a list of possible types is displayed. The icon is also used whenever a list of document wants to associate an icon with each.

3.4.2. Default view

```
<default-view>view_documents</default-view>
```

The default view specifies the name of the Facelet to use for this document if nothing is specified. This corresponds to a file that lives in the webapp, in this case `view_documents.xhtml` is a standard view defined in the base Nuxeo bundle, that 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.

3.4.3. Layout

3.4.3.1. Configuration

Here is the new layout configuration:

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

Layouts are defined in a given mode (default modes are "create", "edit" and "view") ; layouts in the "any" mode will be merged with layouts defined for specific modes.

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

3.4.3.2. Deprecated configuration

Here is the old layout configuration that has been replaced by the above. If present, it is used instead of the new configuration for compatibility purposes.

```
<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>
```

This section defines a series of widgets, that describe what the standard layout of this document type will be. A layout is a series of widgets, which make 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 according to the layout rules. Note that the layout system is still young and is subject to minor changes in the future. Here we define four widgets, displayed as simple input fields or as a text area.

3.4.4. Containment rules

```
<type id="Folder" coretype="Folder">
  <subtypes>
    <type>Sample</type>
  </subtypes>
</type>
<type id="Workspace" coretype="Workspace">
  <subtypes>
    <type>Sample</type>
  </subtypes>
</type>
```

This contributes a rule to an already existing type, `Folder`. The subtypes element specifies which types can be created inside the type in which the element is embedded.

Here, we specify that our `Sample` type can be created in a `Folder` and a `Workspace`.

3.4.5. Summary

The final `OSGI-INF/ecm-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" coretype="Sample">
      <label>Sample document</label>
      <icon>/icons/file.gif</icon>
      <default-view>view_documents</default-view>
      <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>
    <type id="Folder" coretype="Folder">
      <subtypes>
        <type>Sample</type>
      </subtypes>
    </type>
    <type id="Workspace" coretype="Workspace">
      <subtypes>
        <type>Sample</type>
      </subtypes>
    </type>
  </extension>
</component>
```

Part II. Platform Services

Chapter 4. Actions, Views and JSF tags

4.1. Introduction

The User Interface framework uses different kinds of concepts to make the interface configurable in the same way that the application itself is.

Links and pages that appear on the site can be the result of a "static" template written in XHTML language, but can also be the result of a configuration change, taken into account thanks to more "generic" XHTML pages.

The following chapters will explain how configuration and templating combine to achieve the UI of the site.

4.2. Actions

4.2.1. Concepts

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).

4.2.2. Manage actions

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.

4.2.2.1. Register a new action

An action can be registered using the following example extension:

Example 4.1. Example of an action registration

```
<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: org.nuxeo.ecm.platform/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.

4.2.2.2. 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>
```

4.2.2.3. 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 4.2. 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 4.3. 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 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.

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.

4.2.2.4. 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)}" />
    <nxh:commandLink action="#{action.getLink()}">
      <t:htmlTag value="br" rendered="#{row==(rowCount-1)}" />
      <h:outputText value="#{messages[action.label]}" />
    </nxh: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 `nxh: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.

4.3. Views

4.3.1. UI Views

XXX AT: make the difference between a view in a JSF way (navigation case view id, navigation case output) and views in a nuxeo 5 (type view, action link...)

4.3.2. Manage views

XXX AT: see what views we're talking about + navigation case/link issues.

Define a new view

Register a new view

etc.

4.4. Nuxeo JSF tags

Please refer to the tag library documentation available at http://maven.nuxeo.org/nuxeo-platform-parent/nuxeo-platform-ui-web/tlddoc/index.html.

Chapter 5. Layouts

Let our artists go wild on imaginative page layouts.

-- Grant Morrison

5.1. Introduction

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 uses 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.

5.1.1. 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.

5.1.2. 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.

5.1.3. 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.

5.1.4. Modes

The *layout modes* can be anything although some default modes are included in the application: *create*, *edit*, *view* and *search*.

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.

It is possible to override this behaviour 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.

5.2. 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 then easily available from an xhtml page.

5.2.1. Layout registration

Layouts are registered using a regular extension point on the Nuxeo ECM layout service. Here is a sample contribution.

Example 5.1. Sample layout contribution to the layout service.

```
<?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>
```

5.2.2. 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 define, 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. Please refer the widget definition section.

5.2.3. 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.dublincore.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 "dublincore" schema. If the schema you would like to use does not have a prefix, use the schema name instead. Note that most of standard widget types only handle one field. 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 evry 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:

Example 5.2. Sample complex layout contribution to the layout service.

```
<?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="complex">
      <templates>
        <template mode="any">/layouts/layout_default_template.xhtml</template>
      </templates>
      <rows>
        <row>
          <widget>identifier</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>
```

5.3. 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 merged with layouts defined for specific modes.

5.4. Layout display

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

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>
```

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>
```

5.5. Standard widget types

A series of widget types has been defined for the most generic uses cases.

TODO: add links to the tag library documentation for quoted JSF tags.

5.5.1. 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.

5.5.2. 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.

5.5.3. 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.

5.5.4. 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.

5.5.5. 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.

5.5.6. 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.

5.5.7. 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.

5.5.8. 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.

5.5.9. 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.

5.5.10. selectManyDirectory

The selectOneDirectory 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 c tag in create or edit mode, and properties accepted on a `<nxd:directoryEntryOutput />` tag in other modes.

5.5.11. 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 wigdets 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.

5.5.12. 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.

5.6. Custom templates

Some templating feature have been made available to make it easier to control the layouts and widgets rendering.

5.6.1. 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.

Example 5.3. Default layout template

```
<f:subview
  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">
  <f:subview rendered="#{layout.mode != 'edit' and layout.mode != 'create'}">

    <table class="dataInput">
      <tbody>

        <nxl:layoutRow>
          <tr>
            <nxl:layoutRowWidget>
              <td class="labelColumn">
                <h:outputText value="#{widget.label}" rendered="#{!widget.translated}" />
                <h:outputText value="#{messages[widget.label]}" rendered="#{widget.translated}" />
              </td>
              <td class="fieldColumn">
                <nxl:widget widget="#{widget}" mode="#{widget.mode}" value="#{value}" />
              </td>
            </nxl:layoutRowWidget>
          </tr>
        </nxl:layoutRow>

      </tbody>
    </table>

  </f:subview>

  <f:subview rendered="#{layout.mode == 'edit' or layout.mode == 'create'}">

    <table class="dataInput">
      <tbody>

        <nxl:layoutRow>
          <tr>
            <nxl:layoutRowWidget>
              <td class="labelColumn">
                <h:outputText value="#{widget.label}" rendered="#{!widget.translated}"
                  styleClass="#{nxu:test(widget.required, 'required', '')}" />
                <h:outputText value="#{messages[widget.label]}" rendered="#{widget.translated}"
                  styleClass="#{nxu:test(widget.required, 'required', '')}" />
              </td>
              <td class="fieldColumn">
                <nxl:widget widget="#{widget}" mode="#{widget.mode}" value="#{value}" />
              </td>
            </nxl:layoutRowWidget>
          </tr>
        </nxl:layoutRow>

      </tbody>
    </table>

  </f:subview>

</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 applyting 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.

5.6.2. 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.

Example 5.4. Sample template for a widget

```
<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:t="http://myfaces.apache.org/tomahawk"
  xmlns:nxdir="http://nuxeo.org/nxdirectory">

  <t:dataList id="#{widget.id}" var="listItem" value="#{field_0}"
    layout="simple" styleClass="standardList">
    <h:graphicImage value="/icons/html.png" />
    <h:commandLink value="#{listItem}" immediate="true"
      action="#{userManagerActions.viewUser}">
      <f:param name="usernameParam" value="#{listItem}" />
    </h:commandLink>
    <br />
  </t:dataList>

</f:subview>
```

This widget presents the contributors of a document with specific links on each on these user identifier information.

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.

When this template is included in the page, the `widget` variable is made available:

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 dupplicated 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.

5.7. Custom widget types

Custom widget types can be added to the standard list thanks to another extension point on the web layout service.

Here is a sample widget type registration:

Example 5.5. Sample widget type contribution to the layout service.

```
<?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 behaviour given the property value.

The widet 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 behaviours.

5.8. 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 explicitely for the binding to happen. Depending on the widget, other kinds of bindings can be done.

Chapter 6. Event Listeners and Scheduling

6.1. Introduction

Events and event listeners have been introduced at the Nuxeo core level to allow pluggable behaviours 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 when receiving an event.

6.2. 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.api.event.CoreEvent` 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.listener.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 dublincore 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.

6.3. Adding an event listener

Event listeners can be plugged using extension points. Here are some examples of event listeners registration.

Example 6.1. DublincoreListener registration sample

```
<?xml version="1.0"?>
<component name="DublinCoreStorageService" version="1.0.0">
  <extension target="org.nuxeo.ecm.core.listener.CoreEventListenerService"
     point="listener">
    <listener name="dclistener"
      class="org.nuxeo.ecm.platform.dublincore.listener.DublinCoreListener"
      order="120" />
  </extension>
</component>
```

Example 6.2. UIDGenerator listener registration sample with event filtering

```
<?xml version="1.0"?>
<component name="org.nuxeo.ecm.platform.uidgen.service.UIDGeneratorService">
  <extension target="org.nuxeo.ecm.core.listener.CoreEventListenerService"
      point="listener">
    <listener name="uidlistener"
        class="org.nuxeo.ecm.platform.uidgen.corelistener.DocUIDGeneratorListener"
        order="10">
      <eventId>documentCreated</eventId>
    </listener>
  </extension>
</component>
```

The only thing needed to add an event listener is to declare its name and its class. Sometimes the order in which listeners are called matters so an integer order can be set to control it. A filtering on event ids can be done when registering it too, though the notification method could handle it too.

For instance, the UIDgenerator service will only be notified when the event service receives a document creation event.

6.4. Upgrading an event listener

Since release of Nuxeo EP version 5.0 M3, events involving documents send the document model as source of the event. They used to send the document itself, which was wrong and has been changed in a compatible way.

Old school event listeners should still work ok for now, but should be migrated soon as the compatibility may introduce bugs and will be removed shortly.

To migrate your event listener, make it implement the empty interface `org.nuxeo.ecm.core.listener.DocumentModelEventListener`, and make it deal with a `DocumentModel` instead of a `Document` as event source.

If your event listener does not care about the source, or the event it deals with is not a document, you do not have to do anything.

6.5. Adding an event

To add an event, you have to create it and then notify listeners passing the even to the listener service. Here is a sample code on how to do it:

```
CoreEvent coreEvent = new CoreEventImpl(eventId, source, options,
       getPrincipal(), category, comment);

CoreEventListenerService service = NXCore.getCoreEventListenerService();

if (service != null) {
    service.notifyEventListeners(coreEvent);
} else {
    log.error("Error when notifying core event");
}
```

6.6. From CoreEvents to JMS Messages

Events that are fired at the core level are forwarded to a JMS topic called NXPMessage.

This forwarding is done by a dedicated CoreEventListener (called JMSEventListener contributed by the `nuxeo-platform-events-core` bundle).

In order to be sure that when an JMS event is received the associated DocumentModel is available, all document oriented messages that may occur at core level are forwarded to the JMS topic when the session repository is saved (ie: when data is committed).

In some cases, depending on own the Core API is used, some messages can be duplicated within the same transaction (like modifying several times the same document), the JMSEventListener marks all duplicated messages before sending them to JMS, it's JMS messages receiver to choose to process or not the duplicated messages.

During the forwarding on the JMS Topic, the coreEvents are converted to EventMessage. The main difference is that the EventMessage does not contains the DocumentData (ie: all schemas and fields are unloaded), this is done in order to avoid overloading JMS.

6.7. Adding a JMS message listener

The simplest way to add a JMS message listener is simply to define a Message Driven Bean that is bound to the NXPMessage Topic.

Here is a simple example a the definition of such a MDB :

```
@MessageDriven(activationConfig = {
        @ActivationConfigProperty(propertyName = "destinationType", propertyValue = "javax.jms.Topic"),
        @ActivationConfigProperty(propertyName = "destination", propertyValue = "topic/NXPMessages"),
        @ActivationConfigProperty(propertyName = "providerAdapterJNDI", propertyValue = "java:/NXCoreEventsProvi
        @ActivationConfigProperty(propertyName = "acknowledgeMode", propertyValue = "Auto-acknowledge") })
@TransactionManagement(TransactionManagementType.CONTAINER)
public class NXAuditMessageListener implements MessageListener {

    private static final Log log = LogFactory.getLog(NXAuditMessageListener.class);

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public void onMessage(Message message) {
        try {

            final Serializable obj = ((ObjectMessage) message).getObject();
            if (!(obj instanceof DocumentMessage)) {
                log.debug("Not a DocumentMessage instance embedded ignoring.");
                return;
            }

            DocumentMessage docMessage = (DocumentMessage) obj;

            String eventId = docMessage.getEventId();
            log.debug("Received a message with eventId: " + eventId);
...
```

The `DocumentMessage` is a subclass of the `DocumentModel`.

An important point to remember is that the MDB is executed asynchronously in a dedicated thread:

- there is no JAAS Session established: you can not access the repository without this
- the `DocumentMessage` is not bound to an existing `CoreSession`: you can not use the `DocumenMessage` to do lazy loading (ie: `DocumentMessage.getProperty()`)

So, in order to extract some document oriented properties of the document associated to the event, you must:

- Establish a JAAS Session
- get a connected `DocumentModel` using the `DocumentRef` provided by the `DocumentMessage`

Here is a code sample for this:

```
LoginContext lc;
CoreSession session;
String repositoryName = docMessage.getRepositoryName();
try {
    log.debug("trying to connect to ECM platform");
    lc = Framework.login();
    session = Framework.getService(RepositoryManager.class).getRepository(repositoryName).open();
    DocumentModel connectedDoc = session.getDocument(docMessage.getRef());
...
} finally {
   if (session != null)
      CoreInstance.getInstance().close(session.getSessionId())
   if (lc != null)
      lc.logout();
}
```

6.8. Scheduling

XXX TODO: FG

Chapter 7. User Notification Service

7.1. Introduction

The notification framework provides a way to notify the users regarding different events as they happen in the system.

7.2. Notification concept

A notification is an alert that is sent to some users when an event takes place in the system (e.g. a document is created or deleted, a document is modified or published, a comment was entered, etc.).

A notification is defined by following attributes:

- name: must be unique
- channel: email, SMS, Jabber, etc.
- events: a series of events to which it responds
- template: is a file in which is stored the text that is sent to an user - it may be dynamic

In order to define a notification, one must declare all these attributes in a contribution file.

7.3. Notification channels

The channel is the communication channel used by the notification service to send alerts(notifications) to users.

In order to define a new channel (by default only email channel is used) the `ChannelNotificator` interface must be implemented.

It has 2 methods:

- `isInterestedInNotification(Notification docMessage)` usually checks if the channel is right.
- `sendNotification(DocumentMessage docMessage)` sends the actual notification. For now only email notification channel is implemented.

7.4. E-mail notifications

A notification must be defined in a xml file like the default `notification-contrib.xml` that is used by default.

Default defined notification may be disabled by setting *`enabled="false"`*.

In order to define a new notification one must place a definition like this one in his contribution file :

```
<notification name="Modification" channel="email" enabled="true"
              availableIn="workspace" autoSubscribed="false"
              template="modif" subject="Document modified"
              label="label.nuxeo.notifications.modif">
  <event name="documentModified"/>
  <event name="contentSubdocumentModified"/>
</notification>
```

As you may see above a notification must declare a list of events to which it reacts. In our case `documentModified` `contentSubdocumentModified`.

Also a notification must have a name that must be unique within the application. A label must also be specified for i18n.

The attribute enabled is used to enable / disable specific notifications.

The attribute *`autoSubscribed`* is set to true when we want that a notification is sent to all concerned users. In this case within the `eventInfo` map there must be loaded also the users that are concerned. For example if we want that some users (ex: administrators or workflow manager) to get a notification each time a task is assigned to them, we must use `autoSubscribed="true"` and put the usernames of all users in the `eventInfo` of the event under the key `recepients`.

The attribute *`availableIn`* is used in order to restrict the scope of a notification. For example if we want to define a notification that is triggered each time the document is modified, then it would not be used inside a section, because sections contain documents that cannot be modified, only published. So in order to hide this notification inside a section, we specify *`availableIn="workspace"`*. The accepted values are *`workspace`*, *`section`* and *`all`*.

The template attribute specifies the name of a template that will be used for generating the body of the email(notification). This name is associated with a file using another extension point like this: `<template name="modif" src="templates/modif.ftl" />`.

Inside a `*.ftl` file there may be inserted some dynamic parts like the document name, the user triggering the event, etc. Any data that one wishes to put inside the body of the email, or it's subject, he must put that data in the `eventInfo` map under a unique key. Then inside the template file that data will be displayed using *`${key}`*.

For the email notification a subject is used. This subject is a string but is also dynamic following the same rules as the body inside the template files.

Note that if you are writing an HTML based template it will be advised to use HTML encoded letters when there is accentuated letters (in french for example "é" will be "&eacute;").

Chapter 8. Indexing & Searching

This chapter presents the architecture of the indexing and search service in Nuxeo EP 5.

8.1. Introduction

This chapter is under construction. XXX TODO: GR+JA

8.2. Configuration

For obvious performance and volume considerations, the search service doesn't index all the content of the application, nor does it provide the full content in search results. This must be specified in the configuration, along with what to do with the data.

The search service configuration is done by the standard Nuxeo Runtime extension point system. The target is `org.nuxeo.ecm.core.search.service` For a complete example, check the default configuration file: `nuxeo-platform-search-core/OSGI_INF/nxsearch-contrib.xml` (from Nuxeo EP sources).

8.2.1. Concepts

The main concepts are *Resource* and *Field*. A Resource holds several fields. It has a name and a prefix, which is used, e.g, in queries. Resources are supposed to be unsplittable, but they are usually aggregated. It's up to the backend to decide how to handle that aggregation; this goes beyond the scope of the present documentation.

At this point the search service handles documents only, so it's safe to say that documents correspond to aggregates of resources, and resources correspond to schemas. In the future, there'll be more types of resources.

8.2.2. The `indexableDocType` extension point

The core types of documents to index have to be registered against this extension point, in which the schemas to index are bound to each document type.

Here's an example demonstrating the available patterns:

```
<extension target="org.nuxeo.ecm.core.search.service.SearchServiceImpl"
           point="indexableDocType">
(...)
  <indexableDocType name="DocType1" indexAllSchemas="true"/>
  <indexableDocType name="DocType2" indexAllSchemas="true">
    <exclude>unwanted_schema</exclude>
  </indexableDocType>
  <indexableDocType name="DocType3">
    <resource>indexed_schema</resource>
  </indexableDocType>
</extension>
```

In this example, the Search service will index all schemas for documents of type `DocType1`, all schemas except `unwanted_schema` for type `DocType2` and only `indexed_schema` for type `DocType3`.

Each ot these indexable schemas will be processed according to the corresponding indexable schema resource if available in the configuration, or to the default one .

In particular, the behaviour of data from a given schema is homogeneous accross all document types.

8.2.3. The `resource` extension point

This extension point is used to declare an indexable resource. In 5.1M2, the only provided indexable resource type is the `schema` resource type. but the logic will stay the same for future types. Recall that resources are made of *fields*. Here's an example of schema indexing resource, without fields details.

```
<extension target="org.nuxeo.ecm.core.search.service.SearchServiceImpl"
           point="resource" type="schema">

  <resource name="dublincore" prefix="dc" indexAllFields="true">
    <excludedField>issued</excludedField>
    <field name="title" (...) />
  </resource>

  <resource name="book" prefix="bk" type="schema">
    <field name="barcode" (...) />
  </resource>

</extension>
```

The `type` specified that the resource is a docuemnt schema resource.

The `name` and `prefix` attributes are mandatory. They should match the ones from the `schema` extension point of `org.nuxeo.ecm.core.schema.TypeService` as in, e.g.:

```
<schema name="dublincore" prefix="dc" src="schema/dublincore.xsd"/>
<schema name="common" src="schema/common.xsd" />
```

A missing prefix in the core configuration as in the `common` schema declaration in the example above will default to the schema name.

The prefix is important, since all subsequent references to the fields (in queries and raw search results) take the `prefix:fieldName` form.

8.2.4. Field configuration

Fields behavior is designed to be uniform across the different kinds of resources.

8.2.4.1. The `type` attribute

The field type tells the search engine how to process the field. This is a mandatory, *case-insensitive*, attribute.

The following table summarizes the available types. The listed Java classes are guaranteed to work. The backend might implement more converting facilities.

Type	Java classes	Comment
Keyword	String	Meant for technical non binary strings such as vocabulary keys, user identifiers etc. Equality matches are guaranteed to be exact.
Text	String	Upon indexation, these fields are tokenized and analyzed to support fulltext queries.
Path	org.nuxeo.common.utils.Path, String	Dedicated to `STARTSWITH` queries. Equality queries are not guaranteed.
Date	java.util.Calendar	For date and time indexing
Int	int	
Builtin	Reserved for internal use	

8.2.5. Text fields and analyzers

At indexing time, the contents of text fields goes through the process of tokenization and analysis, whose main goal is to provide fulltext search capabilities, usually at the expense of exact matches.

Tokenization means converting the textual content in a stream of words (tokens). During the analysis step, this raw stream is altered to better suit the needs of fulltext searches. It is for example common practice to strip the stream of so-called stop words (most commons words in the language) and to degrade accented characters. One can apply further linguistical processing, such as stemming or synonym analysis.

The name of analyzer to use on a given text field is specified through the `analyzer` attribute. The Search Service acts as a pure forwarder, sending the raw text to the backend, along with the specified analyzer name.

The default value for the `analyzer` attribute is `default`. The attribute is simply ignored for other field types.

8.2.6. Boolean attributes

- To enable queries on a given field, one must set the `indexed` attribute to `true`.
- To make it possible to provide the full field value in search results, one must set the `stored` attribute to `true`. One must keep in mind that the purpose is to present the user a limited yet sufficient set of information along with search results, i.e, in a swifter way than having to fetch it from the core, and not to duplicate the content in the search databases.
- Multivalued fields have to be flagged by setting the `multiple` attribute to `true`.
- Depending on field types and on the processing that's been done by the backend, the possibility to use the field value as a sort key might require some additional resources. To force the backend to give this extra effort, set the `sortable` attribute to `true`.
- The binary attribute is used to mark binary fields, e.g, to trigger conversions (not used in 5.1M2).

8.2.7. Schema resources and fields without configuration

8.2.8. Schema resources

For a schema resource that isn't explicitely declared and nevertheless used, for instance because of an `indexAllSchemas` statement, a default configuration is inferred, with the prefix read from the Core configuration, and fields as below, i.e., as if there was an `indexAllFields="true"` attribute.

8.2.9. Automatic fields configuration

- auto-configured fields are unstored
- If there is only one relevant type from the table above, it is appplied.
- The `multiple` attribute is properly set.
- auto-configured String fields get the `keyword` type.

Example 8.1. Relying on automatic field configuration on all fields but one

```
<resource name="dublincore" prefix="dc" indexAllFields="true" type="schema">
  <field name="title" analyzer="default"
         stored="true" indexed="true" type="Text"
         binary="false" sortable="true"/>
</resource>
```

8.3. Programmatic Searching

The search service exposes the `searchQuery` method as a unique entry point. The method takes as input an instance of `ComposedNXQuery`, which encapsulates the parsed NXQL query and a `SearchPrincipal` instance that will be checked against the security indexes, and paging information for the results.

Although the input of `searchQuery` is an already parsed NXQL statement, we'll use NXQL query strings in the sequel for clarity. NXQL query strings are parsed by the method `parse` of the static `org.nuxeo.ecm.core.query.sql.SQLQueryParser` class.

8.3.1. Fields and literals

Within NXQL requests, references to fields values have to follow the "prefix:fieldName" scheme, where prefix and fieldName have been specified through the configuration extension points (recall that for automatically indexed schema resources, the prefix defaults to the one defined in the schema definition).

Literals (constants) follow the JCR specifications. Notably:

- String literals have to be enclosed in single quotes
- Lists have to be enclosed in parenthesizes

Example 8.2. Sample NXQL queries

Recall that "dc" is the prefix for the "dublincore" schema.

```
SELECT * FROM Document WHERE dc:title='Nuxeo book' ORDER BY dc:created DESC
```

8.3.2. `WHERE` statements

Most `WHERE` statements behave as described in the JCR specification, which is itself based on general SQL. Instead of covering every aspect of NXQL, in the current state of this documentation, we'll focus on differences and behaviours that might appear to be counter intuitive.

8.3.2.1. Text fields

Although the Search Service is meant to provide an unified abstraction on the tasks of indexing and querying, text fields have to be somewhat an exception. Indeed, search engines have very different capabilities, depending on provided analyzers. They are nonetheless all expected to provide a direct syntax for full text searches, that an end user can use from, e.g., an input box on a web page. Given the very special kind of constraint that indexing a text field represents, it's not guaranteed that exact matches are supported.

See Section 8.4.4, “Text fields behavior” from the documentation of the Compass backend to get a more concrete view on this (with examples).

8.3.2.1.1. Conclusions

- The backend uses the closest thing to exact matches it supports to treat `=` predicates.
- The syntax of `LIKE` predicates is backend dependent. It follows the backend's full text query syntax
- The `CONTAINS` from JCR is *not* supported. Use a `LIKE` statement on the main full-text field (see Section 8.3.2.3, “Pseudo fields”).

8.3.2.2. Multi-valued fields

On a multi-valued field, the `=` operator is true if the right operand belongs to the set of field values. The `IN` operator is true if the intersection between the set of field values and the right operand is non empty.

Example 8.3. Finding documents on which user sally contributed

```
SELECT * FROM Document WHERE dc:contributors = 'sally'
```

Example 8.4. Finding documents on which user sally or harry contributed

```
SELECT * FROM Document WHERE dc:contributors IN ('sally', 'harry')
```

This behavior is in conformance with the JCR specification, which states it the following more general terms:

> In the WHERE clause the comparison operators function the same way they do in XPath when applied to multi-value properties: if the predicate is true of at least one value of a multi-value property then it is true for the property as whole.

8.3.2.3. Pseudo fields

The following fields are available on all document resources. They don't correspond to document fields and aren't configurable, that's why they're called *pseudo-fields*.

The names of these fields are synchronized with constants from the class `BuiltinDocumentFields`. Any use from java code should rely on these.

Constant	Field name	Description
FIELD_FULLTEXT	ecm:fulltext	The default full-text aggregator
FIELD_DOC_PATH	ecm:path	The document's path
FIELD_DOC_NAME	ecm:name	The document's name (last component of the path)
FIELD_DOC_URL	ecm:path	The document's URL
ecm:path	The document's path	
FIELD_DOC_NAME	ecm:name	The document's name (last component of the path)
FIELD_DOC_URL	ecm:path	The document's URL
FIELD_DOC_REF	ecm:id	The `DocumentRef` as fetched from the core
FIELD_DOC_PARENT_REF	ecm:parentId	The parent `DocumentRef`
FIELD_DOC_TYPE	ecm:primaryType	The Core type
FIELD_DOC_FACETS	ecm:mixinType	The facets (multiple)
FIELD_DOC_LIFE_CYCLE	ecm:currentLifeCycleState	clear enough

FIELD_DOC_VERSION_LABEL	ecm:versionLabel	version label (a string)
FIELD_DOC_IS_CHECKED_IN_VERSION	ecm:isCheckedInVersion	a boolean (0 or 1) that discriminates between the current version and previous, frozen (checked in) ones.

8.4. The Compass plugin

Compass is a popular wrapper around Apache Lucene. This plugin allows to use it as a backend for the Search Service.

8.4.1. Configuring Compass

Compass configuration is split in a master XML configuration file and one or several mapping files. The latter specifies the treatments that resources and fields (properties in Compass terminology), while the former is to be used to tune JTA transactions, data sources, and to register configuration of analyzers and converters.

The contents of these files are covered in great details in the Compass 1.1 documentation. In the present documentation, we'll focus on integration matters with the Nuxeo Search Service.

8.4.1.1. Configuration files location

All Compass specific configuration files are relative to the classpath of the compass plugin. A default configuration is provided for Nuxeo EP 5 WebApp. To customize it, one sadly has to put the configuration at the right place within the backend's JAR.

Here is an ant fragment to perform this in a JBoss context, assuming that the Search Service has already been installed in the application server and that the server's deployment directory is stored in the `deploy.dir` property.

```
<copy todir="${deploy.dir}/nuxeo.ear/platform/nuxeo-platform-search-compass-plugin-1.0.0-SNAPSHOT.jar"
      overwrite="true" failonerror="false">
  <fileset dir="src/main/resources">
    <include name="myfile.xml" />
  </fileset>
</copy>
```

The backend's JAR is included as a directory in `nuxeo.ear` during the Maven build of `nuxeo-platform-ear` for this single purpose. This is prone to change in the future.

8.4.1.2. Specifying the master configuration XML file name

The Compass backend itself is registered against the Search Service through the `searchEngineBackend` extension point of `org.nuxeo.ecm.core.search.service.SearchServiceImpl`. Your component can use the `configurationFileName` element to specify a path to the master configuration file, like this:

```
<searchEngineBackend name="compass" default="true"
    class="org.nuxeo.ecm.core.search.backend.compass.CompassBackend">
  <configurationFileName>/mycompass.cfg.xml</configurationFileName>
</searchEngineBackend>
```

The default path is `/compass.cfg.xml`.

8.4.2. Global configuration

8.4.2.1. Storage

Compass supports several storage possibilities, called *connections* in Compass configuration objects. The configuration is done trough a Nuxeo Runtime extension point and possibly within the Compass master configuration file. *The extension point always takes precedence over the Compass file*, but can be used to fall back to Compass file, that offers currently more possibilites.

The target is `org.nuxeo.ecm.core.search.backend.compass.CompassBackend`, and the point is `connection`. Contributions are made of a single XML element; the latest one wins.

To set the connection to a file system Lucene store, put a `file` element in the contribution, and set the `path` attribute to the target location. If the path doesn't start with `/`, it will be interpreted as being relative to Nuxeo Runtime's home directory, e.g, `/opt/jboss/server/default/data/NXRuntime` in the default Nuxeo EP installation on JBoss.

Other connection types, notably JDBC, are defined by the Compass configuration file, one has to put the `default` XML element in the contribution, like this:

```
<extension target="org.nuxeo.ecm.core.search.backend.compass.CompassBackend"
           point="connection">
  <default/>
</extension>
```

The default connection is a relative file system one, hosted in the `nxsearch-compass` sub directory of Nuxeo Runtime's home.

8.4.2.2. Analyzers

The master configuration file holds the definition and configuration of analyzers: a lookup name gets associated to an analyzer type and options. The Compass backend makes Compass use directly the name declared in the Search Service as the lookup name, configuration, therefore one has to ensure here that all of these do exist in the Compass configuration.

Together with the registration itself comes the configuration of analyzers. For instance, an analyzer discarding stop words might be given the full list of stop words.

Compass comes with a two predefined analyzers: `default` and `search`. You can reconfigure them as well.

See the relevant part in Compass documentation for details and sample configurations.

8.4.2.3. Converters

Lucene only knows about character strings. Therefore, typed data such as dates and integers must be converted in strings to get into Lucene and back. Compass provides helpers for this and the Compass backend uses them directly.

In the master configuration file, one register available converters in the form of a lookup name and a Java class. Lots of converters are already registered by default, covering most basic types. The `compass.cfg.xml` file shipping with the Compass backend redefines one (the `date` converter).

8.4.3. Mappings for Nuxeo

For the time being, a part of the Search Service configuration has to be duplicated in the Compass mappings XML file.

8.4.3.1. What to describe and syntax

Currently, the Compass backend can't force Compass to use a given converter and/or analyzer on a given field. It *must* therefore be specified in the mappings file, which is itself loaded from the master configuration file. The default name for this file is `nxdocument.cpm.xml`.

Here's a sample, inspired from the mappings file provided with the backend.

```
<?xml version="1.0"?>
<!DOCTYPE compass-core-mapping PUBLIC
    "-//Compass/Compass Core Mapping DTD 1.0//EN"
    "http://www.opensymphony.com/compass/dtd/compass-core-mapping.dtd">

<compass-core-mapping>

  <resource alias="nxdoc"
            sub-index="nxdocs"
            analyzer="default"
            all="false">
    <resource-id name="nxdoc_id"/>

    <resource-property name="dc:created" converter="date"
                       store="yes" index="un_tokenized" />
    <resource-property name="dc:title" analyzer="french" />

  </resource>
</compass-core-mapping>
```

In Compass' terminology, fields are called *properties*. The name of the Compass property corresponding to a Nuxeo indexed field coincides with the field's prefixed name.

Some important remarks:

- Start from the current mappings file shipping with your version of the compass backend and keep it up to date afterwards
- Don't change anything besides `resource-property` elements.
- An exception to the above rules: you may experiment with the `sub-index` attribute, according to your performance needs.

8.4.3.2. Installing the mappings file

Follow the instructions from Section 8.4.1.1, “Configuration files location”

8.4.4. Text fields behavior

Everything in this part applies to fields that have explicitly been declared with a "text" type through the extension point. Anything that's meant for text fields in the mappings configuration files will be ignored if the field has another type.

8.4.4.1. Indexing

At indexing time, the text field is analyzed using the analyzer from the Compass configuration file regardless what has been configured through the Search Service extension point.

8.4.4.2. Searching

Equality statements in WHERE clauses are transformed into the closest thing that Lucene can provide on an analyzed field, namely a phrase query.

On the other hand, LIKE clauses are directly fed to Lucene's `QueryParser`. To search documents whose title starts with "nux", one may write

```
SELECT * FROM Document WHERE dc:title LIKE 'nux*'
```

The following two statements are equivalent. The second one is the QueryParser syntax for phrase queries.

```
... WHERE dc:title='Nuxeo Book'
... WHERE dc:title LIKE '"Nuxeo Book"'
```

Lucene's `QueryParser` syntax is really powerful, you can specify how close two words can be, apply fine grained boosts for the relevance ordering, and more. The only restriction you have on LIKE statements for text fields within the Compass backend is the choice of field. In other words, the colon character is escaped.

You should be aware of the following *trap* in Lucene queries: purely negative queries don't match anything, even if they are themselves nested in a boolean query that has positive statements. The Compass backend uses the standard way to circumvent this limitation, provided that the negative aspect can be seen from the NXQL structure, i.e., not enclosed in a Lucene QueryParser literal.

Example 8.5. Negative queries

Queries that won't return anything:

```
... WHERE dc:title LIKE '-book'
... WHERE dc:title LIKE 'nuxeo' AND dc:title LIKE '-book'
```

Queries that should work as intended (the last three being equivalent):

```
... WHERE dc:title NOT LIKE 'book'
... WHERE dc:title LIKE 'nuxeo' AND dc:title NOT LIKE 'book'
... WHERE dc:title LIKE 'nuxeo' AND NOT dc:title LIKE 'book'
... WHERE dc:title LIKE '+nuxeo -book'
```

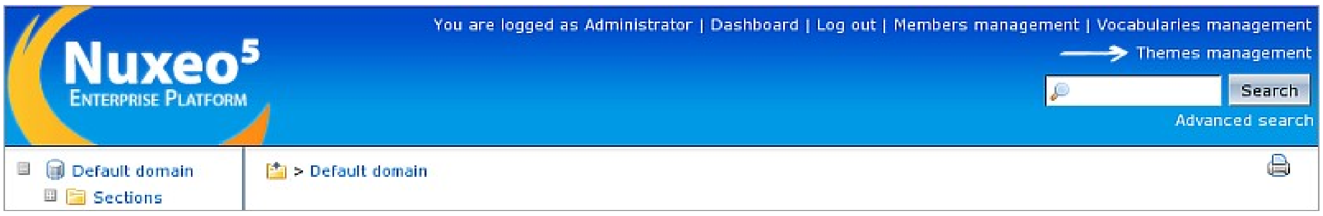
8.5. Building a search UI with QueryModel

Chapter 9. Look and feel

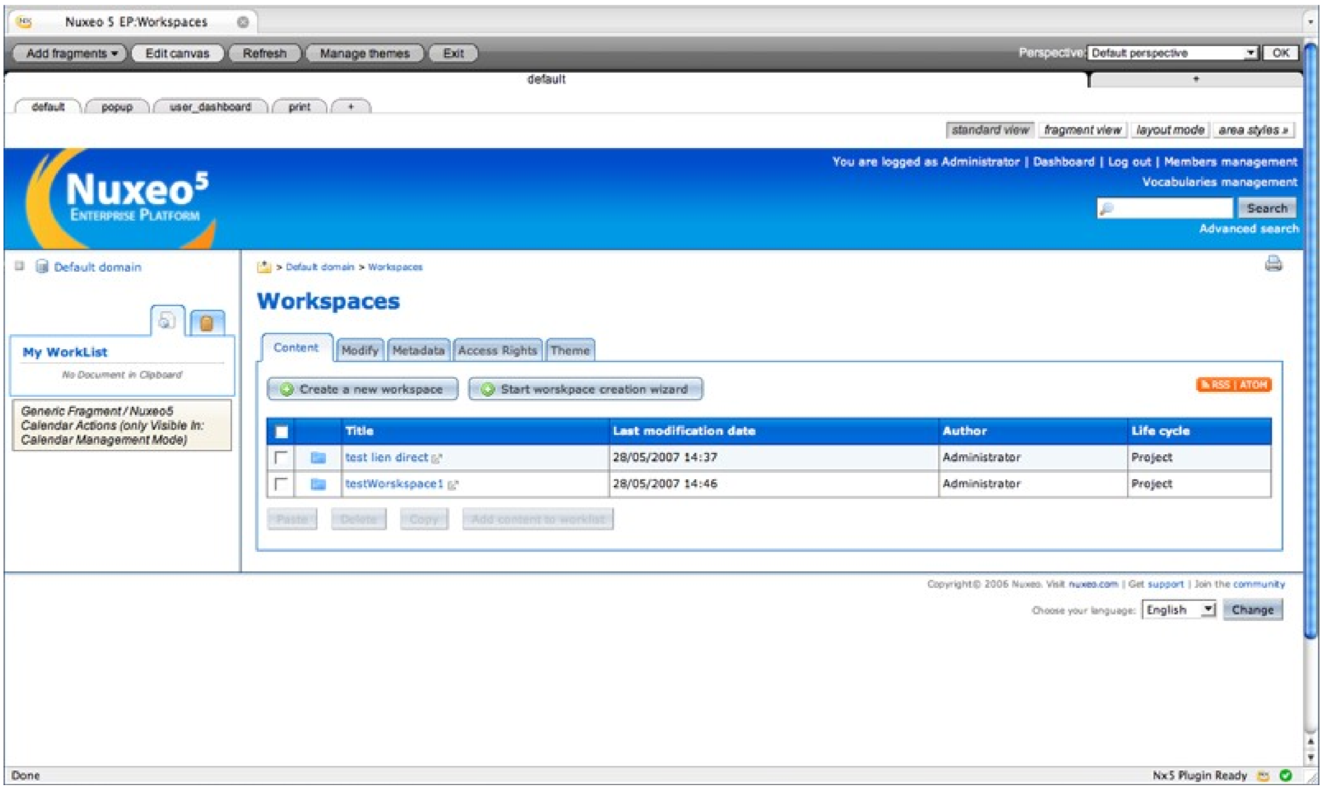
9.1. Introduction

The Nuxeo theme notion is wider than the notion attached to the same word in portal concepts. Indeed, the Nuxeo Theme defines all the look and feel of your webapp: composition, layout and graphical appearance. Nuxeo does not aim at developing a portal, i.e. a JSR 168 container, but it authorizes a kind of page and widget management to get some flexibility in the design you want to give to your webapp. The tool to enable you manage those aspects is "Nuxeo Theme editor", that you can make appear with the following command when you are in Nuxeo Web app:

To switch to Nuxeo Theme editor click on the 'Themes Management' link in the user services panel.



Alternatively, you can type 'shift'+'alt'+'t'. To switch to Nuxeo Theme editor with Mozilla / Firefox < 2.0 type 'alt'+'t'



Title	Last modification date	Author	Life cycle
test lien direct	28/05/2007 14:37	Administrator	Project
testWorskspace1	28/05/2007 14:46	Administrator	Project

9.2. Principle

One special fragment is the Facelet region: in the properties tab of the editor you can specify the name of a faces to directly integrate it into your page.

To use Nuxeo theme editor, you need to understand its model. The main entity is the theme. Then a theme may have many pages. For each pages, you define a layout (a canvas) and add a list of fragments (widget). The graphical editor uses a tab (theme) and sub-tab (page) system. When you want to add a new page or theme click on "+" at the end of the tab list. For a page you have three possible views:

- **wysiwyg:** you can move the widgets and evaluate the rendering.
- **fragments:** to put the widgets in their placeholder (= a cell). You can put many widgets in a place holder.
- **layout:** you can create new rows and divide the rows into cells, and specify the width of each cell (just edit the nn% on the screen).

You can click on a fragment to edit it. When you edit a fragment you have a multitab editor to specify:

- **custom properties** of the fragment: e.g. the text if it is a text fragment,
- **the graphical object** (the widget) associated
- **the style** of each HTML object that composes the widget.
- then you specify the **perspective** in which the fragment can be seen.

One special fragment is the Facelet region: in the properties tab of the editor you can specify the name of a faces to directly integrate it into your page.

9.3. Mechanism

Nuxeo Editor is done for not having to understand the underground mechanism. Yet it can be good to understand the background to better leverage the tool and its possibilities. The Nuxeo component that manages the customization and extension of Nuxeo EP 5 look and feel is: `org.nuxeo.theme.services.ThemeService`. To register a whole theme (widget, style, layout etc. ...) you need to contribute to the extension point "theme" this way:

```
<extension target="org.nuxeo.theme.services.ThemeService" point="themes">
  <theme>
    <src>META-INF/nxthemes-setup.xml</src>
  </theme>
</extension>
```

In the trunk, the default theme is in the webapp project. Having a look inside enables us to discover the main features.

9.3.1. The Elements

The file starts with the elements declaration, we define the pages, the rows (section markup), the cells and the fragments in the cells.

```
<theme name="default">
  <layout>
    <page name="default">
      <section>
        <cell>
          <fragment type="generic fragment" />
        </cell>
        <cell>
          <fragment type="generic fragment" />
        </cell>
        <cell>
          <fragment type="generic fragment" />
        </cell>
      </section>
    </page>
    <page name = ...>
    .
    .
    .
    </page>
 </layout>
</theme>
```

All this markup refers to an `Element` subtype in the java model: `PageElement`, `CellElement`... The **fragment element**, the one that gives the widget oriented capacity to Nuxeo is typed: we have in the default distribution "generic fragment", "action fragments". A typed fragment returns a model to be displayed and edit in the edit mode of the fragments. This model is often (but not always) what we can see in the "properties" tab of the fragment editor. For now there is in the default Nuxeo distribution:

- **generic fragment** (empty model)
- **textual fragment**
- **region fragment**
- **action fragment**

The fragment can also receive a perspective attribute that specifies in which perspective it will be displayed (the fourth tab in the fragment editor). You can then propose to Nuxeo user the same kind of experience you have in Eclipse. The perspective are specified in the **perspective** extension point of `!ThemeService` component.

9.3.2. The format

Then, once you declared all the elements, you can format them through different axes :

- their layout
- their rendering (their view)
- their style

To do this, you put, inside the `<theme>` markup, children markup from those types:

- layout --> `<layout element = ...>`
- rendering --> `<widget element ...>`
- style --> `<style element = ...>`

Those 3 markups use the attribute element to get the reference to which element of your skeleton they will be applied:

```
element="page[3]/section[3]/cell[1]"
```

9.3.2.1. The rendered widget

Indeed each element is rendered by a view.

```
<widget element="page">
  <view>page frame</view>
</widget>
```

This view is defined like this (with another extension point of `!ThemeService: views`):

```
<view name="page frame">
  <element-type>page</element-type>
  <format-type>widget</format-type>
  <class>org.nuxeo.theme.jsf.views.JSFView</class>
  <template>nxthemes/jsf/widgets/page-frame.xml</template>
</view>
```

We can see that a view is associated to an element type. The element types contributions should be reserved to Nuxeo only (one should manage with existing ones). The template markup gives the html/faces/text code to be used for rendering the view. Notice that in the view, you can access the fragment model data through the EL call `nxthemesInfo.model`.

```
<h:outputText escape="false" value="#{nxthemesInfo.model.body}" />
```

9.3.2.2. The layout

The layout properties are given like this (still under the `<theme>` markup) :

```
<layout element="page[3]/section[3]/cell[1]">
  <width>50%</width>
  <padding>20px</padding>
  <margin>0</margin>
</layout>
```

This enables to adjust the position of the fragments inside.

9.3.2.3. The style

Then comes the `style` properties. Again, you apply them to an element:

```
<style element="page[1]/section[3]/cell[1]|page[3]/section[4]/cell[1]">
  <selector path="">
    <color>#757575</color>
    <border-style>solid none none none</border-style>
    <border-color>#003366</border-color>
    <border-width>1px</border-width>
    <background>
      #FFF url(/nuxeo/img/gray_gradient.gif) top left repeat-x
    </background>
    <padding>5px 15px 5px 5px</padding>
  </selector>
  <selector path="div">
    <font-size>9px</font-size>
  </selector>
</style>
```

The selector specifies the markup to which the defined style is applied. The style definition used is the one of the deeper upper-element that has a style definition specified. To be exhaustive, we need to present the filter system (TODO)

9.3.3. The negotiation

We have seen how to define a theme. Now we need to see how a theme is applied. More precisely, how do I manage the choice of the page I will display? In fact, the Nuxeo Theme framework proposes many ways to specify the theme applied to the webapp for a given view:

- with a cookie
- with a request parameter (?theme= ...)
- with an association between a JSF view id and a theme

So how to manage priority when more than one parameter is passed to the framework? The `!ThemeService` component has another extension point to achieve this: the `negotiation` extension point. Not only can it be used to select a theme but it also works with other objects as we will see later.

```
<negotiation object="theme" strategy="nuxeo5">
  <scheme>org.nuxeo.theme.jsf.negotiation.theme.RequestParameter</scheme>
  <scheme>org.nuxeo.theme.jsf.negotiation.theme.CookieValue</scheme>
  <scheme>org.nuxeo.theme.jsf.negotiation.theme.ViewId</scheme>

  <!-- local theme (specific to nuxeo5) -->
  <scheme>org.nuxeo.ecm.webapp.theme.LocalTheme</scheme>
  <scheme>org.nuxeo.theme.jsf.negotiation.theme.DefaultTheme</scheme>
</negotiation>
```

As we can see in the example, the negotiation point defines the order in which the different methods for obtaining the current theme information are applied. This negotiation feature also applies to other Nuxeo Theme objects like the engine, the mode, the perspective.

9.3.4. The engine

Engine and filter are two notions that work together. An engine is the combination of different filters, and a filter is a "sub-unit" of rendering. So the engine is the global renderer of your web app. From the elements skeleton, it will generate the graphical appearance, passing each element through different black boxes, depending on the type of the element. Here is the definition of the default engine of Nuxeo.

```
<engine name="default">
  <renderer element="theme">
    <filter>add widget</filter>
    <filter>collect xmlns</filter>
  </renderer>
  <renderer element="page">
    <filter>add widget</filter>
    <filter>set layout</filter>
    <filter>set style</filter>
  </renderer>
  <renderer element="section">
    <filter>add widget</filter>
    <filter>set layout</filter>
    <filter>set style</filter>
  </renderer>
  <renderer element="cell">
    <filter>add widget</filter>
    <filter>set layout</filter>
    <filter>set style</filter>
  </renderer>
  <renderer element="fragment">
    <filter>control fragment visibility</filter>
    <filter>add widget</filter>
    <filter>set style</filter>
    <filter>write fragment tag</filter>
  </renderer>
</engine>
```

The engine, that you register in the `ThemeService` component through the "engines" extension point lets you add for each type of element some "filters" that will do some work around the markup content at rendering time. Nuxeo already uses filters like the style fitler, that put the style definition you chose, the layout filter, the "drag'n drop" filter... One interesting use of the filters is illustrated with the Nuxeo Theme editor: when you type **shift+alt+t**, the display changes and you are in the WYSIWYG Nuxeo theme editor. But all the components that make your page are still here, they just look a bit different because, some different filters are used. For instance, because of the drag'n drop filter presence, you can move the fragments.

9.3.5. Resource management

Graphical components may need some external resources such as CSS or JavaScript libraries. Nuxeo theme has an embedded resource management system that at rendering time automatically computes the list of the files needed for rendering a page. Resources are served using gzip compression when supported by the browser. JavaScript resources are also compressed using Dojo's ShrinkSafe. Finally all files of a same type (.css or .js) are concatenated. This reduces the number of individual downloads and it enables to manage dependencies between resources. Indeed, at declaration time you can specify the dependencies for a given resource:

```
<extension target="org.nuxeo.theme.services.ThemeService"  point="resources">
  <resource name="controls.js">
    <path>nxthemes/jsf/scripts/scriptaculous/controls.js</path>
    <require>effects.js</require>
    <require>prototype.js</require>
  </resource>
</extension>
```

Then, when you register the view associated to an element, you specify the resources it needs:

```
<view name="nuxeo5 clip board">
  <format-type>widget</format-type>
  <class>org.nuxeo.theme.jsf.views.JSFView</class>
  <template>incl/user_clipboard.xhtml</template>
  <resource>dragdrop.js</resource>
</view>
```

9.3.6. Application

The last concept you need to know about to completely control the look and feel of your application is the application extension point:

```
<extension target="org.nuxeo.theme.services.ThemeService"
```

```
    point="applications">

  <application root="/nuxeo">

    <negotiation>
      <strategy>nuxeo5</strategy>
      <default-engine>default</default-engine>
      <default-theme>default/default</default-theme>
      <default-perspective>default</default-perspective>
    </negotiation>

    <resource-caching>
      <lifetime>36000</lifetime>
    </resource-caching>

    <view id="/create_relation_search_document_popup.xhtml">
      <theme>default/popup</theme>
    </view>
    <view id="/user_dashboard.xhtml">
      <theme>default/user_dashboard</theme>
    </view>
    <view id="/view_calendar.xhtml">
      <perspective>view_calendar</perspective>
    </view>
    <view id="/print.xhtml">
      <perspective>print</perspective>
    </view>
  </application>

</extension>
```

As you can see in the example, an **application** is associated to a web-app root context. There you specify the strategy (a negotiation grouping feature), the default engine, the default theme and perspective. You also specify the caching policy and there you also declare the JSF view id / theme association that we went through earlier in this tutorial.

9.4. Customizing the theme

Eventually, all theme and styling work will be done in the Theme Editor. For now, we have to use both the editor and the file `theme-default.xml` in `org.nuxeo.ecm.platform/nuxeo-platform-webapp-core/src/main/resources/META-INF/`.

What can be done in the editor: page layout, widget moving, fragment styling, page/section/cell preset borders and backgrounds

What must be done in `theme-default.xml` : commons styles and their inheritance

In addition to the `theme-default.xml` come palettes: a bunch of presets for colors, backgrounds, fonts and other css attributes. Nuxeo EP 5 supports text palettes and GIMP/Photoshop palettes (for the colors).

When you add images or modify `theme-default.xml`, you have to redeploy your Nuxeo 5.

In case of doubt, try using the editor, because the produced code is much cleaner and compliant than anything you would write manually :-)

9.4.1. Modifying the current theme using `theme-default.xml`

The file `theme-default.xml` is structured as follows:

- Pages and their layout
- widgets in pages
- definition of predefined styles (using preset values from palettes)
- cell and fragments styling

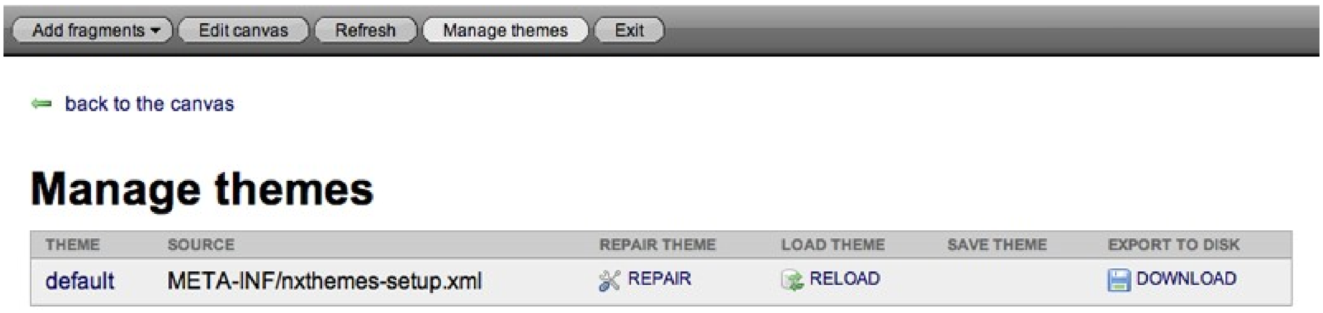
In `theme-contrib.xml` we have our theme called:

```
<!--  themes -->
<extension target="org.nuxeo.theme.services.ThemeService" point="themes">

  <theme>
    <src>META-INF/theme-default.xml</src>
  </theme>

</extension>
```

In the editor, in Manage Themes tab, it gives:



Manage themes

THEME	SOURCE	REPAIR THEME	LOAD THEME	SAVE THEME	EXPORT TO DISK
default	META-INF/nxthemes-setup.xml	REPAIR	RELOAD		DOWNLOAD

This file is deployed in JBoss. If you modify the theme using the editor all changes will be lost so think of downloading the theme to your Desktop, to replace the `theme-default.xml` in you local copy of Nuxeo EP 5.

A good way of working with this file is to add your working copy in `theme-contrib.xml`. It is possible in NXThemes to load several themes and page.

Add your file(s) in `themes-contrib.xml`, for example:

```
<theme>
  <src>file:///home/tsoulcie/NxCheckout/trunk/org.nuxeo.ecm.platform/nuxeo-platform-webapp-core/src/main/resourc
</theme>
```

After a redeployment, in the 'Manage Themes' section we now have a theme that can be reloaded directly from the file-system!

Starting from there, here are two ways of developing smartly:

- Edit your `theme-default.xml` in Eclipse (or in your XML editor) then go to theme editor, in Manage Themes tab and click "Reload" on your local file: you can directly see the changes you made in the XML source.
- Modify the theme inside the editor, then go to Manage Themes tab and click on "Save" action. All changes will be saved in the file.

9.4.1.1. Playing with palettes.

The palettes are in `org.nuxeo.ecm.platform/nuxeo-platform-webapp-core/src/main/resources/themes.palettes/`

They are called in `org.nuxeo.ecm.platform/nuxeo-platform-webapp-core/src/main/resources/OSGI-INF/theme-contrib.xml`:

```
<!-- Styles presets -->
<extension target="org.nuxeo.theme.services.ThemeService" point="presets">

  <palette name="Nuxeo default fonts"
           src="themes/palettes/nxfonts.properties"
           category="font" />
  <palette name="Nuxeo psd colors"
           src="themes/palettes/nxcolors.aco"
           category="color" />
  <palette name="Nuxeo default backgrounds"
           src="themes/palettes/nxbackgrounds.properties"
           category="background" />

</extension>
```

There are 3 default palettes:

- `nxbackgrounds.properties` that specifies the banner's css background and the shadow under it
- `nxcolors.aco` that contains nuxeo default colors in a photoshop palette format
- `nxfonts.properties` that contains default css font, small and 4 levels of titles

The easiest way for you to customize yout Nuxeo EP 5 app is to modify the existing palettes!

For example, in `nxfonts.properties` change the line

```
default=11px Verdana, Arial, sans-serif
```

to

```
default=12px Courier, serif
```

Then all the fonts of the app will be changed to your new value!

We advise you to add your own color palette.

9.4.1.2. Definition of a predefined style

Currently in `nxthemes-setup.xml` we have a style named default buttons, which is defined as:

```
<style name="default buttons">
  <selector path="input.button">
    <background>url(/nuxeo/img/button_1.gif) 0 0 repeat-x #e3e6ea</background>
    <font preset="default (Nuxeo default fonts)"/>
    <margin>5px 10px 10px 0px</margin>
    <color>#000</color>
    <border-style>solid</border-style>
    <border-width>1px</border-width>
    <border-color>#ccc #666 #666 #ccc</border-color>
    <padding>2px 5px 2px 5px</padding>
    <cursor>pointer</cursor>
  </selector>
  <selector path="input.button:hover">
    <color preset="white (Nuxeo psd colors)"/>
    <font preset="default (Nuxeo default fonts)"/>
    <background>url(/nuxeo/img/button_2.gif) 0 0 repeat-x #3f89ef</background>
    <border-color>#0099ff #0066cc #0066cc #0099ff</border-color>
    <border-style>solid</border-style>
    <border-width>1px</border-width>
  </selector>
  <selector path="input.button[disabled=disabled], input.button[disabled=disabled]:hover">
    <color>#c1c1c1</color>
    <font preset="default (Nuxeo default fonts)"/>
    <background>url(/nuxeo/img/button_disabled.gif) 0 0 repeat-x #ebeff4</background>
    <border-color>#ccc #999 #999 #ccc</border-color>
    <cursor>default</cursor>
    <border-style>solid</border-style>
    <border-width>1px</border-width>
  </selector>
</style>
```

We can see that:

- a style that does not apply to an `element` is name

- inside this style, several HTML attributes/classes are called
- palette preset are called, such as the font attribute

Predefined styles are also a good way of efficiently changing the look of your application because you need to change the CSS only once!

9.4.1.3. Using a predefined style

Later in the file we notice that the 'user services' fragment takes the default buttons style preset:

```
<!-- user services -->
   <style
     element="page[1]/section[1]/cell[2]/fragment[1]|page[3]/section[1]/cell[2]/fragment[1]"
     inherit="default buttons">
```

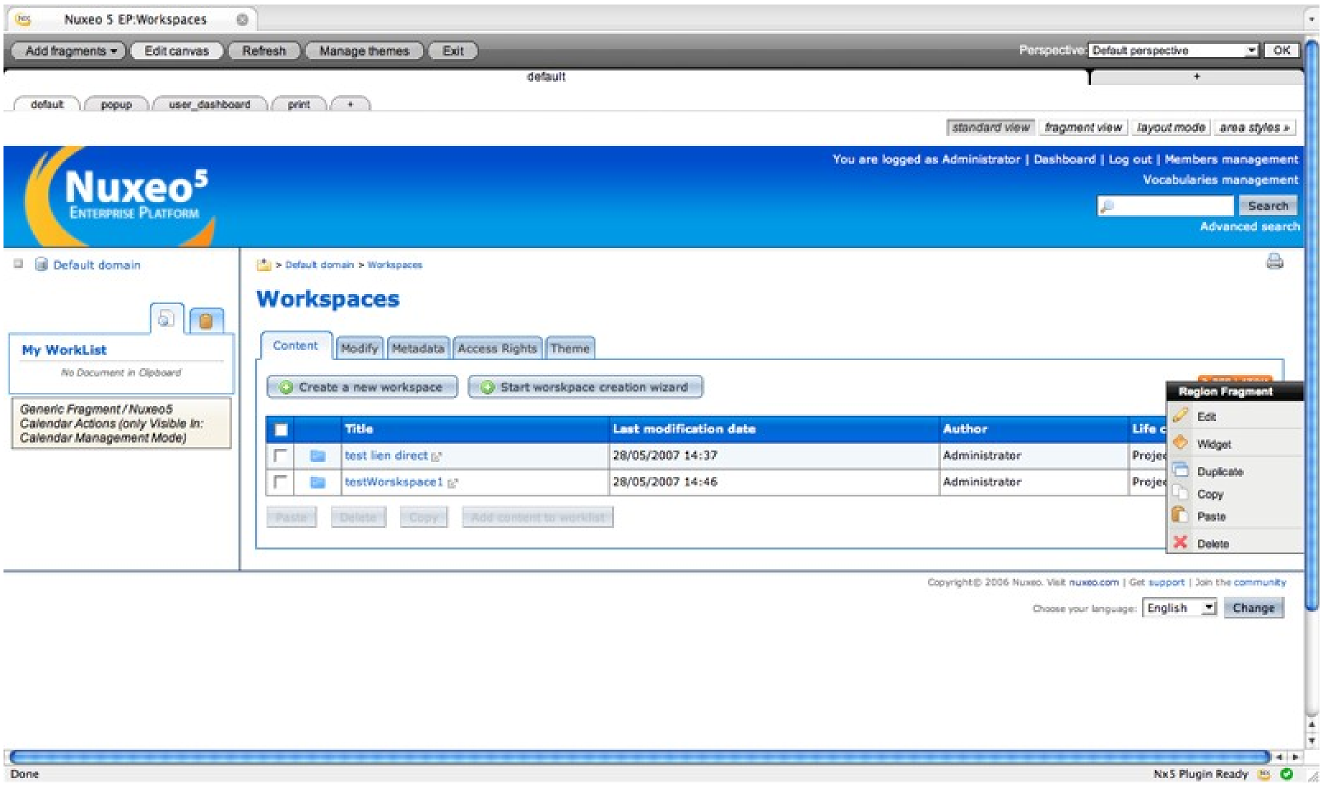
It means that the styles defined for the buttons will be applied to the 'user services' fragment (user links and search in the banner).

9.4.2. Modifying the current theme

9.4.2.1. Styling the theme using the editor

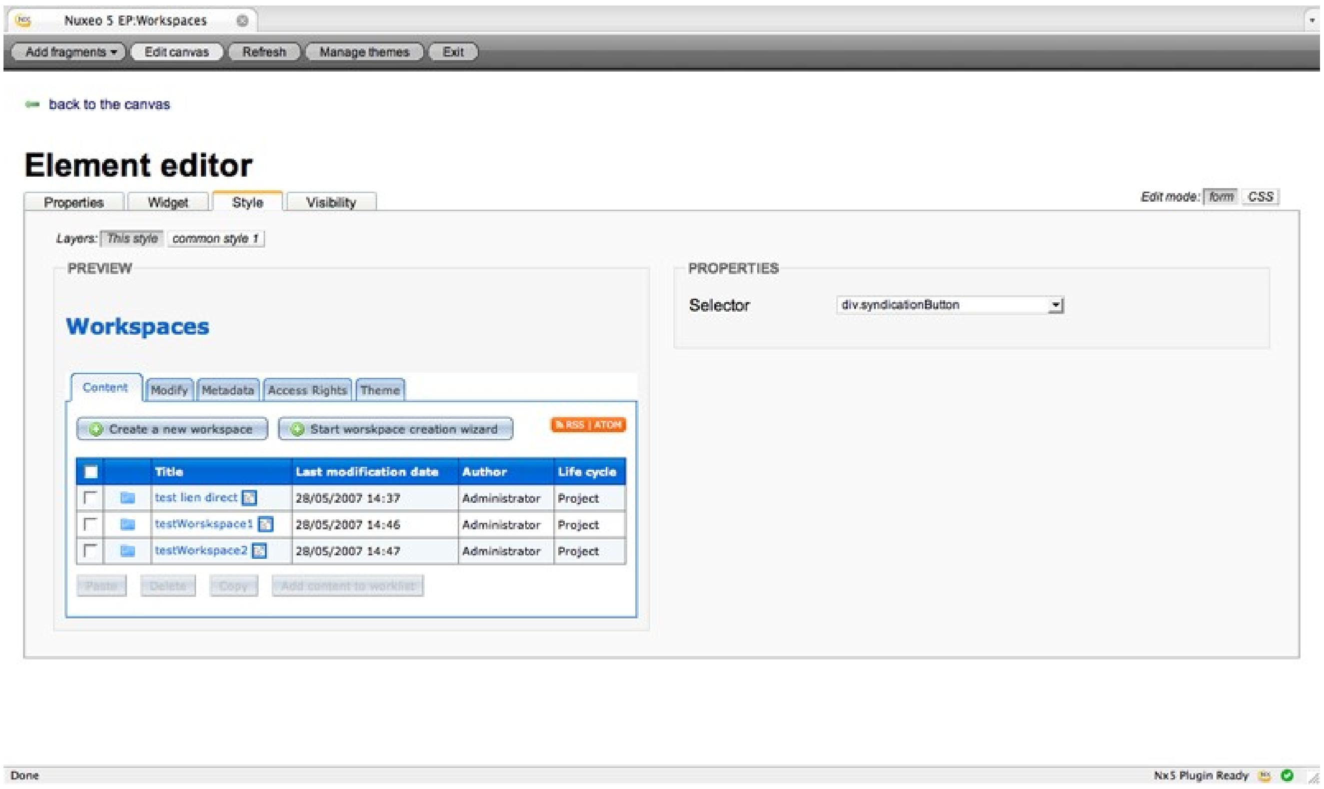
As we explained earlier, layout editing and local styling can be done in the theme editor.

In the editor, click on an element you want to style, click "Edit" in the Menu. Here we chose the RSS/Atom link button



Title	Last modification date	Author	Life c
test lien direct	28/05/2007 14:37	Administrator	Projec
testWorsksspace1	28/05/2007 14:46	Administrator	Projec

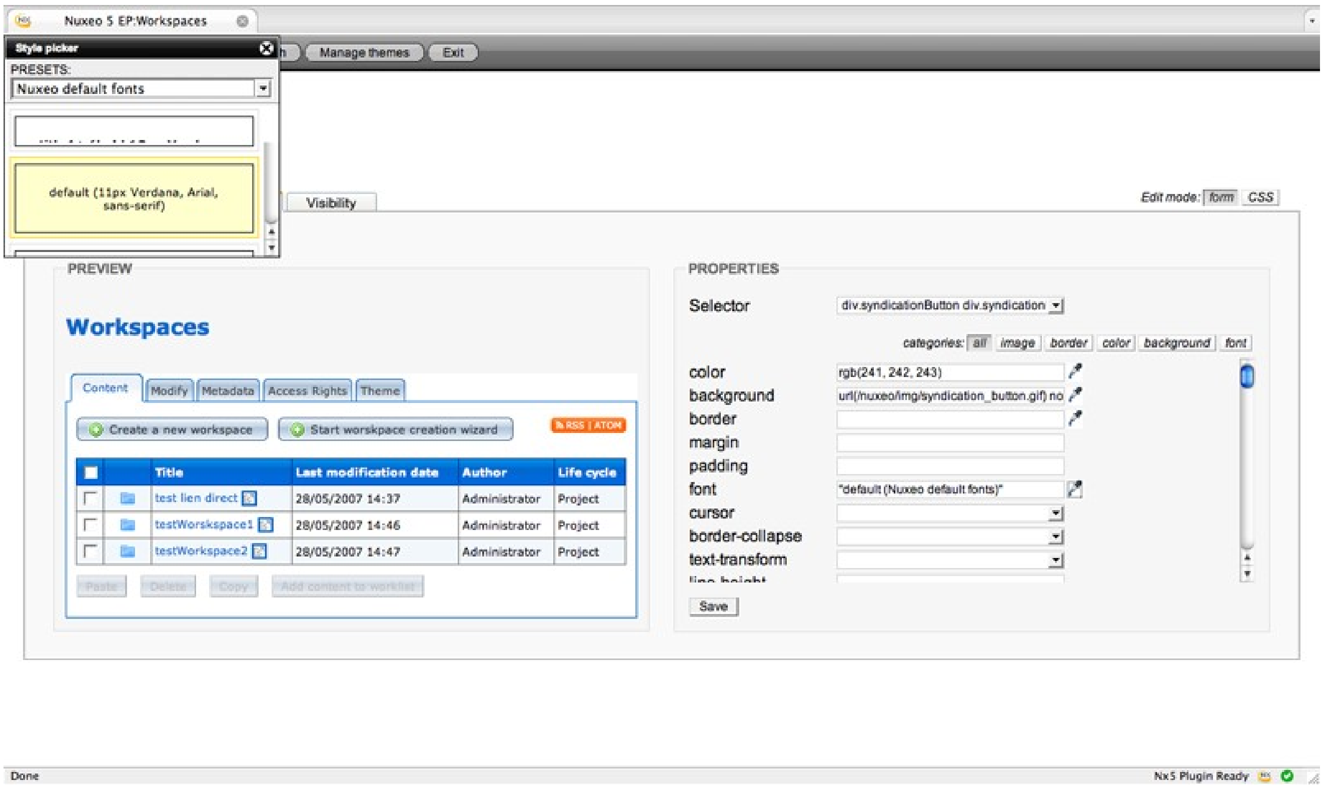
Access the Style tab.



Selector: div.syndicationButton

		Title	Last modification date	Author	Life cycle
		test lien direct	28/05/2007 14:37	Administrator	Project
		testWorskspace1	28/05/2007 14:46	Administrator	Project
		testWorkspace2	28/05/2007 14:47	Administrator	Project

The existing selectors are on the right in the Properties box, otherwise move the mouse over the preview area and click on an element to create a CSS selector path.

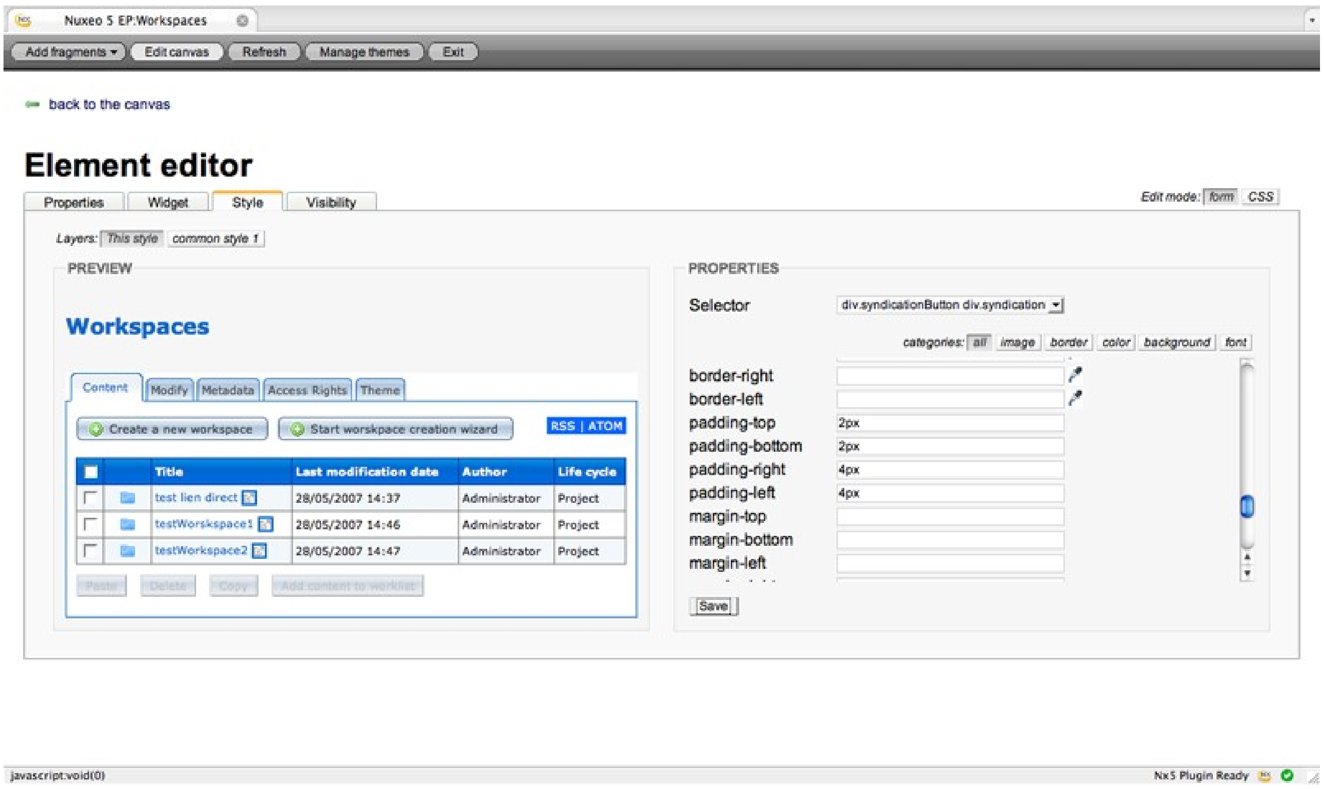


Selector: div.syndicationButton div.syndication

color	rgb(241, 242, 243)
background	url(/nuxeo/img/syndication_button.gif) no
border	
margin	
padding	
font	"default (Nuxeo default fonts)"
cursor	
border-collapse	
text-transform	

		Title	Last modification date	Author	Life cycle
		test lien direct	28/05/2007 14:37	Administrator	Project
		testWorskspace1	28/05/2007 14:46	Administrator	Project
		testWorkspace2	28/05/2007 14:47	Administrator	Project

We choose to change the small font preset to the default one. As you see, the Style picker shows all the palettes and all the presets are rendered. We remove the background property for the syndication links button and add a preset background-color, our RSS/Atom button is all changed now:



Element editor

Workspaces

Content | Modify | Metadata | Access Rights | Theme

Create a new workspace | Start worskpace creation wizard | RSS | ATOM

	Title	Last modification date	Author	Life cycle
	test lien direct	28/05/2007 14:37	Administrator	Project
	testWorskspace1	28/05/2007 14:46	Administrator	Project
	testWorkspace2	28/05/2007 14:47	Administrator	Project

Paste | Delete | Copy | Add content to worklist

Selector	div.syndicationButton div.syndication
border-right	
border-left	
padding-top	2px
padding-bottom	2px
padding-right	4px
padding-left	4px
margin-top	
margin-bottom	
margin-left	

When you are done with managing your theme you might want to save it to your local copy of Nuxeo. Just go in the Manage Themes tab, download the custom theme to your computer, then put it in your repository.

Congratulation, you have just customized the Nuxeo EP theme!

9.4.2.2. Modifying the theme by adding/modifying a fragment

You may want to modify an existing fragment to customize your project, let's say you want your compagny logo instead of Nuxeo EP's and you own corporate links in the footer. We won't create & declare new fragments (as we saw, fragments and their resources are defined in `theme-contrib.xml`), we'll use the default-ones to override Nuxeo EP's, considering you have your own project using Nuxeo EP default as made in the *sample project*.

Here are the steps do to so:

- copy your logo (let's call it `corporate_logo.gif`) to `your.project/src/main/resources/nuxeo.war/img`
- copy and paste `logo.xhtml` and `footer.xhtml` from `org.nuxeo.ecm.platform/nuxeo-platform-webapp/src/main/resources/nuxeo.war/incl` to `your.project/src/main/resources/nuxeo.war/incl` so it's overridden when doing your *ant*.

  Tip

  This is a general principle for *nuxeo.war* folder. The contents of the */img/* folder of your app are the contents of Nuxeo EP's default `.../nuxeo.war/img` folder. Every specific resource in `your.project/.../nuxeo.war/img` come in addition of what is already in default `.../nuxeo.war/img` if non-existing there with same filename, or come **instead** of what is existing in default `.../nuxeo.war/img` if same filename.

- edit `logo.xhtml` that currently contains

```
<div xmlns:h="http://java.sun.com/jsf/html"
     xmlns:ui="http://java.sun.com/jsf/facelets"
     xmlns:f="http://java.sun.com/jsf/core"
     xmlns:t="http://myfaces.apache.org/tomahawk"
     xmlns:nxu="http://nuxeo.org/nxweb/util"
     xmlns:c="http://java.sun.com/jstl/core" class="menu">
```

```
  <div>
    <h:form>
      <h:commandLink action="#{navigationContext.goHome()}">
        <h:graphicImage value="#{logoHelper.logoURL}" alt="Nuxeo EP 5"
                        width="194" height="99" />
      </h:commandLink>
    </h:form>
  </div>
</div>
```

- change the line `<h:graphicImage value="#{logoHelper.logoURL}" alt="Nuxeo EP 5" width="194" height="99" />` for something like `<img src="/nuxeo/img/corporate_logo.gif" alt="My corporate logo" />` and save your changes
- edit `footer.xhtml` that currently contains

```
<div xmlns:h="http://java.sun.com/jsf/html"
     xmlns:ui="http://java.sun.com/jsf/facelets"
     xmlns:f="http://java.sun.com/jsf/core">

<ui:insert name="footer">
  Copyright <f:verbatim>&amp;copy;</f:verbatim>  2006 Nuxeo. Visit <!-- &nbsp; -->
  <h:outputLink value="http://www.nuxeo.com">
    <h:outputText value="nuxeo.com" />
  </h:outputLink> | Get <!-- &nbsp; --><h:outputLink value="http://www.nuxeo.com/en/services/support/">
  <h:outputText value="support" />
    </h:outputLink> | Join the <!-- &nbsp; -->
  <h:outputLink value="http://www.nuxeo.org/sections/community">
    <h:outputText value="community" />
  </h:outputLink>
  <br />
  <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>
</ui:insert>
</div
```

- Change from `Copyright` to `<br />` by something like `<a href="http://yoursite.com">My Corporate Site</a>`, save your changes
- Do an *ant* on your projet, rerun your *jboss* and appreciate the changes...

Congratulation, you have just customized some Nuxeo EP fragments!

9.4.3. Adding a new theme and its pages

to be done.

Chapter 10. Authentication, Users & Groups Management

10.1. Introduction

In Nuxeo EP, the concept of a *user* is needed for two main reasons:

- Users are needed for authentication and authorization to work,
- Users have associated information that can be displayed, for instance to display someone's full name or email address.

An abstraction, the *`UserManager`*, centralizes the way a Nuxeo EP application deals with users (and groups of users). The `UserManager` is queried by the platform's *`LoginModule`* when someone attemps to authenticate against the framework. It is also queried whenever someone wants the last name or email of a user for instance, or to get all users having "Bob" as their first name.

10.2. Users and Groups configuration

The data about users (login, password, name, personal information, etc.) and the groups they belong to (simple members, or any application-related group) are managed through the Directory abstraction. This means that users can be stored in LDAP or in SQL, and groups can be part of the LDAP tree or stored in a SQL table, but the application doesn't see the difference as long as the connectors are configured properly.

10.2.1. Schemas

To define a new source of users, you simply define a new directory (or redefine the default one) to use different connection and schema information. We'll use an example where the pet-name field is added to the user's schema.

Let's start by defining our new schema in a `.xsd` file, `myuser.xsd`:

```
<?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 must be registered in an extension point:

```
<extension target="org.nuxeo.ecm.core.schema.TypeService" point="schema">
  <schema name="myuser" src="myuser.xsd" />
</extension>
```

TODO: FG groups

10.2.2. Directories

The user schema can now be used when we define a new directory, `MyUserDirectory`. A SQL directory is defined like this:

```
<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:

```
<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>

<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>
```

Detailed configuration on SQL Directories and LDAP Directories can be found in Chapter 14, *Directories and Vocabularies*.

10.2.3. UserManager

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>
    <defaultAdministratorId>Administrator</defaultAdministratorId>
    <defaultGroup>members</defaultGroup>
    <userListingMode>search_only</userListingMode>
    <users>
      <directory>MyUserDirectory</directory>
      <emailField>email</emailField>
      <searchFields>
        <searchField>username</searchField>
        <searchField>firstName</searchField>
        <searchField>lastName</searchField>
        <searchField>myfield</searchField>
      </searchFields>
    </users>
  </userManager>
</extension>
```

10.2.4. User Management Interface

Finally, because the different fields of a user are visible when creating or viewing them through the user management screens, we redefine the layout of two standard document types, `User` and `UserCreate` (which are used in the default user management screens and backing beans) to add our 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>
```

10.3. Authentication

10.3.1. Authentication Framework Overview

Nuxeo Authentication is based on the JAAS standard.

Authentication infrastructure is based on 2 main components :

- a JAAS Login Module: NuxeoLoginModule

- a Web Filter: NuxeoAuthenticationFilter

Users and groups and managed via the UserManagerService that handles the indirection to users and groups directories (SQL or LDAP).

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

10.3.2. Pluggable JAAS Login Module

NuxeoLoginModule is a JAAS LoginModule. Its is responsible for handling all login call 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.

Here is the default Jboss security configuration:

```
<domain name="nuxeo-ecm-web">
     <login-module code = "org.nuxeo.ecm.platform.login.NuxeoLoginModule"
            flag = "required">
             <option name="principalClassName">org.nuxeo.ecm.platform.login.NuxeoPrincipal</option>
             <option name="useUserIdentificationInfoCB">true</option>
         </login-module>
      <login-module code="org.jboss.security.ClientLoginModule" flag="required">
           <option name="password-stacking">true</option>
           <option name="restore-login-identity">true</option>
           <option name="multi-threaded">true</option>
      </login-module>
</domain>
<domain name="nuxeo-ecm">
       <login-module code = "org.nuxeo.ecm.platform.login.NuxeoLoginModule" flag = "required">
             <option name="principalClassName">org.nuxeo.ecm.platform.login.NuxeoPrincipal</option>
             <option name="useUserIdentificationInfoCB">true</option>
        </login-module>
</domain>
```

As shown by this configuration, the principals returned by NuxeoLoginModule is org.nuxeo.ecm.platform.login.NuxeoPrincipal.

Each protected service declares the nuxeo-ecm security domain

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss>
  <enterprise-beans>
    <session>
        <ejb-name>DocumentManagerBean</ejb-name>
        <security-domain>nuxeo-ecm</security-domain>
    </session>
  </enterprise-beans>
</jboss>
```

10.3.2.1. NuxeoLoginModule

NuxeoLoginModule mainly handles 2 tasks:

- login user

  This means extract information from the CallBack stack and validate 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 conjonction with UserIdentificationInfoCallback (Nuxeo custom CallBack system), the LoginModule will choose the right LoginPlugin according to the CallBack information.

10.3.2.2. NuxeoLoginModule Plugins

Because validating User identity can be more complexe that just checking login/password, NuxeoLoginModule exposes an extension point to contribute new LoginPlugins

Each LoginPlugin as to implement the org.nuxeo.ecm.platform.login.LoginPlugin interface.

Main method is:

```
 String validatedUserIdentity(UserIdentificationInfo userIdent)
```

that is used to validate the UserIdentificationInfo.

Typically, defaut 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 system could be implemented using this plugin system, most SSO implementations have be moved to the Authentication Plugin at the Web Filter level, because they need a http dialog.

For now, the NuxeoLoginModule has only two way to handle validateUserIdentity:

- default

  Uses UserManager

- Trusted_LM

  This plugin assumes the user identity has already been validated by the authentication filter, so the validatedUserIdentity will always return true. Using this LoginModule plugin, a user will be logged if the user exists in the UserManager. This plugin is used for most SSO system in conjonction with a Authentication plugin that will actually do the work of validating password or token.

10.3.2.3. Remote Login to the EJB layer

The Login system can be used via Remote EJB calls.

You can login as System user:

```
LoginContext lc = Framework.login();
// do some service calls
lc.logout();
```

You can login using user / password:

```
LoginContext lc = Framework.login(userName, password);
// do some service calls
lc.logout();
```

You can also call the login method and pass it directly a CallBackHandler. This can be used in conjonction with org.nuxeo.ecm.platform.api.login.UserIdentificationInfoCallbackHandler.

10.3.3. Pluggable Web Authentication Filter

The Web Authentication filter is responsible for:

- guarding acces to web resources

  The filter can be parametrized 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.

- create the LoginContext

  This means creating the needed callBacks and call the JAAS Login

- store and reestablish login context

  In order to avoid recreating a login context for each request, the LoginContext is cached.

10.3.3.1. NuxeoAuthenticationFilter

The NuxeoAuthenticationFilter is one of the top level filter in Nuxeo Web Filters stack.

For each request it will try to find a existing LoginContext and create a RequestWrapper that will carry the NuxeoPrincipal.

If no existing LoginContext is found it will try to prompt the client for authentication information and will establish the login context.

If order to execute the task of prompting the client and retrieving UserIndetificationInfo, the filter will rely on a set of configured plugins.

Each plugin must:

- Implement org.nuxeo.ecm.platform.ui.web.auth.interfaces.NuxeoAuthenticationPlugin

  The two main methods are :

```
Boolean handleLoginPrompt(HttpServletRequest httpRequest,HttpServletResponse httpResponse, String baseURL);
UserIdentificationInfo handleRetrieveIdentity(HttpServletRequest httpRequest, HttpServletResponse httpResponse);
```

- Define the LoginModule plugin to use if needed

  Typically, SSO AuthenticationPlugin will do all the work and will use the Trusted_LM LoginModule Plugin.

- Define if stating URL must be saved

  AuthenticationPlugins that uses HTTP redirect in order to do the login prompt will let the Filter store the first acceced URL in order to cleanly redirect the user to the page he asked after the authentication is successful.

Additionnaly, AuthenticationPlugin can also implement the `org.nuxeo.ecm.platform.ui.web.auth.interfaces.NuxeoAuthenticationPluginLogoutExtension` if a specific processing must be done when loging out.

Here is a sample xml descriptor for registring 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 additionnal 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. This is, for example, useful for having users using Form based authentication and having RSS clients using Basic Authentication. Because of that AuthenticationPlugin must be ordered. For that purpous, NuxeoAuthenticationFilter uses a dedicated extension point that let you define the AuthenticationChain.

```
<component name="Anonymous.auth.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. Typically, when authentication is needed, the Filter will call the handleLoginPrompt method on the plugins in the order of the authentication chain. When prompt is done, it will call the handleRetrievIndetify in the same order.

Some AuthenticationPlugin may choose not to trigger the LoginPrompt in all cases, for example, the BasicAuthentication plugin, only generate the prompt for specific urls like RSS feeds or restlet calls. This allow the platform to be easily called by Restlets or RSS clients without bothering browser clients.

10.3.3.2. Built-in Authentication Plugins

NuxeoAuthenticationFilter comes with two built-in authentication plugins :

- FORM_AUTH: Form based Authentication

  This is a standard form based authentication. Current implementation let 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 standatd Http Basic Authentication. By default, this plugin only generates the authentication prompt on configured URLs.

There are also additionnal components that provides other Authentication plugins (see below).

10.3.3.3. Additionnal Authentication Plugons

Nuxeo provides a set of other authentication plugins that are not installed by default with the standard Nuxeo5 EP setup. These plugins can be downloaded and installed separatly.

10.3.3.3.1. CAS2 Authentication

This plugin implements a client for CAS2 SSO system (Central Authentication System V2) :

To install this authentication plugin, you need to :

- download the nuxeo-platform-login-cas2 plugin
- put it in $JBOSS_HOME/server/default/deploy/nuxeo.ear/plugins and restart the server
- configure the CAS2 descriptor
- put CAS2 plugin into the authentication chain

In order to configure CAS2 Auth, you need to create an XML configuration file into $JBOSS_HOME/server/default/deploy/nuxeo.ear/config

Here is a sample file named CAS2-config.xml.

```
<component name="MyAPP.Cas2SSO">
   <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>
```

10.3.3.3.2. PROXY_AUTH: Proxy based Authentication

This plugin assumes Nuxeo5 is behind a authenticating reverse proxy that transmit user identity using http headers. This modules has be used on projetcs that uses a apache reverse proxy using client certificates to authenticate. SSO system (Central Authentication System V2) :

To install this authentication plugin, you need to :

- download the nuxeo-platform-login-mod_sso plugin
- put it in $JBOSS_HOME/server/default/deploy/nuxeo.ear/plugins and restart the server
- configure the plugin via an XML descriptor
- put the plugin into the authentication chain

In order to configure this plugin, you need to create an XML configuration file into $JBOSS_HOME/server/default/deploy/nuxeo.ear/config

Here is a sample file named proxy-auth-config.xml.

```
<component name="MyAPP.Mod_sso">
   <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>
```

10.3.3.3.3. NTLM_AUTH: NTLM and IE challenge/response authentication

This plugin uses JCIFS to handle NTLM authentication.

This plugging was partially contributed by Nuxeo EP users and has been reported to work by several users.

If you have troubles with lastest 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, you need to :

- download the nuxeo-platform-login-ntlm plugin
- put it in $JBOSS_HOME/server/default/deploy/nuxeo.ear/plugins and restart the server
- configure the plugin via an XML descriptor
- put the plugin into the authentication chain

In order to configure this plugin, you need to create an XML configuration file into $JBOSS_HOME/server/default/deploy/nuxeo.ear/config

Here is a sample file named ntlm-auth-config.xml.

```
<component name="MyAPP.NTLM">
   <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
         <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>
```

10.3.3.3.4. 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 Nuxeo5 server. See Nuxeo-Http-client-library for more information.

To install this authentication plugin, you need to :

- download the nuxeo-platform-login-portal-sso plugin
- put it in $JBOSS_HOME/server/default/deploy/nuxeo.ear/plugins and restart the server
- configure the plugin via an XML descriptor
- put the plugin into the authentication chain

In order to configure this plugin, you need to create an XML configuration file into $JBOSS_HOME/server/default/deploy/nuxeo.ear/config

Here is a sample file named portal-auth-config.xml.

```
<component name="MyAPP.postal_sso">
   <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>
```

10.3.3.3.5. ANONYMOUS_AUTH: Anonymous authentication plugin

This plugin provides anonymous authentication. Users are automatically logged as a configurable Anonymous user. This modules also includes additionnal 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 install this authentication plugin, you need to :

- download the nuxeo-platform-login-portal-sso plugin
- put it in $JBOSS_HOME/server/default/deploy/nuxeo.ear/plugins and restart the server
- configure the plugin via an XML descriptor (define who the anonymous user will be)
- put the plugin into the authentication chain

In order to configure this plugin, you need to create an XML configuration file into $JBOSS_HOME/server/default/deploy/nuxeo.ear/config

Here is a sample file named anonymous-auth-config.xml.

```
<?xml version="1.0"?>
<component name="org.nuxeo.ecm.platform.login.anonymous.config">

  <!-- Make sure these components are read first -->
  <require>org.nuxeo.ecm.platform.ui.web.auth.defaultConfig</require>
  <require>org.nuxeo.ecm.platform.login.anonymous</require>

  <!-- 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>

  <!-- Override the default authentication chain present in
    nuxeo-platform-ui-web to add ANONYMOUS_AUTH. -->
  <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>
```

Chapter 11. Security Policy Service

11.1. Introduction

The Security Policy Service provides an extension point allowing the creation of additional security policies. It gives the possibility to create policies based for instance on the comparison between documents and user's metadata.

11.2. Architecture

The policies are defined in two places:

- in the core where you can perform custom checks before looking at the ACP.
- in the search where you can patch with your own tests the query before its evaluation.

11.2.1. CorePolicyService

The `org.nuxeo.ecm.core.security.CorePolicyService` interface provides the `checkPolicy(Document doc, NuxeoPrincipal principal, String permission)` method. It is invoked in a first place by the `SecurityService` when trying to access a document and returns `true` if the document is still available after this check. If not, the ACP are not inspected and the permission to access the document is denied.

11.2.2. SearchPolicyService

The `org.nuxeo.ecm.core.search.api.security.SearchPolicyService` interface provides the `applyPolicy(ComposedNXQuery nxqlQuery)` method. It is called by the `SearchServiceImpl` to modify the search query before its execution.

11.3. Policies registry

To register your security policy, you need to write a contribution specifying the class names of your implementations of these two interfaces. For example :

```
<?xml version="1.0"?>
<component name="com.example.security.policy"  version="1.0">
  <extension target="org.nuxeo.ecm.core.security.PolicyService" point="policy">
    <policy class="com.example.security.CorePolicyServiceImpl" type="core" />
    <policy class="com.example.security.SearchPolicyServiceImpl" type="search" />
  </extension>
</component>
```

11.4. Sample security policy

You can find at nuxeo-project-securitypolicy, an example of project to implement a metadata based security policy. This project provides the following changes :

- the fields accessLevel and accessExpired are added to the schema `user`. The first one is an integer and the second, a date. You can modify their values on the user modification panel.
- a new document type, SecureDoc, is created. He has an integer property classificationLevel from the new schema `securitypolicy`. You can modify its value throw a selectbox in the tab called "View Security".
- a security policy restricts the access to the documents of type SecureDoc. If the value of its property classificationLevel is greater than the value of the property accessLevel of the current user, the access is

denied. Moreover, if the date accessExpired is in the past, the value of accessLevel drops to zero.

Chapter 12. Workflow & BPM

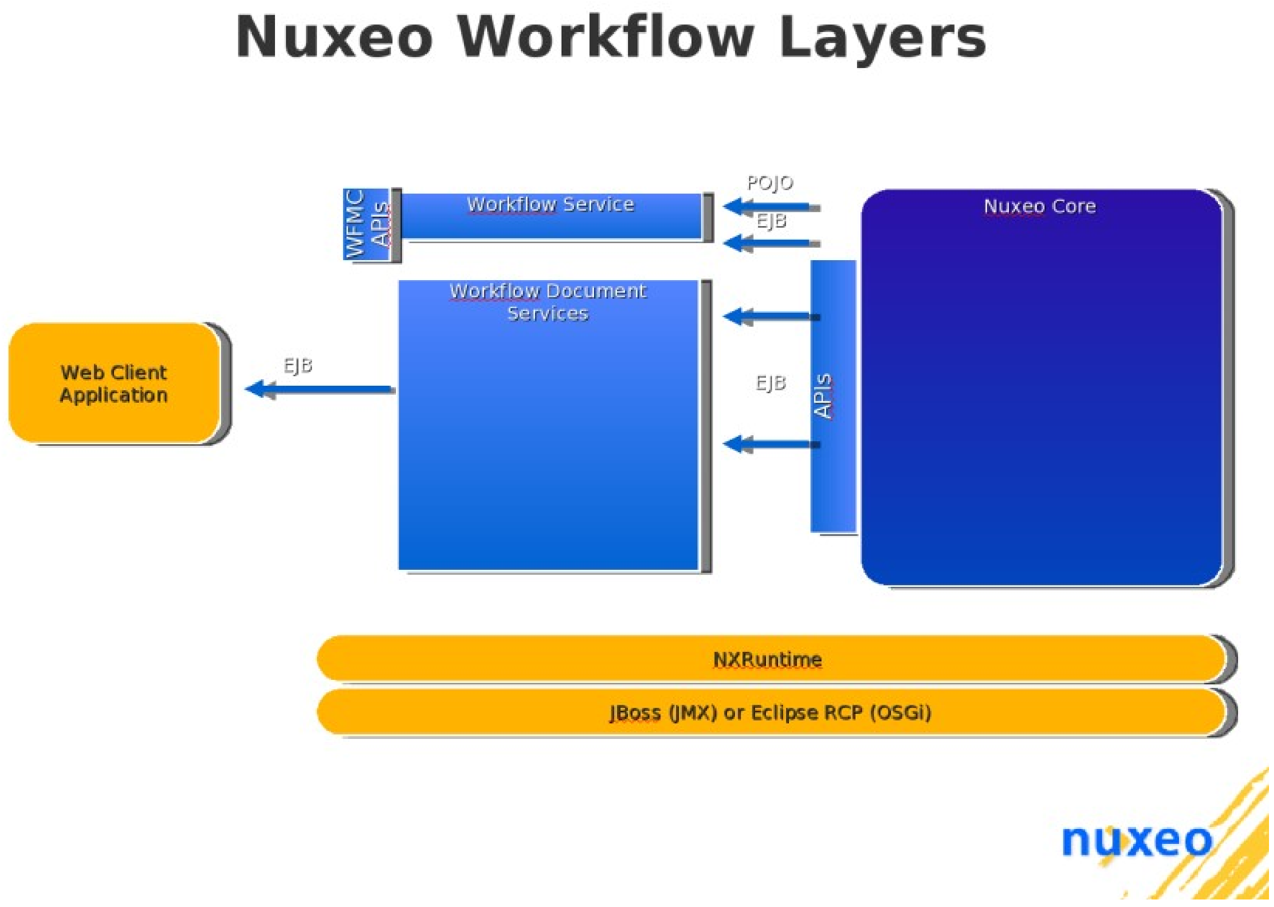
12.1. Introduction

TODO

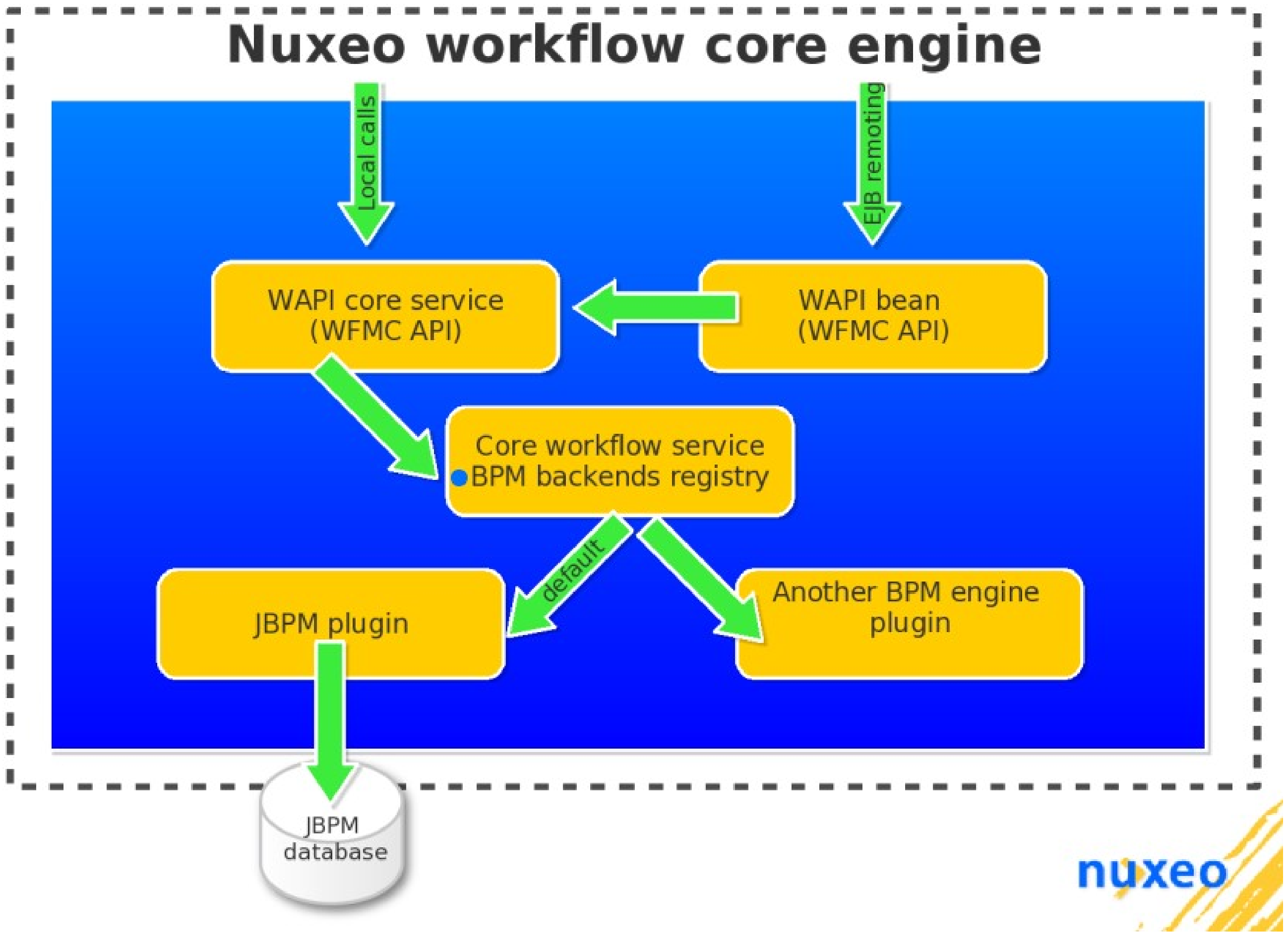
12.2. Architecture

TODO

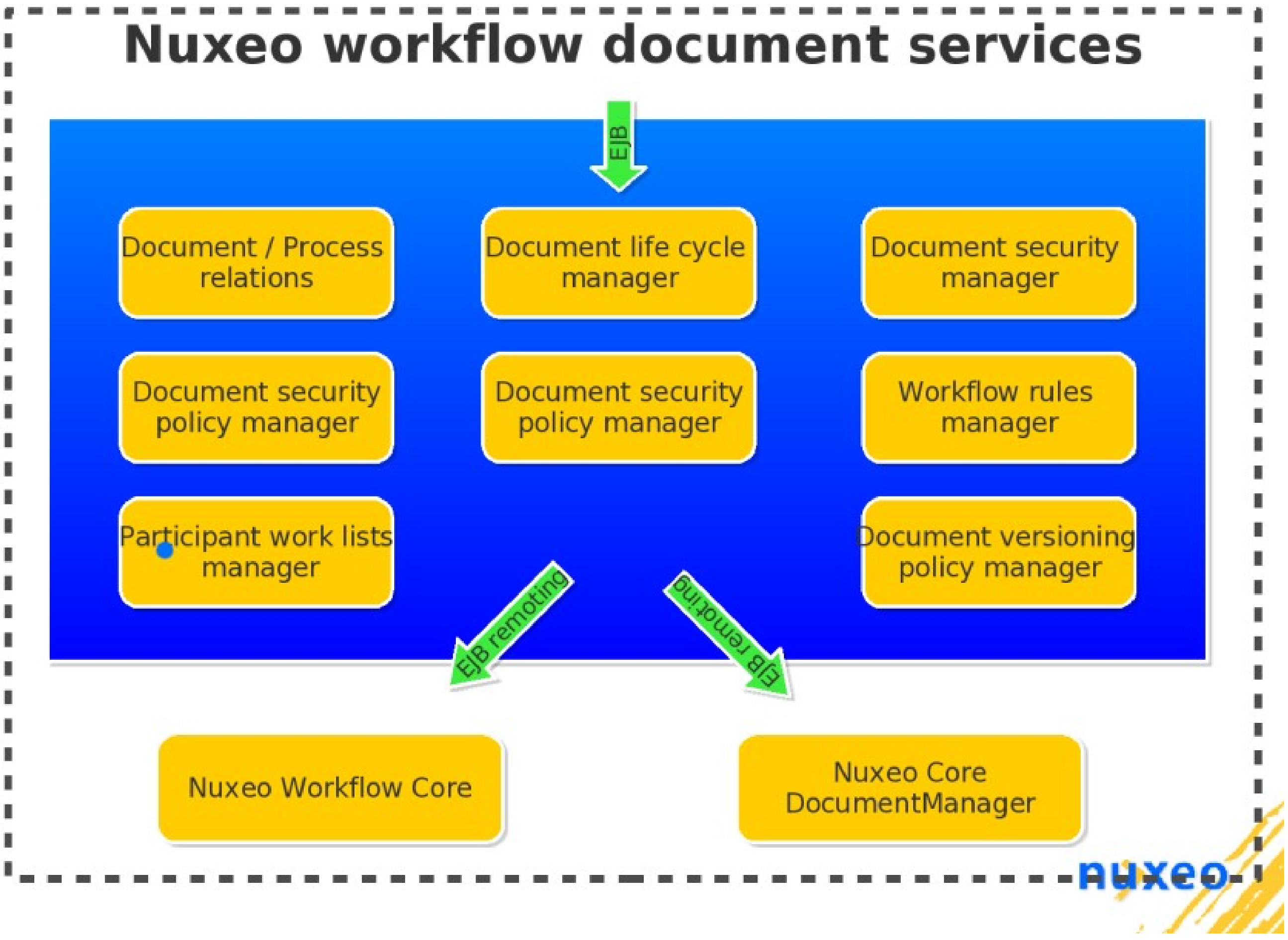
12.2.1. Big picture



12.2.2. Nuxeo Core Workflow



12.2.3. Nuxeo Workflow Document API and services


Nuxeo workflow document services
EJB
Document / Process relations
Document life cycle manager
Document security manager
Document security policy manager
Document security policy manager
Workflow rules manager
Participant work lists manager
Document versioning policy manager
EJB remoting
EJB remoting
Nuxeo Workflow Core
Nuxeo Core DocumentManager
nuxeo

12.3. Deploying process definitions

A workflow definition is designed for a given workflow engine backend and not for the Nuxeo workflow service itself. Nuxeo workflow doesn't specify a new process definition language. Thus it has no Nuxeo specificities speaking of format or process definition language. For instance, if you use jBPM as a backend with Nuxeo 5 then the workflow definition should be a standard jpdl file that you may have designed using your favorite editor or still if you are using Shark as a backend then the workflow definition will be a standard WFMC process definition.

Once your workflow definition has been designed and is ready you can deploy it in Nuxeo workflow. Of course, the target workflow engine backend plugin should be deployed and registered against the workflow service.

12.3.1. Using extension points

The Nuxeo workflow service provides a dedicated extension point for workflow definition deployment. The extension point is called **definition.**

In this case, the workflow definition will be deployed at application server deployment time (For now, this is the case when the application server is starting up since hot deployment is not yet possible using Nuxeo Runtime at the time of writing this document).It means this way of deploying workflow definition is not suitable for all cases. See the next subsections for other ways of deploying workflow definitions.

Below is an example of a jPDL workflow definition contribution for the jBPM backend. This XML fragment would be defined in a contribution registered as a component in a bundle:

```
<?xml version="1.0"?>
```

```
<component name="com.company.workflow.sample.contributions">

  <extension target="org.nuxeo.ecm.platform.workflow.service.WorkflowService"
           point="definition">
    <definition>
      <engineName>jbpm</engineName>
      <mimetype>text/xml</mimetype>
      <definitionPath>workflows/process_definition.xml</definitionPath>
    </definition>
  </extension>

</component>
```

- **engineName:** name specified for the target backend at workflow service registration time (see workflow service backend extension point)
- **mimetype:** mimetype of the workflow definition. This is especially interesting in case of the format is binary. (serialization issue at deployment time)
- **definitionPath:** bundle relative path of the workklow definition to deploy.

In this situation here is how would look the tree:

```
com.company.workflow /
  META-INF /
    workflows /
      process_definition.xml
      MANIFEST.MF
  OSGI-INF /
    workflow-definitions-contrib.xml
```

12.3.2. Using the workflow POJO service

TODO: provides code sample

12.3.3. Using EJB remoting

TODO: provides code sample

12.4. Nuxeo Core document integration

TODO

12.4.1. Security policy

12.4.2. Application level rules and filters

12.5. Document Versioning

12.5.1. Setting the version of a document

The versioning information are stored in the DocumentModel under the fields major_version and minor_version of the `uid.xsd` schema. Set the version as you would any other field:

```
documentModel.setProperty("uid","major_version",1);
documentModel.setProperty("uid","minor_version",1);
```

The field used for version is adaptable via the `properties` extension point of the `org.nuxeo.ecm.platform.versioning.service.VersioningService` component. It allows to define which properties should be used to set versions for a given document type.

```
<versioningProperties>
    <majorVersion>my:major_version</majorVersion>
    <minorVersion>my:minor_version</minorVersion>
    <documentType>File</documentType>
    <documentType>Note</documentType>
  </versioningProperties>
```

12.5.2. Modifying automatically the version of a document

Event such as saving a document or following a life cycle transition can change the document's version number. To use this feature you need first to define which event should be listened to and then how the version number should behave.

The `CoreEventListenerService` is used to define which event to listen to. The default declaration is in `nuxeo-platform-versioning-core`:

```
<listener name="versioninglistener"
    class="org.nuxeo.ecm.platform.versioning.listeners.DocVersioningListener">
  <eventId>lifecycle_transition_event</eventId>
  <eventId>documentCreated</eventId>
  <eventId>beforeDocumentModification</eventId>
  <eventId>documentUpdated</eventId>
  <eventId>documentRestored</eventId>
</listener>
<listener name="versioningChangelistener"
class="org.nuxeo.ecm.platform.versioning.listeners.VersioningChangeListener" />
```

The class `org.nuxeo.ecm.platform.versioning.listeners.DocVersioningListener` implements the behavior of the versioning when those events happen.

To modify the version of a document when a life cycle state is reach you need to define rules with the behavior(increment major, minor or do nothing) using the extension `rules` from `org.nuxeo.ecm.platform.versioning.service.VersioningService`:

```
<versioningRuleEdit name="sampleEditRuleProject" action="ask_user"
lifecycleState="project">
<option value="no_inc" default="true" />
<option value="inc_minor" />
<option value="inc_major" />
</versioningRuleEdit>
```

The default behavior for all type but File and Note is to increase the minor version for each life cycle change. You need to override one of the default rules if you add a new type. The order is important, the list of rules is read and the first match is used.

```
<versioningRuleEdit name="sampleEditRuleAnyState" action="ask_user"
      lifecycleState="*">
      <includeDocType>File</includeDocType>
      <includeDocType>Note</includeDocType>
      <includeDocType>MyNewType</includeDocType>
      <option value="no_inc" default="true" />
      <option value="inc_minor" />
      <option value="inc_major" />
    </versioningRuleEdit>
```

12.5.3. Accessing document from previous version

The `CoreSession` or the seam component `documentVersioning` and `versionedActions` allow to access document from previous version. On `CoreSession`, the method `getVersions` return all the versions of a document.

12.5.4. The versioning service implementation

The versioning service manages the version inside Nuxeo. Two implementations are available:

- The custom Nuxeo service, more flexible and used by default
- The default JackRabbit implementation.

To modify this setting edit the file `config/default-versioning-config.xml`:

```
<!--property name="versioningService" value="org.nuxeo.ecm.core.repository.jcr.versioning.JCRVersioningService"/
<property name="versioningService" value="org.nuxeo.ecm.core.versioning.custom.CustomVersioningService"/>
```

This section shows how to use the versioning module, how to modify the version of a document automatically or manually and how to use a document from a previous version.

12.6. jBPM integration

TODO

12.6.1. Process definition deployment

jBPM comes with a simple servlet allowing one to deploy jPDL process definitions and monitor running process instances. (NB: this is not yet activated on stock Nuxeo 5 instance).

12.6.2. Business handlers and Nuxeo Core integration

12.7. Nuxeo Workflow Web Client

TODO

12.8. Auditing workflow related events

TODO

12.9. Notification on workflow related events

TODO

12.10. Example of a document review process

TODO

Chapter 13. Logging and Audit Service

13.1. Introduction

XXX TODO: JA

13.2. Features

13.3. Architecture

13.4. Extend the audit service

13.4.1. Register new events to log

13.4.2. Customize audit entries schema

Chapter 14. Directories and Vocabularies

14.1. Introduction

TODO OG - General overview of the directory concept and goals

14.2. Directory with a Relational Database Management System (SQL) server as backend

TODO OG

14.3. Directory with an LDAP server as backend

TODO OG

14.4. Handling references between directory entries

TODO OG

14.4.1. References defined by a many-to-many SQL table

TODO OG

14.4.2. Static reference as a dn-valued LDAP attribute

TODO OG

14.4.3. Dynamic reference as a ldapUrl-valued LDAP attribute

TODO OG

14.4.4. Defining inverse references

TODO OG

14.5. Combining multiple directories into a single virtual directory

TODO FG

14.6. The Directory API

TODO OG

14.7. Building custom option lists in forms with vocabularies

TODO OG

Chapter 15. Mimetype detection

15.1. Introduction

The `org.nuxeo.ecm.platform.mimetype.*` packages give all the tools to find the mimetype of a document. The package provides two guessing approach: using file extensions and using the guessing library Jmimemagic (third party tool providing been enhanced detection methods, based on the binary signature of files).

15.2. MimetypeRegistry

All the recognized mimetypes are stored in the `MimetypeRegistry`. Each mimetype definition is a contribution to the `mimetype` extension point of the `org.nuxeo.ecm.platform.mimetype.service.MimetypeRegistryService` component.

```
<component
  name="org.nuxeo.ecm.platform.mimetype.service.MimetypeRegistryService">
  <extension
    target="org.nuxeo.ecm.platform.mimetype.service.MimetypeRegistryService"
    point="mimetype">
    <mimetype normalized="application/vnd.oasis.opendocument.text"
      binary="true" iconPath="odt.png" oleSupported="true">
      <mimetypes>
        <mimetype>application/vnd.oasis.opendocument.text</mimetype>
      </mimetypes>
      <extensions>
        <extension>odt</extension>
      </extensions>
    </mimetype>
  </extension>
</component>
```

A `mimetype` node, bound to a `MimetypeDescriptor` defines a normalized mimetype with the following informations:

- `normalized`: the mimetype entry that is described and that will be returned
- `binary`: a boolean that indicates if the file is a binary one
- `iconPath`: the filename of the image representing the icon
- `oleSupported`: this file mimetype is supported by the `oleExtract` transform plugin - default is `False`
- `onlineEditable`: this mimetype is supported by `online Edit` - default is `False`
- `mimetypes`: list of mimetypes bound to this normalized mimetype
- `extensions`: list of extensions bound to this normalized mimetype

An other defined extension point is `extension` that allow to register extensions that are ambiguous to force mimetype sniffing.

```
<component
  name="org.nuxeo.ecm.platform.mimetype.service.MimetypeRegistryService">
  <extension
    target="org.nuxeo.ecm.platform.mimetype.service.MimetypeRegistryService"
    point="extension">
    <fileExtension name="xml" mimetype="text/xml" ambiguous="true" />
  </extension>
</component>
```

A `fileExtension` node, bound to `ExtensionDescriptor`, has the following attributes:

- `name`: the file extension

- `mimetype`: the associated mimetype
- `ambiguous`: force the mimetype sniffing

At the `MimetypeRegistry` level, methods are provided to dynamically register (or unregister) mimetypes or extensions by code

```
public void testMimetypeRegistration() {
    MimetypeEntry mimetype = getMimetypeSample();
    mimetypeRegistry.registerMimetype(mimetype);
    assertEquals(
            mimetypeRegistry.getMimetypeEntryByName(mimetype.getNormalized()),
            mimetype);
}

private static MimetypeEntryImpl getMimetypeSample() {

    String normalizedMimetype = "application/msword";

    List<String> mimetypes = new ArrayList<String>();
    mimetypes.add("application/msword");
    // fake
    mimetypes.add("app/whatever-word");

    List<String> extensions = new ArrayList<String>();
    extensions.add("doc");
    extensions.add("xml");

    String iconPath = "icons/doc.png";

    boolean binary = true;
    boolean onlineEditable = true;
    boolean oleSupported = true;

    return new MimetypeEntryImpl(normalizedMimetype, mimetypes, extensions,
            iconPath, binary, onlineEditable, oleSupported);
}
```

The `registerMimetype` method, with a `MimetypeEntry` argument, adds the given entry to the registry. After adding the mimetype, one can see that the `MimetypeEntry` retrieved using the initial Normalized name is correct. The `getMimetypeSample` method, helper for the `TestMimetyepRegistryService` test class, shows a definition of a `MimetypeEntry` by code.

Based on the previous entry definition, two registries are built allowing to retrieve the normalized mimetype through file extension (efficient) or by sniffing (accurate).

15.3. Mimetype sniffing

This section describes the underlying process involved in mimetype detections.

The file extension detection rely on the filename that has to be correct. As it is a simple string to associate to the mimetype, we do not develop further. Just note that the filename has to be correctly named and have an extension to retrieve a mimetype with this method.

The sniffing tries to analyse the file content itself in order to guess the mimetype. The third-party tool `Jmimemagic` is used for this and widely enhanced on defining new supported mimetypes and detectors. The `Jmimemagic` tool uses 2 files, `magic.xml` and `magic_1_0.dtd`. We redefine the `XML` one to add new detections (no extension-point there defined from `Jmimemagic` too ;-) l).

Basically, the default mimetype sniffing is based on searching a sequence of characters (or binary values) at a specified `offset`.

```
<match>
  <mimetype>application/pdf</mimetype>
  <extension>pdf</extension>
  <description>PDF document</description>
  <test offset="0" type="string" comparator="=">%PDF-</test>
</match>
```

A `match` is the definition of a magic entry. It contains a `mimetype`, an `extension` and a textual `description` of the defined mimetype. A `test` node, containing the operation to perform is also defined. Here it declares that for an `application/pdf` mimetype, the file has to contain the string `%PDF-` at `offset 0`.

if this method is usually suitable for a lot of files (i.e. one can find some invariants in the format), when used with more complex ones, a simple offset (or a combination) is not enough and we have to refine the detection algorithm. That is what detectors are made for and we have defined some for the 2 major office file formats, MsOffice and OpenOffice.org.

For OpenOffice.org, the zip detection is enhanced

```
<match>
  <mimetype>application/zip</mimetype>
  <extension>zip</extension>
  <description>Zip archive data</description>
  <test offset="0" type="string" comparator="=">PK\003\004</test>
  <match-list>
  <!--  opendocument & OOo 1.x -->
    <match>
      <mimetype>OOo</mimetype>
      <extension>OOo</extension>
      <description>OOo 1.x and OpenDocument file</description>
      <test type="detector" offset="0" length="" bitmask="" comparator="=">
           org.nuxeo.ecm.platform.mimetype.detectors.OOoMimetypeSniffer
      </test>
    </match>
</match>
```

First a simple `offset` detection is performed to qualify a `zip` file, then a sub-match is defined. The detector type indicates that the `org.nuxeo.ecm.platform.mimetype.detectors.OOoMimetypeSniffer` has to be called and this is its responsibility to give all the valid information (`mimetype`, `extension`, `description`) if the file is of correct type.

For MS-Office files, the "magic numbers" (the value to be found at a certain offset) are not that clear, as the magic number defined by `Jmimemagic` (based on the `file` Linux command resource file) is the same for all MS-Office application. Then we invoke a detector for each component that uses the `POI` library to detect what file we deal with.

```
<match>
  <mimetype>application/msword</mimetype>
  <extension>doc</extension>
  <description>Microsoft Office Document</description>
  <test offset="0" type="string" comparator="=">\320\317\021\340\241\261</test>
  <match-list>
    <!--  XLS file by detector -->
    <match>
      <mimetype>application/vnd.ms-excel</mimetype>
      <extension>xls</extension>
      <description>Excel File</description>
      <test type="detector" offset="0" length="" bitmask="" comparator="=">
           org.nuxeo.ecm.platform.mimetype.detectors.XlsMimetypeSniffer
      </test>
    </match>
   <!-- PPT file by detector -->
    <match>
      <mimetype>application/vnd.ms-powerpoint</mimetype>
      <extension>ppt</extension>
      <description>Powerpoint File</description>
      <test type="detector" offset="0" length="" bitmask="" comparator="=">
           org.nuxeo.ecm.platform.mimetype.detectors.PptMimetypeSniffer
      </test>
    </match>
  </match-list>
</match>
```

Once a `Microsoft Office Document` has been detected at `offset 0` in the first match, two sub-matches detectors are defined for `application/vnd.ms-excel` (`org.nuxeo.ecm.platform.mimetype.detectors.XlsMimetypeSniffer`) and `application/vnd.ms-powerpoint` (`org.nuxeo.ecm.platform.mimetype.detectors.PptMimetypeSniffer`). If none returns a correct mimetype, the only possibility remains then `application/msword`. (This may be lightly refactored for simplicity in a near future).

A detector is a class that implements `net.sf.jmimemagic.MagicDetector`. The public `process` method has to detect if the file fulfills the condition. If successful, it returns the mimetypes supported by this file. The public methods `getHandledExtensions` and `getHandledTypes` define the `String` arrays that are used by Jmimemagic to build the final match.

15.4. Invoking the mimetype detection

All the detection is based on `MimetypeRegistry` like object. When invoked, the registry is populated with the information that has been exposed previously. The registry implements the interface `org.nuxeo.ecm.platform.mimetype.interfaces.MimetypeRegistry`.

Once available, directly or from a bean, any `File` or `Blob` can be analyzed and the information retrieved like the mimetype name or the supported extensions list.

```
import org.nuxeo.ecm.platform.mimetype.ejb.MimetypeRegistryBean;
...
    private MimetypeRegistryBean mimetypeRegistry;
...
    public void testSniffWordFromFile() throws Exception {

        File file = FileUtils.getResourceFileFromContext("test-data/hello.doc");

        String mimetype = mimetypeRegistry.getMimetypeFromFile(file);
        assertEquals("application/msword", mimetype);

        List<String> extensions = mimetypeRegistry.getExtensionsFromMimetypeName(mimetype);
        assertTrue(extensions.contains("doc"));
    }
```

In the above example, the `mimetypeRegistry` object used is a `MimetypeRegistryBean`. The purpose is to sniff the mimetype of a given file. The MS-Word file is first read and the `getMimetypeFromFile` method is called. Once the mimetype is retrieved, the `getExtensionsFromMimetypeName` can be called and it gives the associated extensions from the registry.

Note that the API of `org.nuxeo.ecm.platform.mimetype.interfaces.MimetypeRegistry` contains various ways to ask for a mimetype, dealing with `File` or `Blob` objects, with or without default responses. It is worth a look to avoid unneeded work.

Chapter 16. Content Transformation

16.1. Introduction

Transforms are operation that perform any action modifying the input document. It can cover file conversion as well as mail merging or information extraction.

16.2. Plugins module

The various plugins are stored in the `nuxeo-plateform-transforms-plugins`.* module. They are declared as a contribution of the component `org.nuxeo.ecm.platform.transform.service.TransformService` at extension point `plugins`.

The plugins are the engines that perform the needed transformations.

16.2.1. Creating a plugin

A transform plugin follows a framework defined in the `nuxeo-plateform-transforms-core` module. It is basically a class that extends a `org.nuxeo.ecm.platform.transform.interfaces.Plugin`. The main common method that has to be exposed is `transform`

```
     List<TransformDocument> transform(
             Map<String, Serializable> options, TransformDocument... sources);
```

The `transform` method returns a list of `TransformDocument` and accepts an `options map` and `TransformDocument sources`. `TransformDocument` object is defined in `org.nuxeo.ecm.platform.transform.interfaces.TransformDocument` and holds the binary as well as other information such as mimetype.

The `Map<String, Serializable> options` holds all the necessary options to be passed to the plugin. Please note that the keys are the plugin names. See `officeMerger` plugin below for an implementation example.

16.2.2. Declaring a plugin module

We first declare a contribution to the extension point `plugins` of `org.nuxeo.ecm.platform.transform.service.TransformService`:

```
<extension target="org.nuxeo.ecm.platform.transform.service.TransformService" point="plugins">
  <plugin name="any2odt"
      class="org.nuxeo.ecm.platform.transform.plugin.joooconverter.impl.JOOoConvertPluginImpl"
      destinationMimeType="application/vnd.oasis.opendocument.text">
    <sourceMimeType>text/xml</sourceMimeType>
    <sourceMimeType>text/plain</sourceMimeType>
    <sourceMimeType>text/rtf</sourceMimeType>

    <!-- Microsoft office documents -->
    <sourceMimeType>application/msword</sourceMimeType>

    <!-- OpenOffice.org 1.x documents -->
    <sourceMimeType>application/vnd.sun.xml.writer</sourceMimeType>
    <sourceMimeType>application/vnd.sun.xml.writer.template</sourceMimeType>

    <!-- OpenOffice.org 2.x documents -->
    <sourceMimeType>application/vnd.oasis.opendocument.text</sourceMimeType>
    <sourceMimeType>application/vnd.oasis.opendocument.text-template</sourceMimeType>

    <option name="ooo_host_name">localhost</option>
    <option name="ooo_host_port">8100</option>
  </plugin>
</extension>
```

The name attribute will be used to declare the transform chain. The class attribute is the class that will do the effective job of the transformation while the destinationMimeType is the mime-type of the result of the transform.

After attributes, <sourceMimeType> nodes define the allowed input mime-types the transform is supporting. In the presented case, we can see that the any2odt plugin will be able to handle text, Microsoft office word , OOo 1.x and OpenDocument format (OOo2.x) files to output an application/vnd.oasis.opendocument.text OpenDocument file. Options can also be added as we see for <option> tags with ooo_host_name and ooo_host_port attributes.

Plugins can be combined to build transform chains. This chains are declared in a transformer which is a contribution to the extension-point transformers of org.nuxeo.ecm.platform.transform.service.TransformService component.

```
<extension target="org.nuxeo.ecm.platform.transform.service.TransformService"
           point="transformers">
  <transformer name="any2text"
               class="org.nuxeo.ecm.platform.transform.transformer.TransformerImpl">
    <plugins>
      <plugin name="any2pdf"/>
      <plugin name="pdf2text"/>
    </plugins>
  </transformer>
</extension>
```

A transformer is defined by its name. This is this name that will be used to initialize a transform service when using it.

Then, plugins involved in the chain are listed. Wen can see that our any2txt transformer is composed with two chained plugins: any2pdf then pdf2txt. Obviously, a single plugin for a transform is legal as we can see with the use of our previous any2odt plugins.

```
<extension target="org.nuxeo.ecm.platform.transform.service.TransformService"
           point="transformers">
  <!-- This transformer uses a the OOo plugin to transform documents to ODT-->
  <transformer name="any2odt"
               class="org.nuxeo.ecm.platform.transform.transformer.TransformerImpl">
    <plugins>
      <plugin name="any2odt"/>
    </plugins>
  </transformer>
</extension>
```

16.2.3. Using a transform plugin

Once a transform plugin has been declared and the transformer is known, we can use it to perform various transformation actions.

```
TransformService service = NXTransform.getTransformService();
Transformer transformer = service.getTransformerByName("any2pdf");

List<TransformDocument> results = transformer.transform(null,
        new TransformDocumentImpl(sourceStream, "application/vnd.oasis.opendocument.text"));

SerializableInputStream resultStream = results.get(0).getStream();
```

We first get the TransformService that exposes all the available transforms. Then a specific transformer is built with the getTransformerByName method of the service. The name is the one that has been declared in the contribution to the transformers extension-point of the org.nuxeo.ecm.platform.transform.service.TransformService component.

Then the transformer exposes a transform method that returns a list of TransformDocument. The arguments are the:

- options: the plugin options (the keys are the plugin names) - we pass null here as we do not have any options. See `officeMerger` plugin for a more detailed example

- list of sources as `TransformDocument` instances

There are three levels of options that overload. First the `plugin` options are the default. Then any option in the `transformer` that define an option for this plugin overload them. Finally, any code-defined options are merged, overloading any previous option that may have been already defined. Please note again that options are defined on a plugin name basis.

A `TransformDocument` instance can be constructed from:

- a source stream
- a source stream and its mime-type
- a blob

In our example, we use the second way and give a `sourceStream` and the mime-type of an ODF document.

Once the input list processed, the results list contains all the transformed files as `TransformDocument` instances from which you can retrieve the `SerializableInputStream` stream with `getStream` method, the mime-type using `getMimetype` and the blob with `getBlob` method.

An alternate way of using the transform is to call it directly from the service

```
List<TransformDocument> results = service.transform(converter,
        null, new TransformDocumentImpl(stream, sourceMimetype));
```

The arguments are the same and the `converter` name is given as first argument. An other constructor using blobs as input instead of `TransformDocument` is also available. Before calling a transform we can also check that the source mime-type is supported by calling the `isMimetypeSupportedByPlugin` method. Be careful though that this plugin name may be different than the transformer name.

Transforms can be called directly but are also part of the docModifier framework that reacting on events, can call transforms to alter or generate new informations (see below oleExtract plugin or docModifier documentation)

16.3. Available transforms

All the transform plugin packages start with `org.nuxeo.ecm.platform.transform.plugin`. Some of them are optional and not included in the default platform.

16.3.1. Document conversion

The document conversion plugin is a generic transform delivered in Nuxeo that allow transforming a file from a format to an other. It uses the third-party JODconverter tool. The transform is implemented in the `org.nuxeo.ecm.platform.transform.plugin.joooconverter.impl.JOOoConvertPluginImpl` class and defined as a `plugin` contribution to `org.nuxeo.ecm.platform.transform.service.TransformService` .

Then it can be classically called as explained in the previous section:

```
TransformService service = NXTransform.getTransformService();
Transformer transformer = service.getTransformerByName("any2pdf");

List<TransformDocument> results = transformer.transform(null,
        new TransformDocumentImpl(sourceStream, "application/vnd.oasis.opendocument.text"));

SerializableInputStream resultStream = results.get(0).getStream();
```

The `transform` call relays to the `JOOoConvertPluginImpl` that first connects to OOo on port and host defined in the contribution (usually `localhost:8100`) or in the options (not defined in our example). It then acquire an `OpenOfficeDocumentConverter` from JODconverter tool and then can call the `convert` method with the requested target mimetype and source file. Note that in future version, the `StreamOpenOfficeDocumentConverter` class will be used to avoid dealing with `File` objects. This limitation will be solved when a new version of OpenOffice.org (2.3) will be out and solves a regression on loading streams.

Before any call to the underlying OpenOffice.org converter, once in the transformation engine `JOOoConvertPluginImpl`, the source document mimetype is tested. If it is the same than the requested `destinationMimetype` defined by the plugin, the source file is returned immediately as result unless the mimetype occurs in the `<sourceMimeType>` of the plugin to allow self-transformations. By default, `any2pdf` plugin will return immediately if an `application/pdf` file is submitted while any OpenDocument transformation (`any2odt`, `any2ods`, `any2odp`) will process the files as each one contains its own mimetype like `<sourceMimeType>application/vnd.oasis.opendocument.text</sourceMimeType>`. This allow to clean and validate files forged by hand and possibly apply automatic treatments at OpenOffice.org side.

16.3.1.1. Remote engine

Since OOo 2.3.0, it is possible to use `InputStreams` as source documents so that no more `File` objects are needed. This is useful if we want to isolate the OOo server on a separate machine (and if needed use a farm with load balancing if heavy work is needed) As this feature is only available with OOo2.3+ (due to a bug in previous versions), there is a `nuxeo.property` to define the OOo version used.

```
org.nuxeo.ecm.platform.transform.ooo.version=2.3.0
```

At term, the JODConverter tool should be able to return the OOo version it is using, so that this property will not be needed anymore. For the moment, let use this property. But as OOo stream loading has been reported as less efficient than `File`/`URL` by JODConverter users, the streaming method is only used if OOo is really a remote instance. This is analyzed if the `ooo_host_name` transformer option is '`localhost`' or starts with '`127.0.0`'. And of course, only if OOo declared version is greater than 2.3.0.

16.3.1.2. OpenOffice.org loading options

As the underlying engine is based on OpenOffice.org, one can extend the document converter by supporting OOo loading options. The transform plugins options mechanism is fully supported, so they can be defined at plugin, transformer and code level. The available fields to be passed to OOo is listed at MediaDescriptor IDL reference and have to be bound to the plugin name.

Here is an example passing two options for loading conditions of a special document that is a password protected and has an autostart bound event that delete the content of the file to put the word DELETED in the document. This simulate any other heavy process such as document merging or automatic mail-merge from a database.

```
Map<String, Map<String, Serializable>> options = new HashMap<String, Map<String, Serializable>>();
Map<String, Serializable> pdfOptions = new HashMap<String, Serializable>();

pdfOptions.put("Password", "TheDocumentPassWord");
short ALWAYS_EXECUTE_NO_WARN = 4;
pdfOptions.put("MacroExecutionMode", ALWAYS_EXECUTE_NO_WARN);
pdfOptions.put("ReadOnly", false);
pdfOptions.put("Hidden", false);
options.put("any2pdf", pdfOptions);

// Note: due to password, file mimetype can not be sniffed
// so mimetype has to supplied to TransformDocumentImpl
results = service.transform("any2pdf", options,
        new TransformDocumentImpl(getBlobFromPath(path), "application/vnd.oasis.opendocument.text"));
assertTrue(results.size() > 0);

// The macro on the open event replaces the text by "DELETED"
assertEquals("pdf content", "DELETED", DocumentTestUtils.readPdfText(pdfFile));
```

First, we build the plugin options map. The `Password` field is in charge of sending the password string to the OOo loader so that the file can be opened. The `MacroExecutionMode` field defines how macros and script are handled at startup. By default, the consequence of the `-headless` OOo mode, is that the `NEVER_EXECUTE = 0` value is used. One can change this default behaviour by using any value listed in the [OOo MacroExecMode constant group](). One important thing to be noted is that type is important: `Password` require a `String` argument while `MacroExecutionMode` requires a `short` one. The types have to be correct otherwise the field will not be handled by OOo.

By default, JODConverter sets the `ReadOnly` option as true. As we want to modify the document, we will have to set the `ReadOnly` flag to `false`. The `Hidden` flag has also to be set to false. This trick is due to a problem in OpenOffice.org PDF engine that seems not to be able to handle document modification while `Hidden`. As an headless server-deployed OOo instance, this should not be a major problem.

Once Options have been defined, they are globally merged to the options map under the plugin name key (here, `any2pdf`)

Then the transform can be called as usual. Please note that mimetype of password document have to be passed explicitly to the `TransformDocumentImpl` constructor as it can not be sniffed by the `Mimetype` service for the moment.

Finally, as expected a PDF document is returned with its content changed to the `DELETED` string.

16.3.2. Pdfbox

16.3.3. OLE objects extraction

OLE objects are objects included in office files that can be edited as standalone ones. For example, a spreadsheet table may be included in a report so that the presented datas are always up to date.

This plugin is located in `nuxeo-plateform-transform-plugin-oleextract` module and is not include by default in the plateform.

16.3.3.1. Implementation

The purpose of this plugin is to extract all these Ole objects and provide them as standalone files so that they can be checked individually. It is also extended to extract images. It has a classical Transform plugin structure, the plugin name is `oleExtractPlugin` bound to `org.nuxeo.ecm.platform.transform.plugin.oleextract.impl.OfficeOleExtractorPluginImpl`. The transform name is `oleExtract`.

As an example it can be called like

```
List <TransformDocument > results =
         service.transform(TRANSFORMER_NAME, null, new TransformDocumentImpl(stream, mimetype));
TransformDocument result = results.get(0);
List <Map<String, Serializable>> ole =
         (List < Map < String, Serializable > >) result.getPropertyValue("ole:olecontents");
```

First, the transform service is classically called, the `TRANSFORMER_NAME` being set to `oleExtract`. After processing, the only `TransformDocument` returned result contains a property `ole:olecontents` that gives the list of embedded objects that have been extracted.

The `olecontents` schema is defined in `olecontent.xsd`. Each element of the list contains the following fields

```
<xs:complexType name="olecontent">
  <xs:sequence>
    <xs:element name="displayname" type="xs:string" />
    <xs:element name="filename" type="xs:string" />
    <xs:element name="mime-type" type="xs:string" />
    <xs:element name="data" type="nxs:content"/>
    <xs:element name="thumbnail-mime-type" type="xs:string"/>
    <xs:element name="thumbnail-data" type="nxs:content"/>
  </xs:sequence>
</xs:complexType>
```

Each `olecontent` element contains file datas (`data` & `mime-type`) and thumbnail ones (`thumbnail-mime-type` & `thumbnail-data`). The `displayname` is the name retrieved in the office file if it was named otherwise the internal one that has been given in the office file. The `filename` fields is built from the `displayname` and the extension deduced from the `mime-type`.

Based on this new schema, the File doctype is extended in the org.nuxeo.ecm.platform.transform.oleextract.coretypes contributing type and schema needed by oleExtract

```
<extension target="org.nuxeo.ecm.core.schema.TypeService" point="doctype">
  <doctype name="FileWithOle" extends="File">
    <schema name="olecontents" />
  </doctype>
</extension>
```

A new `FileWithOle doctype`, based on `File`, is defined. It can be subtype of `Workspace` and `Folder` and is defined as a contribution of `org.nuxeo.ecm.platform.types.TypeService` which allows to create new documents based on it.

The oleExtract transform plugin has been bound to the `oleExtractModifier` docModifier. The contribution to the extension point is defined in

```
<extension target="org.nuxeo.ecm.platform.modifier.service.DocModifierService"
           point="docTypeToTransformer">
  <documentation>
    docModifier for oleExtract transform plugin.
  </documentation>
  <docModifier name="oleExtractModifier"
               documentType="FileWithOle"
               transformationPluginName="oleExtract"
               sourceFieldName="file:content"
               destinationFieldName="file:content">
    <coreEvent>documentCreated</coreEvent>
    <coreEvent>documentModified</coreEvent>
    <customField name="olecontents:olecontents" transformParamName="ole:olecontents"/>
    <customOutputField outputParamName="ole:olecontents" name="olecontents:olecontents"/>
  </docModifier>
</extension>
```

This contribution reacts on document creation and modification. It receives the initial office file `file:content` and gives back the `olecontents:olecontents` back mapped top the `ole:olecontents` we saw above. The initial file is returned unchanged.

With this docModifier reacting on some events, the oleExtract results can now be integrated in the application. The org.nuxeo.ecm.platform.transform.oleextract.action component defines an ActionService contribution

```
<action id="TAB_OLEOBJECT" link="/incl/tabs/document_oleobjects.xhtml"
        enabled="true" label="action.view.ole" order="49">
  <category>VIEW_ACTION_LIST</category>
  <filter id="view_ole">
    <rule grant="true">
      <type>FileWithOle</type>
    </rule>
  </filter>
</action>
```

The `TAB_OLEOBJECT` action defines a new tab listing the `olecontents:olecontents` elements and providing links for retrieving each file provided the document is of the current type `FileWithOle`.

16.3.3.2. Extraction details

OleExtract is based on the parsing of OpenDocument File format. If the submitted file is an ODF one, then it is unzipped and processed without any connection to OpenOffice.org. If the file is not an ODF one, then a converter plugin is used according to the source mime type. In this case, OpenOffice.org is required as a JODConverter dependency.

Once we have an ODF file it is unzipped and its content.xml is parsed to find `draw:object-ole`, `draw:object` and `draw:image` related tags. For each one, the name of the resource is retrieved if it exists. Then for each found resource, the alternate view is retrieved so that a preview can be proposed when listing this content (still under development)

ODF resources in an ODF file (think at a spreadsheet diagram embedded in a text document) are stored in directories and flat XML form while other resources are stored in a binary format. So for ODF resources the global manifest file is parsed to isolate the files with their correct manifest:media-type so that the new ODF archive for the Ole object can be built. Once the new manifest file is created, the embedded ODF directory is zipped and this binary form is returned.

16.3.4. Office files merger

This transform plugin is contained in `nuxeo-plateform-transform-plugin-officemerger` module and is not include by default in the platform. Its purpose is to build a new file as the result of the merging of the list given as parameters. It uses OpenOffice.org merging capabilities, mainly through the `insertDocumentFromURL` `UNO` method for text documents.

The public method `merge` is available from `OfficeMergerImpl` and returns a `SerializableInputStream` containing the resulting document.

```
OfficeMergerImpl merger = new OfficeMergerImpl();
SerializableInputStream result = merger.merge(sourceFiles, engineType, converter,
         outlineRank, withPageBreaks);
SerializableInputStream result = merger.mergeStreams (sourceStreams, engineType,
         converter, outlineRank, withPageBreaks);
```

Some options have been added to enhance the building of the main document. Here is the list of the arguments of the `merge` method

- `sourceFiles/sourceStreams`: Ordered array of File objects or streams to be merged.
- `engineType`: Depending on the nature of source documents, the OpenOffice.org API to be used is obviously not the same. If source files are text file, the user will probably want to have a text file as a result while if he deals with slides, the results is expected as a presentation. This String argument tells which engine to be used (`text`, `presentation`, `spreadsheet`- only `text` is already implemented) - Default `text`
- `converter`: Once the document is built, the Document Converter plugin can be called automatically to create the final document. This is the converter name that is expected (eg. `any2pdf`) and an exception is raised if it does not exist or the mime type deduced from the `engineType` is not supported. If an empty string is provided, no transform occur at the end of the merging - Default empty
- `outlineRank`: This is the rank (compared to the file list) where a Table Of Content may appear. For example, if the value is 3, then the first two files of the file list are inserted, the T.O.C is built and inserted and then the remaining files are processed. The Table of Content is refreshed at the end of the whole insertion. A value of 0 means no T.O.C. - Default `0`
- `withPageBreaks`: A boolean that adds or removes page breaks between file insertions - Default `true`

The plugin engine, the `merge` method is available directly but the principal use will occur through a `Transform` call. The Transform name is `officeMerger` and the package name is `org.nuxeo.ecm.platform.transform.plugin.officemerger`

```
<plugin name="OfficeMergerPlugin"
        class="org.nuxeo.ecm.platform.transform.plugin.officemerger.impl.OfficeMergerImpl"
        destinationMimeType="application/vnd.oasis.opendocument.text">
  <sourceMimeType>application/msword</sourceMimeType>
  <sourceMimeType>application/vnd.oasis.opendocument.text</sourceMimeType>
  <sourceMimeType>application/vnd.sun.xml.writer</sourceMimeType>
  <option name="ooo_host_name">localhost</option>
  <option name="ooo_host_port">8100</option>
</plugin>
```

Note that OpenOffice.org has to be listening from incoming `UNO` connections on the specified interface `ooo_host_name` and port `ooo_host_port`. If it is not the case, an `OpenOfficeException` exception will be raised.

So, once defined, the `officeMerger` transform can be called passing the options like

```
Map<String, Serializable> mergingOptions = new HashMap<String, Serializable>();
mergingOptions.put("engineType", "text");
mergingOptions.put("converter", "any2pdf");
mergingOptions.put("outlineRank", 0);
mergingOptions.put("withPageBreaks", false);

options.put("officeMerger", mergingOptions);

List<TransformDocument> results = transformer.transform(options, sourceFiles);
```

Note that `mergingOptions` can be incomplete or even `null`. The options will then take their default values.

The results list contains the final merged document, and converted if requested, at first index.

16.3.5. XSL Transformation

The XSLT plugin provides XSL transformations in Nuxeo. It allows you to transform XML documents using a XSL stylesheet as defined in the XSLT Specification. The plugin is implemented in the `org.nuxeo.ecm.platform.transform.plugin.xslt.impl.XSLTPluginImpl` class and defined as a `plugin` contribution to `org.nuxeo.ecm.platform.transform.service.TransformService`.

The XSLT Plugin accepts XML documents as source files, and you must provide the XSL stylesheet as a plugin's option named `stylesheet`. The XSL stylesheet must be provided as a `Blob`.

Then, you can easily transform your documents:

```
final Map<String, Serializable> pluginOptions =
        new HashMap<String, Serializable>();
pluginOptions.put("stylesheet", (FileBlob) getXSLStylesheetBlob(...);
final Map<String, Map<String, Serializable>> options =
        new HashMap<String, Map<String, Serializable>>();
options.put("xslt", pluginOptions);

TransformServiceCommon service = TransformServiceDelegate.getLocalTransformService();
final List<TransformDocument> results = service.transform(
        "xslt", options, xmlSourceFiles);
```

The resulting documents' mime-type is set depending of the `method` attribute of the `xsl:output` element, specified in the XSL stylesheet. If there is no `method` attribute defined in the XSL stylesheet, a default value is chosen for the `method` attribute as defined in the XSLT Specification.

The mime-type is chosen as follows:

- `text/html`: if the output method is `html`.
- `text/plain`: if the output method is `text`.
- `text/xml`: if the output method is `xml`.

If there is no XSL stylesheet provided to the plugin or if an error occurs during the transformation (corrupted xml or xsl for instance), a `TransformException` is thrown.

Chapter 17. Relations

17.1. Introduction

Relations in Nuxeo EP 5 follow concepts as described by the W3C Resource Description Framework (RDF).

The purpose is to provide content management relations (relations between documents of the site for instance) as well as being able to share this information with third party applications, by following the RDF standards.

17.2. Concepts

There are a few jargon terms to understand when dealing with relations.

Let's consider a relation like "document A is a version of document B". This relation is described as a *triplet* or *statement*: it has a subject, "document A", a predicate, "is version of", and an object, "document B".

The statement elements are more generally refered to as *nodes*. More specific kinds of nodes are *literals* and *resources*. A subject and a predicate will always be resources, while the object may be also a literal. In a relation like "document A has title 'documentation'", the object will be the literal string 'documentation'.

Literals are simple nodes, holding information like a string or a date. Resources refer to uniquely identifiable objects, and often use a URI as identifier that looks like a URL. If this URI refers to an identified namespace, we can make a difference between resources using it.

For instance, we can use the dcterms namespace to identify predicates: "http://purl.org/dc/terms/References", "http://purl.org/dc/terms/IsBasedOn","label.relation.predicate.IsBasedOn",...

Documents in the Nuxeo default application use the following namespace: "http://www.nuxeo.org/document/uid/". A document URI would look like "http://www.nuxeo.org/document/uid/618e53c8-409e-40e8-9b73-5493f7e6de88" because we use the JackRabbit identifier to identify fully the resource. Imagine that we use custom unique identifiers for documents, we could use them too, but we should use a different namespace so that we do not mistake the JCR identifier and the custom identifier.

When defining a relation like "document A is a version of document B", we will then build a statement which subject is a resource representing document A, which predicate is a resource representing the "is a version of" information, and which object is a resource representing document B.

If we would like to state that this relation was created as a certain date, we will add a date property to the statement. This can be seen as a relation where the subject would be the statement itself, the predicate a resource representing the "was created at" information, and which object would be a literal representing the given date.

17.3. Configuration

If you would only like to change the storage used for the default graph of Nuxeo EP 5, please refer to Section 32.2.4.2, “Relation service configuration”.

17.3.1. Graph instances

Relations are stored in a *graph*, that can also be called a *model*.

The graph definition is made though an extension point. It holds configuration about where and how to store relations. Here is an example contribution.

Example 17.1. Jena graph configuration for the Relation Service using PostgreSQL as storage

```
<?xml version="1.0" encoding="UTF-8"?>
<component name="MyJenaGraph">
  <require>org.nuxeo.ecm.platform.relations.jena</require>
  <extension target="org.nuxeo.ecm.platform.relations.services.RelationService"
    point="graphs">
    <graph name="default" type="jena">
      <option name="backend">sql</option>
      <option name="databaseType">PostgreSQL</option>
      <option name="datasource">java:/nxrelations-default-jena</option>
      <option name="databaseDoCompressUri">false</option>
      <option name="databaseTransactionEnabled">false</option>
      <namespaces>
        <namespace name="rdf">
          http://www.w3.org/1999/02/22-rdf-syntax-ns#
        </namespace>
        <namespace name="dcterms">http://purl.org/dc/terms/</namespace>
        <namespace name="nuxeo">http://www.nuxeo.org/document/uid/</namespace>
      </namespaces>
    </graph>
  </extension>
</component>
```

This graph uses a Jena graph. Jena is a RDF framework, a plugin has been developed to integrate it to the nuxeo platform. The graph definition requires the plugin to be registered to the application.

The graph is named "default" and declares its connection configuration.

17.3.2. Resource adapters

The graph configuration includes namespaces used for some of the graph resources so that resources with a known namespace can be transformed into any kind of object.

For instance, the namespace "http://www.nuxeo.org/document/uid/" is used to identify documents using their JCR unique identifier. We can register an adapter so that the resource can be transformed into the actual document model it represents.

17.4. Manage relations

17.5. Display relations

Chapter 18. Placeful Configuration

18.1. Introduction

The placeful configuration service allows configuration to be placed on a node in a repository. It is possible to update, remove this configuration and merged it with all the configuration located from this node to the root of the repository.

Placeful Configuration (PC in the rest of this chapter) is useful when you have a number of nodes much bigger than the number of configuration. It allows to change and merge these configuration without having to travel the repository tree.

18.2. Using Placeful Configuration

The PlacefulConfigurationManager is used for all interaction with PC. This service is available via the extension point org.nuxeo.ecm.platform.placeful.configuration.service.PlacefulConfigurationService. Before using it, you need to associate a configuration with a storage in the component definition ???.

You can then use the placeful configuration service. The following code snippet show the basic usage:

```
Path p1 = new Path("/mon/path");
RepositoryLocation repo = new RepositoryLocation("monrepo");
NuxeoPrincipal principal = new NuxeoPrincipalImpl("myself");

PlacefulConfigurationManager pcs = Framework.getService(PlacefulConfigurationManager.class);
LocalTheme lt1 = pcs.createConfigurationEntry(LocalTheme.class, p1, repo, principal);
lt1.setMode("myMode");
pcs.saveConfigurationEntry(lt1);

Map<String, Object> map = new HashMap<String, Object>();
map.put("mode", "myMode");
List<LocalTheme> list =  pcs.getAllConfigurations(LocalTheme.class, map);              ❶

LocalTheme lt = pcs.getConfiguration(LocalTheme.class, repo, p, principal);
LocalTheme ltMerge = pcs.getMergedConfiguration(LocalTheme.class, repo, p, principal);
pcs.removeConfiguration(LocalTheme.class, repo, p1, principal);
```

❶ Query the storage for all the PC that have those field/value.

Note that:

- you never create a PlacefulConfiguration yourself but ask the manager for one.
- You need to save the PC after modification.
- You can not move a PC, you need to remove it and create a new one.

For more information on available methods and class, have a look at the Javadoc.

18.3. Contributing a placeful configuration

A PC is composed of two distinct types of information:

- The information specific to this configuration, for example, a local theme configuration has a mode, a page or a docId.
- The information relative to its "placefulness": the path, principal and repository.

To contribute a configuration to the service you only have to gives the information specific to the configuration. The "placeful" part is taken care of by the service. You also need to give a way to merge the PC. We will create a simple config as an exemple. A "Simple" PC that has only one field: value.

- Create an interface specific for your configuration.

```
interface SimpleConfig{
    setValue();
    getValue();
}
```

- Create the empty interface that will be manipulated by the user. It needs to extends PlacefulConfiguration, Serializable and your specific interface.

```
public interface Simple extends PlacefulConfiguration, SimpleConfig, Serializable {}
```

- Create the implementation of the specific configuration. It needs to implement your specific interface and PlacefulConfigurationConfig. The PlacefulConfigurationConfig interface add the getAssociatedInterface() methods. It returns the "user" interface:

```
SimpleConfigImpl implements SimpleConfig, PlacefulConfigurationConfig {
    private String value;
    public String getValue() {
         return value;
    }
    public void setValue(String value) {
        this.value = value;
    }
    public Class<Simple> getAssociatedInterface() {
        return Simple.class;
    }
}
```

- Create the class for the merge algorithm. It has to implement PlacefulConfigurationAlgoritm. The only function takes a PC and a storage and return a merged PC. Our Simple configuration will do no merge at all:

```
public class SimpleMergeAlgorithm implements PlacefulConfigurationMergeAlgorithm {
    Simple mergeEntries(Simple pc, PlacefulConfigurationStorage storage) {
        return pc;
    }
}
```

- Finally the "user" interface has to know the implementing and merge class. You add the @configurationClass annotation on the "user" interface:

```
@ConfigurationClass(value=SimpleConfigImpl.class, mergeAlgorithm=SimpleMergeAlgorithm.class)
public interface Simple extends PlacefulConfiguration, SimpleConfig, Serializable {};
```

- If you want your PC to be usable by a Directory storage you also need to provide a schema Section 18.4.2, “Directory storage”

In most case you want your merge algorithm to merge all entries from the root to the node of the configuration. For this, you can extend the class AbstractBaseMergeAlgorithm. You then only need to add a method to merge 2 entries.

If your PC is a simple java bean then you can extend BeanMergeAlgorithm . The only things you need to pass is the class of your bean. For each property, the merge algorithm will return the nearest value from the node that is not null.

18.4. Available storage

A storage specifies where the configuration is stored. You can pass specific values to each storage using the properties map (see Section 18.5, “Exemple of extension definition”). The storage gives you also the possibility to define fields that will be indexed. When a user query the storage, only the indexed fields can be queried. At the moment, only String field can be queried and indexed.

It is assumed that an indexed field is a property of the PC and so available via getters. By default, all the "placeful" values (principal, path, repository) are indexed and don't need to be added to the fields list.

18.4.1. In memory storage

The InMemory storage does not use any persistence. You should only use in it in very simple situation such as test. The storageBackend class is: org.nuxeo.ecm.platform.placeful.configuration.storage.InMemoryPlacefulConfigurationStorage. You can pass comma separated values in the fields property. Each value represent a property of the PC. In the backend an index will be created for each property. You can then use it as a search field in the map passed to manager to find configurations (Section 18.2, “Using Placeful Configuration”).

18.4.2. Directory storage

org.nuxeo.ecm.platform.placeful.configuration.storage.SQLDirectoryPlacefulConfigurationStorage implements the storageBackend. You pass it as the class attribute. You also need to add the name of you directory:

```
<storageBackend name="SQL"
    class=
"org.nuxeo.ecm.platform.placeful.configuration.storage.SQLDirectoryPlacefulConfigurationStorage">
  <properties>
    <property name="directoryName">localTheme</property>
  </properties>
</storageBackend>
```

To use directory storage you need to define a directory and the associated schema. The schema only needs to include the base.xsd schemaLocation and the placefulConfiguration.xsd schemaLocation. You can also add any String element that you want to be indexed. Note that the "placeful" fields general for all configurations are included by default and don't need to be added.

Don't forget to declare the schema and directory to the extension point. It is assumed that each field to be indexed is a property of the PC and can be accessed by getters.

18.5. Exemple of extension definition

```
<component
    name="org.nuxeo.ecm.platform.placeful.configuration.defaultContrib">

  <extension
      target=
"org.nuxeo.ecm.platform.placeful.configuration.service.PlacefulConfigurationService"
      point="storage">                                                              ❶❷

    <storageBackend name="RAM"
        class=
"org.nuxeo.ecm.platform.placeful.configuration.storage.InMemoryPlacefulConfigurationStorage">
      <properties>
        <property name="fields">docId,theme,mode</property>                        ❸
      </properties>
    </storageBackend>

  </extension>

  <extension
      target=
"org.nuxeo.ecm.platform.placeful.configuration.service.PlacefulConfigurationService"
      point="configuration">                                                        ❷

    <configuration name="TestConfig" storage="RAM"
        class="org.nuxeo.ecm.platform.placeful.configuration.entry.LocalTheme" />
  </extension>

</component>
```

❶ Definition of the storage.
❷ The configuration reference the storage name using the storage attribute.
❸ Storage specific values.

Chapter 19. Virtual Navigation

19.1. Virtual Navigation

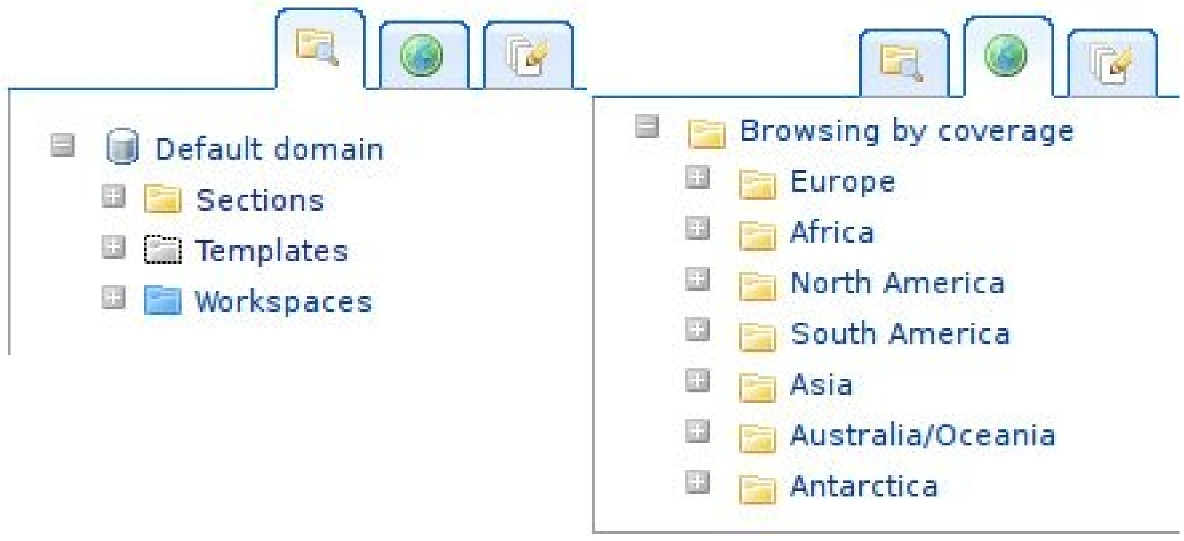
This section is speaking about a Nuxeo addon that provides a new way to navigate in documents.

19.1.1. Virtual Navigation presentation

The Virtual Navigation is oposed to the physical navigation. The physical navigation is the intuitive way to browse your documents as you created them, folders contain documents. Virtual Navigation is based on meta-data linked with every documents. In Nuxeo 5 each document has a set of meta-data that make him more precise and rich. The Virtual Navigation can provide a navigation tree built on meta-data and more precisely on every vocabulary based data.

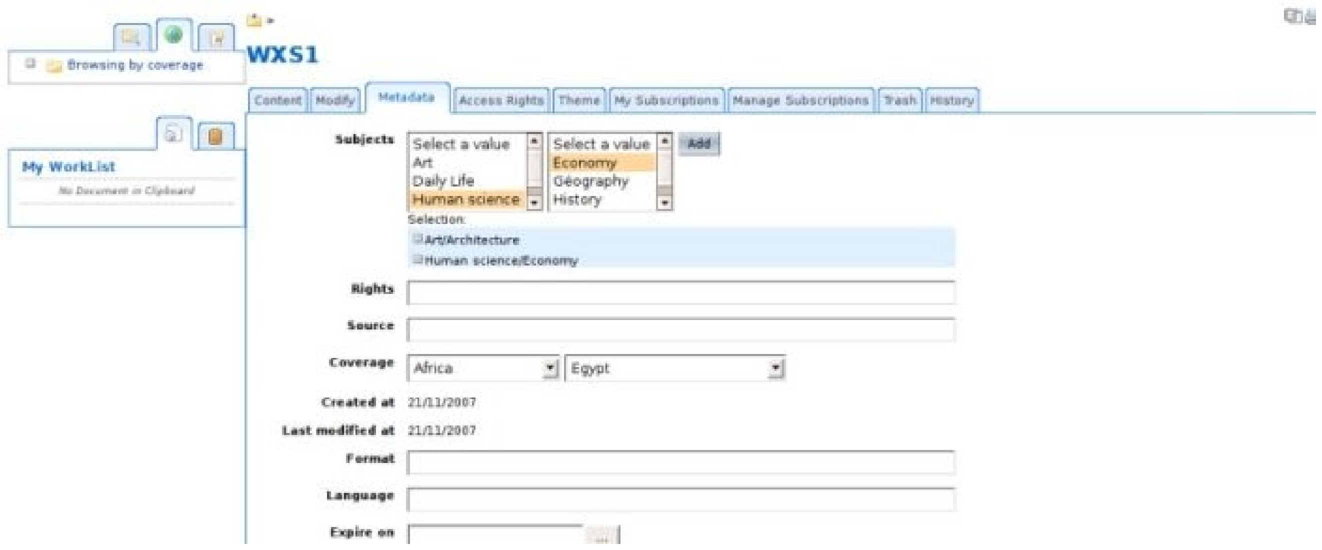
Now the basic Virtual Navigation configuration offers a navigation through coverage and through subjects. For each document it is possible to determine a contry that is relative to the document and one or many subject that well corespond to it.

To activate the Virtual Navigation you have to add the Virtual Navigation addon in the plugin extension folder of your JBoss server. After restarting the server you can see a new wiget in the left hand corner of Nuxeo 5. You can switch between navigation style easily.



The standar physical navigation is still here and the two Virtual Navigation are selectable on the right. As you can see the Virtual Navigation keeps a folder based navigation but this tree browsing is built from vocabularies defined in your Nuxeo 5 instance. Coverage and Subjects are 2 vocabularies that exist in the basic Nuxeo 5 configuration. You may know that tree based vocabularies are not limited in depth and you can have 2 or more level in your vocabulary, for example Coverage allows to select a **continent** and a **country**.

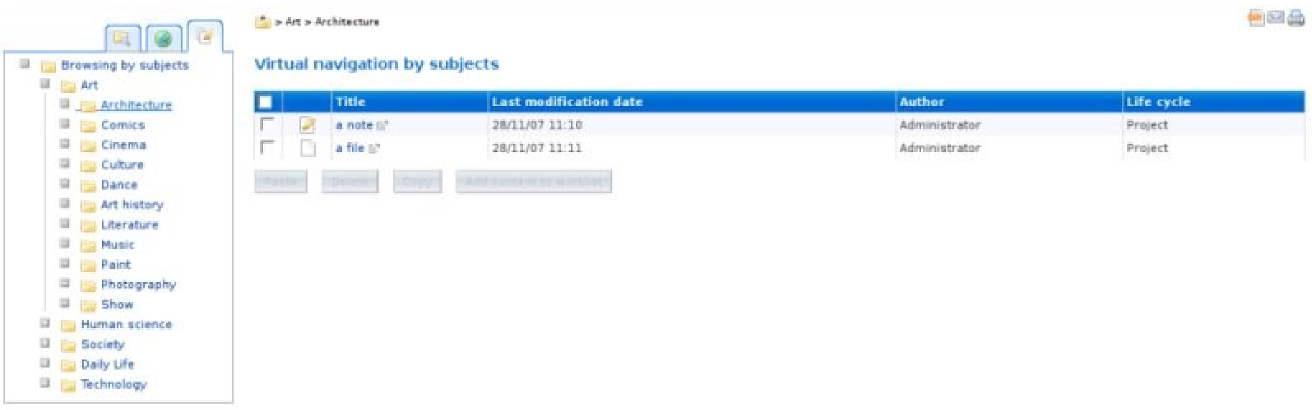
To have good results with Virtual Navigation you have to fill meta-data linked to documents. The Meta-Data view allows the user to provide his own meta-data. Select a country and/or subjects to make your Virtual Navigation efficient. See below to view the Meta-Data configuration screen.



WXS1

Subjects	Select a value / Art / Daily Life / Human science — Select a value / Economy / Geography / History
	Selection: Art/Architecture; Human science/Economy
Rights	
Source	
Coverage	Africa — Egypt
Created at	21/11/2007
Last modified at	21/11/2007
Format	
Language	
Expire on	

When selecting a node in a Virtual Navigation tree it will execute a request to find every documents that contain the wanted meta-data. See the screen below for an example, if you select the Art/Archiecture couple in the Subjects navigation tree every documents that contain Art/Architecture in their Subjects meta-data will be displayed on screen.



Virtual navigation by subjects

Title	Last modification date	Author	Life cycle
a note	28/11/07 11:10	Administrator	Project
a file	28/11/07 11:11	Administrator	Project

19.1.2. Virtual Navigation configuration

The basic Virtual Navigation configuration, as seen above, offers 2 way to navigate through meta-data in addition to the classic phisical navigation, by coverage and by subjects. Here is a complet walkthrough to add a new virtual navigation to a customized project.

19.1.2.1. Required precondition

You have to create a vocabulary that will be used for the new Virtual Navigation, keep in mind that you can set up a multi-level vocabulary, so you can imagine a vocabulary with parent and children. Ther is no limitation on this side.

You have to select the element in a schema that will store your vocabulary data. In the basic Nuxeo 5 configuration it is dublincore:coverage and dublincore:subjects that store vocabularies data for Virtual Navigation. If fact, when you select a country (coverage) in a document's meta-data, the country is stored in the dublincore:coverage element. It's not mandatory to select a dublincore element, you can select an element coming from your own schemas.

19.1.2.2. Set up your vocabulary

If you have a "only-one-level" vocabulary you can create a file to store it like this :

Example 19.1. Example to create your own vocabulary

```
id, label, obsolete
"art","label.directories.topic.art","0"
"human sciences","label.directories.topic.humanscience","0"
"society","label.directories.topic.society","0"
"daily life","label.directories.topic.dailylife","0"
"technology","label.directories.topic.technology","0"
```

If you have a "multi-level" vocabulary you have to split him in sub vocabulary, and keep in mind that you have to indicate which vocabulary element is the child of wich parent vocabulary element, here is an example, the full example can be found in Nuxeo 5 sources, check **topic.csv and subtopic.csv** :

Example 19.2. Example to create your own multi-level vocabulary

```
CONTENT OF THE FIRST FILE (parents)

id, label, obsolete
"art","label.directories.topic.art","0"
"human sciences","label.directories.topic.humanscience","0"
"society","label.directories.topic.society","0"
"daily life","label.directories.topic.dailylife","0"
"technology","label.directories.topic.technology","0"


CONTENT OF THE SECOND FILE (with a "parent" parameter added)

id, label, parent, obsolete
"architecture","label.directories.subtopic.architecture","art","0"
"comics","label.directories.subtopic.comics","art","0"
"rights","label.directories.subtopic.rights","human sciences","0"
"economy","label.directories.subtopic.economy","human sciences","0"
...
```

When you have created your different files for vocabularies you have to register them in an extension point as new vocabularies. In an xml file that you won't forget to place in a OSGI-INF directory and won't forget to register in the MANIFEST.MF file of the package, you have to contribute the following extension point.

For a full example you can check the **nxdirectories-contrib.xml** file of the **webapp-core** package. Don't forget to use the **xvocabulary** schema if your vocabulary is a child of another, if the vocabulary is the first parent or is alone just use the **vocabulary** schema. Don't forget to indicate the parent if your vocabulary has more than one level with the following tag **<parentDirectory></parentDirectory>.**

Example 19.3. How to register new vocabularies

```
<extension target="org.nuxeo.ecm.directory.sql.SQLDirectoryFactory"
  point="directories">

//here your new vocabularies contribution

</extension>
```

19.1.2.3. Set up a new Document Type for search purpose

The query based search service of Nuxeo 5 requires that you create a document type that will be a base for a document model to register data handeld by the query. To understand it more here is an example :

You are browsing your documents by coverage. You are selecting the path Europe/France in the tree. The data Europe/France is the base of the query and need to be registered in a document model created from a document type. In this case the document type can be very simple, cause the query must register only one data at a time. Creating a document type with only one schema that contains one field will be enough.

This document type will be referenced as **query document type** in this walkthrough. Create it and register it as a normal document type. See Nuxeo Book part 6 to get more informations about document type creation.

19.1.2.4. Set up new navigation tree

Now you must contribute to another extension point to create a new navigation tree based on vocabularies you contributed just before. See below for an example of the file you have to create. For a full example you can see the **directorytreemanager-contrib.xml** file in the **virtualNavigation** package.

The new tree contribution needs many informations, **a queryModel** to indicate the query you will use to get your documents, a **schema** and a **field** coming from the query document type you just set up in part 3, an **outcome** that indicates the page where documents will be displayed after the request and a list of vocabularies you can indicate with the tag **<directory></directory>.**

Example 19.4. How to register new navigation trees

```
<require>
  org.nuxeo.ecm.webapp.directory.DirectoryTreeService
</require>

<extension
  target="org.nuxeo.ecm.webapp.directory.DirectoryTreeService"
  point="trees">

//here your new navigation tree contribution

</extension>
```

19.1.2.5. Set up the queryModel and the Result Provider

Now you need to contribute to 2 extension points that will set up the query and the results provider. Here are the 2 extension point you will need to contribute. For a full example see the file **querymodel-contrib.xml** and **resultsprovider-contrib.xml** in the **virtualNavigation** package.

The query model contribution needs many information. In the **docType** parameter you have to put the name of the documentType you created in part 3. In the <predicate></predicate> tag you must set up the name of the element (field) that is the field storing the data used for virtual tree construction.

EXAMPLE :

You want to browse documents by coverage, each of your document have a coverage registered in dc:coverage field. You have to use the **dc:coverage** parameter.

If there is no prefix set the name like this **schema:field**, if there is a prefix set it up like this **prefix:field**. In the operator param put the value STARTSWITH. In the <field/> tag put the schema and the name of the field coming from the **query document type** you set up in part 3.

Example 19.5. How to register query model and results provider

```
<extension
   target="org.nuxeo.ecm.core.search.api.client.querymodel.QueryModelService"
   point="model">

//Here your new query model

</extension>


<extension
    target="org.nuxeo.ecm.webapp.pagination.ResultsProviderService"
    point="model">

//Here your new result provider

</extension>
```

19.1.2.6. Set up the search service

Now you need to register a new indexing configuration in the search service. You will index the field that register the data you want to apply the search on, it is the same data as saw at the end of part 5 . Here is the extension point you have to contribute. For a full example see the **nxsearch-contrib.xml** file in the **search-core** package.

You will have to create a new <resource></resource> tag if it doesn't exist for your schema. For the **name** parameter you have to put the name of your schema, **type** param must be "schema" and **indexAllFields** param must be "true". In a <field/> tag you have to indicate the indexing strategy, the **type** param must be "**Path**", if the field that register your vocabulary is a complexType (list of String for example) you can add the parameter **multiple** and set it at "true".

Example 19.6. How to register the search configuration

```
<extension
    target="org.nuxeo.ecm.core.search.service.SearchServiceImpl"
    point="resource">

//Here your new search configuration

</extension>
```

19.1.2.7. Other details

You must not forget to add some navigation cases in a deployment-frgament.xml file. Those navigation cases must correspond at each **outcome** you set up in part 3. Each page named in navigation cases shall display search result so you have to customize those pages with the good queryModel call. For a full example see **coverage_virtual_navigation.xml** .

All theme contributions and tree navigation widget already exist in **webapp-core** and **virtualNavigation** package, you can take a look for information purpose.

Indeed you have to add vitualNavigation to Nuxeo 5 plugin if you want a fully fonctional Virtual Navigation immediatly.

Chapter 20. Metadata Extraction Service

20.1. Introduction

There are cases when some informations could be retrieved from attached files and keep this info as regular Document properties. This info is refered to as metadata as it is a descriptive info refering the document (file) from which it is extracted. For example given a MS Word document (file) we could retrieve info (metadata) like *author, title, creation date* etc, simply by reading document headers with an appropriate library (that knows how to parse and keep a MS Word document internal structure).

20.2. Metadata extraction module

The metadata extraction feature is composed of several projects:

- `nuxeo-platform-metadataext-api`

- `nuxeo-platform-metadataext-core`

- `nuxeo-platform-metadataext-facade`

- `nuxeo-platform-metadataext-plugins`

The core part defines the metadata extraction component with a `MetaDataExtractionManager` implementation: `org.nuxeo.ecm.platform.metadataext.services.MetaDataExtractionService`.

The service is normally invoked by a dedicated `CoreListener` that passes a `DocumentModel` for which an extraction could be defined. Also the metadata could be extracted by direct invocation from a client code through existing EJB3 facade.

The plugin part provides a plugin (more to come) which is specific to the MS-Word document type. The plugin is a Transformation Plugin (as defined by the Transformation service) and gets invoked as part of a defined transformation chain. The apropriate transformation will be called by the metadata extraction service if there is a contribution for the given Document type, etc.

20.2.1. Defining a contribution for metadata extraction

A contribution can be defined by adding an XML file with the following structure:

```
<extension
    target="org.nuxeo.ecm.platform.metadataext.services.MetaDataExtraction"
    point="extractions">

  <meta-data-extraction inputField="file:content"
      transformationName="MSWordMDExt">

    <outputParams>
      <param propertyName="dc:title">title</param>
      <param propertyName="dc:description">comments</param>
      <param propertyName="dc:created">creationDate</param>
      <param propertyName="dc:modified">creationDate</param>
      <param propertyName="dc:contributors">authors</param>
    </outputParams>

    <coreEvent>documentCreated</coreEvent>
    <coreEvent>documentModified</coreEvent>

    <docType>File</docType>

  </meta-data-extraction>

</extension>
```

The important aspects for MetaDataExtraction service are:

- specification of a source (blob) from where metadata will be extracted. This is defined by the value *`inputField`* which should be a blob document property.

- the mapping of output parameters. This defines a corespondence between the map entries returned as the result of transformation (extraction) and the Document properties names to which the results will be written back.

- the list of core events for which the extraction will be performed.

- the list of document types for which the extraction can be applied.

20.2.2. Specifying input parameters

If the extraction is requiring additional information aside from the provided blob, we can add input parameters to an extraction definition like this:

```
<inputParams>
  <param propertyName="dc:title">title</param>
</inputParams>
```

A map with input parameters will be passed to the extraction plugin having the key the value of tag `param` (in this case 'title').

20.2.3. Chaining extractions

In extreme cases the extractions of metadata could be performed in several steps. That is using the extracted parameters in phase as input parameters for the next phase. This could be achieved by specifying the input params whose values could have been set by a previous extraction (transformation).

20.2.4. Creating a plugin for metadata extraction

We first define a class implementing `org.nuxeo.ecm.platform.transform.interfaces.Plugin` interface like:

```
public class MSWordMDExtractorPlugin extends AbstractPlugin {

    @Override
    public List<TransformDocument> transform(Map<String, Serializable> options,
            TransformDocument... sources) throws Exception {
      ...
    }
...

}
```

Overriding the `transform` method is a must and in this case we will have only one `TransformDocument` as source.

The blob from which metadata could be extracted can be retrieved from `TransformDocument` like: `Blob srcBlob = sources[0].getBlob().`

After the useful information is extracted the properties should be set to a `TransformDocument` that will be returned:

```
TransformDocumentImpl res = new TransformDocumentImpl();
res.setPropertyValue("title", extractedTitle);
```

20.2.5. Using a metadata extraction plugin

Having a plugin class defined that knows how to extract information from a specific type of file (MS Word document, MS Excel, PDF, OO doc etc) we can add the contributions necessary for the metadata extraction to take place.

We need to define first the transformation plugin and the transformation contributions. After that the defined transformation can be refered in the metadata extraction specific contribution.

Step 1: define the transformation plugin:

```
<extension
    target="org.nuxeo.ecm.platform.transform.service.TransformService"
    point="plugins">

  <documentation>
    Set of default transformation plugins for metadata extraction.
  </documentation>

  <plugin name="MSWordMDExtPlugin"
      class="org.nuxeo.ecm.platform.metadataext.plugins.MSWordMDExtractorPlugin"
      destinationMimeType="application/msword">
    <sourceMimeType>application/msword</sourceMimeType>
  </plugin>
</extension>
```

Step 2: define the transformation:

```
<extension
    target="org.nuxeo.ecm.platform.transform.service.TransformService"
    point="transformers">

  <documentation>
    Set of default transformation chains for metadata extraction.
  </documentation>

  <transformer name="MSWordMDExt"
      class="org.nuxeo.ecm.platform.transform.transformer.TransformerImpl">
    <plugins>
      <plugin name="MSWordMDExtPlugin" />
    </plugins>
  </transformer>
</extension>
```

Step 3: define the metadata extraction specific contribution

```
<extension
    target="org.nuxeo.ecm.platform.metadataext.services.MetaDataExtraction"
    point="extractions">

  <meta-data-extraction inputField="file:content"
      transformationName="MSWordMDExt">

...
```

This last one is refering to the above defined transformation.

Chapter 21. Unicity Service

The Unicity service sends a new Message on the JMS bus every time you upload a file already on the server.

21.1. How does it works ?

When creating or updating a file, a digest is computed using the whole file and an encoding algorithm. Next step is an nxql query wich gets every Document with the same Digest. If such documents exist, there references will be forwarded on JMS bus.

21.2. What you need:

The message's eventID is *duplicatedFile*. You need it to catch the event. The documentLocation list is available in the info map of the CoreEvent using *duplicatedDocLocation* key.

21.3. Configuration

How to configure unicity extension point. TODO: needs to be made clearer.

```
<enabled>
Default value is false. Use true if you want to enable this service.
</enabled>
<algo>
Default encoding algorithm is sha-256. You can choose every algorithm supported by java.security.MessageDigest.
</algo>
<field>
A field is an xpath expression giving a particular field of your schema.
It's the reference of a file on the server.
The type's field must be nxs:content.
You can use as many field as you want.
</field>
```

21.4. Snippets

Some code to get you started.

Example 21.1. Sample unicity extension point contribution to the FileManager service.

```
<?xml version="1.0"?>
<component name="org.nuxeo.ecm.platform.filemanager.service.FileManagerService.Plugins">
  <extension
      target="org.nuxeo.ecm.platform.filemanager.service.FileManagerService"
      point="unicity">
    <unicitySettings>
      <enabled>true</enabled>
      <algo>sha-256</algo>
      <field>content</field>
    </unicitySettings>
  </extension>
</component>
```

Example 21.2. Sample code to retrieve DocumentLocations.

```
public void onMessage(Message message) {
    try {
        Serializable obj = ((ObjectMessage) message).getObject();
        if (!(obj instanceof DocumentMessage)) {
            return;
        }
        DocumentMessage doc = (DocumentMessage) obj;

        String eventId = doc.getEventId();

        if ("duplicatedFile".equals(eventId)){
            Object[] documentLocations = (Object[]) doc.getEventInfo().get("duplicatedDocLocation");
            for (Object documentLocation: documentLocations) {
                log.debug(((DocumentLocation)documentLocation).getDocRef());
            }
        }
    }
}
```

Part III. Core Services

Chapter 22. Nuxeo Runtime

22.1. Overview

Nuxeo Runtime is the foundation of the Nuxeo infrastructure. It handles deployment and extensibility of components to target platforms. This component allows the whole Nuxeo infrastructure to be easily ported between Java platforms (Java EE, OSGi, etc.) and features an easy plug-in mechanism that any component can use to declare extension points that can be used by other components to extend the former one.

Nuxeo Runtime uses the OSGi component model and a set of adapters to deploy POJO components to Java host platforms, such as Eclipse/Equinox, or a Java EE 5 application server such as JBoss or WebLogic. When deployed, Nuxeo Runtime components become actual host platform components. For example on JBoss the component is seen as a MBean, while when deployed on Geronimo it is seen as a GBean and on Eclipse it is seen as a native Eclipse plug-in. In short, Nuxeo Runtime offers a new and seamless way to make your Java EE applications and components extensible (as Eclipse developers are already used to).

Nuxeo Runtime is not specific to the Nuxeo platform, it is a generic deployment and extension system that can be used in any Java or Java EE application.

Forget specific build of your applications for a dedicated project or customer and enjoy "Code once, deploy anywhere" for real!

22.1.1. Main Goals

One of the main requirements of the "Nuxeo Core" component is to be deployable on both the JBoss and Eclipse platforms. To ease development and allow as much as code reuse, this requirement raised the need of a common component and packaging model that may be deployed and used on both of these platforms without any code change or repackaging.

To fulfill these needs, Nuxeo Runtime was developed as the foundation layer for all Nuxeo components. Nuxeo Runtime is not a standalone framework. It is, basically, a component model running on top of an existing platform and providing a common, platform-independent, model to power the applicative components of an application. It is our component architecture allowing flexible and true componentization of applications.

Besides its component model, Nuxeo Runtime also defines a common model for packaging. The adopted model is the OSGi bundle model. Actually, OSGi bundles are regular JARs containing an OSGi manifest file.

OSGi technology is more and more popular and is currently used by Eclipse, Geronimo and Jonas as their runtime framework.

This way, applications based on Nuxeo Runtime can run on different platforms without modifications by using an single component and packaging model - without having to care about platform specificity.

22.1.2. Main Features

The main features provided by Nuxeo Runtime are:

1. OSGi native support
2. Extensible component model through extension points
3. Adapters to support host platforms (JBoss and Eclipse support is built-in by default)

22.2. What is OSGi?

OSGi (Open Services Gateway initiative) is an open standards organization founded by Sun Microsystems, IBM, Ericsson and others in March 1999.

OSGi defines a modular and complete Java-based service framework. The deployment units used by this framework are called bundles, so we will refer them as OSGi bundles.

OSGi bundles are normal Java libraries (JAR files) containing a special manifest file (`META-INF/MANIFEST.MF`) describing all aspects related to the bundle like: the bundle name, description, bundle dependencies, exported packages, the bundle classpath, the bundle activator and many other OSG-defined features.

Here is a typical OSGi manifest file:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: NxRuntimeEclipseDemo Plug-in
Bundle-SymbolicName: org.nuxeo.runtime.demo.eclipse.Demo; singleton:=true
Bundle-Version: 1.0.0
Bundle-Activator: org.nuxeo.runtime.demo.eclipse.demo.Activator
Bundle-Vendor: Nuxeo
Bundle-Localization: plugin
Require-Bundle: org.eclipse.ui,
  org.eclipse.core.runtime,
  org.nuxeo.runtime.demo.HelloWorld,
  org.nuxeo.runtime
```

The bundle activator is a Java object that is called when the bundle is started and stopped by the framework. This is the only way available to the application to access the framework functionalities.

Besides bundles and bundle management, OSGi provide a service registry and an API to register the services provided by a bundle and to lookup these services. Also, OSGi defines a Declarative Services specification that significantly simplifies the service-oriented programming model. Through this model services can be defined in XML files inside the bundle and automatically deployed by the framework.

For a complete definition of OSGi, see Wikipedia.

One of the main goal of Nuxeo Runtime is to natively support OSGi frameworks and to use the OSGi bundle model for packaging and deployment. A second goal is to align the Nuxeo Runtime component model with the OSGi Declarative Specifications.

22.3. OSGi Support

Nuxeo Runtime provides built-in integration over OSGi-compliant frameworks. This means Nuxeo Runtime-based components can run “as is” on any OSGi enabled platform.

On other platforms like JBoss, an adapter is required. Nuxeo Runtime eases the creation of such adapters by providing an abstract OSGi adapter that can be customized for any platform.

Note: It doesn't mean you can transform any platform into a fully OSGi-compliant platform using Nuxeo Runtime adapters. This is mainly due to the fact that the adapter is using in the background the host platform's class-loading and deployment model that is incompatible with OSGi specifications.

Adapters only mimic an OSGi environment, using native host platform features, for applications running on top of Nuxeo Runtime.

Many OSGi features are not yet provided by the adapter – but we hope to add more and more features. If you are interested in helping on this, do not hesitate :-).

Currently, one of the most important feature that is missing is OSGi service support, but we are working on this and hope to provide it soon.

When running on OSGi-enabled platforms, no adapter is used and thus all OSGi features are available as given by the host platform. Components are running natively on the OSGi platform without any alteration.

Currently we provide two built-in adapters:

1. JBoss OSGi adapter – used to deploy OSGi bundles on JBoss AS 4.x

2. Test OSGi adapter – used for JUnit testing and can be used on any simple Java application that is not using a complex class loading or deployment mechanism.

22.3.1. Supported Features

Currently, Nuxeo Runtime adapters can provide the following OSGi features:

1. OSGi Bundle deployment
2. The manifest is loaded, the class path processed and the bundle activator instantiated
3. BundleActivator support
4. Activators are notified each time a bundle is started and stopped
5. Fake Bundle and BundleContext implementations that adapts OSGi operations to native operations of the host platform
6. Thus, Bundle Activators can use the common operations defined by the OSGi API.
7. Bundle lifecycle and framework events support
8. Bundle dependencies (as specified in the manifest)

22.3.2. Unsupported Features

The following OSGi features are not supported (yet):

1. The OSGi service layer
2. The OSGi Security layer
3. The OSGi class-loading specifications (the class-loading mechanism of the host platform is used)
4. Some methods of interfaces Bundle and BundleContext (unimplemented Methods will thrown an `UnsupportedOperationException` exception)

22.3.3. Planned Features

1. Supporting the OSGi Service layer
2. Implementing the OSGi Declarative Services based on the runtime component model

22.4. Component Model

The component model provides a flexible way to define, register and locate components. It was designed in order to reuse the same component model on very different platforms like JBoss and Eclipse.

A full support of OSGi declarative service specifications is planned in the mid-term future. Moreover, Nuxeo components can describe any kind of components, not only services.

22.4.1. What are components?

A definition from Wikipedia: A software component is a system element offering a predefined service and able to communicate with other components.

Components as defined by the Nuxeo model are logical units that may depend on and/or extend one another.

The Nuxeo Runtime is responsible to provide a common API to register, locate or extend components. Components are commonly registered using XML descriptor files.

Components can either be declared as subunits of an OSGi bundle or as independent components - as standalone XML files or programmatically registered components.

Components are commonly declared as part of an OSGi bundle through XML descriptors. To declare a component you need to create an XML description file, put it somewhere in the bundle and specify the “Nuxeo-Component” header in the bundle manifest to load components at bundle activation.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: HelloWorldExtension Plug-in
Bundle-SymbolicName: org.nuxeo.runtime.demo.HelloWorldExtension
Bundle-Version: 1.0.0
Bundle-Vendor: Nuxeo
Bundle-Localization: plugin
Require-Bundle: org.nuxeo.runtime.demo.HelloWorld
Nuxeo-Component: OSGI-INF/helloworld-extension.xml
```

22.4.2. Main Features

1. Declarative components through XML descriptors: XML component descriptors are tightly integrated with OSGi – you can specify which components should be deployed at bundle installation phase by using a custom manifest header “Nuxeo-Component”.

2. Dependency between components: Components are installed only when all their prerequisites are met. If prerequisites are not met the components will be put in a pending state until their dependencies are completely resolved. In the same manner when uninstalling a component all the components depending on it will be moved to the pending state.

3. Extensibility through extension points: Each component can let other components extend itself by defining a set of extension points. Other components (or the component itself) may then plug extensions into one of the declared extension points. This flexible extension mechanism draw inspiration from the Eclipse extension points.

4. Life Cycle Events: Component life cycle events are fired by the runtime to anyone interested in. See Adaptable Components for a common use case.

5. OSGi integration: The component model is about to be fully integrated with OSGi and will be soon compliant with the OSGi declarative service model.

6. Platform Independence: The component model can be used on any platform. It provides a single API to register and localize components – the Nuxeo Runtime native API may be used (and in future the OSGi service API will be available too).

22.4.3. Planned Features

1. Complete integration with OSGi declarative services specifications

2. Component lookup through JNDI

22.4.4. Adaptable Components

The runtime implementation may adapt registered components to native components of the host platform. This can be done using the component life cycle notifications.

The JBoss runtime implementation is already doing this to adapt runtime components into JBoss MBean services.

This way components may be seamlessly integrated into the host platform, thus leveraging the host platform functionalities (for example, MBean service management on JBoss)

22.4.5. Flexible Model

You should not be afraid by the “component model” denomination. The runtime component model is not limiting in any way your objects nor imposing extra rules in the development. You don't need to modify your existing objects to derive or implement some runtime classes in order to plug them into the component registry. Objects implementing a component may be of any kind.

The only limitation is that component objects must have a public constructor without arguments (the default constructor) so that they can be instantiated via `newInstance` method on Class.

Anyway, if you want to benefit from extension points mechanism or to respond to component life cycle events like activation or deactivation, then you should either implement the Component interface, or define some methods with a given signature in your object so that they will be called by using Java reflection. See User Guide for more details on this.

In conclusion the component model is not limiting your objects in any way, it only gives you the capability to register your components, extend and locate them in the same way on any platform supported by Nuxeo Runtime.

22.4.6. Component Life Cycle

A component has 3 main life cycle states:

1. `Registered`: the component registration information was created and inserted into the registry. The component dependencies are not yet processed or resolved, so the component cannot be activated and would block other registrations depending on this component.

2. `ResolvedAll`: dependencies of this component are satisfied. The component can be safely activated.

   Other unresolved components waiting for a resolved component are notified and if they have no more dependencies they will be resolved too.

3. `ActivatedComponent`: activation may occur immediately a component is resolved, or programmatically at the user request, or lazily the first time the component is referred. The only requirement for a component to be able to activate is to be resolved.

   Currently only the immediate activation mode is supported.

   When an activated component is deactivated it is put back in the resolved state. If it is unregistered it will be put in the resolved state then an unresolved event is fired and the component regresses to the registered sate and then it is removed from the registry.

   When a component is respectively activated or deactivated the runtime will invoke the activate, respectively the deactivate method of the component, if any. Implementing life cycle methods or not is the programmer's choice. These methods can be used to initialize and destroy the component in the given context. Components are not forced to implement any one of these methods.

   The activation of a component signifies the component is available to be used by other components so that the component should be correctly initialized when it enters this state. Here is the complete life cycle of a component:

   a. XXX ADD GRAPHIC HERE

22.4.7. Component Extensibility

One of the most important feature of the component model is the extension mechanism that enable components to extend one another.

How does it work? Imagine you have a component A that manages an action menu for the application. It wants to let other components contribute actions in an easy and flexible way – for example by using XML files to describe these actions.

To be able to do this, component A should declare an extension point, let's say “actions”. This way other components willing to contribute some actions to the action menu managed by the component A can contribute these actions to the extension point “actions” exposed by the component A.

Obviously one component may declare any number of extension points and any number of extensions contributed to other components.

Components may declare extension points and extension contributions using a simple XML syntax like the following:

```
<?xml version="1.0"?>
<component name="org.nuxeo.ecm.core.schema.TypeService">
  <implementation class="org.nuxeo.ecm.core.schema.TypeService"/>
  <extension-point name="doctype">
    <object class="org.nuxeo.ecm.core.schema.DocumentTypeDescriptor"/>
  </extension-point>
  <extension-point name="schema">
    object class="org.nuxeo.ecm.core.schema.SchemaBindingDescriptor"/>
  </extension-point>
  <extension target="org.nuxeo.ecm.core.schema.TypeService" point="doctype">
    <doctype name="File" extends="Document">
      <schema name="common"/>
      <schema name="file"/>
    </doctype>
  </extension
</component>
```

You can see here a component declaring two extension points and contributing an extension to its own `doctype` extension point.

The content of an extension element is specific to the target extension point. The extension element content is known only by the extension point. How are handled extensions not conforming to the extension point schema is unpredictable – generally they will be ignored and some errors will be logged. There is, for now, no mechanism of validating XML extensions like in Eclipse.

But Nuxeo Runtime provides an easy way to map XML extensions to real Java objects through an XML mapping mechanism called XMap. You can see that each extension point in the example above is specifying an object tag. This means the content of the XML extension should be mapped through XMap to an object instance of this type. If no object tag is specified extensions are returned as DOM elements and thus the component should perform itself the DOM parsing of extension contributions.

For details on the XML mapping see the XMap documentation and/or the JavaDoc.

XXX TODO ADD GRAPHIC

22.4.7.1. Use Cases

Here is the list of some use cases of the extension mechanism identified in the context of the Nuxeo ECM Platform:

1. to define actions and menus
2. to define content schemas (by importing XSD files)
3. to define views (view ids mapped to JSF/XHTML pages)
4. to define content objects (associate a Content Schema with a class that will provide required methods for the content object)
5. to define permissions (usable in security annotations)

6. to define PageFlows for SEAM that, optionally, can extend existing ones
7. to define business processes (in jBPM)
8. to define content transformations (doc -> pdf, doc -> html, odf -> pdf, odf -> html, etc.)
9. to define rules for the rule engine (that can be bound to some objects to run a rule only in a specific folder)
10. scriptable extensions that define scripts binded to interpreters like JavaScript, Groovy, Jython, JRuby, etc
11. to define JMS/Event queues
12. to define event types
13. to define security policies
14. to define Access Control Policies
15. to define NXCore storage backends (JCR, SQL, LDAP, etc.)
16. to define query engines
17. to define indexing engines

22.5. Supported Host Platforms

22.5.1. JBoss Integration

Nuxeo Runtime provides a SAR package containing the Runtime and the JBoss OSGi adapter. This package is an OSGi bundle that acts as the OSGi system bundle.

Any package (directory, .sar, .jar, .ear, .war or any other JBoss-supported archive) will be treated as an OSGi bundle if it contains a valid OSGi manifest. In order for these bundles to be deployed you need to have NXRuntime.sar already deployed in JBoss.

Besides the OSGi adapter and the auto registration of components through bundle manifest the JBoss adapter adds the capability to deploy runtime components as XML files located outside OSGi bundles through the JBoss deployment mechanism. This feature can be useful to register components that provide extensions to other components that can be described by plain XML without any code dependency.

Example of a plain XML component that contributes new document types:

```
<?xml version="1.0"?>
<component name="org.nuxeo.ecm.core.CoreExtensions">
  <extension target="org.nuxeo.ecm.core.schema.TypeService" point="doctype">
    <doctype name="File" extends="Document">
      <schema name="common"/>
      <schema name="file"/>
    </doctype>
    <doctype name="Folder" extends="Document">
      <schema name="common"/>
      <facet name="Folderish"/>
    </doctype>
    <doctype name="Workspace" extends="Document">
      <schema name="common"/>
      <facet name="Folderish"/>
    </doctype>
    <doctype name="Domain" extends="Document">
      <schema name="common"/>
      <facet name="Folderish"/>
    </doctype>
  </extension>
</component>
```

The `NXRuntime.sar` offers two JMX services:

1. The adapter service (`nx:service=adapter`)

   a. deploys OSGi bundles and declared components

   b. provides information about deployed bundles and components through the JBoss JMX Console

2. The XML component deployer (`nx:name=bundleDeployer,type=deployer`) that can deploy XML descriptors as OSGi components

22.5.1.1. Installation

Deploy the `NXRuntime.sar` in JBoss, then deploy your OSGi bundles as common JBoss packages.

That's all, your bundles are deployed and activated.

22.5.2. Eclipse Integration

For Eclipse, a NXRuntime.jar bundle is provided. Since Eclipse is OSGi-compliant, Nuxeo Runtime will not install any adapter (so it is not intervening on the bundle deployment).

When running on OSGi platforms the main role of the runtime is to register components declared inside OSGi bundles (as seen previously through their manifest).

Because Eclipse is not starting automatically OSGi bundles (it starts them only on demand or on class loading), you need to update Eclipse's config.ini and configure it to start Nuxeo Runtime (i.e. org.nuxeo.runtime) when Eclipse starts:

```
osgi.bundles=org.eclipse.equinox.common@2:start, org.eclipse.update.configurator@3:start, org.eclipse.core.runti
```

22.5.2.1. Installation

Update the `configuration.ini` file as described above then copy `NXRuntime.jar` inside Eclipse plugin directory.

Start Eclipse. You're done!

22.6. User Guide

22.6.1. Creating components

Components may be created either from XML descriptor files, either programatically.

In order to register components you always need a runtime context.

22.6.1.1. Runtime context

A runtime context is the context where a component is registered. Contexts should be always associated to the bundle containing the component classes. Through the context a component can access the runtime service and can load classes and retrieve resources from its bundle and other visible bundles.

Runtime Context objects depend on the current implementation of the runtime service.

Nuxeo Runtime is providing three implementation of the RuntimeContext interface:

1. `org.nuxeo.runtime.model.impl.DefaultRuntimeContext`: this is a simple implementation of a context

designed to be used outside an OSGi environment. This context is using the current thread context class loader. It should only be used in simple Java applications like JUnit tests that are not supporting OSGi bundles.

2. `org.nuxeo.runtime.osgi.OSGiRuntimeContext`: this context can be used on any platform supporting OSGi bundles. This context is using the bundle ClassLoader to load classes and find resources.

   This is the right context to use when using Nuxeo Runtime and it is working on any platform as long as it is an OSGi platform or you have an OSGi adapter for it.

3. `org.nuxeo.runtime.jboss.JBossRuntimeContext`: this is a JBoss specific context. It is used by the JBoss runtime implementation to load components deployed as standalone XML files. This context is wrapping the `DeploymentInfo` JBoss object.

Once you have a runtime context object, you can start registering components.

22.6.1.2. Creating components from XML descriptor files

To create a component using its XML description follow these steps:

1. Write the XML description of the component

   Example of a simple XML descriptor:

```
<?xml version="1.0"?>
<component name="org.nuxeo.runtime.EventService">
  <implementation class="org.nuxeo.runtime.services.event.EventService"/>
  <extension-point name="listeners">
    <object class="org.nuxeo.runtime.services.event.ListenerDescriptor"/>
  </extension-point>
</component>
```

2. Load the XML file and register the component

   In order to register a component we always need a runtime context:

```
// retrieve the current bundle
Bundle bundle = ...
// create a context given the current bundle object
RuntimeContext context = new OSGiRuntimeContext(bundle);
// load the component XML file given its location relative to the bundle root
context.deploy("OSGI-INF/MyComponent.xml");
```

Note

The current bundle object is usually retrieved from a BundleActivator in the start(BundleContext context) method. You can also lookup other bundles by their symbolic names given a Bundle object.

The context has several method of deploying (e.g. installing) components. For example the method used previously is identical to:

```
// load the component XML file given its location relative to the bundle root
URL url = context.getLocalResource("OSGI-INF/MyComponent.xml");
if (url != null) {
    context.deploy(url);
}
```

22.6.1.3. Automatic deployment of components

Another, and the easiest way to deploy components is to let the bundle deploy them when started.

This is the recommended method of deploying components.

This can be done by specifying the local paths of the XML description files inside the bundle manifest by using the Nuxeo-Component header as in the following example:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: HelloWorldExtension Plug-in
Bundle-SymbolicName: org.nuxeo.runtime.demo.HelloWorldExtension
Bundle-Version: 1.0.0
Bundle-Vendor: Nuxeo
Bundle-Localization: plugin
Require-Bundle: org.nuxeo.runtime.demo.HelloWorld
Nuxeo-Component: OSGI-INF/MyComponent.xml, OSGI-INF/MySecondComponent.xml
```

This way as soon as the bundle is started the component will be automatically deployed.

XML component are contained as resource files in that bundle and its path should be specified as relative to the bundle root

22.6.1.4. Creating components programmatically.

This method of creating components is not recommended since it is internal to Nuxeo Runtime and it depends on the implementation.

Here is an example on how you can use the API to manually register a component. We assume you are running in an OSGi environment and you have a reference to the bundle object containing the component you want to register.

```
// retrieve the current bundle
Bundle bundle = ...
RegistrationInfoImpl ri = new RegistrationInfoImpl();
// create a context associated to the current bundle
ri.context = new OSGiRuntimeContext(bundle);
ri.name = new ComponentName("my.component");
// set the class name of the component to register
ri.implementation= "org.nuxeo.runtime.example.MyComponent";
// register the component
NXRuntime.getRuntime().getComponentManager().register(ri);
```

22.6.2. Using components

22.6.2.1. Responding to life cycle events

When a component is deployed, Nuxeo Runtime will check its dependencies and if all of these ones are resolved the component is resolved and activated. (In the future, lazy activation or activation on demand will be supported too). If component dependencies are not satisfied the component will be put in a pending queue until all of its dependencies are be resolved.

When activating a component, the runtime will check if the component defines the activate life cycle method and if true, it will call it to get a chance to the component to initialize itself.

The same thing is done when deactivating the component - the runtime will check if the component defines the deactivate life cycle method and if true it will call it to get a chance for the component to dispose itself.

There are two way to define methods life cycle:

1. By implementing the `org.nuxeo.runtime.model.Component interface`

   In this case a cast to Component interface is performed and the life cycle methods are called.

```
public interface Component extends Extensible {
    public void activate(RuntimeContext context) throws Exception;
    public void deactivate(RuntimeContext context) throws Exception;
}
```

1. By simply declaring a public or protected methods on the component object using the right signature.

   In this case the Java reflection mechanism is used to call the methods.

```
public class MyComponent {
    ...
    public void activate(RuntimeContext context) throws Exception {
        ...
    }
    public void deactivate(RuntimeContext context) throws Exception {
        ...
    }
   ...
}
```

22.6.2.2. Looking up components

After a component is activated it can be retrieved using the Nuxeo Runtime API.

There are several methods to look-up a component:

Looking up the component by its name:

```
HelloComponent hc = (HelloComponent)NXRuntime.getRuntime().getComponent(
        "org.nuxeo.runtime.demo.HelloComponent");
```

Looking up the ComponentInstance object corresponding to this component. This object is a proxy to the component object:

```
ComponentInstance ci = NXRuntime.getRuntime().getComponentInstance(
        "org.nuxeo.runtime.demo.HelloComponent");
if (ci != null) {
    HelloComponent hc = (HelloComponent) ci.getInstance();
}
```

22.6.2.3. Working with extension points

Now let's take a look at how a component may define an extension point and how other components may use this extension point to contribute extensions.

22.6.2.3.1. Defining an extension point

A component may define any number of extension points. Extension points are identified inside a component by a unique name. We will describe here how to define extensions using the XML descriptor file. Extension points can also by created by hand using the internal API of Nuxeo Runtime but this is no recommended and it is not documented here.

Extension points are specified in the XML component descriptor using the extension-point tag. This tag has a required attribute name and one or more optional object sub-tags.

1. The `name` attribute

   This should be unique relative to the parent component and is used to identify the extension points inside a component.

2. The `object` sub-tag can be used to define what kind of objects are contributed by XML extensions. These objects will be created from the extension XML fragment by using the XMap engine that maps XML to Java objects through through Java annotations.

   If no `object` sub-tag is specified the extension will be contributed as a DOM element.

   The `object` tag has a required attribute `class` that specify the class name of the objects to contribute.

   The object class will be loaded using the context of the bundle that defined the extension point.

Example of a component declaring two extension points:

1. listeners
2. asyncListeners

```
<?xml version="1.0"?>
<component name="org.nuxeo.runtime.EventService">
  <implementation class="org.nuxeo.runtime.services.event.EventService"/>
  <extension-point name="listeners">
    <object class="org.nuxeo.runtime.services.event.ListenerDescriptor"/>
  </extension-point>
  <extension-point name="asyncListeners">
    <object class="org.nuxeo.runtime.services.event.AsyncListenerDescriptor"/>
  </extension-point>
</component>
```

22.6.2.3.2. Contributing an extension

Once a component declaring some extension points was activated other components may contribute extensions to that extension point.

To declare an extension the `extension` tag is used. This tag must contains a `target` and a `point` attribute.

1. `target`

   The target attribute specifies the name of the component providing the extension point

2. `point`

   The point attribute is the extension point name.

The extension element may contain arbitrary XML. The actual XML content is recognized only by the extension point to where the extension is contributed. This means you should know the correct format for the extension XML.

For this reason it is important for components to document their extension points. If the extension point is using XMap to map XML to Java objects, then you can use annotations existing on the contribution object class to know the XML format. These annotations are very easy to understand and can be used as well as a documentation for the XML extension format.

If you are familiar with Eclipse extension points, you may wonder why Nuxeo Runtime is not using an XSD schema to define the content of an XML extensions. The reason is simple: because inside our ECM project we need to be able to define any type of XML content - even configuration files from external tools we use like for example a Jackrabbit repository configuration. Defining and maintaining XSD schemas for this kind of extensions would be painful.

Anyway, using XMap to map extensions to real Java objects makes it easy to use extensions.

Here is an example on how a component is declaring some contributions to the previously defined extension points:

```
<?xml version="1.0"?>
<component name="my.component">
  <implementation class="MyComponent"/>
  <extension target="org.nuxeo.runtime.EventService" point="listeners">
    <listener class="org.nuxeo.runtime.jboss.RepositoryAdapter">
      <topic>repository</topic>
    </listener>
    <listener class="org.nuxeo.runtime.jboss.ServiceAdapter">
      <topic>service</topic>
    </listener>
  </extension>
</component>
```

You can see how the component is declaring an extension to the listeners extension point defined by the component `org.nuxeo.runtime.EventService`

The result of this declaration is that the EventService will register two listeners, one listening on events from the topic “repository”, the other on events from the topic “service”.

22.6.2.3.3. Registering contributed extension

Extensions are contributed to the target extension point immediately after the component declaring these extensions is activated. If the target component (the component declaring the extension point) was not yet activated the contributed extensions are put in a pending queue and they will be contributed as soon as the target component is activated.

A component willing to declare extension points and accept contributed extensions should declare two protected or public methods: `registerExtension` and `unregisterExtension`.

This can be done either by implementing the `Component` interface, or by declaring these methods with their correct signatures on the component object (as we have seen before for the life cycle methods).

These two methods should have the following signature:

```
public interface Extensible {

    public void registerExtension(Extension extension) throws Exception;

    public void unregisterExtension(Extension extension) throws Exception;

}
```

Note that the `Extensible` interface is extended by the `Component` interface.

When an extension is contributed the `registerExtension` method is called with an argument that points to the actual contributed extension as an `Extension` object.

Components should use this method to do something with the extension (usually to register it somewhere).

When the component contributing the extension is deactivated the Runtime will call the `unregisterExtension` method using the same `Extension` object as a parameter. This gives a chance to the extended component to unregister extensions when they become inactive.

Here is an example of how extensions are registered and unregistered:

```
public class HelloComponent implements Component {
    public final static ComponentName NAME
        = new ComponentName("org.nuxeo.runtime.demo.HelloComponent");

    Collection<HelloMessage> messages = new ArrayList<HelloMessage>();

    public void registerExtension(Extension extension) throws Exception {
        Object[] messages = extension.getContributions();
        for (Object message: messages) {
            HelloMessage msg = (HelloMessage)message;
            this.messages.add(msg);
            System.out.println("Registering message: " + msg.getMessage());
        }
    }

    public void unregisterExtension(Extension extension) throws Exception {
        Object[] messages = extension.getContributions();
        for (Object message: messages) {
            HelloMessage msg = (HelloMessage)message;
            this.messages.remove(msg);
            System.out.println("Un-Registering message: " + msg.getMessage());
        }
    }
...
}
```

You can see how the contributed objects are fetched from the Extension object and then registered into a Java Map. These contributions are objects of type HelloMessage as defined by the extension point (using the object sub-element)

The contributions are also available as a DOM element so you can use this to retrieve contributions in the case you don't use XMap to map XML extensions to Java objects. This DOM element is corresponding to the extension element from the XML component descriptor.

So if you need to retrieve the DOM representation of the extension you can do:

```
public void registerExtension(Extension extension) throws Exception {
  Element element = extension.getElement();

  // parse yourself the DOM element and extract extension data
  ...
}
```

>

22.6.3. XML Component Descriptors

In this section I will describe the most important elements composing an XML component descriptor.

You can inspect the XMap annotations on the class `org.nuxeo.runtime.model.impl.RegistrationInfoImpl` to find all elements that may compose an XML component descriptor.

A complete component descriptor may look like this:

```
<?xml version="1.0"?>
<component name="org.nuxeo.ecm.core.schema.TypeService">
    <implementation class="org.nuxeo.ecm.core.schema.TypeService"/>

    <require>org.nuxeo.ecm.core.api.ServerService</require>
    <require>org.nuxeo.ecm.core.repository.RepositoryService</require>

    <property name="author">Bogdan Stefanescu</property>
    <property name="description">The component description ...</property>

    <extension-point name="doctype">
        <object class="org.nuxeo.ecm.core.schema.DocumentTypeDescriptor"/>
    </extension-point>
    <extension-point name="schema">
        <object class="org.nuxeo.ecm.core.schema.SchemaBindingDescriptor"/>
    </extension-point>

    <extension target="org.nuxeo.ecm.core.api.ServerService"
            point="clientFactory">
        <factory class="org.nuxeo.ecm.core.api.impl.LocalClientFactory"/>
    </extension>
    <extension target="org.nuxeo.ecm.core.schema.TypeService" point="schema">
        <schema name="common" src="schema/common.xsd"/>
        <schema name="core-types" src="schema/core-types.xsd"/>
        <schema name="file" src="schema/file.xsd"/>
    </extension>
</component>
```

Each component is defined inside its own file. As you can see the root element is `component`. This element has a required attribute `name`. Apart this, all other sub-elements are optional.

Here is a list with all supported sub-elements:

1. `implementation`

   This element is used to specify the component implementation class. The element is not required since one may define plain XML components only for contributing some extensions to other components. We will refer to these components as extension components.

2. `require`

   This element can be used to specify dependencies on other components. The component will be resolved and activated only after all these dependency are resolved.

3. `property`

This element can be used to define random properties that will be available later to the component when it will be created.

4. `extension-point`

   This element is used to declare extension points. A component may declare any number of extension points.

   See more details on this in Working with extension points section.

5. `extension`

   This element can be used to declare extensions to other components (or to the current component itself).

   See more details on this in Working with extension points section.

22.7. Integration tests for Nuxeo Runtime applications

While it's obviously a good thing to unit-test one's code, it's usually not enough for a module designed to be ran as part of a Nuxeo Runtime application. It's indeed likely in this situation that the module will depend on services provided by other modules, and even maybe on their default configuration. It's even not uncommon that a project specific modules consists almost exclusively of calls to generic services provided by the base software platform.

22.7.1. The NXRuntimeTestCase base class

`org.nuxeo.runtime.test.NXRuntimeTestCase` is a base class to write JUnit tests for Nuxeo Runtime applications. It sets up the Nuxeo Runtime environment (in the `setUp` method and provides methods to control bundle and resources loading. It's designed to behave in the same manner in Maven and Eclipse situations. Therefore resources must be accessed in a way that does not depend on the actual ordering of classpath.

22.7.1.1. Loading a bundle

To load a whole OSGI bundle, use the `deployBundle` method, whose parameter is the bundle symbolic name, as specified in the manifest

22.7.1.2. Loading one contribution from a bundle

Loading a whole bundle can be too heavy, or bring unwanted default configurations. Therefore, the `deployContrib` method is provided to load just a resource (service definition, extension point contribution, etc.) from a given bundle. It takes two arguments: the bundle symbolic name, and the path to the contrib from the top of bundle.

22.7.1.3. Loading a test resouce

For resources from the test packages, just make an OSGI bundle of the test package, which can be done by creating the `META-INF/MANIFEST.MF` at the top of the target jar, and use deployContrib as above.

22.7.1.4. Sample usage

The following is an excerpt from `org.nuxeo.project.sample`:

```
public class TestBookTitleService extends NXRuntimeTestCase {

    private BookTitleService service;

    private static final String OSGI_BUNDLE_NAME = "org.nuxeo.project.sample";

    private static final String OSGI_TEST_BUNDLE = "org.nuxeo.project.sample.tests";
    public void setUp() throws Exception {
        super.setUp();
        // deployment of the whole nuxeo-project sample bundle
        deployBundle(OSGI_BUNDLE_NAME);

        service = Framework.getService(BookTitleService.class);
    }

    public void testServiceContribution() throws Exception {
        // Lookup is ensured simply by making the 'test' sub-hierarchy a
        // bundle of its own, with a MANIFEST file
        deployContrib(OSGI_TEST_BUNDLE, "sample-booktitle-test.xml");
        assertEquals("FOOBAR Test", service.correctTitle("foobar"));
    }

}
```

22.7.2. Frequent patterns

While working on an integration test, it's always worthwile to prescribe clearly what is to be tested: either API calls to services provided by other modules, consistency with configuration provided by other modules, default configuration for the module being tested. This is especially important for the non-regression aspect of the testing and the cost of maintaining the tests.

The use-cases discussed below require basic knowledge of the Nuxeo ECM framework. Implicitely, it is somehow assumed that the tested code has to interact with one service. In case of multiple target services, one'd have to choose a pattern for each of them.

22.7.2.1. Integration test against base services with testing configuration

We want to check that the API calls from the tested component to other components have the desired effect, but we don't want to rewrite the tests each time the default configuration of the other components change. Typically, this means that we need to deploy the xml contributions that define the services we need, together with the minimal configuration to tie it up together.

Example: the search service is able to configure its indexes automatically from the schemas and core types declaration. We don't want to have to update tests if someone changes the default config that ships with nuxeo-core. Ideally, this test should use `deployBundle` to set up core services, test repository, etc. and then work on dedicated schemas and core types that are loaded by `deployContrib`.

22.7.2.2. Integration test against base services and their default configuration

One can imagine here a core event listener that uses a given schema, a component that needs access to the search service to manipulate some specific documents...

In this case, we need to load the base service and its configuration exactly as they are in the real application and we do want the test to catch errors that are due to a change in said configuration. In this pattern, we'd use `deployBundle` all over the place.

It's likely however that one does not want the test to rely on the default (if any) configuration of the module being tested. If the tested component doesn't carry its configuration but still needs to be deployed within Nuxeo Runtime, `deployBundle` can be used on itself, and then `deployContrib` for the test configuration, after the test package has been upgraded to an OSGI bundle.

Variant: testing of a component and the configuration that comes along. Just think of your tested module as a "base service."

22.7.2.3. Reusing test resources from another component

This is usually neither possible nor recommended. Such situations do appear in the Nuxeo code base, and the proper solution is to provide the wished resources or classes from a `package`, precisely like `org.nuxeo.ecm.runtime.test` does.

22.8. References

1. Nuxeo.org website: http://www.nuxeo.org/
2. OSGi website: http://www.osgi.org/
3. JSR 277, 291 and OSGi, Oh My! - OSGi and Java Modularity, presented by Richard S. Hall at ApacheCon Europe 2006: http://docs.safehaus.org/download/attachments/2995/osgi-apachecon-20060628.pdf

Chapter 23. Nuxeo Core Documentation

23.1. TODO: BS

Start from the previous documentation (OOo) and update it. :-)

This chapter targets developers that would like to use directly Nuxeo Core.

23.2. Overview

Nuxeo Core is the foundation of the Nuxeo ECM project. It defines and provides all the basic services and functionalities needed to build a complete ECM product:

1. a repository model,
2. schema and document type management,
3. a query service,
4. a security model,
5. a document life cycle service,
6. a flexible core event service.

Like every Nuxeo ECM component, Nuxeo Core is running on top of Nuxeo Runtime which defines an OSGi-compatible component model.

23.2.1. Main goals

The main goals of Nuxeo Core are:

1. to provide the common services needed to build a state-of-the-art ECM product,
2. to be accessible both remotely or locally (i.e: to provide a common API accessible both from a remote JVM or directly on the local one),
3. to be deployable anywhere without any modification (through Nuxeo Runtime): in a Java EE application server like JBoss, or embedded in a desktop application like an RCP Eclipse application,
4. to be extensible and flexible; this is inherited from Nuxeo Runtime which provides an extensible component model.

23.2.2. Nuxeo Core Components

Nuxeo Core is composed by the following components:

1. NXCore: core ECM model, services and default implementation
2. NXCoreAPI: defines a client API for NXCore
3. NXCoreFacade: Java EE facade
4. NXJCRConnector: JCR storage backend that leverages jackrabbit this is the default NXCore storage backend

All these components are running on top of Nuxeo runtime (NXRuntime).

23.3. Nuxeo Core Architecture

The Nuxeo Core top level components are all roughly following the same style of development, which is structured in three layers:

1. model layer,
2. implementation layer,
3. facade layer.

There are also a number of services used by top level components to provide them with common functionalities like the schema service, query service, life cycle or security. These services are simple and cannot operate on their own – they need a context to operate on. These services are exposed through top level components and may not follow the layering presented below.

23.3.1. Model Layer (or Internal API)

Top level components provide a model (an API) that is internal to the Core – this means that they are not directly accessible from remote JVMs and should not be directly used by clients.

The model provides a generic API that defines the concepts used by the service and that may have several implementations (using different storage backends for example).

Usually this API cannot be accessed remotely since implementations may use local resources that cannot be sent over the network.

For example, the Repository model defines objects like Document, Property, Session, etc. The JCR-based implementation for the Repository model is directly wrapping JCR (Jackrabbit) nodes that cannot be detached from the local JVM and sent over the network.

23.3.2. Implementation Layer

Each service may have one or more implementations for their model. For example, the Repository service may have several implementation for the model it defines – this could be a JCR-based implementation, an SQL-based one, or something else. The same goes for the Directory service, it defines a model that could have an SQL-based implementation or an LDAP-based one.

Implementations may use very specific resources and configuration, and are hidden by the common model defined by the service. This means that implementation-specific objects or APIs are never used directly by other Core components, they are only accessed by the implementation of the internal API.

23.3.3. Facade Layer (or Public API)

On top of their model, components usually define a facade layer that enables external clients to remotely access service implementations.

This layer is also named the Public API because it defines the API exposed to clients. Any client, local or remote, must use the public API of the component, and must not make calls to the internal API.

The main requirement of the public API is to use only serializable objects that can be sent over the network and reconstructed on the client machine.

23.3.4. Deployment

The architecture presented above makes it possible to access the Core services when the Core is running inside the same JVM as the client application (e.g., when embedded in a desktop application) but also when it is on a remote JVM (e.g., deployed as a module inside an application server). In both cases the Core services are accessed in the same way – through the public API.

23.3.4.1. Local Access

23.3.4.2. Remote Access

23.3.5. Client Session

Usually a client opens a session on a Core service through the facade and can then send requests to the Core service until it closes the session.

While a client is connected to a Core service, the latter should track the client session and restore its state (if any) at each client request. When the session is closed by the client, the Core service releases any resource held by that session.

Any data passed between the client and the Core service is serializable and so it can safely be sent over the network. In this way a client can operate identically when running on the same JVM or when running on a remote one.

23.4. The Repository Model

The repository model is the main functionality provided by the Core; it represents the very raison d'être of the Core. Most of the other Core services were written as auxiliary components to perform specific needs of the repository model or to enrich it.

The repository model, as its name suggest, is describing a software component for managing repositories of documents. Repositories store documents in a tree-like structure that enables grouping documents inside folders in an hierarchic manner.

Besides the storage, the repository provides functionalities like:

1. document versioning,
2. security management,
3. document life cycle,
4. annotations,
5. SQL-like query.

23.4.1. Document and Schemas

Documents are structured objects described by a set of properties. These properties may be used to store document meta-data (e.g., creation date, author, state, etc.) or the document data itself (e.g., binary or text files, attachments, etc.).

The properties that a document may have and their types and constraints are defined through several schemas.

The repository model natively supports XML Schemas to define document schemas.

A Schema is therefore the way the structure and contents of a document is defined. Through schemas you can usually specify things like:

1. what properties are allowed,
2. the type of each property,
3. the default value of the property, if any,
4. the restrictions on the property values, if any,
5. whether a property is mandatory or not.

In order to create and use documents you first need to define their structure. For this you have to define a document type. Then you can create instances of documents of that type.

In some ways document types and schemas are similar to Java classes and interfaces. A document type may implement some schemas in the same way that Java classes implement interfaces, and a document type can extend another document type in the same way that a Java class can extend another class.

Document Types defines one or more schemas that the document structure must satisfy and some other extra properties like facets which will be discussed later.

In conclusion, the unit of work in a repository is the document. To create a new document you must specify the document type and a path. You can either use existing document types or register new types as we will see in the Extension Points section.

For more information on document types and schemas see the section.

23.4.2. Document Facets

A Facet is a behavioral property of a document. As schemas define the document structure and content, facets are used to describe behaviors or capabilities of a document.

For now facets are simple strings attached to a document type to specify a capability for documents of that type. In the future facets may evolve to more complex structures, for example to dynamically provide interfaces to manipulate documents according to a capability they offer.

Currently the Core defines two facets:

1. Folderish: adds folder capability to a document, so that it can have zero or more children documents,
2. Versionable: adds versioning capabilities to a document.

23.4.3. Document Annotations

As we've seen, a document structure and content is strictly defined by the schemas its type implements. But there are many situations where some application-specific data need to be dynamically attached to the document and retrieved later without having to modify the document schemas.

This is very useful for repository extensions that needs to store placeful (i.e., location sensitive) information on a document – information that cannot be specified by any document schema since its type is not necessarily known in advance.

Annotations are not required to be stored through the same data storage as the document itself. For example one may choose to store document in a Jackrabbit-based repository and to store annotations in a dedicated SQL database.

These annotations usually keep some internal state or data about the document. For example, a tool that may use annotations is the workflow service.

23.4.4. Document Access Control

Usually, manipulating documents requires a set of privileges to be granted to the current user. Privileges given to a user over a document are very dependent on the current context and on the document itself.

Usually privileges depend on:

1. the document location (i.e., privileges are placeful),
2. the access rules defined on document parents in the hierarchy,
3. the document state,
4. and generally on any rule that was defined over a particular location on the document parents.

Privileges are a standard example of extra information that needs to be stored on the document in a placeful manner, so it may be a perfect candidate for the annotation service.

But since privileges are very dynamic and may require expensive computations on every document that is accessed, a separate Security Service exists to manage the storage as it sees fit - and not necessarily through annotations on the document. This is more efficient from a performance point of view.

In the following subsections we will see what type of information is stored on the document to enforce security and how security checks are done. To ease comprehension of security concepts and evaluation we will begin the presentation from the smallest unit of security information to the largest one that is stored at the document level.

23.4.4.1. Access Control Entry (ACE)

This is the smallest unit specifying a security rule. It is a very simple object containing three fields:

1. principal: an authenticated entity. For example the user that opened the session on the repository is a principal – but a principal may also be a group of users.
2. permission: the kind of action that may be granted or denied for a principal. This may also be a group of permissions. This corresponds to the Java concept of privilege.
3. granting: specifies whether the given permission is granted or denied to the given principal.

Examples:

1. DENY, John, Read: an access entry that specifies that the reading is denied for the principal John.
2. GRANT, Developers, Drink: an access entry that specifies that drinking is granted for any principal from the developer group.

23.4.4.2. Access Control List (ACL)

An ACL is an ordered list of ACEs. This means it represents a set of access rules. Why ordered? Because usually when evaluating access rules the order is important. This is because evaluation stops on the first DENY or GRANT rule that match the criteria check.

Here is a simple example showing how ordering may influence the security checks. Suppose that we have a principal John that belongs to the Readers group, and an ACL that contains the following two ACEs:

1. DENY, John, Read
2. GRANT, Readers, Read

Suppose we want to check whether principal John is granted reading. Every entry in the ACL is checked (in the order they were defined) and if an entry matches the security check the evaluation stops. Using the example above, John will be denied reading even if it is a member of the Readers group. But if you swap the order of ACEs in the ACL, John will be granted reading.

23.4.4.3. Access Control Policy (ACP)

An ACP is an ordered list of ACLs. Each ACL stored in the ACP is uniquely identified by a name. The ordering is important when security is checked – ACLs at the beginning of the list will be checked first.

The ACP is the object containing the security information that is attached to a document.

Note that ACLs are inherited so that a document will inherit any defined ACLs from its parents in the hierarchy. Inherited ACLs are evaluated after evaluating the local ACLs and from the nearest parent to the remotely related parent.

You may wonder why an ACP is containing several ACLs? And what about ACL names? In a typical situation where security information may only be changed by an administrator through a user interface, a single ACL is enough.

But a complex application may have complex rules to set privileges according to the current document state or context. This is the case for a workflow engine which may decide to revoke or grant privileges depending on the document state or the context.

This means that access rules are changed not only by administrators but also by services like the workflow. To avoid collisions, every tool that needs to change access rules may use its own (named) ACL for setting these rules. If the workflow service considers that its rules are more important than the ones explicitly set by the administrator, it simply places its ACL before the one reserved for the administrator so that it will be evaluated first.

Currently there are two predefined ACLs:

1. local: the local ACL

   The local ACL is the only ACL an administrator may explicitly change through the User Interface.

2. inherited: the inherited ACL

   This ACL is computed each time a security check is performed (unless caching is used). The inherited ACL is the ACL obtained by merging all existing ACLs on the document's hierarchy. This ACL is appended to the ACL list, so it will be evaluated last.

So from a simple security unit like the ACE we end up with a sophisticated structure like inheritable ACPs.

These use cases are not artificial, they are real use cases that a mature ECM product should satisfy.

23.4.4.4. Evaluating Privileges

The evaluation mechanism has been described above. Here is an example of how an evaluation is done.

Let's say the principal John is trying to edit the document D. Editing a document requires the Write permission. Suppose the document D has the path /A/B/C/D – it is a child of the document C which is a child of the document B which is the child of the document A.

To decide if the principal John can edit this document the following steps are taken:

1. The merged ACP for the document D is computed. This ACP is the local ACP set on the document D merged with all parent ACPs. ACLs imported from the parents are appended to the local ACLs so that they will be evaluated at last.

2. Each ACL is evaluated in respect to the order defined by the ACP.

3. Each ACE is evaluated in respect to the order defined by the ACL.
4. If an ACE match a security rule regarding the principal John (or a group which it belongs) and the permission Write (or a permission group from which Write belongs) then the evaluation ends and the access right of the matching ACE is returned
5. If no matching ACE is found then the privilege is denied.

23.4.5. Life Cycle

Within organizations, documents are often regulated. At a given time, a document has a state or is within a phase. The way the document transitions in compliance with regulations from one state to another (or from one phase to another) is in most of the cases defined and managed by business processes or workflows.

Nuxeo Core itself doesn't embed a workflow engine, or still a BPM engine, as such. It only provides a generic way to define document life cycles, the way the document properties related the life cycle are stored and a way to specify which document types follow which life cycles at deployment time.

Thus, the workflow engine that will be deployed along with Nuxeo Core will leverage the API exposed by Nuxeo Core to set the life cycle properties.

The APIs defined in Nuxeo Core regarding life cycle are highly inspired from the JSR-283 specifications that are still in a draft state at the time of writing this document.

Another advantage of such a design is the fact that the life cycle state of a document will be independent of the application (i.e.: workflow variables) and will be embedded within the document itself at storage time, and thus will be exported along with the document properties.

Nuxeo provides a BPM engine that knows how to leverages the Nuxeo Core life cycle API. See http://www.nuxeo.org.

23.4.5.1. Example of document life cycle

Here is a typical lifecycle schema example:

23.4.5.2. Life cycle definition

Nuxeo Core allows one to define life cycle using extension points. (See the Nuxeo Runtime documentation for more information about extension points.). You will find at the end of this document the complete list of extension points defined by the core, you will find an example of life cycle definition there using the life cycle definition extension point.

The life cycle model defined by Nuxeo core is simple stateful, or state-transition engine. Including the following elements:

1. Life cycle definition
2. Life cycle state definition
3. Life cycle state transition definition

Again, here, no policy regarding transitions are specified. The workflow or BPM engine will deal with this. Here are the reasons:

1. It gives more flexibility regarding the policy that needs to be applied on the documents by letting dedicated BPM engines deal with that. Thus this is possible to choose which workflow engine to use for your application. (see NXWorkflow)
2. Current JCR specifications doesn't include a default policy model regarding life cycle so it appears logical to not include this ourself at this layer of the architecture

3. It simplifies the model

This is important to note that the life cycle definition is fully independent from the document types themselves which allows the reuse if life cycle for different document types.

23.4.5.3. Life Cycle Manager

The life cycle manager is responsible of the storage of the life cycle related properties. One could think of storing the life cycle property within the JCR, which is the default implementation provided by NXJCRConnector, or still one could think about storing it in a separated RDBMS apart from the content storage.

Because of this, Nuxeo provides an abstraction for this storage allowing one to define a life cycle manager per life cycle definition.

Let's take a look at the life cycle manager exposed by Nuxeo Core:

You can see that the interface is fairly simple. It basically, only specifies how to store and retrieve the state and the life cycle policy of a given document.

For an example of JCR storage see the JCRLifeCycleManager definition :

http://fisheye.nuxeo.org/browse/~raw,r=4233/nuxeo/ECMPlatform/NXJCRConnector/trunk/src/org/nuxeo/ecm/core/jcr/JC

Note this is how the JSR-283 current specifications specifies the life cycle storage repository side.

You can register your own life cycle managers using the lifecyclemanager extension point defined on the Nuxeo Core side. See the extension points chapter of this document for an example.

23.4.5.4. Document types to life cycles mapping definition

When your life cycle definitions are defined and you did specify the life cycle managers which will take care of the storage you will then need to specify associations in between document types and life cycle.

To achieve this, Nuxeo core defines an extension point allowing one to specify, independently from the document type definition, such an associations. Please, check the example at the end if this document.

23.4.5.5. Core life cycle service

Nuxeo core defines a dedicated life cycle service that is used by the Nuxeo core internals. This service is not exposed at the facade layer because we don't need it there. This service is manipulating directly the repository document themselves. (not references and thus is not suitable for remoting purpose)

Actually, the document model itself has been extended so that you can directly invoke this service through the document session itself at facade layer. See next chapter for an overview of the API.

This service is defined under this namespace org.nuxeo.ecm.core.lifecycle.LifeCycleService.

23.4.5.6. The life cycle document API and the exposure at the facade layer

The document model exposes a life cycle related API. You can take advantage of this API from the document itself if you are working at core level. Here is the api:

23.4.5.7. Core events and listeners

Nuxeo core defines a service dedicated to core events. This service is only responsible of core events and allows third party code to register listeners that will get notified when events occur (and that can take specific actions themselves)

.This service doesn't take advantage of event service such as JMS or still the NXRuntime event service at this level because it needs ti be really fast at event processing to not decrease the repository performances for instance.

By using event listener extensions, you can hook up and bridge on another synchronous or asynchronous messaging systems. Let's take some examples.

1. Nuxeo core defines a bridge to Nuxeo runtime forwarding events on the NXRuntime event service in an asynchronous way. It defines like that a local event loop shared by all components running on top of NXRuntime.

2. The NXEvents component, not part of the Nuxeo core, registers a JMS listener bridging Nuxeo core events to a dedicated JMS topic. It allows message driven beans at the Nuxeo Enterprise Platform to get the Nuxeo core events. (for instance NXAudit)

You could define whatever listeners you need to forward the Nuxeo core events on an external messaging system. See the end of this document for an example of such a registration.

23.4.6. Query Engine

The query engine is designed to provide an SQL-like language, called NXQL, to perform document and directory queries.

NXQL offers standard SQL functionality to search records, but can also take advantage of the hierarchical nature of the content repository to provide path-based searches. NXQL is used as the uniform query syntax to access several kinds of repositories. The query engine itself must process and optimize the query, and dispatch it to the different backends and tables that are referenced in the query.

Updates or creation statements are not covered and must be performed through the repository API.

For more information about the query engine, refer to the document about NXQL.

23.4.7. The Public API

As we've seen the internal repository model is not remotely accessible. Because the Nuxeo Core deployment model requires supporting both local and remote clients, the APIs are separated between an internal API and a Public API, designed to fulfill the deployment needs. Any client should use the Public API to connect to a Nuxeo Repository.

This Public API has only one limitation: any object transferred between the client and the core must be serializable. This way it can be sent over the network and restored on the client side.

So the Public API is in fact is a serializable view of the repository model. This has a performance drawback compared to the internal API since it should transform any model object like a Document into a serializable form, but has the benefit of being totally independent from the JVM where the Core runs.

The main objects composing the Public API are the:

1. DocumentModel: the serializable view of a Document.

2. DataModel: the serializable view of a document subpart described by a schema.

3. CoreSession: a session to the Core repository.

4. CoreInstance: the gateway to the Core. It uses session factories to create new sessions (connections) to the Core.

23.4.7.1. DocumentModel

The document model is a data object that completely describes a document. You can see it as a serializable view of document.

Apart being a data object this object also provides some logic. For example a document model is able to lazy load data from the storage if not already loaded or it may check permissions for a given user on the document it represent.

The data contained by document model is grouped in DataModel objects.

For each document schema there is a DataModel that contains concrete data as specified by the corresponding schema. You can see a DataModel as a data object described by a schema (i.e. a schema instance).

A document contains also data that is not defined by schemas like its internal ID, its name its parent etc. Thus, apart these data models there is some information stored as members on the document model like the document ID, the document name, a reference to the document parent, the ACP information (used for security checks), the session ID etc.

Also the data model contains the list of facets that the document type defines.

One of the most important ability of the document model is to lazy load data the first time the data is required from the client. This feature is important because the document may contain a lot of schemas and fields and it will be a performance problem to load all this data from the storage each time a document model is created.

Usually the client application is using only few DataModel fields like the Tile, Description, CreationDate etc. These are the fields commonly displayed by a tree – like explorer of the repository.

When the client is displaying or editing the document properties – then the document model will load missing data models.

To achieve this there are schemas or fields that are declared to be lazy loaded. When creating a document model from a document only the non-lazy schemas and fields are fetched from the storage. For example a blob field will be always lazy.

23.4.7.2. DataModel

As detailed above the data model is an object containing the concrete data for a document schema.

Each data model is described by the schema name and the map of fields. The data model contains no logic. It is a pure data object.

Apart the fields map the data model contains information about dirty fields (field that was modified by the client) so that when saving changes to the repository only modified fields are saved.

23.4.7.3. CoreSession

The core session is a session on the Nuxeo Core. The session is opened and closed by a client and gives the client the possibility to interact with the Core.

The Core a session connects can be located in a separate JVM or in the current one. To create remote or local sessions you need to use a specific CoreSessionFactory object. These objects are usually specified using extension points but you can also use them programatically.

After creating a session, you can begin to retrieve and modify documents through the API exposed by the CoreSession object.

Example of creating and using a session:

23.4.7.4. CoreInstance

This is the gateway to a Core instance. As mentioned above the Core may be located in a remote JVM. The CoreInstance is using CoreSessionFactory objects (declared through extension points) to connect to a Core instance and to create a session.

23.4.8. Integration with Applications Servers

The repository is plugged into an application server using the a resource adapter as specified by the J2EE Connector Architecture (JCA).

The resource adapter is write over the repository model so it is not dependent on the repository implementation (like for example JackRabbit).

Currently the resource adapter was tested only on JBoss AS

The resource adapter enables the repository to take part on transactions managed by the application server.

23.5. Extension Points

This section aims to cover all existing extension points defined by core components and to give some examples of creating new extensions.

23.5.1. Session Factories

Declaring component: org.nuxeo.ecm.core.api.CoreService

Extension point name: sessionFactory

This extension points is for registering new session factories. Session factories are used to create new Core Sessions.

Currently two session factories are provided:

1. a local session factory – that create sessions to a local Core (that is running in the same JVM as the client)
2. a remote session factory – that create sessions to a remote Core (running in a JBoss Application Server)

Example 23.1. Example Title XXX

```

```

23.5.2. LifeCycle Managers

Declaring component: org.nuxeo.ecm.core.lifecycle.LifeCycleService

Extension point name: lifecyclemanager

This extension points is for registering new life cycle managers. A life cycle manager is responsible for managing and storing document life cycle information.

Example 23.2. Example Title XXX

```

```

Chapter 24. Nuxeo Core Import / Export API

The import / export service is providing an API to export a set of documents from the repository in an XML format and then re-importing them back.

The service can also be used to create in batch document trees from valid import archives or to provide a simple solution of creating and retrieving repository data. This could be used for example to expose repository data through REST or raw HTTP requests.

Export and import mechanism is extensible so that you can easily create you custom format for exported data. The default format provided by Nuxeo EP is described below.

The import / export module is part of the `nuxeo-core-api` bundle and it is located under the `org.nuxeo.ecm.core.api.io` package.

24.1. Export Format

A document will be exported as a directory using as name the document node name and containing a `document.xml` file which hold the document metadata and properties as defined by document schemas. Document blobs if any are by default exported as separate files inside the document directory. There is also an option to export blobs inlined as Base64 encoded data inside the `document.xml`.

When exporting trees document children are put as subdirectories inside the document parent directory.

Optionally each service in nuxeo that store persistent data related to documents like the workflow, relation or annotation services may also export their own data inside the document folder as XML files.

A document tree will be exported as directory tree. Here is an example of an export tree containing relations information for a workspace named `workspace1`:

```
+ workspace1
    + document.xml
    + relations.xml

    + doc1
      + document.xml
      + relations.xml

    + doc2
      + document.xml
      + relations.xml
      + file1.blob

    + doc3
      + document.xml
```

24.1.1. document.xml format

Here is an XML that correspond to a document containing a blob. The blob is exported as a separate file:

```
<?xml version="1.0" encoding="UTF-8"?>

<document repository="default" id="633cf240-0c03-4326-8b3b-0960cf1a4d80">
  <system>
    <type>File</type>
    <path>/default-domain/workspaces/ws/test</path>
    <lifecycle-state>project</lifecycle-state>
    <lifecycle-policy>default</lifecycle-policy>
    <access-control>
      <acl name="inherited">
        <entry principal="administrators" permission="Everything" grant="true"/>
        <entry principal="members" permission="Read" grant="true"/>
        <entry principal="members" permission="Version" grant="true"/>
        <entry principal="Administrator" permission="Everything" grant="true"/>
      </acl>
    </access-control>
  </system>
  <schema xmlns="http://www.nuxeo.org/ecm/schemas/files/" name="files">
    <files/>
  </schema>
  <schema xmlns:dc="http://www.nuxeo.org/ecm/schemas/dublincore/" name="dublincore">
    <dc:valid/>
    <dc:issued/>
    <dc:coverage></dc:coverage>
    <dc:title>test</dc:title>
    <dc:modified>Fri Sep 21 20:49:26 CEST 2007</dc:modified>
    <dc:creator>Administrator</dc:creator>
    <dc:subjects/>
    <dc:expired/>
    <dc:language></dc:language>
    <dc:rights>test</dc:rights>
    <dc:contributors>
      <item>Administrator</item>
    </dc:contributors>
    <dc:created>Fri Sep 21 20:48:53 CEST 2007</dc:created>
    <dc:source></dc:source>
    <dc:description/>
    <dc:format></dc:format>
  </schema>
  <schema xmlns="http://www.nuxeo.org/ecm/schemas/file/" name="file">
    <content>
      <encoding></encoding>
      <mime-type>application/octet-stream</mime-type>
      <data>cd1f161f.blob</data>
    </content>
    <filename>error.txt</filename>
  </schema>
  <schema xmlns="http://project.nuxeo.com/geide/schemas/uid/" name="uid">
    <minor_version>0</minor_version>
    <uid/>
    <major_version>1</major_version>
  </schema>
  <schema xmlns="http://www.nuxeo.org/ecm/schemas/common/" name="common">
    <icon-expanded/>
    <icon/>
    <size/>
  </schema>
</document>
```

You can see that the generated document is containing one [system] section and one or more [schema] sections. The system section contains all system (internal) document properties like document type, path, lifecycle state and access control configuration. For each schema defined by the document type there is a schema entry which contains the document properties belonging to that schema. The XSD schema that correspond to that schema can be used to validate the content of the schema section. Anyway this is true only in the case of inlined blobs. By default, for performance reasons, the blobs are put outside the XML file in their own file.

So instead of encoding the blob in the XML file a reference to an external file is preserved: cd1f161f.blob

Here is how the same blob will be serialized when inlining blobs (an option of the repository reader):

```
  <schema xmlns="http://www.nuxeo.org/ecm/schemas/file/" name="file">
    <content>
      <encoding></encoding>
      <mime-type>application/octet-stream</mime-type>
      <data>
       b3JnLmpib3NzLnJlbW90aW5nLkNhbm5vdENvbm5lY3RFeGNlcHRpb246IENhbiBub3QgZ2V0IGNv
       bm5lY3Rpb24gdG8gc2VydmVyLiAgUHJvYmxlbSBlc3RhYmxpc2hpbmcgc29ja2V0IGNvbm5lY3Rp
       [...]
       </data>
    </content>
    <filename>error.txt</filename>
  </schema>
```

24.1.2. Inlining Blobs

There is an option to inline the blob content in the XML file as a Base64 encoded text. This is less optimized but this is the canonic format to export a document data prior to XSD validation of document schemas.

Of course this is less optimized than writing the raw blob data in external files but provides a way to encode the entire document content in a single file and in a well known and validated format.

By default when exporting documents from the repository blobs are not inlined. To activate the inlining option you must set call the method on the DocumentModelReader you are using to fetch data from the repository:

```
reader.setInlineBlobs(boolean inlineBlobs);
```

24.2. Document Pipe

An export process is a chain of three sub processes:

1. fetching data from repository
2. transforming the data if necessary
3. writing the data to an external system

In the same way an import can be defined as a chain of three sub processes:

1. fetching data from external sources
2. transforming the data if necessary
3. writing the data into the repository

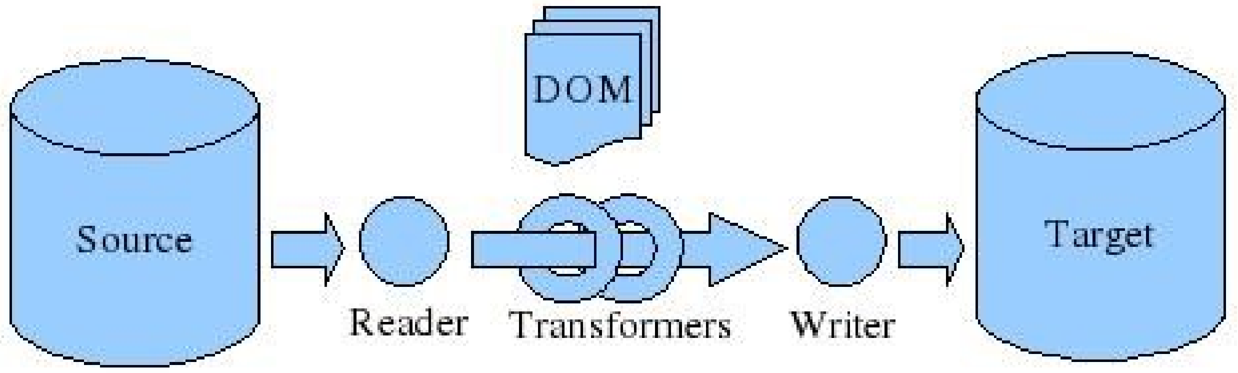
We will name the process chain used to perform imports and exports as a *Document Pipe.*

In both cases (imports and exports) a document pipe is dealing with the same type of objects:

1. A document reader
2. Zero or more document transformers
3. A document writer

So the DocumentPipe will use a reader to fetch data that will be passed through registered transformers and then written down using a document writer.

See the API Examples for examples on how to use a Document Pipe.



24.3. Document Reader

A document reader is responsible to read some input data and convert it into a DOM representation. The DOM representation is using the format explained in Document XML section. Currently dom4j Documents are used as the DOM objects.

For example a reader may extract documents from the repository and to output it as XML DOM objects. Or it may be used to read files from a file system and convert them into DOM objects to be able to import them in a Nuxeo repository.

To change the way document are extracted and transformed to a DOM representation you can implement your own Document Reader. Currently Nuxeo provides several flavors of document readers:

1. Repository readers - these category of readers are used to extract data from the repository as DOM objects. All of these readers are extending DocumentModelReader:

   - SingleDocumentReader - this one reads a single document given its ID and export it as a dom4j Document.

   - DocumentChildrenReader - this one reads the children of a given document and export each one as dom4j Document.

   - DocumentTreeReader - this one reads the entire subtree rooted in the given document and export each node in the tree as a dom4j Document.

   - DocumentListReader - this one is taking as input a list of document models and export them as domj Documents. This is useful when wanting to export a search result for example.

2. External readers used to read data as DOM objects from external sources like file systems or databases. The following readers are provided:

   - XMLDirectoryReader - read a directory tree in the format supported by Nuxeo (as described in Export Format section). This can be used to import deflated nuxeo archives or hand created document directories.

   - NuxeoArchiveReader - read Nuxeo EP exported archives to import them in a repository. Note that only zip archives created by nuxeo exporter are supported.

   - ZipReader - read a zip archive and output DOM objects. This reader can read both Nuxeo zip archives and regular zip archives (hand made). Reading a Nuxeo archive is more optimized - because Nuxeo zip archives entries are added to the archive in a predefined order that makes possible reading the entire archive tree on the fly without unziping the content of the archive on the filesystem first. If the zip archive is not recognized as a Nuxeo archive the zip will be deflated in a temporary folder on the file system and the XMLDirectoryReader will be used to read the content.

To create a custom reader you need to implement the interface `org.nuxeo.ecm.core.api.io.DocumentReader`

24.4. Document Writer

A document writer is responsible to write the documents that exit the pipe in a document store. This storage can be a File System, A Nuxeo Repository or any database or data storage as long as you have a writer that supports it.

The following DocumentWriters are provided by Nuxeo:

1. Repository Writers - These ones are writing documents to a Nuxeo repository. They are useful to perform imports into the repository.

   - `DocumentModelWriter` - writes documents inside a Nuxeo Repository. This writer is creating new document models for each one of the imported documents.

   - `DocumentModelUpdater` - writes documents inside a Nuxeo Repository. This writer is updating documents that have the same ID as the imported ones or create new documents otherwise.

2. External Writers - are writers that write documents on an external storage. They are useful to perform exports from the repository.

- `XMLDocumentWriter` - writes a document as a XML file with blobs inlined.
- `XMLDocumentTreeWriter` - writes a list of documents inside a unique XML file with blobs inlined. The document tags will be included in a root tag

```
<documents> .. </documents>
```

- `XMLDirectoryWriter` - writes documents as a folder tree on the file system. To read back the exported tree you may use XMLDirectoryReader
- `NuxeoArchiveWriter` - writes documents inside a Nuxeo azip archive. To read back the archive you may use the NuxeoArchiveReader

To create a custom writer you need to implement the interface `org.nuxeo.ecm.core.api.io.DocumentWriter`

24.5. Document Transformer

Document transformers are useful to transform documents that enter the pipe and before being sent to the writer. This way you can remove, add or modify some properties from the documents, or other information contained by the exported DOM object.

As documents are expressed as XML DOM objects you can also use XSLT transformations inside your transformer.

To create a custom transformer you need to implement the interface `org.nuxeo.ecm.core.api.io.DocumentTransformer`

24.6. API Examples

Performing exports and imports can be done by following these steps:

1. Instantiate a new DocumentPipe:

```
// create a pipe that will process 10 documents on each iteration
DocumentPipe pipe = new DocumentPipeImpl(10);
```

   The page size argument is important when you are running the pipe on a machine different than the one containing the source of the data (the one from where the reader will fetch data). This way you can fetch several documents at once improving performances.

2. Create a new DocumentReader that will be used to fetch data and put it into the pipe. Depending on the data you want to import you can choose between existing DocumentReader implementation or you may write your own if needed:

```
reader = new DocumentTreeReader(docMgr, src, true);
pipe.setReader(reader);
```

   In this example we use a DocumentTreeReader which will read an entire sub-tree form the repository rooted in 'src' document.

   The `docMgr` argument represent a session to the repository, the 'src' is the root of the tree to export and the 'true' flag means to exclude the root from the exported tree.

3. Create a `DocumentWriter` that will be used to write down the outputed by the pipe.

```
writer = new XMLDirectoryWriter(new File("/tmp/export"));
pipe.setWriter(writer);
```

In this example we instantiate a writer that will write exported data onto the file system as a folder tree.

4. Optionally you may add one or more Document Transformers to transform documents that enters the pipe.

```
MyTransformer transformer = new MyTransformer();
pipe.addTransformer(transformer);
```

5. And now run the pipe ...

```
pipe.run();
```

24.6.1. Exporting data from a Nuxeo repository to a Zip archive

```
DocumentReader reader = null;
DocumentWriter writer = null;

try {
  DocumentModel src = getTestWorkspace();
  reader = new DocumentTreeReader(docMgr, root, true);
  writer = new NuxeoArchiveWriter(new File("/tmp/export.zip"));
  // creating a pipe
  DocumentPipe pipe = new DocumentPipeImpl(10);
  pipe.setReader(reader);
  pipe.setWriter(writer);
  pipe.run();
} finally {
  if (reader != null) {
    reader.close();
  }
  if (writer != null) {
    writer.close();
  }
}
```

24.6.2. Importing data from a Zip archive to a Nuxeo repository

```
DocumentReader reader = null;
DocumentWriter writer = null;
try {
  DocumentModel src = getTestWorkspace();
  reader = new ZipReader(new File("/tmp/export.zip"));
  writer = new DocumentModelWriter(docMgr, "import-domain/Workspaces/ws");

  // creating a pipe
  DocumentPipe pipe = new DocumentPipeImpl(10);
  pipe.setReader(reader);
  pipe.setWriter(writer);
  pipe.run();
} finally {
  if (reader != null) {
    reader.close();
  }
  if (writer != null) {
    writer.close();
  }
}
```

24.6.3. Export a single document as an XML with blobs inlined.

```
DocumentReader reader = null;
DocumentWriter writer = null;

try {
  DocumentModel src = getTestWorkspace();
  reader = new SingleDocumentReader(docMgr, src);

  // inline blobs
  ((DocumentTreeReader)reader).setInlineBlobs(true);
  writer = new XMLDocumentWriter(new File("/tmp/export.zip"));

  // creating a pipe
  DocumentPipe pipe = new DocumentPipeImpl();

  // optionally adding a transformer
  pipe.addTransformer(new MyTransformer());
  pipe.setReader(reader);
  pipe.setWriter(writer); pipe.run();

} finally {
  if (reader != null) {
    reader.close();
  }
  if (writer != null) {
    writer.close();
  }
}
```

Chapter 25. Experimental Topics

25.1. Introduction

This chapter is provided to document experimental features that are being introduced in the current development branch (Nuxeo 5.2-SNAPSHOT).

This content is likely to move to other places after some time.

Note

Your feedback is welcome! If you have use cases and want to help with the development of these technologies, please join the ECM mailing list or the forum.

25.2. Runtime Support for Scripting Languages

25.2.1. Introduction

Preliminary, yet powerful, support for scripting in Nuxeo Runtime has been recently added. This makes scripting available from all Nuxeo's platform. Thanks to this new feature, you can easily uses script from your custom components. This can be very useful for a lot of use cases, like:

- dynamic rules (scripting language as DSLs)
- easily modifiable behaviors
- light and powerful configuration / customization
- etc.

Scripts have access to the whole API thanks to Java scripting integration (JSR-223).

Moreover, scripts can also be run remotely thanks to the Nuxeo Runtime command line. This allows you to create a script on your administration machine, launch it on the remote platform and get the result back. It makes scripting a killer-feature for administration scripts (ex: expire content, bulk content modification, bulk refactoring of the content repository layout, etc.).

25.2.2. Supported languages

I've just integrated scripting support through JSR 223 in nuxeo. This was integrated as a new project `nuxeo-runtime-scripting` which is in SVN but it is was not yet added in the `nuxeo.ear` build (neither in runtime SVN module).

For now only these scripting engines have been integrated:

- Jexl
- JRuby
- Jython
- Groovy
- JavaScript (Rhino)

If needed more will be added later (like PHP for example).

25.2.3. Running a Script

You can run scripts in nuxeo in 3 different ways:

- Put the script inside the `nuxeo.ear/script` directory (you should define this directory through a runtime variable). Then from the Java code you can do:

```
Framework.getService(ScriptingService.class).eval("my_script.js");
```

  where `my_script.js` is the script path relative to the script directory.

  Or you can use the JSR 322 API:

```
ScriptEngineManager factory = new ScriptEngineManager();
ScriptEngine engine = factory.getEngineByName("js");
engine.eval("absolute/path/to/my_script.js");
```

- Let the script inside your `.jar` and register it under using name as a script component. Then you can run the script as follow:

```
Framework.getService(ScriptingService.class).getScript("myScript").eval();
```

  This method chaches the compiled script so it is only supported for languages that support compilation (currently all the engines that comes in Nuxeo).

- Run a script from remote :)

  This can be used for debug, testing or administration. You write a script locally then you run it against a remote Nuxeo EP server. The script will be send to the server and executed on the server then the server will return the result (including STDOUT and STDERR) to the client.

  For security reason this feature can be disabled using a runtime property on the server.

  Here is an example on how you can run a remote script:

```
ScriptingClient client = new ScriptingClient("localhost", 62474);
URL src = RemoteTest.class.getClassLoader().getResource("test.js");
RemoteScript script = client.loadScript(src);
script.eval();
```

  For the following JavaScript script:

```
importPackage(java.lang);
importPackage(org.nuxeo.runtime);
importPackage(org.nuxeo.runtime.api);
importPackage(org.nuxeo.runtime.model);

runtime = Framework.getRuntime();
name = runtime.getName();
version = runtime.getVersion();
desc = runtime.getDescription();
println("Remote runtime: "+name+" v."+version);
println(desc);
println("----------------------------------------");
println("Registered components:");
println("----------------------------------------");
regs = runtime.getComponentManager().getRegistrations();
for (var i=0, size=regs.size(); i<size; i++){
    println(regs.get(i).getName());
}
```

  The following will be printed on the STDOUT of the client:

```
Remote runtime: OSGi NXRuntime v.1.4.0
OSGi NXRuntime version 1.4.0
----------------------------------------
Registered components:
----------------------------------------
```

```
service:org.nuxeo.ecm.core.api.DocumentAdapterService
service:org.nuxeo.ecm.core.repository.RepositoryService
service:org.nuxeo.runtime.remoting.RemotingService
service:org.nuxeo.runtime.EventService
service:org.nuxeo.ecm.platform.login.LoginConfig
...
```

So, this new feature can be used to write pure script based Nuxeo components. Also in future I will try to configure Tomcat to be able to run scripts inside servlets. This means to be able to write web pages in PHP or other supported language for Nuxeo EP ;-)

If you're willing to leverage these new scripting features in your own applications and explore new ways to use scripting to make balance "enterprise" with "agile" in the ECM field, please join the list or the forum and start contributing.

Part IV. SOA, Web Services and various integration solutions

Chapter 26. The Nuxeo Restlet API

Nuxeo integrates the Restlet framework to provide an easy way to contribute new REST API on top of the platform.

26.1. Restlet Integration

The REST API provides an easy way to call Nuxeo services from an external application. Even if REST is a very simple concept, we choose to leverage an existing REST framework to provide an REST API on top of Nuxeo. The selected framework is Restlets (http://www.restlet.org/) that provides a lightweight and flexible REST framework. The Nuxeo REST integration API (`org.nuxeo.ecm.platform.ui.web.restAPI`) provides:

- A runtime service to contribute new restlets
- Base classes for new restlets
- A main servlet that handles the routing between restlets
- An integration with the Nuxeo API and Seam context

To implement a restlet you simply have to implement the `Restlet` interface.

26.1.1. Restlet types in Nuxeo 5

Nuxeo 5 defines 3 differents types of restlets :

- Stateless restlets

  No integration with Seam and no state management. This is the original logic of Restlet.

  You can use `NuxeoBaseStalessRestlet` as base class that provides helpers for accessing main services (like the repository).

- Seam-aware restlets

  For restlet declared as Seam-aware, the Nuxeo Restlet servlet initializes the Seam context before executing the restlet. Thanks to this initialization your restlet can use injection (`@In`) to access the Seam context. This solution gives you the possibility of using existing Seam components. You don't have to use Service Platform API to access the service since you can access Seam delegates for that.

  You can use `NuxeoBaseRestlet` that provides helper API for error handling, security and URL management.

- Conversation-aware restlets

  The are Seam restlets that are tied to a Seam conversation. For these restlets, the Seam context initialization is done in order to setup the current Conversation. Conversationnal restlets must be called with a `convesationId` as parameter.

  Convesationnal restlet can be used if you need to access the current Seam context associated with a browser session. Typically this is what is used by the Firefox helper to upload files.

  You can use `NuxeoBaseRestlet` that provides helper API for error handling, security and URL management.

26.1.2. Restlet URL and parameters mapping

All restlets are bound to one or more URL patterns. These urls patterns are used by Nuxeo `RestletServlet` to determine which restlet needs to receive the call.

When defining URLs you can use {} to have parts of the URL that will be converted as parameters.

For example :

```
/{repo}/{docid}/{filename}/upload
```

will define a URL pattern with 4 parts and you will have access from withing your code to 3 parameters: repo, docId and filename.

These parameters can be used via Restlet API :

```
req.getAttributes().get("repo");
```

You can also access the standard GET/POST parameters via

```
req.getResourceRef().getQueryAsForm().getFirstValue("SomeParameter");
```

26.1.3. Contributing a new restlet

Contributing a new restlet is quite simple.

The first thing to do is to write a new restlet: you can either implement the `Restlet` Interface "by hand" or just inherit from `NuxeoBaseRestlet` or `NuxeoBaseStatelessRestet`.

Once your class is written, you need to contribute to the restlets extension point exposed by `org.nuxeo.ecm.platform.ui.web.restAPI.service.PluggableRestletService`.

```
<?xml version="1.0"?>
<component name="org.nuxeo.ecm.platform.ui.web.restAPI.contrib">

  <extension
      target="org.nuxeo.ecm.platform.ui.web.restAPI.service.PluggableRestletService"
      point="restlets">
    <restletPlugin
        name="upload"
        class="org.nuxeo.ecm.platform.ui.web.restAPI.UploadRestlet"
        enabled="true"
        useSeam="true"
        useConversation="false">
      <urlPatterns>
        <urlPattern>/{repo}/{docid}/{filename}/upload</urlPattern>
      </urlPatterns>
   </restletPlugin>
   ...
```

The `useSeam` and `useConversation` flags define how the Nuxeo Restlet servlet will handle the call.

26.2. Nuxeo default restlets

Nuxeo comes by default with very simple restlet that can be seen as samples.

26.2.1. Browse restlet

The Browse restlet is a simple way to navigate into the respository via REST.

A typicall call to list content of default repository would be :
http://127.0.0.1:8080/nuxeo/restAPI/default/*/browse

If you need to browse the document 95ce52b2-6959-4afa-bc63-396096b376b4 a typicall call would be :
http://127.0.0.1:8080/nuxeo/restAPI/default/95ce52b2-6959-4afa-bc63-396096b376b4/browse

Typical output for this restlet would be:

```
<document title="2fbf878d-9c2f-42c6-acbb-ea339ce15615"
          type="Root"
          id="2fbf878d-9c2f-42c6-acbb-ea339ce15615"
          url="/default/2fbf878d-9c2f-42c6-acbb-ea339ce15615">
  <document title="Default domain"
            type="Domain"
            id="95ce52b2-6959-4afa-bc63-396096b376b4"
            url="/default/95ce52b2-6959-4afa-bc63-396096b376b4"/>
</document>
```

This restlet uses Seam in order to have documentManager injected (this is not a need but rather a simple way of accessing the repository without using the Service API)

Browse restlet registration looks like this:

```
<restletPlugin
    name="browse"
    class="org.nuxeo.ecm.platform.ui.web.restAPI.BrowseRestlet"
    enabled="true"
    useSeam="true">
  <urlPatterns>
    <urlPattern>/{repo}/{docid}/browse</urlPattern>
  </urlPatterns>
</restletPlugin>
```

26.2.2. Export restlet

This restlet is a simple REST frontend on top of IO core service. This can be used to export a document or a document tree as XML.

A typicall call would be :

- http://127.0.0.1:8080/nuxeo/restAPI/default/2fbf878d-9c2f-42c6-acbb-ea339ce15615/export

  to export the document 2fbf878d-9c2f-42c6-acbb-ea339ce15615 as XML

- http://127.0.0.1:8080/nuxeo/restAPI/default/2fbf878d-9c2f-42c6-acbb-ea339ce15615/export?format=ZIP

  to export the document 2fbf878d-9c2f-42c6-acbb-ea339ce15615 as a ZIP archive

- http://127.0.0.1:8080/nuxeo/restAPI/default/2fbf878d-9c2f-42c6-acbb-ea339ce15615/exportTree

  to export the document 2fbf878d-9c2f-42c6-acbb-ea339ce15615 and all it's children as a Zip Archive

- http://127.0.0.1:8080/nuxeo/restAPI/default/2fbf878d-9c2f-42c6-acbb-ea339ce15615/exportTree?format=XML

  to export the document 2fbf878d-9c2f-42c6-acbb-ea339ce15615 and all it's children as XML

This restlet uses Seam in order to have documentManager injected (this is not a need but rather a simple way of accessing the repository without using the Service API)

Export restlet registration looks like this:

```
<restletPlugin
    name="export"
    class="org.nuxeo.ecm.platform.ui.web.restAPI.ExportRestlet"
    enabled="true"
    useSeam="true">
  <urlPatterns>
    <urlPattern>/{repo}/{docid}/export</urlPattern>
    <urlPattern>/{repo}/{docid}/exportSingle</urlPattern>
    <urlPattern>/{repo}/{docid}/exportTree</urlPattern>
  </urlPatterns>
</restletPlugin>
```

26.2.3. Lock restlet

This restlet provide a REST API for Lock Management.

A typicall call would be :

- GET http://127.0.0.1:8080/nuxeo/restAPI/default/2fbf878d-9c2f-42c6-acbb-ea339ce15615/Locking/status

  to get Lock status of the document 2fbf878d-9c2f-42c6-acbb-ea339ce15615

- GET http://127.0.0.1:8080/nuxeo/restAPI/default/2fbf878d-9c2f-42c6-acbb-ea339ce15615/Locking/lock

  or

  LOCK http://127.0.0.1:8080/nuxeo/restAPI/default/2fbf878d-9c2f-42c6-acbb-ea339ce15615/Locking

  to Lock status the document 2fbf878d-9c2f-42c6-acbb-ea339ce15615

- GET http://127.0.0.1:8080/nuxeo/restAPI/default/2fbf878d-9c2f-42c6-acbb-ea339ce15615/Locking/unlock

  or

  UNLOCK http://127.0.0.1:8080/nuxeo/restAPI/default/2fbf878d-9c2f-42c6-acbb-ea339ce15615/Locking

  to Unlock status the document 2fbf878d-9c2f-42c6-acbb-ea339ce15615

This restlet is stateless and uses Nuxeo Service API

Lock restlet registration looks like this

```
<restletPlugin
    name="locking"
    class="org.nuxeo.ecm.platform.ui.web.restAPI.LockingRestlet"
    enabled="true"
    useSeam="false"
    useConversation="false">
  <urlPatterns>
    <urlPattern>/{repo}/{docid}/Locking</urlPattern>
  </urlPatterns>
</restletPlugin>
```

26.2.4. Plugin upload restlet

This restlet provides a REST API for file upload that is used by the Firefox helper.

A typicall call would be :

- POST http://127.0.0.1:8080/nuxeo/restAPI/default/e5125b5e-8b9e-43bd-8959-7e7e5caf2a1b/pluginUpload/myfolder/myfil

  to upload a file

This restlet uses Seam conversation.

pluginUpload restlet registration looks like this

```
<restletPlugin
    name="pluginUpload"
    class="org.nuxeo.ecm.platform.ui.web.restAPI.PluginUploadRestlet"
    enabled="true"
    useSeam="true"
    useConversation="true">
  <urlPatterns>
    <urlPattern>/{repo}/{docid}/pluginUpload</urlPattern>
  </urlPatterns>
</restletPlugin>
```

26.3. Nuxeo RestPack

The RestPack is an additionnal component for Nuxeo EP that provides additionnal sestlets on top of Nuxeo Services

The primary goal of the RestPack is to provide the needed RestAPI to build simple JSR168 portlets (like search portlets) that communicate with Nuxeo via HTTP/XML.

26.3.1. Installing the RestPack

You can download the RestPack from Nuxeo download site or build it from source.

To deploy the module, just copy the jar file in nuxeo.ear/platform folder and restart you JBoss

26.3.2. Restlets included in the RestPack

26.3.2.1. Vocabulary Restlet

The Vocabulary restlet exports as XML the content of a given vocabulary

Sample call :

GET http://127.0.0.1:8080/nuxeo/restAPI/vocabulary/{vocName}/

parameters

- {vocName}

  Name of the vocabulary to export, must be last part of QueryPath

- lang

  GET (QueryString) parameter used to set language used to generate the lables

Sample calls

- http://127.0.0.1:8080/nuxeo/restAPI/vocabulary/subject?lang=en

```
<entries>
<entry id="arts" label="label.directories.subject.arts" translatedLabel="Arts"/>
<entry id="business" label="label.directories.subject.business" translatedLabel="Business"/>
<entry id="computers" label="label.directories.subject.computers" translatedLabel="Computers"/>
...
```

- http://127.0.0.1:8080/nuxeo/restAPI/vocabulary/continent_country?lang=en

```
<entries>
<entry id="africa" label="label.directories.continent.africa" translatedLabel="Africa">
<entry id="Algeria" label="label.directories.country.Algeria" translatedLabel="Algeria" parent="africa"/>
<entry id="Angola" label="label.directories.country.Angola" translatedLabel="Angola" parent="africa"/>
<entry id="Benin" label="label.directories.country.Benin" translatedLabel="Benin" parent="africa"/>
<entry id="Botswana" label="label.directories.country.Botswana" translatedLabel="Botswana" parent="africa"/>
<entry id="Burkina_Faso" label="label.directories.country.Burkina_Faso" translatedLabel="Burkina Faso" parent="a
<entry id="Burundi" label="label.directories.country.Burundi" translatedLabel="Burundi" parent="africa"/>
<entry id="Cameroon" label="label.directories.country.Cameroon" translatedLabel="Cameroon" parent="africa"/>
...
```

26.3.2.2. Syndication Restlet

Export as RSS or ATOM the list of children document of the targeted container

Sample call:

http://127.0.0.1:8080/nuxeo/restAPI/{repoId}/{docId}/{format}

Parameters:

- {repoId}

  Name of the target repository (use default on a standard installation)

- {docId}

  DocumentRef of the target container (use the docId present in the standard nuxeo URL when browsing the webapp)

- {format}

  Defines the syndication format: rss or atom

26.3.2.3. Workflow Tasks Restlet

Export user's workflow tasks as XML or ATOM

Sample call:

http://127.0.0.1:8080/nuxeo/restAPI/{repoId}/workflowTasks/?format=XML/ATOM

Parameters:

- {repoId}

  Name of the target repository (use default on a standard installation)

- format

  Defines the export format: xml or atom

Sample call:

http://127.0.0.1:8080/nuxeo/restAPI/workflowTasks/default/?format=xml

```
<nxt:tasks>
 <nxt:category category="None">
  <nxt:task name="review" workflowType="document_review_approbation" author="Administrator" startDate="2007-10-0
 </nxt:category>
</nxt:tasks>
```

26.3.2.4. QueryModel Restlet

Execute a search via QueryModel and returns DocumenList as XML/RSS/ATOM/JSON

Sample call :

http://127.0.0.1:8080/nuxeo/restAPI/execQueryModel/{queryModelName}

Parameters:

- {queryModelName}

  Name of the QueryModel to excecute

- format

Defines the export format: xml, atom, rss or JSON. This parameter is set as a QueryString parameter

- page

  Defines the page number you want in the result. This parameter is set as a QueryString parameter

- ascending

  Defines ordering ascending: true/false. This parameter is set as a QueryString parameter

- criteria

  Defines ordering columns used for sorting. This parameter is set as a QueryString parameter

- columns

  Defines the columns you want to be included in the resultset. This parameter is set as a QueryString parameter

  This parameter is a simple string containing schema.fieldName tokens separated by "," . The only special field is url. Default value is : dublincore.title,dublincore.description,url

- QueryModel parameters

  For stateless QueryModels, you have to specify the parameters via QP1, QP2 ... (only ordering is important)

  The value $USER is automatically replaced by the name of the current User :

  http://127.0.0.1:8080/nuxeo/restAPI/execQueryModel/USER_DOCUMENTS?QP1=$USER

  For statefull QueryModels, you have to specify the parameters as fieldName=FieldValue

Sample call:

http://127.0.0.1:8080/nuxeo/restAPI/execQueryModel/USER_DOCUMENTS?QP1=$USER&format=JSON

```
[
 {
  "title": "nouveau-fichier",
  "description": null,
  "url": "nxdoc/default/452c122d-07de-422b-b448-b0fef9534a62/view_documents",
  "id": "452c122d-07de-422b-b448-b0fef9534a62"
 },
 {
  "title": "Setup",
  "description": null,
  "url": "nxdoc/default/ae03f7bf-9967-4b5b-b37b-807fd40a6ec7/view_documents",
  "id": "ae03f7bf-9967-4b5b-b37b-807fd40a6ec7"
 },
 {
  "title": "cps",
  "description": null,
  "url": "nxdoc/default/2bad93ca-188f-4ea0-a585-9540a6ed6581/view_documents",
  "id": "2bad93ca-188f-4ea0-a585-9540a6ed6581"
 },
 {
  "title": "testMe",
  "description": null,
  "url": "nxdoc/default/e4de81a9-95e8-49ff-9146-e020f99b8bb8/view_documents",
  "id": "e4de81a9-95e8-49ff-9146-e020f99b8bb8"
 } ... ]
```

http://127.0.0.1:8080/nuxeo/restAPI/execQueryModel/USER_DOCUMENTS?QP1=$USER&format=XML&page=1

```
<results>
<pages pages="3" pageNumber="1"/>
<document id="a3154f03-6baa-4d7d-8bac-579d52a8d304" title="ssl-uil2-service"
    url="nxdoc/default/a3154f03-6baa-4d7d-8bac-579d52a8d304/view_documents"/>
```

```
<document id="e2b26d9a-9140-44f1-ae23-4ad7866911c0" title="build"
    url="nxdoc/default/e2b26d9a-9140-44f1-ae23-4ad7866911c0/view_documents"/>
<document id="bf7de6df-bca1-4d29-8e26-5019ced696fd" title="jboss-service"
    url="nxdoc/default/bf7de6df-bca1-4d29-8e26-5019ced696fd/view_documents"/>
<document id="439a11e8-642d-41f7-ae0f-4ef4d7303d79" title="postgres-jdbc2-service"
    url="nxdoc/default/439a11e8-642d-41f7-ae0f-4ef4d7303d79/view_documents"/>
<document id="467745db-565d-4350-953f-b2fa03733406" title="oil-service"
    url="nxdoc/default/467745db-565d-4350-953f-b2fa03733406/view_documents"/>
<document id="44a2441e-e265-4fee-ad17-9ec24245cccf" title="jbossmq-state"
    url="nxdoc/default/44a2441e-e265-4fee-ad17-9ec24245cccf/view_documents"/>
<document id="8d7e3077-a3be-4d6c-806c-aa97816821e2" title="oracle-jdbc2-service"
    url="nxdoc/default/8d7e3077-a3be-4d6c-806c-aa97816821e2/view_documents"/>
<document id="b431ad04-3439-441e-826b-48b5f0f0c1c8" title="as400-jdbc2-service"
    url="nxdoc/default/b431ad04-3439-441e-826b-48b5f0f0c1c8/view_documents"/>
<document id="e146c6b2-a315-4788-9bbe-5f1fc79c18ac" title="mssql-jdbc2-service"
    url="nxdoc/default/e146c6b2-a315-4788-9bbe-5f1fc79c18ac/view_documents"/>
<document id="b109a672-5ebd-4a01-8a86-79e2430c2f07" title="file-state-service"
    url="nxdoc/default/b109a672-5ebd-4a01-8a86-79e2430c2f07/view_documents"/>
</results>
```

http://127.0.0.1:8080/nuxeo/restAPI/execQueryModel/USER_DOCUMENTS?QP1=$USER&format=XML&page=1&colu

```
<results>
<pages pages="3" pageNumber="1"/>
<document id="a3154f03-6baa-4d7d-8bac-579d52a8d304" icon="/icons/note.gif" title="ssl-uil2-service"/>
<document id="e2b26d9a-9140-44f1-ae23-4ad7866911c0" icon="/icons/note.gif" title="build"/>
<document id="bf7de6df-bca1-4d29-8e26-5019ced696fd" icon="/icons/note.gif" title="jboss-service"/>
<document id="439a11e8-642d-41f7-ae0f-4ef4d7303d79" icon="/icons/note.gif" title="postgres-jdbc2-service"/>
<document id="467745db-565d-4350-953f-b2fa03733406" icon="/icons/note.gif" title="oil-service"/>
<document id="44a2441e-e265-4fee-ad17-9ec24245cccf" icon="/icons/note.gif" title="jbossmq-state"/>
<document id="8d7e3077-a3be-4d6c-806c-aa97816821e2" icon="/icons/note.gif" title="oracle-jdbc2-service"/>
<document id="b431ad04-3439-441e-826b-48b5f0f0c1c8" icon="/icons/note.gif" title="as400-jdbc2-service"/>
<document id="e146c6b2-a315-4788-9bbe-5f1fc79c18ac" icon="/icons/note.gif" title="mssql-jdbc2-service"/>
<document id="b109a672-5ebd-4a01-8a86-79e2430c2f07" icon="/icons/note.gif" title="file-state-service"/>
</results>
```

Chapter 27. Nuxeo HTTP client

Nuxeo HTTP client is a simple helper library to encapsulate restlets calls to Nuxeo platform

Even if you can easily use restlet by directly manipulating HTTP Request (this is one of the purpous of REST), if you use Java on the client side, you may choose to use Nuxeo HTTP client.

The client lib provides two main features:

- Encapsulate restlet HTTP client library
- Encapsulate Nuxeo authentication

27.1. HTTP Client Library

The library mainly encapsulate the Restlet Client API.

The main service object is a `NuxeoServer` that provides attribute configuration and call methods:

```
NuxeoServer nxServer = new NuxeoServer("http://127.0.0.1:8080/nuxeo");

nxServer.setAuthType(NuxeoServer.AUTH_TYPE_BASIC);
nxServer.setBasicAuthentication("Administrator", "Administrator");

List<String> pathParams = Arrays.asList("vocabulary", "country");

Representation res = nxServer.doRestletGetCall(pathParams, null);
```

27.2. HTTP client authentication

For authentication, the HTTO client library proposes 2 implementations:

- Basic Authentication

  Classic Web Authentication
- Shared Secret Authentication

  Designed to be able to impersonate calls

The shared Secret Authentication depends on an additionnal authentication plugin that needs to be deployed on Nuxeo side: `nuxeo-platform-login-portal-sso`.

This authentication system is based on a shared secret between the client and Nuxeo server: you need to configure this shared secret in the configuration file of the server side module, and also to pass this secret to the client http lib. Thanks to this shared secret the client will send the login name and a digest token that will be used to execute the request on the behalf of the login user

A typical use case is a JSR 168 portlet that fetches data from Nuxeo EP. The data retrieval must be done on behalf of the connected user (`request.getPrincipal()`). This allows a portlet to display user's workspaces list, or last documents without the portlet having to know the password of the user.

The authentication token sent between the client is based on the shared secret, the user login, a random data and a timestamp. Althought this should be secure enought for most needs, this trusted communication between a client application and a Nuxeo server should not be done on a HTTP connection that uses public Internet.

Here is a sample call:

```
NuxeoServer nxServer = new NuxeoServer("http://127.0.0.1:8080/nuxeo");

nxServer.setAuthType(NuxeoServer.AUTH_TYPE_SECRET);
nxServer.setSharedSecretAuthentication("JDoe","nuxeo5secretkey");

List<String> pathParams = Arrays.asList("execQueryModel","USER_DOCUMENTS");

Map<String, String> queryParams = new HashMap<String, String>();

queryParams.put("QP1", "$USER");
queryParams.put("format", "JSON");
Representation res = nxServer.doRestletGetCall(pathParams, queryParams);
```

Chapter 28. Nuxeo JSR 168 Integration

28.1. Overview

28.2. Testing Nuxeo Portlets

28.2.1. Prerequisites

We will assume that a JBoss is installed in `/opt/jboss`, and a Nuxeo platform is deployed on it.

28.2.1.1. Install the restPack project

The nuxeo-platform-restPack project contains some useful restlets, including the *sample* restlet used in the following project. You need to install it in your deployed Nuxeo EP.

Checkout the sources of the `nuxeo-platform-restPack` project:

```
svn co https://svn.nuxeo.org/nuxeo/nuxeo-addons/nuxeo-platform-restPack/trunk nuxeo-platform-restPack
```

Then, go into the newly created folder, build the project and copy the new .jar:

```
mvn clean package
cp target/nuxeo-platform-restPack-VERSION.jar /opt/jboss/server/default/deploy/nuxeo.ear/platform
```

Restart JBoss

28.2.1.2. Authentication

To enable the authentication between the portlet and the Nuxeo server, you need to install the `nuxeo-platform-login-portal-sso` project in your deployed Nuxeo EP.

Checkout the sources of the `nuxeo-platform-login-portal-sso` project:

```
svn co https://svn.nuxeo.org/nuxeo/nuxeo-addons/nuxeo-platform-login-portal-sso/trunk \
nuxeo-platform-login-portal-sso
```

To configure the authentication, edit the `Sample-Portal-SSO-descriptor-bundle.xml` file located in the `Sample` folder. Then, copy it in the `config` folder of your deployed Nuxeo EP.

```
cp Sample/Sample-Portal-SSO-descriptor-bundle.xml /opt/jboss/server/default/deployed/nuxeo.ear/config
```

Then, go into the newly created folder, build the project and copy the new .jar:

```
mvn clean package
cp target/nuxeo-platform-login-portal-sso-VERSION.jar \
/opt/jboss/server/default/deploy/nuxeo.ear/platform
```

Restart JBoss

28.2.2. Generate a sample project with nuxeo-archetype-portlet archetype

28.2.2.1. Install the archetype

Before creating a new project, you first need to install `nuxeo-archetype-portlet` on your local repository.

Checkout the sources:

```
svn co http://svn.nuxeo.org/nuxeo/org.nuxeo.archetypes/nuxeo-archetype-portlet/trunk \
nuxeo-archetype-portlet
```

Then, install the archetype on your local repository:

```
mvn clean install
```

28.2.2.2. Create a new project

To create a new project named `my-portlet` in the `com.company.sandbox` area:

```
mvn archetype:create -DartifactId=my-portlet -DgroupId=com.company.sandbox \
  -DarchetypeArtifactId=nuxeo-archetype-portlet \
  -DarchetypeGroupId=org.nuxeo.archetypes \
  -DarchetypeVersion=1.0-SNAPSHOT
```

There are two arguments:

- ArtifactId: usually the name of your project, with '-' to separate the words if there are many
- GroupId: the domain name of your project. Usually the package parent name of your classes.

Maven should have generated the following source layout:

```
    my-portlet
      |-- pom.xml
      `-- src
          `-- main
              |-- java
              |   `-- com
              |       `-- company
              |           `-- sandbox
              |               `-- portlet
              |                   `-- sample
              |                       `-- NuxeoSamplePortlet
              |-- resources
              |   `-- org
              |       `-- nuxeo
              |           `-- portlet
              |               `-- sample
              |                   `-- i18n
              |                       |-- SamplePortletMessages_en.properties
              |                       `-- SamplePortletMessages_fr.properties
              |
              `-- webapp
                  |-- index.jsp
                  `-- WEB-INF
                      |-- portlet.xml
                      |-- web.xml
                      |-- jsp
                      |   |-- edit.jsp
                      |   |-- sampleHelp.jsp
                      |   `-- sampleView.jsp
                      `-- tld
                          |-- c.tld
                          |-- fmt.tld
                          `-- fn.tld
```

28.2.3. Test the newly created portlet

Right now, you can build the portlet, but it won't deploy on Jahia. In fact, `portlet.xml` still references the wrong class.

Edit `portlet.xml` and find the line:

```
<portlet-class>org.nuxeo.portlet.sample.NuxeoSamplePortlet</portlet-class>
```

In our example, the class is now in the package `com.company.sandbox.portlet.sample`, change the line to reference the right class:

```
<portlet-class>com.company.sandbox.portlet.sample.NuxeoSamplePortlet</portlet-class>
```

You can now build the portlet and deploy it on Jahia. In your project folder:

```
  mvn clean package
  cp target/my-portlet.war /opt/jahia/tomcat/webapps/jahia/WEB-INF/var/new_webapps
```

Just wait a few seconds that Jahia finds your new portlet and deploys it. Then, go to Jahia and add the portlet. Go to the edit page of the portlet and fill the different informations (like Nuxeo Server URL, UserName, Password, ...).

When all is configured, write your name and send it. You should have "Hello your_name!" as response.

28.3. Developping Nuxeo Portlets

28.3.1. NuxeoPortlet class

Using NuxeoPortlet as base class make easier the developement of Portlets which have to communicate with Nuxeo EP.

NuxeoPortlet has some useful methods:

- error handling
- call a restlet on the configured NuxSeo server
- global preferences handling (informations stored in global preferences are shared with all the users of the portlet)

28.3.2. Project from nuxeo-archetype-portlet archetype

If you use the archetype to make your project, you will have some default behaviors:

- an administrator role is already defined (in web.xml and portlet.xml), so you can link, for instance, Jahia administrator role to your portlet administrator role. In your code, use the method `isAdministrator()` from NuxeoPortlet class to know if the current user is an administrator or not.
- a basic, but with all the informations needed to connect to a Nuxeo EP, edition page for the portlet is already done. All the informations are stored in global preferences for the portlet (through the `getGlobalPreferences()` and `saveGlobalPreferences()` methods). That means you can allow only administrators to modify these preferences, but they will be shared with all the users. You can of course

create your own edition page and override the default behaviour.

- the action requests are dispatched in convenient methods, depending of the portlet mode. For instance, in the class `NuxeoSamplePortlet`, we don't use the `processAction()` method, but `processViewAction()` when portlet is in VIEW mode and `processHelpAction()` when in HELP mode. You don't have to deal with the different portlet modes. Need to process a view action request? just use `processViewAction()`.

See the nuxeo-portlet-search project as an example of how to use or override the defaults behaviours. This portlet allows the user to make a simple search or an advanced one (like in Nuxeo EP) on the configured Nuxeo server.

28.3.3. portlet.xml

Beside the portlet class, `portlet.xml` lets you customize the portlet description, name, title.

There are also some default init parameters which are used to build the global preferences of the portlet. All the init parameters will be stored in the global preferences, so you can add the parameters you want and then use them in your portlet through the global preferences.

All the init parameters already in `portlet.xml` are needed by `NuxeoPortlet` to behave correctly.

28.3.4. Restlets

The portlet communicates with the Nuxeo server through some restlets. In our example, we call the restlet `sample` with a name as parameter.

To do a restlet call, use the method `doRestletCall()` from `NuxeoPortlet`, it makes your call and returns the result as a `Representation` (use `getText()` on it to have a `String` containing the result). The `doRestletCall()` method takes a `RestletCall` object as argument which contains the different parameters to do the restlet call: the restlet name, the path parameters and the query parameters.

See `NuxeoSamplePortlet` class or `nuxeo-portlet-search` project as example of how it works.

The `doRestletCall()` builds the URL to call from the configured Nuxeo server in the global preferences and from the parameters of the `RestletCall` object. In our example:

```
http://localhost:8080/nuxeo/sample?name=my_name
------------------------------ ------- ------------
               |                 |          `-- the query parameters of the RestletCall object
               |                 `-- the restlet name of the RestletCall object which will be
               |                     in the path parameters
               `-- the Nuxeo server URL from the global preferences
```

If all is well configured in the global preferences, you don't need to bother with authentication, just call the restlet.

To make a specific work, you have to develop your own restlets and contrib them to the `nuxeo-platform-restPack` project. Then rebuild the project, copy the .jar in your deployed Nuxeo EP and restart JBoss. You can now call your newly created restlets in your portlet.

28.4. Available portlets

28.4.1. Nuxeo Search Portlet

This portlet allows the user to search documents in a Nuxeo repository through a portlet container.

28.4.1.1. Installation

You just have to compile the portlet and install it into a portlet container, Jahia for instance, as explained in the previous sections.

After the installation of the prerequisites, you need to deploy on your portlet container the packaged portlet.

If you don't have already a packaged portlet, checkout the sources and package it:

```
svn co https://svn.nuxeo.org/nuxeo/nuxeo-addons/nuxeo-portlets/trunk/nuxeo-portlet-search \
nuxeo-portlet-search
cd nuxeo-portlet-search
mvn package
```

Then deploy the packaged portlet in Jahia, while Jahia is running:

```
cp target/nuxeo-portlet-search.war /opt/jahia/tomcat/webapps/jahia/WEB-INF/var/new_webapps
```

28.4.1.2. How to use Nuxeo Search Portlet

Before making any test, a Nuxeo EP should be running and you must configure the portlet for a Nuxeo EP:

- Attach a group or a user from Jahia (the one who will be considered as Administrator for the portlet) to the Administrator role of the portlet.
- Then, go to the edit view of the portlet and fill the different fields, so that the portlet can communicate with your Nuxeo EP.

This portlet provides the two search methods of Nuxeo EP, the simple one and the advanced one. They have the same behavior than the ones in Nuxeo EP.

Chapter 29. Desktop integration tools

This chapter is dedicated to modules to improve the user experience through integration of browser and office productivity software with a Nuxeo Enterprise Platform server.

The source code of the desktop integration tools is gathered here: `https://svn.nuxeo.org/nuxeo/desktop-integration`.

29.1. Drag and drop browser extensions

The login page of Nuxeo EP advertises two links to download and install browser extensions that makes it possible to import any files from the client's desktop file manager into the Nuxeo EP repository as new documents by drag and dropping the files or folders into the browser window.

Packaged installers for those extensions are made available on the `http://updates.nuxeo.org` URL.

29.1.1. Server side import service: the FileManagerService

All drag and drop desktop browser extensions use the `FileManagerService` and it's contributed file importers to perform the actual document creation.

TODO: add here the list of remote interfaces to the FileManagerService API (REST, SOAP?).

29.1.2. Microsoft Internet Explorer plugin

The Internet Explorer plugin source code is available at: `https://svn.nuxeo.org/nuxeo/desktop-intergration/nuxeo-dragdrop-ie-extension` . This module is a coded using the C# language and needs the dotNET runtime version 2.0 or later.

TODO: give technical details on the WS protocol used to perform the file uploads.

29.1.3. Mozilla Firefox plugin

The Firefox plugin source code is available at: `https://svn.nuxeo.org/nuxeo/desktop-intergration/nuxeo-dragdrop-ff-extension` .

The firefox plugin is packaged as a regular XPI and is platform-independent and uses the built-in Javascript API of Mozilla to upload the content of the files as POST request to a RESTful web service URL.

TODO: give details on the remote RESTlets used for this implementation.

29.2. Online document editing with LiveEdit

LiveEdit is the generic name of a set of client- and server-side components meant to seemingly integrate remote Office document editing without having to rely of manual upload files through in-browser HTML forms.

29.2.1. Functional overview

The generic functional use case is the following: the users use a standard web browser to login and browse the web interface of Nuxeo workspaces. Upon a click on a link flagged 'edit online' on a Nuxeo document having an Office file as attachment, the user automatically gets the content of the file open for editing in the right client side application (e.g. Microsoft Word, OpenOffice.org, ...)

When saving changes, the new version of the file is automatically re-uploaded to the Nuxeo server through a SOAP or RESTful web service to update the original document content and make the changes available to the other users of the workspace.

The version number can be incremented upon LiveEdit editing. A lock can be optionally set on the original document so that two users do not overwrite each other changes in concurrent LiveEdit sessions.

LiveEdit components should also support 2 auxiliary use cases:

- creation of a new Nuxeo document with an empty client-side generated attachment of the specified mimetype;
- creation of a new Nuxeo document with an client-side generated attachment preinitialized with the copy of an existing binary attachment stored elsewhere.

The following introduce the details of the 3 use cases implemented by the LiveEdit system along with the technical components at work.

29.2.2. Functional use cases

29.2.2.1. Editing the attachment of an existing document of the Nuxeo repository

The user wants to a edit a non-empty Office file stored as a property of a Nuxeo document.

The user authenticate to Nuxeo (login/password or some implementation of SSO) with her web browser (IE or Firefox) and browse her workspaces till the summary view of a document she has the rights to edit.

If the document has Blob storing property containing a non-empty office file with a mimetype flagged as live editable (MS Office and OpenOffice related mimetypes), the web interface generate a link (marked 'edit online') that automatically opens the right desktop application with the content of the attachment in opened in the editing window.

Upon saving, the desktop editor opens popup leaving the user with the option either to save a local copy or the save on the Nuxeo server. For the latter the user can also choose to increment the minor or major version number or to overwrite the current version.

Upon completion the user can check by browsing back to her document that attachment has be updated with her changes on the Nuxeo server.

29.2.2.2. Creating a new document with an empty attachment

The user wants to create a new Nuxeo document from scratch directly from an Office productivity application.

The user authenticate to Nuxeo (login/password or some implementation of SSO) with her web browser (IE or Firefox) and has the possibility to click on a document creation menu with the following items:

- New Word document
- New Powerpoint presentation
- New Excel spreadsheet
- etc. (similar options for OpenOffice.org apps)

Clicking one of those links automatically opens the right desktop application with a new empty document opened in the editing window.

Upon saving, the desktop editor opens popup leaving the user with the option either to save a local copy or the save on the Nuxeo server. For the latter the user has to choose among a flat list of candidate remote locations selected are labeled with both a title (e.g. 'My Workspace') and a path to the location (e.g. `/default-domain/workspace/my-workspace`)

Upon completion the user can check by browsing to her new document at the selected location. The file attachment of that document has the content of the saved Office file.

29.2.2.3. Creating a new document preinitialized with an existing attachment

The user wants to create a new Nuxeo document directly from an Office productivity application but reusing the content of an attachment stored as a property of an existing Nuxeo document.

The user authenticate to Nuxeo (login/password or some implementation of SSO) with her web browser (IE or Firefox) and browse her workspaces till reaching a document with a non-empty Office file attachment. The view of that file carries a link labeled `'edit online as a new document'` .

Clicking on that link opens the right desktop application preinitialized with a copy of the content of the template document attachment.

Upon saving, the desktop editor opens popup leaving the user with the option either to save a local copy or the save on the Nuxeo server. For the latter the user has to choose among a flat list of candidate remote locations selected are labeled with both a title (e.g. `'My Workspace'`) and a path to the location (e.g. `/default-domain/workspace/my-workspace`)

Upon completion the user can check by browsing to her new document at the selected location. The file attachment of that document has the content of the saved Office file.

29.2.3. Architectural overview

29.2.3.1. Overview of the components

The LiveEdit feature is built upon a set of the following interacting components:

- webapp components to generate a link to a generated XML bootstrap file describing the file to edit remotely along with connection parameters. Such a sample bootstrap file is provided in the technical annexes of this specification file.
- a browser protocol handler (i.e. plugin for Internet Explorer or Firefox) that parses the xml bootstrap size and launch the relevant desktop application according to the mimetype
- an editor plugin for each desktop application (MS Office, OpenOffice) to be able to make the desktop application fetch the file from Nuxeo through SOAP or REST GET with connection parameters provided in the bootstrap file and save it back to the server using SOAP or REST POST as well.

  Before and after editing the document, the editor plugin can fetches a list of configurable actions to call on the server through SOAP or REST GET (lock/unlock, check in/ check out, validate, etc.).
- a web service component (EJB3 stateful bean with JAXWS extensions) to implement the required methods along with the WSDL definition need by the desktop client

The source code of the various LiveEdit client side components is available at: `https://svn.nuxeo.org/nuxeo/desktop-intergration/nuxeo-liveedit-*` .

29.2.3.2. The Bootstrap client module (part 1)

The Bootstrap module must intercept the click on the "online edit" link using a dedicated protocol handler packaged as a browser plugin.

The "online edit" link has the following pattern:

```
nxedit:http://localhost:8080/nuxeo/nxliveedit.faces?action=edit&repoID=[repoID]&docRef=[docRef]&schema=
```

The protocol handler will be called by the OS/Browser and receive the URL. In turn it will receive the XML bootstrap file.

In case of create use cases the previous patterns change as follows:

- user case #2: no docid but need to provide the type of the future document and field location of the blob to be stored in:
  - `nxedit:http://localhost:8080/nuxeo/nxliveedit.faces?action=create&repoID=[repoID]&mimetype=[mime`
- user case#3: docid and field path of the original blob AND doctype and fieldpath to of the document that will host the result:
  - `nxedit:http://localhost:8080/nuxeo/nxliveedit.faces?action=createFromTemplate&templateRepoID=[te`

The Bootstrap client module rewrites each of those URIs as valid HTTP GET by swapping the prefix:

- `http://localhost:8080/nuxeo/nxliveedit.faces?[query_parameters]`

The Bootstrap client protocol must strip the "nxedit:" prefix of the URI to get the HTTP URL and thus work either with SSL or not:

- `nxedit:http://localhost:8080/nuxeo/nxliveedit.faces?[query_parameters]`

should be transformed into:

- `http://localhost:8080/nuxeo/nxliveedit.faces?[query_parameters]`

while:

- `nxedit:https://localhost:8080/nuxeo/nxliveedit.faces?[query_parameters]`

should be transformed into:

- `https://localhost:8080/nuxeo/nxliveedit.faces?[query_parameters]`

In order to generate valid `nxedit:` URIs it is strongly advised to use the JSF functions defined in the `org.nuxeo.ecm.platform.ui.web.tag.fn.DocumentModelFunctions` class.

The functions are available under the `nxd` (`http://nuxeo.org/nxweb/document`) namespace:

- `liveEditUrl(DocumentModel)` : get the `nxedit:` URL to edit a document file attachment (default File-like types)
- `liveEditUrl(DocumentModel, String, String, String)` : get the nxedit: URI to edit a document providing schema, blob field and filename field names
- `liveCreateUrl(String)` : get the nxedit: URI to create a new document of type File providing the mimetype as argument
- `liveCreateUrl(String, String, String, String, String)` : get the nxedit: URI to create a new document with parameters: mimetype, doctype, schema, blob and filename field names
- `liveCreateFromTemplateUrl(DocumentModel)` : get the nxedit: URI to create a new document of type File reusing the content of the blob of the provided template DocumentModel (assumed to have the "file" schema).
- `liveCreateFromTemplateUrl(DocumentModel, String, String, String, String, String, String)` : get the nxedit: URI to create a new document from template. Parameters are: template DocumentModel, template schema, template blob field, target document type, target schema, target, blob field name, target filename field.

29.2.3.3. The Bootstrap server module

The Bootstrap server module will be a simple Seam component called using the info passed as request parameters to generate the XML payload of the bootstrap file.

The boostrap server module is currently located in webapp-core here:

- `org.nuxeo.ecm.webapp.liveedit.LiveEditBootstrapHelper`

29.2.3.4. Bootstrap data download

The HTTP response to that GET URL is a bootstrap file containing an XML payload. This file contains:

- the action ID so the client knows to interpret it
- repo name
- document unique reference (in case of editing) or template reference (new from template)
- the document Nuxeo type of the document to create or edit
- the fieldpath hosting the binary attachment
- associated filename
- associated mime-type
- principal name
- whether the result of this editing session should be saved as a new Nuxeo document else where in the repository (creation use cases)

In case of creating from a sample document:

- the document id of the template
- the fieldpath hosting the binary attachment to initialized the editor with

The XML payload further contain a copy of all the HTTP request headers and cookies + basic auth credentials and the adress of the WSDL description of the LiveEdit webservice.

Please refer to the sample XML bootstrap file in the annexes for more details on the syntax. Some fields (eg. document reference) might be empty or missing in case of document creation (use cases #2 and #3).

29.2.3.5. The bootstrap client module (part 2)

The bootstrap module receives and parses the content of the XML bootstrap file. According to a set of configurable rules the bootstrap module launch the right editor with bootstrap file as command line parameter.

29.2.3.6. Authentication management during bootstrap

The Bootstrap client will need to do an http call to get the xml file from the server. This call must be authenticated. So the protocol handler must reuse the browser session.

29.2.3.7. The client editor and its plugin

In case of document editing (use case #1):

- call WS to get list of pre-edit actions
- display a dialog for letting user select action
- call WS to download the file

- call WS to get list of post-edit actions
- display a dialog for letting user select action
- save and upload the file to Nuxeo Server
- terminate (close the WS session)

In case of document creation (use case #2 and #3):

- call WS to get list of pre-edit actions
- display a dialog for letting user select action
- call WS to download the template file (use case #3)
- call WS to get list of post-edit actions (e.g. choose title)
- call WS for the list of candidate server locations
- display a dialog with actions and dropdown list of candidate locations
- create new document and upload the file to Nuxeo Server
- terminate (close the WS session)

29.2.3.8. Authentication of the client editor

All WS requests (SOAP and RESTful) from the editor plugin back to the WS server should reuse all the HTTP cookies along with any basic auth parameters to ensure the request will pass through any authenticating reverse proxies (e.g. CAS, mod_sso, ...) as if they were the original browser.

29.2.4. The Web Service component

It is responsible to answer the WS calls of the editor client. Most of its business logics should be defined has an overridable extension point so that customer project can change most of the LiveEdit global behavior without having to re-compile / re-package the client part.

In particular the list of candidate locations to 'save as new document' is provided by the WS server-side API to the LiveEdit client. The list should default to the list of Workspaces the user currently has the "AddChildren" permission. The WS server should dynamically compute that list according to an extensible service (i.e. overridable by a extension point) so that customer project can register a custom Java class that is responsible to implement the custom business logics in case the list of workspaces is not enough.

The location selected by default should also be defined on the server side and overridable by the same extension point.

29.2.5. More on editor launch

Based on a configuration file containing mimetype/editor mapping, the bootstrap module will launch an editor. This configuration file should look like that:

```
<editors> <editor name="MSOOfficePlugin">
<pluginType>.net<pluginType> <mime-types>
<mime-type>application/msword</mime-type> ...
</mime-types> </editor> <editor
name="OOfficePlugin">
<pluginType>exec<pluginType> <mime-types>
<mime-type>application/vnd.oasis.opendocument</mime-type>
... </mime-types> </editor> ... </editors>
```

This is very important that bootstrap client can be separated from the editors plugins, because there will plugins contributed for specific editors. The simplest and most neutral way of launching an editor plugin is just executing the editor plugin passing it a copy of the bootstrap file. This file will be the same as the one returned by Nuxeo server with additional authentication information: cookies and Login/Password.

29.2.6. More on pre- and post-editing actions

Actions available on the document may depend on the custom project specifications, and it is important that it is totally transparent for the client plugin UI: we don't want to build a version of the client plugins for each project.

So the idea is that the WebService will provide the client editor plugin a simple list of actions, the client will simply display available actions, and eventually ask the server to execute them without knowing the underlying logic.

This "action logic" is somehow close to what we already do in the web layer, an action defines an action id and a label.

If several actions can be done (like checkout + lock), then they will be combined as compound actions: checkout, lock, checkout_and_lock. This way we won't have to handle associations conditions on the client side: the user can always select at most one action: none / lock / checkout / checkout_and_lock

Chapter 30. Nuxeo WebDAV interface

The `nuxeo-platform-webdav-server` module provides a WebDAV interface on top of Nuxeo services.

30.1. WebDAV clients

There are several WebDAV clients available. Each of them has its specific behavior, and also some limitations.

30.1.1. Path vs displayName

In WebDAV specification each accessible resource must have a name (part of the URL) and can have a displayName (defines how the resource should be displayed to the user). Unfortunately, most WebDAV client don't use the displayName property even if they ask the server to send it. Because of this limitations, most client use the last part of the URL as display name (with all limitations due to URL encoding). In addition of the display problem, this has side effect on other features like renaming. When trying to rename a resource (for example after creating a new Collection/Folder), most clients will send a move operation (change the path) instead of sending a propatch on displayName.

Inside Nuxeo 5, the path (id) and the display name (title) are clearly separated:

- id/path is generated from the title but is not equal: non standard characters are escaped and length is limited to avoid having too long URLs.

  Foe example, a folder with title "New Folder" will have "new-folder" as a name.

- The name has to be unique within a given container, but 2 documents can have the same title.

30.1.2. Filesystem resource vs Nuxeo DocumentModel artifact

Inside WebDAV, each container is a collection. Most WebDAV client consider that a collection is not a document, but a simple folder. This limitation is important because Nuxeo supports folderish documents, and document types that can contain several files. Because of that, depending on the document types we may want to map, a Nuxeo document could be seen as a collection or as a resource. Unfortunately, since all clients consider collections as simple folders, this is not very useful.

30.1.3. MS Web Folder client

One of the most popular WebDAV client are Web Folders that are included inside Windows OS. Unfortunately depending on the version of Windows (including service pack) and also on the version of MSOffice the webdav client will be implemented in a different way with different bugs and behavior. Web Folders are not the worse WebDAV client, but they are definitely the less predictable...

For a complete bug listing of Web Folders depending on version please see
http://www.greenbytes.de/tech/webdav/webdavfaq.html,
http://www.greenbytes.de/tech/webdav/webfolder-client-list.html,
http://www.greenbytes.de/tech/webdav/webdav-redirector-list.html

30.2. Fooling WebDAV clients

Because of the path vs name problem, the Nuxeo 5 WebDAV connector introduce some hacks to force some webdav client displaying the correct names.

This hacks can be activated of not depending on the User-Agent header provided by each client.

30.2.1. Available hacks

- Virtual path for lief resources (ie: non collections)

  Because most client uses the last part of the URL to get the displayName, a simple hack to have resources displayed correctly is to have a specific URL for them.

  By default a resource with URL http://nxServer/nuxeo/dav/default/default-domain/workspaces/my-workspace-1/file-1 will be displayed as "file-1". But if file-1 contains a file named "MyFile.doc", when using a virtual URL like http://nxServer/nuxeo/dav/default/default-domain/workspaces/my-workspace-1/file-1/WebFolder_FileName/MyFile. then the display will be correct.

- Use filename as resource name

  The resource URL will be http://nxServer/nuxeo/dav/default/default-domain/workspaces/my-workspace-1/file-1/MyFile.doc. This is very much like the previous hack, but this is harder to resolve on the server side, because there may be several file resources containing MyFile.doc in the same container.

- Use fake get parameters for collections names

  Getting the good displayName for collection resources is a little bit harder, previous hacks won't work.

  In some webdav client, the url parsing is so weak, that we can use URLs like : http://nxServer/nuxeo/dav/default/default-domain/workspaces/my-workspace-1?displayName=/My%20Workspace% to have the collection correctly displayed as "My Workspace 1". Although this hack does not work with most clients, it works with most Web Folders version.

30.2.2. Configuring Nuxeo WebDAV connector for each client.

The Nuxeo WebDAV connector enables you to configure for each WebDAV clients:

- The hacks you want to enable
- If you want full or relative URLs
- If your client needs MS specific DAV Headers (required for some versions of Web Folders)

Configuration can be contributed using a simple Extension Point.

```
<?xml version="1.0"?>
<component name="org.nuxeo.ecm.platform.webdav.config.defaultContrib">

  <extension target="org.nuxeo.ecm.platform.webdav.config.WebDavConfigurationService" point="DavClientConfigurat
    <documentation>
      Configuration for MS Web Folders (both versions) and WebDrive.
    </documentation>
    <davClientConfiguration name="MSWebFolders" enabled="true">
      <needGetParameterForCollectionNamming>true</needGetParameterForCollectionNamming>
      <needVirtualPathForLief>false</needVirtualPathForLief>
      <useFileNameAsRessourceName>true</useFileNameAsRessourceName>
      <needFullURLs>true</needFullURLs>
      <needMSDavHeader>true</needMSDavHeader>
      <userAgentPatterns>
        <pattern>Microsoft-WebDAV</pattern>
        <pattern>Microsoft Data Access Internet Publishing Provider</pattern>
      </userAgentPatterns>
    </davClientConfiguration>
   ...
```

As you can see, each configuration set is attached to one or more User-Agent substring. Unfortunately, Web Folders clients do not send information about their version, even if it would be very helpful.

30.3. Nuxeo EP WebDAV implementation

30.3.1. Nuxeo EP WebDAV-specific features

Because of the limitations explained earlier, the Nuxeo EP WebDAV implementation provides some features to be able to work with existing WebDAV clients.

30.3.1.1. VirtualPath management

When URLs hack are activated, the clients will be able to display the resources with their displayName (title) instead of the name. But some of them will be fooled enough to use this displayName in URLs to do propfind or move calls. Because of that, the server must be able to resolve URLs that are constitued partially of path and partially of displayNames like:
http://nxServer/nuxeo/dav/default/default-domain/workspaces/My%20Workspace%201.

The Nuxeo EP WebDAV connector implements a custom URL resolver that is able to resolve these URLs. In also maintains a cache of these virtual URLs to speed up resolution (and also make it more consistent is case of name collisions).

30.3.1.2. WebDAV resource adapters

Displaying your Nuxeo documents as simple files (that's what will do most WebDAV client) can be very restrictive.

The Nuxeo WebDAV connector uses DocumentModel adapters to define how a Nuxeo Document must be mapped to a WebDAV resources. This adapter defines:

- How the display name of your resource will be generated.

  Simple document title or filename of the main file field...

  The default built-in behavior will be to return the filename for each document that has the file schema and otherwise return the title.

- How your DocumentModel will respond to a GET request.

  The default built-in behavior will be to return:

  - a folder listing as HTML for each folderish resource
  - the binary file for Documents that uses the file schema
  - the XML export for other non folderish resources

New DavAdapters can be contributed to define specific WebDAV mapping for your document types. For that, the Nuxeo WebDAV connector provides an extension point to let you register a new class implementing`org.nuxeo.ecm.platform.webdav.adapters.DavResourceAdapter` and associate it to a document type.

```
<extension target="org.nuxeo.ecm.platform.webdav.config.WebDavConfigurationService"
      point="DavAdapter">

  <davAdapter name="NoteDavAdapter" enabled="true">
    <typeName>Note</typeName>
    class>org.nuxeo.ecm.platform.webdav.adapters.NoteDavResourceAdapter</class>
  </davAdapter>
</extension>
```

30.3.2. Known limitations

Locking management is restricted to exclusive write locks, exactly as in Nuxeo Core.

Propset is for now not implemented as none of the used WebDAV client seems to use this method.

WebDAV versioning extension are not implemented.

30.4. Using the Nuxeo WebDAV connector

30.4.1. Installing the WebDAV connector

The Nuxeo WebDAV Connector is an optional additional component for Nuxeo EP.

In order to install the WebDAV features, you need to download the WebDAV Connector jar (or build it from source), copy the jar in `nuxeo.ear/platform` and restart your server.

The WebDAV connector works for both 5.1 and 5.2 versions of Nuxeo EP, you just need to download the associated version or build it using the right POM file.

30.4.2. Connecting a client to Nuxeo WebDAV connector

The WebDAV URL for the default domain is http://$NuxeoServer/nuxeo/dav/default/default-domain.

In order to use MS Web Folders, you just need to go to "My Network Places" and choose "Add a new Network place". You can then enter the Nuxeo WebDAV URL and login/password.

Chapter 31. Reporting: Eclipse BIRT Driver

31.1. Overview

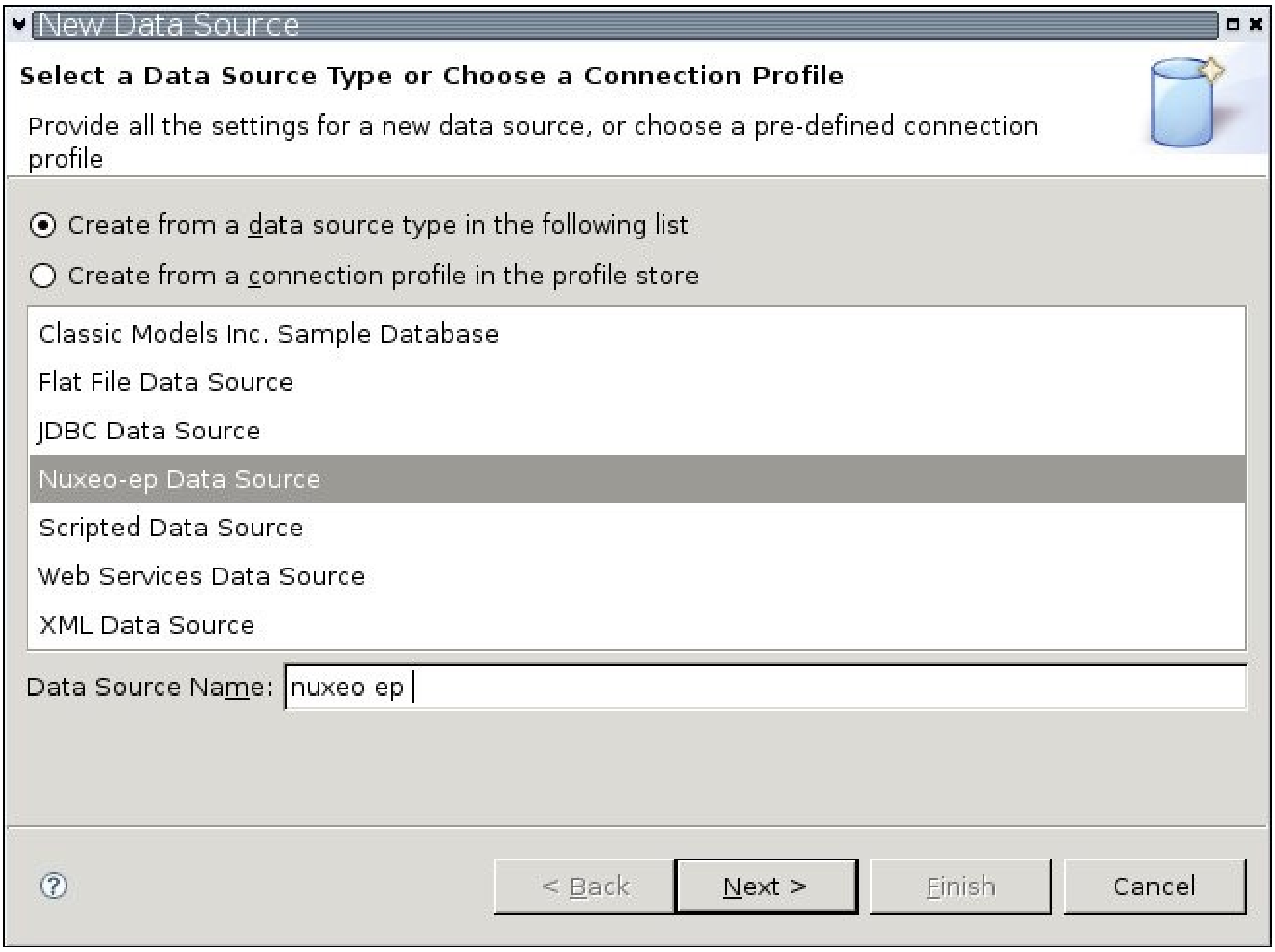
BIRT is an open source Eclipse-based reporting system that integrates with Java/J2EE application to produce compelling reports. The BIRT driver for Nuxeo enables BIRT to be used as reporting engine for the Nuxeo Content Repository. It basically gives an easy way to query the repository and create report from the results. Thanks to BIRT, reports can be run inside Eclipse or as servlets on the server side.

31.2. How to use it

In a Eclipse instance with Business Intelligence and Reporting Tools installed (http://www.eclipse.org/birt/phoenix/), deploy the following Nuxeo plugins:

- org.nuxeo.common
- org.nuxeo.ecm.client
- org.nuxeo.ecm.core.api
- org.nuxeo.ecm.core.query
- org.nuxeo.ecm.core.schema
- org.nuxeo.ecm.jboss_connector
- org.nuxeo.ecm.platform.search.api
- org.nuxeo.ecm.platform.usermanager.api
- org.nuxeo.logging
- org.nuxeo.runtime
- org.nuxeo.runtime.config
- org.nuxeo.birt.oda.nuxeoep
- org.nuxeo.birt.oda.nxueoep.ui

When eclipse is restarted a new "Data Source Should be available"


New Data Source

Select a Data Source Type or Choose a Connection Profile

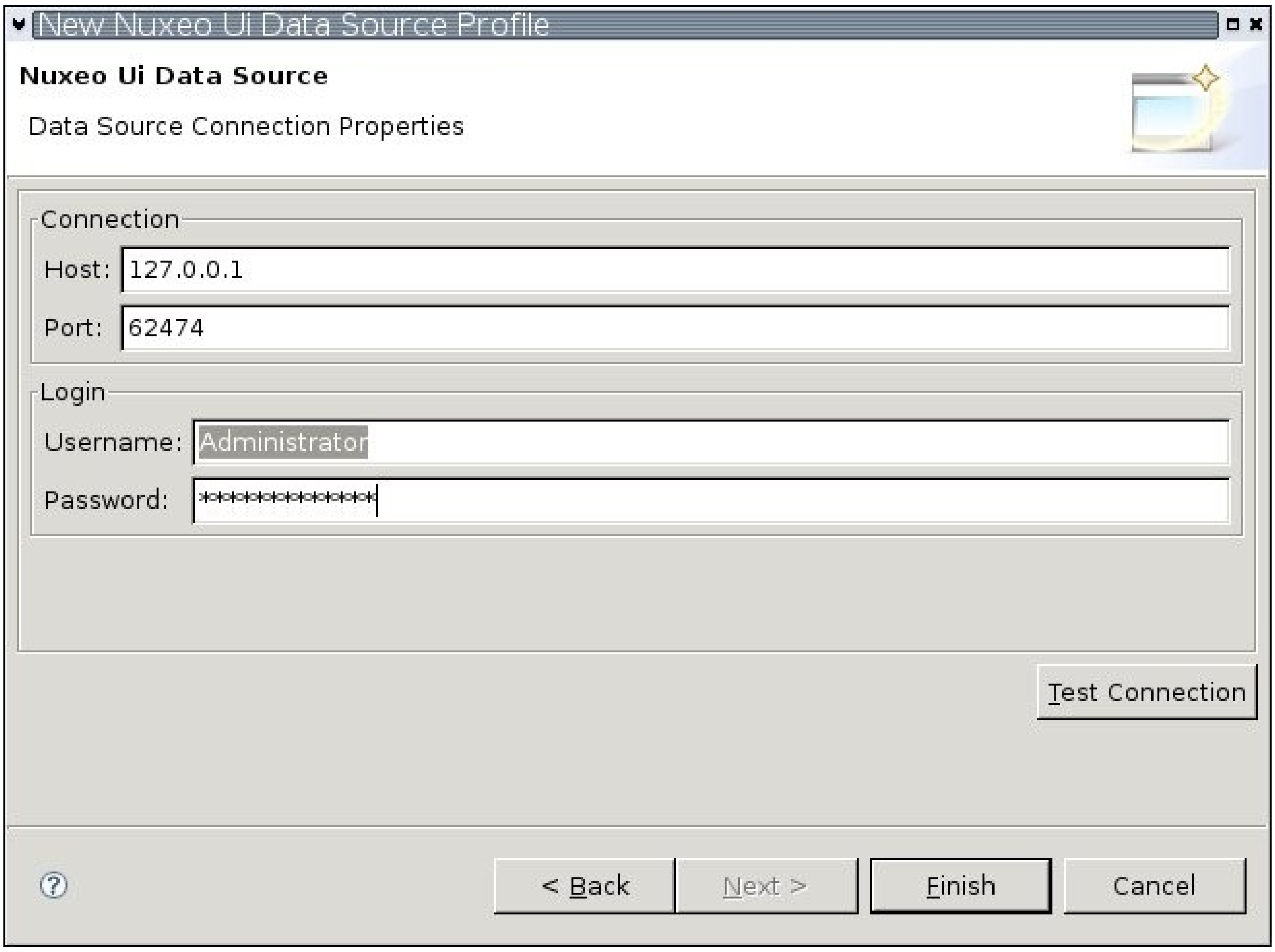
Provide all the settings for a new data source, or choose a pre-defined connection profile

[✓ Create from a data source type in the following list]

Create from a connection profile in the profile store

Classic Models Inc. Sample Database
Flat File Data Source
JDBC Data Source
Nuxeo-ep Data Source
Scripted Data Source
Web Services Data Source
XML Data Source

Data Source Name: nuxeo ep


New Nuxeo Ui Data Source Profile

Nuxeo Ui Data Source

Data Source Connection Properties

Connection

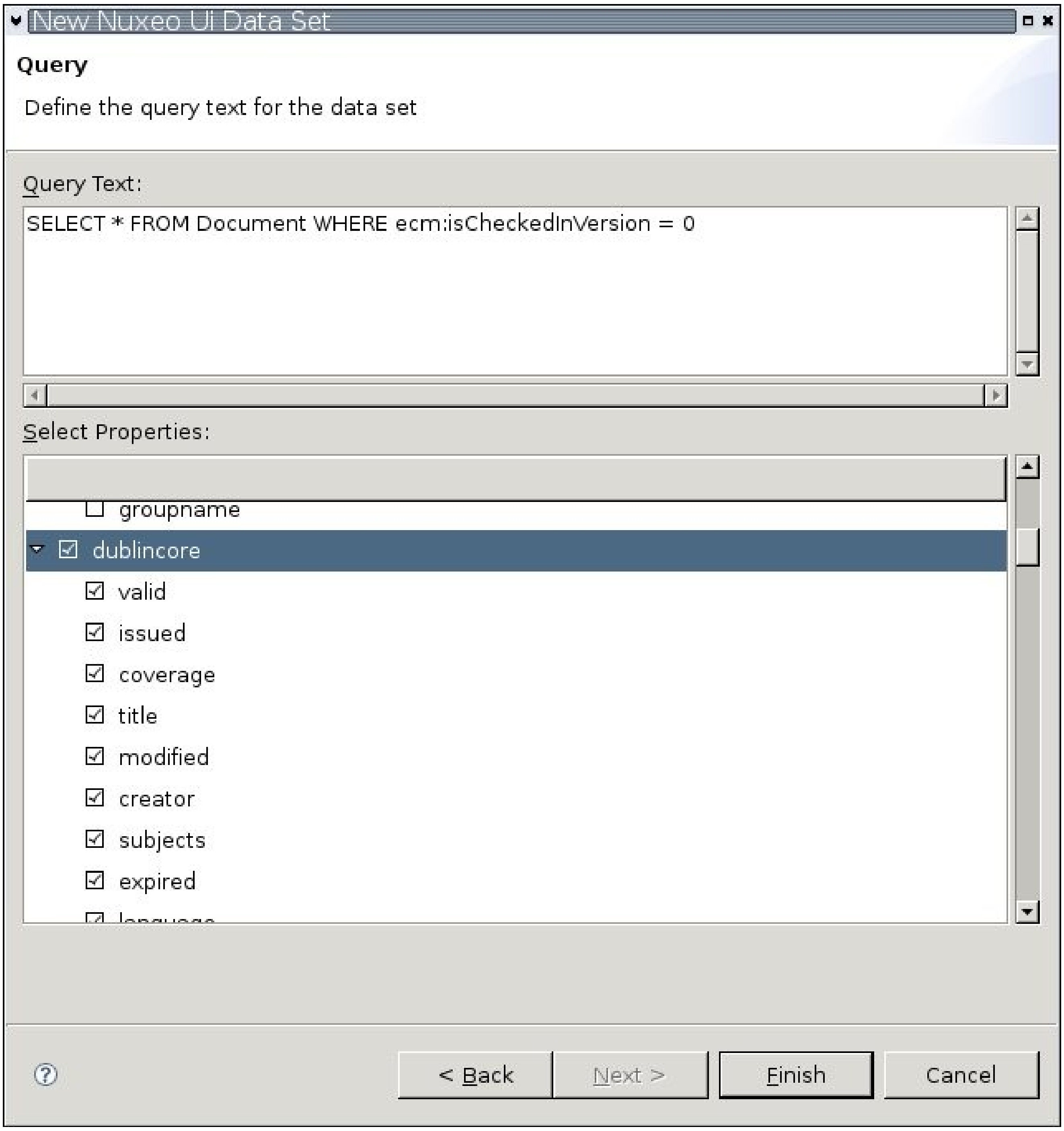
Host: 127.0.0.1

Port: 62474

Login

Username: Administrator

Password: **************



Query

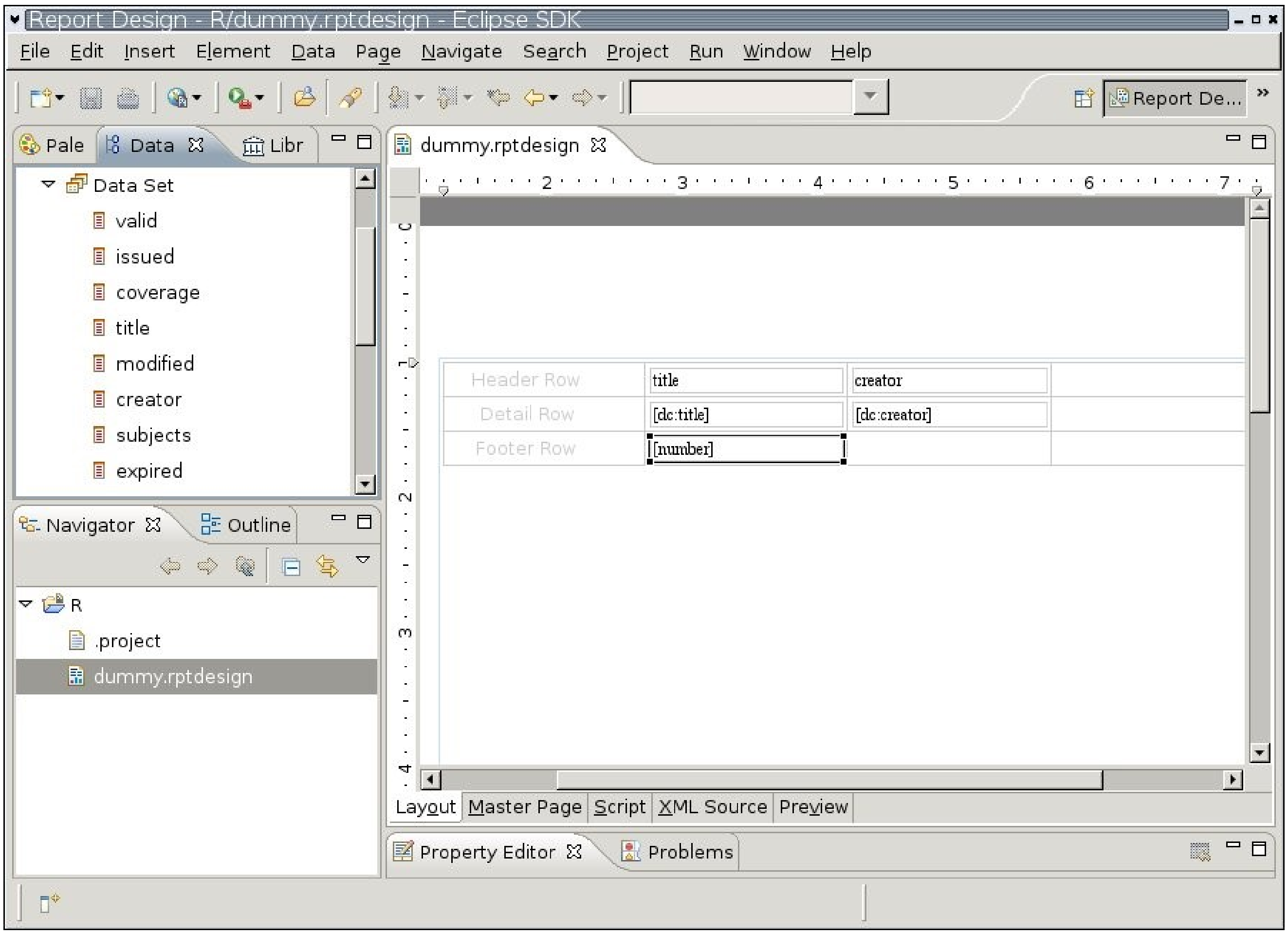
Define the query text for the data set

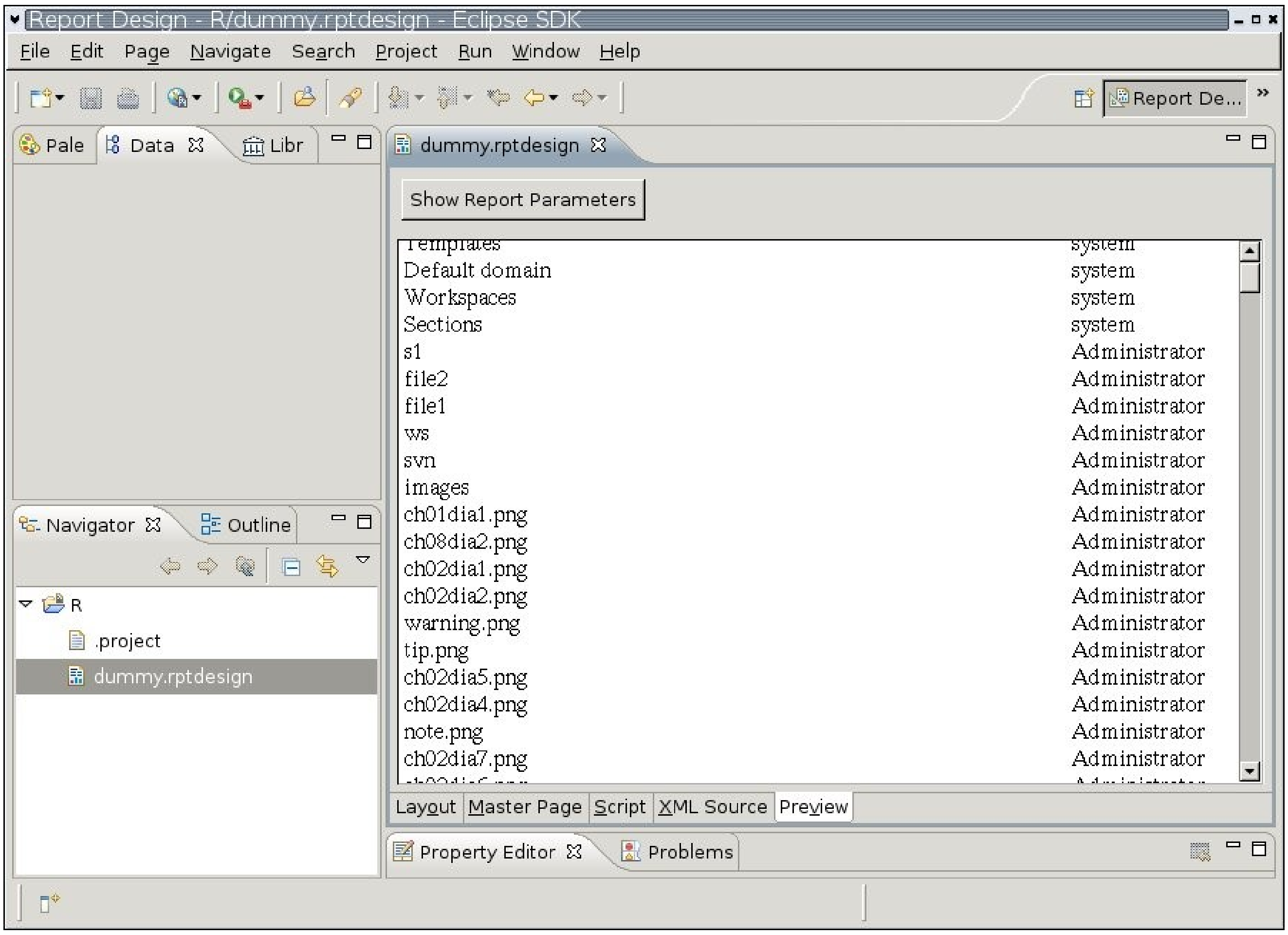
Query Text:

```
SELECT * FROM Document WHERE ecm:isCheckedInVersion = 0
```

Select Properties:

- ☐ groupname
- ☑ dublincore
  - ☑ valid
  - ☑ issued
  - ☑ coverage
  - ☑ title
  - ☑ modified
  - ☑ creator
  - ☑ subjects
  - ☑ expired
  - ☑ language




Report Design - R/dummy.rptdesign - Eclipse SDK
File Edit Page Navigate Search Project Run Window Help
Report De...
Pale Data Libr
Navigator Outline
R
.project
dummy.rptdesign
Show Report Parameters
Layout Master Page Script XML Source Preview
Property Editor Problems

```
Templates                              system
Default domain                         system
Workspaces                             system
Sections                               system
s1                                     Administrator
file2                                  Administrator
file1                                  Administrator
ws                                     Administrator
svn                                    Administrator
images                                 Administrator
ch01dia1.png                           Administrator
ch08dia2.png                           Administrator
ch02dia1.png                           Administrator
ch02dia2.png                           Administrator
warning.png                            Administrator
tip.png                                Administrator
ch02dia5.png                           Administrator
ch02dia4.png                           Administrator
note.png                               Administrator
ch02dia7.png                           Administrator
```

Figure 31.5.

Figure 31.4. Data set is ready to be used in report

Figure 31.3. In the Data Set dialog, type NXQL query and select fields & schemas you would like to use in the report

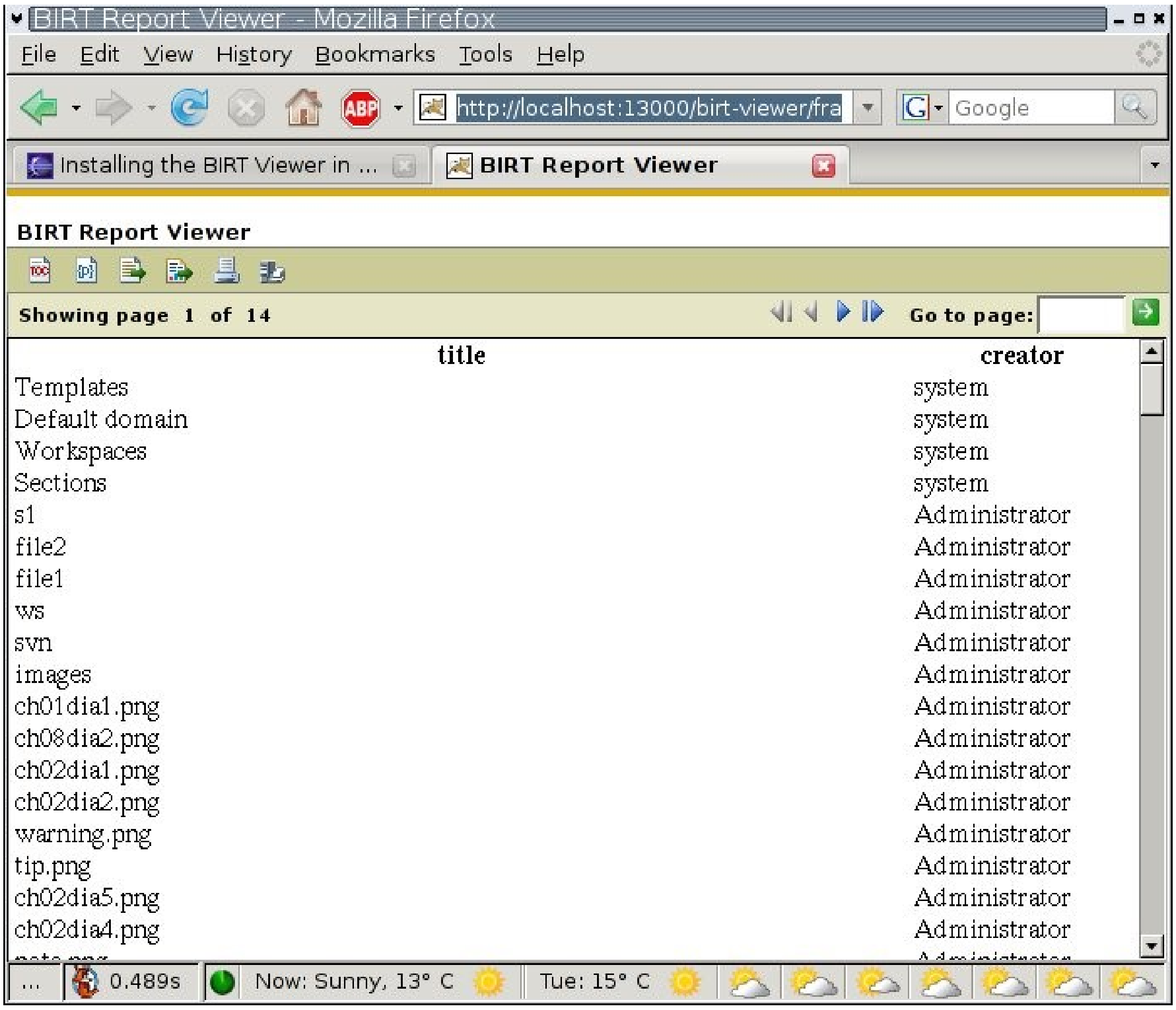
Figure 31.2. Fill login information

Figure 31.1. In the Data Source Type screen, select "Nuxeo Data Source":

31.3. Tomcat integration HOWTO

The new created report can be deployed in Tomcat to be available online:

1. Install the birt-viewer application following instruction from http://www.eclipse.org/birt/phoenix/deploy/viewerSetup.php
2. Deploy the plugins listed in previous section (except org.nuxeo.birt.oda.nxueoep.ui) in $TOMCAT_HOME/webapps/birt-viewer/WEB-INF/platform/plugins
3. Just copy the report file from workspace to $TOMCAT_HOME/webapps/birt-viewer/report
4. The report is available with an URL similar to http://localhost:13000/birt-viewer/frameset?__report=report/dummy.rptdesign


BIRT Report Viewer - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:13000/birt-viewer/fra
Google
Installing the BIRT Viewer in ...
BIRT Report Viewer
BIRT Report Viewer
Showing page 1 of 14
Go to page:
0.489s
Now: Sunny, 13° C
Tue: 15° C

title	creator
Templates	system
Default domain	system
Workspaces	system
Sections	system
s1	Administrator
file2	Administrator
file1	Administrator
ws	Administrator
svn	Administrator
images	Administrator
ch01dia1.png	Administrator
ch08dia2.png	Administrator
ch02dia1.png	Administrator
ch02dia2.png	Administrator
warning.png	Administrator
tip.png	Administrator
ch02dia5.png	Administrator
ch02dia4.png	Administrator

Figure 31.6.

Part V. Administration

Chapter 32. Administration

In this chapter you will learn how to setup and manage a Nuxeo EP server in a production context (as opposed to a demo / evaluation or development context).

32.1. SMTP Server configuration

On the Nuxeo EP built-in types, you can manage e-mailing and notifications. Before getting this features working, you need to configure your SMTP server. Nuxeo EP relies on the application server mail service for mailing stuff. So you just need to configure the `mail-service.xml` file in `$JBOSS_HOME/server/default/deploy/`. You can find examples of how to use this file in the JBoss wiki, and detailed information about the properties of this file in the JavaMail Javadoc.

32.2. RDBMS Storage and Database Configuration

To run Nuxeo EP for real-world application, you need to get rid of the default embedded database and set up a real RDBMS server (such as PostgreSQL, MySQL, Oracle, etc.) to store Nuxeo EP's data.

In order to define a SQL DB as repository back-end you have to :

- install the JDBC driver for your DBMS,
- configure the Nuxeo Core storage, modifying the default repository configuration,
- configure your database.
- Eventually, configure storage for other Nuxeo services.
- You may also add a new repository configuration (and optionally disable the old one).

32.2.1. Storages in Nuxeo EP

Nuxeo EP manages several types of data: Documents, relations, audit trail, users, groups ...

Each type of data has it's own storage engine and can be configured separatly. All storages can use RDBMS but some of them can also use the filesystem.

This means you can have a RDBMS only configuration or a mixed configuration using RDBMS and filesystem. You can even use sereval RDBMS if you find a use case for that.

For a lot of services, RDBMS access are encapsulated by JPA or hibernate calls, so the storage should be RDBMS agnostic as long as there is a hibernate dialect for the DB.

32.2.2. Install the JDBC driver

To enable the connection to the database, you first need to install the JDBC driver into `$JBOSS_HOME/server/default/lib/`.

Here are some drivers download locations:

- PostgreSQL JDBC Drivers
- MySQL JDBC Drivers

32.2.3. Configure Nuxeo Core storage

This part will explain you how to configure Nuxeo Core repository, using Jackrabbit as storage back-end, to store the repository's data into your RDBMS.

This documentation will use PostgreSQL 8.x as an example. The setup for other RDBMS should be very similar.

32.2.3.1. Set up the repository configuration

Edit `$JBOSS_HOME/server/default/deploy/nuxeo.ear/config/default-repository-config.xml`.

Example 32.1. Default repository configuration

```
<?xml version="1.0"?>
<component name="default-repository-config">
  <documentation>
    Defines the default JackRabbit repository used for development and testing.
  </documentation>
  <extension target="org.nuxeo.ecm.core.repository.RepositoryService"
    point="repository">
    <documentation>
      Declare a JackRabbit repository to be used for development and tests. The
      extension content is the Jackrabbit XML configuration of the repository.
    </documentation>
    <repository name="default"
      factory="org.nuxeo.ecm.core.repository.jcr.JCRRepositoryFactory"
      securityManager="org.nuxeo.ecm.core.repository.jcr.JCRSecurityManager"
      forceReloadTypes="false">
      <Repository>
        <!--
          virtual file system where the repository stores global state
          (e.g. registered namespaces, custom node types, etc.)
        -->
        <FileSystem class="org.apache.jackrabbit.core.fs.local.LocalFileSystem">
          <param name="path" value="${rep.home}/repository" />
        </FileSystem>
        <!--
          security configuration
        -->
        <Security appName="Jackrabbit">
          <!--
            access manager:
            class: FQN of class implementing the AccessManager interface
          -->
          <AccessManager class="org.apache.jackrabbit.core.security.SimpleAccessManager">
            <!-- <param name="config" value="${rep.home}/access.xml"/> -->
          </AccessManager>
          <LoginModule class="org.apache.jackrabbit.core.security.SimpleLoginModule">
            <!-- anonymous user name ('anonymous' is the default value) -->
            <param name="anonymousId" value="anonymous" />
            <!--
              default user name to be used instead of the anonymous user
              when no login credentials are provided (unset by default)
            -->
            <!-- <param name="defaultUserId" value="superuser"/> -->
          </LoginModule>
        </Security>

        <!--
          location of workspaces root directory and name of default workspace
        -->
        <Workspaces rootPath="${rep.home}/workspaces" defaultWorkspace="default" />
        <!--
          workspace configuration template:
          used to create the initial workspace if there's no workspace yet
        -->
        <Workspace name="${wsp.name}">
          <!--
            virtual file system of the workspace:
            class: FQN of class implementing the FileSystem interface
          -->
          <FileSystem class="org.apache.jackrabbit.core.fs.local.LocalFileSystem">
            <param name="path" value="${wsp.home}" />
          </FileSystem>

          <!--
            persistence manager of the workspace:
            class: FQN of class implementing the PersistenceManager interface
          -->
          <PersistenceManager class="org.apache.jackrabbit.core.state.obj.ObjectPersistenceManager">
          </PersistenceManager>

          <!--
            Search index and the file system it uses.
            class: FQN of class implementing the QueryHandler interface
          -->
```

```
            <SearchIndex class="org.apache.jackrabbit.core.query.lucene.SearchIndex">
              <param name="path" value="${wsp.home}/index" />
            </SearchIndex>
          </Workspace>

          <!--
            Configures the versioning
          -->
          <Versioning rootPath="${rep.home}/version">
            <!--
              Configures the filesystem to use for versioning for the respective
              persistence manager
            -->
            <FileSystem class="org.apache.jackrabbit.core.fs.local.LocalFileSystem">
              <param name="path" value="${rep.home}/version" />
            </FileSystem>

            <!--
              Configures the persistence manager to be used for persisting version state.
              Please note that the current versioning implementation is based on
              a 'normal' persistence manager, but this could change in future
              implementations.
            -->
            <PersistenceManager class="org.apache.jackrabbit.core.state.obj.ObjectPersistenceManager">
            </PersistenceManager>
          </Versioning>

          <!--
            Search index for content that is shared repository wide
            (/jcr:system tree, contains mainly versions)
          -->
          <SearchIndex class="org.apache.jackrabbit.core.query.lucene.SearchIndex">
            <param name="path" value="${rep.home}/repository/index" />
          </SearchIndex>
        </Repository>
      </repository>
    </extension>
</component>
```

Change the two PersistenceManager sections defining various database connection settings.

Refer to the Jackrabbit documentation for more information, and to the Jackrabbit Javadoc for details about configuring the PersistenceManager (here a `SimpleDbPersistenceManager`).

In particular, decide if you want the binary blobs stored inside the database or in the filesystem (change `externalBLOBs` to `true` if you want them outside the database, in the filesystem).

Using externalized Blobs can provide a performance improvment in particular if you need to store a lot of big files. Depending on the RDBMS used, there may also be a max size limit for blob if you store them in the RDBMS (for PostgeSQL 8.2 blobs are limited to 1 GB).

Here are some examples:

Example 32.2. Sample configuration for a PostgreSQL Jackrabbit repository

```
          <!-- Workspaces configuration. Nuxeo only uses the default workspace. -->
          (...)
            <PersistenceManager class="org.apache.jackrabbit.core.state.db.SimpleDbPersistenceManager">
              <param name="driver" value="org.postgresql.Driver"/>
              <param name="url" value="jdbc:postgresql://localhost/nuxeo"/>
              <param name="user" value="postgres"/>
              <param name="password" value="password"/>
              <param name="schema" value="postgresql"/>
              <param name="schemaObjectPrefix" value="jcr_${wsp.name}_"/>
              <param name="externalBLOBs" value="false"/>
            </PersistenceManager>
          (...)
          <!-- Versioning configuration. -->
            (...)
            <PersistenceManager class="org.apache.jackrabbit.core.state.db.SimpleDbPersistenceManager">
              <param name="driver" value="org.postgresql.Driver"/>
              <param name="url" value="jdbc:postgresql://localhost/nuxeo"/>
              <param name="user" value="postgres"/>
              <param name="password" value="password"/>
              <param name="schema" value="postgresql"/>
              <param name="schemaObjectPrefix" value="jcr_ver_"/>
              <param name="externalBLOBs" value="false"/>
            </PersistenceManager>
```

```
```

Example 32.3. Sample configuration for a MySQL Jackrabbit repository

```
        <!-- Workspaces configuration. Nuxeo only uses the default workspace. -->
        (...)
          <PersistenceManager class="org.apache.jackrabbit.core.state.db.SimpleDbPersistenceManager">
            <param name="driver" value="com.mysql.jdbc.Driver"/>
            <param name="url" value="jdbc:mysql://localhost/nuxeo"/>
            <param name="user" value="mysql"/>
            <param name="password" value="password"/>
            <param name="schema" value="mysql"/>
            <param name="schemaObjectPrefix" value="jcr_${wsp.name}_"/>
            <param name="externalBLOBs" value="true"/>
          </PersistenceManager>
        (...)
        <!-- Versioning configuration. -->
          (...)
          <PersistenceManager class="org.apache.jackrabbit.core.state.db.SimpleDbPersistenceManager">
            <param name="driver" value="com.mysql.jdbc.Driver"/>
            <param name="url" value="jdbc:mysql://localhost/nuxeo"/>
            <param name="user" value="mysql"/>
            <param name="password" value="password"/>
            <param name="schema" value="mysql"/>
            <param name="schemaObjectPrefix" value="jcr_ver_"/>
            <param name="externalBLOBs" value="true"/>
          </PersistenceManager>
```

For JackRabbit, there are some persistance manager are specific to a RDBMS :

- for PostgreSQL: you may use org.apache.jackrabbit.core.persistence.bundle.PostgreSQLPersistenceManager
- for MySQL: you may use org.apache.jackrabbit.core.persistence.bundle.MySqlPersistenceManager
- for Oracle 10: you may use org.apache.jackrabbit.core.persistence.bundle.OraclePersistenceManager
- for MSSQL2005 you mau use : org.apache.jackrabbit.core.persistence.bundle.MSSqlPersistenceManager

32.2.3.2. Set up your RDBMS

Create the database in the database server, enable IP connection, setup permissions and test the connection.

32.2.3.3. Start Nuxeo EP

You can now start JBoss AS and verify that your new repository is used!

32.2.4. Configure Storage for other Nuxeo Services

Many services beyond Nuxeo Core Repository are using an SQL database to persist their data, such as:

- Relations Service: RDF data is stored in SQL,
- Audit Service: Audit logs are stored via JPA,
- Directories: entries can be stored into an SQL database.

By default, all these services use the JBoss AS's embedded HSQLDB.

32.2.4.1. Configuring datasources

Each service use a dedicated datasource to define the database connection. In most case you can simply do the following:

- deploy the needed JDBC Driver in `$JBOSS_HOME/server/default/lib`,
- modify the datasource definitions files in `$JBOSS_HOME/server/default/deploy/nuxeo.ear/datasources`.

For example, if you would like to store audit logs in PostgreSQL you can deploy the PostgreSQL JDBC driver. To do that, edit `nxaudit-logs-ds.xml` as in the following example:

Example 32.4. Datasource for the Audit Service using PostgreSQL

```
<?xml version="1.0"?>
<datasources>
  <local-tx-datasource>
    <jndi-name>nxaudit-logs</jndi-name>
    <connection-url>jdbc:postgresql://localhost/logs</connection-url>
    <driver-class>org.postgresql.Driver</driver-class>
    <user-name>username</user-name>
    <password>password</password>
  </local-tx-datasource>
</datasources>
```

We recommend to enable XA transactions if your database server support it (for PostgreSQL, you have to use 8.x versions). The following datasource definition example enables XA transaction for the Audit Service using PostgreSQL.

Example 32.5. Datasource for the Audit Service using PostgreSQL with XA transactions

```
<?xml version="1.0"?>
<datasources>
   <xa-datasource>
     <jndi-name>nxaudit-logs</jndi-name>
     <xa-datasource-class>org.postgresql.xa.PGXADataSource</xa-datasource-class>
     <xa-datasource-property name="ServerName">localhost</xa-datasource-property>
     <xa-datasource-property name="PortNumber">5432</xa-datasource-property>
     <xa-datasource-property name="DatabaseName">logs</xa-datasource-property>
     <xa-datasource-property name="User">postgres</xa-datasource-property>
     <xa-datasource-property name="Password">password</xa-datasource-property>
     <track-connection-by-tx/>
   </xa-datasource>
</datasources>
```

See Datasources Configuration on the JBoss wiki for more examples of datasources.

This method works for most services:

- Audit: `nxaudit-logs-ds.xml`
- Placeful Configuration Service & Subscriptions: `nxplaceful-ds.xml`
- UID generator: `nxuidsequencer-ds.xml`
- jBPM engine: `nxworkflow-jbpm-ds.xml`
- Workflow Document Service: `nxworkflow-documents-ds.xml`
- Archive management: `nxarchive-records-ds.xml`
- Relations: `nxrelations-default-jena-ds.xml`

- Compass search engine: `nxsearch-compass-ds.xml`

32.2.4.1.1. Special MySQL needs

MySQL is a quirky database which sometimes needs very specific options to function properly.

The datasources use by Jena (`nxrelations-default-jena-ds.xml` and `nxcomment-jena-ds.xml`) need to use a "relax autocommit" mode. To enable that, change the `connection-url` in the datasources to something like:

```
<connection-url>
  jdbc:mysql://localhost/nuxeo?relaxAutoCommit=true
</connection-url>
```

The datasource use by Compass (`nxsearch-compass-ds.xml`) needs to be put in "relax autocommit" too, and in addition it needs an "emulate locators" option:

```
<connection-url>
  jdbc:mysql://localhost/nuxeo?relaxAutoCommit=true&amp;emulateLocators=true
</connection-url>
```

This is documented at http://static.compassframework.org/docs/latest/jdbcdirectory.html .

Note the `&amp;` syntax for the URL parameters instead of just `&` because the URL is embedded in an XML file.

32.2.4.2. Relation service configuration

The Relation Service uses a datasource to define the data storage. However modifying the datasource is not enough, you also have to tell to the Jena engine which database dialect is used, as it doesn't auto-detect it.

To do that, edit the `sql.properties` file in `$JBOSS_HOME/server/default/deploy/nuxeo.ear/config/` and change the definition of the `org.nuxeo.ecm.sql.jena.databaseType` property. The possible values are:

- `Derby`
- `HSQL`
- `MsSQL`
- `MySQL`
- `Oracle`
- `PostgreSQL`

In the same file, the property `org.nuxeo.ecm.sql.jena.databaseTransactionEnabled` is `false` by default, but must be set to `true` for Oracle.

Please refer to the Jena Site for more information about database support.

The value of the above properties are used as variables by the extension point defining the Jena configuration, so that they only have to be changed in one place.

32.2.4.3. Compass search engine dialect configuration

The Compass plugin is configured using a datasource, but at the time of this writing it still needs some additional configuration in a file embedded in its Jar. You should go to `$JBOSS_HOME/server/default/deploy/nuxeo.ear/platform/` and inside the directory `nuxeo-platform-search-compass-plugin-5.1-SNAPSHOT.jar` (the version number may be different) then edit the file `compass.cfg.xml`. You will find a section like:

```
<connection>
  <jdbc managed="true"
    dialectClass="org.apache.lucene.store.jdbc.dialect.HSQLDialect"
    deleteMarkDeletedDelta="3600000">
    <dataSourceProvider>
      <jndi lookup="java:/nxsearch-compass" />
    </dataSourceProvider>
  </jdbc>
</connection>
```

The `dialectClass` has to be changed according to your datasource configuration. The possible values end with `MySQLDialect`, `PostgreSQLDialect` , etc. They are documented in the Compass documention about SQL dialects .

32.2.5. Setting up a new repository configuration

If you just want to change the `default` repository name appearing in the url `http://.../nuxeo/nxdoc/default/`, modify the repository name value in:

- `default-repository-config.xml`
- `platform-config.xml`

TODO: Nuxeo configuration has changed, the two above sections need to be updated.

32.2.5.1. Add the new repository configuration

Create a repository definition as a contribution to the extension point `org.nuxeo.ecm.core.repository.RepositoryService` (for example: `MyRepo-repositoy-config.xml`) in `$JBOSS_HOME/server/default/deploy/nuxeo.ear/config/`.

You can take example on the default repository definition $JBOSS_HOME/server/default/deploy/nuxeo.ear/config/default-repository-config.xml.

You have to properly configure the following aspects:

- the name of the component (`name="org.nuxeo.project.sample.repository.MyRepo"`), which must be unique among component names,
- the name of the repository (`<repository name="MyRepo">`), which is used to refer to it from your application and must also be unique among repository names,
- the various database connection settings (driver, user, password, schema, etc.),
- decide if you want the binary blobs stored inside the database or in the filesystem (change `externalBLOBs` to `true` if you want them outside the database).

Refer to the Jackrabbit documentation for more information, and to the Jackrabbit Javadoc for details about configuring the `SimpleDbPersistenceManager`.

Example 32.6. Sample configuration for a PostgreSQL Jackrabbit repository

```
<?xml version="1.0"?>
<component name="org.nuxeo.project.sample.repository.MyRepo">
  <extension target="org.nuxeo.ecm.core.repository.RepositoryService" point="repository">
    <repository name="MyRepo"
                factory="org.nuxeo.ecm.core.repository.jcr.JCRRepositoryFactory"
                securityManager="org.nuxeo.ecm.core.repository.jcr.JCRSecurityManager"
                forceReloadTypes="false">
      <Repository>
        <FileSystem class="org.apache.jackrabbit.core.fs.local.LocalFileSystem">
          <param name="path" value="${rep.home}/repository"/>
        </FileSystem>
        <Security appName="Jackrabbit">
          <AccessManager class="org.apache.jackrabbit.core.security.SimpleAccessManager">
        </AccessManager>
          <LoginModule class="org.apache.jackrabbit.core.security.SimpleLoginModule">
            <param name="anonymousId" value="anonymous"/>
          </LoginModule>
        </Security>

        <!-- Workspaces configuration. Nuxeo only uses the default workspace. -->
        <Workspaces rootPath="${rep.home}/workspaces" defaultWorkspace="default"/>
        <Workspace name="${wsp.name}">
          <FileSystem class="org.apache.jackrabbit.core.fs.local.LocalFileSystem">
            <param name="path" value="${wsp.home}"/>
          </FileSystem>
          <PersistenceManager class="org.apache.jackrabbit.core.state.db.SimpleDbPersistenceManager">
            <param name="driver" value="org.postgresql.Driver"/>
            <param name="url" value="jdbc:postgresql://localhost/nuxeo"/>
            <param name="user" value="postgres"/>
            <param name="password" value="password"/>
            <param name="schema" value="postgresql"/>
            <param name="schemaObjectPrefix" value="jcr_${wsp.name}_"/>
            <param name="externalBLOBs" value="false"/>
          </PersistenceManager>
          <SearchIndex class="org.apache.jackrabbit.core.query.lucene.SearchIndex">
            <param name="path" value="${wsp.home}/index"/>
          </SearchIndex>
        </Workspace>

        <!-- Versioning configuration. -->
        <Versioning rootPath="${rep.home}/version">
          <FileSystem class="org.apache.jackrabbit.core.fs.local.LocalFileSystem">
            <param name="path" value="${rep.home}/version"/>
          </FileSystem>
          <PersistenceManager class="org.apache.jackrabbit.core.state.db.SimpleDbPersistenceManager">
            <param name="driver" value="org.postgresql.Driver"/>
            <param name="url" value="jdbc:postgresql://localhost/nuxeo"/>
            <param name="user" value="postgres"/>
            <param name="password" value="password"/>
            <param name="schema" value="postgresql"/>
            <param name="schemaObjectPrefix" value="jcr_ver_"/>
            <param name="externalBLOBs" value="false"/>
          </PersistenceManager>
        </Versioning>

        <!-- Index for repository-wide information, mainly versions. -->
        <SearchIndex class="org.apache.jackrabbit.core.query.lucene.SearchIndex">
          <param name="path" value="${rep.home}/repository/index"/>
        </SearchIndex>
      </Repository>
    </repository>
  </extension>
</component>
```

32.2.5.2. Declare the new repository to the platform

TODO: this should be moved to a different section as it doesn't pertain to the SQL configuration itself.

You have now replaced the default repository (`demo`) with your newly defined one (`MyRepo`). To actually use it, create or edit the file `MyPlatform-Layout-config.xml` in `$JBOSS_HOME/server/default/deploy/nuxeo.ear/config/` and configure the parameters as shown in the following example.

```
<?xml version="1.0"?>
<component name="MyPlatformLayout">

  <require>org.nuxeo.ecm.platform.api.DefaultPlatform</require>

  <extension target="org.nuxeo.ecm.platform.util.LocationManagerService" point="location">
    <locationManagerPlugin> <!-- This disable the default (demo) repository -->
      <locationEnabled>false</locationEnabled>
      <locationName>demo</locationName>
    </locationManagerPlugin>
    <locationManagerPlugin> <!-- This enable your new repository -->
      <locationEnabled>true</locationEnabled>
      <locationName>MyRepo</locationName> <!-- Use the name of your repository -->
    </locationManagerPlugin>
  </extension>

 <extension target="org.nuxeo.ecm.core.api.repository.RepositoryManager"
    point="repositories">
    <documentation>The default repository</documentation>
    <repository name="MyRepo" label="My Repository"/>
  </extension>

  <extension target="org.nuxeo.runtime.api.ServiceManagement" point="services">
      <service class="org.nuxeo.ecm.core.api.CoreSession" name="MyRepo" group="core">
```

```
          <locator>%DocumentManagerBean</locator>
      </service>
  </extension>

</component>
```

This sample configuration creates a new repository in the `core` Group that will be assigned to the default server. If you want to have it located on an other server you can use:

```
<extension target="org.nuxeo.runtime.api.ServiceManagement" point="servers">
  <!-- define new locator for group MyGroup -->
  <server class="org.nuxeo.runtime.api.JBossServiceLocator">
    <group>MyGroup</group>
    <property name="java.naming.factory.initial">org.jnp.interfaces.NamingContextFactory</property>
    <property name="java.naming.provider.url">jnp://MyServer:1099</property>
    <property name="java.naming.factory.url.pkgs">org.jboss.naming:org.jnp.interfaces</property>
  </server>

  <!-- bind MyRepo to MyGroup -->
  <extension target="org.nuxeo.runtime.api.ServiceManagement" point="services">
    <service class="org.nuxeo.ecm.core.api.CoreSession" name="MyRepo" group="MyGroup">
      <locator>%DocumentManagerBean</locator>
    </service>
  </extension>

</extension>
```

32.3. LDAP Integration

32.3.1. For users/groups storage backend

The user interface in Nuxeo EP gets the data from NXDirectory. As a consequence you can choose your source. By default, the users/groups data is stored in a SQL database. If you want to get the users from a LDAP directory, you need to deploy one of the following configuration:

- Users in LDAP, groups in SQL

  Go to the `examples` sub-folder and copy the `default-ldap-users-directory-bundle.xml` file in the `nuxeo.ear/config` folder of the JBoss instance (or bundle it in a jar, cf packaging in this guide). This sample setup replaces the default userDirectory configuration SQL with users fetched from the LDAP server. The groupDirectory remains unaffected by this setup. You might want to copy the file `default-virtual-groups-bundle.xml` and adjust defaultAdministratorId to select a user from your LDAP that have administrative rights by default. You can also configure the section on defaultGroup to make all users members of some default group (typically the members group) so that they have default right without having to make them belong to groups explicitly.

- Users and groups in LDAP

  Copy the users setup as previously; moreover copy the `default-ldap-groups-directory-bundle.xml` file in the `nuxeo.ear/config` folder of the JBoss instance. This sample setup which is dependent on the previous one additionally overrides the default `groupDirectory` setup to read the groups from the LDAP directory typically from `groupOfUniqueNames` entries with fully qualified `dn` references to the user entries or to subgroups. You can edit the `nuxeo.ear/config/*.xml` files on the JBoss instance, but you will need to restart JBoss to take changes into account.

32.4. OpenOffice.org server installation

OpenOffice.org (OOo) is used server-side for different tasks such as file transforms (eg. to PDF) or other advanced tasks. It has to run in listen mode and some tools has been developed to ease the settings. They are available at the Nuxeo svn tool section.

For OOo versions lower than 2.3, OOo has to run under a graphical interface. If the server that hosts OOo is Linux server that has no X installed, then the X virtual frame buffer tool `Xvfb` (or `xvfb-run` depending of your distribution) can be used to create a suitable display.

```
Xvfb :77 -auth Xperm -screen 0 1024x768x24
```

Since version 2.3.0, no display is required anymore so OpenOffice.org behaves as a real server. No more need of the above `Xvfb` and `export DISPLAY` commands.

The `UNO` protocol implies that OOo is opened in listen mode on a network interface. In our environment, the `8100 port` is used. The interface to be used is the one that OOo will listen to. We use here `localhost` which imply that OOo will be called by the machine hosting it. On a local network, we will have to use the IP adress from when the connection will arrive.

This listening can be done by adding some arguments to the command line launching OOo or by adding this connection info in OOo `Setup.xcu` configuration file, so that it is automatic each time it is launched.

```
<node oor:name="Office">
  <prop oor:name="ooSetupConnectionURL">
    <value>socket,host=localhost,port=8100;urp;StarOffice.Service</value>
  </prop>
</node>
```

This OOo registry modification has been packaged as an extension that eases the deployment:

```
<OOoInstallatioPath>/program/unopkg add nxOOoAutoListen.oxt
```

Then OOo can be launched:

```
export DISPLAY=":77.0"
/opt/openoffice.org2.2/program/soffice -headless -nofirststartwizard
```

The `-headless` switch lets OOo manage to answer every question that may occur with a default response avoiding dead-locks. It also hides the OOo windows if a display is available by default (eg. Ms Windows). Note that the `-invisible` switch is not used as it is redundant and errors on PDF exports may occur. The `-nofirststartwizard` skips any post-install wizard, allowing to use OOo directly after the installation without any user parametrization. If the nxOOoAutoListen extension has not been installed, the following switch will let OOo listen on the correct interface:port `"-accept=socket,host=localhost,port=8100;urp;StarOffice.Service"`.

Note that the platform used for both the JVM and OOo have to be consistent to allow UNO protocol to work: with a 32 bits JVM, you'll have to use the 32 bits OOo version while the 64 bits one will be mandatory for a 64 bits JVM.

32.5. Run Nuxeo EP with a specific IP binding

To be able to call the Nuxeo Services remotely using the Nuxeo Framework (e.g. when using Nuxeo RCP), you will need to bind an IP address when running the server. To do this, a few step are needed:

- in `$JBOSS_HOME/server/default/deploy/nuxeo.ear/config/nuxeo.properties` change the value of `org.nuxeo.ecm.instance.host`, remplace "localhost" by the IP adress of the JBoss server
- in `$JBOSS_HOME/server/default/deploy/nuxeo.ear/config/mbeans/core-events-client-service.xml` for `java.naming.provider.url` replace the 127.0.0.1 value by the IP adress of the Jboss server

- start jboss with the -b option:

  ```
  ./run.sh -b IP_adress_of_the_server
  ```

32.6. Virtual Hosting

The Nuxeo webapp can be virtual hosted behind a HTTP/HTTPS reverse proxy, like Apache.

32.6.1. Motivations for virtual hosting

Virtual hosting provides several advantages:

- Support for HTTPS

  HTTPS support in Apache is easy and flexible to setup.

  Apache can also be used to handle certificate authentication.

- URL filtering

  You can use Apache filtering tools to limit the URLs that can be accessed via the reverse proxy.

- Handle HTTP cache for static resources

  Nuxeo 5 generates standard HTTP cache headers for all static resources (images, JavaScript...).

  These resources are by default cached on the client side (in the browser cache). For performance reasons, it can be useful to host these resources in the reverse proxy cache.

32.6.2. Virtual hosting configuration for Apache 2.x

32.6.2.1. Reverse proxy with mod_proxy

For this configuration, you will need to load and enable `mod_proxy` and `mod_proxy_http` modules.

```
ProxyPass /nuxeo/ http://Nuxeo5ServerInternalIp:8080/nuxeo/
ProxyPassReverse /nuxeo/ http://Nuxeo5ServerInternalIp:8080/nuxeo/
```

You can also use rewrite rules to achieve the same result:

```
ProxyVia On
ProxyRequests Off
RewriteEngine On
RewriteRule /nuxeo(.*) http://Nuxeo5ServerInternalIp:8080/nuxeo$1 [P,L]
```

This configuration will allow you to access the Nuxeo EP webapp via `http://ApacheServer/nuxeo/`.

The Nuxeo webapp will generate the URLs after reading the http header `x-forwarded-host`.

Unfortunately, this header does not specify the protocol used, so if your Apache is responding to HTTPS, you will need to send Nuxeo EP a specific header to indicate the base URL that the webapp must use when generating links.

```
RequestHeader append nuxeo-virtual-host "https://myDomainName/"
ProxyPass /nuxeo/ http://Nuxeo5ServerInternalIp:8080/nuxeo/
ProxyPassReverse /nuxeo/ http://Nuxeo5ServerInternalIp:8080/nuxeo/
```

This will require you to load and activate `mod_header` module.

32.6.2.2. Reverse proxy with mod_jk

`mod_jk` allows you to communicate between Apache and Tomcat via the ajp1.3 protocol.

```
JkWorkersFile /etc/apache2/jk/workers.properties
JkLogFile     /var/log/mod_jk.log
JkLogLevel    info
JkMount /nuxeo ajp13
JkMount /nuxeo/* ajp13
```

The `workers.properties` file will contain the list of Nuxeo EP Tomcat servers. The AJP13 tomcat listener should be enabled by default on port 8009.

```
worker.list=ajp13
worker.ajp13.port=8009
worker.ajp13.host=Nuxeo5ServerInternalIp
worker.ajp13.type=ajp13
worker.ajp13.socket_keepalive=1
worker.ajp13.connection_pool_size=50
```

Once again, if you use HTTPS, you will need to set the Nuxeo-specific header to tell the webapp how to generate URLs:

```
RequestHeader append nuxeo-virtual-host "https://myDomainName/"
```

This will require you to load and activate the `mod_header` module.

32.6.2.3. Configuring http cache

The Nuxeo webapp includes a Servlet Filter that will automatically add header cache to some resources returned by the server.

By using the `deployment-fragement.xml` you can also put some specific resources behind this filter :

```
<extension target="web#FILTER">
  <filter-mapping>
    <filter-name>simpleCacheFilter</filter-name>
    <url-pattern>/MyURLsToCache/*</url-pattern>
  </filter-mapping>
</extension>
```

When Nuxeo EP is virtual hosted with apache you can use `mod_cache` to use the reverse-proxy as cache server.

You can also use Squid or any other caching system that is compliant with the standard HTTP caching policy.

32.7. The Nuxeo Shell

32.7.1. Overview

The Nuxeo Shell is a command line tool for everyone who needs simple and quick remote access to a Nuxeo EP server. You can see it as the swiss army knife for the Nuxeo EP world. It can be used by regular users to browse or modify content, by advanced users to administrate nuxeo servers, or by programmers to test and debug.

The Nuxeo Shell is designed as an intuitive and extensible command line application. It can both serve as a user, administrator or programmer tool, or as a framework to develop new Nuxeo command line clients.

The application is based on the Nuxeo Runtime framework and thus offers the same extension point mechanism you can find on the server application.

The main features of the command line tool are:

- Two operating modes: an interactive and a batch mode.
- Advanced command line editing like:
  - auto-completion
  - command history
  - command line colors
  - command line shortcuts like: CTRL+K, CTR+A, etc.
- JSR223 scripting integration. You can thus connect and execute commands on a Nuxeo server in pure scripting mode.
- Extensibility - using Nuxeo Runtime extension points

The advanced command line handling is done using the JLine library.

The only requirement is Java 5+. The tool works on any Unix-like OS, on Windows and on Mac OS X.

32.7.2. User Manual

The Nuxeo Shell application is packaged as a zip archive. To install it, you need to unzip and copy the content in a folder.

The application folder structure is as follow:

```
+ nxshell
  + app
    + bundles
    + config
    + data
    + lib
  + lib
  - Launcher.class
  - log4j.properties
  - launcher.properties
  - nxshell.sh
```

- The `lib` folder contains JARs needed by the application launcher. The `Launcher.class` is the application launcher and `launcher.properties` contains configuration for the launcher.
- `log4j.properties` is as you expect the configuration for log4j.
- `nxshell.sh` is a shell script that launches the application.
- The `app` folder contains the application code and data.
  - `app/bundles` contains the application OSGi bundles. These bundles will be deployed using a minimal implementation of OSGi (the same that is used on the server side). We may replace the default implementation by equinox later.
  - `app/lib` contains third party libraries needed by the application bundles.
  - `app/config` contains the application configuration files.
  - `app/data` contains temporary data.

The only file you may need to change is the `nxshell.sh` script. Here is the content of this file:

```
#!/bin/bash
```

```
#JAVA_OPTS="$JAVA_OPTS -Xdebug -Xrunjdwp:transport=dt_socket,address=127.0.0.1:8788,server=y,suspend=y"
JAVA_OPTS="$JAVA_OPTS -Dorg.nuxeo.runtime.1.3.3.streaming.port=3233"
java $JAVA_OPTS Launcher launcher.properties $@
```

If you uncomment the first line, you will be able to run Nuxeo Shell in debug mode.

The second line [must] be commented out if on the server you are running on a nuxeo runtime 1.4.SNAPSHOT. When running against Nuxeo EP 5.1 you need to let this uncommented.

You can run the application in two modes:

1. In batch mode - in this mode you need to specify the command that will be executed. After the command excecution the process will exit. Here is an example of a command executed in batch mode:

```
./nxshell.sh export /path/to/remote/doc /path/to/local/folder
```

2. In interactive mode - in this mode you can share a single session to run multiple commands against the remote Nuxeo EP. To start the application in that mode you should run the "interactive" command as follows:

```
./nxshell.sh interactive
```

   After entering in interactive mode a command prompt will be displayed. At this prompt you can enter commands in the same way you specify them on the command line in batch mode.

   When not connected to a server, the prompt will be displayed as:

```
|>
```

   After connecting to a server named, let's say "nuxeo-platform", the prompt will be displayed as:

```
|nuxeo-platform>
```

So, as we've seen in the above examples, executing a command is done by using the the following syntax:

```
command [options] [parameters]
```

where options and parameters are optional and are specific to the command you run.

Example:

```
import -u /path/to/doc /path/to/local_file
```

In this case "import" is the command, "-u" is a flag option and "path/to/doc" and "/path/to/local_file" are both parameters

Parameters are stand alone arguments (they are not bound to a specific option) and can be retrieved programatically using their index. In the example above, the first parameter will have the index 0 while the second the index 1.

32.7.2.1. Command Options

Command options are defined by a name and optionaly a shortcut (a short name). When reffering to an option using it's name you should prefix it by two hyphens '--'. When using short names you should only use one hyphen as a prefix. For example if you have an option named "host" and using a short name of "h" the following commands are equivalent:

```
./nxshell.sh interactive -h locahost

./nxshell.sh interactive --host locahost
```

Options may take values or may be used as flags turning on / off a specific option by their simple presence.

When using long names you should specify the option values imediately after the option. However when using short names you can group options together. Let say for example we have a command that support 4 options: a, v, i, o. a and v are flag options and both i and o takes an argument as value. In this case the you can group options like the following:

```
command -avi in_file -o out_file
```

or

```
command -avio in_file out_file
```

or anyhow you want. You should keep in mind that when grouping options that take arguments, these arguments will be assigned to each of this options in the order they were specified on the command line.

32.7.2.1.1. Global Options

Besides the specific options that commands may define, there are several global options that apply to all commands. These are:

- host (**--host** or **-h**) the Nuxeo EP host where to connect - defaults to localhost
- port (**--port** or **-p**) the Nuxeo EP port where to connect - defaults to 62474
- username (**--username** or **-u**) the username to use - defaults to the "system" user
- password (**--password** or **-P**) the password to use

32.7.2.2. Commands

There is the list of all built-in commands of nuxeo shell.

[Notes:]

1. Many of these built-in commands are not yet implemented at the timewriting this document. I will mark them using *.
2. The commands can be grouped in two categories:
   - offline commands - command that doesn't need a connection to work. Example: help, log etc.
   - online commands - commands that requires a connection to work. These commands are automatically connecting if no connection is currently established.
3. Some commands make no sense and are not available in both modes (batch and interactive). This will be specified by each command if it is the case.

32.7.2.2.1. interactive

Runs in the interactive mode. This command is not available when already in interactive mode.

Has no specific options.

```
./nxshell.sh interactive
```

32.7.2.2.2. help

Displays the help page.

Takes an optional parameter which is the name of a command.

By default, displays the global help page. If a command is given as an argument, displays the help page for that command.

```
./nxshell.sh help ls
```

32.7.2.2.3. log *

Displays the client log. Available only in *interactive* mode

32.7.2.2.4. connect *

Connects to a given server. If a connection is already established, close it first.

Available only in *interactive* mode

32.7.2.2.5. disconnect *

Disconnects from the current connected server. If no connection exists, does nothing.

Available only in *interactive* mode

32.7.2.2.6. ls

Lists the children of the current folder in the repository.

By default, Folderish types are displayed in blue.

Available only in *interactive* mode

32.7.2.2.7. cd

Changes the current directory in the repository (to a Folderish document)

32.7.2.2.8. pwd

Displays the current path in the repository.

32.7.2.2.9. view

Views info about document. The information displayed can be controlled using these command options:

--all (-a) - view all data

--system (-s) - view only the system data

--acp - view the ACLs on that docuemnt

32.7.2.2.10. rm

Removes a document or a tree of documents.

32.7.2.2.11. mkdir

Creates a Folder document.

32.7.2.2.12. put

Uploads a blob to a file document. If the target document doesn't exists, creates it.

32.7.2.2.13. putx

Creates a document other than a file. Metadata (and blobs) are specified in the Nuxeo export format.

32.7.2.2.14. get

Downloads the blob from a File document.

32.7.2.2.15. getx

Gets a document as an XML file (as export, but only for a document).

32.7.2.2.16. export

Exports documents from the repository.

32.7.2.2.17. import

Imports documents into the repository.

32.7.2.2.18. chperm *

Changes a privilege on the given document.

32.7.2.2.19. adduser *

Creates a new user.

32.7.2.2.20. addgroup *

Creates a new group.

32.7.2.2.21. find *

Search the repository using the NXQL query language.

32.7.2.2.22. lstypes *

32.7.2.2.23. viewtype *

32.7.2.2.24. lsusers *

32.7.2.2.25. lsgroups *

32.7.2.2.26. viewuser *

32.7.2.2.27. viewgroup *

32.7.2.3. Examples

32.7.3. Extending the shell

If you a need a functionality not provided by the Nuxeo Shell, you can simply add it by writing a Java class and a XML file to describe your extensions. By using declarative extensions, you can control things like auto-completion and help pages for your own commands.

32.7.3.1. Registering Custom Commands

32.7.3.1.1. Defining help pages

32.7.3.1.2. Controlling auto-completion

Part VI. Annexes

Appendix A. Frequently Asked Questions

A.1. Deployment and Operations

Nuxeo supports and certifies the following hardware and software:

Hardware:

- Intel/AMD 32-bit & 64-bit
- SPARC 32-bit & 64-bit

Operating Systems:

- RedHat 3.x, 4.x, 5.x
- Debian 4.0, Ubuntu Server 6.06 LTS and 7.04
- Solaris 10
- Windows Server 2003
- MacOS X 10.4.x

RDBMS:

- PostgreSQL 8.x
- MySQL 5.x
- Oracle Database 9i, Oracle Database 10g

Java Runtime Environment (JRE):

- Java 5 aka 1.5.0 (update 11 recommended)
- Java 6 aka 1.6.0 (update 11 recommended)

Java EE application servers:

- JBoss AS 4.0.4 GA and 4.0.5 GA
- JBoss AS 4.2.0 GA (in progress)
- Glassfish v2 (in progress)
- BEA WebLogic 10 (in progress)

The most used configuration is JBoss AS 4.0.4 GA using JRE 1.5.0_11 on RedHat AS 4.x running on Intel x86 hardware.

Intel-based hardware (bi-Dual Core, Quad Core or bi-Quad Core), 4GB of RAM. The required disk space only depends on the data volume to store (raw requirements to be secure: size of files to manage * 2).

Each release (major and minor) is delivered with an upgrade procedure, new features list, improvements list, fixed bugs list and known bugs/limitations list. Moreover the issue tracker is public (it allows everybody to see the status of the software, known/ongoing bugs and issues, features/improvement roadmap, etc.).

An installation guide is available. Upgrade procedures are delivered along each release.

An administration guide is available and updated with each release.

A User Guide is delivered with each release.

A developer guide is delivered and constantly improved.

The reference manual assembles all the documentation available for users, developers, operation teams, etc.

Nuxeo provides several other documents/resources such as the API (Javadoc), some tutorials, specific *Archetypes* for Maven 2 (useful to quickly bootstrap new plugins / projects), etc.

The documentation is available in the English language. Translation to French, Spanish, German and Italian are supported/provided by the community (if you require a specific language, you can order it from Nuxeo).

The documentation of included software are either included in the reference documentation if it's useful for common operations, either linked if not. All the documentation of Nuxeo EP and bundled software packages is freely available.

Yearly major release and quarterly minor release. The high-level roadmap is published by Nuxeo and updated frequently. Detailed roadmap is available from the issue tracker (all details are available on each issue such as comments, status, votes, related commit in the SCM, etc.).

See installation procedure. XXX Add link.

The "Administration and Operation Guide" describes available monitoring points. In short, Nuxeo EP offers a set of JMX services to monitor all critical points of the application (standard Java EE applications monitoring system). Moreover, logs can be broadcasted using log4j capabilities (SNMP, email, etc.). Both should be usable by all major monitoring software.

Nuxeo EP is fully based on Java EE 5 and supports related clustering and HA features. JBoss Clustering is the recommended clustering and HA solution for Nuxeo EP's services. Nuxeo EP services can be configured for performance clustering and/or HA clustering (depending on the capabilities and requirements of each service).

This is possible through configuration of the application server (ex: JBoss AS / Tomcat). Nuxeo EP relies on the application server for all the network configuration.

Nuxeo EP entirely depends on the Java EE application server for all network related configuration. It is not bind in any way to the physical network configuration of the server. Hence it is possible to change the hostname of the server and restart the machine without causing any problem to Nuxeo EP.

Fail-over relies on JBoss Clustering for Nuxeo EP services. Here is the HA system used for each category of services:

- *Nuxeo Core* (Content Repository): HA clustering only. It relies on the native RDMBS replication system (Oracle RAC or PostgreSQL replication solutions). Data integrity has to be trustable and enforced.

- *Nuxeo Search* (Search Engine): HA and performance clustering. It can use a shared filesystem (if indexes are stored on the filesystem) or can rely on the RDMBS replication solution. If data integrity is corrupt, a reindexing of the content is be sufficient to restore it.

- *EJB3-based Services*: HA and performance clustering. Use native EJB3 clustering and load-balancing from JBoss Clustering. Services using data persistence rely on RDMBS replication (for HA) that needs to be trustable and enforced.

- *Web Client/App*: can use HA and performance clustering (using JBoss Clustering). Does not need data sync.

To achieve the highest level of data integrity, Nuxeo recommends storing binary files as BLOBs directly in the database (hence use a RDBMS offering optimized BLOBs storage (such as Oracle or PostgreSQL). Using this mechanism, Nuxeo EP can store all its data into the RDBMS (including request/search engine indexes) and relies on it to enforce data integrity. Moreover, Nuxeo EP is *fully* transactional and relies on JTA (+ XA) for transaction management (that enforce data integrity across data sources).

Nuxeo EP has been designed to completely rely on RDBMS data integrity (that can be considered trustable nowadays). One can use RDBMS tools to check data integrity/consistency and data failure if any. If the data model is corrupted, Nuxeo EP warns about it when the repository starts. Indexes (from Nuxeo Search) can be verified and easily be rebuilt by reindexing the content if any problem occurs.

The maximal impact is service downtime and data restoration from backups. Data integrity errors are reported in the logs and can then be sent via email notifications, SNMP and any log4j capabilities.

Nuxeo EP offers an applicative data import/export service (using XML serialization of documents) that can be used as an incremental backup/restore system. For an efficient backup system, Nuxeo recommends using native RDBMS tools (that can offer incremental backups, snapshots, hot restore, etc.).

The restoration speed is the native database write performances. We do not have more statistics yet (but should be available by July 2007, benchmark are in progress on this point).

When all datasources for storage are using the same database (the recommended setup), the RDBMS can achieve a consistent backup (usually at low cost for the user service). Restore can only be launched when the system is stopped.

Content object restore can be done using the import/export service. Here is the procedure to achieve this:

1. Get IDs of content object to restore using, for example, the audit service/log (ex: get all DocIds from "CreateObject" log entries for a particular user).
2. Get those document from a backup (done via the export service) and copy them in a directory (the standard export format use one directory per content object which is easing a lot this operation).
3. Use a command line import/export client to (re-)import document in this directory.
4. You're done.

RDBMS backup can be handled as usual using legacy backup scripts for this RDBMS. Applicative backups can be launched using the import/export client CLI. There is not supported scripts at the moment (but they could easily be written).

Appendix B. Detailed Development Software Installation Instructions

This chapter is provided to help you install the software needed to work on your Nuxeo projects.

[TODO: refactor this chapter as in many cases, the packages may be installed using some package management system (for instance on Mac OS: **port install apache-ant maven2**).]

B.1. Installing Java 5

Nuxeo EP uses the latest generation of Java technologies and thus requires a Java 5 VM such as the reference implementation by Sun.

You may already have the right Java development kit install can enter in the command line:

```
javac -version
```

. This should return something like:

```
javac "1.5.0_10"
```

If the java version is 1.5.x, you can skip the "Installing java 5" section.

Warning

Nuxeo EP doesn't currently compile under Java 6 (the 1.6.0 JDK from Sun). Even if it did, it is not clear that JBoss, the application server we are targeting, works under Java 6.

Note

Java 5 is also sometimes called Java 1.5 or 1.5.0.

B.1.1. Using the Sun Java Development Kit (Windows and linux)

Sun Microsystems provides freely downloadable version of the Java Development Kit (JDK), that is needed to compile the Nuxeo platform.

For the purpose of Nuxeo development, you should download the latest release of the JDK 5.0 (JDK 5.0 Update 11 at the time of this writing) from `http://java.sun.com/javase/downloads/index_jdk5.jsp`

B.1.2. Using a package management systems (Linux)

Some Linux distributions now include Java 5 in their official repositories.

For instance with Ubuntu (Edgy and later):

- enable "multiverse" (2 lines to uncomment) in your `/etc/apt/sources.list`
- update your package indexes:

```
$ sudo apt-get update
```

- install the full Java 5 stack (probably not all is necessary):

```
$ sudo apt-get install "sun-java5-*"
```

- ensure Java 5 is now the default JVM on your system (instead of gcj and friends by default):

```
$ sudo update-alternatives --set java /usr/lib/jvm/java-1.5.0-sun/jre/bin/java
```

(TODO: add similar instructions for Fedora Core and Debian)

B.1.3. Manual installation (Linux)

You can also manually install Java by following the instructions of this page: http://www.java.com/en/download/linux_manual.jsp

B.1.4. Setting up JAVA_HOME (Windows, Linux, Mac OS)

This will be required by tools such as Maven (see later).

B.1.4.1. Windows

Follow these instructions:

- type 'windows' +'pause'
- click on advanced tab
- click on "environment variables" (at the bottom)
- click on new and enter "`JAVA_HOME`" for variable name and "`C:\Program Files\Java\jdk1.5.0_10`" (adapt to your own JDK install)
- don't forget to click on ok to close the environment variables window.

B.1.4.2. Linux

In your `.bashrc` (or `.zshrc`, ...) add something like:

```
export JAVA_HOME=/usr/lib/jvm/java-1.5.0-sun
```

B.1.4.3. Mac OS

Under Mac OS X, if you have properly installed a JDK (XXX: check how), you will need to put in your your `.bashrc` (or `.zshrc`, ...) add something like:

```
export JAVA_HOME=/Library/Java/Home
```

B.2. Installing Ant

Ant will be used for building process. If you didn't set it up already on your computer, you can download it here.

Then need to have Ant setup and on your PATH environment variable.

For linux:

Add something like the following in your `.bashrc`:

```
export PATH=/opt/apache-ant-1.7.0/bin:$PATH
```

For Windows:

- type 'windows' +'pause'
- click on advanced tab
- click on "environment variables" (at the bottom)
- click on path variable and click modify
- add something like this at the end of the PATH definition: `;c:\program files\apache-ant-1.7.0\bin`
- don't forget to click on OK to close the environment variables window.

You can then check that your installation is correct by typing:

```
ant -version
```

B.3. Installing Maven

B.3.1. What is Maven?

Quoting the Wikipedia entry for Maven:

> Maven is a software tool for Java programming language project management and automated software build. It is similar in functionality to the Apache Ant tool, but has a simpler build configuration model, based on an XML format. Maven is hosted by the Apache Software Foundation, where it was formerly part of the Jakarta Project.
>
> Maven uses a construct known as a Project Object Model (POM) to describe the software project being built, its dependencies on other external modules and components, and the build order. It comes with pre-defined targets for performing certain well defined tasks such as compilation of code and its packaging.
>
> A key feature of Maven is that it is network-ready. The core engine can dynamically download plugins from a repository, the same repository that provides access to many versions of different Open Source Java projects, from Apache and other organizations and developers. This repository and its reorganized successor the Maven 2 repository are the de facto distribution mechanism for Java applications. Maven provides built in support not just for retrieving files from this repository, but to upload artifacts at the end of the build. A local cache of downloaded artifacts acts as the primary means of synchronizing the output of projects on a local system.
>
> Nuxeo is now fully "maven managed".

Nuxeo holds a Maven repository here: `http://archiva.nuxeo.org/archiva/`.

B.3.2. Installing Maven

You should then install Maven 2 on your development box by downloading the latest tarball from http://maven.apache.org/download.html and then untar the archive in `/opt` (for instance).

We recommend that you use the latest version of Maven (2.0.8 at the time of this writing).

As usual, you have to put the `mvn` executable into the path of your environment (cf. Ant)

Then add the `bin/` subdir in your `PATH` by adding something like the following in your `.bashrc`:

```
export PATH=/opt/maven-2.0.8/bin:$PATH
```

In a new shell you can then test:

```
$ mvn --version
Maven version: 2.0.8
```

B.4. Installing JBoss AS

Nuxeo EP default target is the JBoss application server with EJB3 support enabled. To enable the EJB3 support, you should install JBoss with the latest version of the JEMS installer:

```
$ sudo java -jar jems-installer-1.2.0.GA.jar
```

While executing the installation wizard, you must select `ejb3` install. You can leave all other parameters to their default values.

You would get PermGenSpace errors if you run JBoss without this configuration:

- Linux:

  Edit `/opt/jboss/bin/run.conf` and add the following line at the end of the file

```
JAVA_OPTS="$JAVA_OPTS -XX:MaxPermSize=128m"
```

  Restart JBoss.

- Windows:

  Edit `$JBOSS/bin/run.bat` and add the following line after the line : `set JAVA_OPTS=%JAVA_OPTS% -Dprogram.name=%PROGNAME%`

```
set JAVA_OPTS=%JAVA_OPTS% -XX:MaxPermSize=128m
```

  Restart JBoss.

B.4.1. JBoss AS listening ports customization

The common task for JBoss users is making it to communicate over a single HTTP server. This is quite useful for network administration, making it easier to go through firewalls. This section describes the necessary steps to make JBoss communicate primarily over HTTP

B.4.1.1. Tomcat Web server

JBoss is shipped with built-in Tomcat web server. This server is configured in 'deploy/jbossweb-tomcat55.sar/server.xml' By default only two connectors are enabled: HTTP connector (port 8080) and AJP connector (port 8009). Generally speaking you need only one of them. The former connector is needed if standalone HTTP server built in JBoss is used. You may want to configure it to listen the default HTTP port 80. The latter connector is needed only if you want to couple JBoss server with external web server like Apache, in this case it is reasonable, for security issues to change the binding address to 'localhost' (of course if Apache runs on the same host).

B.4.1.2. HTTP invoker web application

The JBoss default configuration deploys a special service that can be used to expose different JBoss services in the HTTP server. It is located in 'deploy/http-invoker.sar'. The configuration file 'deploy/http-invoker.sar/META-INF/jboss-service.xml' may be tweaked to tune the AS to specific needs. By default the service provides HTTP invoker MBean for EJB ('jboss:service=invoker,type=http') and two HTTP proxy MBeans that marshal the requests to the Naming service MBean ('jboss:service=invoker,type=http,target=Naming' and 'jboss:service=invoker,type=http,target=Naming,readonly=true'). If you need to provide HTTP interface to another MBeans, you also may specify the proxy services in 'deploy\http-invoker.sar\META-INF\jboss-service.xml'. For instance the SRP service for JBoss authentication may be exposed here.

The service also deploys web application 'deploy/http-invoker.sar/invoker.war', that configures the servlets that convert HTTP requests into invocation of MBeans/EJB methods. If you add HTTP proxies to MBeans, you may need to add servlets that handle the corresponding URI.

Important note. If HTTPS protocol is used the configuration should not use the default host name because the virtual host name used in the URL (say 'www.nuxeo.org') and exposed in SSL certificates usually differs from the computer name where JBoss is running. To accomplish this get rid of the following attributes: InvokerURLPrefix, InvokerURLSuffix, UseHostName, replacing them with a single InvokerURL attribute, like this:

```
<mbean code="org.jboss.invocation.http.server.HttpProxyFactory"
   name="jboss:service=invoker,type=https,target=Naming">
   <!-- Compose the invoker URL from the cluster node address -->
   <attribute name="InvokerURL">
      https://www.nuxeo.org/invoker/JMXInvokerServlet
   </attribute>
   <attribute name="ExportedInterface">
      org.jnp.interfaces.Naming
   </attribute>
   <attribute name="JndiName"></attribute>
   <attribute name="ClientInterceptors">
   <interceptors>
      <interceptor>org.jboss.proxy.ClientMethodInterceptor</interceptor>
      <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
      <interceptor>org.jboss.naming.interceptors.ExceptionInterceptor</interceptor>
      <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
   </interceptors>
   </attribute>
   <depends>jboss:service=invoker,type=https</depends>
</mbean>
<!-- The rest MBeans should also use InvokerURL attribute only,
   make sure you specify the right host name
-->
```

B.4.1.3. JNDI service

This is the core service of JBoss and should never be disabled. Nevertheless this service does not need own listening port (1099,1098), so just change the '1099' to '-1':

```
<mbean code="org.jboss.naming.NamingService"
   name="jboss:service=Naming"
   xmbean-dd="resource:xmdesc/NamingService-xmbean.xml">
   <!-- The call by value mode. true if all lookups are unmarshalled using
   the caller's TCL, false if in VM lookups return the value by reference.
   -->
   <attribute name="CallByValue">false</attribute>
   <!-- The listening port for the bootstrap JNP service. Set this to -1
     to run the NamingService without the JNP invoker listening port.
   -->
   <attribute name="Port">-1</attribute>
   <!-- The bootstrap JNP server bind address. This also sets the default
   RMI service bind address. Empty == all addresses, use localhost to hide this from
   network
    -->
   <attribute name="BindAddress">localhost</attribute>
   <!-- The port of the RMI naming service, 0 == anonymous, you cannot use -1 here -->
   <attribute name="RmiPort">1098</attribute>
   <!-- The RMI service bind address. Empty == all addresses, use localhost to hide this from
   network
    -->
   <attribute name="RmiBindAddress">localhost</attribute>
   <!-- The thread pool service used to control the bootstrap lookups -->
   <depends optional-attribute-name="LookupPool"
      proxy-type="attribute">jboss.system:service=ThreadPool</depends>
</mbean>
```

B.4.1.4. Default JBoss EJB invokers

You may deinstall the JRMP and Pooled invokers completely. Just comment out the MBeans that provide the corresponding services in 'conf/jboss-service.xml'.

Important note. The JBoss specifies the invokers for EJB in 'conf/standardjboss.xml' file. The default is 'jboss:service=invoker,type=jrmp' invoker. To change it to HTTP invoker you need to add invoker bindings for all EJB types deployed in your applications. Generally it means you need to copy all "*-rmi-invoker" bindings into "*-http-invoker" bindings, replacing "<invoker-mbean>jboss:service=invoker,type=jrmp</invoker-mbean>" with "<invoker-mbean>jboss:service=invoker,type=http</invoker-mbean>" for the new bindings. Also you will need to make the HTTP invoker default for all EJB container configurations replacing "<invoker-proxy-binding-name>*-rmi-invoker</invoker-proxy-binding-name>" with "<invoker-proxy-binding-name>*-http-invoker</invoker-proxy-binding-name>" correspondingly.

The easiest (but probably not the right) way for JBoss 4.0.x is to replace the string 'jboss:service=invoker,type=jrmp' with 'jboss:service=invoker,type=http' in this file by any text editor. It may be not correct if you want to mix both invokers for your EJBs.

B.4.1.5. JBoss EJB3 invoker

The EJB3 invoker which is specified in 'deploy/ejb3.deployer/META-INF/jboss-service.xml' uses JBoss remoting mechanism. By default it is bound to socket with a connector listening TCP/IP port 3873. This should be changed to the servlet locator:

```
<mbean code="org.jboss.remoting.transport.Connector"
       name="jboss.remoting:type=Connector,name=DefaultEjb3Connector,handler=ejb3">
   <depends>jboss.aop:service=AspectDeployer</depends>
   <attribute name="InvokerLocator">
      servlet://${jboss.bind.address}/invoker/Ejb3InvokerServlet
   </attribute>
   <attribute name="Configuration">
      <handlers>
         <handler subsystem="AOP">org.jboss.aspects.remoting.AOPRemotingInvocationHandler</handler>
      </handlers>
   </attribute>
</mbean>
```

The corresponding servlet should be added to invoker web application descriptor ('http-invoker.sar/invoker.war/WEB-INF/web.xml'):

```
<servlet>
   <servlet-name>Ejb3InvokerServlet</servlet-name>
   <description>
      The ServerInvokerServlet receives requests via HTTP protocol
      from within a web container and passes it onto the
      ServletServerInvoker for processing.
   </description>
   <servlet-class>
      org.jboss.remoting.transport.servlet.web.ServerInvokerServlet
   </servlet-class>
   <init-param>
      <param-name>locatorUrl</param-name>
      <param-value>
         servlet://${jboss.bind.address}/invoker/Ejb3InvokerServlet
      </param-value>
      <description>
         The servlet server invoker locator url
      </description>
   </init-param>
   <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
   <servlet-name>Ejb3InvokerServlet</servlet-name>
   <url-pattern>/Ejb3InvokerServlet/*</url-pattern>
</servlet-mapping>
```

B.4.2. Affected JBoss services

As JRMP invoker is used in many other JBoss services, so it should be replaced. The affected MBeans are "jboss:service=ClientUserTransaction" (conf/jboss-service.xml), "jboss.jmx:type=adaptor,name=Invoker,protocol=jrmp,service=proxyFactory".

B.4.2.1. Client User Transaction

The Client User Transaction service depends on two JRMP Proxy Factories described in the nested MBeans. Every JRMP proxy factory should be replaced with HTTP proxy factory:

```
<mbean
   code="org.jboss.tm.usertx.server.ClientUserTransactionService"
   name="jboss:service=ClientUserTransaction"
   xmbean-dd="resource:xmdesc/ClientUserTransaction-xmbean.xml">
  <depends>
    <mbean code="org.jboss.invocation.http.server.HttpProxyFactory"
         name="jboss:service=proxyFactory,target=ClientUserTransactionFactory">
      <attribute name="InvokerName">jboss:service=invoker,type=http</attribute>
      <attribute name="JndiName">UserTransactionSessionFactory</attribute>
      <attribute name="ExportedInterface">
        org.jboss.tm.usertx.interfaces.UserTransactionSessionFactory
      </attribute>
      <attribute name="ClientInterceptors">
        <interceptors>
          <interceptor>org.jboss.proxy.ClientMethodInterceptor</interceptor>
          <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
        </interceptors>
      </attribute>
      <depends>jboss:service=invoker,type=http</depends>
    </mbean>
  </depends>
  <depends optional-attribute-name="TxProxyName">
    <mbean code="org.jboss.invocation.http.server.HttpProxyFactory"
         name="jboss:service=proxyFactory,target=ClientUserTransaction">
      <attribute name="InvokerName">jboss:service=invoker,type=http</attribute>
      <attribute name="JndiName"></attribute>
      <attribute name="ExportedInterface">
        org.jboss.tm.usertx.interfaces.UserTransactionSession
      </attribute>
      <attribute name="ClientInterceptors">
        <interceptors>
          <interceptor>org.jboss.proxy.ClientMethodInterceptor</interceptor>
          <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
        </interceptors>
      </attribute>
      <depends>jboss:service=invoker,type=http</depends>
    </mbean>
  </depends>
</mbean>
```

Note that JRMP Proxy factory attributes differ from attributes of HTTP proxy factory.

B.4.2.2. JMX adaptor

The JMX adaptor is adapted for HTTP as following ('deploy/jmx-invoker-service.xml'):

```
<mbean code="org.jboss.invocation.http.server.HttpProxyFactory"
       name="jboss.jmx:type=adaptor,name=Invoker,protocol=http,service=proxyFactory">
  <attribute name="InvokerURL">https://www.nuxeo.org/invoker/JMXInvokerServlet</attribute>
  <depends optional-attribute-name="InvokerName">jboss.jmx:type=adaptor,name=Invoker</depends>
  <attribute name="ExportedInterface">org.jboss.jmx.adaptor.rmi.RMIAdaptor</attribute>
  <attribute name="JndiName">jmx/invoker/HttpAdaptor</attribute>
</mbean>
```

and ('deploy/console-mgr.sar/META-INF/jboss-service.xml'):

```
<mbean code="org.jboss.console.manager.PluginManager"
       name="jboss.admin:service=PluginManager">
  <depends>jboss.jmx:type=adaptor,name=Invoker,protocol=http,service=proxyFactory</depends>
  <!-- the rest stays intact -->
</mbean>
```

B.4.2.3. Datasource adaptors

You need to set the invoker explicitly for all deployed data sources. The element <jmx-invoker-name> should be added to all <local-tx-datasource> and <xa-datasource> elements. Otherwise the server will complain about missing JRMP invoker which is used by default:

```
<datasources>
        ...
   <local-tx-datasource>
      <!-- specify explicitly the invoker to use -->
      <jmx-invoker-name>jboss:service=invoker,type=https</jmx-invoker-name>
      <!-- the rest stays intact -->
               ...
   </local-tx-datasource>
        ...
</datasources>
```

B.5. Installing a Subversion client

The official svnbook is a very good reference for both beginners and advanced subversion users.

B.5.1. Generic subversion clients with linux

The Nuxeo EP source repository is a Subversion repository thus you will need a subversion client to access the source code. Most Linux distributions provide the `svn` command line tool. To install it under Ubuntu / Debian:

```
$ sudo apt-get install subversion
```

Under Fedora Core, this should become:

```
$ yum install subversion
```

B.5.2. Windows

For MS Windows, we recommend to use the TortoiseSVN Subversion client. You can also directly use the Subversion command-line client from Subversion website.

B.6. Chapter Key Point

In this chapter, we learned:

- how to install a Java development environment (JDK) on your machine
- how to install Ant and Maven, two mandatory tools for building and deploying your own projects on top of the Nuxeo platform
- how to install the JBoss AS 4 application server that will act as a container for the Nuxeo application
- how to install a Subversion client, which is not mandatary anymore (but may prove useful for you own sake)

Appendix C. Coding and Design Guidelines

C.1. Introduction

"Follow a common coding standard, so that all the code in the system looks as if it was written by a single — very competent — individual. The specifics of the standard are not important: what is important is that all the code looks familiar, in support of collective ownership."

—Ron Jeffries *XProgramming.com*

Coding Standards are a good idea. Every team should adopt a coding style and standard, and stick to it. The code produced by that team should have a consistent look and feel that is devoid of individual preferences and fetishes.

—"Uncle" Bob Martin *ObjectMentor website*

The primary goal of this chapter is to provide common conventions, as well as guidelines that aim at ensuring a high level of quality (with respect to maintainability, extensibility, modularity, security, testability, documentation, etc.) throughout the Nuxeo project.

As such, it is primarily written for the platform's developers, both "core" developers and contributors, and developers that write extension modules that will someday find a place in the Nuxeo codebase.

If you're working on your own project with no plan to contribute code to the Nuxeo platform, you will probably be still interested in reading this chapter, and adapt the conventions stated here to your company's or organization's own coding standards.

C.2. External Coding Standards

Rewriting a whole coding standard for the Nuxeo project would be a poor use of our time, since there are already several documents and books that do a fine job in this respect.

We've opted for a more pragmatic, two-pronged approach:

1. reference reputable external guides or books, and document any difference
2. propose default settings for common Eclipse tools (including the built-in code formatter and the CheckStyle plugin), tuned to comply to these conventions.

The official coding standard for the project is: "The Elements of Java Style" [Vermeulen2000] which is a little book that can be bought from Amazon or other book dealers.

If a PDF suits you better, you can download an earlier version of the book from here.

Note however that these guidelines have been written in 2000 (last century!) and some of the recommendations need the be updated in light of this millenium's experience.

There are also architectural guidelines that need to be followed. At this moment, they are written here (section 4 - "Guidelines")

C.3. Some points that need attention

C.3.1. Java code formating

Readability counts.

—Tim Peters *The Zen of Python*

Java code should be formatted using Sun's Code Conventions for the Java Programming Language.

Regarding code formatting (which is just one small part of the book and PDF mentioned above), we'll be using the standard Eclipse default style, with a couple of minor tweaks (see this README.txt for how to configure Eclipse to support this style).

The major points are:

- 4 spaces (no tabs!!!) indentation
- spaces before and after = signs and most binary operators
- spaces after *,*
- no space after *(* or before *)*
- regarding code block, we are using 1TBS ("One True Brace") style:

  Bad:

```
if (xxx)
{
    yyy
}
```

  Good:

```
if (xxx) {
    yyy
}
```

- make a block even for only one statement:

  Bad:

```
if someThing() doSomething;

if someThing()
    doSomething;
```

  Good:

```
if someTest() {
    doSomething;
}
```

- Don't prefix your instance variables by "this." when it's not mandatory.

  Why? Because with modern IDEs, instance variables are typeset in a different color than local variables, hence you don't need the extra information provided by the "this." prefix to recognize which is which.)
- etc.

C.3.2. XML code formatting

1. XML code should also be formatted, using 2 spaces for each indent level (not 4 spaces).

   Badly formatted XML code can be reformatted using the Eclipse formatter.

Always check that the result is better than the original, and tweak afterwards if needed.

C.3.3. Design

1. Use interfaces every time it is possible instead of concrete classes.

   Ex: declare a variable or argument as a "Map", not as a "HashMap".

   Why? Using interfaces makes it possible to use several different implementations. Actually, interfaces are the basis of component-oriented programming.

2. But don't overdo it if it's not necessary.

   Why? The more code we have, the more expensive maintainance is.

3. Avoid cyclic dependencies.

   Why? This is a common design principle, called the "Acyclic Dependency Principle" or "ADP".

   How? use JDepend or a JDepend plugin for you IDE (ex: http://andrei.gmxhome.de/jdepend4eclipse/) or look at the JDepend report by Maven.

   More info:

   - How JDepend Changed My Java Packaging
   - Managing Your Dependencies with JDepend
   - Uncle Bob Martin's "Design Principles and Design Patterns" (pages 18-22, "The Acyclic Dependencies Principle (ADP)")

C.3.4. Unit tests

> If it's not tested, it's broken.
>
> —Bruce Eckel

1. Write unit tests. A lot.

   How? Learn how to use JUnit. Use tests in existing modules as examples. Remember that well-tested modules have at least as many lines of test code than not test code.

2. Check that your unit tests have a coverage of most of the critical parts of your code.

   How? Look at the Cobertura report by Maven, or use the TPTP or EMMA (http://www.eclemma.org/) plugins for Eclipse or use the native code-coverage function of IDEA.

3. Design your API so as to make unit testing possible and easy.

   See API Design As If Unit Testing Mattered

Some cosmetic remarks related to writing unit tests:

1. Use "assertEquals(constant, value)", not "assertEquals(value, constant)"

   Why? Because one has to choose a convention and this one

2. Use "assertNull(value)", not "assertEquals(null, value)"

Why? Because code is shorter and looks cleaner this way.

C.3.5. Security

1. Read Graff and van Vyk's "Secure Coding" [Graff2003] book. More info on the securecoding.org website.
2. If you don't have access to the book, read at least Secure Coding: The State of the Practice (including the Architectural Principles, the Design Ideas and the Java Tips sections) from one of the two authors.
3. Read these slides from JavaOne 2006.

C.3.6. Naming convention

1. Don't use underscores ("_") in variables, methods or class names, use camelCase notation instead.

   Why? Because it is the Java convention.
2. Check some good articles on naming things, like Ottinger's Rules for Variable and Class Naming.

   It is especially important that we all use the same words (in case there is an ambiguity) to refer to the same concepts (like "fetch" vs. "retrieve", for instance).

C.3.7. Information hiding

1. Minimize the accessibility of classes and members (for instance by making them "private" unless you have a reason to believe that they will be used from other classes, and more generally by using the weakest access modifier), so as to keep you API clean and easy to understand for other developers.

   Reference: "Effective Java" [Bloch2001], item 12.

C.3.8. Use modern Java features

1. Use Java 5's "foreach" construct instead of explicit iterators whenever possible (which is not always the case).
2. Use java 5's generics.

   Why? This will improve the amount of static checks done by the compiler.

C.3.9. Documentation: Comments and Javadoc

> Reuse is something that is far easier to say than to do. Doing it requires both good design and very good documentation. Even when we see good design, which is still infrequently, we won't see the components reused without good documentation.
>
> —D. L. Parnas *The Mythical Man-Month: Essays on Software Engineering*

1. Always use comments to document what is not obvious.

   Why? Otherwise, it will be harder for others (even you!) to maintain or extend your code. Some people may even end up refactoring your code under wrong assumptions.

   See for instance item #28 of Bloch's "Effective Java" [Bloch2001] for more insight, including this most important principle:

The doc comment for a method should describe succinctly the contract between the method and its client. With the exception of methods in classes designed for inheritance, the contract should say what the method does rather than how it does its job.

2. Write javadocs according to the javadoc manual and the official Sun javadocs guidelines.

   Why? Because javadocs are compiled by a standard tool, you must conform to the syntax mandated by this tool.

3. Common javadoc guidelines that are often forgotten include:

   - Don't include XML tags without escaping the < and > characters.
   - Remember that the first sentence (i.e. everything until the first dot) is used as a short description of the package / class / method / etc. it documents. Hence end all your sentences with a dot.
   - Use <p> for separating paragraphs, not <br> or
   - Don't use </p> in the source though as it is useless and rather bad looking.
   - If your javadoc starts with "This class" or "This methods", this is most probably wrong.

4. Proofread carefully (including checks for spelling and grammar errors) and check that your javadocs provide useful information for third-party developers (or your teammates) by actually reading the generated doc.

   How? See either "Project -> Generate Javadocs" from Eclipse, or "mvn javadoc:javadoc" from the project root and browse the docs.

   Remember that you can see a preview of how your javadoc will look like from Eclipse by using the "Javadoc" tab in the Java perspective.

5. Don't mix Javadoc comments with implementation comments.

   Javadocs comments are intended to be read by clients of the API, implementation comments by people who need to review you code or who will end up maintaining it (including yourself).

   For instance, don't put "TODO" markers in your javadoc because they are intended for yourself or other people co-developing or maintaining your code, not clients of the API.

6. Start by documenting the most important and less obvious things.

   If you don't have enough time to document everything, document first the API part of your code, because that's what third-party developers are going to use.

   Also documenting the non-obvious (like giving an overview of a package or an important class) is more important than writing for instance, that a "getTitle()" method "@returns the title".

7. Write package-level javadocs.

   Package-level Javadoc (just package.html files in your source code) are the most important javadocs you have to write, because that's what people reading the javadoc-generated site will read first, and because that's where the information is usually the less obvious.

8. Sign your code (in the modules headers).

   It is very important that people who will read / maintain your code know that you are the author, so that they can ask you questions, give you feedback, etc.

9. When you've borrowed code from another open source project, always document it so that:

- we are sure that there is no license conflict with the original code
- we understand why the code in question doesn't follow our own coding conventions or isn't unit-tested as it should (it should anyway, but this is another story)

10. Put markers as (inline) comments to document parts of your code that will need further work.
    - use FIXME for really serious (like, release-blocking) issues.
    - use the TODO for remaining tasks.
    - don't use XXX which is usually associated with appropriate content.
    - use BBB to mark code used to ensure compatibility with a deprecated feature, that will be removed after a certain release.
    - do not leave TODO markers behind when they are not relevant (for instance for dummy classes used in tests and auto-generated by the IDE from an interface or an abstract class).
11. Look critically at the javadoc-generated site and try to improve it.

    Either go to http://maven.nuxeo.com/apidocs/ and check the apidoc for your project, or run *mvn javadoc:javadoc* locally and browse the generated apidoc, and ask yourself the simple question: "if I was a third-party developer, would I understand how to use the API by reading this".

C.3.10. Deprecation

1. Don't use deprecated features (= API), either from Java or third-party libraries, or from the Nuxeo framework.

   Hint: they should appear as stroked-out in your IDE.
2. Deprecate your own API when you have to, but make sure you write comments that explain to your API's clients how to migrate from the old API to the new one. Use BBB markers (see above) if needed.
3. Deprecated APIs should be maintained at least for 1 release cycle, at least for external clients of the APIs, but we should strive to switch to the new APIs internally in 1 release cycle.

C.4. Methodology tips

Here are a few points and tips to keep in mind.

C.4.1. Use the power of your IDE (and its plugins)

Modern IDEs (+ adequate plugins, if necessary) include many code-checking functions that help find out issues:

1. For Eclipse, the simplest plugin to use is TPTP.

   How?
   - Use the update manager to get the TPTP plugins.
   - Right-click on your project, and select "Analysis".
   - Enable all the rules, and run the analysis on your module.
   - Fix the issues that appear serious (you still have to think).

2. You should also use the Checkstyle and FindBugs Eclipse plugins to ensure minimal bug count and maximal coding style coherence.

   See these great slides from JavaOne 2007 for instance.

3. Read Improving code with Eclipse plugins on developerWorks for more background information on the subject.

   From the article:

   > This article covers what I consider to be the "big five" code analysis areas:
   >
   > - Coding standards
   > - Code duplication
   > - Code coverage
   > - Dependency analysis
   > - Complexity monitoring
   >
   > These analysis areas can be uncovered using a number of the following slick Eclipse plugins:
   >
   > - CheckStyle: For coding standards
   > - PMD's CPD: Enables discovering code duplication
   > - Coverlipse: Measures code coverage
   > - JDepend: Provides dependency analysis
   > - Eclipse Metrics plugin: Effectively spots complexity

   NB: Coverlipse may or may not work correctly, an alternative is EclEmma which is very similar.

C.4.2. Refactor

1. When something looks wrong in the code ("smell"), refactor it, but make sure you don't break anything.

   How?

   - Check that the code you are refactoring is covered by some unit tests.
   - Refactor. (Tip: a modern IDE (Eclipse and IDEA for instance) has some functions that may help.)
   - Check that the code still compiles (including dependent modules) and all the unit tests are still passing, and commit.

C.5. Important references

Here is a list of useful stuff to read:

1. Joshua Bloch's "Effective Java" [Bloch2001] book. Highly recommended. Probably the best "advanced" Java book.
2. Joshua Bloch's "Designing Effective API" slides and video.

3. ObjectMentor's design articles (click on "Design Principles"), the most important one being Principles and Patterns.

Appendix D. Development Tools and Process

This chapter describes useful development tools.

D.1. Code Quality with Eclipse Plugins

Eclipse can benefit for several plugins for improving code quality, in both adherence to the project's coding standard and in removing bugs.

D.1.1. Using Checkstyle

TODO

D.1.1.1. The Short Story

If you are already familiar with the Checkstyle Eclipse plugin, just configure it to use the `checkstyle.xml` at the root of the SVN directory for nuxeo-ep:
`http://svn.nuxeo.org/trac/nuxeo/browser/nuxeo-ep/trunk/checkstyle.xml`

D.1.1.2. Longer Story: installation and Configuration

TODO

D.1.1.3. Notes on Reported Issues

TODO

D.1.2. Using TPTP

TODO

D.1.3. Using FindBugs

TODO

D.2. Profiling with NetBeans Profiler

IDEA doesn't provide an integrated profiler and Eclipse's profiler, provided by the TPTP project, doesn't currently work on Mac OS, one can use NetBeans to profile the Nuxeo platform.

Here is how to do it:

1.

D.3. NXPointDoc Documentation tool

Nuxeo heavily uses extension points. In order to manage them nxPointDoc tool has been created. Its purpose is to explore all XML files and build the documentation of each explored components. Cross links and indexes are also built to ease the navigation.

The NxPointDoc pages are available at `http://svn.nuxeo.org/nxpointdoc/`

The tool is written in Python and the following libraries are needed:

- Genshi for templating
- ElementTree for XML processing
- Pygments for code highlighting

D.3.1. Documenting a component

A component XML file is structured as follow:

```
<component ...>

  <!--############## Component configuration ############-->
  <!-- implementation class (optional) -->
  <implementation>...</implementation>

  <!-- component properties (optional) -->
  <property name="..." value="..."/>
  ...
  <property name="..." value="..."/>

  <!--############### Extension points ################-->
  <!-- extension points are optional -->
  <extension-point ...>
      ...
  </extension-point>
  ...

  <!--############### Contributions ################-->
  <!-- contributions are optional -->
  <extension ...>
      ...
  </extension>
  ...

</component>
```

We can see that the only required element is the component element (although it is useless to have an empty component). So there are 3 main sections (any of these sections are optional).

- Component configuration: This section defines the component implementation class and some properties to initialize the component (This section content may be modified in future especially when aligning nuxeo components with OSGi services).
- Extension points: This sections contains all the extension point declared by the component.
- Contributions (`extension` tag): This section contains all the contributions made by this component to other components.

To add documentation to these elements a `<documentation>` tag will be used. An element may have different content depending on what it is documenting. While some information is already available in other XML elements in the file, there is no need to duplicate these information inside the documentation provided though the element. For example the name of the component can be retrieved from the name attribute of the component element, the implementation class name from the implementation element etc.

To format the description text, we can use XHTML tags and javadoc-like markers such as `@property`, `@schema` etc. Javadoc-like links are also supported: `@see` points on the javadoc, `@component` points on another component documentation. For example, `{@see org.nuxeo.ecm.core.schema.types.Type}` will point on the `http://maven.nuxeo.org/apidocs` corresponding page while `{@component org.nuxeo.ecm.webapp.directory.DirectoryTreeService}` will explicitly insert a link to the related NxPointDoc page. Code colorization is also supported through the `<code language='xml'> ... </code>` tags. If no language is given, `xml` is taken by default. Java (`language='java'`) and many other languages are supported (see Pygments pages).

Regarding `<component>` documentation, the following elements are available:

- `@author`: may be duplicated for multiple authoring

- `@version`
- `@property`
- `@see`: points to Javadoc
- `@component`: points to nxpointdoc
- `@deprecated`
- `component@name` attribute: the component name
- `component/implementation` tag: the implementation class
- `component/require` tag: required elements
- `component/documentation` tag: the description

For `<extension-point>` the following elements have to be used:

- `@author`: may be duplicated for multiple authoring
- @schema
- `@deprecated`
- `@see`
- `@component`
- `component/extension-point@name` attribute: the name
- `component/extension-point/documentation` tag - the description
- If the extension point is using `object` sub-elements, the `DTD` should be extracted from the XMap annotated class, otherwise the user may specify the `DTD` using the `@schema` marker inside the `documentation` element

For <extension>, describing contributions to an extension-point, we have

- `@author`: may be duplicated for multiple authoring
- `@see`
- `@component`
- `@deprecated`
- `component/extension@target` attribute - rendered as a link to the component documentation
- `component/extension@point` - rendered as a link to the extension point documentation
- `component/extension-point/documentation` description

Here is a short example of what a component xml file may look like.

```
<?xml version="1.0"?>
<component name="org.nuxeo.ecm.MyService">
  <documentation>
  My demo service
  <p/>
  This service does nothing
  @property home home directory to be used to create temp files
  @property timeout the time interval in seconds
  @version 1.0
  @author Bogdan
  </documentation>
  <require>org.nuxeo.ecm.Service1</require>
  <require>org.nuxeo.ecm.Service2</require>
  <implementation class=”org.nuxeo.ecm.core.demo.Service2”/>
  <property name=”home” value=”/home/bstefanescu”/>
  <property name=”interval” value=”20” type=”Integer”/>

  <extension target="org.nuxeo.ecm.SchemaService" point="schema">
      <documentation>Common and Dublin Core schemas</documentation>
      <schema name="dublincore" src="schema/dublincore.xsd" />
      <schema name="common" src="schema/common.xsd" />
  </extension>

  <extension-point name="repository">
      <documentation>Register new repositories</documentation>
      <object class="org.nuxeo.ecm.RepositoryDescriptor"/>
  </extension-point>

</component>
```

Feel free to browse the NxPointDoc site and teh corresponding xml file to go deeper. The systematic link to the source code svn repository may help you.

D.3.2. Creating the NxPointDoc site

nxpointdoc is a command line program that creates the whole site from a source repository `SOURCE_DIR` to a target publication directory `TARGET_DIR`. Each component is analyzed and all related pages created. Index pages are then created.

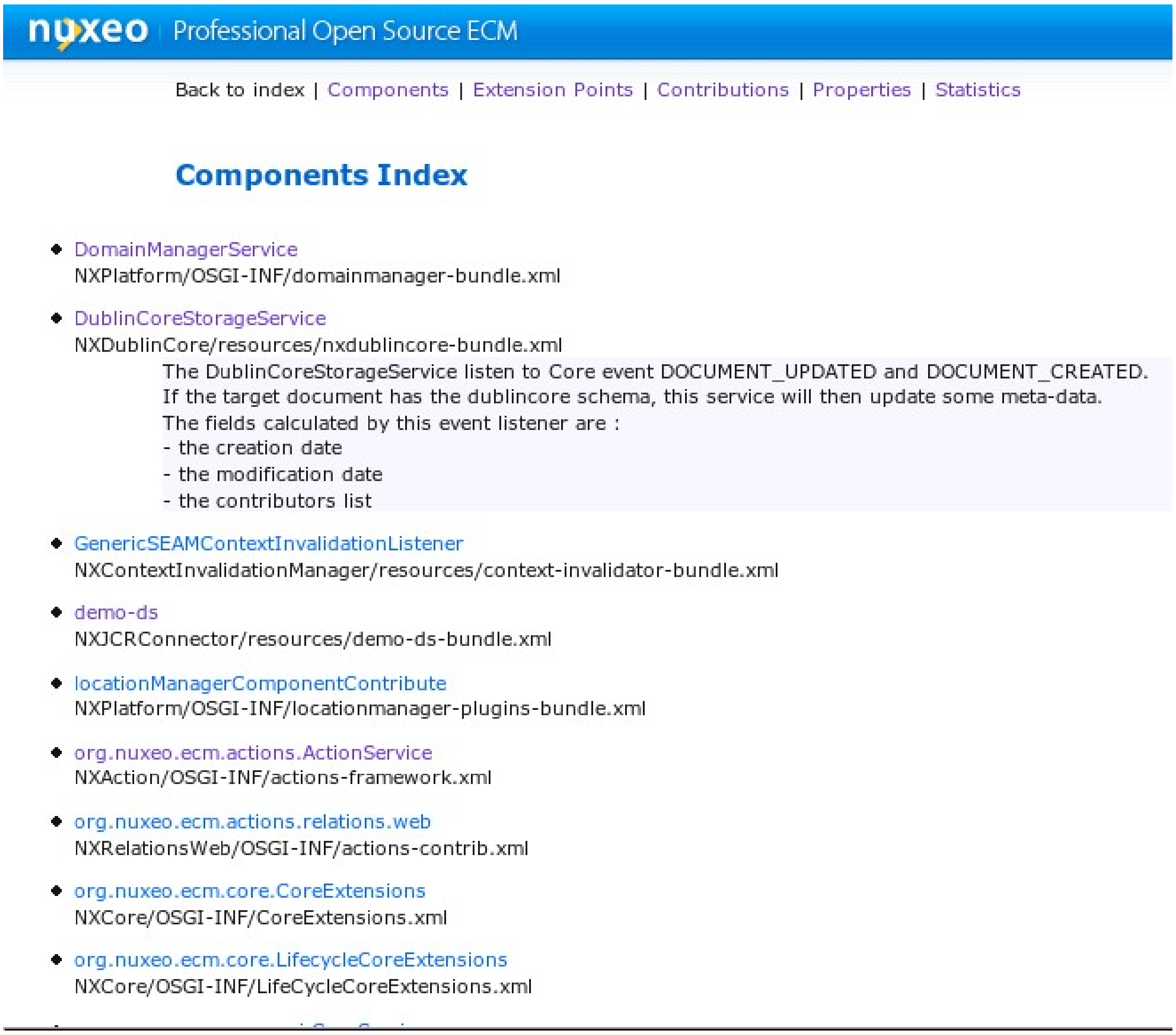
```
$ ./nxpointdoc.py -h
usage: nxpointdoc.py [options]

options:
--version                          show program's version number and exit
-h, --help                         show this help message and exit
--source=SOURCE_DIR                Source root directory containing xml component files
--target=TARGET_DIR                Target directory for the generated documentation
--template=TEMPLATE                Genshi template for component html file
--template-index=TEMPLATE_INDEX    Genshi template for index html file
--allow-xhtml-comment=ALLOW_XHTML_COMMENT 'no' to not interpret xhtml tags in comment
--color=COLOR_CODE                 'no' to not color <code> contents
```

Valid `SOURCE_DIR` and `TARGET_DIR` are mandatory. Template files have to exist. The one delivered have the `.template` extension and can be used as is.

D.3.3. Browsing NxPointDoc

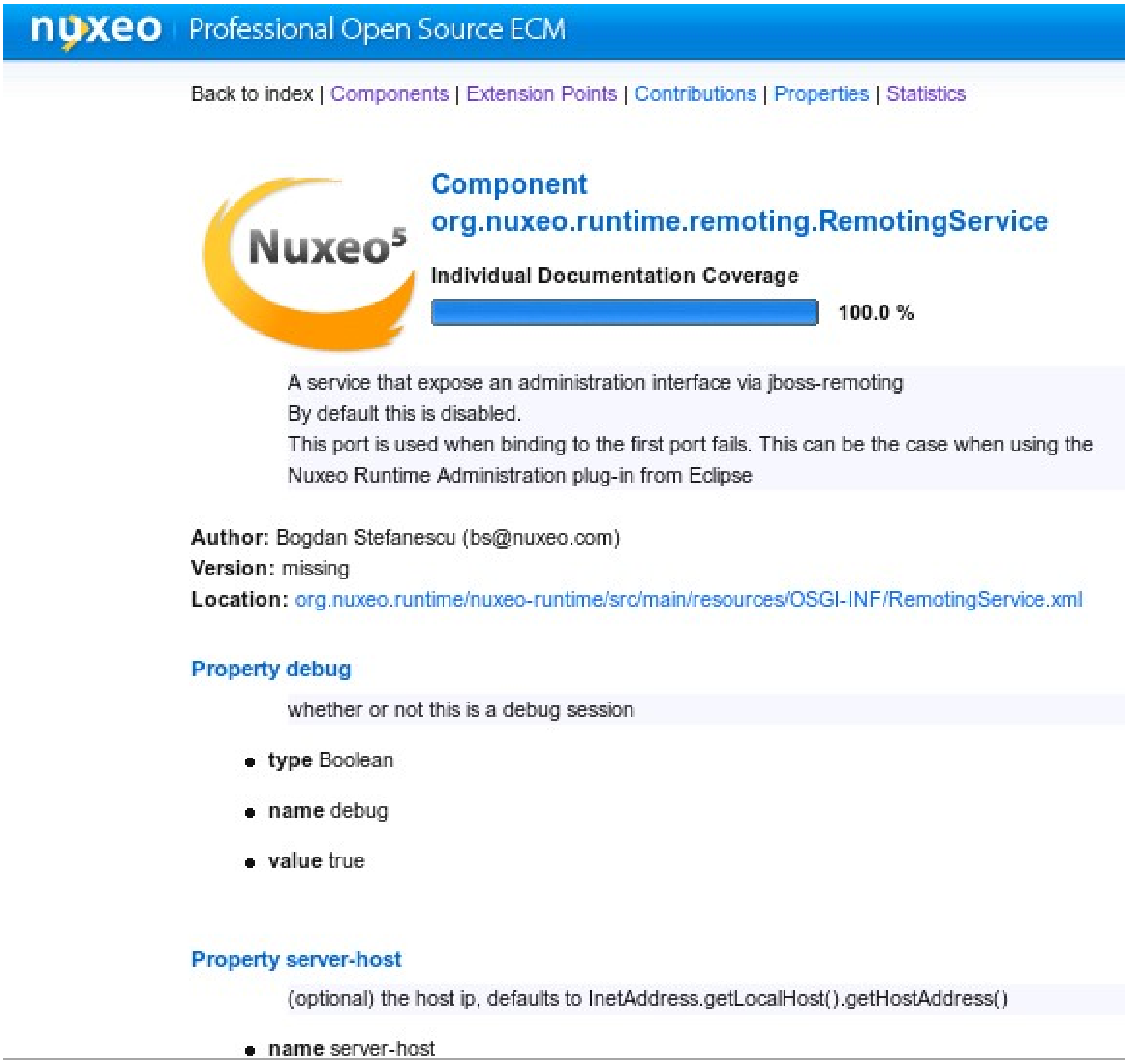
NxPointDoc generates 3 indexes that are the entry points; The documentation is accessible at `http://svn.nuxeo.org/nxpointdoc/` with 3 indexes related to components, extension points and contributions. Each one is an entry point for the documentation. The Indexes give the name and the first line of the documentation. An hyperlink allows to see the detail of the examinated item.



Professional Open Source ECM
Back to index | Components | Extension Points | Contributions | Properties | Statistics

Components Index

- DomainManagerService
  NXPlatform/OSGI-INF/domainmanager-bundle.xml
- DublinCoreStorageService
  NXDublinCore/resources/nxdublincore-bundle.xml
  The DublinCoreStorageService listen to Core event DOCUMENT_UPDATED and DOCUMENT_CREATED.
  If the target document has the dublincore schema, this service will then update some meta-data.
  The fields calculated by this event listener are :
  - the creation date
  - the modification date
  - the contributors list
- GenericSEAMContextInvalidationListener
  NXContextInvalidationManager/resources/context-invalidator-bundle.xml
- demo-ds
  NXJCRConnector/resources/demo-ds-bundle.xml
- locationManagerComponentContribute
  NXPlatform/OSGI-INF/locationmanager-plugins-bundle.xml
- org.nuxeo.ecm.actions.ActionService
  NXAction/OSGI-INF/actions-framework.xml
- org.nuxeo.ecm.actions.relations.web
  NXRelationsWeb/OSGI-INF/actions-contrib.xml
- org.nuxeo.ecm.core.CoreExtensions
  NXCore/OSGI-INF/CoreExtensions.xml
- org.nuxeo.ecm.core.LifecycleCoreExtensions
  NXCore/OSGI-INF/LifeCycleCoreExtensions.xml


nuxeo Professional Open Source ECM
Back to index | Components | Extension Points | Contributions | Properties | Statistics
Nuxeo5

Component org.nuxeo.runtime.remoting.RemotingService

Individual Documentation Coverage

100.0 %

A service that expose an administration interface via jboss-remoting
By default this is disabled.
This port is used when binding to the first port fails. This can be the case when using the Nuxeo Runtime Administration plug-in from Eclipse

Author: Bogdan Stefanescu (bs@nuxeo.com)
Version: missing
Location: org.nuxeo.runtime/nuxeo-runtime/src/main/resources/OSGI-INF/RemotingService.xml

Property debug

whether or not this is a debug session

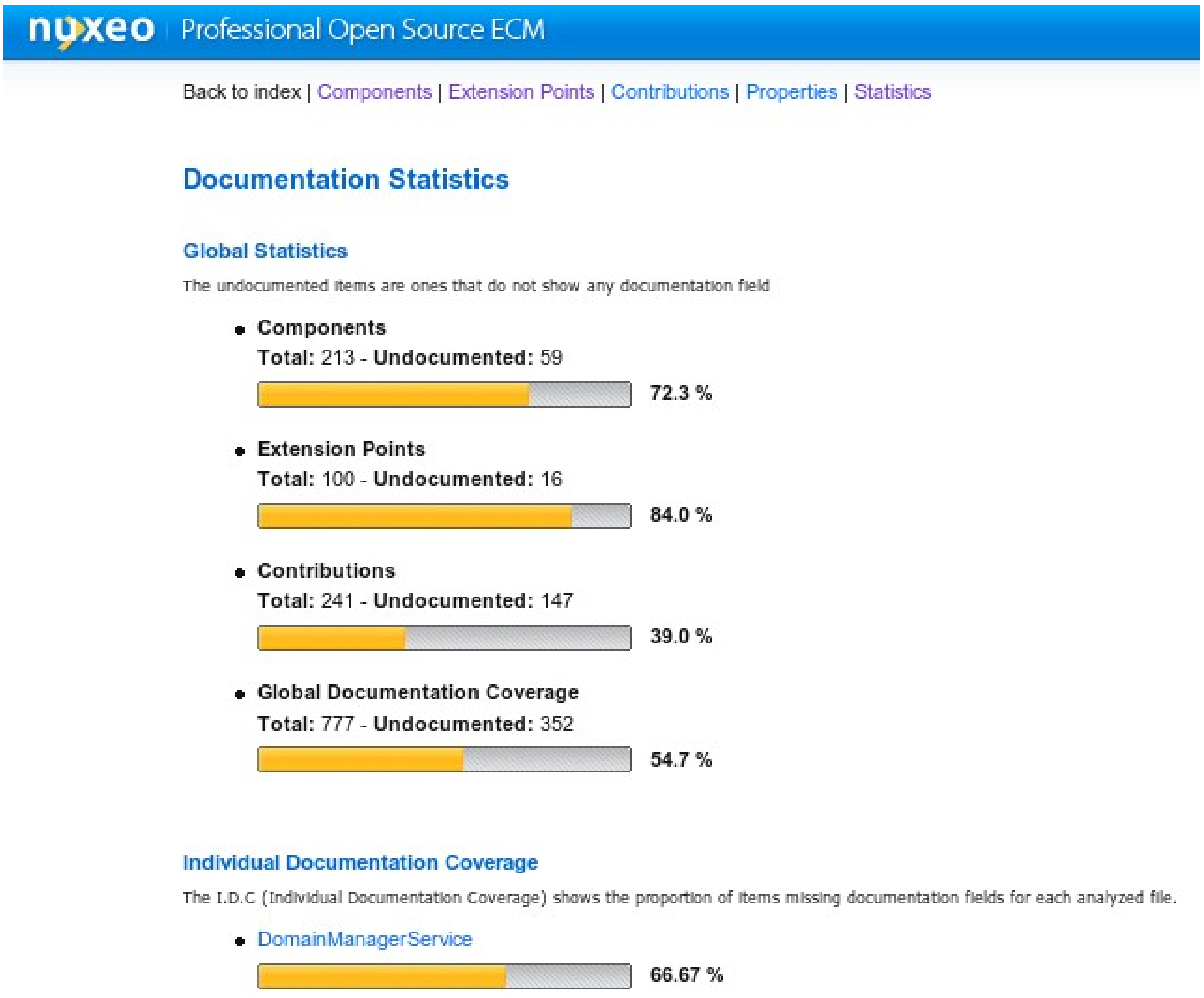
- **type** Boolean
- **name** debug
- **value** true

Property server-host

(optional) the host ip, defaults to InetAddress.getLocalHost().getHostAddress()

- **name** server-host

The statistic gives some rough indicators on the documentation coverage, globally or for each component file. The `G.D.C` stands for Global Documentation Coverage while the `I.D.C` stands for Individual Documentation Coverage. They show the ratio between all the information/documentation that is written over all the entries that are considered as mandatory (like author, documentation, etc.). The higher these indicators are, the better it is.


nuxeo Professional Open Source ECM

Back to index | Components | Extension Points | Contributions | Properties | Statistics

Documentation Statistics

Global Statistics

The undocumented items are ones that do not show any documentation field

- **Components**
  Total: 213 - **Undocumented:** 59
  72.3 %
- **Extension Points**
  Total: 100 - **Undocumented:** 16
  84.0 %
- **Contributions**
  Total: 241 - **Undocumented:** 147
  39.0 %
- **Global Documentation Coverage**
  Total: 777 - **Undocumented:** 352
  54.7 %

Individual Documentation Coverage

The I.D.C (Individual Documentation Coverage) shows the proportion of items missing documentation fields for each analyzed file.

- DomainManagerService
  66.67 %

Appendix E. Commercial Support

E.1. About Us

Nuxeo 5 is developed and published by Nuxeo SAS, a company with offices in France and the UK.

Through our commercial offer, Nuxeo Connect, we deliver enterprise-grade functional and technical support, certified software patches and updates, and management tools that assist you during every stage of the application life-cycle - from design and development, throughout testing and deployment, to operations and monitoring. Nuxeo Connect helps reduce business and technical risks, increase productivity, speed time to deployment and improve your success rate for all Nuxeo-based projects.

We're also happy to work with partners, IT Integrators or ISVs, to deliver the best possible applications to customers.

E.2. Contact information

E.2.1. General

Web: `www.nuxeo.com`

E-mail: contact@nuxeo.com

E.2.2. France

18-20, rue Soleillet

75020 Paris - France

Tel: +33 1 40 33 79 87

Fax: +33 1 43 58 14 15

E.2.3. UK

Garden Studios, 11-15 Betterton Street, London WC2H 9BP

Tel: +44-207-043-7933