

eXo Platform 3.5 Developers Guide

eXo Platform ()

Copyright © 2010

| | |
|--|----|
| 1. Introduction | 1 |
| 1.1. Welcome to eXo Platform 3.5 | 1 |
| 1.2. About this guide | 2 |
| 2. Glossary | 3 |
| 3. Set Up Your Project | 7 |
| 4. eXo Architecture Primer | 8 |
| 4.1. Kernel | 8 |
| 4.1.1. Containers | 8 |
| 4.1.2. Services | 9 |
| 4.1.3. Service configuration | 9 |
| 4.1.4. Plugins | 12 |
| 4.1.5. Configuration loading sequence | 12 |
| 4.2. GateIn extensions | 13 |
| 4.2.1. Default Portal Container | 14 |
| 4.2.2. Register Extension | 14 |
| 4.3. Java Content Repository | 16 |
| 4.3.1. Repositories and workspaces | 16 |
| 4.3.2. Tree structure: working with nodes and properties | 17 |
| 5. Create Your Own Portal | 19 |
| 5.1. Create your extension project | 19 |
| 5.2. Structure of portal, pages and menus | 20 |
| 5.2.1. Page layout | 20 |
| 5.2.2. Visibility of pages | 22 |
| 5.2.3. Page access permission | 23 |
| 5.3. Add/remove languages | 23 |
| 5.3.1. Add new languages | 24 |
| 5.3.2. Remove languages | 24 |
| 5.4. Create custom look and feel | 25 |
| 5.4.1. Structure stylesheet | 25 |
| 5.4.2. Configure skin in GateIn | 28 |
| 5.4.3. Configure skin in WCM | 35 |
| 5.4.4. Create and apply Global stylesheet | 43 |
| 5.4.5. How to customize the Admin bar | 45 |
| 5.5. Add JavaScript to your portal | 46 |
| 6. Work with Content | 48 |
| 6.1. Document types | 48 |
| 6.2. WCM templates | 48 |
| 6.3. Document type | 49 |
| 6.4. Dialog Syntax | 49 |
| 6.4.1. Interceptors | 49 |
| 6.4.2. Hidden fields | 50 |
| 6.5. Manage template service | 53 |
| 6.6. Taxonomies | 55 |
| 7. Work with Applications | 56 |
| 7.1. Application integration | 56 |
| 7.2. Develop your own applications | 57 |
| 7.2.1. Gadget vs Portlet | 57 |
| 7.2.2. Gadget development quickstart with eXo IDE | 57 |
| 7.2.3. Portlet Bridges | 57 |
| 8. System Integration | 59 |
| 8.1. Authentication | 59 |
| 8.1.1. Single-Sign-On (SSO) | 59 |

| | |
|--|----|
| 8.1.2. Central Authentication Service (CAS) | 59 |
| 8.1.3. Kerberos SSO on Active Directory | 60 |
| 8.2. Users integration | 60 |
| 8.2.1. Organization Service | 60 |
| 8.2.2. Memberships, Groups and Users | 60 |
| 8.2.3. Organization API | 62 |
| 8.3. LDAP Integration | 62 |
| 8.3.1. Connection Settings | 62 |
| 8.3.2. Organization Service Configuration | 63 |
| 8.3.3. Active Directory sample configuration | 69 |
| 8.3.4. Picketlink IDM | 70 |
| 8.4. Email | 71 |
| 9. eXo Platform 3.5 APIs | 72 |
| 9.1. Definitions of API Levels | 72 |
| 9.1.1. Use Provisional or Experimental APIs | 73 |
| 9.2. Platform APIs | 73 |
| 9.2.1. Java APIs | 73 |
| 9.2.2. JavaScript APIs | 74 |
| 9.2.3. Web Services | 74 |
| 9.3. Provisional APIs | 74 |
| 9.3.1. Java APIs | 74 |
| 10. Cookbook | 75 |
| 10.1. How to Copy a Site | 75 |
| 11. New Features | 79 |
| 11.1. Navigation by content | 79 |
| 11.2. What is Navigation By Content? | 79 |
| 11.3. Actual content navigation | 79 |
| 11.4. How-To | 80 |
| 11.5. Actions on Navigation By Content | 81 |
| 11.5.1. Create a new product | 83 |
| 11.5.2. Develop your product content | 84 |

Chapter 1. Introduction

1.1. Welcome to eXo Platform 3.5

eXo Platform is the first and only integrated, cloud-ready user experience platform for building and deploying transactional websites, managing web and social content and creating gadgets and dashboards. eXo Platform lets companies leverage their existing Java infrastructure, while accommodating changing user behavior driven by consumer web technologies, such as social networks, social publishing and forums.

The following illustration gives you the overall architecture of eXo Platform 3.5.

The foundation of eXo Platform 3.5 is an enterprise portal and content management system. This provides a powerful set of REST-based services for rapid website development, content management and gadget-based development and deployment. eXo Platform 3.5 includes the following features.

- An enterprise portal serves as a powerful framework for developing portlets and other web-based user interfaces, and is based on the open source GateIn portal project co-developed by eXo and Red Hat.
- Web Content Management extends portal-based applications, allowing you to build dynamic and content-rich websites.
- Document Management for capturing and organizing documents and unstructured content, with content storage in the built-in Java Content Repository (JCR).
- xCMIS is an implementation of the full stack of Java-based CMIS (Content Management Interoperability Specification) services, so eXo-based applications can integrate with existing content management tools.
- Business Process Management (BPM) from Bonita Open Solution enables you to define workflow processes with automatic actions for web content, documents and more.
- Cloud-ready features allow eXo Platform 3.5 to run in multi-tenant environments, so social intranets and websites can take advantage of the benefits of private and public cloud platforms.

With eXo Platform 3.5, you can customize and extend your portal-based applications with user experience services to build social intranets and extranets.

- Enterprise social network features allow users to connect, collaborate within dedicated spaces, and publish real-time updates in activity streams. Support for OpenSocial provides a framework for building gadgets that can display and mash up activity information for contacts, social networks, applications and services.
- Collaboration & communication tools let you build a more productive and interactive dashboard for social intranet users. Intuitive Mail, Chat, Calendar and Address Book functionalities can seamlessly extend your portal-based web applications.
- Knowledge management features, including Forums, Answers, FAQs, and a complete enterprise wiki, can transform an extranet into an interactive online community and valuable knowledge base.
- Custom development in the web-based IDE is an intuitive web-based development environment where you can build, test and deploy client applications, such as gadgets and mash-ups, and RESTful services online. Offering the ability to extend eXo Platform online, eXo IDE instantly publishes any application that you can create and deploy immediately in your portal-based solutions.

- CRaSH is an open source tool to view and query content within a JCR server at runtime. It enables you to browse JCR trees, and serves as a shell for executing JCR operations easily, such as importing or exporting data securely. You can now extend the shell by writing Groovy commands, without recompiling easily.
- Native mobile applications for iPhone, iPad and Android allow users to easily and securely access their personalized intranet dashboards, activity streams, documents and more.

1.2. About this guide

The eXo Platform Developer Guide presents a complete overview of the eXo Platform 3.5 capabilities. This guide is intended for system integrators who want to know how to leverage eXo Platform in their customer projects and IT enterprises who need to customize and deploy their portals. Accordingly, this guide introduces the eXo Platform architecture, and shows developers how to perform some of the most common tasks needed for working with eXo Platform 3.5. It also serves as an entry point for the Reference Guide, which provides more in-depth technical details about eXo Platform 3.5. As a result, developers could be able to customize their own portals with eXo Platform 3.5.

This guide includes the following topics.

| | |
|--|--|
| Introduction | Overview of eXo Platform 3.5, of the developer guide and its intended readers. |
| Glossary | Terms commonly used in the developers' aspect in the process of implementing the eXo Platform applications. |
| Set Up Your Project | Step-by-step instructions on how to set up projects. |
| eXo Architecture Primer | Introduction to Kernel, GateIn Extensions, and Java Content Repository and their components. |
| Create Your Own Portal | Steps of creating a portal. |
| Work with Content | Topics related to the eXo Platform content. |
| Work with Applications | Instructions on how to integrate applications into your portal and how to deploy your own applications. |
| Integrate eXo Platform 3.5 into one information system | Topics related to the eXo Platform 3.5 integration into information systems through specific topics, such as authentication, user integration, LDAP integration and Email configuration. |
| eXo Platform 3.5 APIs | Information about APIs. |
| Cookbook | Introduction to Cookbook, particularly steps on how to copy a site. |
| New Features | New features integrated into eXo Platform 3.5. At present, in this chapter, only one feature called Navigation By Content is introduced. |

Chapter 2. Glossary

This chapter will provide you explanations about some technical terms which are used throughout the documentation.

- **Container templates** are those used to contain the UI components in a specific layout and display them on the portal page.
- **ConversationState** is an object which stores all information about the state of the current user. This **ConversationState** object also stores acquired attributes of an Identity which is a set of principals to identify a user.
- **Data container** implements the physical data storage. It enables different types of backend (such as RDB, FS files) to be used as a storage for the JCR data. With the main Data Container, other storages for persisted Property Values can be configured and used. The eXo JCR persistent data container can work in two configuration modes.
 - Multi-database: A database for each workspace (used in the standalone eXo JCR service mode).
 - Single-database: All workspaces persisted in one database (used in the embedded eXo JCR service mode; for example in eXo portal).

The data container uses the JDBC driver to communicate with the actual database software. For example, any JDBC-enabled data storage can be used with the eXo JCR implementation.

- **Database Creator** (DBCcreator) is a service that is responsible for executing the DDL (Data Definition Language) script in runtime. A DDL script may contain templates for database name, username, and password which will be replaced by real values at execution time.
- **Drives** are customized workspaces with:
 - a configured path where the user will start when browsing the drive.
 - a set of views with limitations to available actions, such as editing or creating contents while being in the drive.
 - a set of permissions to limit the access (and view) of the drive to a restricted number of people.
 - a set of options to describe the behavior of the drive when users browse it.
- **eXo Cache** is one which all applications on the top of eXo JCR need. This can rely on an *org.exoplatform.services.cache.ExoCache* instance managed by *org.exoplatform.services.cache.CacheService*.
- **eXoContainer** behaves like a class loader that is responsible for loading services/components. The eXoContainer class is inherited by all the containers, including RootContainer, PortalContainer, and StandaloneContainer. It itself inherits from a PicoContainer framework which allows eXo to apply the IoC Inversion of Control principles.
- **External Plugin** allows adding configuration for services and components easily.
- **Folksonomy** is a system of classification derived from the practice and a method of collaboratively creating and managing tags to annotate and categorize content. This practice is also known as collaborative tagging

social classification social indexing and social tagging (Source: <http://en.wikipedia.org>).

- **Gadgets** are web-based software components based on HTML, CSS, and JavaScript. They allow developers to easily write useful web applications that work anywhere on the web without modification. (Source: opensocial.org)
- **Groovy template** is widely used in eXo UI framework. It leverages the usage of Groovy language, a scripting language for Java. The template file consists of HTML code and Groovy code blocks.
- **JCR WebDav** is a service that allows a JCR repository to be accessed via WebDav.
- **JobSchedulerService** defines a job to execute a given number of times during a given period. It is a service that is in charge of unattended background executions commonly known for historical reasons as batch processing.
- **JodConverter** (Java OpenDocument Converter) is a tool that converts documents into different office formats and vice versa.
- **JCR Item** can be a node or a property.
- **ListenerService** is an event mechanism which allows triggering and listening to events under specific conditions inside eXo Platform. This mechanism is used in several places in eXo Platform, such as login/logout time, creating/updating users, and groups.
- **LockManager** stores lock objects so it can give a lock object or can release it. Also, LockManager is responsible for removing Locks that live too long.
- **Namespace** is the name of a node and property which may have a prefix delimited by a single ':' colon character. This name indicates the namespace of the item (Source JSR 170) and is used to avoid the naming conflict.
- **Navigation node** looks like a label of the link to page on the Navigation bar. By clicking a node, the page content is displayed. A node maps a URI and a portal page for the portal's navigation system.
- **Navigation** looks like a menu which is to help users visualize the site structure and to provide hyperlinks to other parts on a Portal. Thus, a bar which contains navigations is called the Navigation bar.
- **Node type** defines child nodes and properties which a node may (or must) have. Every node type has attributes, such as name, supertypes, mixin status, orderable child nodes status, property definitions, child node definitions and primary item name (Source: JSR 170).
- **Node** is an element in the tree structure that makes up a repository. Each node may have zero or more child nodes and zero or more child properties. There is a single root node per workspace which has no parent. All other nodes have only one parent.
- **Organization listener** provides a mechanism to receive notifications via an organization listener, including UserEventListener, GroupEventListener and MembershipEventListener.
 - UserEventListener is called when a user is created, deleted or modified.
 - GroupEventListener is called when a group is created, deleted or modified.
 - MembershipEventListener is called when a membership is created or removed.
- **Organization management** is a portlet that manages users groups and memberships. This portlet is often managed by administrators to set up permission for users and groups.

- **OrganizationService** is a service that allows accessing the Organization model. This model is composed of users, groups, and memberships. It is the basis of eXo's personalization and authorizations and is used for all over the platform.
- **Path constraint** restricts the result node to a scope specified by a path expression. The following path constraints must be supported exact child nodes descendants and descendants or self (Source: JCR 170).
- **Permission** is a permission settings control which actions users can or cannot perform within the portal and are set by the portal administrators. Permission types specify what a user can do within the portal.
- **Portal Page** consists of one or more various portlets. Their layouts are defined by container templates. To display a portal page, this page must be mapped to a navigation node.
- **Portal skins** are graphic styles that display an attractive user interface. Each skin has its own characteristics with different backgrounds, icons, color, and more.
- **PortalContainer** is a type of container which is created at the startup of the portal web application in the init method of the PortalController servlet.
- **Portlet** is a web-based application that provides a specific piece of content to be included as part of a portal page. In other words, portlets are pluggable user interface components that provide a presentation layer to information systems. There are two following types of portlet:
 - **Functional Portlets** support all functions within the portal. They are integrated into the portal that can be accessed through toolbar links.
 - **Interface Portlets** constitute the interface of a portal. eXo Portal consists of some Interface Portlets, such as Banner Portlet, Footer Portlet, Homepage Portlet, Console Portlet, Breadcrumb Portlet and more.
- **Property constraint** is one which may be specified by a query on the result nodes by way of property constraints (Source: JCR 170).
- **Property** is an element in the tree structure that makes up a repository. Each property has only one parent node and has no child node.
- **Repository** holds references to one or more workspaces.
- **eXo REST framework** is used to make eXo services (for example, the components deployed inside eXo Container) simply and transparently accessible via HTTP in a RESTful manner. In other words, those services should be viewed as a set of REST Resources-endpoints of the HTTP request-response chain. Those services are called **ResourceContainers**.
- **RootContainer** is a base container. It plays an important role during the startup but it is recommended that it should not be used directly.
- **RTL Framework** (Right To Left Framework) is a framework that handles the text orientation depending on the current locale settings. It consists of four components, including Groovy template, Stylesheet, Images, and Client java.
- **StandaloneContainer** is a context independent eXo Container. It is also used for unit tests.
- **Taxonomy** is used to sort documents to ease searches when browsing documents online.
- **Tree structure** is defined as a hierarchical structure with a set of linked nodes and properties.
- **Type constraint** specifies the common primary node type of the returned nodes plus possibly additional

mixin types that they also must have. Type constraints are inheritance-sensitive in which specifying a constraint of node type x will include all nodes explicitly declared to be type x and all nodes of subtypes of x (Source: JSR 170).

- **Web Content** is the textual, visual or aural content that is encountered as part of the user experiences on a website. It may include other things, such as texts images, sounds, videos, and animations.
- **Workspace** is a container of single rooted tree that includes items.

Chapter 3. Set Up Your Project

This section aims at helping you apply eXo Platform to your projects successfully by introducing eXo architecture and giving step-by-step instructions for you to build your own custom portal based on eXo Platform.

Requirements:

- JDK (Java Development Kit) 6.0
- SVN 1.6+
- Maven 2.2.1+
- Tomcat 6.0.26 or JBoss 5.1.0

Install and configure Maven:

You need to add a system environment variable *MAVEN_OPTS* (it could be in a .profile startup script on Linux/MacOS operating systems or in the global environment variables panel on Windows).

- Windows:

```
set MAVEN_OPTS=-Xshare:auto -Xms128M -Xmx1G -XX:MaxPermSize=256M
```

- Linux/MacOS:

```
export MAVEN_OPTS="-Xshare:auto -Xms128M -Xmx1G -XX:MaxPermSize=256M"
```

Maven settings:

1. Save the *settings.xml* file to the path: *HOME/.m2/settings.xml*.
2. Edit and change the *local-properties* profile, including:

- *exo.projects.directory.dependencies* contains the application servers, and Openfire.
- each *exo.projects.app.AS-NAME.version* contains the name and version of the application servers.



Note

If the *settings.xml* file has been existing, you can merge them. You will need the followings:

- The *local-properties* profile, which defines properties, is used to build application server distributions of our products.
- The [repository](#) to download our dependencies.

Chapter 4. eXo Architecture Primer

With the aim of helping readers further understand about the eXo architecture, including Kernel, GateIn extensions and Java Content Repository via concepts, services, configuration, and more, this chapter is divided into 3 main sections with their sub-topics as follows:

The [Kernel](#) part presents the following main contents:

- [Containers](#)
- [Services](#)
- [Service configuration](#)
- [Plugins](#)
- [Configuration loading sequence](#)

The [GateIn extensions](#) part includes contents of the followings:

- [Default Portal Container](#)
- [Register Extension](#)

The [Java Content Repository](#) part shows you the following main contents:

- [Repositories and workspaces](#)
- [Tree structure: working with nodes and properties](#)

4.1. Kernel

All eXo Platform services are built around the eXo Kernel, or the service management layer, which manages the configuration and the execution of all components. The main kernel object is the eXo Container, a micro-container that glues services together through the dependency injection. The container is responsible for loading services/components.

This part introduces concepts of Container and Services with an overview before configuring basic services.

4.1.1. Containers

A container is always required to access a service, because the eXo Kernel relies on the dependency injection. This means that the lifecycle of a service (for example, instantiating, opening and closing streams, disposing) is handled by a dependency provider, such as the eXo Container, rather than the consumer. The consumer only needs a reference to an implementation of the requested service. The implementation is configured in an **.xml** configuration file that comes with every service. To learn more about the dependency injection, visit [here](#).

eXo Platform provides two types of containers: RootContainer and PortalContainer.

The RootContainer holds the low level components. It is automatically started before the PortalContainer. You will rarely interact directly with the RootContainer except when you activate your own extension. The

PortalContainer is created for each portal (one or several portals). All services started by this container will run as embedded in the portal. It also gives access to components of its parent RootContainer.

In your code, if you need to invoke a service of a container, you can use the ExoContainerContext helper from any location. The code below shows you a utility method that you can use to invoke any eXo Platform service.

```
public class ExoUtils {  
    /**  
     * Get a service from the portal container  
     * @param type : component type  
     * @return the concrete instance retrieved in the container using the type as key  
     */  
    public <T>T getService(Class<T> type) {  
        return (T)ExoContainerContext.getCurrentContainer().getComponentInstanceOfType(type);  
    }  
}
```

Then, invoking becomes as easy as:

```
OrganizationService orgService = ExoUtils.getService(OrganizationService.class)
```

4.1.2. Services

Containers are used to gain access to services. The followings are important characteristics of services:

- Because of the Dependency Injection concept, the interface and implementation for a service are usually separate.
- Each service has to be implemented as a singleton, which means it is created only once per portal container in a single instance.
- A component equals a service. A service must not be a large application. A service can be a little component that reads or transforms a document where the term "component" is often used instead of service.

For example, in the lib/folder, you can find services for databases, caching, LDAP:

- `exo.core.component.database-x.y.z.jar`
- `exo.kernel.component.cache-x.y.z.jar`
- `exo.core.component.organization.ldap-x.y.z.jar`

4.1.3. Service configuration

To declare a service, you must add the **.xml** configuration file to a specific place. This file can be in the jar file, in a webapp or in the external configuration directory. If you write a new component for the eXo Container, you should always provide a default configuration in your jar file. This default configuration must be in the `/conf/portal/configuration.xml` file in your jar.

A configuration file can specify several services, so there can be several services in one jar file.

4.1.3.1. Kernel XML Schema

Containers configuration files must comply with the kernel configuration grammar. Thus, all configurations will contain an XSD declaration like this:

```
<configuration xmlns="http://www.exoplatform.org/xml/ns/kernel_1_1.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.exoplatform.org/xml/ns/kernel_1_1.xsd http://www.exoplatform.org/xml/ns/kernel_1_1.xsd">
  </configuration>
```

The `kernel_1_1.xsd` file mentioned in the example above can be found inside `exo.kernel.container-x.y.z.jar!org/exoplatform/container/configuration/` along with other versions.

4.1.3.2. Components

The service registration within the container is done with the `<component>` element.

For example, open the `exo-ecms-core-services-x.y.z.jar` file. Next, open the `/conf/portal/configuration.xml` file. You will see:

```
<component>
  <type>org.exoplatform.services.deployment.ContentInitializerService</type>
</component>
<component>
  <key>org.exoplatform.services.cms.CmsService</key>
  <type>org.exoplatform.services.cms.impl.CmsServiceImpl</type>
</component>
```

Each component has a *key* which matches with the qualified Java interface name (`org.exoplatform.services.cms.CmsService`).

The specific implementation class of the component (`CmsServiceImpl`) is defined in the `<type>` tag.

If a service does not have a separate interface, the `<type>` will be used as the key in the container. This is the case of `ContentInitializerService`.

4.1.3.3. Parameters

You can provide initial parameters for your service by defining them in the configuration file. The followings are different parameters:

- value-param
- properties-param
- object-param
- collection
- map
- native-array

4.1.3.3.1. Value-param

You can use the `value-param` to pass a single value to the service.

```
<component>
  <key>org.exoplatform.services.resources.LocaleConfigService</key>
```

```

<type>org.exoplatform.services.resources.impl.LocaleConfigServiceImpl</type>
<init-params>
  <value-param>
    <name>locale.config.file</name>
    <value>war:/conf/common/locales-config.xml</value>
  </value-param>
</init-params>
</component>

```

The LocaleConfigService service accesses the value of value-param in its constructor.

```

package org.exoplatform.services.resources.impl;
public class LocaleConfigServiceImpl implements LocaleConfigService {
  public LocaleConfigServiceImpl(InitParams params, ConfigurationManager cmanager) throws Exception {
    configs_ = new HashMap<String, LocaleConfig>(10);
    String confResource = params.getValueParam("locale.config.file").getValue();
    InputStream is = cmanager.getInputStream(confResource);
    parseConfiguration(is);
  }
}

```

4.1.3.3.2. Object-param

For the object-param component, you can look at the LDAP service:

```

<component>
  <key>org.exoplatform.services.ldap.LDAPService</key>
  <type>org.exoplatform.services.ldap.impl.LDAPServiceImpl</type>
  <init-params>
    <object-param>
      <name>ldap.config</name>
      <description>Default ldap config</description>
      <object type="org.exoplatform.services.ldap.impl.LDAPConnectionConfig">
        <field name="providerURL"><string>ldaps://10.0.0.3:636</string></field>
        <field name="rootdn"><string>CN=Administrator,CN=Users,DC=exoplatform,DC=org</string></field>
        <field name="password"><string>exo</string></field>
        <field name="version"><string>3</string></field>
        <field name="minConnection"><int>5</int></field>
        <field name="maxConnection"><int>10</int></field>
        <field name="referralMode"><string>ignore</string></field>
        <field name="serverName"><string>active.directory</string></field>
      </object>
    </object-param>
  </init-params>
</component>

```

The object-param is used to create an object (which is actually a Java Bean) passed as a parameter to the service. This object-param is defined by a name, a description and exactly one object. The object tag defines the type of the object, while the field tags define parameters for that object.

You can see how the service accesses the object in the code below:

```

package org.exoplatform.services.ldap.impl;

public class LDAPServiceImpl implements LDAPService {
  // ...
  public LDAPServiceImpl(InitParams params) {
    LDAPConnectionConfig config = (LDAPConnectionConfig) params.getObjectParam("ldap.config").getObject();
    ...
  }
  // ...
}

```

The passed object is LDAPConnectionConfig, which is a classic Java Bean. It contains all fields defined in the

configuration files and also the appropriate getters and setters (not listed here). You also can provide default values. The container creates a new instance of your Java Bean and calls all setters whose values are configured in the configuration file.

```
package org.exoplatform.services.ldap.impl;

public class LDAPConnectionConfig {
    private String providerURL = "ldap://127.0.0.1:389";
    private String rootdn;
    private String password;
    private String version;
    private String authenticationType = "simple";
    private String serverName = "default";
    private int minConnection;
    private int maxConnection;
    private String referralMode = "follow";
    // ...
}
```

4.1.3.3. Rest of parameter types

Other possible parameter types are Collection, Map and Native Array. See the exhaustive reference in the Kernel reference guide.

4.1.4. Plugins

Some components may want to offer some extensibilities. For this, they use a plugin mechanism based on the injection method. To offer an extension point for plugins, a component needs to provide a public method that takes an instance of *org.exoplatform.container.xml.ComponentPlugin* as a parameter.

Plugins enable you to provide the structured configuration outside the original declaration of the component. This is the main way to customize eXo Platform to your needs.

You can have a look at the configuration of the TaxonomyPlugin of the TaxonomyService as below:

```
<external-component-plugins>
  <target-component>org.exoplatform.services.cms.taxonomy.TaxonomyService</target-component>
  <component-plugin>
    <name>predefinedTaxonomyPlugin</name>
    <set-method>addTaxonomyPlugin</set-method>
    <type>org.exoplatform.services.cms.taxonomy.impl.TaxonomyPlugin</type>
    <init-params><!-- ... --></init-params>
  </component-plugin>
</external-component-plugins>
```

The `<target-component>` defines components that host the extension point. The configuration is injected by the container using the method defined in `<set-method>` (`addTaxonomyPlugin()`). The method accepts exactly one argument of the *org.exoplatform.services.cms.categories.impl.TaxonomyPlugin* type.

The content of `<init-params>` is interpreted by the TaxonomyPlugin object.

4.1.5. Configuration loading sequence

The Kernel startup follows a well-defined sequence to load configuration files. The objects are initialized in the container only after the whole loading sequence is done. Hence, by placing your configuration in the upper location of the sequence, you can override a component declaration by yourself. You will typically do this when you want to provide your own implementation of a component, or declare custom init-params.

**Note**

The *external-component-plugins* declarations are additive, so it is NOT possible to override them.

The loading sequence involves loading successively configurations for the RootContainer then from the PortalContainers:

- Services default RootContainer configurations from JAR files: */conf/configuration.xml*.
- External RootContainer configuration will be found at *\$exo.conf.dir/configuration.xml*.
- Services default PortalContainer configurations from JAR files: */conf/\$PORTAL/configuration.xml*.
- Web applications configurations from WAR files */WEB-INF/conf/configuration.xml*.
- External configuration for services of the portal will be found at *\$exo.conf.dir/portal/\$PORTAL/configuration.xml*.

**Note**

- *\$exo.conf.dir* is a system property that points to the external configuration file on the file system. It is passed to the JVM in the startup script like *-Dexo.conf.dir=gateln*.
- *\$PORTAL* is the name of the portal container. By default, the name "portal" is unique.

4.2. Gateln extensions

GateIn extensions are special **.war** files that are recognized by eXo Platform and contribute to custom configurations to the PortalContainer. To create your own portal, you will have to create a GateIn extension.

The extension mechanism makes possible to extend or even override portal resources in almost plug-and-play way. You simply add a **.war** archive with your custom resources to the war folder and edit the configuration of the PortalContainerConfig service. Customizing a portal does not involve unpacking and repacking the original portal **.war** archives. Instead, you need to create your own **.war** archive with your own configurations, and modify resources. The content of your custom **.war** archive overrides the resources in the original archives.

The most elegant way to reuse configuration for different coexisting portals is by way of extension mechanism - by inheriting resources and configuration from existing web archives, and just adding extra resources to it, and overriding those that need to be changed by including modified copies.

**Note**

Starter is a web application that has been added to create and start all the portals, such as portal containers, at the same time once all the other web applications have already been started. In fact, all other web applications can potentially defined several things at startup, such as skins, Javascripts, Google gadgets and configuration files. The loading order is important as we can define skins or configuration files or a Javascript again from a web application 1. This could depend on another Javascript from a web application 2. Thus, if the web application 2 is loaded after the web application 1, you will get errors in the merged Javascript file.

If you ship servlets or servlet filters as part of your portal extension, and if you need to access specific resources of portal at any time during processing of the servlet or filter request, you need to make sure that the *servlet/filter* is associated with the current container. The proper way to do that is to make your servlet extend the *org.exoplatform.container.web.AbstractHttpServlet* class. This will not only properly initialize the current PortalContainer for you, but also set the current thread's context classloader to one that looks for resources in associated web applications in the order specified by Dependencies configuration (as explained in the Extension mechanism section).

As similar to filters, make sure that your filter class extends *org.exoplatform.container.web.AbstractFilter*. Both AbstractHttpServlet and AbstractFilter have the method *getContainer()*, which returns the current PortalContainer.

4.2.1. Default Portal Container

eXo Platform comes with a pre-configured PortalContainer named "portal". This portal container configuration ties the core and extended services stack. The default Portal Container is started from *portal.war* and naturally maps to the */portal* URL.

The GateIn extension mechanism lets you extend the portal context easily. This feature is fundamental as it shields you from any changes we may want to do in *portal.war*. You do not need to be afraid of upgrades anymore as your *extension.war* will be clearly separated from the *portal.war*.

To achieve this extensibility, the PortalContainer activates two advanced features:

- A unified classloader: any classpath resource, such as property files, will be accessible as if it was inside the *portal.war*.



Note

This is valid only for resources but not for Java classes.

- A unified servlet context: any web resource contained in your *extension.war* will be accessible from */portal/uri*.

The next part explains what to do to make a simple extension for "portal" container.

4.2.2. Register Extension

The webapps are loaded in the order defined in the list of dependencies of the *PortalContainerDefinition*. You then need to deploy the *starter.war*; otherwise, the webapps will be loaded in the default application server's order', such as the loading order of the Application Server.

If you need to customize your portal by adding a new extension and/or a new portal, you need to define the related *PortalContainerDefinitions* and to deploy the starter. Otherwise, you do not need to define any *PortalContainerDefinition* or deploy the starter.

First, you need to tell eXo Platform to load *WEB-INF/conf/configuration.xml* of your extension, you need to declare it as a *PortalContainerConfigOwner*. Next, open the file *WEB-INF/web.xml* of your extension and add a listener:

```
<web-app>
```

```

<display-name>my-portal</display-name>
<listener>
  <listener-class>org.exoplatform.container.web.PortalContainerConfigOwner</listener-class>
</listener>
<!-- ... -->
</web-app>

```

You need to register your extension in the portal container. This is done by the **.xml** configuration file like this:

```

<external-component-plugins>
  <target-component>org.exoplatform.container.definition.PortalContainerConfig</target-component>
  <component-plugin>
    <name>Change PortalContainer Definitions</name>
    <set-method>registerChangePlugin</set-method>
    <type>org.exoplatform.container.definition.PortalContainerDefinitionChangePlugin</type>
    <init-params>
      <object-param>
        <name>change</name>
        <object type="org.exoplatform.container.definition.PortalContainerDefinitionChange$AddDependencies">
          <field name="dependencies">
            <collection type="java.util.ArrayList">
              <value>
                <string>my-portal</string>
              </value>
              <value>
                <string>my-portal-resources</string>
              </value>
            </collection>
          </field>
        </object>
      </object-param>
      <value-param>
        <name>apply.default</name>
        <value>true</value>
      </value-param>
    </init-params>
  </component-plugin>
</external-component-plugins>

```

A *PortalContainerDefinitionChangePlugin* plugin is defined to the *PortalContainerConfig*. The plugin declares a list of dependencies that are webapps. The *apply.default=true* indicates that your extension is actually extending *portal.war*. You need to package your extension into a **.war** file and put it to the tomcat webapps folder, then restart the server.

In your portal, if you want to add your own property file to support localization for your keys, you can do as follows:

- Put your property file into the */WEB-INF/classes/locale/portal* folder of your extension project.
- Add an external plugin declaration to the *.xml* configuration file.

```

<external-component-plugins>
  <!-- The full qualified name of the ResourceBundleService -->
  <target-component>org.exoplatform.services.resources.ResourceBundleService</target-component>
  <component-plugin>
    <!-- The name of the plugin -->
    <name>Sample ResourceBundle Plugin</name>
    <!-- The name of the method to call on the ResourceBundleService in order to register the ResourceBundles -->
    <set-method>addResourceBundle</set-method>
    <!-- The full qualified name of the BaseResourceBundlePlugin -->
    <type>org.exoplatform.services.resources.impl.BaseResourceBundlePlugin</type>
    <init-params>
      <!--values-param>
        <name>classpath.resources</name>
        <description>The resources that start with the following package name should be load from file system</description>
        <value>locale.portlet</value>
      </values-param-->
    </init-params>
  </component-plugin>
</external-component-plugins>

```

```

<values-param>
  <name>init.resources</name>
  <description>Store the following resources into the db for the first launch </description>
  <value>locale.portal.sample</value>
</values-param>
<values-param>
  <name>portal.resource.names</name>
  <description>The properties files of the portal , those file will be merged
  into one ResoruceBundle properties </description>
  <value>locale.portal.sample</value>
</values-param>
</init-params>
</component-plugin>
</external-component-plugins>

```

4.3. Java Content Repository

All data of eXo Platform are stored in a Java Content Repository (JCR). JCR is the Java specification ([JSR-170](#)) for a type of Object Database tailored to the storage, searching, and retrieval of hierarchical data. It is useful for the content management systems, which require storage of objects associated with metadata. The JCR also provides versioning, transactions, observations of changes in data, and import or export of data in XML. The data in JCR are stored hierarchically in a tree of nodes with associated properties.

Also, the JCR is primarily used as an internal storage engine. Accordingly, eXo Content lets you manipulate JCR data directly in several places.

4.3.1. Repositories and workspaces

A content repository consists of one or more workspaces. Each workspace contains a tree of items.

To access the repository's content from a service:

```

import javax.jcr.Session;

import org.exoplatform.services.jcr.RepositoryService;
import org.exoplatform.services.jcr.core.ManageableRepository;
import org.exoplatform.services.jcr.ext.common.SessionProvider;
import org.exoplatform.services.wcm.utils.WCMCoreUtils;

// For example
RepositoryService repositoryService = WCMCoreUtils.getService(RepositoryService.class);
ManageableRepository manageableRepository = repositoryService.getRepository(repository);
SessionProvider sessionProvider = WCMCoreUtils.getSessionProvider();
Session session = sessionProvider.getSession(workspace, manageableRepository);

```

You can use the session object to retrieve your node content.

```

String path = "/"; // put your node path here
Node node = (Node) session.getItem(path);

```

The `javax.jcr.Repository` object can be obtained via one of the following ways:

- Using the eXo Container "native" mechanism. All Repositories are kept with a single `RepositoryService` component. So it can be obtained from eXo Container as below:

```

RepositoryService repositoryService = (RepositoryService) container.getComponentInstanceOfType(RepositoryService.class);
Repository repository = repositoryService.getRepository("repositoryName");

```

- Using the eXo Container "native" mechanism with a thread local saved "current" repository (especially if you plan to use a single repository which covers more than 90% of usecases)

```
// set current repository at initial time
RepositoryService repositoryService = (RepositoryService) container.getComponentInstanceOfType(RepositoryService.class);
repositoryService.setCurrentRepositoryName("repositoryName");
....
// retrieve and use this repository
Repository repository = repositoryService.getCurrentRepository();
```

- Using JNDI as specified in JSR-170.

```
Context ctx = new InitialContext();
Repository repository =(Repository) ctx.lookup("repositoryName");
```

Next, you need to log in the server to get a Session object by either of two ways:

- Creating a Credential object, for example:

```
Credentials credentials = new SimpleCredentials("exo", "exo".toCharArray());
Session jcrSession = repository.login(credentials, "production");
```

- Logging in by using:

```
Session jcrSession = repository.login("production");
```

This way is only applied when you run an implementation of eXo Platform embedded in the portal.

In embedded cases, the implementation will directly leverage the organization and security services that rely on LDAP or DB storage and JAAS login modules. Single-Sign-On products can now also be used as eXo Platform v.2 which supports them.

JCR Session common considerations:

- `javax.jcr.Session` is not a safe object of thread. So, you should never try to share it between threads.
- Do not use System session from the user-related code because a system session has unlimited rights. Call `ManageableRepository.getSystemSession()` from the process-related code only.
- Call `Session.logout()` explicitly to release resources assigned to the session.
- When designing your application, you should take care of the Session policy inside your application. Two strategies are possible: Stateless (Session per business request) and Stateful (Session per User) or some mixings.

4.3.2. Tree structure: working with nodes and properties

Every node can only have one primary node type. The primary node type defines names, types and other

characteristics of the properties and child nodes that a node is allowed (or required) to have. Every node has a special property called `jcr:primaryType` that records the name of its primary node type. A node may also have one or more mixin types. These are node type definitions that can mandate extra characteristics (for example, more child nodes, properties and their respective names and types).

Data are stored in properties, which may hold simple values, such as numbers, strings or binary data of arbitrary length.

The JCR API provides methods to define node types and node properties, create or delete nodes, and add or delete properties to an existing node.

You can refer to Section 6.2.3. Node Read Methods in the JCR Specification document.

Chapter 5. Create Your Own Portal

When working with eXo Platform, it is important not to modify the source code. This will ensure compatibility with future upgrades, and support will be simplified. To customize your portal, you need to create an extension project by providing your own artifacts as a set of wars/jars/ears.

This chapter will show you how to create a portal via the following topics:

- [Create your extension project](#)
- [Structure of portal, pages and menus](#)
- [Add/remove languages](#)
- [Create custom look and feel](#)
 - [Structure stylesheet](#)
 - [Configure skin in GateIn](#)
 - [Configure skin in WCM](#)
 - [Create and apply Global stylesheet](#)
 - [Customize Admin bar](#)
- [Add JavaScript to your portal](#)

5.1. Create your extension project

A custom extension contains two mandatory items:

- **extension webapp** contains resources and kernel configurations.
- **extension activator jar** identifies your webapp as a dependency of the portal container.

To see the sample extension package, visit [here](#).

Once you have modified the sample extension to build your own portal, use the *maven clean install* command to create the archive files.

To deploy your extension in Tomcat:

1. Add the *sample-ext.war* file from the *sample/extension/war/target/* to the *tomcat/webapps* directory.
2. Add the *starter* folder from *starter/war/target/* to the *tomcat/webapps* directory.
3. Rename the *starter* directory (unzipped folder) to *starter.war*.



Note

This will only work if the *starter.war* is the last .war file to be loaded, so you may need to rename it if your war files are loaded in the alphabetical order.

4. Add the .jar file named `exo.portal.sample.extension.config-X.Y.Z.jar` from `sample/extension/config/target/` to the `tomcat/lib` directory.
5. Add the .jar file named `exo.portal.sample.extension.jar-X.Y.Z.jar` from `sample/extension/jar/target/` to the `tomcat/lib` directory.

For the JBoss deployment with more details, refer to Reference Guide on [here](#).

5.2. Structure of portal, pages and menus

You can create multiple pages within a single portal. Permissions can be defined to make them visible only to specific groups and/or users. This chapter describes how to define this structure.

5.2.1. Page layout

The configuration of "classic" portal can be found in the `/src/main/webapp/WEB-INF/conf/sample-ext/portal/portal/classic` directory of your extension webapp.

Portal:

The `portal.xml` file describes the layout and portlets common to all pages of the portal.

```
<portal-config xmlns="http://www.gatein.org/xml/ns/gatein_objects_1_0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.gatein.org/xml/ns/gatein_objects_1_0 http://www.gatein.org/xml/ns/gatein_objects_1_0.xsd">
  <portal-name>classic</portal-name>
  <locale>en</locale>
  <access-permissions>Everyone</access-permissions>
  <edit-permission>*:/platform/administrators</edit-permission>
  <properties>
    <entry key="sessionAlive">onDemand</entry>
  </properties>

  <portal-layout>
    <portlet-application>
      <portlet>
        <application-ref>web</application-ref>
        <portlet-ref>BannerPortlet</portlet-ref>
        <preferences>
          <preference>
            <name>template</name>
            <value>par:/groovy/groovy/webui/component/UIBannerPortlet.gtmpl</value>
            <read-only>>false</read-only>
          </preference>
        </preferences>
      </portlet>
      <access-permissions>Everyone</access-permissions>
      <show-info-bar>>false</show-info-bar>
    </portlet-application>

    <portlet-application>
      <portlet>
        ...
      </portlet>
    </portlet-application>

    <portlet-application>
      <portlet>
        ...
      </portlet>
    </portlet-application>

  </portal-layout>
  <page-body> </page-body>

  <portlet-application>
    <portlet>
      ...
    </portlet>
  </portlet-application>
</portal-config>
```

```

    </portlet>
  </portlet-application>
</portal-layout>
</portal-config>

```

As you can see, each portlet can be configured with a set of preferences, which will be further detailed in Chapter 6. The tag `<page-body>` `</page-body>` is a placeholder for the different pages of your portal. When the user opens a new portal page, all portlets of the portal layout (`portal.xml`) are remained, whereas the content of `<page-body>` switches to the page opened by the user.

Pages:

The `pages.xml` file is used to describe the content of pages of your portal. In particular, pages of your portal will be inside the `<page-body>` tag of the `portal.xml` file above.

This is the example of the `pages.xml` file of the classic portal.

```

<page>
  <name>homepage</name>
  <title>Home Page</title>
  <access-permissions>Everyone</access-permissions>
  <edit-permission>*:/platform/administrators</edit-permission>
  <container id="ClassicBody" template="system:/groovy/portal/webui/container/UITableColumnContainer.gtmpl">
    <access-permissions>Everyone</access-permissions>
    <container id="ClassicLeft" template="system:/groovy/portal/webui/container/UIContainer.gtmpl">
      <access-permissions>Everyone</access-permissions>
      <portlet-application>
        <portlet>
          <application-ref>presentation</application-ref>
          <portlet-ref>SingleContentViewer</portlet-ref>
          <preferences>
            <preference>
              <name>repository</name>
              <value>repository</value>
              <read-only>>false</read-only>
            </preference>
            ...
            <preference>
              <name>ShowTitle</name>
              <value>>false</value>
              <read-only>>false</read-only>
            </preference>
          </preferences>
        </portlet>
        <title>Introduce</title>
        <access-permissions>Everyone</access-permissions>
        <show-info-bar>>false</show-info-bar>
        <show-application-state>>false</show-application-state>
        <show-application-mode>>false</show-application-mode>
      </portlet-application>
    </container>
  </container>
</page>

```



Note

See Chapter 6 to learn more about the portlet configuration within the `pages.xml` file.

Navigation:

The `navigation.xml` is used to associate the links in your navigation with your portal pages. The navigation is a tree structure of "navigation nodes". Each navigation node points to one page (called the page node). The tree structure corresponds to a hierarchy of menus and sub-menus of a site.

If the pattern `#{}` is used, the link label will be loaded from the portal resource bundle. To learn more, refer to [Internationalization Configuration](#).

```
<node-navigation>
  <owner-type>portal</owner-type>
  <owner-id>classic</owner-id>
  <priority>1</priority>
  <page-nodes>
    <node>
      <uri>home</uri>
      <name>home</name>
      <label>#{portal.classic.home}</label>
      <page-reference>portal::classic::homepage</page-reference>
    </node>
    <node>
      <uri>webexplorer</uri>
      <name>webexplorer</name>
      <label>#{portal.classic.webexplorer}</label>
      <page-reference>portal::classic::webexplorer</page-reference>
    </node>
  </page-nodes>
</node-navigation>
```

This navigation tree is shown and reused in different portlets, such as sitemap, navigation or breadcrumbs. The breadcrumbs portlet renders the position of the current navigation node.



Note

For the top nodes, the URI and the navigation node name must have the same value. For other nodes, the URI is composed like `<uri>contentmanagement/fileexplorer</uri>` where 'contentmanagement' is the name of the parent node and 'fileexplorer' is the name of node (`<name>fileexplorer</name>`).

5.2.2. Visibility of pages

When you configure the *navigation.xml* file, sometimes you need to set the visibility of page (node).

To configure, simply put `<visibility>type_of_visibility</visibility>` as a child of `<node>` tag.

GateIn supports 4 types of page visibility, including:

- **DISPLAYED:** The page will be displayed.
- **HIDDEN:** The page is not visible in the navigation but can be accessed directly with its URL.
- **SYSTEM:** It is a system page which is visible to superusers. In particular, only superusers can change or delete this system page.
- **TEMPORAL:** The page is displayed in related time range. When the visibility of TEMPORAL page is configured, the start and end date can be specified by using `<startpublicationdate>` and `<endpublicationdate>`. For example:

```
<node>
  ...
  <visibility>TEMPORAL</visibility>
  <startpublicationdate>01/13/2011 12:46:38</startpublicationdate>
  <endpublicationdate>01/20/2011 18:46:42</endpublicationdate>
</node>
```

- To hide a page, use `<visibility>HIDDEN</visibility>`:

```
<node-navigation xmlns="http://www.gatein.org/xml/ns/gatein_objects_1_0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <page-nodes>
    <node>
      <uri>sitemap</uri>
      <name>sitemap</name>
      <label>#{portal.classic.sitemap}</label>
      <visibility>HIDDEN</visibility>
      <page-reference>portal::classic::sitemap</page-reference>
    </node>
    ...
  </page-nodes>
</node-navigation>
```

5.2.3. Page access permission

You can easily restrict the access of selected pages and navigation nodes to certain groups and users. You just need to create *pages.xml* and *navigation.xml* files in folders named after the group and user to which you want to give permission:

- *sample-ext/portal/group/group-name/your files*
- *sample-ext/portal/user/username/your files*

As you know, the permission on a page has two types: access and edit.

To configure them, edit the *pages.xml* file and put tags: `<access-permissions>` and `<edit-permission>`. Currently, eXo Platform supports several access permissions but only one edit permission. It means that you can configure to make many groups to have the access permission but you can only set one group for edit permission.

For example:

```
<page>
  <name>newStaff</name>
  <title>New Staff</title>
  <access-permissions>manager:/organization/management/executive-board;member:/organization/management/executive-board</access-permissions>
  <edit-permission>manager:/organization/management/executive-board</edit-permission>
  ...
</page>
```

5.3. Add/remove languages

All languages are put in the *myextension.war/WEB-INF/conf/common/locales-config.xml* directory. Each language consists of information related to key, output-encoding, input-encoding, description and orientation. Different languages will be defined in corresponding *resource bundle* files with keys specified in the *locale-config.xml* file.

All languages defined in the *locale-config.xml* file are listed in **Interface Language Settings**.

5.3.1. Add new languages

To add a new language, you need to add the corresponding language node in the *locale-config.xml* file. Next, you must create a new *resource bundle* file containing the suffix name as key of the added node.

For example, to add Italian, do as follows:

1. Add the following node to the *locale-config.xml* file.

```
<locale-config>
  <locale>it</locale>
  <output-encoding>UTF-8</output-encoding>
  <input-encoding>UTF-8</input-encoding>
  <description>Default configuration for Italian locale</description>
</locale-config>
```

2. Create a new *resource bundle* as **webui_it.properties** in the *myextension.war/WEB-INF/classes/locale/portal* folder.



Note

This step is necessary because the Resource Bundle Service of the portal will find keys and values in the *resource bundle* of each corresponding language.

3. Restart the server.

To check if the added language takes effect, simply click the **Change Language** button on the top right corner of the portal. In the **Interface Language Setting** window, you will see Italian as shown:

5.3.2. Remove languages

To remove an existing language, you need to delete the relevant language node in the *locale-config.xml* file and all files containing the suffix name as the key of language.

For example, to remove French, do as follows:

1. Find and remove the following node from the *locale-config.xml* file.

```
<locale-config>
  <locale>fr</locale>
  <output-encoding>UTF-8</output-encoding>
  <input-encoding>UTF-8</input-encoding>
  <description>Default configuration for france locale</description>
</locale-config>
```

2. Continue removing all *resource bundle* files containing the suffix name as **fr** in all folders.



Note

It is recommended this step should be done to delete unnecessary data in the application.

3. Restart the server.

To check if the French removal takes effect, simply click the **Change Language** button on the top right corner of the portal. In the **Interface Language Setting** window, French will not exist on the **Interface Language Setting** window any longer.

5.4. Create custom look and feel

5.4.1. Structure stylesheet

5.4.1.1. Page skin Elements

The complete skinning of a page can be decomposed into three main parts:

Portal skin

The portal skin contains styles for the HTML tags (for example, div, th, td) and the portal UI (including the toolbar). This should include all UI components, except for window decorators and portlet specific styles.

Window styles

The CSS styles are associated with the portlet window decorators. The window decorators contain control buttons and borders surrounding each portlet. Individual portlets can have their own window decorators selected, or be rendered without one.

Portlet skins

The portlet skins affect how portlets are rendered on the page. The portlet skins can affect in two main ways:

5.4.1.1.1. Portlet Specification CSS Classes

The [portlet specification](#) defines a set of CSS classes that should be available to portlets. eXo Platform provides these classes as a part of the portal skin. This enables each portal skin to define its own look and feel for these default values.

5.4.1.1.2. Portlet skins

eXo Platform provides a means for portlet CSS files to be loaded that is based on the current portal skin. This enables a portlet to provide different CSS styles to better match the current portal's look and feel.



Note

The window decorators and the default portlet specification CSS classes should be considered as separate types of skinning components, but they need to be included as a part of the overall portal skin. The portal skin must include CSS classes of these components or they will not be displayed correctly. A portlet skin does not need to be included as part of the portal skin and can be included within the portlets web application.

5.4.1.2. Best practices to customize a skin

The skin folder structure must be prepared once you start the design. Follow these conventions and best

practices to ease the integration of your design in eXo Platform.

5.4.1.3. Name files and folders

The id and class names are defined after the WebUI components name and portlets name with the 'UI-' as prefix. The same rule is applied for folder that contains components and portlets. It will help you find and edit correct files easily. For example, the UI portlet will be named as UIFooterPortlet, or UIBannerPortlet and the UI component will be named as UIToolbarContainer, or UIVerticalTab.

5.4.1.4. Folder structure

- **Portal skin:**

The portal skin will appear as a single link to a CSS file. This link will contain contents from all the portal skin classes merged into one file. This enables the portal skin to be transferred more quickly as a single file instead of many smaller files included with every page render.

The general folder structure for portal skin:

```
/webapp/skin/NameOfPortalSkin/portal
```

For example:

```
/webapp/skin/DefaultSkin/portal
```

The main entry CSS file:

The main entry CSS file should be placed right in the main portal skin folder. The file is the main entry point to the CSS class definitions for the skin:

```
/webapp/skin/NameOfPortalSkin/Stylesheet.css
```

For example:

```
/webapp/skin/SkinBlue/Stylesheet.css
```

The folder structure for WebUI components:

```
/webapp/skin/SkinBlue/webui/component/YourUIComponentName
```

For example:

```
/webapp/skin/SkinBlue/webui/component/UIToolbarContainer
```

Window decorator CSS is put in:

```
webapp/skin/PortletThemes/Stylesheet.css
```

Where to put images for portal skin?

The images for portal skin should be put in the background folder right in the Portal skin folder and for each UI component.

For example:

```
/webapp/skin/SkinBlue/webui/component/UIProfileUser/SkinBlue/background
```

In summary, the folder structure for a new portal skin should be:

webapp

- skin
 - NameOfPortalSkin
 - stylesheet.css
 - webui
 - component
 - UIComponentName
 - NameOfPortalSkin.css
 - NameOfPortalSkin
 - background

For example:

webapp

- skin
 - DefaultSkin
 - stylesheet.css
 - webui
 - UISpaceSearch
 - DefaultSkin.css
 - DefaultSkin
 - background

- **Portlet skin:**

Each portlet on a page may contribute its own style. The link to the portlet skin will only appear on the page if that portlet is loaded on the current page. A page may contain many portlet skin CSS links or none. The link ID will be named like {portletAppName}{PortletName}. For example, ContentPortlet in *content.war* will have the *id="contentContentPortlet"*.

General folder structure for portlet skin: */webapp/skin/portlet/webui/component/YourUIPortletName*

and for the Groovy skin: */webapp/groovy/portlet/webui/component/YourUIPortletName/*

For example:

- */webapp/skin/portlet/webui/component/UIBannerPortlet*
- */webapp/groovy/portlet/webui/component/UIBannerPortlet*

Portlet images folder: */webapp/skin/portlet/YourUIPortletName/PortalSkinName/background*

For example:

- */webapp/skin/portlet/UIBannerPortlet/BlueSkin/background*

Portlet themes

Main entry CSS:

- */webapp/skin/PortletThemes/Stylesheet.css*
- */webapp/skin/PortletThemes/background*
- */webapp/skin/PortletThemes/icons*

5.4.2. Configure skin in GateIn

GateIn provides support for skinning the entire User Interface (UI) of portal, including all common portal elements, custom skins and window decorator for individual portlets. Skins are designed to help you pack and reuse common graphic resources.

5.4.2.1. Select skins within the configuration files

The default skin can be set in the portal configuration files. The skin configured as default is used by GateIn as the administrator starts/restarts the server. To change the default skin, add a skin tag to the *portal.war/WEB-INF/conf/portal/portal/classic/portal.xml* configuration file. To change the skin to MySkin, do as follows:

```
<portal-config>
  <portal-name>classic</portal-name>
  <locale>en</locale>
  <access-permissions>Everyone</access-permissions>
  <edit-permission>*/platform/administrators</edit-permission>
  <skin>MySkin</skin>
  ...
</portal-config>
```

5.4.2.2. Skins in the page markup

The GateIn 3.2 skin not only contains CSS styles for the portal's components, but also shares components that may be reused in portlets. When GateIn 3.2 generates the page markup of portal, it inserts stylesheet links in the page's head tag. There are two main types of CSS links which appear in the head tag: one to the portal skin CSS file and the other to the portlet skin CSS file.

1. **Portal Skin** appears as a single link to a CSS file. This link contains contents from all portal skin classes merged into one file. The portal skin will be transferred more quickly as a single file instead of multiple

smaller files.

2. **Portlet Skin** only appears as the link on the page if that portlet is loaded on the current page. A page may contain many CSS links of portlet skins or none.

In the code fragment below, you can see two types of links:

```
<head>
...
<!-- The portal skin -->
<link id="CoreSkin" rel="stylesheet" type="text/css" href="/eXoResources/skin/Stylesheet.css" />
<!-- The portlet skins -->
<link id="web_FooterPortlet" rel="stylesheet" type="text/css"
  href="/web/skin/portal/webui/
component/UIFooterPortlet/DefaultStylesheet.css" />
<link id="web_NavigationPortlet" rel="stylesheet" type="text/css"
  href="/web/skin/portal/webui/
component/UINavigationPortlet/DefaultStylesheet.css" />
<link id="web_HomePagePortlet" rel="stylesheet" type="text/css"
  href="/portal/templates/skin/
webui/component/UIHomePagePortlet/DefaultStylesheet.css" />
<link id="web_BannerPortlet" rel="stylesheet" type="text/css"
  href="/web/skin/portal/webui/
component/UIBannerPortlet/DefaultStylesheet.css" />
...
</head>
```



Note

Window styles and portlet specification CSS classes are included within the portal skin.

5.4.2.3. SkinService

SkinService in GateIn 3.2 is to manage various types of skins. It is used to discover and deploy skins into the portal.

5.4.2.3.1. Configure skins

GateIn 3.2 automatically discovers web archives that contain a file descriptor for skins (WEB-INF/gatein-resources.xml). This file is to specify which portal, portlet and window decorators will be deployed into the skin service. The full schema can be found in the lib directory: *exo.portal.component.portal.jar/gatein_resources_1_0.xsd*. Here is an example where a skin (MySkin) with its CSS location is defined, and a few window decorator skins are specified:

```
<gatein-resources>
  <portal-skin>
    <skin-name>MySkin</skin-name>
    <css-path>/skin/myskin.css</css-path>
    <overwrite>false</overwrite>
  </portal-skin>

  <!-- window style -->
  <window-style>
    <style-name>MyThemeCategory</style-name>
    <style-theme>
      <theme-name>MyThemeBlue</theme-name>
    </style-theme>
    <style-theme>
      <theme-name>MyThemeRed</theme-name>
    </style-theme>
  </window-style>
</gatein-resources>
```


5.4.2.3.2. ResourceRequestFilter

With the Right-To-Left support, you need to retrieve all CSS files through a Servlet filter and to configure the web application to activate this filter. This has been already done for the eXoResources.war web application which contains the default skin. Any new web applications containing skinning CSS files need to have the following added to their web.xml:

```
<filter>
  <filter-name>ResourceRequestFilter</filter-name>
  <filter-class>org.exoplatform.portal.application.ResourceRequestFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>ResourceRequestFilter</filter-name>
  <url-pattern>*.css</url-pattern>
</filter-mapping>
```



Note

The configuration of Portlet Skin takes an optional parameter application-name, which is the web application wrapping skinned portlet. Hence, the display-name element in web.xml needs to be coherent with the application-name in the gatein-resources.xml file.

```
<portlet-skin>
  <!-- the application-name must be identical to display-name of the web application wrapping HomePagePortlet po
  <application-name>web</application-name>
  <portlet-name>HomePagePortlet</portlet-name>
  <skin-name>Default</skin-name>
  <css-path>/templates/skin/webui/component/UIHomePagePortlet/DefaultStylesheet.css</css-path>
</portlet-skin>
```

5.4.2.4. Default skin

The default skin of GateIn 3.2 is located as part of the eXoResource.war. The main files associated with the skin include:

- *WEB-INF/gatein-resources.xml*: defines the skin settings to use.
- *WEB-INF/web.xml*: contains the resource filter with the display-name set.
- *skin/Stylesheet.css*: contains the CSS class definitions for this skin.

The following block of CSS illustrates content of the *Stylesheet.css* file:

```
@import url(DefaultSkin/portal/webui/component/UIPortalApplicationSkin.css); (1)
@import url(DefaultSkin/webui/component/Stylesheet.css); (2)
@import url(PortletThemes/Stylesheet.css); (3)
@import url(Portlet/Stylesheet.css); (4)
```

In which:

- (1) Skin of portal page. UIPortalApplicationSkin.css defines CSS classes shared by all the portal pages.
- (2) Skins of various portal-owned components, such as WorkingWorkspace, MaskWorkspace, PortalForm, and more.

- (3) Window decorator skins.
- (4) The portlet specification CSS classes. (The CSS styles defined in Portlet Specification JSR286)

To make a default skin flexible and highly reusable, instead of defining all CSS classes in this file, CSS classes are arranged in nested stylesheet files, based on the `@import` statement. This makes easier for new skins to reuse parts of the default skin. To reuse a CSS stylesheet from the default portal skin, you need to refer to the default skin from `eXoResources`. For example, to include the window decorators from the default skin within a new portal skin, you need to use the following import:

```
@import url(/eXoResources/skin/Portlet/Stylesheet.css);
```



Note

When the portal skin is added to the page, it merges all CSS stylesheets into a single file.

5.4.2.5. Create new skins

5.4.2.5.1. Create new portal skins

5.4.2.5.1.1. Configure portal skins

You need to specify the new portal skin in the *gatein-resources.xml* file. You also need to specify the name of new skin, where to locate its CSS stylesheet file and whether to overwrite the existing portal theme with the same name.

```
<gatein-resources>
  <portal-skin>
    <skin-name>MySkin</skin-name>
    <css-path>/skin/myskin.css</css-path>
    <overwrite>false</overwrite>
  </portal-skin>
</gatein-resources>
```

The default portal skin and window styles are defined in the *eXoResources.war/WEB-INF/gatein-resources.xml* file.



Note

The CSS for the portal skin needs to contain CSS for all window decorators and portlet specification CSS classes.

5.4.2.5.1.2. Portal skin preview icon

You can preview the appearance of portal skin when selecting it. To display the preview image of deployed skins, the current skin must be aware of all those icons. Hence, each skin must contain preview images of all other skins.

For any portal skin, the paths to preview images are specified in CSS class `UIChangeSkinForm`:

- *eXoResources/src/main/webapp/skin/DefaultSkin/portal/webui/component/customization/UIChangeSkinForm/Stylesheet.*

For the portal named MySkin, it is required to define the following CSS classes:

```
.UIChangeSkinForm .UIItemSelector .TemplateContainer .MySkinImage
```

The default skin would be aware of skin icons if the preview screenshot is placed in:

- *eXoResources.war:/skin/DefaultSkin/portal/webui/component/customization/UIChangeSkinForm/background.*

The CSS stylesheet for the default portal needs to have the following updated with the preview icon CSS class. For the skin named MySkin, it is required to update the following:

- *eXoResources.war:/skin/DefaultSkin/portal/webui/component/customization/UIChangeSkinForm/Stylesheet.css.*

For now, amending the deployed package eXoResources is inevitable (modifying the default war/jar breaches development convention of GateIn-based products). The problem would be resolved in future GateIn versions in which different skin modules are fully independent, for example, there will be no preview image duplication.

```
.UIChangeSkinForm .UIItemSelector .TemplateContainer .MySkinImage {
margin: auto;
width: 329px; height:204px;
background: url('background/MySkin.jpg') no-repeat top;
cursor: pointer ;
}
```

5.4.2.5.2. Create new window styles

Window style is the CSS applied to the window decorator. When the administrator selects a new application to add to a page, he can decide which style of decorator surrounding the window if any.

5.4.2.5.2.1. Configure window styles

Window style is defined within the *gatein-resources.xml* file used by the SkinService to deploy the window style into the portal. Window styles can belong to a window style category. This category and window styles need to be specified in the resources file. The following *gatein-resource.xml* fragment will add MyThemeBlue and MyThemeRed to the MyTheme category.

```
<window-style>
  <style-name>MyTheme</style-name>
  <style-theme>
    <theme-name>MyThemeBlue</theme-name>
  </style-theme>
  <style-theme>
    <theme-name>MyThemeRed</theme-name>
  </style-theme>
</window-style>
```

The windows style of the default skin is configured in the *eXoResources.war/WEB-INF/gatein-resources.xml* file.



Note

When a window style is defined in the *gatein-resources.xml* file, it will be available to all portlets regardless of whether the current portal skin supports the window decorator or not. When a new window decorator is added, it should be added to all portal skins or the portal skins should share a

common stylesheet for window decorators.

5.4.2.5.2.2. Window style CSS

In order for the SkinService to display the window decorators, it must have CSS classes with the specific naming related to the window style name. The service will try and display CSS based on this naming. The CSS class must be included as part of the current portal skin for the window decorators to be displayed. The window decorator CSS classes for the default portal theme are located at *eXoResources.war/skin/PortletThemes/Stylesheet.css*.

5.4.2.5.2.3. Set the default window style

To set the default window style for a portal, you need to specify the CSS classes for a theme called DefaultTheme.



Note

You do not need to specify the DefaultTheme in the *gatein-resources.xml* file.

5.4.2.5.3. Create new portlet skins

Portlets often require additional styles that may not be defined by the portal skin. GateIn 3.2 defines additional stylesheets for each portlet and will append the corresponding link tags to the head. The ID attribute of <link> element will be in the *portletAppName/PortletName* form. For example, the ContentPortlet in content.war takes "content/ContentPortlet" as ID. To define a new CSS file to be included whenever a portlet is available on a portal page, the following fragment needs to be added in the *gatein-resources.xml* file.

```
<portlet-skin>
  <application-name>portletAppName</application-name>
  <portlet-name>PortletName</portlet-name>
  <skin-name>Default</skin-name>
  <css-path>/skin/DefaultStylesheet.css</css-path>
</portlet-skin>
<portlet-skin>
  <application-name>portletAppName</application-name>
  <portlet-name>PortletName</portlet-name>
  <skin-name>OtherSkin</skin-name>
  <css-path>/skin/OtherSkinStylesheet.css</css-path>
</portlet-skin>
```

This action will load DefaultStylesheet.css or OtherSkinStylesheet.css when the DefaultSkin or OtherSkin is used respectively.



Note

If the current portal skin is not defined as part of the supported skins, the portlet CSS class will not be loaded. The portlet skins should be updated whenever a new portal skin is created.

5.4.2.5.4. Change portlet icons

Each portlet can be represented by an unique icon that you can see in the portlet registry or page editor. This icon can be changed by adding an image to the directory of portlet web application: *skin/DefaultSkin/portletIcons/icon_name.png*. The icon must be named after the portlet. For example, the icon of account portlet must be named AccountPortlet and located at:

skin/DefaultSkin/portletIcons/AccountPortlet.png.



Note

You must use *skin/DefaultSkin/portletIcons/* for the directory to store the portlet icon regardless of using any skins.

5.4.2.5.5. Configure right-to-left skins

The SkinService handles stylesheet rewriting to accommodate the orientation. It works by appending *-lt* or *-rt* to the stylesheet name. For example, */web/skin/portal/webui/component/UIFooterPortlet/DefaultStylesheet-rt.css* will return the same stylesheet as */web/skin/portal/webui/component/UIFooterPortlet/DefaultStylesheet.css* but processed for the RT orientation. The *-lt* suffix is optional. Stylesheet authors can annotate their stylesheet to create content that depends on the orientation.

Example 1. This example uses the orientation to modify the float attribute that will make the horizontal tabs either float on left or on right:

```
float: left; /* orientation=lt */
float: right; /* orientation=rt */
font-weight: bold;
text-align: center;
white-space: nowrap;
```

The LT produced output will be:

```
float: left; /* orientation=lt */
font-weight: bold;
text-align: center;
white-space: nowrap;
```

The RT produced output will be:

```
float: right; /* orientation=rt */
font-weight: bold;
text-align: center;
white-space: nowrap;
```

Example 2. In this example, you need to modify the padding based on the orientation:

```
color: white;
line-height: 24px;
padding: 0px 5px 0px 0px; /* orientation=lt */
padding: 0px 0px 0px 5px; /* orientation=rt */
```

The LT produced output will be:

```
color: white;
line-height: 24px;
padding: 0px 5px 0px 0px; /* orientation=lt */
```

The RT produced output will be:

```
color: white;
line-height: 24px;
padding: 0px 0px 0px 5px; /* orientation=rt */
```

5.4.2.6. Override skins with extension

The extension mechanism of GateIn 3.2 enables the skin definition to be replaced with the skin resource configured in the extension-deployed web application. This is the example where the CSS path of default portal skin needs to be modified without touching the GateIn's files.

```
<gatein-resources>
  <portal-skin>
    <skin-name>MySkin</skin-name>
    <css-path>/skin/myskin.css</css-path>
    <overwrite>false</overwrite>
  </portal-skin>
</gatein-resources>
```

Do as follows:

1. Create a web application whose *gatein-resources.xml* contains the same content as the above xml block, except the element `<css-path>` is modified.
2. Register the artifact in the dependencies list of extended Portal Container.
3. Ensure that once the server has deployed the artifact, it does not load any web application with *gatein-resources.xml* configuring the same portal skin.

By the time those lines are written out, the control of loading order required in the third step relies totally on the Web Container Integration (WCI), hence the conflict may occur. That would be resolved by the concept of portal skin priority introduced in the coming version of GateIn.

5.4.3. Configure skin in WCM

5.4.3.1. Goal

This tutorial will help you to create a new layout and skin, and give some best practices you should know and respect when creating a new layout and skin for your portal and page.

5.4.3.2. Assumptions

- You have an extension named "MyPortal", and you need to create a site called "MySite" inside this portal.
- The `<myportal_path>` below is replaced by the based path of your "MyPortal" extension.
- If you see any folder/file which does not exist, you should create a new one.

5.4.3.3. Customize portal's layout



Note

This section is related to the configuration. You can see a sample [here](#). You can leave all the

portlet's preferences as blank, that means the default value will be taken and you do not need to care about it at this time.

- For example, you will have a layout like this:

In which:

- Branding: A branding application
- Top navigation: A top navigation application

A table column container with three nested containers:

- Left Column and Right Column: contain one application for each.
- Main content: contains the page body.

And here is the "short version" of *portal.xml*.

- `<myportal_path>/src/main/webapp/WEB-INF/conf/myportal/portal/portal/mysite/portal.xml`

```
<!-- ... -->

<portlet-application>
  <!-- Branding application. You can use WCM web content *exo:webContent* for the content and SCV portlet to display content -->
</portlet-application>

<portlet-application>
  <!-- navigation application. You can use WCM web content *exo:webContent* for the content and SCV portlet to display content -->
</portlet-application>

<container id="MySite" template="system:/groovy/portal/webui/container/UITableColumnContainer.gtmpl">
  <container id="LeftColumn" template="system:/groovy/portal/webui/container/UIContainer.gtmpl">
    <!-- One or more application(s) here -->
    <portlet-application>
    </portlet-application>
  </container>

  <container template="system:/groovy/portal/webui/container/UIContainer.gtmpl">
    <page-body>
    </page-body>
  </container>

  <container id="RightColumn" template="system:/groovy/portal/webui/container/UIContainer.gtmpl">
    <!-- One or more application(s) here -->
    <portlet-application>
    </portlet-application>
  </container>
</container>

<portlet-application>
  <!-- Footer application. You can use WCM web content *exo:webContent* for the content and SCV portlet to display content -->
</portlet-application>

<!-- ... -->
```

As you see in the *portal.xml* file above, every **container** tag has an **id** attribute, for example "`<container id = 'RightColumn'>`". When you create a CSS file, the property applied for this container should have the following name manner:

```
${container_id}TDContainer
```

and the details of this container:

```
RightColumnTDContainer
```

The reason is, when you have a look in the file system: */groovy/portal/webui/container/UITableColumnContainer.gtmpl* shown above, you will see this code fragment:

```
<table class="UITableColumnContainer"
  style="table-layout: fixed; margin: 0px auto;">
  <tr class="TRContainer">
    <% for(uiChild in uicomponent.getChildren()) {%>
      <td class="${uiChild.id}TDContainer TDContainer"><%
        uicomponent.renderUIComponent(uiChild) %></td> <% } %>
    </tr>
  </table>
```

So, in the **table** element (which represents the outer container), there are many **td** elements, each of which has the **class** attribute that equals to the **id** of the corresponding child component plus the "TDContainer" string literal.

To learn more, see [Customize portal and page's style](#).

5.4.3.4. Customize page's layouts



Note

This section is related to the configuration. You can see a sample [here](#). You can leave all the portlet's preferences as blank, that means the default value will be taken and you do not need to care about it at this time.

- Like *portal.xml*, you can define the layout for each page in your site as shown in the following example:

```
<!-- ... -->

<portlet-application> <!-- A custom document for content and SCV portlet to display -->
</portlet-application>

<portlet-application> <!-- A CLV portlet with a custom template. -->
</portlet-application>

<portlet-application> <!-- A CLV portlet with another custom template. -->
</portlet-application>

<!-- ... -->
```

5.4.3.5. Customize portal and page's style

If you want your skin to be applied for every page, do as follows:

1. Go to **Sites Explorer > Shared drive > CSS** folder.

2. Create a new CSS document which contains your stylesheet. You can use any name for this document and put a priority number.

If you want your skin to be applied for your **MySite** page only, do as follows:

1. Open the browser > **Sites Explorer** portlet > **Sites management** drive > **MySite/css** folder.
2. Create a new CSS document which contains your stylesheet for the portal and the page layout. You can use any name for this document and put a priority number.



Warning

This document should contain **ONLY** one stylesheet for the page and portal level.

The following is one sample stylesheet:

```
/* ... */
.LeftColumnTDContainer {
/* ... */

}

.RightColumnTDContainer {
/* ... */

}

/* ... */
```



Note

The order of applying CSS files (of site and web content) depends on their own **priority** property value. It means that we can apply the site CSS first and then web content CSS, or vice versa.

5.4.3.6. Customize CLV portlet's template



Note

This section is related to the configuration. You can see a sample [here](#).

- Apply your HTML/Groovy template code for this template.

For

example:

`<myportal_path>/src/main/webapp/WEB-INF/conf/myportal/customized/template/list/ACustomizedCLVTemplate.gtmpl`

```
<div id="$uicomponent.id" class="ACustomizedCLVTemplate">
  <div class="ListContents">
    <!-- something here -->
  </div>
</div>
```

- Now, you need to import this template to the database.

<myportal_path>/src/main/webapp/WEB-INF/conf/myportal/customized/template/configuration.xml

```
<external-component-plugins>
  <target-component>org.exoplatform.services.cms.views.ApplicationTemplateManagerService</target-component>
  <component-plugin>
    <name>ACustomizedCLVTemplate</name>
    <set-method>addPlugin</set-method>
    <type>org.exoplatform.services.cms.views.PortletTemplatePlugin</type>
    <description>This is a sample customized CLV template</description>
    <init-params>
      <value-param>
        <name>portletName</name>
        <value>Content List Viewer</value>
      </value-param>
      <value-param>
        <name>portlet.template.path</name>
        <value>war:/conf/myportal/customized/template</value>
      </value-param>
      <object-param>
        <name>default.folder.list.viewer</name>
        <description>Default folder list viewer groovy template</description>
        <object type="org.exoplatform.services.cms.views.PortletTemplatePlugin$PortletTemplateConfig">
          <field name="templateName">
            <string>ACustomizedCLVTemplate.gtmpl</string>
          </field>
          <field name="category">
            <string>list</string>
          </field>
        </object>
      </object-param>
    </init-params>
  </component-plugin>
</external-component-plugins>
```

5.4.3.7. Customize CLV template's style

To customize the CLV template's style, do as follows:

1. Go to **Sites explorer** portlet > **Sites management** drive > **MySite/css** folder.
2. Create a new CSS document which contains your stylesheet for the portal and the page layout. You can use any name for this document and put a priority number.



Warning

This document should contain **ONLY** the stylesheet for **THIS** template. If you have another template, you should create a new CSS document.

- The following is one sample stylesheet:

```
/* ... */
.ACustomizedCLVTemplate {
/* ... */

}

.ListContents {
/* ... */

}

/* ... */
```

3. Export the document and now you have an XML file.

Please check out [this tutorial](#) to know how to import this XML into database.

5.4.3.8. Customize Document's skin



Note

This section is related to the configuration. You can see a sample [here](#).

First, you need to create a new document definition.

- `<myportal_path>/src/main/webapp/WEB-INF/conf/myportal/customized/document/ACustomizedDocument.xml`

```
node type name :exo:customizedDocument
properties: exo:name(type : String), exo:title(type : String), exo:content(type : String)
```

You also need to configure it to make sure it is imported to the database.

- `<myportal_path>/src/main/webapp/WEB-INF/conf/myportal/customized/document/definition-configuration.xml`

```
<external-component-plugins>
  <target-component>org.exoplatform.services.jcr.RepositoryService</target-component>
  <component-plugin>
    <name>ACustomizedDocument</name>
    <set-method>addPlugin</set-method>
    <type>org.exoplatform.services.jcr.impl.AddNodeTypePlugin</type>
    <priority>200</priority>
    <init-params>
      <values-param>
        <name>autoCreatedInNewRepository</name>
        <description>ACustomizedDocument document definition</description>
        <value>war:/conf/myportal/customized/document/ACustomizedDocument.xml</value>
      </values-param>
    </init-params>
  </component-plugin>
</external-component-plugins>
```

Next, create the templates for this document, including:

- Dialog: see the sample [here](#)

`<myportal_path>/src/main/webapp/WEB-INF/conf/myportal/customized/document/dialog.gtmpl`

```
<div class="UIForm ACustomizedDocument">
  <% uiForm.begin() %>
  <!-- Document dialog content is here -->
  <% uiForm.end() %>
```

- View: see the sample [here](#)

`<myportal_path>/src/main/webapp/WEB-INF/conf/myportal/customized/document/view.gtmpl`

```
<style>
  <% _ctx.include(uicomponent.getTemplateSkin("exo:customizedDocument", "Stylesheet")); %>
</style>
<!-- Document view template content is here -->
```

- Stylesheet: see the sample [here](#)



Warning

This document should contain **ONLY** the stylesheet for **THIS** template.

`<myportal_path>/src/main/webapp/WEB-INF/conf/myportal/customized/document/stylesheet.css`

```
/* ... */

.ACustomizedDocument {
  /* ... */
}

/* ... */
```

- You also need to import them to database.

`<myportal_path>/src/main/webapp/WEB-INF/conf/myportal/customized/document/template-configuration.xml`

```
<external-component-plugins>
  <target-component>org.exoplatform.services.cms.templates.TemplateService</target-component>
  <component-plugin>
    <name>addTemplates</name>
    <set-method>addTemplates</set-method>
    <type>org.exoplatform.services.cms.templates.impl.TemplatePlugin</type>
    <init-params>
      <value-param>
        <name>autoCreateInNewRepository</name>
        <value>true</value>
      </value-param>
      <value-param>
        <name>storedLocation</name>
        <value>war:/conf/myportal/customized/document</value>
      </value-param>
      <value-param>
        <name>repository</name>
        <value>repository</value>
      </value-param>
      <object-param>
        <name>template.configuration</name>
        <description>configuration for the location of nodetypes templates to inject in jcr</description>
        <object type="org.exoplatform.services.cms.templates.impl.TemplateConfig">
          <field name="nodeTypes">
            <collection type="java.util.ArrayList">
              <value>
                <object type="org.exoplatform.services.cms.templates.impl.TemplateConfig$NodeType">
                  <field name="nodetypeName">
                    <string>exo:customizedDocument</string>
                  </field>
                  <field name="documentTemplate">
                    <boolean>true</boolean>
                  </field>
                  <field name="label">
                    <string>Customized Document</string>
                  </field>
                  <field name="referencedView">
                    <collection type="java.util.ArrayList">
                      <value>
                        <object type="org.exoplatform.services.cms.templates.impl.TemplateConfig$Template">
                          <field name="templateFile">
```

```

        <string>view.gtmpl</string>
      </field>
      <field name="roles">
        <string>*</string>
      </field>
    </object>
  </value>
</collection>
</field>
<field name="referencedDialog">
  <collection type="java.util.ArrayList">
    <value>
      <object type="org.exoplatform.services.cms.templates.impl.TemplateConfig$Template">
        <field name="templateFile">
          <string>dialog.gtmpl</string>
        </field>
        <field name="roles">
          <string>webdesigner:/platform/web-contributors</string>
        </field>
      </object>
    </value>
  </collection>
</field>
<field name="referencedSkin">
  <collection type="java.util.ArrayList">
    <value>
      <object type="org.exoplatform.services.cms.templates.impl.TemplateConfig$Template">
        <field name="templateFile">
          <string>stylesheet.css</string>
        </field>
        <field name="roles">
          <string>*</string>
        </field>
      </object>
    </value>
  </collection>
</field>
</object>
</value>
</collection>
</field>
</object>
</value>
</collection>
</init-params>
</component-plugin>
</external-component-plugins>

```

Finally, you should create some initial contents and export them to XML files.

To import this XML into database, you can set up the deployment like this:

```

<external-component-plugins>
  <target-component>org.exoplatform.services.wcm.deployment.WCMContentInitializerService</target-component>
  <component-plugin>
    <name>Content Initializer Service</name>
    <set-method>addPlugin</set-method>
    <type>org.exoplatform.services.wcm.deployment.plugins.XMLDeploymentPlugin</type>
    <description>XML Deployment Plugin</description>
    <init-params>
      <object-param>
        <name>ACME Logo data</name>
        <description>Deployment Descriptor</description>
        <object type="org.exoplatform.services.deployment.DeploymentDescriptor">
          <field name="target">
            <object type="org.exoplatform.services.deployment.DeploymentDescriptor$Target">
              <field name="repository">
                <string>repository</string>
              </field>
              <field name="workspace">
                <string>collaboration</string>
              </field>
              <field name="nodePath">
                <string>/sites content/live/acme/web contents/site artifacts</string>
              </field>
            </object>
          </field>
        </object>
      </object-param>
    </init-params>
  </component-plugin>
</external-component-plugins>

```

```

    </field>
    <field name="sourcePath">
      <string>war:/conf/wcm/artifacts/site-resources/acme/Logo.xml</string>
    </field>
  </object>
</object-param>
</init-params>
</component-plugin>
</external-component-plugins>

```

5.4.4. Create and apply Global stylesheet

Global stylesheet is the shared stylesheet which is applied to your entire site or a set of pages, depending on your configuration. When you want to make changes on your site, you only need to create a new global stylesheet, or edit the existing global stylesheet.

Global stylesheets of eXo Platform are put into the CSS folder to manage the stylesheet of your desired site. This section aims at showing you how to create and apply your own global stylesheet by **Content Explorer** and by configuration.

5.4.4.1. Create and apply the global stylesheet by Content Explorer

1. Select **My Spaces > Content Explorer > Sites Management**.
2. Select one site node in the **Sites Management** pane, for example **acme**, and then select the CSS folder.
3. Click **Add Document** to open the form to create the new global stylesheet.
4. Enter the name of global stylesheet into the **Name** field, for example **GlobalStylesheet_Organge**.
5. Set the value as "True" to activate your global stylesheet for your site in the **Active** field. By default, "True" will be set when you create a new global stylesheet. If you select "False", your newly created global style will be disabled.
6. Input one positive integer into the **Priority** field, for example "10".



Note

The successful application of your newly created global stylesheet depends on values in both **Active** and **Priority** fields. If "True" is set in many global stylesheets, the system will automatically render all global stylesheets in the CSS folder in the descending order and get the stylesheet with the highest priority. Thus, after selecting "True", you need to pay attention to the priority level so that the selected priority of your stylesheet is higher than those of other global stylesheets in the CSS folder.

The default global stylesheet will be automatically created in the CSS folder when you create a new site. However, this global stylesheet can be overwritten by either setting "False" for its **Active** field or setting the higher priority for other global stylesheet than that of the default global stylesheet.

7. Define your styles in the **CSS data** field. Here, you can directly enter your CSS rules, or copy and paste them from your favorite text editor.

For example, you can define your styles with the following information:

8. Click the **Save as Draft** button to save your newly created global stylesheet. You will see your global stylesheet in the **Sites Management** pane.

- To edit your global stylesheet or predefined global stylesheets, simply select the corresponding file and click **Edit Document** on the action bar to open the **Edit** form. To rename the document, right-click the corresponding global stylesheets in the **Sites Management** pane, then select **Rename**.

5.4.4.2. Create and apply the global stylesheet by configuration

After being created as above, your desired global stylesheet can be initialized automatically when the application is started by doing as follows:

1. Open the CSS folder.
2. Select **System** and click **Export Node** on the action bar.
3. Copy and paste the file you have exported, for example **StylesheetOrange.xml**, into the folder containing all stylesheets for your site, such as `"/acme-website/WEB-INF/conf/acme-portal/wcm/artifacts/site-resources/acme/"`.
4. Add the code below to the file where all global stylesheets will be initialized for your site, such as **acme-deployment-configuration.xml** in the folder `"/acme-website/WEB-INF/conf/acme-portal/wcm/deployment/"`.

```
<object-param>
  <name>ACME Stylesheet Green data</name>
  <description>Deployment Descriptor</description>
  <object type="org.exoplatform.services.deployment.DeploymentDescriptor">
    <field name="target">
      <object type="org.exoplatform.services.deployment.DeploymentDescriptor$Target">
        <field name="repository">
          <string>repository</string>
        </field>
        <field name="workspace">
          <string>collaboration</string>
        </field>
        <field name="nodePath">
          <string>/sites content/live/acme/css</string>
        </field>
      </object>
    </field>
    <field name="sourcePath">
      <string>war:/conf/acme-portal/wcm/artifacts/site-resources/acme/StylesheetOrange.xml</string>
    </field>
    <field name="cleanupPublication">
      <boolean>true</boolean>
    </field>
  </object>
</object-param>
```

5. Save the file where you have added the code in **Step 4** and clean data created in the previous start-up.
6. Start eXo Platform. After accessing the Content Explorer page, you will find your global stylesheet in the CSS folder of the relevant site.

5.4.4.3. Check display of global stylesheets

You can have several global stylesheets in one site. To see differences when applying various global stylesheets, for example **GlobalStylesheet_Blue** and **GlobalStylesheet_Orange**, do as follows:

1. Activate the **GlobalStylesheet_Blue** and **GlobalStylesheet_Orange** by turns.
2. Click **My Spaces** on the administration bar and select the relevant site, for example "acme".

If you activate **GlobalStylesheet_Blue**, your site will be displayed as below:

If you activate **GlobalStylesheet_Orange**, your site will be displayed in such a different way:

5.4.5. How to customize the Admin bar

5.4.5.1. Change the color scheme

The current color of Admin bar is dark blue and yellow. However, you can change the color to match your brand colors.

The Admin bar looks like this:

The style of Admin bar is defined in the **stylesheet.css** located in `src/main/webapp/skin/officeSkin/UIToolbarContainer`.

The CSS code below shows how to modify the Admin bar to look like this:

```
/*toolbar background color*/
.UIWorkingWorkspace .UIToolbarContainer .NormalContainerBlock .ToolbarContainer
{
    background: #20263f;
    height: 31px;
    border: none;
}

/*toolbar text color*/
.UIWorkingWorkspace .UIToolbarContainer .ToolbarContainer a.TBIcon {
    color: white;
    font-weight: normal;
    padding: 0 12px;
    display: block;
    white-space: nowrap;
    background: none;
    margin-left: 0;
    zoom: 1;
}

/*toolbar text color when hover*/
.UIWorkingWorkspace .UIToolbarContainer .ToolbarContainer a:hover {
    color: #8F8F8F;
}

/*background color of drop-down menu*/
.UIWorkingWorkspace .UIToolbarContainer .UITab .MenuItemContainer {
    border: 1px solid transparent;
    background: #20263f;
    margin-top: 0;
}

/*background color and border color of drop-down menu*/
.UIWorkingWorkspace .UIToolbarContainer .UITab .MenuItem {
    border-top: 1px solid #414760;
    border-bottom: none;
    background: #20263f;
    opacity: 1;
}
```



```

height: 28px;
line-height: 28px;
text-align: left;
}

/*text color of menu items*/
.UIWorkingWorkspace .UIToolbarContainer .ToolbarContainer .UITab .MenuItemContainer .MenuItem a
{
padding: 0 24px 0 35px; /* orientation=lt */
padding: 0 35px 0 24px; /* orientation=rt */
font-family: "Lucida Sans";
font-size: 12px;
color: white;
display: block;
font-weight: normal;
white-space: nowrap;
}

/*text color of menu items when hover*/
.UIWorkingWorkspace .UIToolbarContainer .ToolbarContainer .UITab .MenuItemContainer .MenuItem a:hover
{
color: #92918f;
background-color: transparent;
}

/*color of title bar text and background in drop-down menu*/
.UIWorkingWorkspace .UIToolbarContainer .TitleBar {
color: #20263f;
background: #92a9b9;
font-weight: bold;
padding: 0px 5px;
}

```

5.4.5.2. Change the content of the Admin bar

Portlets on Admin bar

The Admin bar is a special container composed of portlets. It is defined in *WEB-INF/conf/portal/portal/sharedlayout.xml*. In the illustration above, each circle represents a portlet defined in the Admin bar and configured in *sharedlayout.xml*.

It is possible to override the whole Admin bar by adding it into an extension at *custom-extension.war!WEB-INF/conf/portal/portal/sharedlayout.xml*.

The *sharedlayout.xml* file configures the current displayed portlets on the Admin bar. For example, to remove the dashboard menu, you will need to remove the following block:

```

...
<container id="UserToolBarDashboardPortlet" template="system:/groovy/portal/webui/container/UIContainer.gtmpl">
  <portlet-application>
    <portlet>
      <application-ref>exoadmin</application-ref>
      <portlet-ref>UserToolBarDashboardPortlet</portlet-ref>
    </portlet>
    <access-permissions>Everyone</access-permissions>
    <show-info-bar>false</show-info-bar>
  </portlet-application>
</container>
...

```

5.5. Add JavaScript to your portal

This can be done entirely within your extension by customizing the *gatein-resources.xml* configuration.

To add a JavaScript library, for example jQuery, create the *war:/WEB-INF/gatein-resources.xml_file*.

```
<javascript>
  <param>
    <js-module>jQuery</js-module>
    <js-path>/javascript/jquery.js</js-path>
    <js-priority>0</js-priority>
  </param>
</javascript>
```

In which:

- *<js-module>* is the namespace of your JavaScript.
- *<js-path>* is the path to your JavaScript file.
- *<js-priority>* is an optional tag. This tag is used to indicate the loading order of JavaScript files across all eXo Platform. Its value is of the integer type. If its value is not negative (≥ 0), the loading priority is sorted by the descending order. If its value is negative (< 0), the loading priority of the JavaScript file depends on the loading order of the web app (.war file) containing the JavaScript file.

Chapter 6. Work with Content

This chapter presents the issues related to the content in eXo Platform and you can know how to work with the content via the following topics:

- [Document types](#)
- [WCM templates](#)
- [Document type](#)
- [Dialog Syntax](#)
- [Taxonomies](#)

6.1. Document types

Each document type is represented by a node type in the JCR. Therefore, to create a new document type, you have to add a new node type. There are two ways to do this: through XML configuration files, or through the administration portlet.

To learn more about how to use the administration portlet, refer to the [Administration Guide](#).

Otherwise, you can create a new document type by configuring the file `/nodetypes-configuration.xml` in your extension.

```
<nodeType hasOrderableChildNodes="false" isMixin="true" name="exo:newnodetype" primaryItemName="">
  <supertypes>
    <supertype>exo:article</supertype>
  </supertypes>
  <propertyDefinitions>
    <propertyDefinition autoCreated="true" mandatory="true" multiple="false" name="text" onParentVersion="COPY">
      <valueConstraints/>
    </propertyDefinition>
    <propertyDefinition autoCreated="false" mandatory="true" multiple="false" name="date" onParentVersion="COPY">
      <valueConstraints/>
    </propertyDefinition>
  </propertyDefinitions>
</nodeType>
```

By defining a supertype, you can reuse other node types and extend them with more properties (just like inheritance in Object Oriented Programming).

6.2. WCM templates

The templates are applied to a node type or a metadata mixin type. There are two types of templates, including:

- **dialogs** are html forms that enable you to create node instances.
- **views** are html fragments used to display nodes.

From the ECM admin portlet, **Manage Template** lists all existing node types that have been associated to

Dialog and/or View templates. These templates can be attached to permissions (in the usual membership:group form), so that the specific one is displayed according to user rights (which can be useful in a content validation workflow activity).

6.3. Document type

The checkbox 'Document Type' is to define if the node type should be a Document Type or not. If this checkbox is selected, the Site Explorer considers such nodes as user content and applies the following behavior:

- The View template will be used to display the DocumentType nodes.
- The document types nodes can be created by the **Add Document** action. The forms shown in the **Add Document** action are the corresponding Dialog templates of each document type.
- Non-document types are hidden unless the 'Show non document types' option is checked.

Templates are written using Groovy Templates and will require some experiences with JCR API and HTML notions.

6.4. Dialog Syntax

Dialogs are Groovy Templates that generate forms by mixing static HTML fragments and Groovy calls to the components responsible for building the UI at runtime. As a result, you will get a simple but powerful syntax.

6.4.1. Interceptors

By placing interceptors in your template, you will be able to execute a Groovy script just before and just after saving the node. Pre-save interceptors are mostly used to validate input values and their overall meaning while the post-save interceptor can be used to do some manipulations or references for the newly created node, such as binding it with a forum discussion or wiki space.

To place interceptors, use the following fragment:

```
<% uicomponent.addInterceptor("ecm-explorer/interceptor/PreNodeSaveInterceptor.groovy", "prev");%>
```

Interceptor Groovy scripts are managed in the 'Manage Script' section in the ECM admin portlet. They must implement the CmsScript interface. Pre-save interceptors obtain input values within the context:

```
public class PreNodeSaveInterceptor implements CmsScript {

    public PreNodeSaveInterceptor() {
    }

    public void execute(Object context) {
        Map inputValues = (Map) context;
        Set keys = inputValues.keySet();
        for(String key : keys) {
            JcrInputProperty prop = (JcrInputProperty) inputValues.get(key);
            println("    --> "+prop.getJcrPath());
        }
    }

    public void setParams(String[] params) {
    }
}
```

```
}
}
```

Whereas the post-save interceptor is passed the path of the saved node in the context:

```
<% uicomponent.addInterceptor("ecm-explorer/interceptor/PostNodeSaveInterceptor.groovy", "post");%>

public class PostNodeSaveInterceptor implements CmsScript {

    public PostNodeSaveInterceptor() {
    }

    public void execute(Object context) {
        String path = (String) context;

        println("Post node save interceptor, created node: "+path);
    }

    public void setParams(String[] params) {
    }
}
```

6.4.2. Hidden fields

In the next code sample, each argument is composed of a set of keys and values. The order of arguments are not important and only the key matters. That example defines a field with the id as "hiddenField2", which will generate a hidden field. The value of this field will be automatically set to UTF-8 and no visible field will be printed on the form.

```
String[] hiddenField2 = ["jcrPath=/node/jcr:content/jcr:encoding", "visible=false", "UTF-8"];
uicomponent.addHiddenField("hiddenInput2", hiddenField2);
```

Once the form has been saved, the date value will be saved under the relative JCR path `./exo:image/jcr:lastModified`.

6.4.2.1. Non-value field

You cannot either see the non-value field on the form or input value for them. Its value will be automatically created or defined when you are managing templates.

```
String[] hiddenField1 = ["jcrPath=/node/jcr:content", "nodetype=nt:resource", "mixintype=dc:elementSet", "visible=false"];
uicomponent.addHiddenField("hiddenInput1", hiddenField1);
```

6.4.2.2. Non-editable fields

It is possible to create widgets that are non-editable (and then only used to print some information).

```
String[] fieldCategories = ["jcrPath=/node/exo:category", "multiValues=true", "reference=true", "editable=false"];
```

6.4.2.3. Create node type or mixin type

In many cases, when creating an instance where the node is out of form, you must still specify the CMS service about the node structure. Particularly, you must define if which node type is child of the newly created node or

if the current node has any mixin type attributed.

By defining these arguments, the node and its children are created with the correct node type and mixin type.

See the following example:

```
String[] hiddenField = ["jcrPath=/node/jcrcontent", "nodetype=nt:resource", "mixintype=exo:rss-enable", "visible=if-not-null"];
uicomponent.addHiddenField("hiddenInput", hiddenField) ;
```

6.4.2.4. Hidden field with default value

In the previous sample, the value was automatically created and set according to the current date. However, it is also possible to set a default value for a field.

```
String[] hiddenField = ["jcrPath=/node/jcrcontent/jcr:mimeType", "image/jpeg"] ;
uicomponent.addHiddenField("hiddenInput", hiddenField) ;
```

6.4.2.5. Visible without null fields

It is possible to tell that a widget should be visible only if its value is not null or when the form is used to edit the node which has been existing.

```
String nameArgs[] = ["jcrPath=/node", "mixintype=mix:votable", "visible=if-not-null"];
uicomponent.addMixinField("name", nameArgs )
```

6.4.2.6. WYSIWYG widget

Widgets are natively part of the Platform product to provide a simple and easy way for users to get information and notification on their application. They complete the portlet application that focuses on more transactional behaviors.

The "What You See Is What You Get" widget is one of the most powerful tools. It renders an advanced JavaScript text editor with many functionalities, including the ability to dynamically upload images or flash assets into a JCR workspace and then to refer to them from the created HTML text.

```
String[] fieldSummary = ["jcrPath=/node/exo:summary", "options=basic"] ;
uicomponent.addWYSIWYGField("summary", fieldSummary) ;
```

```
String[] fieldContent = ["jcrPath=/node/exo:text", "options=toolbar:CompleteWCM,'height:410px'", ""] ;
uicomponent.addRichtextField("content", fieldContent)
```

The "options" argument is used to tell the component which toolbar should be used.

By default, there are three options for the toolbar:

- basic: a minimal set of tools is shown.
- default: a large set of tools is shown, no "options" argument is needed in that case.
- CompleteWCM: a full set of tools is shown.

There is also a simple text area widget, which has text-input area only:

```
String [] descriptionArgs = ["jcrPath=/node/exo:title", "validate=empty"];
uicomponent.addTextAreaField("description", descriptionArgs);
```

6.4.2.7. Simple select box widget

The select box widget enables you to render a select box with static values. These values are enumerated in a comma-separated list in the "options" argument. The argument with no key (here "text/html") is selected by default.

```
String[] mimetype = ["jcrPath=/node/jcrcontent/jcr:mimeType", "text/html", "options=text/html,text/plain"];
uicomponent.addSelectBoxField("mimetype", mimetype);
```

As usual, the value will be stored at the relative path defined by the jcrPath directive argument.

For more examples on how to create WCM templates, refer to [Reference Guide](#).

6.4.2.8. Advanced dynamic select box

In many cases, the previous solution with static options is not good enough and one would like to have the select box checked dynamically. That is what eXo Platform provide thanks to the introduction of a Groovy script as shown in the code fragment below.

```
String[] args = ["jcrPath=/node/exodestWorkspace", "script=ecm-explorer/widget/FillSelectBoxWithWorkspaces:groovy"];
uicomponent.addSelectBoxField("destWorkspace", args);
```

The script itself implements the CMS Script interface and the cast is done to get the select box object as shown in the script code which fills the select box with the existing JCR workspaces.

```
import java.util.List ;
import java.util.ArrayList ;

import org.exoplatform.services.jcr.RepositoryService;
import org.exoplatform.services.jcr.core.ManageableRepository;

import org.exoplatform.webui.form.UIFormSelectBox;
import org.exoplatform.webui.core.model.SelectItemOption;
import org.exoplatform.services.cms.scripts.CmsScript;

public class FillSelectBoxWithWorkspaces implements CmsScript {

    private RepositoryService repositoryService_;

    public FillSelectBoxWithWorkspaces(RepositoryService repositoryService) {
        repositoryService_ = repositoryService;
    }

    public void execute(Object context) {
        UIFormSelectBox selectBox = (UIFormSelectBox) context;

        ManageableRepository jcrRepository = repositoryService_.getRepository();
        List options = new ArrayList();
        String[] workspaceNames = jcrRepository.getWorkspaceNames();
        for(name in workspaceNames) {
            options.add(new SelectItem(name, name));
        }
        selectBox.setOptions(options);
    }

    public void setParams(String[] params) {
```

```
}
}
```



Note

It is also possible to provide a parameter to the script by using the argument "scriptParams".

6.4.2.9. Widget with selector

One of the most advanced functionalities of this syntax is the ability to plug your own component that shows an interface, enabling you to select the value of the field.

In the generated form, you will see an icon which is configurable thanks to the selectorIcon argument. The syntax is a bit more complex but not much.

```
String[] groupArgs = ["jcrPath=/node/exogroup", "selectorClass=org:exoplatform:ecm:webui:selector:UIGroupMemberS
uicomponent.addActionField("group", groupArgs);
```

You can plug your own component using the selectorClass argument. It must follow the eXo UIComponent mechanism and implements the interface ComponentSelector:

```
package org.exoplatform.ecm.webui.selector;

import org.exoplatform.webui.core.UIComponent;
public interface ComponentSelector {
    public UIComponent getSourceComponent() ;
    public void setSourceComponent(UIComponent uicomponent, String[] initParams) ;
}
```

6.4.2.10. Multi-valued widget

A widget can have multiple values if you add the argument "multiValues=true" to the directive.

6.5. Manage template service

The Template service enables you to create dialogs and view templates for each node type registered. Each node type may have many dialogs and view templates. The template will be used when creating or viewing nodes.

.../webapps/portal/WEB-INF/conf/ecm/ecm-templates-configuration.xml

```
<component>
  <key>org.exoplatform.services.cms.templates.TemplateService</key>
  <type>org.exoplatform.services.cms.templates.impl.TemplateServiceImpl</type>
  .....
</component>
```

As usual, one can register a plugin inside the service. This plugin initializes default dialogs and views template of any node type as nt:file, exo:article, exo:workflowAction, exo:sendMailAction, and more.

```
<component-plugins>
  <component-plugin>
```



```

<name>addTemplates</name>
<set-method>addTemplates</set-method>
<type>org.exoplatform.services.cms.templates.impl.TemplatePlugin</type>
.....
</component-plugin>
</component-plugins>

```

With init-parameters as:

```

<init-params>
  <value-param>
    <name>autoCreateInNewRepository</name>
    <value>true</value>
  </value-param>
  <value-param>
    <name>storedLocation</name>
    <value>war:/conf/ecm/artifacts/templates</value>
  </value-param>
  <value-param>
    <name>repository</name>
    <value>repository</value>
  </value-param>
  <object-param>
    <name>template.configuration</name>
    <description>configuration for the localtion of templates to inject in jcr</description>
    <object type="org.exoplatform.services.cms.templates.impl.TemplateConfig">
      <field name="nodeTypes">
        <collection type="java.util.ArrayList">
          <value>
            <object type="org.exoplatform.services.cms.templates.impl.TemplateConfig$NodeType">
              <field name="nodetypeName">
                <string>exo:article</string>
              </field>
              <field name="documentTemplate">
                <boolean>true</boolean>
              </field>
              <field name="label">
                <string>Article</string>
              </field>
              <field name="referencedView">
                <collection type="java.util.ArrayList">
                  <value>
                    <object type="org.exoplatform.services.cms.templates.impl.TemplateConfig$Template">
                      <field name="templateFile">
                        <string>/article/views/view1.gtmpl</string>
                      </field>
                      <field name="roles">
                        <string>*</string>
                      </field>
                    </object>
                  </value>
                </collection>
              </field>
              <field name="referencedDialog">
                <collection type="java.util.ArrayList">
                  <value>
                    <object type="org.exoplatform.services.cms.templates.impl.TemplateConfig$Template">
                      <field name="templateFile">
                        <string>/article/dialogs/dialog1.gtmpl</string>
                      </field>
                      <field name="roles">
                        <string>*</string>
                      </field>
                    </object>
                  </value>
                </collection>
              </field>
            </object>
          </value>
        </collection>
      </field>
    </object-param>
  </init-params>

```

6.6. Taxonomies

Taxonomy is a particular classification arranged in a hierarchical structure. Taxonomy trees in eXo Platform will help you organize your content into categories.

When you create a new taxonomy tree, you will add a pre-configured *exo:action* (*exo:scriptAction* or *exo:businessProcessAction*) to the root node of the taxonomy tree. This action is triggered when a new document is added anywhere in the taxonomy tree. The default action moves the document to the physical storage location and replaces the document in the taxonomy tree with a symlink of the *exo:taxonomyLink* type pointing to it. The physical storage location is defined by a workspace name, a path and the current date and time.

Like adding document types, taxonomy trees can be managed through the Administration portlet, or by adding XML files.

To configure taxonomy trees by adding configuration files in the `/webapp/WEB-INF/conf/acme-portal/wcm/taxonomy/` directory, create a new file called *\$taxonomyName-taxonomies-configuration.xml*. For example, if the name of your taxonomy tree is "acme", the file should be named *acme-taxonomies-configuration.xml*.

You can view the file here:
\$PLF-HOME_/samples/acme-website/webapp/src/main/webapp/WEB-INF/conf/acme-portal/wcm/taxonomy/acme-taxonom

As you can see, the value-params enable you to define the repository, workspace, name of the tree and its JCR path. You can then configure permissions for each group of users in the portal, and the triggered action when a new document is added to the taxonomy tree. Finally, you can describe the structure and names of the categories inside your taxonomy tree.

Chapter 7. Work with Applications

Applications play an important role in each eXo service and so it is necessary for you to further understand about them.

This chapter will help you know how to integrate an application into your portal and how to develop your own application:

- [Application integration](#)
- [Develop your own applications](#)
 - [Gadget vs Portlet](#)
 - [Portlet Bridges](#)
 - [Portlet Bridges](#)

7.1. Application integration

To add a portlet to one of your portal's pages, you should configure the *pages.xml* file located at */war/src/main/webapp/WEB-INF/conf/sample-ext/portal/portal/classic/*.

Here is an example of the portlet configuration inside *pages.xml*:

```
<portlet-application>
  <portlet>
    <application-ref>presentation</application-ref>
    <portlet-ref>SingleContentViewer</portlet-ref>
    <preferences>
      <preference>
        <name>repository</name>
        <value>repository</value>
        <read-only>>false</read-only>
      </preference>
      <preference>
        <name>workspace</name>
        <value>collaboration</value>
        <read-only>>false</read-only>
      </preference>
      <preference>
        <name>nodeIdentifier</name>
        <value>/sites content/live/acme/web contents/site artifacts/Introduce</value>
        <read-only>>false</read-only>
      </preference>
      <!-- ... -->
    </preferences>
  </portlet>
  <title>Homepage</title>
  <access-permissions>Everyone</access-permissions>
  <show-info-bar>>false</show-info-bar>
  <show-application-state>>false</show-application-state>
  <show-application-mode>>false</show-application-mode>
</portlet-application>
```

Details:

| XML tag name | Description |
|-----------------|---|
| application-ref | The name of the webapp that contains the portlet. |

| XML tag name | Description |
|------------------------|---|
| portlet-ref | The name of the portlet. |
| title | The title of the page in HTML speaking. |
| access-permission | Define who can access the portlet. |
| show-info-bar | Show the top bar with the portlet title. |
| show-application-state | Show the collapse/expand icons. |
| show-application-mode | Show the change portlet mode icon. |
| preferences | Contain a list of <i>preferences</i> specific to each portlet. Each <i>preference</i> has a <i>name</i> and a <i>value</i> . You can also lock it by setting the <i>read-only</i> element to true. To learn more, refer to eXo JCR and Extension Services Reference . |

7.2. Develop your own applications

7.2.1. Gadget vs Portlet

It is important to understand distinctions between gadgets and portlets. Portlets are user interface components that provide fragments of markup code from the server side, while gadgets generate dynamic web content on the client side. With Gadgets, small applications can be built quickly, and mashed up on the client side using the lightweight Web-Oriented Architecture (WOA) technologies, like REST or RSS.

For more information on how to develop gadgets and portlets, see in the GateIn Reference Guide:

- [Portlet development](#)
- [Gadget development](#)

7.2.2. Gadget development quickstart with eXo IDE

eXo Platform facilitates easy gadget development, via its powerful, web-based IDE. You can learn more about the basic principles of gadget development [here](#).

7.2.3. Portlet Bridges

The Portlet Bridge is an adapter for a web framework to the portlet container runtime. It works ideally with framework that does not expose the servlet container with the limited support for the full portlet API.

The Java™ Specification Request 168 Portlet Specification (JSR 168) standardizes how components for portal servers are developed. This standard has industry backing from major portal server vendors. A Portlet Bridge allows you to create a JSR-168 compliant portlet with very little change on your existing web application.

For example, the JSF Bridge allows you to transparently deploy your existing JSF Applications as a Portlet Application or Web Application.

<http://wiki.apache.org/myfaces/PortletBridge>

The JBoss implementation of the Portlet Bridge has enhancements to support other web frameworks, such as RichFaces and Seam.

<http://jboss.org/portletbridge/docs.html>

Chapter 8. System Integration

This chapter will show you how to integrate eXo Platform 3.5 into your information system through the specific topics below:

- [Authentication](#)
 - [Single-Sign-On \(SSO\)](#)
 - [Kerberos SSO on Active Directory](#)
- [Users integration](#)
 - [Organization Service](#)
 - [Memberships, Groups and Users](#)
 - [Organization API](#)
- [LDAP Integration](#)
 - [Connection Settings](#)
 - [Active Directory sample configuration](#)
 - [Active Directory sample configuration](#)
- [Email](#)

8.1. Authentication

8.1.1. Single-Sign-On (SSO)

When logging into the portal, you can gain access to many systems through portlets using a single identity. However, in many cases, the portal infrastructure must be integrated with other SSO-enabled systems. There are many different Identity Management solutions available. The GateIn documentation gives detailed configuration for different SSO implementation. For more details, see [Single Sign On](#).

8.1.2. Central Authentication Service (CAS)

Central Authentication Service (CAS) is a Web Single-Sign-On (WebSSO), developed by JA-SIG as an open source project. CAS enables you to work on different applications to log in only once and to be recognized and authenticated by all applications. Details about CAS can be found [here](#).

The CAS integration consists of two steps:

- Installing or configuring the CAS server.
- Setting up the portal to use the CAS server.

For more information on CAS configuration, refer [here](#).

8.1.3. Kerberos SSO on Active Directory

eXo Portal 3.5 supports the Kerberos authentication on a Microsoft Active Directory. You will need to configure both your Active Directory server and the application server.

The implementation makes possible to use SPNEGO or NTLM. The client will get two authentication headers, including **Negotiate** and **NTLM** and will use whichever supported by the browser. In Firefox, it is possible to manage authentication types, but it is not in Internet Explorer; therefore, SPNEGO will be used.

To learn more about how to configure Kerberos SSO, see [here](#).

8.2. Users integration

8.2.1. Organization Service

To specify the initial Organization configuration, the content of your extension.war in */WEB-INF/conf/organization/organization-configuration.xml* should be edited. This file uses the portal XML configuration schema. It lists several configuration plugins.

The plugin of *org.exoplatform.services.organization.OrganizationDatabaseInitializer* type is used to specify a list of membership types, groups, and users to be created.

8.2.2. Memberships, Groups and Users

The predefined membership types are specified in the **membershipType** field of the OrganizationConfig plugin parameter.

```
<field name="membershipType">
  <collection type="java.util.ArrayList">
    <value>
      <object type="org.exoplatform.services.organization.OrganizationConfig$MembershipType">
        <field name="type">
          <string>member</string>
        </field>
        <field name="description">
          <string>member membership type</string>
        </field>
      </object>
    </value>
    <value>
      <object type="org.exoplatform.services.organization.OrganizationConfig$MembershipType">
        <field name="type">
          <string>owner</string>
        </field>
        <field name="description">
          <string>owner membership type</string>
        </field>
      </object>
    </value>
  </collection>
</field>
```

The predefined groups are specified in the **group** field of the OrganizationConfig plugin parameter.

```
<field name="group">
  <collection type="java.util.ArrayList">
    <value>
      <object type="org.exoplatform.services.organization.OrganizationConfig$Group">
```

```

    <field name="name">
      <string>portal</string>
    </field>
    <field name="parentId">
      <string/>
    </field>
    <field name="type">
      <string>hierachy</string>
    </field>
    <field name="description">
      <string>the /portal group</string>
    </field>
  </object>
</value>
<value>
  <object type="org.exoplatform.services.organization.OrganizationConfig$Group">
    <field name="name">
      <string>community</string>
    </field>
    <field name="parentId">
      <string>/portal</string>
    </field>
    <field name="type">
      <string>hierachy</string>
    </field>
    <field name="description">
      <string>the /portal/community group</string>
    </field>
  </object>
</value>
  ...
</collection>
</field>

```

The predefined users are specified in the **membershipType** field of the OrganizationConfig plugin parameter.

```

<field name="user">
  <collection type="java.util.ArrayList">
    <value>
      <object type="org.exoplatform.services.organization.OrganizationConfig$User">
        <field name="userName">
          <string>root</string>
        </field>
        <field name="password">
          <string>exo</string>
        </field>
        <field name="firstName">
          <string>root</string>
        </field>
        <field name="lastName">
          <string>root</string>
        </field>
        <field name="email">
          <string>exoadmin@localhost</string>
        </field>
        <field name="groups">
          <string>member:/admin,member:/user,owner:/portal/admin</string>
        </field>
      </object>
    </value>
    <value>
      <object type="org.exoplatform.services.organization.OrganizationConfig$User">
        <field name="userName">
          <string>exo</string>
        </field>
        <field name="password">
          <string>exo</string>
        </field>
        <field name="firstName">
          <string>site</string>
        </field>
        <field name="lastName">
          <string>site</string>
        </field>
        <field name="email">
          <string>exo@localhost</string>
        </field>
      </object>
    </value>
  </collection>
</field>

```



```

    </field>
    <field name="groups">
      <string>member:/user</string>
    </field>
  </object>
</value>
...
</collection>
</field>

```

8.2.3. Organization API

The *exo.platform.services.organization* package has five main components: user, user profile, group, membership type and membership. There is an additional component that serves as an entry point into Organization API - *OrganizationService* component which provides handling functionality for the five components. For more details, take a look at the [GateIn documentation](#).

8.3. LDAP Integration

If you have an existing LDAP server, the eXo predefined settings will likely not match your directory structure. eXo LDAP organization service implementation was written with flexibility in mind and can certainly be configured to meet your requirements.

The configuration is done in the *ldap-configuration.xml* file, and this part will explain the numerous parameters which it contains.

8.3.1. Connection Settings

First, start by connection settings which will tell eXo how to connect to your directory server. These settings are very close to the [JNDI API](#) context parameters. This configuration is activated by the init-param *ldap.config* of service *LDAPServiceImpl*.

```

<component>
  <key>org.exoplatform.services.ldap.LDAPService</key>
  <type>org.exoplatform.services.ldap.impl.LDAPServiceImpl</type>
  <init-params>
    <object-param>
      <name>ldap.config</name>
      <description>Default ldap config</description>
      <object type="org.exoplatform.services.ldap.impl.LDAPConnectionConfig">
        <field name="providerURL">
          <string>ldap://127.0.0.1:389,10.0.0.1:389</string>
        </field>
        <field name="rootdn">
          <string>CN=Manager,DC=exoplatform,DC=org</string>
        </field>
        <field name="password">
          <string>secret</string>
        </field>
        <!-- field name="authenticationType"><string>simple</string></field -->
        <field name="version">
          <string>3</string>
        </field>
        <field name="referralMode">
          <string>follow</string>
        </field>
        <!-- field name="serverName"><string>active.directory</string></field -->
      </object>
    </object-param>
  </init-params>
</component>

```

- **providerURL**: LDAP server URL (see [PROVIDERURL](#)). For multiple LDAP servers, use comma separated list of host:port (For example, [ldap://127.0.0.1:389](#),10.0.0.1:389).
- **rootdn**: distinguished name of user that will be used by the service to authenticate on the server (see [SECURITYPRINCIPAL](#)).
- **password**: password for user *rootdn* (see [SECURITYCREDENTIALS](#)).
- **authenticationType**: type of authentication to be used (see [SECURITYAUTHENTICATION](#)). Use one of *none*, *simple*, *strong*. Default is *simple*.
- **version**: LDAP protocol version (see [java.naming.ldap.version](#)). Set to 3 if your server supports LDAP V3.
- **referralMode**: one of *follow*, *ignore*, *throw* (see [REFERRAL](#)).
- **serverName**: you will need to set this to *active.directory* to work with Active Directory servers. Any other value will be ignored and the service will act as on a standard LDAP.

8.3.2. Organization Service Configuration

Next, you need to configure the eXo **OrganizationService** to show how the directory is structured and how to interact with it. This is managed by a couple of init-params: **ldap.userDN.key** and **ldap.attribute.mapping** in file **ldap-configuration.xml** (by default located at *portal.war/WEB-INF/conf/organization*)

```
<component>
  <key>org.exoplatform.services.organization.OrganizationService</key>
  <type>org.exoplatform.services.organization.ldap.OrganizationServiceImpl</type>
  [...]
  <init-params>
    <value-param>
      <name>ldap.userDN.key</name>
      <description>The key used to compose user DN</description>
      <value>cn</value>
    </value-param>
    <object-param>
      <name>ldap.attribute.mapping</name>
      <description>ldap attribute mapping</description>
      <object type="org.exoplatform.services.organization.ldap.LDAPAttributeMapping" />
      [...]
    </object-param>
  </init-params>
  [...]
</component>
```

ldap.attribute.mapping maps your LDAP to eXo. At first, there are two main parameters to configure in it:

```
<field name="baseURL">
  <string>dc=exoplatform,dc=org</string>
</field>
<field name="ldapDescriptionAttr">
  <string>description</string>
</field>
```

- **baseURL**: root dn for eXo organizational entities. This entry cannot be created by eXo and must have existed in the directory already.
- **ldapDescriptionAttr**: Name of a common attribute that will be used as description for groups and membership types.



Warning

In Core, the `ldapDescriptionAttr` key is present but not consistently used everywhere in code. When using **Core**, consider that the description is always mapped to the 'description' attribute.

Other parameters are discussed in the following sections.

8.3.2.1. Users

8.3.2.1.1. Main parameters

Here are the main parameters to map eXo users to your directory:

```

<field name="userURL">
  <string>ou=users,ou=portal,dc=exoplatform,dc=org</string>
</field>
<field name="userObjectClassFilter">
  <string>objectClass=person</string>
</field>
<field name="userLDAPClasses">
  <string>top,person,organizationalPerson,inetOrgPerson</string>
</field>

```

- **userURL**: base dn for users. Users are created in a flat structure under this base with a **dn** of the form: **ldap.userDN.key=username,userURL**.

For example:

```
uid=john,cn=People,o=MyCompany,c=com
```

However, if users exist deeply under *userURL*, eXo will be able to retrieve them.

Example:

```
uid=tom,ou=France,ou=EMEA,cn=People,o=MyCompany,c=com
```

- **userObjectClassFilter**: Filter used under *userURL* branch to distinguish eXo user entries from others.

Example: *john* and *tom* will be recognized as valid eXo users but *EMEA* and *France* entries will be ignored in the following subtree:

```

uid=john,cn=People,o=MyCompany,c=com
  objectClass: person
  ...
ou=EMEA,cn=People,o=MyCompany,c=com
  objectClass: organizationalUnit
  ...
  ou=France,ou=EMEA,cn=People,o=MyCompany,c=com
    objectClass: organizationalUnit
    ...
      uid=tom,ou=EMEA,cn=People,o=MyCompany,c=com
        objectClass: person
        ...

```

- **userLDAPClasses:** commas are used to separate list of classes used for creating users.

When a new user is created, an entry will be created with the given *objectClass* attributes. The classes must at least define *cn* and any attribute referenced in the user mapping.

For example, adding the user *Marry Simons* could produce:

```
uid=marry,cn=users,ou=portal,dc=exoplatform,dc=org
objectclass: top
objectClass: person
objectClass: organizationalPerson
objectClass: inetOrgPerson
...
```

8.3.2.1.2. User mapping

The following parameters maps LDAP attributes to eXo User Java objects attributes.

```
<field name="userUsernameAttr">
  <string>uid</string>
</field>
<field name="userPassword">
  <string>userPassword</string>
</field>
<field name="userFirstNameAttr">
  <string>givenName</string>
</field>
<field name="userLastNameAttr">
  <string>sn</string>
</field>
<field name="userDisplayNameAttr">
  <string>displayName</string>
</field>
<field name="userMailAttr">
  <string>mail</string>
</field>
```

- **userUsernameAttr:** username (login)
- **userPassword:** password (used when the portal authentication is done by eXo login module)
- **userFirstNameAttr:** first name
- **userLastNameAttr:** last name
- **userDisplayNameAttr:** display name
- **userMailAttr:** email address

In the example above, the user *Marry Simons* could produce:

```
uid=marry,cn=users,ou=portal,dc=exoplatform,dc=org
userPassword: XXXX
givenName: Marry
sn: Simons
displayName: Marry Simons
mail: marry.simons@example.org
uid: marry
...
```

8.3.2.2. Groups

eXo Platform groups can be mapped to organizational or applicative groups defined in your directory.

```
<field name="groupsURL">
  <string>ou=groups,ou=portal,dc=exoplatform,dc=org</string>
</field>
<field name="groupLDAPClasses">
  <string>top,organizationalUnit</string>
</field>
<field name="groupObjectClassFilter">
  <string>objectClass=organizationalUnit</string>
</field>
```

- **groupsURL**: base dn for eXo groups

Groups can be structured hierarchically under *groupsURL*. For example, groups, including *communication*, *communication/marketing* and *communication/press*, would map to:

```
ou=communication,ou=groups,ou=portal,dc=exoplatform,dc=org
...
ou=marketing,ou=communication,ou=groups,ou=portal,dc=exoplatform,dc=org
...
ou=press,ou=communication,ou=groups,ou=portal,dc=exoplatform,dc=org
...
```

- **groupLDAPClasses**: commas are used to separate list of classes used for group creation.

When a new group is created, an entry will be also created with the given objectClass attributes. The classes must define at least the required attributes: **ou**, **description** and **l**.



Note

The **l** attribute corresponds to the **City** property in OU property editor.

For example, adding the *human-resources* group could produce:

```
ou=human-resources,ou=groups,ou=portal,dc=exoplatform,dc=org
objectclass: top
objectClass: organizationalunit
ou: human-resources
description: The human resources department
l: Human Resources
...
```

- **groupObjectClassFilter**: This filter is used under the *groupsURL* branch to distinguish eXo groups from other entries. You can also use a complex filter if you need.

Example: groups *WebDesign*, *WebDesign/Graphists* and *sales* could be retrieved in:

```
l=Paris,dc=sites,dc=mycompany,dc=com
...
ou=WebDesign,l=Paris,dc=sites,dc=mycompany,dc=com
...
```

```

ou=Graphists,WebDesign,l=Paris,dc=sites,dc=mycompany,dc=com
...
l=London,dc=sites,dc=mycompany,dc=com
...
ou=Sales,l=London,dc=sites,dc=mycompany,dc=com
...

```

8.3.2.3. Membership types

Membership types are the possible roles that can be assigned to users in groups.

```

<field name="membershipTypeURL">
  <string>ou=memberships,ou=portal,dc=exoplatform,dc=org</string>
</field>
<field name="membershipTypeLDAPClasses">
  <string>top,organizationalRole</string>
</field>
<field name="membershipTypeNameAttr">
  <string>cn</string>
</field>

```

- **membershipTypeURL**: base dn for membership types storage.

eXo stores membership types in a flat structure under *membershipTypeURL*. For example, roles, including *manager*, *user*, *admin* and *editor* could be defined by the subtree:

```

ou=roles,ou=portal,dc=exoplatform,dc=org
...
cn=manager,ou=roles,ou=portal,dc=exoplatform,dc=org
...
cn=user,ou=roles,ou=portal,dc=exoplatform,dc=org
...
cn=admin,ou=roles,ou=portal,dc=exoplatform,dc=org
...
cn=editor,ou=roles,ou=portal,dc=exoplatform,dc=org
...

```

- **membershipTypeLDAPClasses**: commas are used to separate list of classes for creating membership types.

When a new membership type is created, an entry will be also created with the given *objectClass* attributes. The classes must define the required attributes: **description**, **cn**.

For example, adding the membership type *validator* would produce:

```

cn=validator,ou=roles,ou=portal,dc=exoplatform,dc=org
objectclass: top
objectClass: organizationalRole
...

```

- **membershipTypeNameAttr**: Attribute will be used as the name of the role.

For example, if *membershipTypeNameAttr* is *cn*, the role name will be *manager* for the following membership type entry:

```

cn=manager,ou=roles,ou=portal,dc=exoplatform,dc=org

```

8.3.2.4. Memberships

Memberships are used to assign a role within a group. They are entries that are placed under the group entry of their scope group. Users in this role are defined as attributes of the membership entry.

- For example, to designate *tom* as the *manager* of the group *human-resources*:

```
ou=human-resources,ou=groups,ou=portal,dc=exoplatform,dc=org
...
cn=manager,ou=human-resources,ou=groups,ou=portal,dc=exoplatform,dc=org
member: uid=tom,ou=users,ou=portal,dc=exoplatform,dc=org
...
```

The parameters to configure memberships are:

```
<field name="membershipLDAPClasses">
  <string>top,groupOfNames</string>
</field>
<field name="membershipTypeMemberValue">
  <string>member</string>
</field>
<field name="membershipTypeRoleNameAttr">
  <string>cn</string>
</field>
<field name="membershipTypeObjectClassFilter">
  <string>objectClass=organizationalRole</string>
</field>
```

- **membershipLDAPClasses**: the commas are used to separate the list of classes for creating memberships.

When a new membership is created, an entry will be also created with the given *objectClass* attributes. The classes must at least define the attribute designated by *membershipTypeMemberValue*. Example: Adding membership *validator* would produce:

```
cn=validator,ou=human-resources,ou=groups,ou=portal,dc=exoplatform,dc=org
objectclass: top
objectClass: groupOfNames
...
```

- **membershipTypeMemberValue**: Multi-valued attribute is used in memberships to reference users that have the role in the group.

Values should be a user dn.

Example: *james* and *root*, who have *admin* role within the group *human-resources*, would give:

```
cn=admin,ou=human-resources,ou=groups,ou=portal,dc=exoplatform,dc=org
member: cn=james,ou=users,ou=portal,dc=exoplatform,dc=org
member: cn=root,ou=users,ou=portal,dc=exoplatform,dc=org
...
```

- **membershipTypeRoleNameAttr**: Attribute of the membership entry whose value refers to the membership type.


```

    <field name="serverName"><string>active.directory</string></field>
  </object>
  [...]
</component>
<component>
  <key>org.exoplatform.services.organization.OrganizationService</key>
  [...]
  <object type="org.exoplatform.services.organization.ldap.LDAPAttributeMapping">
    [...]
    <field name="userAuthenticationAttr"><string>mail</string></field>
    <field name="userUsernameAttr"><string>sAMAccountName</string></field>
    <field name="userPassword"><string>unicodePwd</string></field>
    <field name="userLastNameAttr"><string>sn</string></field>
    <field name="userDisplayNameAttr"><string>displayName</string></field>
    <field name="userMailAttr"><string>mail</string></field>
    [...]
    <field name="membershipTypeLDAPClasses"><string>top,group</string></field>
    <field name="membershipTypeObjectClassFilter"><string>objectClass=group</string></field>
    [...]
    <field name="membershipLDAPClasses"><string>top,group</string></field>
    <field name="membershipObjectClassFilter"><string>objectClass=group</string></field>
  </object>
  [...]
</component>

```



Note

There is a Microsoft limitation: the password cannot be set in AD via unsecured connection, you have to use the LDAPs protocol.

Here is how to use the LDAPs protocol with the Active Directory:

1. Set up AD to use SSL:

- i. Add the Active Directory Certificate Services role.
- ii. Install the right certificate for the DC machine.

2. Enable Java VM to use the certificate from AD:

- i. Import the root CA used in AD, to keystore, such as: `keytool -importcert file 2008.cer -keypass changeit -keystore /home/user/java/jdk1.6/jre/lib/security/cacerts`.

- ii. Set the Java options as below:

```

JAVA_OPTS="${JAVA_OPTS} -Djavax.net.ssl.trustStorePassword=changeit
-Djavax.net.ssl.trustStore=/home/user/java/jdk1.6/jre/lib/security/ca"

```

8.3.4. Picketlink IDM

eXo Platform uses the PicketLink IDM component to keep the necessary identity information, such as users, groups, memberships. While the legacy interfaces are still used (`org.exoplatform.services.organization`) for the identity management, there is a wrapper implementation that delegates to the PicketLink IDM framework. For further information, visit [here](#).

The project `exo.core` defines the API for Organization Service, and the eXo Platform implementation of API. For the Organization Service plugged in the eXo Platform product, you are flexible in switching between: eXo Organization Service, PicketLink and your own implementation. The configuration to switch between various Organization Service implementations can be found in `portal.war/WEB-INF/conf/configuration.xml`:

```

<!--PicketLink IDM integration -->
<import>war:/conf/organization/idm-configuration.xml</import>

<!--Former exo implementations -->
<!--<import>war:/conf/organization/exo/hibernate-configuration.xml</import> -->
<!-- <import>war:/conf/organization/exo/jdbc-configuration.xml</import> -->
<!--for organization service used active directory which is user lookup server -->
<!-- <import>war:/conf/organization/exoactivedirectory-configuration.xml</import> -->
<!--for organization service used ldap server which is user lookup server -->
<!-- <import>war:/conf/ldap-configuration.xml</import> -->

```

If you want to switch between different implementations, you just need to uncomment the corresponding `<import>` and leave others commented:

```

<!--PicketLink IDM integration -->
<import>war:/conf/ldap-configuration.xml</import>
<!-- <import>war:/conf/organization/idm-configuration.xml</import> -->
<!--Former exo implementations -->
<!--<import>war:/conf/organization/exo/hibernate-configuration.xml</import> -->
<!-- <import>war:/conf/organization/exo/jdbc-configuration.xml</import> -->
<!--for organization service used active directory which is user lookup server -->
<!-- <import>war:/conf/organization/exoactivedirectory-configuration.xml</import> -->
<!--for organization service used ldap server which is user lookup server -->

```

8.4. Email

The email service can use any SMTP account configured in `$JBOSS_HOME/server/default/conf/gatein/configuration.properties` or `$TOMCAT_HOME/gatein/conf/configuration.properties` if you are using Tomcat.

The relevant section looks like:

```

# Email
gatein.email.smtp.username=
gatein.email.smtp.password=
gatein.email.smtp.host=smtp.gmail.com
gatein.email.smtp.port=465
gatein.email.smtp.starttls.enable=true
gatein.email.smtp.auth=true
gatein.email.smtp.socketFactory.port=465
gatein.email.smtp.socketFactory.class=javax.net.ssl.SSLSocketFactory

```

It is pre-configured for Gmail, so any Gmail account can easily be used. You simply need to use the full Gmail address as username, and fill in the password.

In corporate environments, you will want to use your corporate SMTP gateway. When using it over SSL, like in the default configuration, you may need to configure a certificate trust-store, containing your SMTP server's public certificate. Depending on the key sizes, you might also need to install Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files for your Java Runtime Environment.

Chapter 9. eXo Platform 3.5 APIs

This chapter presents the information about APIs that help you build your own applications from eXo services via the following topics:

- [Definitions of API Levels](#)
 - [Use Provisional or Experimental APIs](#)
- [Platform APIs](#)
 - [Java APIs](#)
 - [JavaScript APIs](#)
 - [Web Services](#)
- [Provisional APIs](#)
 - [Java APIs](#)

9.1. Definitions of API Levels

APIs vary according to the maturity level. It is important to understand the eXo Platform's general approach to the API change management. The different levels of APIs are described in the following table:

| API Level | Test Suite | Clients | Documentation | Support | Compatibility X.Y.Z(+1) | Compatibility X.Y(+1) |
|------------------|------------|---------|---------------|-------------|-------------------------|-----------------------|
| Platform API | | | | | | |
| Provisional API | | | | | | |
| Experimental API | | | | Best effort | Best effort | |
| Unsupported API | | | | | | |

Test Suite: A suite of tests that can be run against the API to detect changes.

Clients: The API has been used successfully by at least 2 different teams, using the API Documentation only.

Documentation: The API has a clean JavaDoc and reference documentation.

Support: The eXo Support team provides help on the code that uses this API, and fixes any reported bugs.

Compatibility X.Y.Z(+1): The compatibility between maintenance versions (X.Y.Z and X.Y.Z+1) is guaranteed. If there is any change between X.Y and X.Y+1, the eXo Support team will help by upgrading the code.

Compatibility X.Y(+1): The compatibility between minor versions (X.Y and X.Y+1) is guaranteed. If there is any change between X and X+1, the eXo Support team will help by upgrading the code.

Best Effort: You will receive assistance, but eXo Platform cannot guarantee any specific result.

9.1.1. Use Provisional or Experimental APIs

These APIs are provided to give an "early look" at which will be available in upcoming versions of eXo Platform. These APIs are not final, but they can be used to start developing your application.

Provisional APIs are APIs close to being frozen, but that need a last look from users. They can be used by third-party developers for their own apps, with the knowledge that only a limited compatibility guarantee is offered.

Experimental APIs are APIs that are likely to change. They are published to get feedback from the community. These APIs have been tested successfully, but have not yet had enough feedback from developers.

9.2. Platform APIs

9.2.1. Java APIs

Portlet API: (JSR 168 and JSR 286) A Java standard that defines how to write portlets. This is the way to develop Java applications that are integrated into eXo Platform.

WSRP 1.0 on JBoss: A network protocol for integrating remote portlets into eXo Platform.

JAX-RS: (JSR 311) A standard API that provides support for creating REST-like services.

JCR (JSR 170): A standard API that provides access to a content repository.

JCR Service Extensions: A set of APIs that provide extended functionalities for the JCR, such as observation, permissions, and access to a repository.

Java EE 6: eXo Platform supports the Java EE 5 APIs, so you can develop applications using this standard.

Cache: An API used for data caching.

Event and Listener: An API for listening and sending events within eXo Platform.

Organization: An API and SPI for accessing user, group and membership information.

Portal Container Definition: This API is used to configure your portal.

Taxonomy: An API that allows you to organize your content.

Link Management: An API that provides a way to manage links when developing WCM features.

Publication Management: An API that provides different ways to manage the publication of content when developing WCM features.

WCM Composer: An API to get content shown in the website. The cache management is used in this service, and methods to update the content cache.

Template WCM: An API to provide views and dialogs to node types (system or document).

XML Configuration: A set of DTD for configuring eXo Platform.

9.2.2. JavaScript APIs

OpenSocial 0.8 Gadget Specification: A standard that defines how to write gadgets and provide APIs. Gadgets are particularly useful for integrating external applications into eXo Platform.

9.2.3. Web Services

CMIS: A standard API that gives access to the content repository via REST and SOAP Web services.

FTP: A standard protocol for exchanging documents.

OpenSocial 0.8 REST Protocol: A standard API for accessing the social graph and activity streams.

WebDAV: A standard protocol for exchanging document over HTTP.

9.3. Provisional APIs

9.3.1. Java APIs

UI Extensions: An API to plug new actions into the eXo Platform UI.

DocumentExplorer Toolbar

ECM Admin

Activity Sharing

Chapter 10. Cookbook

10.1. How to Copy a Site

This section describes how to copy a work done on the eXo Platform server, such as creating navigations, node types and templates, to another eXo Platform servers. To copy the whole site, you need to do the following actions:

Step 1. Copy the site content folder with its version history by following the substeps:

1. Go to the **Sites Management** drive.
2. Open the site node, for example "acme".
3. Click **Export Node** to export the node with its version history as below:
4. Select **Export** or **Export version history** to perform exporting.
5. Click **Import Node** to open the **Import Node** form.
6. Select the exported nodes and version history to be imported.

One pop-up message will appear to inform that you have imported successfully as below:

Step 2. Copy navigation nodes of sites as follows:

1. Add a new drive to both target and source servers.
2. Export the navigation node.
3. Import the nodes navigation.

Step 3. Copy the template of node type as follows:

1. Add the **System** drive to both servers.
2. Open *system:/jcr:system/exo:namespaces/{namespace_name}*, and export it.
3. Open *system:/jcr:system/exo:namespaces/*, and import the exported file as described in **Step 2**.
4. Open *system:/jcr:system/jcr:nodetypes/{node_type}*, and export it.
5. Open *system:/jcr:system/jcr:nodetypes/*, and import the exported file as described in **Step 4**.



Note

If you have some specific JCR namespaces and node types, you need to import them into the new server.

Step 4. Copy the WCM templates as follows:

1. Open the **DMS Administration** drive.
2. Open *dms-system:/exo:ecm/templates/{node_type}*, and export it.
3. Open *dms-system:/exo:ecm/templates/*, and import the exported file.

Also, for CLV/PCLV templates, you can find all template views defined in the *dms-system:/exo:ecm/views* path with:

- **userviews:** this folder contains views of Sites Explorer with a set of actions.
- **templates:** where you can find all gtmpl templates of:
 - Category Navigation Portlet templates.
 - Content List Viewer (CLV) templates and its paginator templates.
 - content-browser templates (Deprecated Portlet).
 - ecm-explorer templates define how to display nodes in the Sites Explorer portlet, such as CoverFlow template, IconView template.
 - Parameterized Content List Viewer (PCLV) templates and its paginator templates.
 - WCM Advanced Search is used in the WCM Search portlet to define the form, layout, result and result's paginator.

If you want to reuse one of the non-predefined templates above, simply export and import it into the new server at the same place.



Note

If you have some specific WCM (CLV/PCLV) views and/or templates of node types, you will need to import them into the new server.

Step 5. Copy a Taxonomy tree.

By importing the whole site as described in the **Copy the site content folder with its version history** section, you will also have the Taxonomy tree imported. The default location where the site's Taxonomy is placed in a sub-folder named **category**. So, you do not need to export or import them because this step is automatically done. But the Taxonomy tree definition is still not fully imported in the new server. What you now need to do is to add this Taxonomy Tree definition by following those steps:

1. Open the **DMS Administration** drive in the new server.
2. Go to *dms-system:/exo:ecm/exo:taxonomyTrees/definition/*.

3. Add a symlink to the **Taxonomy Tree Root Node**, for example *collaboration:/sites/content/live/acme/categories/acme*.

The name of symlink is displayed as "acme".

The symlink will be generated as below:

In some cases, to see changes, you need to clean cache by disconnecting or restarting the server.

Step 6. Copy metadata templates as follows:

1. Open the **DMS Administration** drive in the new server.
2. Go to */exo:ecm/metadata/{meta_data_name}*.
3. Export and import it in the same location in the new server again.

Step 7. Copy queries via the following substeps:

1. Open the **DMS Administration** drive in the new server.
2. Go to */exo:ecm/queries/{query_name}*.
3. Export and import it in the same location in the new server again.

Step 8. Copy scripts as follows:

1. Open the **DMS Administration** drive in the new server.
2. Go to */exo:ecm/scripts/ecm-explorer*.

You will find three folders referring to the three types of groovy scripts in eXo Platform, including:

- action: The action scripts are launched when an ECM action triggers them. For more information, refer to **Actions Concept**.
- interceptor: Interceptor scripts are triggered before and/or after the JCR node is saved, or when a node is created or edited. They are used to either validate the value entered in a form or to manipulate the newly created node, for example, to map the new node with a forum thread or any other type of discussion areas.
- widget: Widget scripts are used to fill widgets, such as a selectbox in a dynamic way.

3. Export your customized script in the same location in the new server.

Step 9. Copy drive configurations as follows:

1. Open the **DMS Administration** drive in the new server.
2. Go to */exo:ecm/exo:drives/{drive_name}*.
3. Export and import it in the same location in the new server again.

Step 10. Copy gadgets as follows:

1. Open the drive that points into **Portal-System** Workspace.
2. Go to your gadget to *portal-system:/production/app:gadgets/{gadget_namee}*.
3. Export and import it in the same location in the new server again.

Step 11. Restart the server.

After importing the site navigation nodes, the site looks like:

So, you need to restart the server first. The site looks like:

Chapter 11. New Features

This chapter presents new features added to eXo Platform 3.5. At present, in this chapter, you will have opportunity to learn about one feature called **Navigation By Content** with the following specific topics:

- [What is Navigation By Content?](#)
- [Actual content navigation](#)
- [How-To](#)
- [Actions on Navigation By Content](#)
 - [Create a new product](#)
 - [Develop product content](#)

11.1. Navigation by content

11.2. What is Navigation By Content?

Navigation By Content is a feature which allows users to browse content of each page easily. With this feature, users experiencing eXo Platform can navigate from a page to another or browse site content inside one page directly from a contextual menu.

11.3. Actual content navigation

One of the powerful features of Enterprise Content Management System (ECMS) that comes out with eXo Platform 3.5 is the ability to navigate in site contents using taxonomies. This functionality can easily be added in a page with the help of two **Content List Viewer** (CLV) portlets. The pre-configured example can be found in the **News** page of the sample ACME website. In this example, the */site contents/live/acme/events/All* node will be used.

To add "Actual content navigation" to a page:

1. Log in to the sample ACME website.
2. Add a new page, for example "Events".
3. Parameterize this page with the two-column container.
4. Add two content list portlets.

i. First portlet:

In which:

- **Folder path** = */site contents/live/acme/events/All*

- **Header** = Browse by:
- **Template** = CatgoryTree.gtmpl
- **Contextual Folder** = Disabled
- **Show in page** = Events
- **With** = folder-id

ii. Second portlet:

In which:

- **Folder path** = /site contents/live/acme/events/All
- **Template** = OneColumnCLVTemplate.gtmpl
- **Contextual Folder** = Enabled
- **Show in page** = Details
- **With** = content-id

As a result, the created "Events" page will look like:

You can now navigate from the left portlet to see contents displayed in the right portlet.

The new **Navigation By Content** feature will traduce this example in a contextual menu.

11.4. How-To

You need to attach your root folder/node to some page nodes from the homepage (the drop-down menu holds your new contextual menu):

1. Go to the **Content Explorer** page and navigate to */site contents/live/acme/events/All*.

2. Click the **Content Navigation** button.

3. Fill values into the navigation form, including:

- **Visible** = true. This node will be navigable.
- **Target parent navigation** = Events. The contextual menu will be attached to the **Events** drop-down menu.
- **Clickable** = false. This node will not be clickable.
- **Page for list** = catalog. This page is a system page that contains a content list viewer portlet and will be used to display the list of child nodes.

- **Page for detail** = detail. This page is a system page that contains a single content viewer portlet and will be used to display details of child nodes.

4. Save changes, then go back to the **Acme/Overview** homepage.

You will see changes from the **Events** drop-down menu.

In which:

- **Visible:** The */site contents/live/acme/events/All* node is navigable and its child nodes are rendered in the contextual menu.
- **Target parent navigation:** The */site contents/live/acme/events/All* node is attached to the site menu item called **Events**.
- **Clickable:** The */site contents/live/acme/events/All* node is not clickable but all its child nodes are clickable.
- **Page for list:** The list of child nodes (if a child node is **directory/folder**) will be rendered in the following page.

Click the **Earth** menu item from the contextual menu and see that contents of the *Earth* directory are rendered in a separate page (catalog):

- **Page for detail:** The details of child nodes (if a child node is a sample content) will be rendered in this page:

Select the **Power 1 - Fire** menu item from the contextual menu to see the **Fire** content which is displayed in a separate page (details):

11.5. Actions on Navigation By Content

To restrict the visibility of some contents:

1. Go to the **Content Explorer** page and navigate under */site contents/live/acme/events/All/Fire*.
2. Click the **Content Navigation** button.
3. Uncheck the field visible and save.
4. Go back to the homepage and see that the Fire sub-menu is not displayed in the contextual menu.

To sort elements of the contextual menu:

1. Go to the **Content Explorer** page and navigate under */site contents/live/acme/events/All*.
2. Select the */site contents/live/acme/events/All/Earth* node.
3. Click the **Content Navigation** button.
4. Set the **Display order** field to 1 and save.

5. Select the */site contents/live/acme/events/All/Water* node.
6. Click the **Content Navigation** button.
7. Set the **Display order** field to 2 and save.
8. Select the */site contents/live/acme/events/All/Air* node.
9. Click the **Content Navigation** button.
10. Set the **Display order** field to 3 and save.
11. Go back to the homepage and see that the display order from the contextual menu is sorted to Earth, Water, Air. Note that the Fire sub-menu is not displayed because it is set to "invisible" in the previous example.

To restore a node to the contextual menu (if you already removed it) and attach it to another page:

1. Go to the **Content Explorer** page and navigate under */site contents/live/acme/events/All/Fire*.
2. Click the **Content Navigation** button.
3. Fill values into the navigation form fields, including:
 - **Visible** = true
 - **Target parent navigation** = News
 - **Clickable** = false
 - **Page for list** = catalog
 - **Page for detail** = detail
4. Save changes and go back to the **Acme/Overview** homepage and see that the Fire node is attached to the **News** drop-down menu from the site menu:

However, if you want to add your newly created content directly to the contextual menu, you need to add the **populateToMenu** action first.

To add your newly created contents to the contextual menu:

1. Go to the **Content Explorer** page and navigate under */site contents/live/acme/events/All/Fire*.
2. Click the **Manage Actions** button and add the **exo:populateToMenu** action.
3. Create a document under */site contents/live/acme/events/All/Fire* (for example, uploading a file) and publish it.
4. Go back to the homepage and see that your newly created document is added to the contextual menu.

The sample ACME website comes with a configured navigation by content menu:

You can click the **Vision** sub-menu and see contents of vision directory rendered in the **catalog** page:

For example, select the X-Ray content and see the newly implemented content using new visual effects.

- "Benefits" and "Features" tabs:
- Coverflow section:
- Related documents:

11.5.1. Create a new product

To create a new product:

1. Go to the **Content Explorer** page and navigate under *somePath/someDirectory*.
2. Click the **Add Document** button.
3. Select **Product**.
4. Fill the product dialog form.
 - Name
 - Title
 - Illustration Image
 - Summary
 - Benefits
 - Features
5. Save changes.

To improve your newly created product, for example "sampleProduct":

1. Go under *somePath/someDirectory/sampleProduct/medias/images*.
2. Upload some images and publish them.
3. Go under *somePath/someDirectory/sampleProduct/medias/videos*.
4. Upload a video and publish it.
5. Go under *somePath/someDirectory/sampleProduct/documents*.
6. Create two directories, including Sales materials and Technical documentation.
7. Upload a PDF document and publish it under each sub-folder.

8. Add **sampleProduct** to some categories or add it to **Content List Portlet**.

Your newly created product is ready to be displayed in some pages.

9. Publish your newly created product.

Note that you can select this content from a CLV:

As a result, the content will be displayed in a detailed page as follows:

11.5.2. Develop your product content

The product content is composed of fields and folders:

- Fields: Name, Title, Illustration Image, Summary, Benefits, and Features.
- Folders: documents, medias/images, medias/videos.

Folders are created within the product content when the name field is created.

This can be achieved (from the .gtmpl product dialog) as follows:

```
<tr>
    <td class="FieldLabel"><%= _ctx.appRes("Product.dialog.label.name") %></td>
    <td class="FieldComponent">
        <%
            String[] productFieldName = ["jcrPath=/node", "mixintype=mix:votable,mix:commentable", "editable=if-null"];
            uicomponent.addTextField("name", productFieldName) ;
            String[] documentsFolder = ["jcrPath=/node/documents", "nodetype=nt:folder", "mixintype=exo:documentFolder"];
            String[] mediasFolder = ["jcrPath=/node/medias", "nodetype=exo:multimediaFolder", "defaultValues=medias"];
            String[] imagesFolder = ["jcrPath=/node/medias/images", "nodetype=nt:folder", "defaultValues=images"] ;
            String[] videoFolder = ["jcrPath=/node/medias/videos", "nodetype=nt:folder", "defaultValues=videos"] ;
            uicomponent.addHiddenField("documentsFolder", documentsFolder);
            uicomponent.addHiddenField("mediasFolder", mediasFolder);
            uicomponent.addHiddenField("imagesFolder", imagesFolder);
            uicomponent.addHiddenField("videoFolder", videoFolder);
        %>
    </td>
</tr>
```

Other fields are created almost in the same way:

- **Title:**

```
<tr>
    <td class="FieldLabel"><%= _ctx.appRes("Product.dialog.label.title") %></td>
    <td class="FieldComponent">
        <%
            String[] productFieldTitle = ["jcrPath=/node/exo:title", "validate=empty", "editable=if-null"];
            uicomponent.addTextField("title", productFieldTitle) ;
        %>
    </td>
</tr>
```

- **Illustration Image:**

```

<%
private void setUploadFields(name) {
    String[] illustrationHiddenField1 = ["jcrPath=/node/medias/images/illustration", "nodetype=nt:file", "mi
    String[] illustrationHiddenField2 = ["jcrPath=/node/medias/images/illustration/jcr:content", "nodetype=r
    String[] illustrationHiddenField3 = ["jcrPath=/node/medias/images/illustration/jcr:content/jcr:encoding"
    String[] illustrationHiddenField4 = ["jcrPath=/node/medias/images/illustration/jcr:content/jcr:lastModif
    String[] illustrationHiddenField5 = ["jcrPath=/node/medias/images/illustration/jcr:content/dc:date", "vi
    uicomponent.addHiddenField("illustrationHiddenField1", illustrationHiddenField1);
    uicomponent.addHiddenField("illustrationHiddenField2", illustrationHiddenField2);
    uicomponent.addHiddenField("illustrationHiddenField3", illustrationHiddenField3);
    uicomponent.addCalendarField("illustrationHiddenField4", illustrationHiddenField4);
    uicomponent.addCalendarField("illustrationHiddenField5", illustrationHiddenField5);
    String[] fieldImage = ["jcrPath=/node/medias/images/illustration/jcr:content/jcr:data"] ;
    uicomponent.addUploadField(name, fieldImage) ;
}
%>
<tr>

    <td class="FieldLabel"><%= _ctx.appRes("Product.dialog.label.illustrationImage") %></td>
    <td class="FieldComponent">
        <%
            String illustration = "illustration";
            if(ProductNode != null && ProductNode.hasNode("medias/images/illustration") && (uicompon
                def imageNode = ProductNode.getNode("medias/images/illustration") ;
                def resourceNode = imageNode.getNode("jcr:content");
                if(resourceNode.getProperty("jcr:data").getStream().available() > 0) {
                    def imgSrc = uicomponent.getImage(imageNode, "jcr:content");
                    def actionLink = uicomponent.event("RemoveData", "/medias/images/illustr
                        %>
                            <div>
                                <image src="$imgSrc" width="100px" height="80px"/>
                                <a onclick="$actionLink">
                                    <%= _ctx.appRes("Product.dialog.label.summary") %></td>
    <td class="FieldComponent">
        <%
            String[] fieldSummary = ["jcrPath=/node/exo:summary", "options=Basic", "" ] ;
            uicomponent.addRichTextField("summary", fieldSummary) ;
        %>
    </td>
</tr>

```

• Benefits:

```

<tr>

    <td class="FieldLabel"><%= _ctx.appRes("Product.dialog.label.benefits") %></td>
    <td class="FieldComponent">
        <div class="UIFCKEditor">
            <%
                String[] productFieldBenefits = ["jcrPath=/node/exo:productBenefits", "options=toolbar:CompleteW
                uicomponent.addRichTextField("productBenefits", productFieldBenefits) ;
            %>
        </div>
    </td>

```



```
</tr>
```

- **Features:**

```
<tr>
  <td class="FieldLabel"><%= _ctx.appRes("Product.dialog.label.features") %></td>
  <td class="FieldComponent">
    <div class="UIFCKEditor">
      <%
        String[] productFieldFeatures = ["jcrPath=/node/exo:productFeatures", "options=toolbar:CompleteW
        uicomponent.addRichTextField("productFeatures", productFieldFeatures) ;
      %>
    </div>
  </td>
</tr>
```

Now let's look at the product's view form.

Illustration image, title and summary are grouped together:

```
<!-- Hot news -->
<div class="BigNews ClearFix">
  <!-- Begin illustrative image -->
    <%
      RESTImagesRenderService imagesRenderer = uicomponent.getApplicationComponent(RESTImagesRenderService.class);
      def imageURI = imagesRenderer.generateImageURI(currentNode.getNode("medias/images/illustration"));
      if (imageURI != null){
    %>
    <a class="Image"></a>
    <%
      }
    %>
    <div class="Content">
      <!-- Begin title -->
      <%
        if(currentNode.hasProperty("exo:title")) {
          def title = currentNode.getProperty("exo:title").getString();
          %>
          <a href="#" class="Title">$title</a>
          <div class="Index1">$title</div>
          <%
        }
      %>
      <!-- End title -->
      <!-- Begin summary -->
      <%
        if(currentNode.hasProperty("exo:summary")) {
          def summary = currentNode.getProperty("exo:summary").getString();
          %>
          <div class="Summary">$summary</div>
          <%
        }
      %>
      <!-- End summary -->
    </div>
  </div>
</div>
```

- **Benefits and Features fields are rendered in two tabs. It uses the jQuery library (already integrated into eXo Platform 3.5).**

```
<div id="sectionsTabs" class="ui-tabs">
  <ul class="ui-tabs-nav ClearFix">
    <li class="ui-state-default">
      <!-- Begin Benefits head section -->
      <a class="ArrowCtrl" href="#tab-benefits"><%= _ctx.appRes("Product.view.label.benefits") %></a>
    </li>
  </ul>
</div>
```

```

        <!-- End Benefits head section -->
    </li>
    <li class="ui-tabs-selected">
        <!-- Begin Features head section -->
        <a class="ArrowCtrl" href="#tab-features"><%= _ctx.appRes("Product.view.label.features") %>
        <!-- End Features head section -->
    </li>
</ul>
<div id="tab-benefits">
    <%
        if(currentNode.hasProperty("exo:productBenefits")) {
            def benefits = currentNode.getProperty("exo:productBenefits").getString();
            print benefits;
        }
    %>
</div>
<div id="tab-features">
    <%
        if(currentNode.hasProperty("exo:productFeatures")) {
            def features = currentNode.getProperty("exo:productFeatures").getString();
            print features;
        }
    %>
</div>
</div>

<script type="text/javascript">
    jQuery.noConflict();
    jQuery(document).ready(function() {
        jQuery("#sectionsTabs").tabs();
    });
</script>

```

- The jQuery-based feature is display of the product's images (coverflow) from the images folder.

```

<div class="jqProBoxC">
    <!-- Begin jCarouselLite part -->
    <button class="jqprev">&nbsp;</button>
    <div class="jCarouselLite">
        <ul>
            <%
                FOR IMAGE IN PRODUCT'S IMAGE FOLDER
                String imgSrc = "";
                /*
                GET THE IMAGE PATH
                imgSrc = GET THE IMAGE PATH;
                */
            %>
            <li></li>
            <%
            %>
        </ul>
    </div>
    <button class="jqnext">&nbsp;</button>
    <!-- End jCarouselLite part -->
</div>
<script type="text/javascript">
jQuery.noConflict();
    jQuery(document).ready(function(){
        //jQuery.noConflict();
        jQuery(".jCarouselLite").jCarouselLite({
            btnNext: ".jqprev",
            btnPrev: ".jqnext",
            //auto: 500,
            //speed: 500
        });
    });
</script>

```

- Documents and videos are simply displayed within the view form as follows:

1. Get the node path (document or video).
2. Use some customized CSS classes to display a link for this node.

Labels and/or messages displayed in the dialog and the view form are localized.

Note the use of this instruction as below:

```
<td class="FieldLabel"><%= _ctx.appRes( "Product.dialog.label.summary" ) %></td>
[ ... ]
<h1><%= _ctx.appRes( "Product.view.label.seeItInAction" ) %></h1>
```

This is achieved by adding locale files. For example:

```
<Product>
  <view>
    <label>
      <benefits>Benefits</benefits>
      <features>Features</features>
      <seeItInAction>See it in action</seeItInAction>
      <resources>Resources</resources>
      <videos>Videos</videos>
    </label>
  </view>
</Product>
```

Make sure that locale files are added to the resource bundle configuration. If locale files (dialogs and views) are under the *classes/locale/wcm* directory, use the following code:

```
<value>locale.wcm.dialogs</value>
<value>locale.wcm.views</value>
```