Nuxeo Enterprise Platform - Version 5.1

The administration guide

5.1 / 5.2

Table of Contents

Part I. Introduction

Chapter 1. Preface

1.1. What this Book Covers

The primary focus of this book is the presentation of Nuxeo EP 5.1, from the perspective of its architecture, configuration and exploitation.

1.2. What this book doesn't cover

This book is not an end-user or developer manual for Nuxeo. If you are interested in such a document, we recommend that you get it from `http://doc.nuxeo.org/`.

1.3. Target Audience

As a administration guide for the Nuxeo platform, this book has several intended audiences:

- System integrators, who need to understand how to configure and adapt the Nuxeo platform to their customers needs.
- System administrators, who need to understand how to configure the platform for the specific needs of their hosting environment.
- Core developers, who work on the platform and need an administration documentation.

Readers of this book are presumed familiar with the Java EE and XML technologies.

1.4. About Nuxeo

Founded in 2000, Nuxeo SAS is part of the "second wave" of open source companies, and focuses on developing and supporting applications, instead of system software or development tools.

By entering the ECM field early in 2002, Nuxeo has established itself as the leader of open source ECM, with customers for critical projects in the Government, Energy and Finance sectors. Nuxeo currently has 40 employees, about half of them developers.

Nuxeo has customers and partners in Europe, in Northern and Southern America, in Africa and in India.

1.5. About Open Source

Simply put, Open source is a better way to develop and support software.

Nuxeo fully embraces the open source vision, by fostering collaboration with a community of external contributors.

Chapter 2. Introduction

This chapter will give you an overview of ECM and the motivation for using the Nuxeo platform in your next ECM project.

2.1. Enterprise Content Management

According to the AIIM, the Association for Information and Image Management, ECM is defined as "the technologies used to capture, manage, store, preserve, and deliver content and documents related to organizational processes".

2.1.1. Why ECM?

A March 2005 white paper on “The Hidden Costs of Information Work” by the IDC in the United States found that the average office employee spent approximately one day per week organizing and filing documents that they used. This can equate to a considerable amount of cost and inefficiency. A further twelve hours was spent on managing document approval, managing document routing and publishing to other channels. Nine hours was spent searching for documents.

2.2. The Nuxeo ECM platform

2.3. Introduction FAQ

2.3.1. What are Nuxeo EP 5, Nuxeo EP and Nuxeo RCP?

Nuxeo EP 5 is the 5th version of the open source platforms developed by Nuxeo (the four previous ones where known as "CPS").

Nuxeo EP or "Enterprise Platform" is the server part of Nuxeo EP 5. It is a Java EE application intended to run in a standard Java EE 5 application server like JBoss. It can be accessed by end users from a web browser, from office productivity suites like MS-Office or OpenOffice.org, or from rich client developed using the Nuxeo RCP technology (see below).

Nuxeo RCP or "Rich Client Platform" is a platform for building rich client applications, that usually connect to a Nuxeo EP server.

Nuxeo EP and Nuxeo RCP run on top of a common runtime, "Nuxeo Runtime", and share a common set of core components, called "Nuxeo Core".

2.4. Intended audience

2.5. What this book covers

Part 1: Introduction

Part 2:

Part 3:

Part 4:

Chapter 3. General Overview

3.1. Introduction

3.1.1. 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.

3.1.1.1. 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 needs 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 a 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

3.1.1.2. 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 gives 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 we worked a lot on caching and lazy fetching.

With Nuxeo EP we incorporated this requirement from the beginning:

- Distributed caching
- Lazy fetching on schemas and fields

3.1.1.3. 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 has 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.

3.1.1.4. 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.

3.1.1.5. 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.

3.1.1.6. 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.

3.1.1.7. 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 won't 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 ....

3.1.2. 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.

3.1.2.1. 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.

3.1.2.2. 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.

3.1.2.3. 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).

3.1.2.4. 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
    - ...

3.2. Nuxeo Runtime: the Nuxeo EP component model

3.2.1. 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.

3.2.1.1. 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.

3.2.1.2. 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.

3.2.2. 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.

3.2.2.1. 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.

3.2.2.2. 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.

3.2.3. 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>
```

3.2.4. Extension points and Nuxeo 5

3.2.4.1. 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 an externalized index
  - Corporate security policy: implement a specific security manager that will enforce business rules

- Event handlers

  Nuxeo platform lets 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 its 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 ....

3.2.4.2. Nuxeo 5 Platform development model

Nuxeo 5 development model is heavily based on the usage of extension points. When a project requires specific features we try as much of possible to include it as an extension of the existing framework rather than writing 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.

3.3. Nuxeo EP layered architecture

3.3.1. 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 handle 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 be 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 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 is 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 writes logs in DB via JPA.

- Audit View (UI Layer)

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

3.3.2. API and Packaging impacts

The layer organization can also be seen in the API.

3.3.2.1. 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).

3.3.2.2. Service Stack API

Each service exposes its 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.

3.3.2.3. 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.

3.3.2.4. Packaging

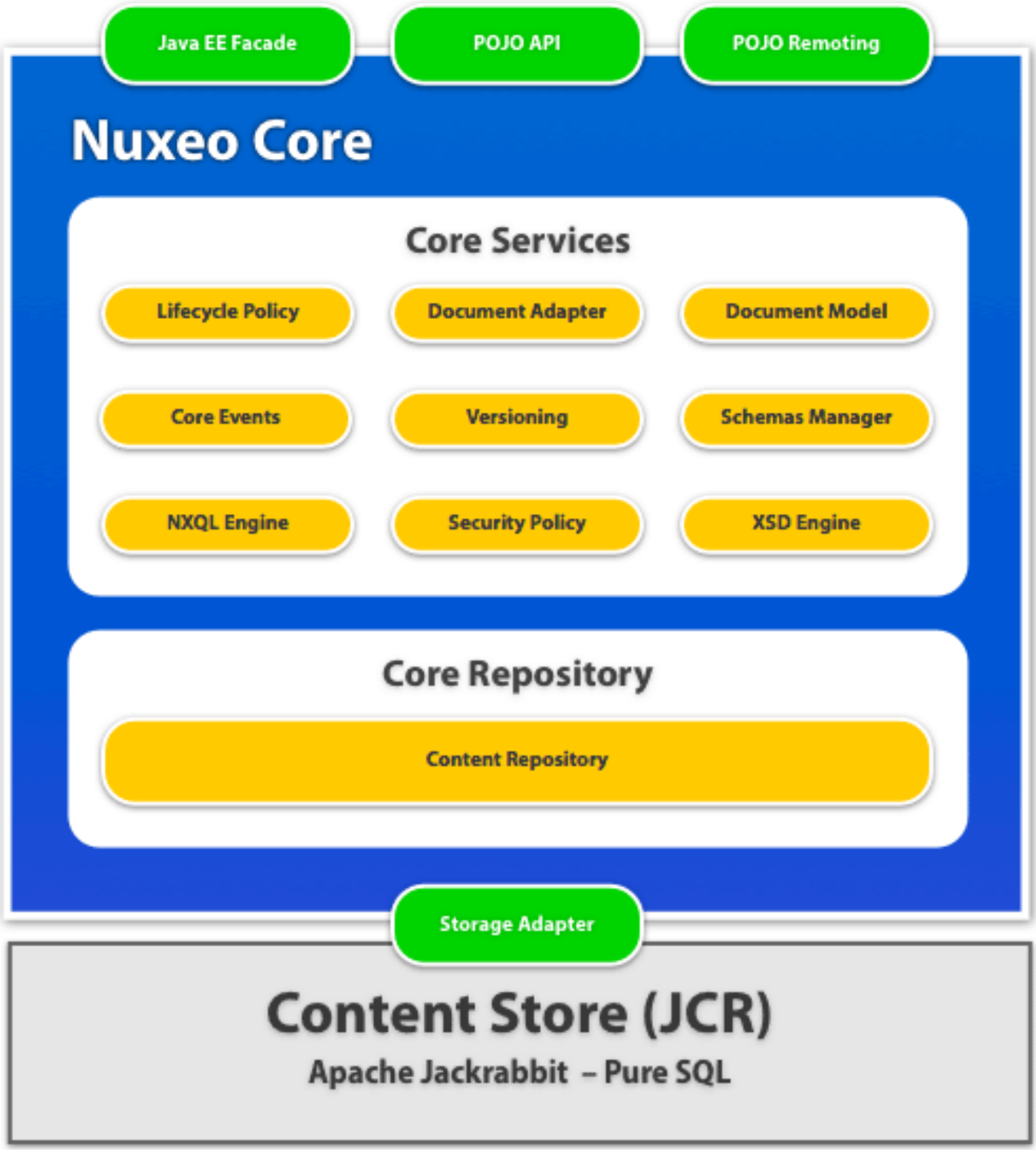
Most features are made of several Java project and generate several Maven 2 artifact.

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.

3.3.3. Illustration of the layered architecture

XXX TODO

3.4. Core Layer overview



3.4.1. Features of Nuxeo Core

Nuxeo core provides all the storage services for managing documents.

- Schema service

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

- Repository service

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

- Versioning service

Lets you configure how to store versions.

- Security service

  Manages data level security checks

- Lifecycle service

  Manages life cycle state of each document

3.4.2. Nuxeo Core main modules

3.4.2.1. Nuxeo Repository Service

The repository service lets 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)

3.4.3. 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.

3.4.4. 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 possible 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 contributing life-cycle definition

  This includes states and transitions. On 5.2, since 5.2.0, it is possible to define additional initial states to the default state, by adding a keyword to the state definition, for instance: <state name="approved" description="Content has been validated" initial="true">. The desired initial state can be passed in the document model context data used for creation: document.putContextData("initialLifecycleState", "approved").

```
<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>
```

- 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>
```

Life-Cycle service is detailed later in this document.

3.4.5. 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 lets 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.

3.4.6. 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 dublincore 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.

3.4.7. 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.

3.4.8. Versioning system

The documents in the repository can be versionned.

Nuxeo Core provides:

- A pluggable version storage manager

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

- A pluggable versionning policy

  This lets 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.

3.4.9. 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.

3.4.10. 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 support an 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 embed 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)

3.4.11. 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:

- its own security descriptors
- its own lifecycle information
- its own DocumentRef

3.4.12. Core API

3.5. Service Layer overview

3.5.1. 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.

3.5.2. 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.

3.5.3. Platform API

Each service provides its 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.

3.5.4. 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.

3.5.5. Some examples of Nuxeo EP services

3.6. Web presentation layer overview

3.6.1. Technology choices

3.6.1.1. 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

3.6.1.2. 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 JavaScript
- A built-in event system

3.6.2. Componentized web application

3.6.2.1. 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 useful

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

- 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

3.6.2.2. 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 its 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.

3.6.2.3. 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.

3.6.2.4. 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 its 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.

3.6.2.5. Key web framework services

Part II. Administration

You will learn here how to setup and manage a Nuxeo EP server in a production context (as opposed to a demo / evaluation or development context).

Chapter 4. OS requirements, existing and recommended configuration

This chapter presents information about the running environment. Listing all required software, giving a recommended configuration and listing some others, known as operational, this chapter aims to help you to validate or define your production environment but the list is not exhaustive and needs to be completed with the users' experience.

4.1. Required software

- JBoss application server with EJB3 support enabled
- Java Development Kit (JDK)

4.2. Recommended configuration

4.2.1. Hardware configuration

Nuxeo EP is designed to be scalable and thus to be deployed on many servers. It can be installed on only one server for a start, and can also easily be installed on many servers. The constant is that there is the need to have one high-end server with good performances. Then the other servers can be more lower-end.

So the numbers below are given for the one needed high-end server.

- RAM: 2Gb is the minimum requirement for using Nuxeo EP
- CPU: Intel Core2 or equivalent and upper

  You might be better avoiding machines from the Intel Pentium 4 Xeon series since some models have a too small amount of cache. This impairs performance greatly compared to other CPU architecture of the same generation. Intel Pentium 4 servers are quite widespread because of an attractive price politic.
- Storage (disk) space: the minimum Nuxeo installation, along with the needed JBoss and libs, takes something between 200Mb and 250Mb on a filesystem. Then the final size will of course depend on the amount of data that will be stored in Nuxeo. A safe bet (until we provide better numbers) is to consider data space ratio of 1.5 to 2.

4.2.2. Default configuration

The default persistence configuration is lightweight and easy to use, but it is not made for performance.

- default Nuxeo 5.1 uses:
  - HSQL for SQL Data (directories, JBPM, Relations ...)
  - FileSystem persistence for Document repository
- default Nuxeo 5.2 uses:
  - Derby for SQL Data (directories, JBPM, Relations ...)
  - FileSystem persistence for Document repository

4.2.3. For optimal performances

- Linux
- PostgreSQL 8.2

  Use PostgreSQL for document repository and all other services except for Compass search engine (nxsearch-compass-ds.xml) which should be set to use filesystem.

  Configure the document repository to externalize the blobs to filesystem.

4.3. Known working configurations

4.3.1. OS

- Debian GNU/Linux 5.0 Lenny
- Linux Ubuntu 32 and 64 bits, Edgy, Feisty and Hardy (8.04).
- Linux Mandriva 2008.1
- Unix
- Mac OS X 10.4, 10.5
- Ms Windows 2003 server 32 and 64 bits, Windows XP

4.3.2. JVM

- Sun JDK 1.5 update 14, 15, 16

4.3.3. Storage backends

Different backends may be set as well for Nuxeo Core repository as for all other nuxeo services that persist data. See Chapter 7, *RDBMS Storage and Database Configuration* for more details, here is a list of known working backends.

- PostgreSQL 8.2
- PostgreSQL 8.3

  This version need a workaround to be applied as it is much stricter than PostgreSQL 8.2 with respect to value casting.

  Execute the following commands in your PostgreSQL console:

```
CREATE FUNCTION pg_catalog.text(integer) RETURNS text STRICT IMMUTABLE LANGUAGE SQL AS 'SELECT textin(int4out($1
CREATE CAST (integer AS text) WITH FUNCTION pg_catalog.text(integer) AS IMPLICIT;
COMMENT ON FUNCTION pg_catalog.text(integer) IS 'convert integer to text';

CREATE FUNCTION pg_catalog.text(bigint) RETURNS text STRICT IMMUTABLE LANGUAGE SQL AS 'SELECT textin(int8out($1)
CREATE CAST (bigint AS text) WITH FUNCTION pg_catalog.text(bigint) AS IMPLICIT;
COMMENT ON FUNCTION pg_catalog.text(bigint) IS 'convert bigint to text';
```

  See FAQ about using PostgreSQL 8.3.

- MySQL
- Oracle 10
- MsSQL 2005

- HSQL

  Default Nuxeo 5.1 embedded database.

- Derby

  Default Nuxeo 5.2 embedded database.

4.3.4. LDAP

- OpenLDAP
- Ms Active directory

Chapter 5. Log configuration

The log levels and log outputs in Nuxeo are controlled at the server application level. In JBoss this is done through Log4j. The configuration file is `$JBOSS_HOME/server/default/conf/log4j.xml`.

Changing log level (or log threshold) can be done like shown below for each appender.

```
<param name="Threshold" value="INFO"/>
<param name="Threshold" value="DEBUG"/>
```

Administrators are expected to rotate and/or compress logs so that they do not fill up all a partition free space and so that they do not reach maximum file size. Log4j provides rolling/rotating capabilities. External programs such as logrotate can also be used.

Chapter 6. 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.

Chapter 7. 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).

7.1. Storages in Nuxeo EP

Nuxeo EP manages several types of data: Documents, relations, audit trail, users, groups ...

Each type of data has its own storage engine and can be configured separately. 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 several 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.

7.2. Installing 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

7.3. Configuring Nuxeo Core Storage

Nuxeo Core stores data for the documents themselves: the hierarchy of documents, their metadata and security, and the attached files. There are two main Nuxeo Core Storage backends: the recent (available in Nuxeo 5.2) Visible Contents Store (VCS), based on a mapper to native RDBMS tables, and the previous JCR-based backend, using Jackrabbit.

This section will show you how to configure each backend, using PostgreSQL 8.3 as an example underlying storage. The setup for other RDBMS should be very similar.

7.3.1. Visible Content Store configuration

To set up VCS, you first need create a datasource for it. The datasource is *not* a standard JDBC datasource, so

has different syntax, even though it contains information about JDBC connection parameters. This file is usually named `$JBOSS_HOME/server/default/deploy/nuxeo.ear/datasources/default-repository-ds.xml`.

Example 7.1. Datasource for VCS using PostgreSQL

```
<?xml version="1.0"?>
<connection-factories>
  <tx-connection-factory>
    <jndi-name>NXRepository/default</jndi-name>
    <xa-transaction/>
    <track-connection-by-tx/>
    <adapter-display-name>Nuxeo SQL Repository DataSource</adapter-display-name>
    <rar-name>nuxeo.ear#nuxeo-core-storage-sql-ra-1.5-SNAPSHOT.rar</rar-name>
    <connection-definition>org.nuxeo.ecm.core.storage.sql.Repository</connection-definition>
    <config-property name="name">default</config-property>
    <config-property name="xaDataSource" type="java.lang.String">org.postgresql.xa.PGXADataSource</config-proper
    <config-property name="property" type="java.lang.String">ServerName=localhost</config-property>
    <config-property name="property" type="java.lang.String">PortNumber/Integer=5432</config-property>
    <config-property name="property" type="java.lang.String">DatabaseName=nuxeo</config-property>
    <config-property name="property" type="java.lang.String">User=nuxeo</config-property>
    <config-property name="property" type="java.lang.String">Password=password</config-property>
    <max-pool-size>20</max-pool-size>
  </tx-connection-factory>
</connection-factories>
```

You will then need to specify the actual repository configuration, usually store in a file named `$JBOSS_HOME/server/default/deploy/nuxeo.ear/config/default-repository-config.xml`

Example 7.2. Repository Configuration for VCS

```
<?xml version="1.0"?>
<component name="default-repository-config">
  <extension target="org.nuxeo.ecm.core.repository.RepositoryService" point="repository">
    <repository name="default"
      factory="org.nuxeo.ecm.core.storage.sql.coremodel.SQLRepositoryFactory">
      <repository name="default">
        <indexing>
          <!-- example configuration for H2
          <fulltext analyzer="org.apache.lucene.analysis.fr.FrenchAnalyzer"/>
          -->
          <!-- example configuration for PostgreSQL
          <fulltext analyzer="french"/>
          -->
          <!-- example configuration for Microsoft SQL Server
          <fulltext catalog="nuxeo" analyzer="french"/>
          -->
        </indexing>
        <!-- uncomment this to enable clustering
             delay is in milliseconds
               default delay is 0 (no delay before processing invalidations)
        <clustering enabled="true" delay="1000" />
        -->
      </repository>
    </repository>
  </extension>
</component>
```

7.3.2. JCR backend configuration

First you have to specify a datasource in `$JBOSS_HOME/server/default/deploy/nuxeo.ear/datasources/default-repository-ds.xml`.

Example 7.3. Datasource for JCR backend

```
<?xml version="1.0"?>
<!DOCTYPE connection-factories PUBLIC
  "-//JBoss//DTD JBOSS JCA Config 1.5//EN"
```

```
  "http://www.jboss.org/j2ee/dtd/jboss-ds_1_5.dtd">
<connection-factories>
  <mbean code="org.nuxeo.ecm.core.repository.JBossRepository"
         name="nx:type=repository,name=default">
    <constructor>
      <arg type="java.lang.String" value="default"/>
    </constructor>
  </mbean>
  <tx-connection-factory>
    <jndi-name>NXRepository/default</jndi-name>
    <adapter-display-name>NX Repository Adapter</adapter-display-name>
    <rar-name>nuxeo.ear#nuxeo-core-jca-${project.version}.rar</rar-name>
    <connection-definition>org.nuxeo.ecm.core.model.Repository</connection-definition>
    <xa-transaction/>
    <!-- Configuration properties. -->
    <config-property name="name">default</config-property>
  </tx-connection-factory>
</connection-factories>
```

Then edit `$JBOSS_HOME/server/default/deploy/nuxeo.ear/config/default-repository-config.xml`.

Example 7.4. 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.

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 improvement 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 7.5. Sample configuration for a PostgreSQL Jackrabbit repository

```
        <!-- Workspaces configuration. Nuxeo only uses the default workspace. -->
        (...)
          <PersistenceManager class="org.apache.jackrabbit.core.persistence.bundle.PostgreSQLPersistenceManager"
            <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.persistence.bundle.PostgreSQLPersistenceManager"
    <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 7.6. Sample configuration for a MySQL Jackrabbit repository

```
<!-- Workspaces configuration. Nuxeo only uses the default workspace. -->
(...)
  <PersistenceManager class="org.apache.jackrabbit.core.persistence.bundle.MySqlPersistenceManager">
    <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.persistence.bundle.MySqlPersistenceManager">
    <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>
```

Note: "schemaObjectPrefix" must have different values in workspace & versioning configuration

For JackRabbit, there are some persistence manager specific to each 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 may use : org.apache.jackrabbit.core.persistence.bundle.MSSqlPersistenceManager

7.3.3. Set up your RDBMS

Create the database in the database server, enable IP connection, setup permissions and test the connection.

7.3.4. Start Nuxeo EP

You can now start JBoss AS and verify that your new repository is used!

7.4. Configuring 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.

7.4.1. Configuring datasources

Each service can use a dedicated datasource to define the database connection. However, to simplify configuration, all datasources are JNDI NamingAlias pointing to a single datasource (NuxeoDS). If you want to change the default database, you can simply do the following:

- deploy the needed JDBC Driver in `$JBOSS_HOME/server/default/lib`,
- modify the datasource definition file in `$JBOSS_HOME/server/default/deploy/nuxeo.ear/datasources/unified-nuxeo-ds.xml`.

The default configuration has a commented out PostgresSQL configuration.

To use a dedicated datasource for a service, you need to modify its configuration file. If you would like to store audit logs in PostgreSQL using its own datasource, remove the NamingAlias in `nxaudit-logs-ds.xml` and replace it with the datasource configuration example:

Example 7.7. 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 7.8. 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`

7.4.1.1. Special MySQL needs

MySQL is a quirky database which sometimes needs very specific options to function properly.

The datasources used 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 used 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 `&` syntax for the URL parameters instead of just `&` because the URL is embedded in an XML file.

7.4.2. 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.

7.4.3. Compass search engine dialect configuration

Note that this section is obsolete in Nuxeo 5.2, and should not be used anymore.

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/system/` 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 .

7.5. 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.

7.5.1. 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 `BundleDbPersistenceManager`.

Example 7.9. 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="${wsp.home}"/>
          </FileSystem>
          <PersistenceManager class="org.apache.jackrabbit.core.persistence.bundle.PostgreSQLPersistenceManager"
            <param name="driver" value="org.postgresql.Driver"/>
            <param name="url" value="jdbc:postgresql://localhost/nuxeo"/>
            <param name="user" value="postgres"/>
            <param name="password" value="password"/>
            <param name="schema" value="postgresql"/>
            <param name="schemaObjectPrefix" value="jcr_${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.persistence.bundle.PostgreSQLPersistenceManager"
            <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>
```

7.5.2. 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>
```

Chapter 8. Data Migration

This chapter describes how to modify existing document types.

8.1. Summary of scenarios

Five different situations can happen when you want to make evolve your data models.

- adding a new schema to a document type
- removing a schema from a document type definition
- adding a new field to an existing schema
- removing a field from a schema
- changing field type (from integer to string for example)

8.1.1. Table of situations

This table sums up how data migration are handled : YES if the operation is supported, NO if not and into brackets, reference to the relative procedure.

Action / Backend	JackRabbit on FileSystem	JackRabbit on SQL	SQL Storage
Add a schema	YES (1.2.1)	YES (1.3)	YES (1.4.1)
Remove a schema	NO (1.2.2)	YES (1.3)	YES (1.4.2)
Add a field	YES (1.2.1)	YES (1.3)	YES (1.4.2)
Remove a field	NO (1.2.2)	YES (1.3)	YES (1.4.2)
Modify field type	NO (1.2.3)	YES (1.3)	YES (1.4.3)

8.1.2. Procedures with JCR Storage on filesystem

8.1.2.1. Adding field / schema

When adding a schema to a document type or a field to a schema, you need to rebuild JCR nodetypes which are defined in `nodetypes/custom_nodetypes.xml`. The force reloading is done by setting the `forceReloadTypes` attribute in `$JBOSS_HOME/server/default/deploy/nuxeo.ear/config/default-repository-config.xml` to `true`:

```
<repository name="default"
  factory="org.nuxeo.ecm.core.repository.jcr.JCRRepositoryFactory"
  securityManager="org.nuxeo.ecm.core.repository.jcr.JCRSecurityManager"
  forceReloadTypes="true">

   ...

</repository>
```

Keep in mind that this setting could not be applied in a production environment.

8.1.2.2. Removing field /schema

Removing an attribute or a schema is not supported by the JCR backend. When JCR gets a nodetype related to

a document, it tries to give a type to each properties. If a field or a schema is removed, JackRabbit cannot associate a type to the node and it raises the following error:

```
Exception: javax.jcr.nodetype.ConstraintViolationException. message: no matching property definition found for M
```

If you do not want to use a schema or a field any more, keep them in your doctype / schema definition and never use them in your interface.

8.1.2.3. Modifying a field type

Creating new documents with new field type definition is possible.

However all previously created documents will not be modifiable.anymore: if you try to save your modifications, you will get a `NullPointerException` caused by JCR which will not be able to retrieve the field with the new given type. At reading, all document attributes from the schema containing the modified field will be empty.

8.1.3. Procedures with JCR Storage on SQL

These use cases were tested on JCR Storage with PostgreSQL. for data persistence.

All manipulations on document structure are supported without changing any configuration file.

8.1.4. Procedure with SQL Storage

8.1.4.1. Adding a schema

In the SQL storage, adding a schema is natively supported. No extra operation is needed to take into account the new schema.

8.1.4.2. Adding a field, removing field /schema

Adding a new field, removing a field or a schema are supported operations in the SQL storage. They need to modify the database with SQL queries (Alter ...)

8.1.4.3. Modifying a field type

Modifying a field type is also supported in the SL storage: this operation needs some SQL commands to be effective:

- first, create a new column with the desired type
- then, populate your column by casting value from previous type to new one
- remove old column
- finally, alter table to rename new column with field name

Chapter 9. LDAP Integration

9.1. 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.

Chapter 10. OpenOffice.org server installation

OpenOffice.org (OOo) is used on server-side for different tasks such as file transforms (eg. to PDF) or other advanced tasks.

10.1. Installation

See http://www.openoffice.org/ for installation procedure if not provided by your OS.

10.1.1. Start server

Since version 2.3, OpenOffice can be started in headless mode. This means that under linux, you non longer need to run a Xvfb.

Use the oooctl control script or, depending on your system and installation method, start OOo running :

- for Linux:

```
/path/to/openoffice/program/soffice.bin
            -headless -nofirststartwizard
            -accept="socket,host=localhost,port=8100;urp;StarOffice.Service"
```

- for Mac OS X:

```
/path/to/openoffice.app/Contents/MacOS/soffice.bin
            -headless -nofirststartwizard
            -accept="socket,host=localhost,port=8100;urp;StarOffice.Service"
```

- for Windows:

```
soffice.exe -headless
            -nofirststartwizard
            -accept="socket,host=localhost,port=8100;urp;StarOffice.Service"
```

10.1.2. Parameters

10.1.2.1. `-headless`

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.

10.1.2.2. `-nofirststartwizard`

The `-nofirststartwizard` skips any post-install wizard, allowing to use OOo directly after the installation without any user parameterization.

Install nxSkipInstallWizard.oxt extension to make this parameter permanent.

10.1.2.3. `-accept`

The `UNO` protocol implies that OOo is opened in listening mode on a network interface. This parameter let OOo listen on the right interface and port.

Install nxOOoAutoListen.oxt extension to set this listening permanent.

Alternate method: this could also be done by adding this connection info in OOo `Setup.xcu` configuration file.

```
<node oor:name="Office">
  <prop oor:name="ooSetupConnectionURL">
    <value>socket,host=localhost,port=8100;urp;StarOffice.Service</value>
  </prop>
</node>
```

10.1.3. Installing an extension

Nuxeo has developed some tools to ease the settings, they are available at the Nuxeo svn tools ooo section as extensions to install in OpenOffice. This can be done thru the GUI via the menu "Tools>Extensions Manager>Add..." or by opening the wanted extension with OpenOffice. To do it from a command line, run:

```
/path/to/openoffice/program/unopkg add extension.oxt
```

10.1.4. Notes

10.1.4.1. Multi-machine

On multi-machine deployment, we recommend to install OOo on the server running the webapp (ie stateless server on a bi-machine "stateless/stateful" installation).

10.1.4.2. OpenOffice lower than 2.3.x

For OOo versions lower than 2.3, a graphical interface is needed. 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
export DISPLAY=":77.0"
```

```
xvfb-run -a /path/to/openoffice/program/soffice.bin
        -headless -nofirststartwizard
        -accept="socket,host=localhost,port=8100;urp;StarOffice.Service"
```

10.1.4.3. 32/64 bits

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.

10.2. Running OpenOffice as a Daemon

10.2.1. Nuxeo OOo Daemon

Starting with 5.2-M4, Nuxeo includes a Daemon to start and manage OpenOffice server.

This daemon is based on the OpenOffice Server Daemon project

OOo Daemon provides :

- OpenOffice server automatic startup

Depending on configuration the OOo Server may be started automatically when Nuxeo starts or on first call.

- OpenOffice server instances pooling

  Daemon can manage several OpenOffice instance (workers) and distribute tasks across them. In order to avoid leaks, OOo workers are recycled.

10.2.2. Configuring Nuxeo OOo Daemon

Configuration is done via an xml extension point. Defaut config can be editer in file located in `nuxeo.ear/config/ooo-config.xml`.

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

  <extension target="org.nuxeo.ecm.platform.convert.oooserver.OOoDaemonManagerComponent"
    point="oooServerConfig">

  <OOoServer>
    <!-- enable Nuxeo Daemon to manage OOo server instances : default is true -->
    <enableDaemon>true</enableDaemon>
    <!-- define OOo server listen IP : used even if daemon is disabled -->
    <oooListenIP>127.0.0.1</oooListenIP>
    <!-- define OOo server listen port : used even if daemon is disabled -->
    <oooListenPort>8100</oooListenPort>
    <!-- define Daemon listen port : used only if daemon is enabled -->
    <oooDaemonListenPort>8101</oooDaemonListenPort>
    <!-- define number of OOo worker process : used only if daemon is enabled -->
    <oooWorkers>1</oooWorkers>

    <!-- define OOo installation path : used only if daemon is enabled -->
    <!-- if not defined Nuxeo will try to find the path automatically -->
    <!--<oooInstallationPath>/usr/lib/openoffice/program</oooInstallationPath>-->

    <!-- define jpipe library path : used only for OOo 3 -->
    <!--<jpipeLibPath>/opt/openoffice.org/ure/lib</jpipeLibPath>-->

    <!-- define number of time a worker process can be used before being recycled: used only if daemon is enable
    <oooWorkersRecycleInterval>10</oooWorkersRecycleInterval>

    <!-- define is Daemon is started at server startup : used only if daemon is enabled -->
    <autoStart>false</autoStart>

  </OOoServer>

  </extension>

</component>
```

In most cases, you should not have to define the location of your OpenOffice installation as the Daemon will try to find it in the standard installation path.

If you want to use OpenOffice 3.x, you have to specify the jpipeLibPath in order to enable jpipe that is used by the Daemon to communicate with OpenOffice workers.

Chapter 11. Automatic starting and stopping of JBoss

The procedure described here is targeted for the Debian Etch distribution, and should be valid for any Debian-based GNU/Linux distribution such as Ubuntu. In other GNU/Linux distributions some commands may be different.

Create the following shell script /etc/init.d/nuxeo:

```
#!/bin/sh
JAVA_HOME=/usr/lib/jvm/java-1.5.0-sun
export JAVA_HOME
JBOSS_HOME=/usr/local/nuxeo-ep-5.2
$JBOSS_HOME/bin/jbossctl $@
```

Then enable the autostart creating the links in the rcX.d directories running the command (as root):

```
$ update-rc.d nuxeo defaults
```

Output:

```
Adding system startup for /etc/init.d/nuxeo5 ...
/etc/rc0.d/K20nuxeo -> ../init.d/nuxeo
/etc/rc1.d/K20nuxeo -> ../init.d/nuxeo
/etc/rc6.d/K20nuxeo -> ../init.d/nuxeo
/etc/rc2.d/S20nuxeo -> ../init.d/nuxeo
/etc/rc3.d/S20nuxeo -> ../init.d/nuxeo
/etc/rc4.d/S20nuxeo -> ../init.d/nuxeo
/etc/rc5.d/S20nuxeo -> ../init.d/nuxeo
```

Now restart the machine and verify that nuxeo is started automatically looking at the log file.

If you want to remove the automatic startup use the command (as root):

```
$ update-rc.d -f nuxeo remove
```

Output:

```
Removing any system startup links for /etc/init.d/nuxeo ...
/etc/rc0.d/K20nuxeo
/etc/rc1.d/K20nuxeo
/etc/rc2.d/S20nuxeo
/etc/rc3.d/S20nuxeo
/etc/rc4.d/S20nuxeo
/etc/rc5.d/S20nuxeo
/etc/rc6.d/K20nuxeo
```

Chapter 12. 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 `$JBOSS_HOME/server/default/deploy/nuxeo.ear/config/nuxeo.properties` change the value of `org.nuxeo.ecm.instance.host`, replace "localhost" by the IP address of the JBoss server,
- in `$JBOSS_HOME/server/default/deploy/nuxeo.ear/config/`platform-config.xml

  mono-server: ${org.nuxeo.ecm.instance.host} is used to reference address set in nuxeo.properties

  multi-server: either use IP or set DNS aliases in your hosts file (default names in Nuxeo packages are usually something like nxcoreserver, nxsearchserver, nxplatformserver, nxjmsserver, nxwebserver, nxdbserver).
- in `$JBOSS_HOME/server/default/deploy/nuxeo.ear/config/datasources/core-events-ds.xml` replace, if enabled, the 127.0.0.1 value of `java.naming.provider.url` by the IP address of the Jboss server,
- start jboss with the -b option:

  ./run.sh -b server_IP_address

IP behavior change between JBoss 4.0.x and JBoss 4.2.x

When no IP is specified, JBoss 4.0 is bound to listening on any address, not only on localhost.

For obvious security reasons, JBoss 4.2 (version required by Nuxeo EP 5.2) has a different behavior. If you still want this, use **-b 0.0.0.0**

For a server in production, see SecureJBoss.

Chapter 13. Virtual Hosting

The Nuxeo webapp can be virtual hosted behind a HTTP/HTTPS reverse proxy, like Apache.

13.1. 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.

13.2. Virtual hosting configuration for Apache 2.x

13.2.1. 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/"
```

This will require you to load and activate `mod_header` module.

13.2.2. 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.

13.2.3. 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.

Chapter 14. Firewall configuration, open ports

Port translation

See Using a non-standard IP address and ports about port translation, this chapter focuses on the ports list to open on a mono or multi-server installation.

14.1. Standard Nuxeo-EP

Table 14.1. Default ports used by Nuxeo

Type	Protocol	Port
Web	HTTP	8080
EJB3 calls	RMI	4444 / 4445
		3873 / 1098
JNDI	JNDI	1099
Runtime remoting et streaming	TCP	62474
		3233 (no more used since Nuxeo 5.1.3.2)
JMS	TCP	8093
LDAP (if used)	LDAP	389
DBMS (if used)	JDBC	5432 for PostgreSQL
JODConverter	URP	8100

14.2. Bi-machine "Stateful/Stateless"

As said before, the bi-machine configuration allow to duplicate the stateless server. All duplicated stateless servers will need to access some ports on the stateful server, while the stateful server need only access to at least one stateless server.

Chapter 15. Backup, restore and reset

Note

For server migration (moving a Nuxeo instance from a server to another), follow the backup procedure on the source server and then the restore procedure on the destination server.

Note

Nuxeo-shell may be used for import/export purpose but it's not the same as the "system backup" described here.

15.1. Backup

Important

It is highly recommended to stop nuxeo during backup.

If you use a database, you may switch it in a "hot backup" mode to ensure that all new incoming requests will wait the end of the backup process.

Backup these directories and files:

- Nuxeo libraries deployed in JBoss as they are specific to the running Nuxeo EP version

  `$JBOSS/server/default/lib/nuxeo*`

- Nuxeo EP EAR (Entreprise ARchive)

  `$JBOSS/server/default/deploy/nuxeo.ear/`

- data stored on filesystem

  `$JBOSS/server/default/data/`

- it could be useful to backup logs

  `$JBOSS/server/default/log/`

Backup your database(s) if you use one (some).

Finally, backup any file you have customized. Here is a short list of such files :

- if you changed JVM startup parameters (run.bat, run.conf) or use Nuxeo's startup script (jbossctl, jbossctl.conf, bind.conf)

  `$JBOSS/server/bin`

- if you changed the logging levels (log4j.xml)

  `$JBOSS/server/default/conf`

- if you added some .jar files for specific JDBC drivers

  `$JBOSS/server/default/lib`

- if you changed JBoss configuration (like mail-service.xml)

```
$JBOSS/server/default/deploy
```

15.2. Backup before an upgrade

If you plan an upgrade, you may backup separately the configuration files in order to easily apply again your configuration on the default one (take care not to loose any evolution on these files):

- Main configuration is in `$JBOSS/server/default/deploy/nuxeo.ear/config`
- Datasources are defined in `$JBOSS/server/default/deploy/nuxeo.ear/datasources`
- Compass backend is configured in `$JBOSS/server/default/deploy/nuxeo.ear/system/nuxeo-platform-search-compass-plugin*/compass.cfg.xm`

15.3. Restore

All you need is to restore previously saved database(s), files and directories.

15.4. Reset

You can simply reset Nuxeo by removing its data: delete `$JBOSS/server/default/data/` and empty all used databases. On start, Nuxeo EP recreates all its tables, files and directories.

Chapter 16. High availability, fail-over, clustering, replication and load balancing solutions

Here are some sample configurations. There are many other possible options.

16.1. Introduction

For such purpose, we will deploy Nuxeo on multiple servers. Each Nuxeo server is a specific part of a standard Nuxeo server (called "mono-machine", versus "multi-machine"). We currently recommend to use nuxeo-2parts packaging with a "stateless" and a "stateful" server (previous recommended packaging was nuxeo-3parts with "core", "search" and "web" servers).

Stateless server may be duplicated as many times you need, allowing **load-balancing / clustering** between front instances.

Stateful server must be unique. It can be replicated to ensure **fail-over**. High availability will depend on the replication solution and time required to switch from a dead server to the replicated one.

High Availability is the result of mixing replication, load-balancing, and optimization. Time window for loosing data or unavailable services will depend on the replication, load-balancing and JMS configuration and solution chosen.

16.2. Fail-over based on PostgreSQL replication (warm-standby mode)

Using native PostgreSQL replication called "Warm Standby Using Point-In-Time Recovery (PITR)" to set up a primary-standby configuration. See http://www.postgresql.org/docs/8.3/interactive/warm-standby.html

In this configuration, there are, at least, two servers: a primary and a standby.

- The primary server runs normally, exporting its data to make them available for the standby.
- The standby server runs as read-only, being permanently synchronized with the primary. There could be multiple standby servers.

If the primary server fails, then:

- The standby server is awaken and starts its fail-over procedure to become the primary server.
- The primary can now be repaired and set up as standby, or we can set up a new standby server.
- If the primary server fails and then immediately restarts, it must be switched to standby mode if the standby one has started its failover procedure. A script can be done to watch the primary and inform the standby when it has to switch mode.
- While there is not a new primary and a standby up, we are in a degenerated state.
- Standby became primary, a new standby server has to be set, either on the down primary after restart or on a new system.
- Once we have running primary and standby servers, we can consider that we've completed a fail-over procedure, switching primary and standby roles.

As said on PostgreSQL documentation: [so, switching from primary to standby server can be fast but requires some time to re-prepare the failover cluster. Regular switching from primary to standby is useful, since it allows regular downtime on each system for maintenance. This also serves as a test of the fail-over mechanism to ensure that it will really work when you need it. Written administration procedures are advised.]

16.2.1. Install PostgreSQL contribution pg_standby

Download and compile pg_standby .

For example, from a tgz postgresql 8.2.6 installation, do :

```
cd postgresql-8.2.6/contrib
mkdir pg_standby
cd pg_standby
wget https://projects.commandprompt.com/public/pgsql/repo/trunk/pgsql/contrib/pg_standby/Makefile
wget https://projects.commandprompt.com/public/pgsql/repo/trunk/pgsql/contrib/pg_standby/pg_standby.c
cd ../..
./configure
cd -
make
make install
```

This builds a **pg_standby** binary that is used on standby server. Alternative scripts may be written to replace it.

16.2.2. Using Nuxeo tools

Download the Nuxeo replication tools on each server: http://svn.nuxeo.org/nuxeo/tools/replication/trunk .

Data from all datasources and data from the repository need to be synchronized between two Nuxeo instances. Except all that is stored depending on the datasources and the repository configuration, there are still JCR indexes to synchronize that are in:

```
$JBOSS/server/default/data/NXRuntime/repos
```

Be careful that this file that may contain server-specific data (i.e. its IP address or DNS alias) :

```
$JBOSS/server/default/data/NXRuntime/repos/default/default.xml
```

Copy and parameterize `init-var.sh.sample` to `init-var.sh`. This documentation assume you have set postgresql-8.2 and nuxeo-server services; if not, the scripts can do it (see "`make_symlinks`" in "`init-nuxeo.sh`" and "`init-system.sh`").

SSH between servers must be enabled and host keys verified (for postgres and jboss users). PostgreSQL process will have to su jboss user on both servers (primary and standby).

Here are the commands to execute at initialization :

1. On each server (first on primary, next on standby) :

```
# initialize PostgreSQL database
./failover.sh pg-init

service nuxeo-server start
# if it's a new Nuxeo instance, do login and some minimal tricks (navigating root workspace), then logout
# this will generate some basic data
```

2. On primary server :

```
./failover.sh backup
./failover.sh init-primary
```

3. On standby server :

```
service nuxeo-server stop
./failover.sh restore
./failover.sh init-standby
```

In case of primary server failure (or to switch servers for maintenance or tests purpose), do:

1. On ex-standby (future primary) server :

```
./failover.sh switch-to-primary
```

2. On ex-primary (future standby) server, if it can be reboot (eventually after reparation):

```
./failover.sh switch-to-standby
```

3. If the ex-primary server is dead, a new standby server can be set:

```
# check standby URL set on primary in init-vars.sh
./failover.sh pg-init
service nuxeo-server start
# login, logout
service nuxeo-server stop
./failover.sh restore
./failover.sh init-standby
```

Chapter 17. The Nuxeo Shell

17.1. 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.

17.2. 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 specify the command that will be executed. After the command execution the process will exit. The command has to be passed as the first argument.

   Here is an example of a command executed in batch mode:

```
$ ./nxclient.sh export /path/to/remote/doc /path/to/local/folder
```

   Here is an example of a command executed in batch mode while also passing parameters:

```
$ ./nxclient.sh ls --host 192.168.2.54
```

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 in one of two ways:

```
$ ./nxclient.sh interactive
```

```
$ ./nxshell.sh
```

   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 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 programmatically using their index. In the example above, the first parameter will have the index 0 while the second the index 1.

17.2.1. Command Options

Command options are defined by a name and optionally a shortcut (a short name). When referring to an option using its 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 localhost
$ ./nxshell.sh interactive --host localhost
```

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 immediately 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.

17.2.1.1. 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
- debug (**--debug** or **-d**) to start with debug mode (logs at DEBUG level)

17.2.2. Commands

There is the list of all built-in commands of nuxeo shell.

Notes:

1. At the time of this writing some of these built-in commands are not yet implemented. Those are be marked with an asterisk (*).
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.

17.2.2.1. interactive

Runs in the interactive mode. This command is not available when already in interactive mode.

Has no specific options.

```
$ ./nxshell.sh interactive
```

17.2.2.2. 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
```

17.2.2.3. log

Manage logs from console without having to edit log4j.xml. Allow to switch on/off a debug mode, add/remove loggers, ...

log filename [log level [package or class]] creates a file (created in $PWD/log/ if **filename** do not contain path separator, else given path is used). It will record logs at level greater or equal to **level** (if specified, else default level is INFO; available values are TRACE, DEBUG, INFO, WARN, ERROR, FATAL). If a **package or a class** (canonical name) is specified, logs will be filtered from this package/class.

log off [package or class] stops logging. Applies on all custom appenders (console and filenames) if no **package or class** is specified. Don't worry about the warning **log off** command may cause (the logger is re-created):

```
log4j:WARN No appenders could be found for logger (org.nuxeo.ecm.shell.commands.InteractiveCommand).
log4j:WARN Please initialize the log4j system properly.
```

log debug switches debug mode on/off. Same result as starting the shell with **-d** (**--debug**) option. This decrease the log level to DEBUG on console and filenames.

See log4j documentation for more information.

Available only in *interactive* mode

17.2.2.4. connect *

Connects to a given server. If a connection is already established, close it first.

Available only in *interactive* mode

17.2.2.5. disconnect *

Disconnects from the current connected server. If no connection exists, does nothing.

Available only in *interactive* mode

17.2.2.6. ls

Lists the children of the current folder in the repository.

By default, Folderish types are displayed in blue.

Note that the **ls directory-name** command is accepted but won't list the content of `directory-name`, it will only list the content of the current directory. This might be improved in the future to provide a behavior alike the one of the Unix ls command.

Available only in *interactive* mode

17.2.2.7. tree

Displays the complete tree structure of the documents as it would be returned by the Unix **tree** command.

Available only in *interactive* mode

17.2.2.8. cd

Changes the current directory in the repository (to a Folderish document)

17.2.2.9. pwd

Displays the current path in the repository.

17.2.2.10. 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 document

17.2.2.11. rm

Removes a document or a tree of documents.

17.2.2.12. mkdir

Creates a Folder document.

17.2.2.13. put

Uploads a blob to a file document. If the target document doesn't exists, creates it.

17.2.2.14. putx

Creates a document other than a file. Metadata (and blobs) are specified in the Nuxeo export format.

17.2.2.15. get

Downloads the blob from a File document.

17.2.2.16. getx

Gets a document as an XML file (as export, but only for a document).

17.2.2.17. export

Exports documents from the repository.

17.2.2.18. import

Imports documents into the repository.

17.2.2.19. script

Makes it possible to run external scripts. Here are examples of such scripts.

From interactive mode use:

```
script --file <path_to_your_script> <args>
```

17.2.2.20. chperm *

Changes a privilege on the given document.

17.2.2.21. useradd

Creates new user(s).

To create user joe:

```
useradd joe
```

To create the users define in the users.csv CSV file:

```
useradd --file users.csv
```

17.2.2.22. groupadd *

Creates new group(s).

17.2.2.23. groupmod

Modifies a group

To add user joe to the company-service1 group:

```
groupmod --user joe company-service1
```

To set the users of the company-service1 group to be only joe

```
groupmod --set --user madarche company-service1
```

To add the users in the users_for_group.csv CSV file to the company-service1 group:

```
groupmod --file users_for_group.csv company-service1
```

To set the users of the company-service1 group to be the users define in the users_for_group.csv CSV file:

```
groupmod --file users_for_group.csv company-service1
```

17.2.2.24. select

Search the repository using the NXQL query language. For example:

```
select * from Document where ...
```

This command can be used as a starting point to write another script to perform search service queries if needed.

This command is only available in the Nuxeo 5.2 branch.

17.2.2.25. lstypes *

17.2.2.26. viewtype *

17.2.2.27. lsusers *

17.2.2.28. lsgroups *

17.2.2.29. viewuser *

17.2.2.30. viewgroup *

17.3. Troubleshooting

17.3.1. Check listened IP

Be sure on what precise IP address and what precise port the server is listening to. To find out, use the **netstat** command on the server host. The example below shows that, on the same host, there are 2 servers listening on the 62474 port but on different IP addresses. You should always use as arguments to the nuxeo shell the exact IP address and port you find out.

```
$ sudo netstat -ntlap | grep 62474
tcp        0      0 ::ffff:192.0.0.11:62474     :::*                        LISTEN      3623/java
tcp        0      0 ::ffff:192.0.0.10:62474     :::*                        LISTEN      1346/java
```

Alternatively you can use the **lsof** command.

```
$ sudo lsof -i :62474
COMMAND   PID       USER   FD   TYPE    DEVICE SIZE/OFF NODE NAME
java     3623      jboss  266u  IPv6 0x5d69a70      0t0  TCP [::192.0.0.11]:62474 (LISTEN)
java     1346      jboss  266u  IPv6 0x5d7af30      0t0  TCP [::192.0.0.10]:62474 (LISTEN)
```

17.3.2. Check connected server

Be sure to be connected on the right server. To do so, issue a **view** command. This will display information on the current context:

```
192.0.0.10> view
------------------------------------------------------------------------
UID: 2a13db70-f133-473c-9a90-6838d01610aa
Path: /
Type: Root
------------------------------------------------------------------------
Title: 2a13db70-f133-473c-9a90-6838d01610aa
Author: null
Created: null
Last Modified: null
------------------------------------------------------------------------
Description: 2a13db70-f133-473c-9a90-6838d01610aa
------------------------------------------------------------------------
```

17.3.3. Multi-machine case

When working with multi-machine installation, connect the Nuxeo Shell on the server running Nuxeo Core (i.e. the stateful server on a bi-machine configuration).

17.4. Extending the shell

If you a need a new command (a new functionality) not provided by the Nuxeo Shell, you can add it very simply by writing a Nuxeo plugin in exactly the same manner you would for a Nuxeo server instance. This is done by writing a Java class for each new shell command and declaring (that is registering) each command in a single XML file as a contribution to an extension point.

Here is an example of how to register two imaginary new custom commands **addapplicants** and **purgeobsoletedocs**.

17.4.1. Registering New Custom Commands

This is how to register the new commands:

```
<?xml version="1.0"?>
<component name="com.company.nuxeo.shellcommands" version="1.0">

  <documentation>
    Extra Nuxeo Shell commands shown as examples for the Nuxeo Book.
  </documentation>

  <extension target="org.nuxeo.ecm.shell.CommandLineService" point="commands">

    <command name="purgeobsoletedocs" class="com.company.nuxeo.shellcommands.ObsoleteDocsPurgeCommand">
      <description>
        Purge obsolete documents considering based on predefined hard-coded logic.
      </description>
      <help>Purge obsolete documents</help>
    </command>

    <command name="addapplicants" class="com.company.nuxeo.shellcommands.ApplicantsAdderCommand">
      <description>
        Adds applicants by creating the right folders with the right permissions.
      </description>
      <params>
        <param index="0" type="path" />
      </params>
      <help>Adds applicants</help>
    </command>

  </extension>

</component>
```

17.4.2. Java Code for the new commands

This is how to write a Java class for a new command:

```
public class ApplicantsAdderCommand extends AbstractCommand {

    public static final String COMMAND_NAME = "addApplicants";

    public void run(CommandLine cmdLine) throws Exception {
        String[] elements = cmdLine.getParameters();
        // parse command line
        if (elements.length != 1) {
            log.error("Usage: " + COMMAND_NAME + " file");
            return;
        }
        File file = new File(elements[0]);

        addApplicants([...]);
    }

    void addApplicants([...]) {
    }

}
```

17.4.3. Building the shell plugin

Finally the `pom.xml` file for the plugin to the Nuxeo shell needs to at least contain the following dependency to be able to build extend the `AbstractCommand` class. Building the plugin will generate a `XXX.jar` file. Note that you should of course replace the *version* given as an example with the version suited for your need.

```
<dependencies>

  [...]
```

```
  <dependency>
    <groupId>org.nuxeo.ecm.platform</groupId>
    <artifactId>nuxeo-shell-commands-base</artifactId>
    <version>5.1.7-SNAPSHOT</version>
  </dependency>

  [...]

</dependencies>
```

17.4.4. Deploying the shell plugin

1. Install or decide on using a nuxeo shell already installed program.
2. Copy the generated `XXX.jar` file into the `bundles` directory of the nuxeo shell installed program.
3. Delete the `data` directory, if there is any, of the nuxeo installed program. This is to purge any caching of registered JARs.

The next time the nuxeo shell is restarted your new commands will be available.

Part III. 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://maven.nuxeo.org/`.

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. Commercial Support

C.1. About Us

The Nuxeo open source ECM platform 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.

C.2. Contact information

C.2.1. General

Web: `www.nuxeo.com`

E-mail: contact@nuxeo.com

C.2.2. France

18-20, rue Soleillet

75020 Paris - France

Tel: +33 1 40 33 79 87

Fax: +33 1 43 58 14 15

C.2.3. UK

Garden Studios, 11-15 Betterton Street, London WC2H 9BP

Tel: +44-207-043-7933