eXo Platform 3.0 Developers Guide

eXo Platform ()

Chapter 1. Introduction

1.1. Welcome to eXo Platform

eXo Platform 3.0, the user experience platform for Java, is comprised of Core and Extended Services.

- **Core Services**
  - *GateIn Portal*: a powerful framework for developing portlets and other web-based user interfaces
  - *eXo Content*: extends portal-based applications with Enterprise Content Management (ECM) capabilities
    - *eXo WCM*: web content management services
    - *xCMIS*: an implementation of the full stack of Java-based CMIS (Content Management Interoperability Specification) services on top of eXo WCM
    - *eXo Workflow*: integrated BPM (business process management) capabilities
  - *eXo IDE*: an intuitive web-based development environment that allows developers to build, test and deploy client applications (such as gadgets and mashups) and REST-ful services online
  - *CRaSH*: enables easy browsing of JCR trees, and serves as a shell for executing JCR operations
- **Extended Services**
  - *eXo Social*: a framework for building gadgets that can display and mash-up activity information for contacts, social networks, applications and services
  - *eXo Collaboration*: easily add Mail, Chat, Calendar and Address Book services to portal-based web applications
  - *eXo Knowledge*: adds Forum, Answers and FAQ functionality to portal-based apps, for collecting, organizing and sharing user knowledge

eXo Platform is a fully supported and commercially licensed product based on eXo open source projects. Designed for enterprise use, it has been packaged and tested to optimize production readiness and administration. eXo Platform runs on JBoss, Spring, Tomcat, WebSphere, and other Java applications servers, and can be used with most relational database systems, including MySQL and Oracle.

1.2. Who should read this guide?

This guide describes how to get started with eXo Platform, specifically for

- *System Integrators* : developers who want to know how to leverage eXo Platform in their customer projects.
- *Enterprise IT* : developers that need to customize and deploy a portal in their enterprise.

This guide introduces the eXo architecture, and shows developers how to perform some of the most common tasks needed for working with the Platform. It also serves as an entry point for the Reference Guide, which provides more in-depth technical detail about eXo Platform 3.0.

At the end, you will be able to create your own portal customization with eXo Platform .

Chapter 2. Setting Up Your Project

This guide will help developers use eXo Platform successfully for their projects. It introduces eXo architecture and gives step-by-step instructions for building your own custom portal based on eXo Platform.

> **Note**
>
> While this document covers the most common tasks used when developing on eXo Platform, it is not a complete reference document. The full Reference Guide is available in the delivery package and is linked rather than replicated here.

Requirements:

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

Environment:

Add a system environment variable MAVEN*OPTS (it could be in a .profile startup script on Linux/MacOS operating systems or in 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:

Save the file `settings.xml` in `HOME/.m2/settings.xml` http://wiki.exoplatform.org/xwiki/bin/download/Main/Building%20from%20sources/settings.xml

Edit and change the `local-properties` profile:

- `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 you have an existing file settings.xml, you can merge them all together. At the minimum, you will need the following:

- The local-properties profile, which defines the properties used to build application server distributions of our

products

- The repository http://repository.exoplatform.org/public to download our dependencies

Chapter 3. eXo Architecture Primer

3.1. Architecture overview

3.2. Kernel

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

This chapter will introduce the concepts of Container and Services, and will give you a starting point for the basic configuration of Services.

3.2.1. Containers

A container is always required in order to access a service, because the eXo Kernel relies on dependency injection. This means that the life-cycle of a service (instantiating, opening and closing streams, disposing, etc.) is handled by a dependency provider (i.e. the eXo Container) rather than the consumer. The consumer only needs a reference to an implementation of the requested service, which is provided in the configuration.xml file that comes with every service. You can read more about dependency injection here http://en.wikipedia.org/wiki/Dependency_injection.

eXo has two several kinds of container, the `RootContainer` and the `PortalContainer`. The RootContainer holds the lower level components. It is automatically started before the PortalContainer. You will rarely interact directly with it except to activate your own extension (more on this further). The PortalContainer is created at the startup of the portal web application (portal.war). All services started by this container will run embedded in the portal. It also gives access to the 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 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)
```

3.2.2. Services

Containers are used to gain access to services. Important characteristics of services are:

- 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 = a service. A service doesn't have to be a large application that does big things. A service can be a little component that reads or transforms a document, in which case 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

3.2.3. Service configuration

To declare a service, you must add an xml configuration file in a specific your classpath. A configuration file can specify several services, so there can be several services in one jar file.

3.2.3.1. Kernel XML Schema

Containers configuration files must withe the kernel configuraiton grammar. Thus all configuration will contain an XSD declaration like this :

```
<configuration xmlns="http://www.exoplaform.org/xml/ns/kernel_1_1.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSche
</configuration>
```

The `kernel11.xsd` file mentionned in the example above can be found inside `exo.kernel.container-x.y.z.jar!org/exoplatform/container/configuration/` along with other versions.

3.2.3.2. Components

The registration of a service within the container is done with the `<component>` element. For example, open the `exo-ecms-core-services-x.y.z.jar` file; inside this jar open `/conf/portal/configuration.xml`. 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 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`.

3.2.3.3. Parameters

You can provide initial parameters for your service by defining them in the configuration file. There are different kind of parameters:

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

3.2.3.4. Value-param

You can use the value param to pass values to methods inside the service.

```
<component>
  <key>org.exoplatform.portal.config.UserACL</key>
  <type>org.exoplatform.portal.config.UserACL</type>
  <init-params>
    <value-param>
      <name>access.control.workspace</name>
      <description>groups with memberships that have the right to access the User Control Workspace</description
      <value>*:/platform/administrators,*:/organization/management/executive-board</value>
    </value-param>
  </init-params>
</component>
```

The UserACL service accesses to the value-param in its constructor.

```
package org.exoplatform.portal.config;
public class UserACL {

public UserACL(InitParams params) {
UserACLMetaData md = new UserACLMetaData();
ValueParam accessControlWorkspaceParam = params.getValueParam("access.control.workspace");
if(accessControlWorkspaceParam != null) md.setAccessControlWorkspace(accessControlWorkspaceParam.getValue());
...
```

> **Note**
>
> In this case, the UserACL service has the same <key> and <type>. This corresponds to the special case of a single implementation service. The developer may decide not to create an interface if there will not be more than one implementation of the service.

3.2.3.5. Object-param

For the object-param component, we 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>
```

An object-param is being 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.

Let's see how the service accesses the object:

```
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 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";
  // ...
```

3.2.3.6. More parameter types

Other possible parameter types are object-parameters, Collection, Map and Native Array. See the exhaustive reference in the kernel reference guide.

3.2.4. Plugins

Some components may want to offer some extensibility. For this, they use a plugin mechanism based on method injection. 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 parameter.

Plugins allow you to provide structured configuration from outside the original declaration of the component. This is the main way you will use to customize eXo Platform for your needs.

Let's have a look at the configuration of the `TaxonomyPlugin` of the `TaxonomyService`:

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

The content of `<init-params>` corresponds to the structure of the `TaxonomyPlugin` object.

3.2.5. Configuration Loading Sequence

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

Note

`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:

1. Services default RootContainer configurations from JAR files `/conf/configuration.xml`
2. External RootContainer configuration, will be found at `$exo.conf.dir/configuration.xml`
3. Services default PortalContainer configurations from JAR files `/conf/$PORTAL/configuration.xml`
4. Web applications configurations from WAR files `/WEB-INF/conf/configuration.xml`
5. 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 a path in filesystem. It is passed to the JVM in the startup script like with ~~Dexo.conf.dir=gatein~~
- $PORTAL is the name of the portal container. By default, there is only one and it is called 'portal'

3.3. Gatein Extensions

Gatein extensions are special war files that are recognized by eXo Platform and contribute to custom configurations to the PortalContainer. In order to create your own portal, you will effectively create a Gatein extension.

The extension mechanism makes it possible to extend or even override portal resources in an almost plug-and-play fashion, by simply dropping in a .war archive with the resources and configuring its location in the portal's classpath. Customizing a portal do not involve unpacking and repacking the original portal .war archives. Instead, you create your own .war archive with your own configurations, and eventually modified resources that override the resources in the original archive.

3.3.1. The Default Portal Container

eXo Platform comes with a preconfigured PortalContainer named "portal". This portal container configuration ties together the core and extended services stack. It is started from `portal.war` and naturally maps to the `/portal` uri.

The gatein extensions mechanism lets you very easily extend the /portal context. This capacity is fundamental as it shields you from any changes we may want to do in portal.war. You do not need to fear of upgrades anymore as your extension war will be clearly separated from the portal war.

To achieve this magical extensibility, the PortalContainer turns on two advanced features :

- a unified classloader : any classpath resource, such as xml configuration files, will be accessible as if it was inside the portal.war
- a unified servlet context : any web ressource contained in your extension war will be accessible from /portal/ uri

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

3.3.2. Registering Your Extension

First, you will want eXo to load the WEB-INF/conf/configuration.xml of your extension. For this, declare it as a `PortalConfigOwner` in web.xml :

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

Then you need to register your extension by appending it to the list of *dependencies* that already contribute to the portal container. This is done by an xml configuration liek 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>
```

We define a `PortalContainerDefinitionChange$AddDependencies` plugin to the `PortalContainerConfig`. The plugin declares a list of dependencies that are webapps. The `apply.default=true` indicates that your extension is acutally extending portal.war.

3.4. Java Content Repository

All data in eXo Platform is stored in a Java Content Repository (JCR). JCR is a Java specification (JSR-170) for a type of Document database. It is particularly useful for content management systems, which require storage of objects associated with metadata. A JCR also provides versioning, transactions, observations of changes in data, and import and export of data in XML. The data in a JCR is stored hierarchically in a tree of Nodes with associated Properties.

Also, the JCR is primarily used as an internal storage engine, eXo Content let's you manipulate JCR data directly in several places. This quick intro will tell you how to quickly get your hands on it.

3.4.1. Repositories and workspaces

A content repository consists of one or more workspaces, each of which contains a tree of items.

To access a repository's content at a component level :

```
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);
```

3.4.2. 3.2. Tree structure: working with nodes and attributes

Every node can only have one primary node type. The primary node type defines the 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 (i.e., more child nodes, properties and their respective names and types).

Data is stored in the Properties, which may hold simple values such as numbers and strings or binary data of arbitrary length.

Note

include sample JCR data

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.

Chapter 4. Creating Your Own Portal

When working with eXo, it is important to not modify the source code. This will ensure compatibility with future upgrades, and will simplify support.

To customize your portal, you need to create an extension project by providing your own artifacts as a set of wars/jars/ears.

4.1. 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.

A sample extension package is provided here: http://anonsvn.jboss.org/repos/gatein/portal/trunk/examples/extension/

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

To deploy your extension in Tomcat, follow these steps:

- Add the file `sample-ext.war` from `sample/extension/war/target/` to the `tomcat/webapps` directory.
- Add the folder `starter` from `starter/war/target/` to the `tomcat/webapps` directory.
- Rename the directory (unzipped folder) `starter` 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 alphabetical order.

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

For JBoss deployment and more details, refer to the Reference Guide.

4.2. Portal, pages and menus structure

You can create as many pages as you want 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.

4.2.1. Page layout

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

- **Portal:**

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

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<portal-config
    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_obje
    xmlns="http://www.gatein.org/xml/ns/gatein_objects_1_0">
  <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>

      <page-body> </page-body>

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

- **Pages:**

The `pages.xml` file is used to describe the content of the pages of your portal. In other words, what will be inside the <page-body> tag of the `portal.xml` file above. Here is an example of the classic portal pages.xml.

```
```

```
<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

This section is meant to help you organize the layout and structure of your portal. Review the "Working with applications" chapter to learn more about portlet configuration within the pages.xml file.

- **Navigation:**

The `navigation.xml` is used to associate the links in your navigation (called page-node) with your portal pages.

If the pattern #{} is used then the label of the link will be loaded from the portal resource bundle (link to the ref guide about resource bundles

```
<?xml version="1.0" encoding="UTF-8"?>
<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 can have multiple views inside portliest, such as breadcrumbs that render the current view node, the site map or the menu portlets.

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

4.2.2. Visibility of pages

You can easily set the visibility of select pages and navigation to certain groups and users. Simply 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

4.3. Customize your portal

One of the first steps in any web project is to integrate a graphic chart. This can be done entirely within your extension, by customizing the portal configuration.

In order to add a JavaScript library (for example JQuery), follow these steps:

- Create the following folder, if it does not already exist: `/war/src/main/webapp/WEB-INF/conf/script/groovy`
- Within this folder, create a `JavascriptScript.groovy` file
- Add the following line to the groovy file

```
JavascriptService.addJavascript("Name_Of_Library", "/path_to/java_script/JavaScript_Lib.js", ServletContext);
```

For example:

```
JavascriptService.addJavascript("eXo.myproject.Jquery", "/javascript/eXo/myproject/jquery.js", ServletContext);
```

Chapter 5. Working with Content

5.1. 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 how to use the administration portlet, refer to the Admin Guide, found here:

(!)	need link to admin guide

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

(!)	Comment from Philippe: this part should go in the JCR part in the architecture primer.

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

5.2. WCM templates

The templates are applied to a NodeType or a metadata MixinType. Two kinds of templates exist :

-
  - dialogs : are html forms that allow to create node instances
  - views : are html fragments used to display nodes

From the ECM admin portlet, Manage Template lists existing NodeTypes 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 a specific one is displayed according to the rights of the user (which can be useful in a content validation workflow activity).

5.2.1. DocumentType

The checkbox allows to say if the nodetype should be considered as a DocumentType. File Explorer considers such nodes as user content and applies the following behavior :

-
  - View template will be used to display the DocumentType nodes
  - DocumentTypes nodes can created by the 'Add Document' action
  - non DocumentType are hidden (unless 'Show non document types' option is checked)

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

5.2.2. The 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. The result is a simple but powerful syntax.

5.2.3. Interceptors

By placing interceptors in your template, you will be able to execute a groovy script just before and just after the node-save. Pre node-save interceptors are mostly used to validate input values and their overall meaning while the post node-save interceptor can be used to do some manipulation or reference 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 node-save are passed 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 post node-save one are 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) {}
```

5.2.4. Hidden fields

In the next code sample, each argument is composed of a set of keys and values. The order of the arguments are not important and only the key matters. That example builds a field which has the id "hiddenInput4", which will generate a date which will not be visible. In other words the value of the field will be automatically set to the current date value and no visible field will be printed on the form.

```
String[] hiddenField4 = ["jcrPath=/node/jcrcontent/jcr:lastModified", "visible=false"];
uicomponent.addCalendarField("hiddenInput4", hiddenField4);
```

Once the form is saved, that date value will be saved in under the relative JCR path ./exo:image/jcr:lastModified

5.2.5. Non value field, nodetype or mixintype creation

In many cases, when creating a Node instance out of a form, one must still tell the CMS service about the structure of that node. In other words, the template creator must tell what nodetype is a child of the newly created node or if that current node has any mixin type attributed.

By defining these arguments, the node and its children are created with the correct nodetype and mixintype.

See the following example :

```
String[] hiddenField = ["jcrPath=/node/jcrcontent", "nodetype=nt:resource", "mixintype=exo:rss-enable", "visible
 uicomponent.addHiddenField("hiddenInput", hiddenField) ;
```

5.2.6. 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.

Furthermore, as no widget is specified then a text widget is used. The widget will still not be visible.

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

5.2.7. Non editable and visible if not null fields

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

Furthermore, it is possible to tell that a widget should be visible only if its value is not null, in other words when the form is used to edit an already existing node.

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

5.2.8. WYSIWYG widget

The "What You See Is What You Get" widget is one of the most powerful one. It prints 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 reference them from the created HTML text.

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

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

By default there are two options for the toolbar:

-
  - basic: a minimal set of icons is printed
  - default: a large set of icons is printed, no options argument is needed in that case

There is also a simple textarea widget:

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

5.2.9. Simple selectbox widget

We also provide a selectbox widget with a simple default mechanism where we can print all the available static options, separated with comma, in the "options" argument of the directive. The argument with no key (here "text/html") is the selected option of the list.

```
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 example on how to create WCM templates, take a look at the reference guide.

5.3. Taxonomies

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

When you create a new taxonomy tree, you add a preconfigured `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 by a symlink of type `exo:taxonomyLink` that points to it. The physical storage location is defined by a workspace name, a path and the current date and time. As with adding document types, taxonomy trees can be managed either 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-taxonomie*

As you can see, the value-params allow 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, as well as 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 6. Working with Applications

6.1. Application integration

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

Here is an example of a 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>
```

Here are the details about this configuration:

XML tag name	Description	Example
`application-ref`	The name of the webapp that contains the portlet	
`portlet-ref`	The name of the portlet	
`title`	The title of the page, HTML speaking	
`access-permission`	Who can access the portlet	:platform/users (membership:group)
`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`	Contains a list of `preference`	

XML tag name	Description	Example
	specific to each portlet. Each `preference` has a `name` and a `value`. You can also lock it by setting the `read-only` element to true. CF reference guide.	

6.2. Developing your own applications

6.2.1. Gadget vs Portlet

It is important to understand the distinction 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 relatively quickly, and mashed up on the client-side using lightweight Web Oriented Architecture (WOA) technologies, like REST or RSS.

More information about developing gadgets and portlets can be found in the GateIn Reference Guide:

- Portlet development
- Gadget development

6.2.2. Gadget development quick-start with IDE

eXo Platform facilitates easy gadget development, via its powerful, web-based IDE. You can learn more about the basic principles of Gadget development at http://code.google.com/apis/gadgets/docs/gs.html.

6.2.3. Portlet Bridges

A Portlet Bridge allows you to create a JSR-168 compliant portlet with very little change to 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 7. System Integration

This chapter will help you integrate eXo Platform 3.0 to your information system.

7.1. Authentication

7.1.1. SSO

When logging into the portal, users 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: http://docs.jboss.com/gatein/portal/3.1.0-FINAL/reference-guide/en-US/html_single/index.html#chap-Reference_Guide-SS

7.1.2. Central Authentication Service (CAS)

Central Authentication Service (CAS) is a Web Single Sign-On (WebSSO), developped by JA-SIG as an open-source project. CAS allows users working on different applications to log-in only once be recognized and authentified by all applications. Details about CAS can be found here http://www.ja-sig.org/products/cas/.

The CAS integration consists of two parts; the first part consists of installing or configuring a CAS server, the second part consists of setting up the portal to use the CAS server.

Read the GateIn documentation for more CAS configuration: http://docs.jboss.com/gatein/portal/3.1.0-FINAL/reference-guide/en-US/html_single/index.html#sect-Reference_Guide-Sin

7.1.3. Kerberos SSO on Active Directory

eXo Portal 3.0 supports Kerberos authentication on a Microsoft Active Directory. You will need to configure both your Active Directory server and the application server.

In this example, we suppose that the complete name of the machine on which Tomcat server runs is ubu.exoua-int, and that it runs on the Linux host (Ubuntu 7.04). This machine must be in a Windows domain.

Our implementation makes it possible to use SPNEGO or NTLM. The client will get two authentication headers 'Negotiate' and 'NTLM' and will use whichever is supported by the browser. In Firefox it is possible to manage authentication types, but it isn't in IE, therefore SPNEGO will be used.

Reference guide for configuring Kerberos SSO: NEED LINK

7.2. Users integration

7.2.1. Organization Service

To specify the initial Organization configuration, the content of yourextension.war:/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 type org.exoplatform.services.organization.OrganizationDatabaseInitializer is used to specify a list of membership types, a list of groups, and a list of users to be created.

7.2.2. 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$Group">
        <field name="name">
          <string>portal</string>
        </field>
        <field name="parentId">
          <string></string>
        </field>
        <field name="type">
          <string>hierachy</string>
        </field>
        <field name="description">
          <string>the /portal group</string>
        </field>
      </object>
    </value>
    <value>
      <object type="org.exoplatform.services.organization.OrganizationConfig$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>
        <field name="groups"><string>member:/user</string></field>
      </object>
    </value>
    ...
  </collection>
</field>
```

7.2.3. 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, that provides handling functionality for the five components. For more details, take a look at the GateIn documentation .

7.3. Email

The e-mail 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 preconfigured for GMail, so that any GMail account can easily be used (simply 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 default configuration, you may need to configure a certificate truststore, containing your SMTP server's public certificate. Depending on the key sizes, you may then also need to install Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files for your Java Runtime Environment.