Developer Guide

eXo Platform

The guide you are reading is intended to be a rich introduction to the depth of **eXo Platform**. As a developer, you can use this guide to have a solid framework for achieving your goals on the platform. Throughout this guide, you will meet a high-level tour of eXo Platform and its capabilities via the following chapters:

- **Get Started**

  An introduction to terms commonly used, the process to set up Maven settings, and understanding of eXo Architecture Primer.

- **Create Your Own Portal**

  A procedure that instructs you to create your extension project, define a default portal and its structure, manage languages, create a custom look and feel, add JavaScript, or create custom templates.

- **Work With Content**

  Topics which are related to creating a new content manually in eXo Platform, such as node type, dialog syntax, CKEditor, taxonomy, template service or Navigation By Content.

- **Work With Applications**

  Instructions on how to integrate, then to deploy an application into your portal, and to extend eXo applications.

- **System Integration**

  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**

  Definitions of API levels, and a list of eXo Platform APIs and provisional APIs.

- **Cookbook**

  Steps on how to copy a site to another eXo Platform server.

- **Upgrade eXo Platform**

  All requirements, what needs to be prepared/adapted in eXo Platform extension, and how to update Maven dependencies, configurations, components and extensions.

Table of Contents

Get Started

Before jumping directly into the development tasks, you need to learn about the basic knowledge of eXo Platform 3.5 via the following sections:

- **Glossary**

  Technical terms which are used throughout the documentation.
- **Set up Maven settings**

  Requirements and how to install a Maven project.
- **eXo Architecture Primer**

  Introduction to the eXo architecture, including:
  - **Kernel**

    Concepts of containers, services, plugins and configuration loading sequence.
  - **GateIn extensions**

    The special *.war* files that are recognized by eXo Platform and contribute to custom configurations to the PortalContainer.
  - **Java Content Repository**

    A Java Content Repository (JCR) where all data of eXo Platform are stored.

1.1. Glossary

This section gives you explanations of some technical terms which are used throughout the documentation.

Container templates

Templates which are used to contain the UI components in a specific layout and display them on the portal page.

ConversationState

An object which stores all information about the state of the current user. This object also stores acquired attributes of an Identity which is a set of principals to identify a user.

Data container

An object which 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 (DBCreator)

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

Customized workspaces which include:

- 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

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

An object which 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

One which allows adding configuration for services and components easily.

Folksonomy

A system of classification which is 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. For more details, see: http://en.wikipedia.org.

Gadgets

Web-based software components which are based on HTML, CSS, and JavaScript. They allow developers to easily write useful web applications that work anywhere on the web without modification. For more details, see: opensocial.org.

Groovy template

A template which 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

A service that allows accessing a JCR repository via WebDav.

JobSchedulerService

One which 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)

A tool which converts documents into different office formats and vice versa.

JCR Item

One which may be a node or a property.

ListenerService

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

One which 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

The name of a node or 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

A node which 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

One which 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

One which 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

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

One which 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

A portlet that manages users groups and memberships. This portlet is often managed by administrators to set up permission for users and groups.

OrganizationService

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

One which 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: JSR-170).

Permission

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

A page which 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

Graphic styles which display an attractive user interface. Each skin has its own characteristics with different backgrounds, icons, color, and more.

PortalContainer

A type of container which is created at the startup of the portal web application in the init method of the PortalController servlet.

Portlet

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

One which may be specified by a query on the result nodes by way of property constraints (Source: JSR-170).

Property

An element in the tree structure that makes up a repository. Each property has only one parent node and has no child node.

Repository

One which holds references to one or more workspaces.

eXo REST framework

One which 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

A base container which plays an important role during the startup. However, it is recommended that it should not be used directly.

RTL Framework (Right To Left Framework)

A framework which handles the text orientation depending on the current locale settings. It consists of four components, including Groovy template, Stylesheet, Images, and Client java.

StandaloneContainer

One which is a context independent eXo Container. It is also used for unit tests.

Taxonomy

One which is used to sort documents to ease searches when browsing documents online.

Tree structure

One structure which is defined as a hierarchical structure with a set of linked nodes and properties.

Type constraint

One which 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

A 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

A container of single rooted tree which includes items.

See also

- Set up Maven settings
- eXo Architecture Primer

1.2. Set up Maven settings

Before setting up a complete project/extension, you need to have knowledge of how to install a Maven project.

Requirements

- JDK (Java Development Kit) 6.0
- SVN 1.6+
- Maven 2.2.1+
- Tomcat 6.0.32 or JBoss EAP 5.1.1

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 *HOME/.m2/settings.xml* path.

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.

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.

Note

In Linux environments, the ulimit limits the system-wide resource used. When running eXo Platform, you may get the error message about "Too many open files" because the ulimit had limited the opened files. By default, the number of open files is limited to "1024". You should execute the command "ulimit -n 8196" as root before starting the server to avoid this issue.

See also

- Glossary
- eXo Architecture Primer

1.3. eXo Architecture Primer

- 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.

- GateIn extensions

  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.

- Java Content Repository

  A repository which is tailored to the storage, searching, and retrieval of hierarchical data. All data of eXo Platform are stored in a Java Content Repository (JCR).

See also

- Glossary
- Set up Maven settings

1.3.1. 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.

1.3.1.1. 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 services.

```
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)
```

1.3.1.2. 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 the following databases, caching, and 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

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.

The service configuration is performed in:

- Kernel XML Schema [7]
- Components [7]
- Parameters [8]

Kernel XML Schema

The 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.

Components

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

For example, open the */webapps/exo-wcm-core/WEB-INF/conf/wcm-core/core-services-configuration.xml* file. You will see the following code:

```
<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*.

Parameters

You can provide initial parameters for your service by defining them in the configuration file.

- **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);
  }
}
```

- **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";
  // ...
}
```

- **Rest of parameter types:** Other possible parameter types are Collection, Map and Native Array. See the exhaustive reference in the Kernel reference guide.

1.3.1.3. 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.

1.3.1.4. 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.

Configurations for the *RootContainer* are loaded first, and then for 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

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

1.3.2. GateIn 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. That is, you can inherit resources and configurations from existing web archives, then simply add extra resources and configurations to your extension, and override ones which need to be changed by including modified copies.

Note

Starter is a web application which has been added to create and start all the portal containers once all the other web applications have already been started. Generally, each web application of a portal container defines several things, such as skins, JavaScripts, Google gadgets and configuration files, at its startup, so the loading order is important. For example, at startup of the web application 1, skins or configuration files or JavaScripts are defined that could depend on another JavaScript from the 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 these servlets/servlet filters need to access specific resources of a portal during the process of the servlets or filters request, make sure that these servlets/filters are associated with the current portal container. The proper way to do that is making 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 servlets or servlet filters which looks for resources in associated web applications in the order specified by dependencies configuration.

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

Default Portal Container

eXo Platform comes with a pre-configured PortalContainer named "portal". The configuration of this portal container ties the core and the 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. With this feature, you only need to make your desired modifications on your extension, but NOT on the *portal.war*. As a result, your upgrades will become simple as your *extension.war* is totally independent of the *portal.war*.

This extensibility is achieved via 2 advanced features of the *PortalContainer*:

- 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 ServletContext: any web resources contained in your *extension.war* will be accessible from /portal/ uri.

Tip

For more details on how to make a simple extension for a "portal" container, see the Register Extension [11] section.

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*.

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-->
    <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>
```

1.3.3. 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, **Content** lets you manipulate JCR data directly in several places.

In Java Content Repository, there are 2 main parts:

- **Repositories and workspaces:** A content repository consists of one or more workspaces. Each workspace contains a tree of items.

- **Tree structure - 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 the number of its allowed child nodes. 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 from an existing node. You can refer to the "6.2.3 Node Read Methods" in the JCR Specification document.

Access the repository's content from a service

1. Get the session object.

```
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);
```

i. Obtain the *javax.jcr.Repository* object via one of the following ways:

The first way

Call the *getRepository()* method of *RepositoryService*.

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

The second way

Call the *getCurrentRepository()* method, especially when 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();
```

The third way

Using JNDI as specified in JSR-170.

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

ii. Log in the server to get a Session object by either of two ways:

The first way

Create a Credential object, for example:

```
Credentials credentials = new SimpleCredentials("exo", "exo".toCharArray());

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

The second way

Log in the server without using a Credential object.

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

This way is only applied when you run an implementation of eXo Platform. The eXo Platform 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.

Note

There are some JCR Session common considerations as follows:

Because *javax.jcr.Session* is not a safe object of thread, it is recommended that you should not share it between threads.

Do not use the 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.

2. Retrieve your node content of the session object.

```
String path = "/"; // put your node path here

Node node = (Node) session.getItem(path);
```

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**

  Steps to deploy an extension project.
- **Define a default portal**

  Required steps on how to configure a default portal.
- **Structure of portal, pages and menus**

  Steps on how to define the structure throughout the portal navigation and visibility of pages.
- **Enable/Disable a drive creation**

  Ways to use parameters which allow developers to specify if a drive is automatically created or not.
- **Add/Remove a language**

  Steps on how to define or remove a specific language through the locale configuration file.
- **Create a custom look and feel**

  Elements and various topics on how to create/customize a skin.
- **Add JavaScript to your portal**

  Instructions on how to add a JavaScript library.
- **Create custom templates for pages**

  How to create a custom page template for **Page Creation Wizard**.

2.1. 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.

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.

6. Create a sample-ext.xml file (this file is a Context Descriptor to configure a context within Tomcat server and to ensure that the extension gets loaded before the starter.war). Then add this file to the *tomcat/conf/Catalina/localhost* directory with the following content.

```
<Context crossContext="true" debug="0" docBase="sample-ext" path="/sample-ext" reloadable="true"/>
```

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

See also

- Define a default portal
- Structure of portal, pages and menus
- Enable/Disable a drive creation
- Add/Remove a language
- Create a custom look and feel
- Add JavaScript to your portal
- Create custom templates for pages

2.2. Define a default portal

When entering the link of the portal in the address bar of the browser without defining a specific portal (e.g. *localhost:8080/portal*), you will be directed to a default portal.

To configure the default portal, the portal must already exist first. Then, you just use the *NewPortalConfigListener* plugin as the sample code below.

To use this plugin in the component configuration, you must use the following target-component:

```
<target-component>org.exoplatform.portal.config.UserPortalConfigService</target-component>
```

The sample code below is the configuration of the portal named "*default*".

```
<external-component-plugins>
    <target-component>org.exoplatform.portal.config.UserPortalConfigService</target-component>
    <component-plugin>
      <name>default.portal.config.user.listener</name>
      <set-method>initListener</set-method>
      <type>org.exoplatform.portal.config.NewPortalConfigListener</type>
      <description>this listener init the portal configuration</description>
      <init-params>
        <value-param>
```

```
            <name>default.portal</name>
            <description>The default portal for checking db is empty or not</description>
            <value>default</value>
          </value-param>
          <object-param>
            ....
          </object-param>
        </init-params>
      </component-plugin>
</external-component-plugins>
```

Init-param

Name	Type	Value	Description
default.portal	String	default	The name of the default portal to which you are redirected when accessing the portal.

See also

- Create your extension project
- Structure of portal, pages and menus
- Enable/Disable a drive creation during the portal creation
- Add/Remove a language
- Create a custom look and feel
- Add JavaScript to your portal
- Create custom templates for pages

2.3. 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.

Portal navigation

The portal navigation incorporates the pages that can be accessed even when the user is not logged in assuming the applicable permissions allow the public access). For example, several portal navigations are used when a company owns multiple trademarks, and sets up a website for each of them.

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

- **portal.xml**

The *portal.xml* file describes the layout and portlets that will be shown on all pages. The layout usually contains the banner, footer, menu and breadcrumbs portlets. eXo Platform 3.5 is extremely configurable as every view element (even the banner and footer) is a portlet.

```
<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">
  <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>
    <entry key="showPortletInfo">1</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>
          <application-ref>web</application-ref>
          <portlet-ref>NavigationPortlet</portlet-ref>
        </portlet>
        <access-permissions>Everyone</access-permissions>
        <show-info-bar>false</show-info-bar>
      </portlet-application>

      <portlet-application>
        <portlet>
          <application-ref>web</application-ref>
          <portlet-ref>BreadcumbsPortlet</portlet-ref>
        </portlet>
        <access-permissions>Everyone</access-permissions>
        <show-info-bar>false</show-info-bar>
      </portlet-application>

      <page-body> </page-body>

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

    </portal-layout>

  </portal-config>
```

Each portlet can be configured with a set of preferences, which will be further detailed in the Work With Applications chapter.

It is also possible to apply a nested container that can also contain portlets. Row, column or tab containers are then responsible for the layout of their child portlets.

The <page-body> </page-body> tag is a placeholder for the different pages of your portal, it defines where Platform should render the current page. 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.

The defined "classic" portal is accessible to "Everyone" (at */portal/classic*) but only members of the group */platform/administrators* can edit it.

- **pages.xml**

The structure of the *pages.xml* configuration file is very similar to that in the *portal.xml* of the "classic" portal and it can also contain container tags. Each application can decide whether to render the portlet border, the window state, the icons or portlet's mode.

This is the example of the *pages.xml* file of the "classic" portal.

```
<page-set 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">

  <page>
    <name>homepage</name>
    <title>Home Page</title>
    <access-permissions>Everyone</access-permissions>
    <edit-permission>*:/platform/administrators</edit-permission>
    <portlet-application>
      <portlet>
        <application-ref>web</application-ref>
        <portlet-ref>HomePagePortlet</portlet-ref>
        <preferences>
          <preference>
            <name>template</name>
            <value>system:/templates/groovy/webui/component/UIHomePagePortlet.gtmpl</value>
            <read-only>false</read-only>
          </preference>
        </preferences>
      </portlet>
      <title>Home Page portlet</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>
  </page>

  <page>
    <name>sitemap</name>
    <title>Site Map</title>
    <access-permissions>Everyone</access-permissions>
    <edit-permission>*:/platform/administrators</edit-permission>
    <portlet-application>
      <portlet>
        <application-ref>web</application-ref>
        <portlet-ref>SiteMapPortlet</portlet-ref>
      </portlet>
      <title>SiteMap</title>
      <access-permissions>Everyone</access-permissions>
      <show-info-bar>false</show-info-bar>
    </portlet-application>
  </page>
  .......
</page-set>
```

Note

See the Work With Applications chapter to learn more about the portlet configuration within the *pages.xml* file.

- **navigation.xml**

The *navigation.xml* file defines all the navigation nodes of the portal. The syntax is simply using the nested node tags. Each node refers to a page defined in the *pages.xml* file that will be explained later.

If the administrator wants to create node labels for each language, he will have to use *xml:lang* attribute in the label tag with the value of *xml:lang* is set to the relevant locale.

Otherwise, if the administrator wants the node label is localized by resource bundle files, the *#{...}* syntax will be used. The enclosed property name serves as a key that is automatically passed to the internationalization mechanism. Thus, the emphasis property name is replaced with a localized value taken from the associated properties file matching the current locale.

```
<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>*).

Visibility of pages

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

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

eXo Platform 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 from menu and navigation, 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"
 xsi:schemaLocation="http://www.gatein.org/xml/ns/gatein_objects_1_0 http://www.gatein.org/xml/ns/gatein_objects_1_0">
  <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>
```

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 groups and users 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 (view the page) 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>
```

See also

- Create your extension project
- Define a default portal
- Enable/Disable a drive creation
- Add/Remove a language
- Create a custom look and feel
- Add JavaScript to your portal
- Create custom templates for pages

2.4. Enable/Disable a drive creation

During the portal creation, a drive with the same name as the portal is also automatically created. However, you can decide if such a drive is automatically created or not by using two parameters named *autoCreatedDrive*, and *targetDrives* in the *CreateLivePortalEventListener* external component plugin.

```
<external-component-plugins>
  <target-component>org.exoplatform.services.listener.ListenerService</target-component>
  <component-plugin>
    <name>org.exoplatform.portal.config.DataStorage.portalConfigCreated</name>
    <set-method>addListener</set-method>
    <type>org.exoplatform.services.wcm.portal.listener.CreateLivePortalEventListener</type>
    <description>this listener creates a new live portal content storage.</description>
    <init-params>
      <value-param>
        <name>autoCreatedDrive</name>
        <description>A drive will be automatically created during the portal creation.</description>
        <value>false</value>
      </value-param>
      <values-param>
        <name>targetDrives</name>
        <description>The list of drives which are automatically created during the portal creation with
          "autoCreatedDrive=false".
        </description>
        <value>acme</value>
      </values-param>
    </init-params>
  </component-plugin>
</external-component-plugins>
```

- If *autoCreatedDrive=true*, a drive will be automatically created during the portal creation regardless of *targetDrives*. In case *autoCreatedDrive* is not specified, then its default value is *true*.
- If *autoCreatedDrive=false*, only drives listed in *targetDrives* are created. In case *targetDrives* is not specified, no drives are created.

See also

- Create your extension project
- Define a default portal
- Structure of portal, pages and menus
- Add/Remove a language
- Create a custom look and feel
- Add JavaScript to your portal
- Create custom templates for pages

2.5. Add/Remove a language

Developer can define new or remove a defined language through the locale configuration file. The resource is managed by *org.exoplatform.services.resources.LocaleConfigService* as following:

```
<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>
```

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

Add a new language

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 the 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>
```

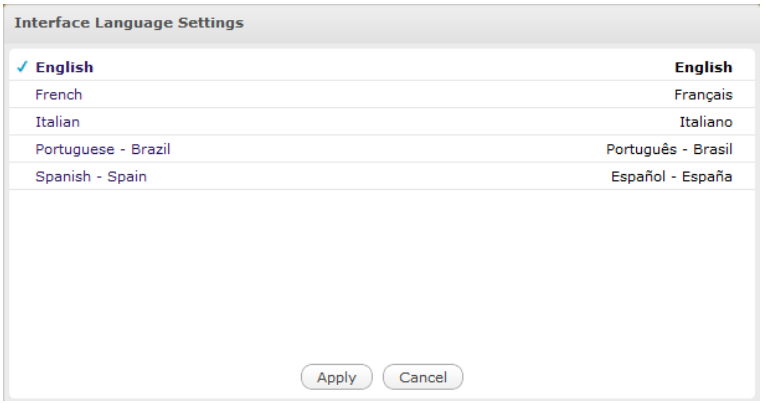
2. Create a new *resource bundle* as **webui_it.properties** in the *myextension.war/WEB-INF/classes/locale/portal* folder.

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, hover the mouse over your username on the **Administration** bar and click **Change Language**. In the **Interface Language Settings** window that appears, you will see the Italian is listed as below:



Remove a language

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.

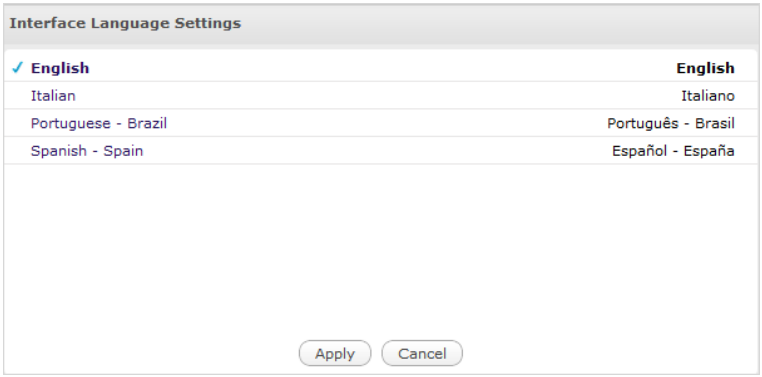
Note

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

3. Restart the server.

To check if French is removed, hover the mouse over your username on the **Administration bar** and click the **Change Language**.

In the **Interface Language Settings** window that appears, French is no longer listed.



See also

- Create your extension project
- Define a default portal
- Structure of portal, pages and menus
- Enable/Disable a drive creation
- Create a custom look and feel
- Add JavaScript to your portal
- Create custom templates for pages

2.6. Create a custom look and feel

- **Platform skin elements**

  The complete skinning of eXo Platform that can be decomposed into 3 main parts: Portal skin, Window style, Portlet skin, and further details of SkinService, ResourceRequestFilter and default skin.

- **Skin the portlet**

  How to define additional stylesheets for each portlet, and to change its icon.

- **Override skins with extension**

  The way to replace a skin definition with the skin resource configured in the extension-deployed web application, and steps to override skins with extension.

- **Create a new skin**

  Steps to create a new skin web archive and preview icon, to skin the window style as well as to configure the right-to-left skin.

- **Configure Platform skin**

  The detailed topics which allow you to configure the eXo Platform skin effectively, such as how to select skins or to customize layouts, styles, templates or shared layouts.

- **Customize Document's skin**

  Instructions on how to create a new document definition and configure it, to create templates, to create and export content into XML files, and finally to set up a deployment for importing the XML files into database.

- **Best practices to customize a skin**

  Conventions and best ways to ease the integration of the design in eXo Platform.

By following the information detailed in this section, you are able to customize a look and feel of your project effectively.

See also

- Create your extension project
- Define a default portal
- Structure of portal, pages and menus
- Enable/Disable a drive creation
- Add/Remove a language
- Add JavaScript to your portal
- Create custom templates for pages

2.6.1. Platform skin elements

Platform provides support for skinning the entire portal User Interface (UI) including your own portlets. Skins are designed to help you pack and reuse common graphic resources.

The complete skinning of eXo Platform can be decomposed into three main parts: Portal skin, Window style, Portlet skin.

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 style

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 skin

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

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.

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 a part of the portal skin and can be included within the portlets web application.

More details of the eXo Platform skin elements are described in:

- SkinService
- ResourceRequestFilter
- Default skin

2.6.1.1. SkinService

Platform skin is processed by the SkinService. It is used to discover and deploy skins into the portal.

- **Configure skins**

eXo Platform 3.5 has a file descriptor for skins (WEB-INF/gatein-resources.xml) which is to specify which portal, portlet and window decorators will be deployed into the skin service. Platform can automatically discover web archives containing this file gatein-resource.xml. The full schema can be found in the lib directory: *exo.portal.component.portal.jar/gatein_resources_1_0.xsd*.

Here is the *gatein-resources.xml* file of a sample skin (called "MySkin") which defines the portal skin with its CSS location, window style and its portlet skins:

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

<!-- define the portlet skin -->

<portlet-skin>
  <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>
   <overwrite>false</overwrite>
</portlet-skin>

  <!-- define the 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>
```

2.6.1.2. ResourceRequestFilter

The ResourceRequestFilter is used to map process skin request mapping from portal, portlet to the exact skin that was defined in portal. ResourceRequestFilter is configured in the *web.xml* file in the skin web archive.

```
<webapp>
<display-name>eXoResources</display-name>
<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>
</webapp>
```

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.

2.6.1.3. Default skin

The default skin of eXo Platform 3.5 is in the *eXoResource.war* file. The main files associated with the skin are:

- *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.

Note

To get a new portal skin declared in *gatein-resources.xml* loaded successfully, *display-name* of the extension webapp (in *web.xml*) should be identical to the context path of the extension webapp.

The following block of CSS illustrates content of the *skin/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 the portal page. The *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.

2.6.2. Skin the portlet

Portlets often require additional styles that may not be defined by the portal skin. eXo Platform 3.5 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.

Change portlet icons

Each portlet can be represented by a 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.

2.6.3. Override skins with extension

The extension mechanism of eXo Platform 3.5 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 Platform's files.

```
<gatein-resources>
  <portal-skin>
    <skin-name>Default</skin-name>
    <css-path>/skin/Defaultskin/Stylesheet.css</css-path>
    <overwrite>false</overwrite>
    <css-priority>0</css-priority>
  </portal-skin>
</gatein-resources>
```

(The css-path specifies the stylesheet of the new skin.)

Override skins with extension

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. 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.

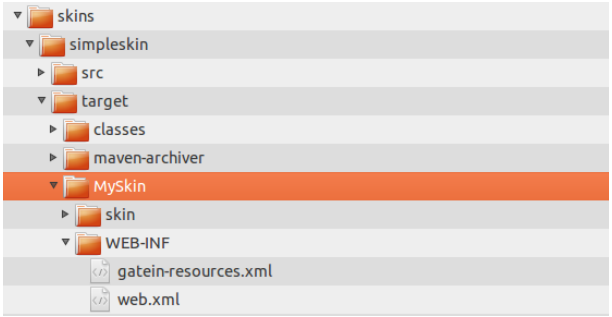
2.6.4. Create a new skin

Creating a new skin is not a simple topic because this task requires many steps to have a wished skin for your product. There are many options for you to create a new skin that depends on your various demands. A typical procedure of creating a new skin includes certain main steps, for example:

- Create a new skin web archive [31]
- Create the skin preview icon [32]
- Skin the window style [33]
- Configure the right-to-left skin [34]

Create a new skin web archive

To create a new skin (called "MySkin"), you should create a new skin web archive with the following structure:



- The *web.xml* is the file that you will define the ResourceRequestFilter.

- The *gatein-resource.xml* will define your new skin (for portal, portlet or window style).
- The **skin** folder will contain images and stylesheets of your skin.

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>
    <css-priority>0</css-priority>
  </portal-skin>
</gatein-resources>
```

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

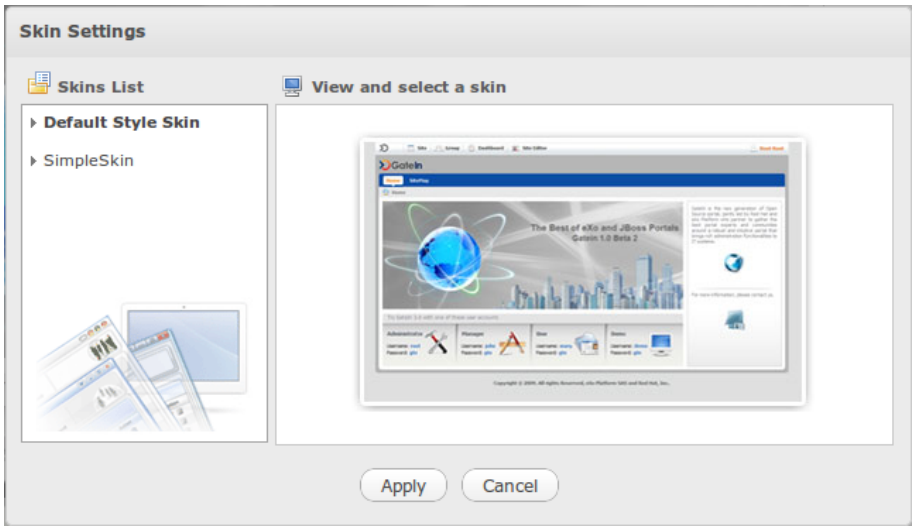
Note

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

eXo Platform provides the "CSS priority" concept which controls the loading order of skins. The skin with lower "css-priority" value will be loaded first.

Create the skin preview icon

When selecting a skin it is possible to see a preview of what the skin will look like. The current skin needs to know about the skin icons for all the available skins, otherwise it will not be able to show the previews. When creating a new portal it is recommended to include the preview icons of the other skins and to update the other skins with your new portal skin preview.



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

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

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*.

Now, amending the deployed package eXoResources is inevitable (modifying the default war/jar breaches development convention of Platform-based products). The problem would be resolved in future eXo Platform 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 ;
}
```

Skin the window style

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.

Configure window styles

Window style is defined within the *gatein-resources.xml* file used by the SkinService to deploy the window style. Window styles can belong to a window style category. This category and window styles need to be specified in the resources file. For example, 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.

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*.

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.

Configure the right-to-left skin

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 */
```

2.6.5. Configure Platform skin

See also

- Platform skin elements
- Skin the portlet
- Override skins with extension
- Create a new skin
- Customize Document's skin
- Best practices to customize a skin

2.6.5.1. 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 Platform as the administrator starts/restarts the server.

Simply add a skin tag to the *portal.war/WEB-INF/conf/portal/portal/classic/portal.xml* configuration file.

To change skin to YourSkin, use the following code:

```
<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>
```

2.6.5.2. Skins in the page markup

The eXo Platform 3.5 skin not only contains CSS styles for the portal's components, but also shares components that may be reused in portlets. When eXo Platform 3.5 generates the page markup of portal, stylesheet links will be inserted 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.

- **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.
- **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

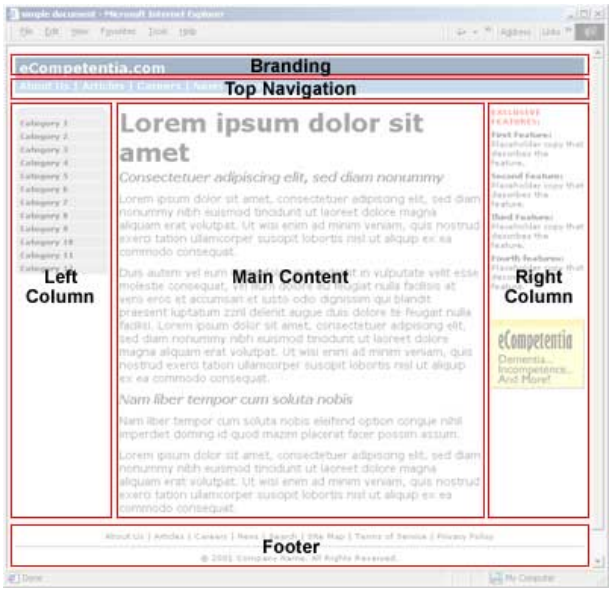
Window styles and portlet specification CSS classes are included within the portal skin.

2.6.5.3. 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: Contain the page body.

And here is the fragment of *portal.xml* located in this path: *<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 -->
</portlet-application>

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

2.6.5.4. 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>

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

2.6.5.5. Customize portal and page's style

Apply your skin into all pages

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.

Apply your skin into your MySite page only

1. Click  **--> Sites Explorer--> 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.

Note

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

The following is the 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.

2.6.5.6. 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>
```

2.6.5.7. Customize CLV template's style

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.

Note

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

- The following is the 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 the database.

2.6.5.8. Configure a shared layout

Existing configurations of sharedlayout.xml

There are currently 4 configurations of *sharedLayout.xml* in eXo Platform:

- *portal/WEB-INF/conf/portal/portal/sharedlayout.xml*
- *webos-ext/WEB-INF/conf/portal/portal/sharedlayout.xml*

- *ecm-wcm-extension/WEB-INF/conf/portal/portal/sharedlayout.xml*
- *platform-extension/WEB-INF/conf/portal/portal/sharedlayout.xml*

The order of *sharedlayout.xml* loading is important. The last loaded extension defines the *sharedlayout.xml* file which the system uses to define portlets displayed in the **Admin** bar.

To make sure your extension is last loaded, look at the configuration file: *jar:/conf/configuration.xml*. The **PortalContainerConfig** component can be used to define the order of war loading.

How it works?

The code which is responsible for *sharedlayout.xml* loading is located in the *POMDataStorage.java* class from portal.

```
public Container getSharedLayout() throws Exception
  {
    String path = "war:/conf/portal/portal/sharedlayout.xml";
    String out = IOUtil.getStreamContentAsString(confManager_.getInputStream(path));
    ByteArrayInputStream is = new ByteArrayInputStream(out.getBytes("UTF-8"));
    IBindingFactory bfact = BindingDirectory.getFactory(Container.class);
    UnmarshallingContext uctx = (UnmarshallingContext)bfact.createUnmarshallingContext();
    uctx.setDocument(is, null, "UTF-8", false);
    Container container = (Container)uctx.unmarshalElement();
    generateStorageName(container);
    return container;
  }
```

You can see the path is hard-coded. Once the good configuration file is chosen by the portal container, this one will use it to add portlets to the **Admin** bar. You can see that all sharedlayout portlets are configured with the *show-info-bar* parameter set to "true".

Override Sharedlayout.xml

To override the *sharedlayout.xml* file, you need to add your configuration file into the *custom-extension.war/WEB-INF/conf/portal/portal/Sharedlayout.xml* file.

Note

Remember to configure your extension loaded to override the other *sharedlayout.xml* configuration.

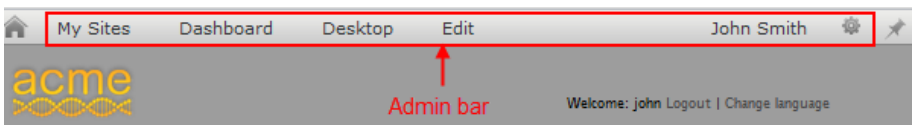
Example of the Sharedlayout.xml configuration: Customize the Admin bar

The **Admin** bar is a special container which is composed of portlets and defined in *WEB-INF/conf/portal/portal/sharedlayout.xml*.

If you want to redefine these portlets, you need to override this file by creating your own *sharedlayout.xml* located into your extension: *custom-extension.war!WEB-INF/conf/portal/portal/sharedlayout.xml*.

Followings are 3 typical examples of the Admin bar configuration: removing a portlet, adding a new portlet and changing the color scheme.

Remove a portlet from the Admin bar



In the illustration above, each circle represents a portlet defined in the **Admin** bar and configured in *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>
 ...
```

Add a portlet to the Admin bar

In the same way you removed a portlet from the Admin bar, you can add your own portlet by editing your *sharedlayout.xml*.

```
...
<container id="UILogoutPortlet" template="system:/groovy/portal/webui/container/UIContainer.gtmpl">
  <access-permissions>*:/platform/users</access-permissions>
  <portlet-application>
    <portlet>
      <application-ref>platformNavigation</application-ref>
      <portlet-ref>UILogoutPortlet</portlet-ref>
    </portlet>
    <access-permissions>*:/platform/users</access-permissions>
    <show-info-bar>false</show-info-bar>
  </portlet-application>
</container>
...
```

The style of portlet container can be changed by editing the CSS file: *eXoResources/skin/DefaultSkin/portal/webui/component/view/UIToolbarContainer/Stylesheet.css*.

```
.UIToolbarContainer .UILogoutPortletTDContainer {
 float: right; /* orientation=lt */
 float: left; /* orientation=rt */
}
```

Change the color scheme

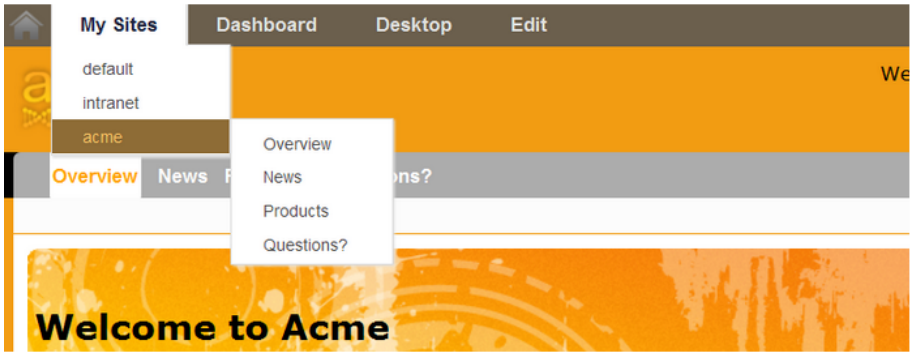
The current color of the **Admin** bar is gray gradient. However, you can change the color to match your brand colors.

The default style of the **Admin** bar.

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

Edit this CSS file to customize the **Admin** bar to your preferred color scheme.

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




```
.UIWorkingWorkspace #UIToolbarContainer .HomeLinkTDContainer {
 line-height: 31px;
   margin: 0px 5px;
   vertical-align: middle;
}

.UIWorkingWorkspace .UIToolbarContainer .ToolbarContainer {
 /*background: url(background/BgToolbarContainer.gif) repeat-x left top;*/
 height: 31px;
 border: none;
}

.UIWorkingWorkspace .UIToolbarContainer .ToolbarContainer .UITab .MenuItemContainer .MenuItem a {
 padding: 0 22px 0 22px;
 font-size: 12px!important;
 color: #4c4c4c;
 display: block;
 font-weight: normal;
 white-space: nowrap;
}

.UIToolbarContainer .ToolbarContainer .PinLink {
   padding: 0px;
}

.UIWorkingWorkspace .UIToolbarContainer .ToolbarContainer a.TBIcon {
 /*color: #2f3334;*/
 font-weight: normal;
 padding: 0 20px;
 display: block;
 white-space: nowrap;
 background: none;
 margin-left: 0;
 zoom: 1;
 font-size:14px!important;
 font-family: verdana;
 border: 1px solid transparent;
 border-bottom: none;
 line-height: 29px;
 height: 29px;
}

.UIWorkingWorkspace .UIToolbarContainer .ToolbarContainer a.SetupMenuItem  {
 padding: 0 8px;
 line-height:25px;
 }
```

2.6.6. 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*

```
code 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.

Note

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 the 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 localtion 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>
       </object-param>
      </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>
          <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>
```

2.6.7. 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.

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.

Folder structure

Portal skins

The portal skin will appear as a single link to a CSS file. This link will contain content 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
```

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* s
- */webapp/skin/PortletThemes/background*
- */webapp/skin/PortletThemes/icons*

2.7. 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.

See also

- Create your extension project
- Define a default portal
- Structure of portal, pages and menus
- Enable/Disable a drive creation
- Add/Remove a language
- Create a custom look and feel
- Create custom templates for pages

2.8. Create custom templates for pages

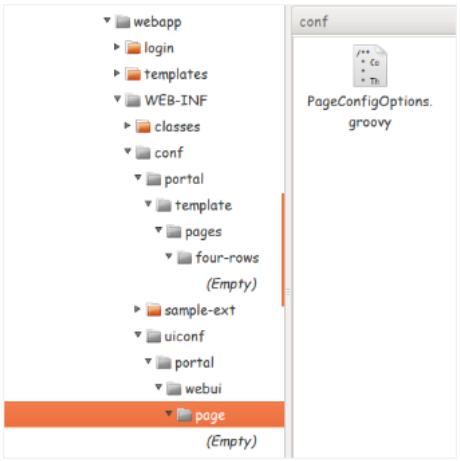
eXo Platform 3.5 provides you with some built-in templates when you create a new page via **Page Creation Wizard**. In this section, you will learn how to create a custom page template for **Page Creation Wizard**.

You should use the extension to add the page layout configuration. In this guide, you are going to work with the */examples/extension* project.

Create a custom page template

1. Add your sample template "FourRowsLayout" by overriding the *PageConfigOptions.groovy* file (portal.war).

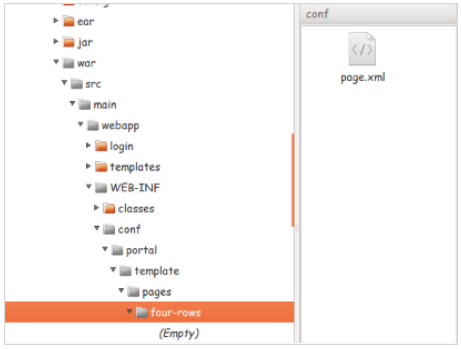
Add it to this path: */examples/extension/war/src/main/webapp/WEB-INF/conf/uiconf/portal/webui/page/PageConfigOptions.groovy*



Sample code:

```
...
SelectItemCategory samplePageConfigs = new SelectItemCategory("samplePageConfigs"):
categories.add(samplePageConfigs):
samplePageConfigs.addSelectItemOption(new SelectItemOption("samplePage.FourRowsLayout","four-rows","FourRowsLayout:"));
...
```

2. Add your template file into the */examples/extension/war/src/main/webapp/WEB-INF/conf/portal/template/pages/four-rows/page.xml* file.



Sample code:

```
<page
xmlns:xsi="http://www.w3.org.2001/XMLSchema-instance"
xsi:schemaLocation="httlp://www.gatein.org/xml/ns/gatein_object_1_2.xsd http://gatein.org/xml/ns/gatein_object_1_2.xsd"
xmlns:"http://gatein.org/xml/ns/gatein_object_1_2.xsd">
<name></name>
<container template="system:/groovy/portal/webui/container/UIContainer.gtmpl">
<access-permission>Everyone<access-permission>
</container>

<container template="system:/groovy/portal/webui/container/UIContainer.gtmpl">
<access-permission>Everyone<access-permission>
</container>

<container template="system:/groovy/portal/webui/container/UIContainer.gtmpl">
<access-permission>Everyone<access-permission>
</container>

<container template="system:/groovy/portal/webui/container/UIContainer.gtmpl">
<access-permission>Everyone<access-permission>
</container>
</page>
```

You can refer to the default template of eXo Platform in this path: */web/portal/src/main/webapp/WEB-INF/conf/portal/template/pages*

3. Add the stylesheet to make the template preview image.

Add your own stylesheet in your extension webapp:

```
.UIItemSelector .<FourRowsLayout>{
 width: 270px; height: 170px;
 margin: auto;
 background: url('background/ItemSelector.gif') no-repeat left -1700px;
}
```

You can refer to the eXo Platform 3.5's default stylesheet in this path: */web/eXoResources/src/main/webapp/skin/DefaultSkin/webui/component/UISelector/UIItemSelector/Stylesheet.css*

4. Deploy your extension project. You will see the template and its preview image in the **Page Creation Wizard**.

See also

- Create your extension project
- Define a default portal
- Structure of portal, pages and menus
- Enable/Disable a drive creation

- Add/Remove a language
- Create a custom look and feel
- Add JavaScript to your portal

Work With Content

This chapter represents issues related to creating a new content manually via the following topics:

- **Node type**

  Ways to define your node type and document type, as well as to select types of Content template.

- **Dialog Syntax**

  Groovy Templates that generate forms by mixing static HTML fragments and Groovy calls to the components responsible for building the UI at runtime.

- **Customize CKEditor**

  A WYSIWYG editor (text editor) which allows you to see what the published results look like while editing your text.

- **Taxonomy**

  A particular classification arranged in a hierarchical structure that helps you organize your content into categories.

- **Template Service**

  A service which allows developers to create dialogs and view templates for each node type register.

- **Navigation By Content**

  A feature which allows developers to browse content of each page easily. With this feature, users experiencing eXo Platform 3.5 can navigate from a page to another or browse site content inside one page directly from a contextual menu.

Note

eXo Platform provides you with 2 options to create the content for your new extension:

- Create new content manually.
- Import an existing content into your extension.

3.1. Node type

To create a content for your extension, you first need to define a node type which represents the document type in the JCR. There are 2 ways to define your node type:

- Via the **Content Administration** portlet. For more details on how to create a note type via the **Content Administration** portlet, see the **Manage node types** in the eXo Platform 3.5 user guide.
- Via the .xml configuration files by creating a *nodetypes-configuration.xml* file in your extension as below.

```
<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" protected="false" requiredType="String">
      <valueConstraints/>
```

```
    </propertyDefinition>
    <propertyDefinition autoCreated="false" mandatory="true" multiple="false" name="date" onParentVersion="COPY" protected="false" requiredType="Date">
      <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).

Document type

The **Document Type** checkbox is to define if the node type should be a **Document Type** or not. If this checkbox is selected, the **Sites 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 Content** 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 Nodes** option is checked.

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

Content template

After defining your node type, you need to select templates which are applied to a node type or a metadata mixin type. eXo Platform provides 2 types of Content templates, including:

- **dialogs** which are HTML forms for creating node instances.
- **views** which are HTML fragments for displaying nodes.

From the **Content Administration** portlet, the **Manage Template** module lists all existing node types that have been associated with **Dialog** and/or **View** templates. These templates can be attached to permissions (in the usual *membership:group* form), so which specific templates are displayed according to user rights (which can be useful in a content validation workflow activity).

Enable JavaScrip

In eXo Platform, you can specify whether JavaScript is allowed to run on a field of the content template or not by using the *"option"* parameter.

1. Go to **Content Administration --> Content Presentation --> Manage Templates**.

2. Edit your desired template.

3. Select the **Dialog** tab, and then edit the content of *Dialog1* in the **View & Edit Template** form.

4. Add *option = noSanitization* to the code in the **Main** field as the example below.

For example:

```
String [] htmlArguments = ["jcrPath = / node / default.html / JCR: content / JCR: data", "options = toolbar: CompleteWCM, height: '410px ', noSanitization" htmlContent];
```

Note

By default, JavaScript is disabled for any fields of some content templates to prevent the XSS attacks.

See also

3.2. 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.

See also

- Node type
- Customize CKEditor
- Taxonomy
- Template Service
- Navigation By Content

3.2.1. 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) {
  }
}
```

3.2.2. 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.

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) ;
```

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"]; uicomponent.addTextField("categories",
 fieldCategories);
```

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=false"] ;
uicomponent.addHiddenField("hiddenInput", hiddenField) ;
```

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) ;
```

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 )
```

WYSIWYG widget

Widgets are natively part of the eXo 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.

WYSIWYG stands for What You See Is What You Get. This 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) ;
```

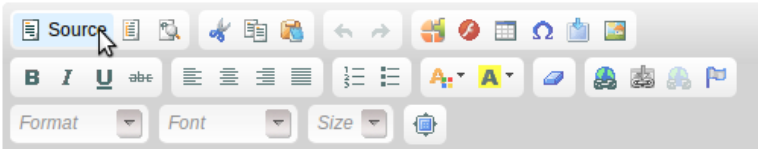
```
String[] fieldContent = ["jcrPath=/node/exo:text", "options=toolbar:CompleteWCM,'height:410px'", ""] ;
uicomponent.addRichtextField("content", fieldContent)
```

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

By default, there are five options for the toolbar: CompleteWCM, Default, BasicWCM, Basic, SuperBasicWCM.

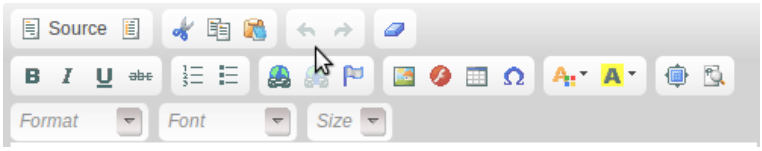
- CompleteWCM: a full set of tools is shown.

The following buttons are shown: Source, Templates, Show Blocks, Cut, Copy, Paste Text, Undo, Redo, SpellCheck, WCM Insert Gadget, Flash, Table, Insert Special Character, WCM Insert Content Link, Bold, Italic, Underline, Strike Through, Justify Left, Justify Center, Justify Right, Justify Full, Ordered List, Unordered List, Text Color, Background Color, Remove Format, Link, WCM Insert Portal Link, Unlink, Anchor, Style, Font Format, Font Name, Font Size, Maximize.



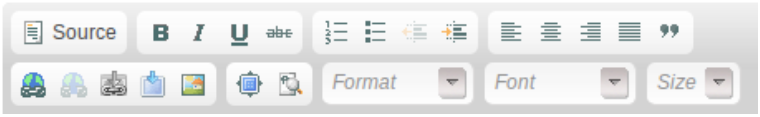
- Default: a large set of tools is shown, no "options" argument is needed in that case.

The following buttons are shown: Source, Templates, Cut, Copy, PasteText, Undo, Redo, SpellCheck, RemoveFormat, Bold, Italic, Underline, Strike Through, Ordered List, Unordered List, Link, Unlink, Anchor, Image, Flash, Table, Special Character, Text Color, Background Color, Show Blocks, Style, Font Format, Font Name, Font Size, Maximize.



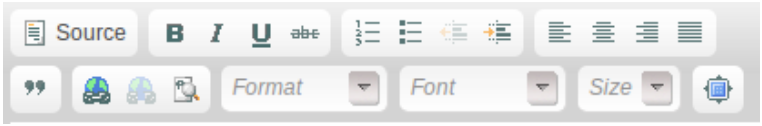
- BasicWCM: a minimal set of tools is shown.

The following buttons are shown: Source, Bold, Italic, Underline, Strike Through, OrderedList, UnorderedList, Outdent, Indent, Justify Left, Justify Center, Justify Right, JustifyFull, Blockquote, Link, Unlink, WCM Insert Portal Link, WCM Insert Content Link, Show Blocks, Style, Font Format, Font Name, FontSize, Maximize.



- Basic:

The following buttons are shown: Source, Bold, Italic, Underline, Strike Through, Ordered List, Unordered List, Outdent, Indent, Justify Left, Justify Center, Justify Right, Justify Full, Blockquote, Link, Unlink, Show Blocks, Style, Font Format, Font Name, Font Size, Maximize.



- SuperBasicWCM:

The following buttons are shown: Source, Bold, Italic, Underline, Justify Left, Justify Center, Justify Right, Justify Full, Link, Unlink, WCM Insert Portal Link, WCM Insert Gadget, WCM Insert Content Link.

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) ;
```

Create a custom RichText editor fields

In the WYSIWYG widget [57] section, you already know about a set of default toolbars (CompleteWCM, Default, BasicWCM, Basic, SuperBasicWCM). In this section, you will learn how to create a **RichText** editor with custom buttons.

Just edit the configuration file and modify or add new items to the configuration file of the **RichText** editor is located in: *apps/resource-static/src/main/Webapp/eXoConfig.js*

Take a look at the eXoConfig.js file to see a definition of a custom toolbar named "MyCustomToolbar":

```
FCKConfig.ToolbarSets["MyCustomToolbar"] = [
  ['Source','Templates','-','FitWindow','ShowBlocks'],
 ['Cut','Copy','PasteText','-','SpellCheck','-','Undo','Redo'],
 ['WCMInsertGadget','Flash','Table','SpecialChar', 'WCMInsertContent'],
 '/',
                    ['Bold','Italic','Underline','StrikeThrough','-','JustifyLeft','JustifyCenter','JustifyRight','JustifyFull','-','OrderedList','UnorderedList','-','TextColor','BGColor','-
','RemoveFormat'],
 ['Link','WCMInsertPortalLink','Unlink','Anchor'],
 '/',
 ['Style','FontFormat','FontName','FontSize']
] ;
```

Every toolbar set is composed of a series of "toolbar bands" that are grouped in the final toolbar layout. The bands items move together on new rows when resizing the editor.

Every toolbar band is defined as a separated JavaScript array of strings. Each string corresponds to an available toolbar item defined in the editor code or in a plugin.

- Put the desired button names in square bracket ("[" & "]") and separate them by commas to create a toolbar band. You can look at the above code to know all the possible toolbar item. If the toolbar item does not exist, a message will be displayed when loading the editor.
- Include a separator in the toolbar band by putting the "-" string on it.
- Separate each toolbar brands with commas.
- Use slash ("/") to tell the editor that you want to force the next bands to be rendered in a new row and not following the previous one.

Note

The last toolbar band must have no comma after it.

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 the WCM Templates section.

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", "scriptParams=production"];
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".

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:UIGroupMemberSelector"];
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) ;
```

```
}
```

Multi-valued widget

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

3.3. Customize CKEditor

- Installation

  Instructions on how to do a fresh installation of CKEditor and steps to upgrade an existing CKEditor installation.
- File and Folder Structure

  Information about the file and folder structure inside CKEditor and introduction to CKEditor in eXo Platform.
- Configuration in CKEditor

  Instructions on how to set configurations, to change the CKEditor skin, to add a new toolbar, and to create a basic plugin for CKEditor.

CKEditor is a WYSIWYG editor (text editor) which allows you to see what the published results look like while editing your text. It brings to the common web-editing features found on desktop-editing applications like Microsoft Word, and OpenOffice. To have more information about a WYSIWYG editor, see the WYSIWYG widget [57] section.

See also

- Node type
- Dialog Syntax
- Taxonomy
- Template Service
- Navigation By Content

3.3.1. Installation

You can install CKEditor easily by selecting an appropriate procedure (fresh installation or upgrade) by doing the following steps:

Fresh Installation

Note

This way is used to install CKEditor for the first time.

1. Download the latest version of the editor here.

2. Extract (decompress) the downloaded archive to a directory called *ckeditor* in the root of your website.

Note

You can place the files in any path of your website. The *ckeditor* directory is the default one.

Upgrade an existing CKEditor installation

1. Rename your old editor folder to a backup folder, for example, "ckeditor_old".

2. Download the latest version of the editor here.

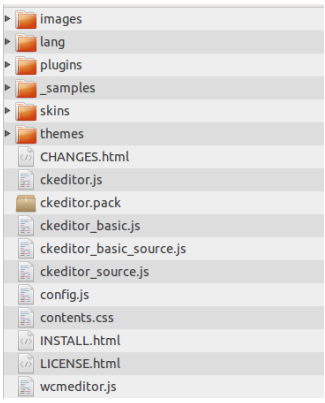
3. Extract/Decompress the downloaded archive to the original editor directory, for example, "ckeditor".

4. Copy all configuration files that you have changed from the backup folder to their corresponding positions in the new directory. These files could include (but not limited to) the following files:

- config.js
- contents.css
- plugins/templates/templates/default.js
- plugins/styles/styles/default.js
- plugins/pastefromword/filter/default.js

3.3.2. File and Folder Structure

Inside CKEditor



There are the following folders:

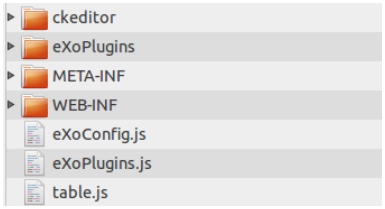
Folders	Description
samples	Contain CKEditor samples.
source	Contain CKEditor source code.
adapters	Contain CKEditor adapters. It may be removed if you do not use any adapters, like the jQuery one.
images	Contain CKEditor graphics files.
lang	Contain CKEditor language files.
plugins	Contain plugin files and is necessary for CKEditor to work.
skins	Contain CKEditor skin files along with toolbar buttons and stylesheet definitions.
themes	Contains CKEditor theme.

Files	Description
ckeditor.js	The heart of CKEditor application. It is the unique compressed file which contains all of codes to run CKEditor.
ckeditorbasic.js	The compressed file like *ckeditor.js*, but it is only a bootstrap which contains the core functionality only, so the rest of the code can be loaded at later time, avoiding delaying the initial load of the page.
ckeditorsource.js	An uncompressed version of *ckeditor.js*.
ckeditorbasicsource.js	An uncompressed version of *ckeditorbasic.js*.

Files	Description
config.js	Allow users to customize some configurations.
contents.css	Define the stylesheets of the CKEditor application.
ckeditor.asp	Used for ASP integration.
ckeditor.php	Used for PHP integration.
ckeditorphp4.php	Used for PHP4 integration.
ckeditorphp5.php	Used for PHP5 integration.
ckeditor.pack	Re-build the compression version of 2 files: *ckeditor.js*, and *ckeditorbasic.js*.

CKEditor in Context of eXo Platform

There is a *.war* package named *eXoStaticResources* which integrates the CKEditor application into eXo Platform. The source code is placed inside the *apps/resources-static/src/main/webapp* folder.



Its structure consists of the folders and files below:

Folders & Files	Description
ckeditor	Contain all source codes of CKEditor v3.3.2.
eXoPlugins	Contain the source code of 3 external plugins: {}Insert Content Link{}, {}Insert Portal Link{}, {}WCM Insert Gadget{}.
eXoConfig.js	Register 3 external plugins, and define some types of the toolbar.
eXoPlugins.js	Define some utility functions which can be used by 3 external plugins.

3.3.3. Configuration in CKEditor

Set configurations

CKEditor comes with a rich set of configuration options that make it possible to customize its appearance, features, and behavior. The main configuration file is named *config.js*. This file can be found in the root of the CKEditor installation folder (*/webapps/eXoStaticResources/ckeditor/config.js*). By default, this file is mostly empty. To change the CKEditor configuration, add the settings that you want to modify to the *config.js* file.

For example:

```
CKEDITOR.editorConfig = function( config )
{
    config.language = 'en';
    config.uiColor = '#AADC6E';
};
```

Instead of using the default *config.js* file, you can create a copy of that file in anywhere in your website and simply point the editor instances to load it. For example, in eXo Platform, the configuration file for CKEditor is placed at */webapps/eXoStaticResources/eXoConfig.js*, so the content of the *config.js* file will be:

```
CKEDITOR.editorConfig = function( config )
{
    config.customConfig = "../eXoConfig.js";
};
```

Change the CKEditor skin

You can change the CKEditor skin by adjusting a single configuration option. In eXo Platform, to change the CKEditor skin, do as follows:

1. Open the */webapps/eXoStaticResources/eXoConfig.js* configuration file of CKEditor.

2. Set up a skin for CKEditor. It may be the name of the skin folder inside the editor installation path, or the name and the path separated by a comma.

```
config.skin = 'v2';
config.skin = 'myskin,/customstuff/myskin/';
```

By default, CKEditor has 3 skins for users to select: **v2**, **kama**, and **office2003**. They are placed in the */webapps/eXoStaticResources/ckeditor/skins* folder.

Add a new toolbar

CKEditor is a full-featured WYSIWYG editor, but not all of its options are needed in all cases. Therefore, the toolbar customization is one of the most common and required tasks when dealing with CKEditor.

- **Toolbar Definition** is a JavaScript array which contains the elements to be displayed in all toolbar rows available in the editor. In eXo Platform, the toolbar definition is placed in the */webapps/eXoStaticResources/eXoConfig.js* file. The following code snippet contains the default CKEditor toolbar set in eXo Platform.

```
config.toolbar_Default = [
    ['Source','Templates'],
    ['Cut','Copy','PasteText','-','SpellCheck'],
    ['Undo','Redo','-','RemoveFormat'],
   '/',
    ['Bold','Italic','Underline','Strike'],
    ['NumberedList','BulletedList'],
    ['Link','Unlink','Anchor'],
    ['Image','Flash','Table','SpecialChar'],
    ['TextColor','BGColor'],
    ['Maximize', 'ShowBlocks'],
    ['Style','Format','Font','FontSize']
] ;
```

- To add a new toolbar in eXo Platform, open the */webapps/eXoStaticResources/eXoConfig.js* configuration file of CKEditor, then add the following code snippet to it.

```
config.toolbar_MyToolbar =
[
 ['Bold', 'Italic', '-', 'NumberedList', 'BulletedList', '-', 'Link', 'Unlink','-','About']
];
```

- To show the newly added toolbar, you have to add it to a field of a WCM template. For example, to show the new toolbar on the content field of HTML file, you need to modify the dialog template of HTML file as follows:

```
String[] fieldSummary = ["jcrPath=/node/jcr:content/jcr:data", "", "options=toolbar:MyToolbar, noSanitization"] ;
uicomponent.addRichtextField("contentHtml", fieldSummary) ;
```

By adding a new HTML file, you will see the new toolbar (MyToolbar) on the content field:

Create a basic plugin for CKEditor

Assuming that you will develop a *timestamp* plugin that inserts the current date and time into the editing area of CKEditor. The *timestamp* will be added after a user clicks a dedicated toolbar button. The implementation makes use of the *insertHtml* function which can be also easily adjusted to insert any other HTML elements into CKEditor.

1. Create a directory inside the *eXoPlugins* directory for CKEditor with the *timestamp* plugin.

2. Place the *plugin.js* file that contains the plugin logic inside the newly created *timestamp* folder. Also, you will need a toolbar icon for the plugin by adding an *images* folder and subsequently placing the *timestamp.png* file inside it.

3. Modify the *plugin.js* file in which you will write the behavior.

The following is the code used to create a simple plugin named *timestamp*:

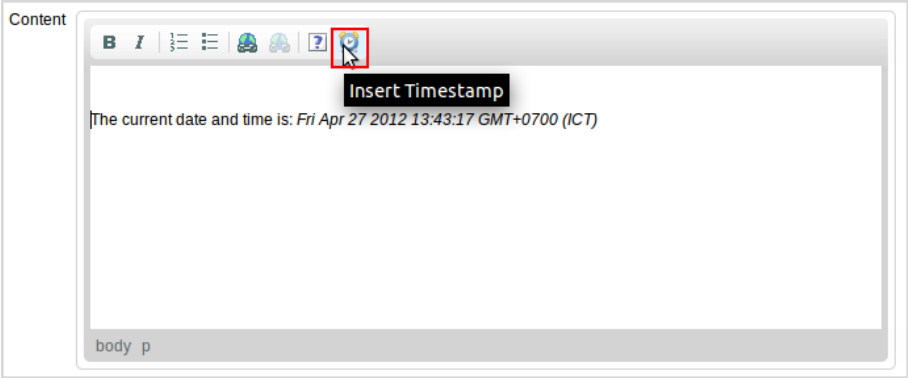
```
CKEDITOR.plugins.add( 'timestamp',
{
 init: function( editor )
 {
  editor.addCommand( 'insertTimestamp',
   {
    exec : function( editor )
    {
     var timestamp = new Date();
     editor.insertHtml( 'The current date and time is: <em>' + timestamp.toString() + '</em>' );
    }
   });
  editor.ui.addButton( 'Timestamp',
  {
   label: 'Insert Timestamp',
   command: 'insertTimestamp',
   icon: this.path + 'images/timestamp.png'
  } );
 }
} );
```

To use the created plugin, plug it to CKEditor by using the following code:

```
(function() {CKEDITOR.plugins.addExternal('timestamp',CKEDITOR.eXoPath+'eXoPlugins/timestamp/','plugin.js');})();
  ....
```

```
config.extraPlugins = 'content,insertGadget,insertPortalLink,timestamp';
....
config.toolbar_MyToolbar =
[
        ['Bold', 'Italic', '-', 'NumberedList', 'BulletedList', '-', 'Link', 'Unlink','-','About','Timestamp']
];
```

The following is the illustration about the Timestamp plugin added to the CKEditor.



3.4. Taxonomy

Taxonomy is a particular classification arranged in a hierarchical structure. The 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 **Content Administration** portlet, or by adding .xml configuration files.

Configure a taxonomy tree by adding the 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 in *$PLF-HOME_/samples/acme-website/webapp/src/main/webapp/WEB-INF/conf/acme-portal/wcm/taxonomy/acme-taxonomies-configuration.xml*.
  - 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.

See also

- Node type
- Dialog Syntax
- Customize CKEditor
- Template Service
- Navigation By Content

3.5. Template Service

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

You can find the template service configuration in */webapps/ecm-wcm-core/WEB-INF/conf/wcm-core/core-services-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 location 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>
</object-param>
</init-params>
```

See also

- Node type
- Dialog Syntax
- Customize CKEditor
- Taxonomy
- Navigation By Content

3.6. Navigation By Content

- Actual content navigation

  Steps to add "actual content navigation" to a page, including the way to configure the right and left portlets.
- Add content to the navigation

  The detailed procedure to attach your root folder/node to some page nodes.
- Actions on Navigation By Content

  Instructions on how to restrict the visibility of some content, to sort elements of the contextual menu, to restore a node to the contextual menu and attach it to another page, and to add your newly created content to the contextual menu.
- Create data for Navigation By Content

  TWO typical examples of creating data for **Navigation By Content**, including how to create a Product page, and to develop your own Product content.
- Create a new Content List template

  Knowledge and typical steps to create a new template that is used in the **Content List** portlet.

Navigation By Content is a feature which allows users to browse content of each page easily. With this feature, users experiencing eXo Platform 3.5 can navigate from a page to another or browse site content inside one page directly from a contextual menu.

See also

- Node type
- Dialog Syntax
- Customize CKEditor
- Taxonomy
- Template Service

3.6.1. 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 content 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, all content in the */Sites Management/acme/events/All* node will be used.

Add "Actual content navigation" to a page

1. Log into the sample ACME website.

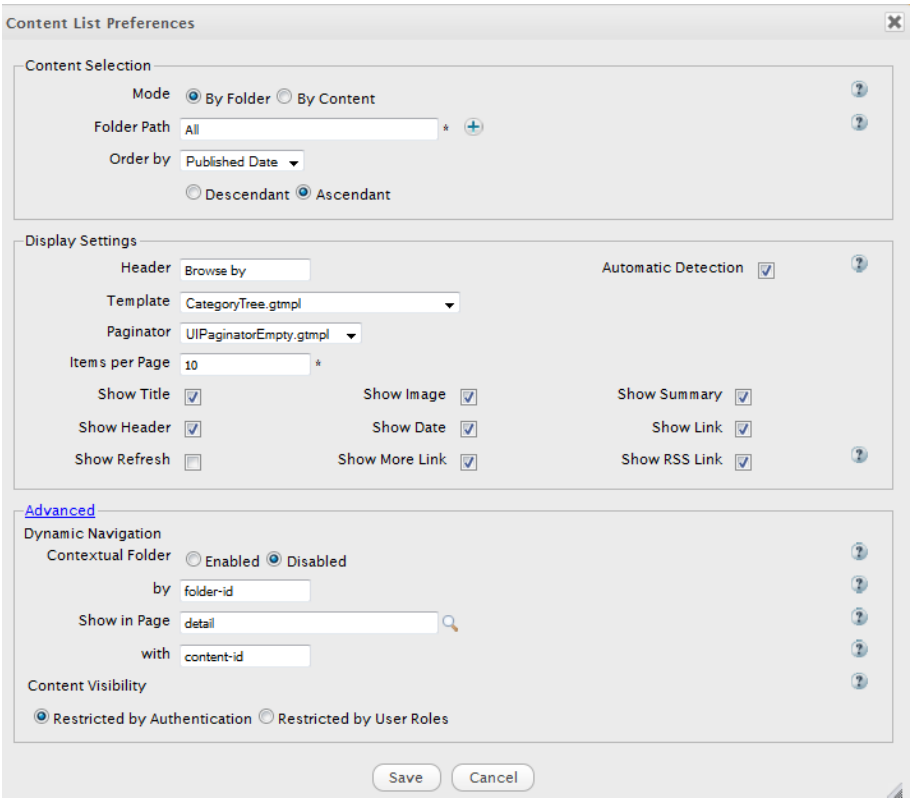
2. Add a new page, for example "Events".

3. Parameterize this page with the **Autofit Two Columns** container.

4. Add two **Content List** portlets to each column.

5. Add content.

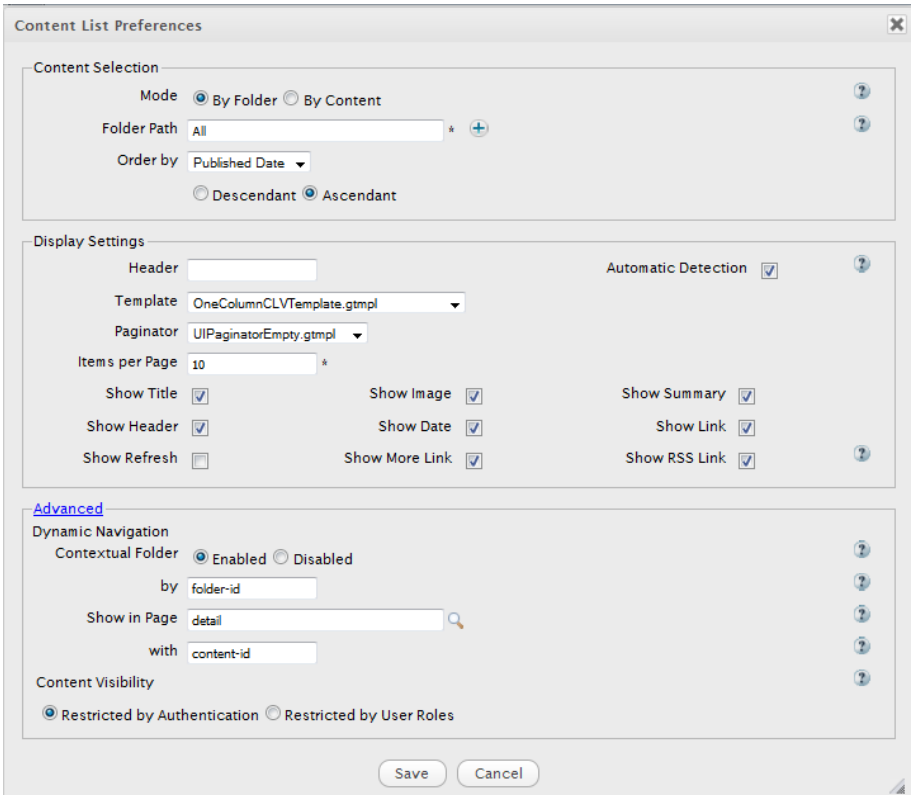
i. Configure the left portlet as follows:



In which:

- **Folder path** = */Sites Management/acme/events/All*: The path to the folder that contains the content.
- **Header** = *Browse by*: The title of all content that is listed in the content list viewer.
- **Template** = *CategoryTree.gtmpl*: The template used for displaying the content list.
- **Contextual Folder** = *Disabled*: The Contextual Content property is set to "Disable", the Advanced pane is closed by default and a single content will be opened by an URL containing the content path.
- **Show in page** =*Events*: A single content in CLV will be shown in the **Events** page.
- **With** = *folder-id*: The parameter containing the content path.

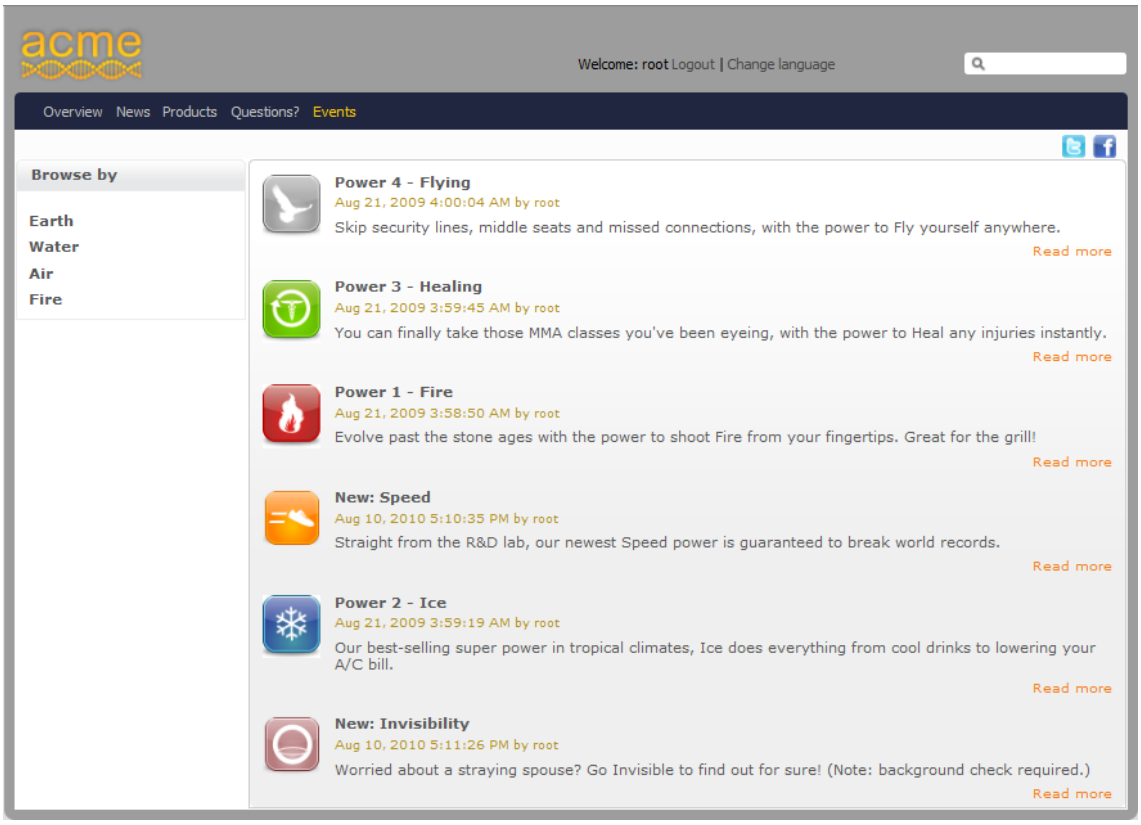
ii. Configure the right portlet as follows:



In which:

- **Folder path** = */Sites Management/acme/events/All*: The path to the folder that contains the content.
- **Template** = *OneColumnCLVTemplate.gtmpl*: The template used for displaying the content list.
- **Contextual Folder** = *Enabled*: The Contextual Content property is set to "Enable". This portlet is configured with the provided parameter (content-id by default).
- **Show in page** = *Details*: A single content in CLV will be shown in the "Details" page.
- **With** = *content-id*: The parameter containing the content path.

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



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

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

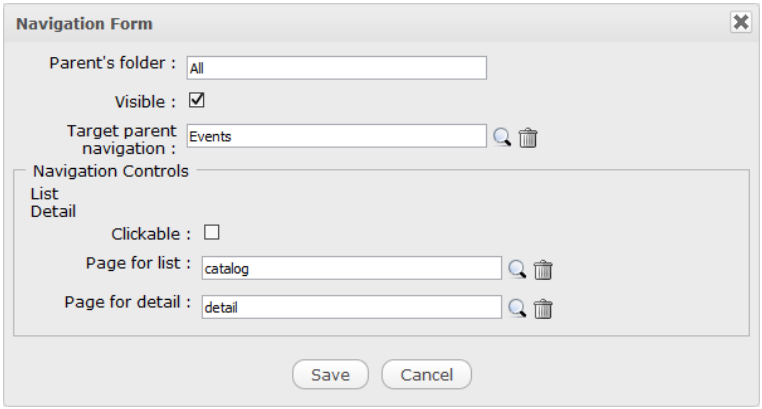
3.6.2. Add content to the navigation

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 **Sites Explorer** page and navigate to */Sites Management/acme/events/All*.

2. Click the **Content Navigation** button, the **Navigation** form will appear. If you do not see this button on the **Action** bar, add this button via the **Content Administration** page.

3. Fill values into the **Content Navigation** form:

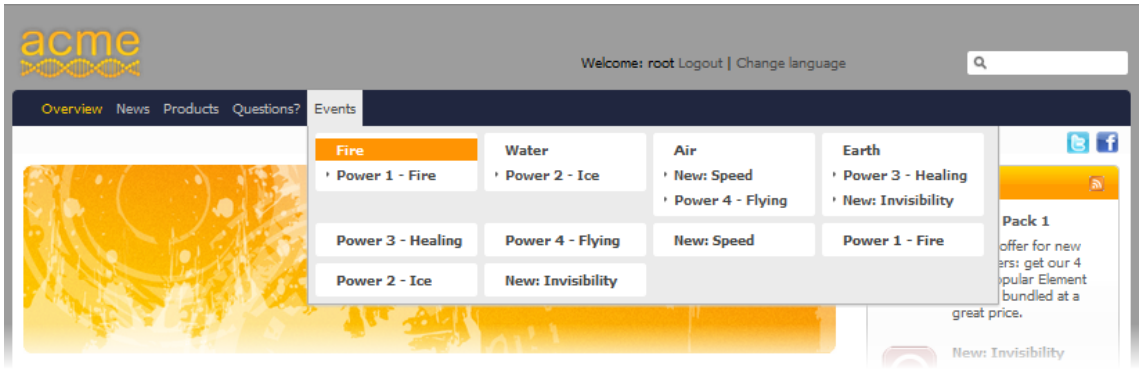


In which:

- **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.

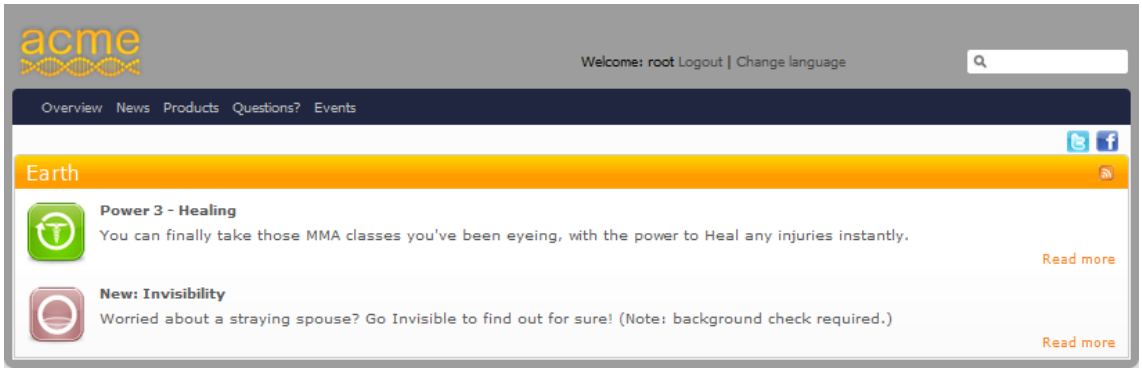
4. Save changes, then go back to the ACME homepage. You will see changes from the **Events** drop-down menu.



In which:

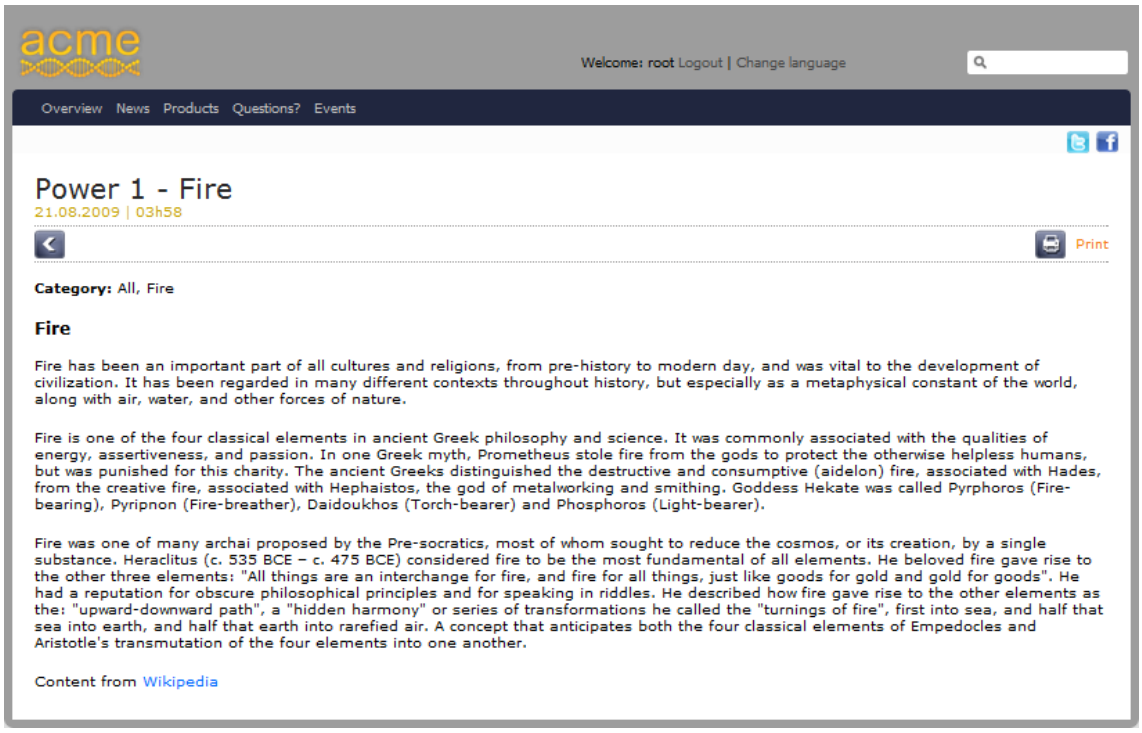
- **Visible**: The */Sites Management/acme/events/All* node is navigable and its child nodes are rendered in the contextual menu.
- **Target parent navigation**: The */Sites Management/acme/events/All* node is attached to the site menu item called **Events**.
- **Clickable**: The */Sites Management/acme/events/All* node is not clickable but all of 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, you will see that content 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 displayed in a separate page (details):



3.6.3. Actions on Navigation By Content

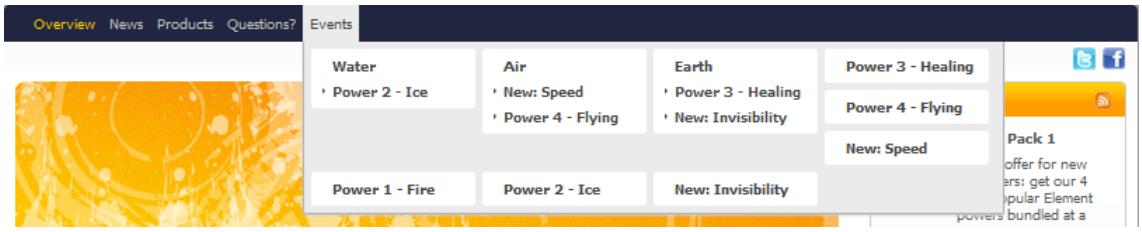
Restrict the visibility of some content

1. Go to the **Sites Explorer** page and navigate to the */Sites Management/acme/events/All/Fire* node.

2. Click the **Content Navigation** button to open the **Navigation** form.

3. Uncheck the **Visible** field and save.

4. Go back to the ACME homepage. You will see that the **Fire** sub-menu is not displayed in the contextual menu anymore.



Sort elements of the contextual menu

1. Go to the **Sites Explorer** page and navigate to the */Sites Management/acme/events/All* node.

2. Select the */Sites Management/acme/events/All/Earth* node.

3. Click the **Content Navigation** button to open the **Navigation** form.

4. Set the **Display order** field to "1" and save.

5. Select the */Sites Management/acme/events/All/Water* node.

6. Click the **Content Navigation** button.

7. Set the **Display order** field to "2" and save.

8. Select the */Sites Management/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 ACME homepage. You will see that the display order from the contextual menu is **Earth**, **Water**, **Air**. Note that the **Fire** sub-menu is not displayed because it is set to "Invisible" in the previous example.

Restore a node to the contextual menu and attach it to another page

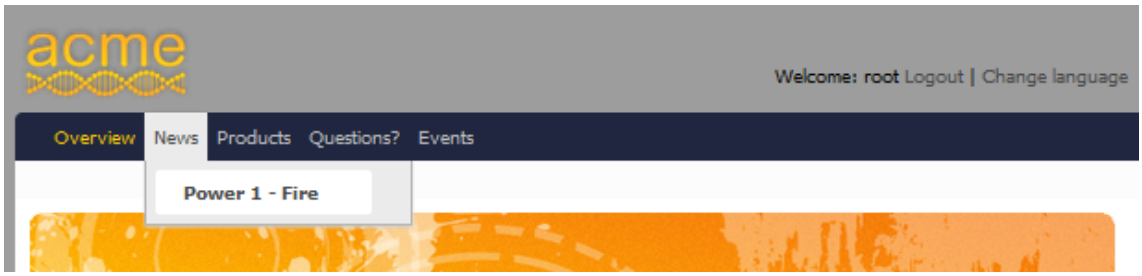
1. Go to the **Sites Explorer** page and navigate to the */Sites Management/acme/events/All/Fire* node.

2. Click the **Content Navigation** button to open the **Navigation** form.

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. You will see that the **Fire** node is attached to the **News** drop-down menu from the site menu:

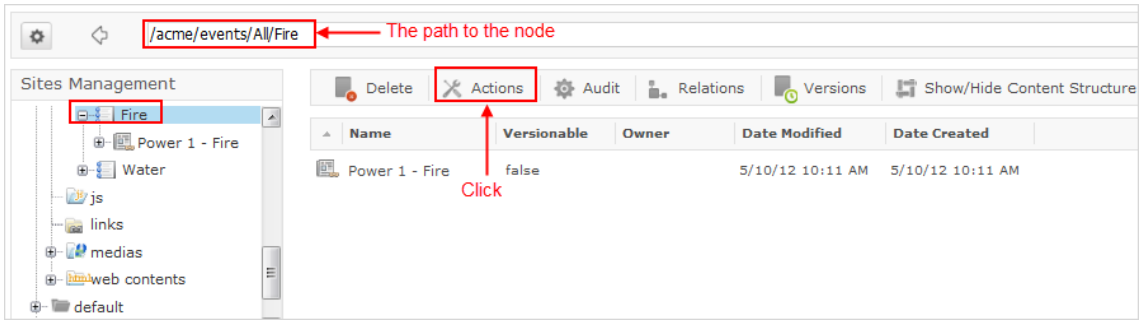


However, if you want to add your newly created content directly to the contextual menu, you need to add the **populateToMenu** action first.

Add your newly created content to the contextual menu

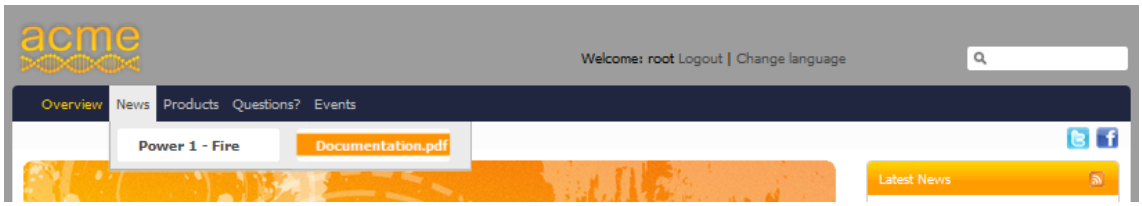
1. Go to the **Sites Explorer** page and navigate to the */Sites Management/acme/events/All/Fire* node.

2. Click the **Actions** button and add the **exo:populateToMenu** action.

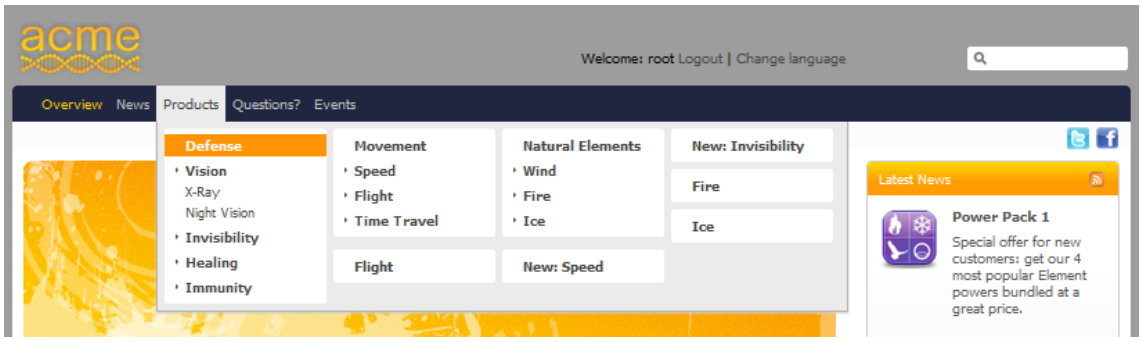


3. Create a document under the */Sites Management/acme/events/All/Fire* node and publish it.

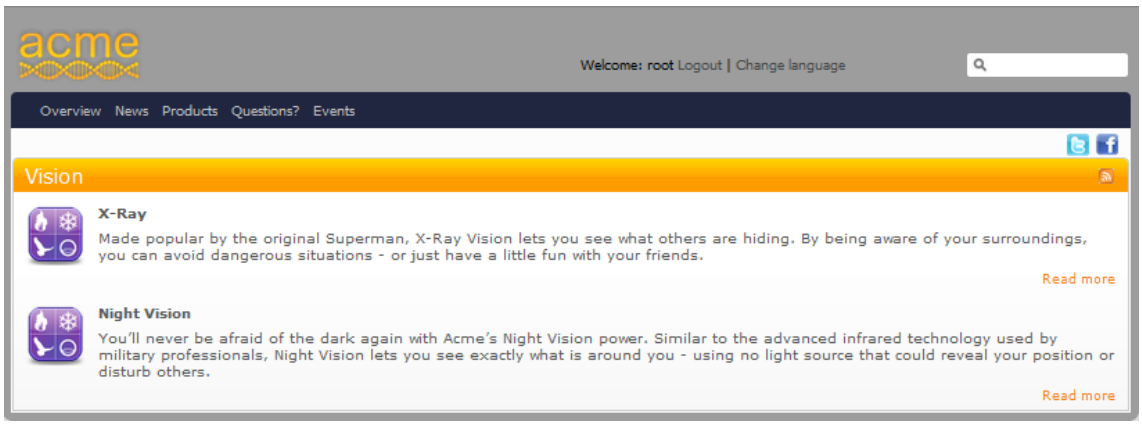
4. Go back to the homepage. You will see that your newly created document is added to the contextual menu.



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

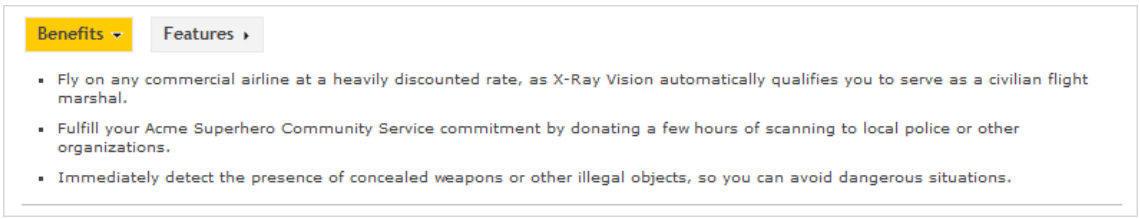


You can click the **Vision** sub-menu and see content of **Vision** directory rendered in the **catalog** page:

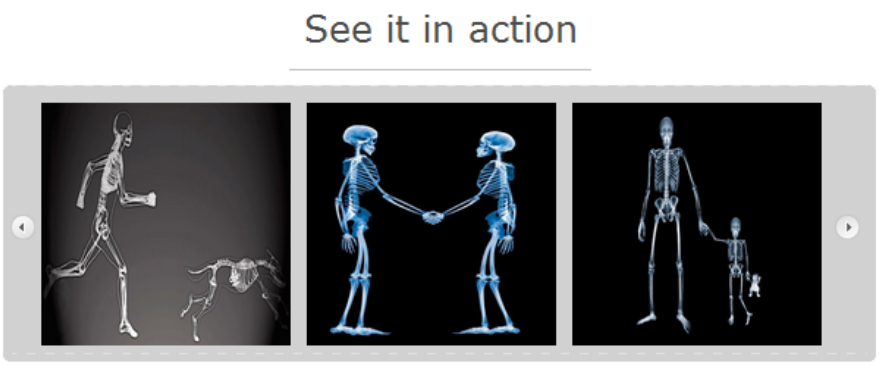


Select the **X-Ray** content and see the newly implemented content using new visual effects and presentation.

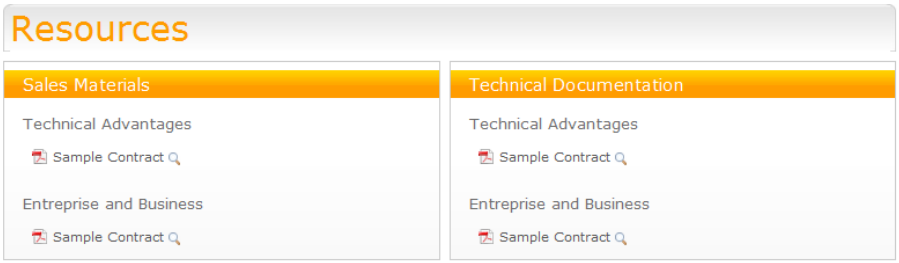
- "Benefits" and "Features" tabs:



- Coverflow section:



- Related documents:



3.6.4. Create data for Navigation By Content

- Create a Product page

  Steps to create content and to add media files to enrich the Product page.

- Develop your own Product content

  Steps to create fields in the Product content type, and to develop the Product's view form.

Note

By following 2 typical examples above, you can create data for Navigation By Content easily.

3.6.4.1. Create a Product page

Create content about the product

1. Go to the **Sites Explorer** page and navigate to *somePath/someDirectory*.

2. Click the **New Content** button on the **Action** bar. There will be a several content type appears.

3. Select the **Product** content type.

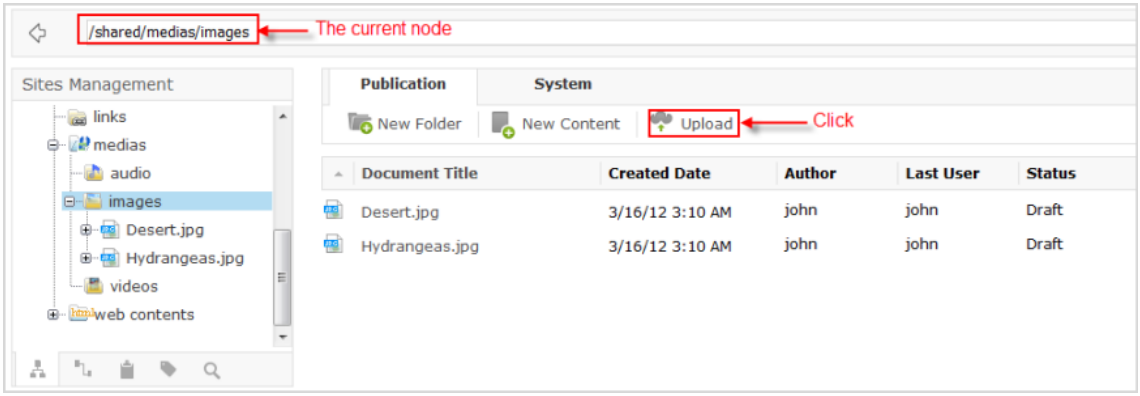
4. Fill the **Product** dialog form.

5. Save changes.

Add media files to enrich the Product page

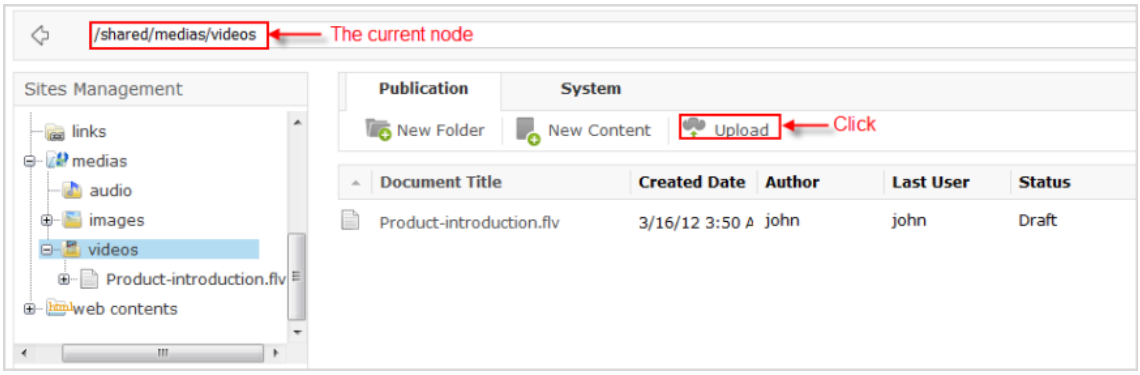
1. Go to *somePath/someDirectory/sampleProduct/medias/images*.

2. Click **Upload** on the **Action** bar to upload some images from your local device and publish them.



3. Go to *somePath/someDirectory/sampleProduct/medias/videos*.

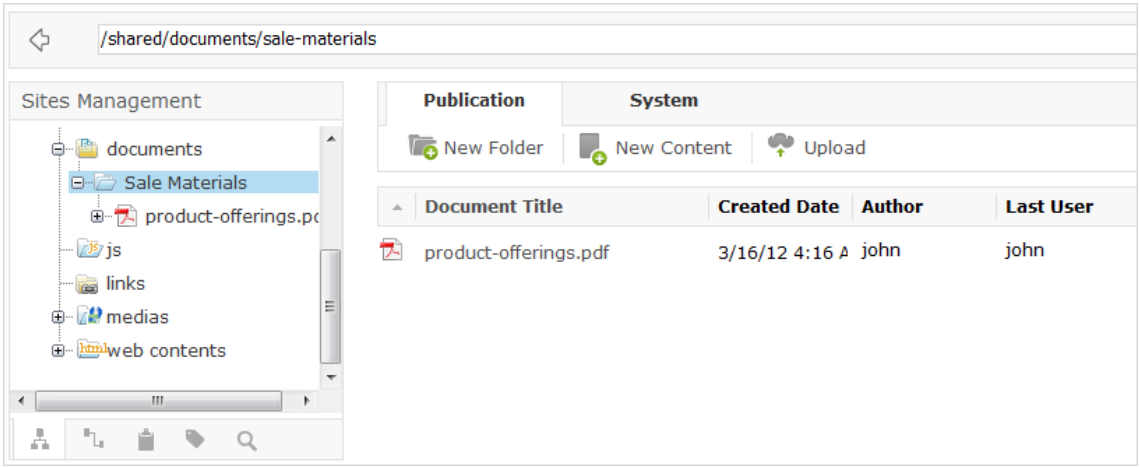
4. Click **Upload** on the **Action** bar to upload some videos from your local device and publish it.



5. Go to *somePath/someDirectory/sampleProduct/documents*.

6. Create two directories, including **Sale Materials** and **Technical Documentations**.

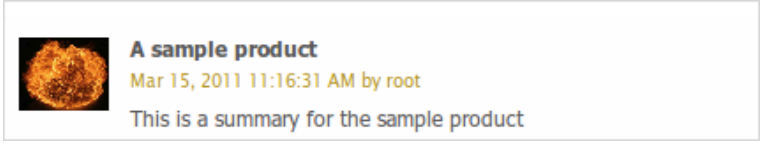
7. Upload a PDF document and publish it under each sub-folder.



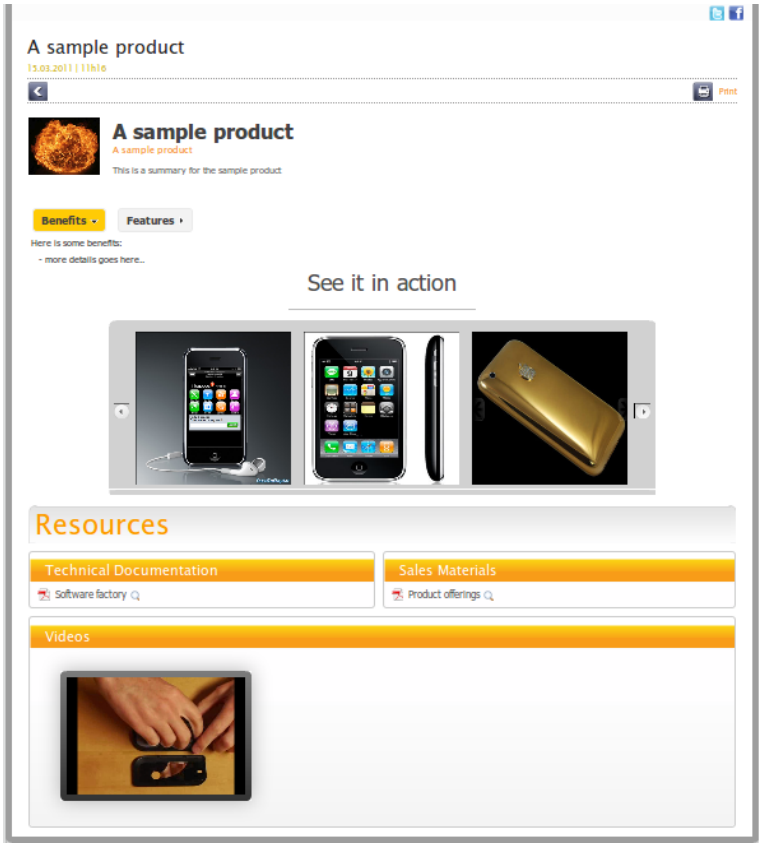
8. Add your **sampleProduct** to some categories or add it to the **Content List** portlet.

Your newly created product is now 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:



3.6.4.2. Develop your own Product content

The sample **Product** page is composed of the following fields and folders:

- **Product** content type fields: Name, Title, Illustration Image, Summary, Benefits, and Features.

(The **Product** content type is the template specified for the **Product** page.)

- Other content folders: documents, medias/images, medias/videos.

(These folders contain documents and media files to enrich the **Product** page.)

Create fields in the Product content type

- **Name** Other content 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","validate=name,empty"] ;
 uicomponent.addTextField("name", productFieldName) ;
 String[] documentsFolder = ["jcrPath=/node/documents", "nodetype=nt:folder","mixintype=exo:documentFolder", "defaultValues=documents"] ;
 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", "mixintype=mix:referenceable", "defaultValues=illustration"];
 String[] illustrationHiddenField2 = ["jcrPath=/node/medias/images/illustration/jcr:content", "nodetype=nt:resource", "mixintype=dc:elementSet", "visible=false"];
 String[] illustrationHiddenField3 = ["jcrPath=/node/medias/images/illustration/jcr:content/jcr:encoding", "visible=false", "UTF-8"];
 String[] illustrationHiddenField4 = ["jcrPath=/node/medias/images/illustration/jcr:content/jcr:lastModified", "visible=false"];
 String[] illustrationHiddenField5 = ["jcrPath=/node/medias/images/illustration/jcr:content/dc:date", "visible=false"];
 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") && (uicomponent.findComponentById(illustration) == null)) {
    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/illustration/jcr:content");
     %>
      <div>
       <image src="$imgSrc" width="100px" height="80px"/>
       <a onclick="$actionLink">
        <img src="/eXoResources/skin/DefaultSkin/background/Blank.gif" class="ActionIcon Remove16x16Icon"/>
       </a>
      </div>
     <%
    } else {
     setUploadFields(illustration);
    }
   } else {
    setUploadFields(illustration);
   }
  %>
 </td>
</tr>
```

- **Summary:**

```
<tr>
 <td class="FieldLabel"><%=_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:CompleteWCM", ""] ;
  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:CompleteWCM", ""] ;
  uicomponent.addRichtextField("productFeatures", productFieldFeatures) ;
 %>
 </div>
 </td>
</tr>
```

Develop the Product's view form

The illustration image, title and summary are grouped together:

```
<!-- Hot news -->
<div class="BigNews ClearFix">
 <!-- Begin illustrative image -->
  <%
  RESTImagesRendererService imagesRenderer = uicomponent.getApplicationComponent(RESTImagesRendererService.class);
  def imageURI = imagesRenderer.generateImageURI(currentNode.getNode("medias/images/illustration"),null);
  if (imageURI != null){
  %>
 <a class="Image"><img width="93" src="$imageURI" alt=""></a>
  <%
  }
  %>
 <div class="Content">
 <!-- Begin title -->
 <%
   if(currentNode.hasProperty("exo:title")) {
  def title = currentNode.getProperty("exo:title").getString();
  %>
  <a href="#" class="Title">$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>
```

In which:

- Name: The name of the product.
- Title: The title of the product.
- Illustration Image: The image that is used as an illustration for the product.
- Summary: The summary about the product that goes with the illustration.
- Benefits: The benefits of the product.
- Features: The features of the product.

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

```
<div id="sectionsTab" 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>
   <!-- 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")%></a>
   <!-- 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 used to display the product's images (in the coverflow view) from the images folder.

```
<div class="jQProBoxC">
<!-- Begin jCarouselLite part -->
<button class="jQprev"> </button>
<div class="jCarouselLite">
 <ul>
   <%
        FOR IMAGE IN PRODUCT'S IMAGE FOLDER
  String imgSrc = "";
  /*
  GET THE IMAGE PATH
  imgSrc = GET THE IMAGE PATH;
  */
  %>
  <li><img src="$imgSrc" width="204" height="200"/></li>
  <%
    %>
 </ul>
</div>
<button class="jQnext"> </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 to a document or video.

2. Use some customized CSS classes to display a link for this node.

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

The use of this instruction is described as below:

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

This can be 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>
```

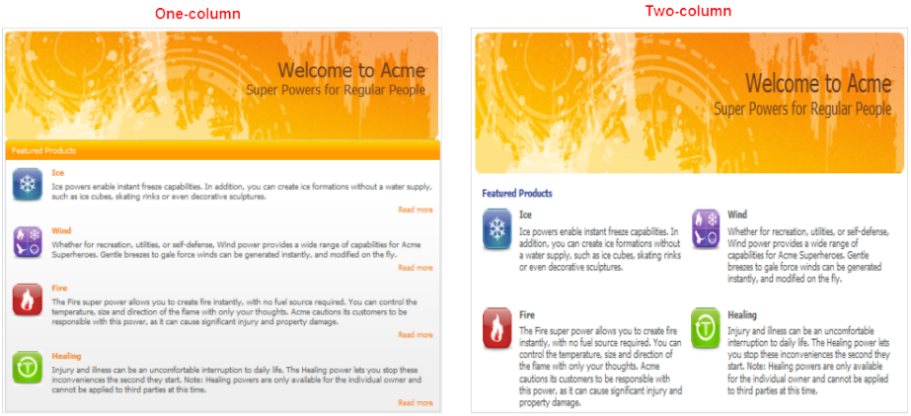
Make sure that locale files are added to the resource bundle configuration. If locale files (dialogs and views) are under the *classes/locale/wcm* directory, use the following code:

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

3.6.5. Create a new Content List template

eXo Platform provides many powerful features to manipulate and expose any types of content on a page. This is due to the fact that eXo Platform stores all the content in its Java Content Repository (JCR) and renders the content on a page using Groovy Templates.

In this section, you will learn how to create a new template that is used in the **Content List** portlet. For example, in the sample ACME site, you can show the content in One-column or Two-column display just by selecting different templates:



Before writing a new template, it is important to learn where templates are stored.

eXo Content Service: Template Storage

Like many things inside eXo Platform, eXo JCR is used to store templates. Templates are just a special type of content. This allows developers to easily write and test code without following a complex deployment process, but also it makes it easy to export a running configuration to another one. To do this, you just need to use the standard JCR export/import features.

All templates and eXo Content Service configurations are stored inside a specific JCR workspace named "dms-system".

Each template type (for example, Document Type, or Content List) is stored in a specific location. In this case, you are going to work on the **Content List** portlet so templates are stored inside the `/exo:ecm/views/templates/content-list-viewer/list/` folder.

The following steps allow you to inspect this folder using the eXo CRaSH utility. If you are not interested in it, you can jump to the next section [84]. CRaSH is a shell for Java Content Repositories, the source of CRaSH is available on Google Code. So in the terminal window:

1. Connect to CRaSH using the telnet client: ***`telnet 127.0.0.1 5000`***.

2. Connect to the JCR workspace using the following command: ***`connect -u root -p gtn -c portal dms-system`***.

Where: `-u` is the user, `-p` is the password, `-c` is the Portal Container, and `dms-system` is the workspace to use.

3. Move the folder that contains all templates for the **Content List** portlet: ***`cd /exo:ecm/views/templates/content-list-viewer/list/`***.

4. List all templates using the `ls` command.

You can see the list of all templates available for the **Content List** portlet.

Create a new Content List template using IDE

Before doing the following steps, be sure that you are connected to a user who belongs to the */Developer* group. The *root* user can be used for the simplicity reason.

1. Access IDE by clicking **--> IDE**.

2. Switch to the *dms-system* workspace by clicking **My Spaces --> on Window --> Select Workspace** from the IDE menu. Next, select the *dms-system* location in the dialog box and click **OK**.

3. Navigate to the template location from the file structure on the left: `/exo:ecm/views/templates/content-list-viewer/list/`

4. Create a new template by clicking **File --> New --> Groovy Template** from the IDE menu.

5. Save the file as "MyNewTemplate.gtmpl".

6. Enter some basic codes:

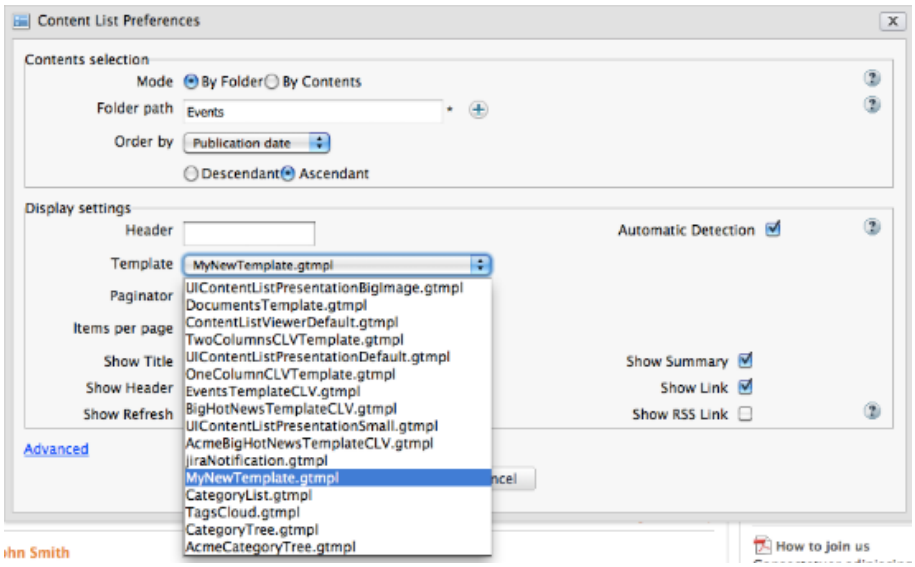
```
<h1>This is my template</h1>
The date is <= new Date()>
```

7. Save the template, then go back to the homepage of the ACME site.

8. Switch to the **Edit** mode by clicking **Edit** on the **Administration** bar.

9. Hover your cursor over the top of the list of news and click .

10. Select "MyNewTemplate" from the list of templates, then click **Save**.



You have created your new template, and use it on a page. Now, you should add some more interesting codes to the template to really loop over the content based on the portlet configuration. But before this, you need to understand caching and code modification.

eXo Template and Cache

To improve performance and a running system, the compiled version of the template is cached by default. This is the reason why you do not see any change when you are modifying a template. There are 2 ways to work around this:

- Run eXo Platform in the Debug mode. In this case, nothing is cached.

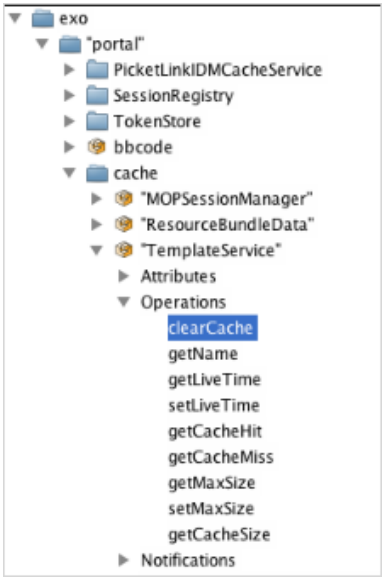
- Invalidate the cache manually using JMX.

Since working with no cache at all is not an option, here is the MBean you have to use for invalidating the Template Service cache:

- `exo:portal="portal",service=cache,name="TemplateService"`

Then, call the `clearCache` operation on it.

You can use any methods to call your MBeans operation. Here, JConsole will be used:



Do not forget to call this operation each time you modify your template to ensure that eXo recompiles the template.

Access content in the template

The current code of the template is really simple. Now, you need to add code to print the content in the page. To do this, you are going to use some **Content** programmings once again in IDE.

The template used by the **Content List** portlet is based on the following Java class: `org.exoplatform.wcm.webui.clv.UICLVPresentation`. This class is responsible for setting the complete context that you can use in the template, such as:

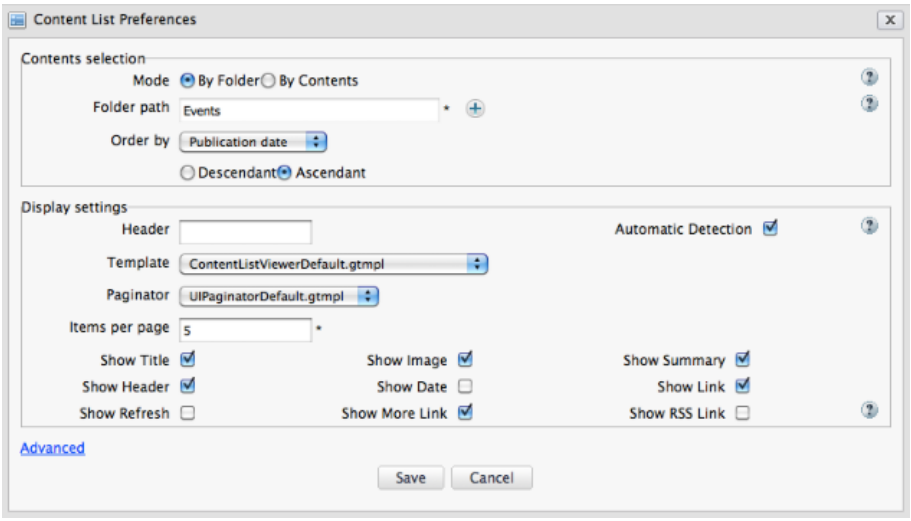
- The folder or category that contains the content to show. The **Folder Path** field is in the preference screen.
- The display settings: title, number of documents, elements to show and more.

Here is the code to access these preferences:

```
// import all the classes need in the template
import javax.jcr.Node;
import org.exoplatform.wcm.webui.paginator.UICustomizeablePaginator;
import org.exoplatform.wcm.webui.clv.UICLVPortlet;
import org.exoplatform.wcm.webui.Utils;
import org.exoplatform.services.wcm.core.NodeLocation;

// get the portlet preferences

def header = uicomponent.getHeader();
def isShowRssLink = uicomponent.isShowRssLink();
def isShowHeader = uicomponent.isShowField(UICLVPortlet.PREFERENCE_SHOW_HEADER);
def isShowRefresh = uicomponent.isShowField(UICLVPortlet.PREFERENCE_SHOW_REFRESH_BUTTON);

def isShowTitle = uicomponent.isShowField(UICLVPortlet.PREFERENCE_SHOW_TITLE);
def isShowDate = uicomponent.isShowField(UICLVPortlet.PREFERENCE_SHOW_DATE_CREATED);
def isShowLink = uicomponent.isShowField(UICLVPortlet.PREFERENCE_SHOW_LINK);
def isShowReadmore = uicomponent.isShowField(UICLVPortlet.PREFERENCE_SHOW_READMORE);
def isShowImage = uicomponent.isShowField(UICLVPortlet.PREFERENCE_SHOW_ILLUSTRATION) ;
```

```
def isShowSummary = uicomponent.isShowField(UICLVPortlet.PREFERENCE_SHOW_SUMMARY);
```

The `uicomponent` object is defined by the container class of the portlet that calls the template. This class contains many utility methods. The code above retrieves all the preferences of the portlet. Because the name is self-explanatory, it is not necessary to detail them, especially when you look at the preferences screen below:



Now, the template has all the preferences, it is time to loop on the content on printing the information.

The **Content Service** provides API to manipulate the content, including pagination of content. The idea behind this is to let the Content Service manage the JCR query, sorting, caching and pagination of data. So in your template, you will mainly manage 2 classes to loop through the content to show:

- `uicomponent.getUIPageIterator()` - a paginator object that is configured based on the portlet preferences.
- `uicomponent.getCurrentPageData()` - a list of the content (JCR Nodes) that should be printed on the current page.

So, you can print all the content of the page as a simple HTML list:

```
<ul style="margin: 20px">
<%
    for (viewNode in uicomponent.getCurrentPageData()) {
      def title = viewNode.getProperty("exo:title").getString()
      print("<li>$title</li>");
    }
%>
</ul>
```

Just copy this code to your template, save it, then refresh the cache and go to your page. You should see the list of the content in a simple HTML list.

On each content (Node), the **Content API** provides some helper methods to easily manipulate the content and avoid using the JCR API directly. In the following code, you can see the most important methods accessing the content properties:

```
def itemName = viewNode.getName();
def itemLink = uicomponent.getURL(viewNode);
def webdDavLink = uicomponent.getWebdavURL(viewNode);
def itemDateCreated = uicomponent.getCreatedDate(viewNode);
def itemModifiedDate = uicomponent.getModifiedDate(viewNode);
def itemAuthor = uicomponent.getAuthor(viewNode);
def imgSrc = uicomponent.getIllustrativeImage(viewNode);
def itemTitle = uicomponent.getTitle(viewNode);
```

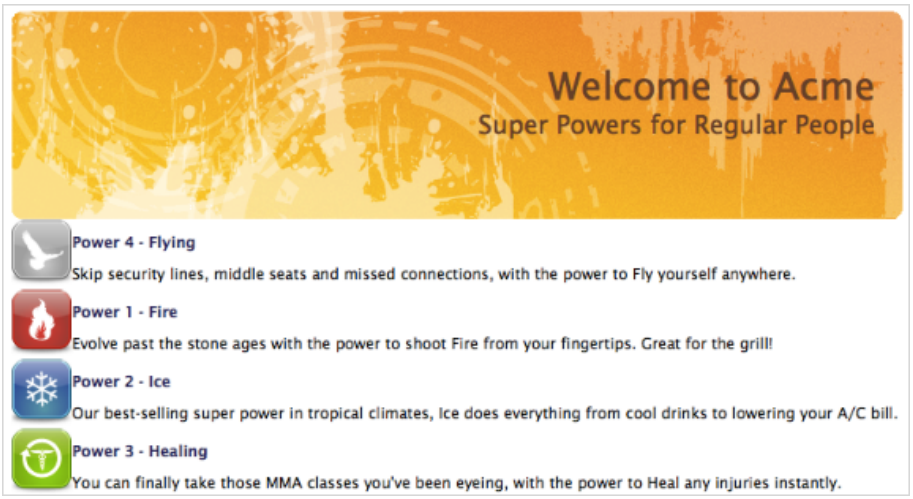
```
def itemSummary = uicomponent.getSummary(viewNode);
```

One important point is the fact that these methods are responsible for many things, for example: formatting dates, returning complete URLs that depends on the context of the portlet.

Based on these methods, you can now work on the presentation of the information on the page. For example, you can click the image and the title to go in the detailed view of the article. This is done simply by using the following code:

```
<%
  for (viewNode in uicomponent.getCurrentPageData()) {
    def itemName = viewNode.getName();
    def itemLink = uicomponent.getURL(viewNode);
    def webdDavLink = uicomponent.getWebdavURL(viewNode);
    def itemDateCreated = uicomponent.getCreatedDate(viewNode);
    def itemModifiedDate = uicomponent.getModifiedDate(viewNode);
    def itemAuthor = uicomponent.getAuthor(viewNode);
    def imgSrc = uicomponent.getIllustrativeImage(viewNode);
    def itemTitle = uicomponent.getTitle(viewNode);
    def itemSummary = uicomponent.getSummary(viewNode);

     %>
<div style="overflow: auto;">
  <img src="$imgSrc" align="left">
  <h3><a href="$itemLink">$itemTitle</a></h3>
  $itemSummary
</div>

<%
  }
%>
```

For the simplicity reason, this code does not manage any null value. Also, the template does not deal with the portlet preferences, such as the "Header", "RSS" links. The Website should look like:



The last important point is to add the support for the in-context editing that allows users to edit the content directly from the template. Once again, this is done with a method of the `uicomponent` object that creates a DIV around the content:

```
<%
  for (viewNode in uicomponent.getCurrentPageData()) {
    def itemName = viewNode.getName();
    def itemLink = uicomponent.getURL(viewNode);
    def webdDavLink = uicomponent.getWebdavURL(viewNode);
    def itemDateCreated = uicomponent.getCreatedDate(viewNode);
    def itemModifiedDate = uicomponent.getModifiedDate(viewNode);
    def itemAuthor = uicomponent.getAuthor(viewNode);
```

```
    def imgSrc = uicomponent.getIllustrativeImage(viewNode);
    def itemTitle = uicomponent.getTitle(viewNode);
    def itemSummary = uicomponent.getSummary(viewNode);

    %>
<div style="overflow: auto;">
  <%=uicomponent.addQuickEditDiv("MyTemplateContentEditor", viewNode)%>
  <img src="$imgSrc" align="left">
  <h3><a href="$itemLink">$itemTitle</a></h3>
  $itemSummary
  < /div>
</div>

<%
  }
%>
```

The 15 and 19 lines are new in this template and provide support for the **Quick Edit** feature.

After creating your own template for **Content Service** using the embedded IDE, you are free to use your imagination for adding cool features to your site.

Work With Applications

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

This chapter includes the following main topics:

- **Integrate an application**

  Instructions on how to add an application (especially a portlet) to your portal's paces.
- **Develop your own application**

  Instructions on how to develop a gadget for eXo Platform, and information about Portlet Bridges.
- **Extend eXo applications**

  Concept and mechanism of UI Extension framework which allows the customization and extensibility of eXo applications through simple plugins.

4.1. Integrate an application

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 in the *pages.xml* file:

```
<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.
`portlet-ref`	The name of the portlet.
`title`	The title of the page.
`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 *preferences* 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". To learn more, refer to eXo JCR and Extension Services Reference.

See also

- Develop your own application
- Extend eXo applications

4.2. Develop your own application

- Develop a gadget for eXo Platform

  Introduction to the resources which can be shared for all gadgets and their categories, instructions on how to apply for a gadget and to customize the gadget thumbnail.

- Portlet Bridges

  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.

See also

- Integrate an application
- Extend eXo applications

4.2.1. Develop a gadget for eXo Platform

Note

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 the Portlet development and Gadget development sections.

Because a gadget is basically an independent HTML content, the gadget UI, such as layout, font, color may be different. Thus, making consistent in the look and feel of all eXo Platform gadgets is very important. In this part, it is assumed that developers have already known how to write a gadget in eXo Platform. The information here helps developers make a consistent look and feel with eXo Platform skin, even when this skin may be changed in the future.

Note

To get more information on how to develop gadgets, see the Gadget development and Develop gadgets via a powerful web-based IDE of eXo Platform sections.

4.2.1.1. Resources

To achieve the consistent look and feel, you have to collect the common features of all gadgets as much as possible and put in a place where it can be shared for all gadgets. You will use exo-gadget-resources for this purpose. It is a *.war* file that contains commonly used static resources (stylesheets, images, JavaScript libraries, and more) available for all gadgets in eXo Platform at runtime:

```
/exo-gadget-resources
 |__skin
 |  |__exo-gadget
 |  |  |__images
 |  |  |__gadget-common.css
 |  |__...(3rd-party components' CSS)
 |__script
    |__jquery
    |  |__1.6.2
    |  |__...(other jQuery versions)
    |  |__plugins
    |__utils
```

The resources are divided into 2 categories: skin and script.

Skin: is the place for the shared stylesheets of the gadgets (*exo-gadget/gadget-common.css*) and other third-party components styles adapted to the eXo Platform skin (jqPlot, Flexigrid, and more). This is a copy of the component's original CSS with some modifications to match the eXo Platform's skin. You can find this original file at the component's folder under *exo-gadget-resources/script* , then link to it or make your own copy (put it in your gadget's folder and refer to it in gadget's .xml file) to suit your need.

The *gadget-common.css* file is the main place for the global stylesheets. When the eXo Platform skin is changed, updating stylesheets in this file will affect all gadgets skins accordingly. In this file, you will define stylesheets applied for all gadgets, such as gadget title, gadget body, fonts, colors, tables, and some commonly used icons, such as drop-down arrow, list bullet, setting button, and more.

For example:

- UIGadgetThemes: the gadget container.
- TitGad: the gadget title.
- ContTit: the gadget title content.
- GadCont: the gadget content.
- SettingButton: the setting button for gadget's User Preferences.

Script: is the place for commonly used third-party JavaScript libraries (e.g: jQuery and its plugins) and a library of useful utility scripts (the utils folder).

jQuery and plugins:

- *jquery/<version>*: jQuery JavaScript library.
- *jquery/plugins/jqplot*: Charts and Graphs for jQuery.
- *jquery/plugins/flexigrid*: Lightweight but rich data grid.
- *jquery/plugins/date.js*: JavaScript date library.

- *jquery/plugins/jquery.timers*: JavaScript timer.

(Here you should keep the latest and several versions of jQuery because some plugins may not work with the latest version. Several versions of a plugin are also kept).

The utilities scripts:

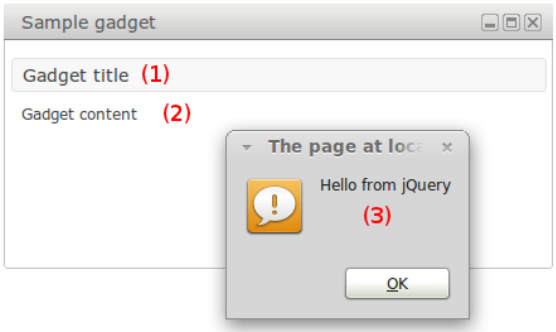
- *utils/pretty.date.js*: Calculate the difference from a point of time in the past to the present and display "4 months 3 weeks ago", for example.

4.2.1.2. Apply for a gadget

A gadget should use static resources available in *exo-gadget-resources* instead of including them in their own package. This helps reduce packaging size, avoid duplication (considering that every gadget includes a version of jQuery is in its own package) and take advantages of automatic skin changing/updating when the resources of *exo-gadget-resources* are updated to the new eXo Platform skin. To make it work, the applied gadget must use the CSS classes defined in *gadget-common.css* (at least for the gadget title) like the sample gadget below:

```
<Module>
<ModulePrefs title="Sample gadget"/>
<Content type="html">
 <head>
  <link href="/exo-gadget-resources/skin/exo-gadget/gadget-common.css" rel="stylesheet" type="text/css"/>
  <script language="javascript" src="/exo-gadget-resources/script/jquery/1.6.2/jquery.min.js" type="text/javascript"/>
  <script language="javascript" type="text/javascript">
   $(function(){
    alert("Hello from jQuery");  (3)
   });
  </script>
 </head>
 <body>
  <div class="UIGadgetThemes">
   <div class="TitGad ClearFix">
    <div class="ContTit">Gadget title</div>  (1)
   </div>
   <div class="GadCont">  (2)
    Gadget content
   </div>
  </div>
 </body>
</Content>
</Module>
```

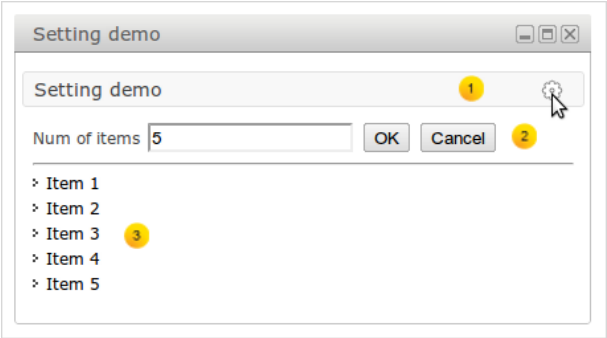
The sample gadget:



The sample user settings of a gadget

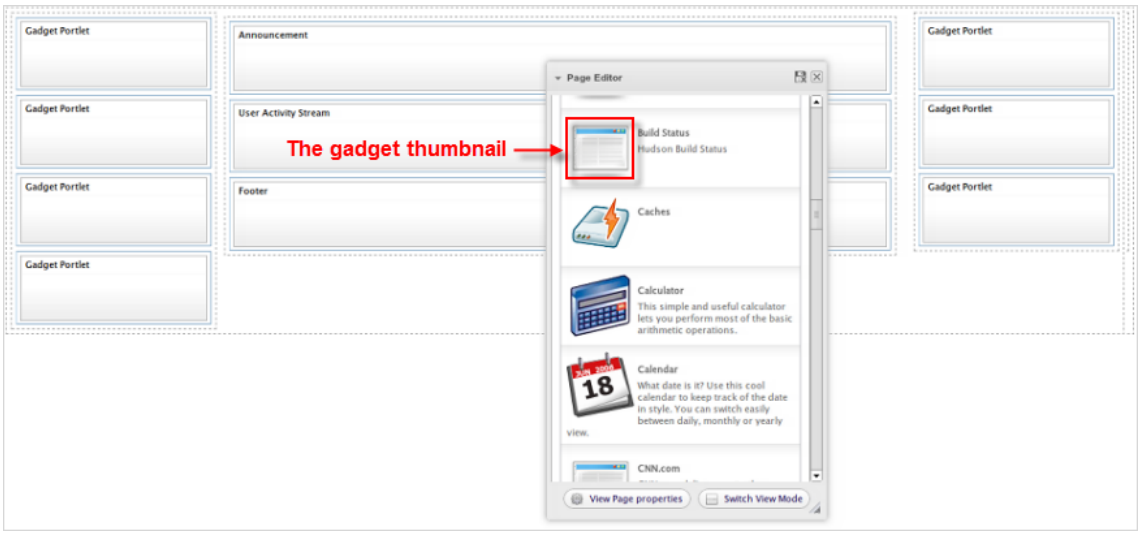
The following gadget gives the demo of a user settings screen which uses eXo Platform standard icons (setting button, list item bullet, and more) that are defined in the *gadget-common.css*.

```
<Module>
<ModulePrefs title="Setting demo">
  <Require feature="dynamic-height"/>
  <Require feature="setprefs"/>
</ModulePrefs>
<Content type="html">
  <head>
    <link href="/exo-gadget-resources/skin/exo-gadget/gadget-common.css" rel="stylesheet" type="text/css"/>
    <style type="text/css">
      .SettingButton:hover {cursor:pointer;}
    </style>
    <script language="javascript" src="/exo-gadget-resources/script/jquery/1.6.2/jquery.min.js" type="text/javascript"/>
    <script language="javascript" type="text/javascript">

      $(function(){
        var prefs = new gadgets.Prefs();
        var defaultNumItems = 3;

        function displayItems(){
          var numItems = prefs.getInt("numItems") || defaultNumItems;
          $("#content").html("");
          for(i=0; i&lt;=numItems; i++) {
            $("#content").append("&lt;div class='IconLink'&gt;Item " + (i+1) + "&lt;/div&gt;");    (3)
          }
          gadgets.window.adjustHeight($(".GadCont").get(0).offsetHeight);
        }

        $(".SettingButton").live("click", function(){
          $("#txtNumItems").val(prefs.getInt("numItems") || defaultNumItems);
          $("#setting").toggle();
          gadgets.window.adjustHeight($(".GadCont").get(0).offsetHeight);
        });

        $("#btnOk").live("click", function(){
          var sNumItems = $("#txtNumItems").val();
          prefs.set("numItems", parseInt(sNumItems) || defaultNumItems);
          $("#setting").hide();
          displayItems();
        });

        $("#btnCancel").live("click", function(){
          $("#setting").hide();
          gadgets.window.adjustHeight($(".GadCont").get(0).offsetHeight);
        });

        displayItems();
      });

    </script>
  </head>
  <body>
    <div class="UIGadgetThemes">
      <div class="TitGad ClearFix">
        <div class="ContTit">
          Setting demo
          <div class="SettingButton" style="display:block; width:20px; height:20px"/>   (1)
        </div>
      </div>
      <div class="GadCont">
        <div id="setting" style="display:none;">   (2)
          <label for="txtNumItems">Num of items</label>
          <input id="txtNumItems" type="text"/>
          <input id="btnOk" type="button" value="OK"/>
          <input id="btnCancel" type="button" value="Cancel"/>
          <hr/>
        </div>
        <div id="content"/>
      </div>
    </div>
  </body>
</Content>
</Module>
```



4.2.1.3. Customize the gadget thumbnail

The gadget thumbnails are displayed in the **Page Editor** window when you edit a page. The thumbnail image size needs to be consistent for all gadgets in the list. The current size (in eXo Platform 3.5) is 80 x 80px, so you should select an image of this size in PNG (preferred), JPG or GIF format for your gadget thumbnail.



- The image can also be one from a public website (absolute URL), for example: <ModulePrefs title="Sample gadget" thumbnail="http://www.example.com/images/sample-icon.jpg"/>; or from an internal image (relative URL): <ModulePrefs title="Sample gadget" thumbnail="image/sample-icon*80x80.png"/>.

4.2.2. 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 JavaTM 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. For more details, see 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. For more details, see http://jboss.org/portletbridge/docs.html.

4.3. Extend eXo applications

- **UI Extension components**

Necessary information about UIExtensionManager, UIExtensionPlugin, introduction to UI Extension definition and its class, parent UI components, and details about filters of each UI Extension.

- **Mechanism**

  Details about the working process of each UI Extension, including: setting up, loading and activating.

What is UI Extension framework?

UI Extension is an extension of UI Component. As you know, extension is an object that contains programming for extending the capabilities of data available to a more basic program. Here, UI Extension helps expanding the dynamic children of UI Component. With UI Extension, you can add, change or remove a lot of children in UI Component more easily than in traditional ways.

Why use UI Extension framework?

It is simple for you and your team to control applications containing few components that are all constant. But when you start an application which contains a lot of components, transactions, filters and permissions on each component, it is really a disaster. As each developer may handle problems in their own way, it also likely raises the convention problem. Thus, UI Extension framework was created to solve the management dynamic components on the applications and free developers from controlling too many of them.

The main goals of this framework are:

- Create simple child UI Components.
- Apply a filter on each component for a variety of purposes more easily.
- Add or remove extensions simply by configuration.

See also

- How to create an activity plugin for Social?
- How to make your own ECMS UI Extensions?
- How to add action extensions to Wiki?

4.3.1. UI Extension components

UIExtensionManager

This class is used to manage all extensions available in the system. The target is to create the ability to add a new extension dynamically without changing anything in the source code. UIExtensionManager is implemented by *UIExtensionManagerImpl*.

```
<component>
  <key>org.exoplatform.webui.ext.UIExtensionManager</key>
  <type>org.exoplatform.webui.ext.impl.UIExtensionManagerImpl</type>
</component>
```

UIExtensionPlugin

This class allows you to define new extensions in the configuration file dynamically (for example: *configuration.xml*). As you want *UIExtensionManager* to manage every extension, you have to plug *UIExtensionPlugin* into it:

```
<external-component-plugins>
   <target-component>org.exoplatform.webui.ext.UIExtensionManager</target-component>
   <component-plugin>
    <name>add.action</name>
    <set-method>registerUIExtensionPlugin</set-method>
    <type>org.exoplatform.webui.ext.UIExtensionPlugin</type>
      ...
   </component-plugin>
</external-component-plugins>
```

Definition of UI Extensions

Each UI Extension is defined as an object param:

```
...
<object-param>
        <name>EditPage</name>
        <object type="org.exoplatform.webui.ext.UIExtension">
          <field name="type"><string>org.exoplatform.wiki.UIPageToolBar</string></field>
          <field name="rank"><int>300</int></field>
          <field name="name"><string>EditPage</string></field>
          <field name="component"><string>org.exoplatform.wiki.webui.control.action.EditPageActionComponent</string></field>
        </object>
 </object-param>
...
```

In which:

- **Name**: the extension's name.
- **Object Type**: point to the UI Extension lib class.
  - **Type**: the "parent" UI component which is extended by your UI Extension.
  - **Rank**: used to sort by Collection of UI Extension.
  - **Component**: point to the UI Extension definition class.

UI Extension Definition class

This class is used to define filters, actions and templates of UI Extension:

```
@ComponentConfig(
  events =
{(listeners = EditPageActionComponent.EditPageActionListener.class);})
  public class EditPageActionComponent extends UIComponent {
  private static final List<UIExtensionFilter> FILTERS = Arrays.asList(new UIExtensionFilter[] { new IsViewModeFilter() });
  @UIExtensionFilters
   public List<UIExtensionFilter> getFilters() {
    return FILTERS;
}
  public static class EditPageActionListener extends UIPageToolBarActionListener<EditPageActionComponent> {
    @Override
    protected void processEvent(Event<EditPageActionComponent> event) throws Exception {
      ...
      super.processEvent(event);
    }
  }
...
```

Parent UI Component

This is what your UI Extension will be added to (in this example, the parent UI Componet is *UIPageToolBar*). All extensions of this component are got by UIExtensionManager.

```
UIExtensionManager manager = getApplicationComponent(UIExtensionManager.class);

List<UIExtension> extensions = manager.getUIExtensions(EXTENSION_TYPE);
 public List<ActionComponent> getActions() throws Exception {
   ....
   List<UIExtension> extensions = manager.getUIExtensions(EXTENSION_TYPE);
   if (extensions != null) {
     for (UIExtension extension : extensions) {
       UIComponent component = manager.addUIExtension(extension, context, this);
// Child UI Component has been made by UI Extension
// It's available to use now
       ...
     }
   }
   return activeActions;
 }
```

Internal filter

Each UI Extension has a list of filters depending on variety of purposes. It indicates which UI Extension is accepted and which is denied. You are free to create your own filter extended from *UIExtensionAbstractFilter*. Internal filters are a part of the business logic of your component. For example, if your component is only dedicated to articles, you will add an internal filter to your component that will check the type of the current document.

```
public class IsViewModeFilter extends UIExtensionAbstractFilter {
  public IsViewModeFilter(String messageKey) {
   super(messageKey, UIExtensionFilterType.MANDATORY);
 }
 @Override
 public boolean accept(Map<String, Object> context) throws Exception {
   UIWikiPortlet wikiPortlet = (UIWikiPortlet) context.get(UIWikiPortlet.class.getName());
   return(wikiPortlet.getWikiMode() == WikiMode.VIEW||wikiPortlet.getWikiMode() == WikiMode.VIEWREVISION);
 }
 @Override
 public void onDeny(Map<String, Object> context) throws Exception {
   // TODO Auto-generated method stub
 }
```

Your filter will define which type of filter it belongs to (in *UIExtensionFilterType*). There are 4 types:

Types	Description
MANDATORY	Check if the action related to the extension can be launched and if the component related to the extension can be added to the WebUI tree. This filter is required to launch the action and add the component related to the extension to the WebUI tree. If it succeeds, you need to check the other filters. If it fails, you need to stop.
REQUISITE	Check if the action related to the extension can be launched. This filter is required to launch the action to the WebUI tree. If it succeeds, you need to check the other filters. If it fails, you need to stop.
REQUIRED	

Types	Description
	Check if the action related to the extension can be launched and can be used for adding warnings. This filter is required to launch the action. If it succeeds or fails, you need to check the other filters.
OPTIONAL	Check if the action related to the extension can be launched and can be used for the auditing purpose. This filter is not required to launch the action. If it succeeds or fails, you need to check the other filters.

There are 2 conditions for filtering: Accept and onDeny.

- **Accept**: Describe the "Accept" condition, and how an UI Extension can accept by a context.
- **onDeny**: What you will do after the filter denies an UI Extension by a specific context (generating a message for pop-up form, for example).

You have known how and where the filter is put in an UI Component, but when it is gonna fire?

It falls into 2 situations: when you get it and when it is action fire. Thus, you should ensure that your UI Extension is always trapped by its filter.

External filter

External filters are mainly used to add new filters that are not related to the business logic to your component. A good example is the *UserACLFilter* which allows you to filter by access permissions.

For example, to make the EditPage action only be used by *manager:/platform/administrators*, do as follows:

- Create an external filter:

```
public class UserACLFilter implements UIExtensionFilter {
  /**
   * The list of all access permissions allowed
   */
  protected List<String> permissions;
  /**
   * {@inheritDoc}
   */
  public boolean accept(Map<String, Object> context) throws Exception {
    if (permissions == null || permissions.isEmpty()) {
      return true;
    }
    ExoContainer container = ExoContainerContext.getCurrentContainer();
    UserACL userACL = (UserACL) container.getComponentInstance(UserACL.class);
    for (int i = 0, length = permissions.size(); i < length; i++) {
      String permission = permissions.get(i);
      if (userACL.hasPermission(permission)) {
        return true;
      }
    }
    return false;
  }

  /**
   * {@inheritDoc}
   */
  public UIExtensionFilterType getType() {
    return UIExtensionFilterType.MANDATORY;
  }

  /**
   * {@inheritDoc}
   */
```

```
  public void onDeny(Map<String, Object> context) throws Exception {}
}
```

- Add the external filter to an UI Extension in the *configuration.xml* file:

```
<object-param>
  <name>EditPage</name>
  <object type="org.exoplatform.webui.ext.UIExtension">
    <field name="type"> <string>org.exoplatform.wiki.UIPageToolBar</string> </field>
 <field name="rank"><int>300</int></field>
    <field name="name"> <string>EditPage</string> </field>
    <field name="component"><string>org.exoplatform.wiki.webui.control.action.EditPageActionComponent</string> </field>
    <!-- The external filters -->
    <field name="extendedFilters">
      <collection type="java.util.ArrayList">
        <value>
          <object type="org.exoplatform.webui.ext.filter.impl.UserACLFilter">
            <field name="permissions">
              <collection type="java.util.ArrayList">
                <value>
                  <string>manager:/platform/administrators</string>
                </value>
              </collection>
            </field>
          </object>
        </value>
      </collection>
    </field>
  </object>
</object-param>
```

4.3.2. Mechanism

The UI Extension's working process is divided into 3 phases:

- Setting up [99]
- Loading [99]
- Activating [100]

Setting up

At first, you must add Dependencies to *pom.xml*. In this phase, you are going to install all elements of UI Extension framework in the *configuration.xml* file:

- *UIExtensionManager* is implemented by *UIExtensionManagerImpl*.
- Plug *UIExtensionPlugin* in *UIExtensionManager* by using the `registerUIExtensionPlugin()` method.
- List all the UI Extension's definitions. You can also define your own external filter (optional).
- Create the parent UI Component class.
- Create the UI Extension class.
- Create the internal filters.

Loading

UIExtensionPlugin is responsible for looking up all UI Extension definitions, thus you can use it to obtain all UI Extensions, then plug it into *UIExtensionManager*. At present, all UI Extensions in your project will be managed by *UIExtensionManager*. Now you can get UI Extensions everywhere by invoking the `getUIExtensions(String objectType)` method.

In the UI Component class, implement a function which:

- Retrieve a collection of UI Extensions which belongs to it by *UIExtensionManager*:

```
List<UIExtension> extensions = manager.getUIExtensions("org.exoplatform.wiki.UIPageToolBar");
```

- Transform them into *UIComponent* and add them to the parent UI Component:

```
// You are free to create a context
Map<String, Object> context = new HashMap<String, Object>();
context.put(key,Obj);
// UIExtensionManager will depend on this context and extension to add or does not add extension to UI Component(this)
UIComponent component = manager.addUIExtension(extension, context, this);
```

The `addUIExtension()` method is responsible for adding extensions to an UI Component. It depends on:

- UIExtension, in particular, the UIExtension's filter. Either internal filter or external filter has the `accept` method, thus the adding process will be successful if `accept` returns 'true' and vice versa.
- Context will be the parameter of the `accept` method.

Activating

The final step is to present UI Extension in a template.

As all UI Extensions are presently becoming children of UI Component, you can implement UI Component's action thanks to UI Extension's action. For example:

```
<%for(entry in uicomponent.getActions()) {
  String action = entry.Id();
  def uiComponent = entry;
  String link = uiComponent.event(action);
%>
  <a href="$link" class="$action" title="$action" %>"><%= action %></a>
<%}%>
```

Note

You are free to customize your action's Stylesheet.

System Integration

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

- **Authentication**

  The information which is related to the authentication, including:

  - **Single-Sign-On (SSO) [101]**
  - **Central Authentication Service (CAS) [101]**
  - **Kerberos SSO on Active Directory [101]**
- **Users integration**

  The information which is related to the users integration, including:

  - **Organization Service [102]**
  - **Memberships, Groups and Users [102]**
  - **Organization API [104]**
- **LDAP Integration**

  Explanations of numerous parameters of the eXo LDAP organization service, including:

  - **Connection Settings**
  - **Organization Service Configuration**
  - **Active Directory sample configuration**
  - **Picketlink IDM**
- **Email**

  Ways to configure the email service.

5.1. Authentication

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 SSO - Single Sign On.

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, see here.

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.

See also

- Users integration
- LDAP Integration
- Email

5.2. Users integration

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.

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

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 five components. For more details, see the Organization API section.

See also

- Authentication
- LDAP Integration
- Email

5.3. LDAP Integration

- Connection Settings

  Description of parameters used in the connection settings.
- Organization Service Configuration

  Required information for configuring Organization Service that shows how the directory is structured and how to interact with it.
- Active Directory sample configuration

  Introduction to an alternative configuration for active directory, and steps to use the LDAPs protocol with the Active Directory.
- Picketlink IDM

  Information about the PicketLink IDM component which eXo Platform currently uses for keeping the necessary identity information, such as users, groups, memberships.

Note

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 which contains numerous parameters.

See also

- Authentication
- Users integration
- Email

5.3.1. 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 PROVIDER URL). 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 SECURITY PRINCIPAL).
- **password**: password for user *rootdn* (see SECURITY CREDENTIALS).
- **authenticationType**: type of authentication to be used (see SECURITY AUTHENTICATION). 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.

5.3.2. 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 the *ldap-configuration.xml* file (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.

5.3.2.1. Users

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
  ...
```

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
  ...
```

5.3.2.2. 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
  ...
```

5.3.2.3. 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
```

5.3.2.4. 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. For example, in the following membership entry:

```
cn=manager,ou=human-resources,ou=groups,ou=portal,dc=exoplatform,dc=org
```

The *cn* attribute is used to designate the *manager* membership type. In other words, the name of role is given by the 'cn' attribute.

- **membershipTypeObjectClassFilter**: Filter is used to distinguish membership entries under groups. You can use the more complex filters. For example, the following is a filter used for a customer that needs to trigger a dynlist overlay on OpenLDAP.

```
(&amp;amp;(objectClass=ExoMembership)(membershipURL=&#42;))
```

Note

Please pay attention to the xml escaping of the '&' (and) operator.

5.3.2.5. User profiles

eXo User profiles also have entries in the LDAP but the actual storage is still done with the hibernate service. You will need the following parameters:

```
<field name="profileURL">
  <string>ou=profiles,ou=portal,dc=exoplatform,dc=org</string>
</field>
<field name="profileLDAPClasses">
  <string>top,organizationalPerson</string>
</field>
```

- **profileURL**: base dn to store user profiles.
- **profileLDAPClasses**: Classes used for creating user profiles.

5.3.3. Active Directory sample configuration

Here is an alternative configuration for active directory that you can find in **activedirectory-configuration.xml**.

```
[...]
<component>
  <key>org.exoplatform.services.ldap.LDAPService</key>
  [..]
  <object type="org.exoplatform.services.ldap.impl.LDAPConnectionConfig">
  <!-- for multiple ldap servers, use comma separated list of host:port (Ex. ldap://127.0.0.1:389,10.0.0.1:389) -->
  <!-- whether or not to enable ssl, if ssl is used ensure that the javax.net.ssl.keyStore & java.net.ssl.keyStorePassword properties are set -->
  <!-- exo portal default installed javax.net.ssl.trustStore with file is java.home/lib/security/cacerts-->
```

```
<!-- ldap service will check protocol, if protocol is ldaps, ssl is enable (Ex. for enable ssl: ldaps://10.0.0.3:636 ;for disable ssl: ldap://10.0.0.3:389 ) -->
<!-- when enable ssl, ensure server name is *.directory and port (Ex. active.directory) -->
  <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>site</string></field>
  <field name="version"><string>3</string></field>
  <field name="referralMode"><string>ignore</string></field>
  <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"

5.3.4. 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 -->
```

5.4. Email

The email service can use any SMTP account configured in the *configuration.properties* file.

File location

File	Tomcat	JBoss
configuration.properties	*$PLATFOORM_TOMCAT_HOME/gatein/conf/configuration.properties*	*$PLATFORM_JBOSS_HOME/server/default/conf/gatein/configuration.properties*

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.

See also

- Authentication
- Users integration
- LDAP Integration

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**

  Introduction to the different levels of APIs, Provisional or Experimental APIs.
- **Platform APIs**

  Definitions of levels of Platform APIs, including:
  - **Java APIs [116]**
  - **JavaScript APIs [117]**
  - **Web Services [117]**
- **Provisional APIs**

  A list of provisional APIs.

6.1. 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.Z1) is guaranteed. If there is any change between X.Y and X.Y1, the eXo Support team will help by upgrading the code.

Compatibility X.Y(1): The compatibility between minor versions (X.Y and X.Y1) is guaranteed. If there is any change between X and X1, the eXo Support team will help by upgrading the code.

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

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.

See also

- Platform APIs
- Provisional APIs

6.2. Platform APIs

This section summarizes a list of eXo Platform APIs which can be categoried into:

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:** This API is used to configure your portal.
- **TaxonomyService:** An API that allows you to organize your content.
- **LinkManager:** An API that provides a way to manage links when developing WCM features.
- **PublicationManager:** An API that provides different ways to manage the publication of content when developing WCM features.
- **WCMComposer:** An API to get content shown in the website. The cache management is used in this service, and methods to update the content cache.
- **NewFolksonomy:** An API to manage all the tags and their styles. Currently, it just supports adding/editing/removing the **Private** & **Public** tags.
- **ApplicationTemplateManager:** An API to manage dynamic groovy templates for WCM-based products.
- **NodeFinder:** An API to find a node with a given path.

- **JodConverter:** An API to convert documents into different office formats.
- **SiteSearchService:** An API that allows users to find all information matching with their given keyword.
- **SEOService:** An API that manages SEO data of a page or a content.
- **TimelineService:** An API that allows documents to be displayed by days, months and years.
- **XML Configuration:** A set of DTD for configuring eXo Platform.
- **ActivityManager:** An API to manage activities (create/delete/like activity; delete/like/create comment; get activities of space/user/connection).
- **IdentityManager:** An API to manage identities (create identity, save profile, get connection of an identity).
- **RelationshipManager:** An API which is used to work with connections between 2 identities, interact between identities, get list access to get list of connections, incomings, outgoings.
- **SpaceService:** An API that provides methods for working with space (create new space, delete, get space by url, groupid, display name, pretty name).

JavaScript APIs

- **OpenSocial 1.1 Gadget Specification:** A standard that defines how to write gadgets and provide APIs. Gadgets are particularly useful for integrating external applications into eXo Platform.

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 1.1 REST Protocol:** A standard API for accessing the social graph and activity streams.
- **WebDAV:** A standard protocol for exchanging document over HTTP.

See also

- Definitions of API Levels
- Provisional APIs

6.3. Provisional APIs

Java APIs

- **UI Extensions:** An API to plug new actions into the eXo Platform UI.
- **ECM Admin and DocumentExplorer Toolbar**
- **Activity Sharing**

See also

- Definitions of API Levels
- Platform APIs

Cookbook

This chapter currently focuses on how to copy a work done on the eXo Platform server, such as creating navigations, node types and templates, to another eXo Platform servers throughout these topics:

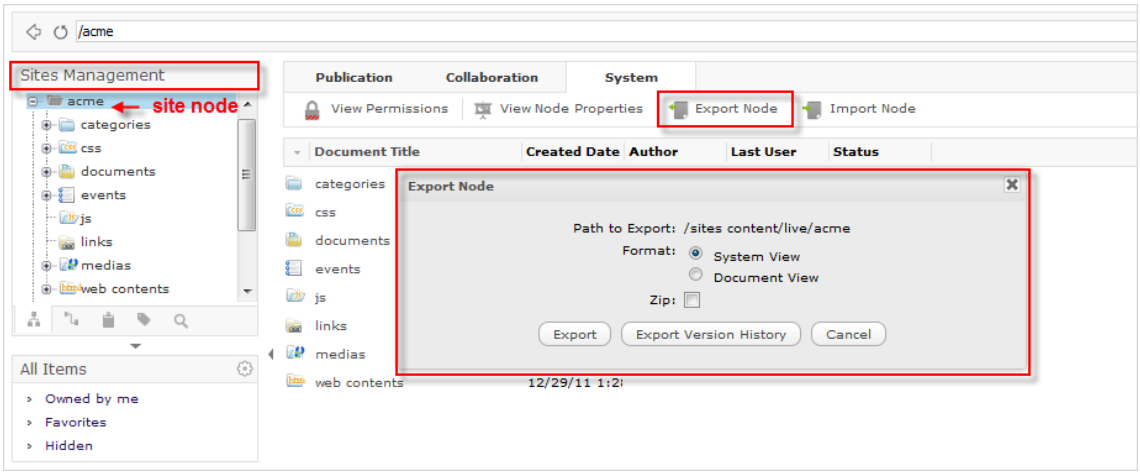
The procedure of each step will be detailed as follows:

Step 1. Copy a site's content folder with its version history.

1. Go to the **Sites Management** drive.

2. Open the site node, for example "acme".

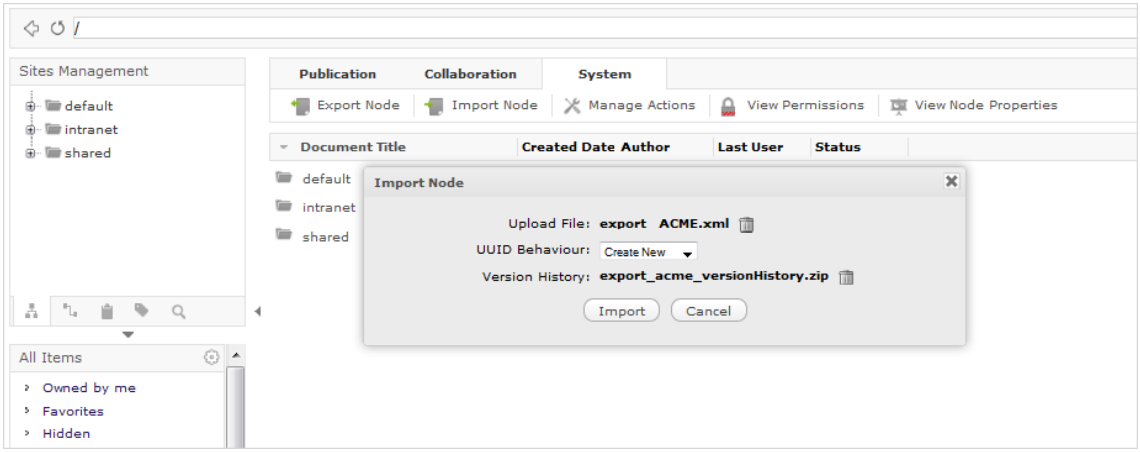
3. Click **Export Node** to export the node with its version history as below:



4. Select **Export** and **Export Version History** to perform the exporting.

5. Navigate to the node where you want to import the file, then click **Import Node** to open the **Import Node** form.

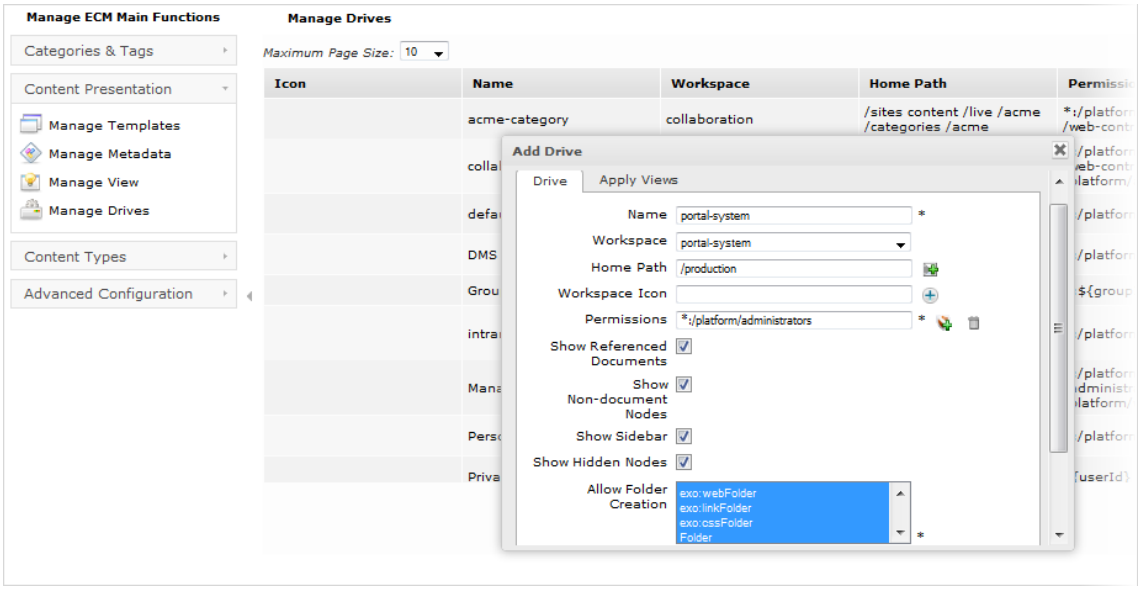
6. Select the exported nodes and version history to be imported.



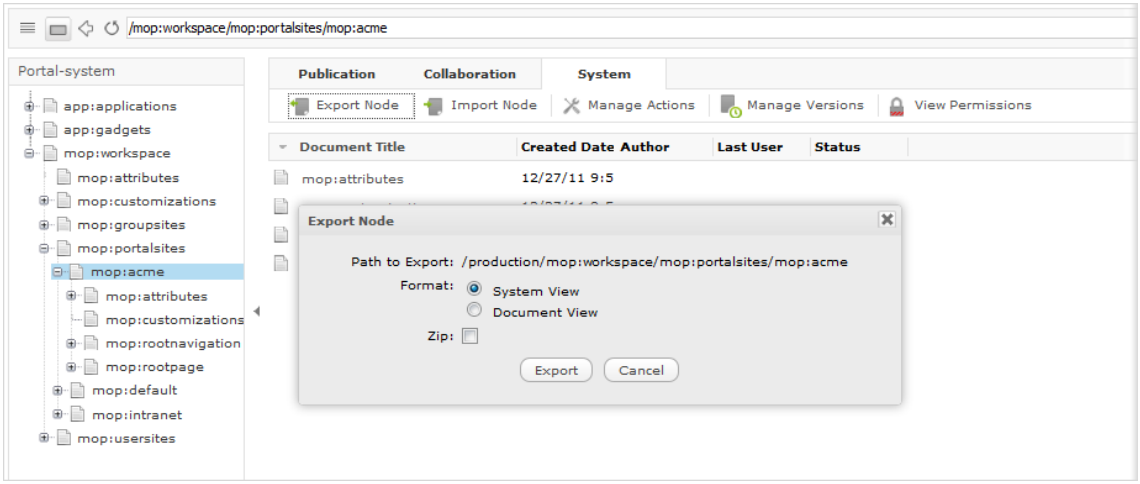
One pop-up message will appear to inform that you have imported successfully.

Step 2. Copy navigation nodes of sites.

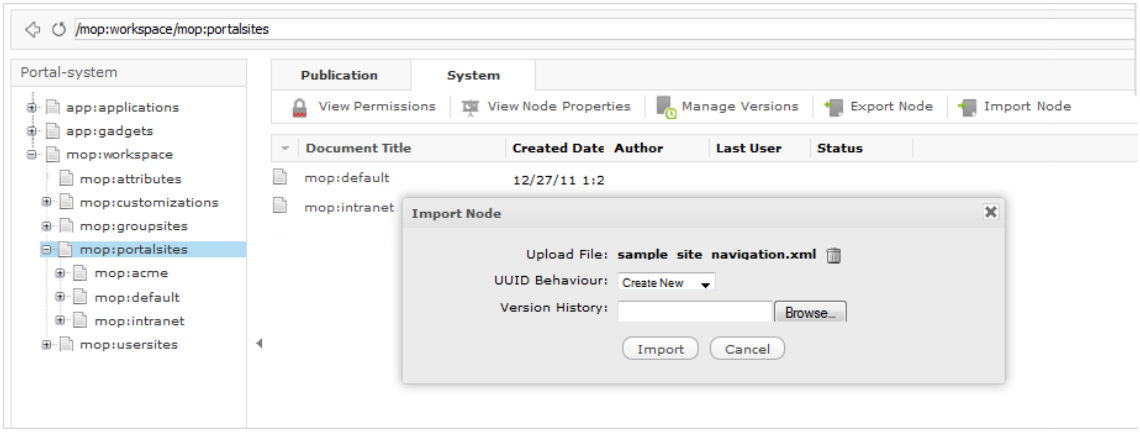
1. Go to the **Content Administration** page and add a new drive to both target and source servers.



2. Export the navigation node.

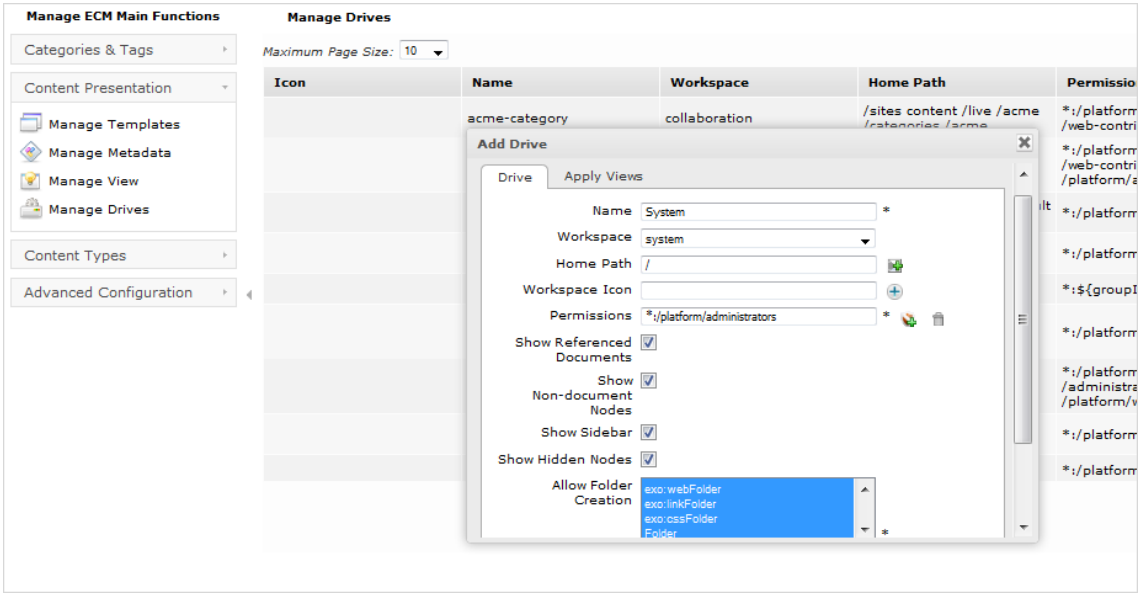


3. Import the nodes navigation.

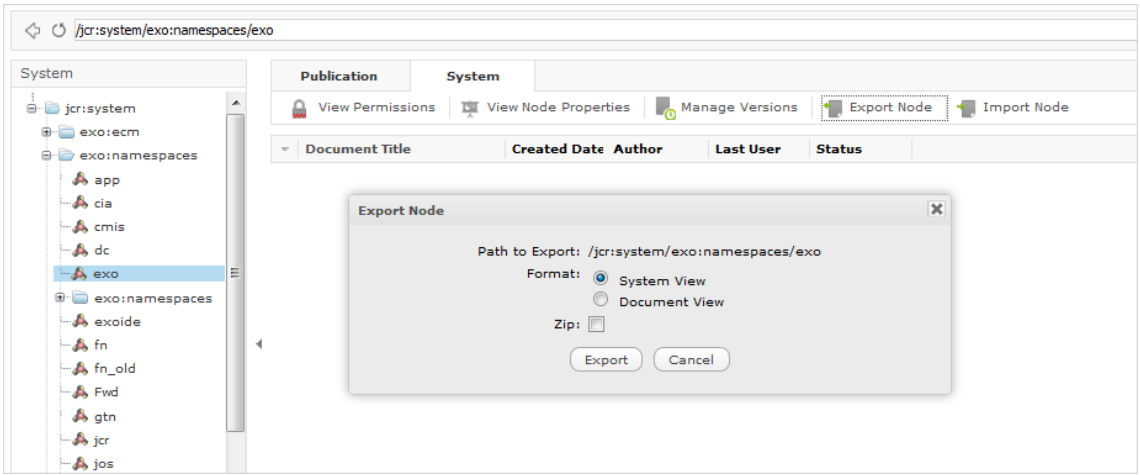


Step 3. Copy templates of node types.

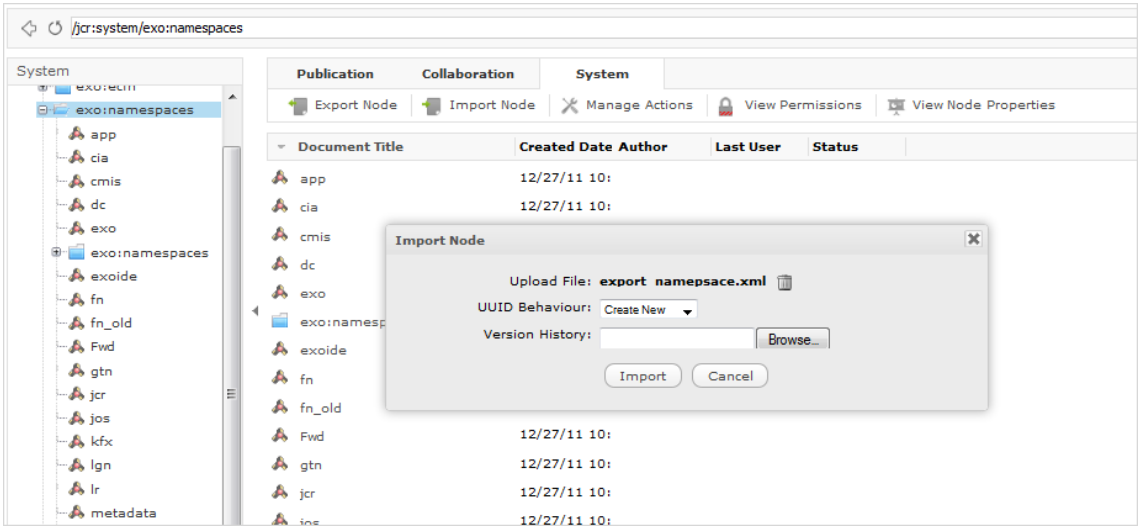
1. Add the **System** drive to both servers.



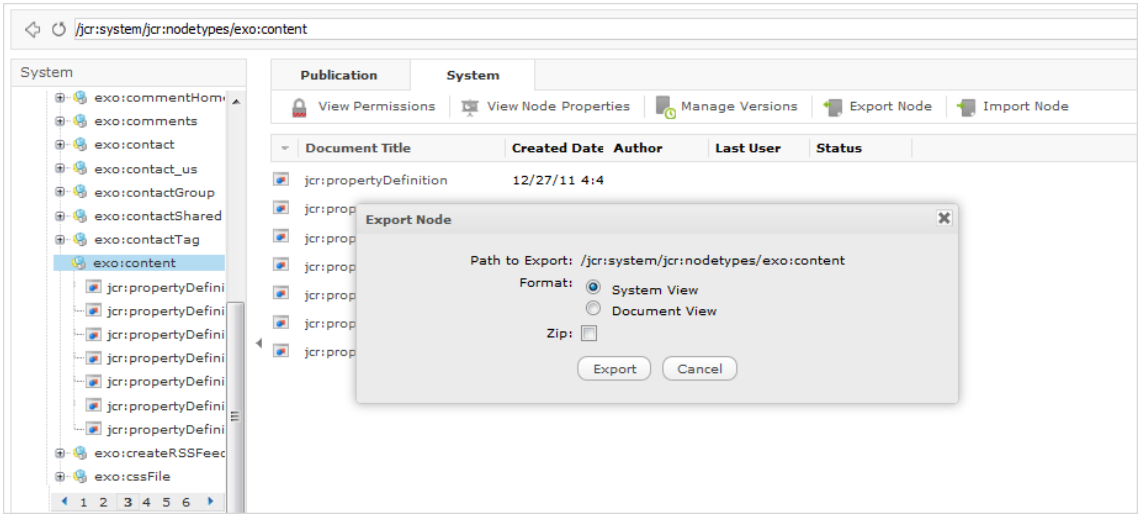
2. Open *system:/jcr:system/exo:namespaces/{namespace_name}*, and export it.



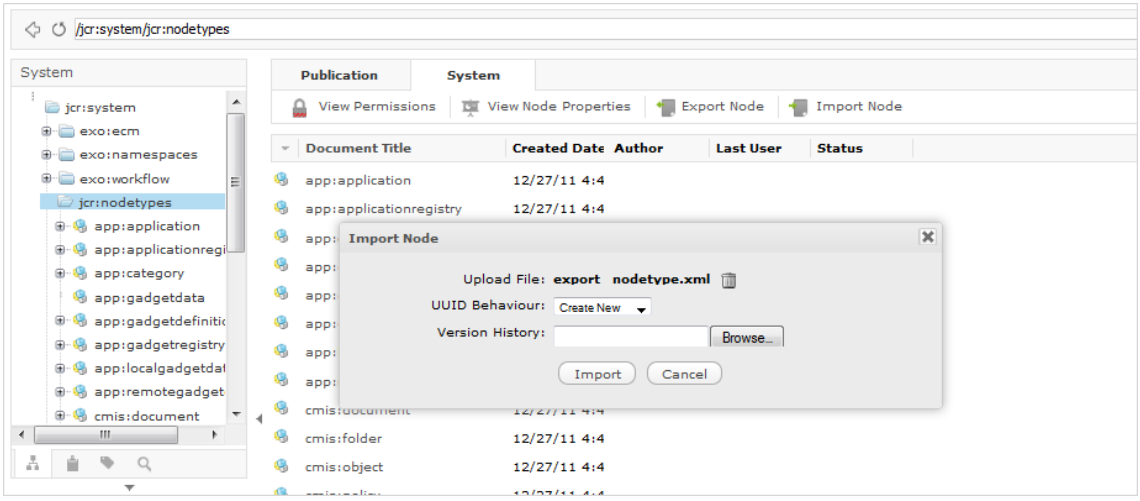
3. Open *system:/jcr:system/exo:namespaces/*, and import the exported file as described in **Step 2** [120].



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** [123].



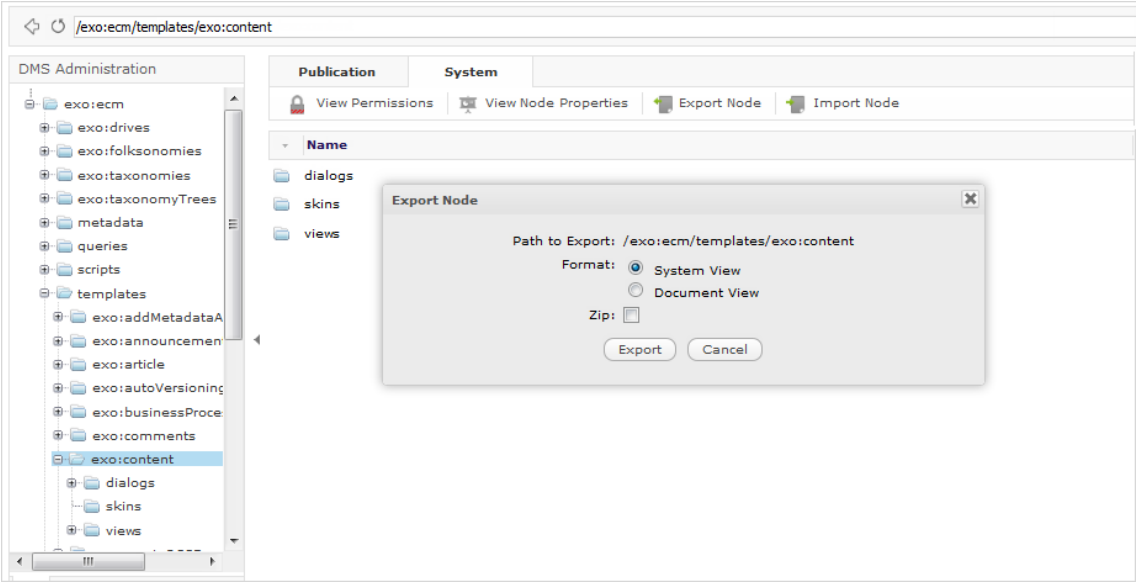
Note

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

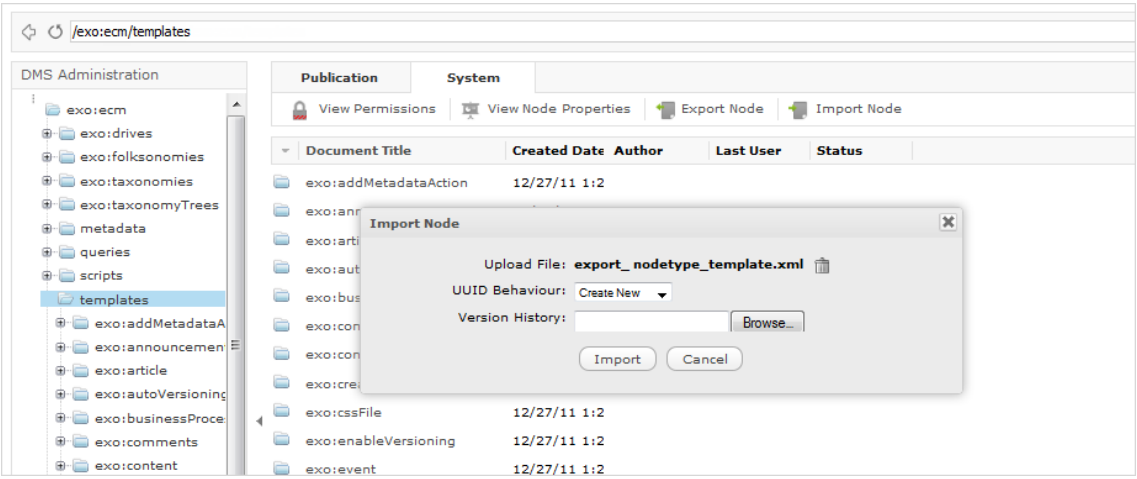
Step 4. Copy the WCM template.

1. Open the **DMS Administration** drive.

2. Open *dms-system:/exo:ecm/templates/{node_type}*, and export it.



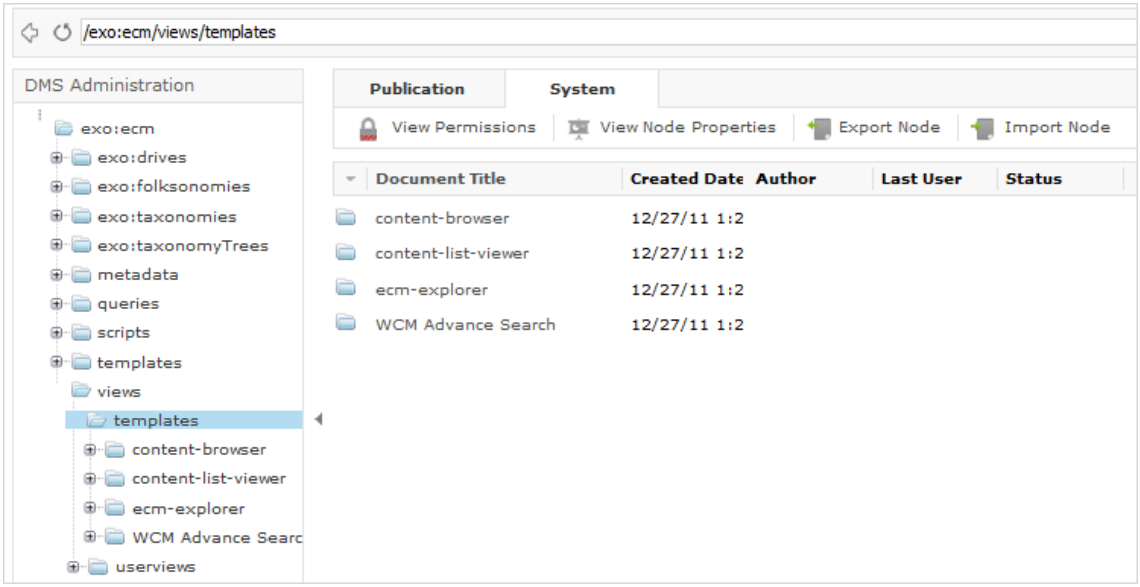
3. Open *dms-system:/exo:ecm/templates/*, and import the exported file.



Also, for the 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.

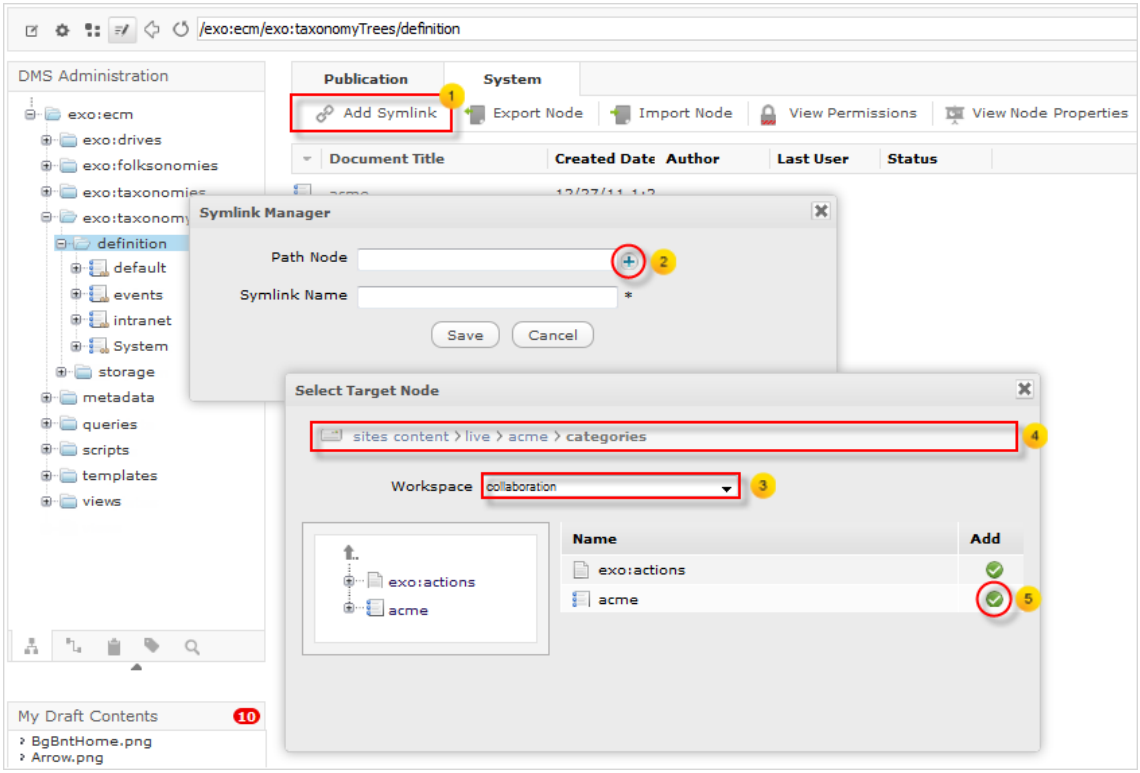
Step 5. Copy a Taxonomy tree.

By importing the whole site as described in the **Copy a site's content folder with its version history** [119] section, you will also have the Taxonomy tree imported. The default location where the site's Taxonomy is placed in a sub-folder is 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 need to do is to add this Taxonomy tree definition by following these steps:

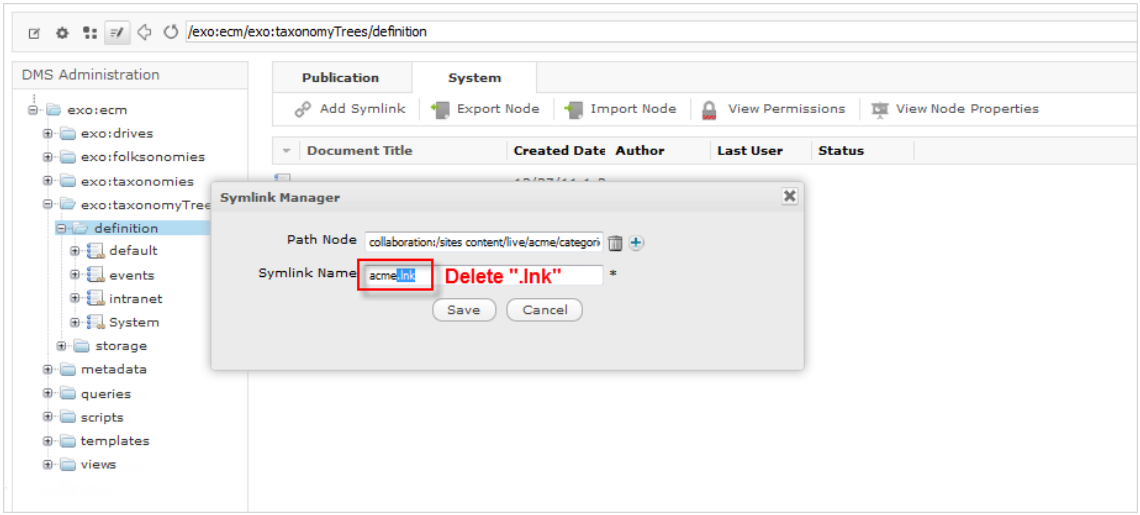
1. Open the **DMS Administration** drive in the new server.

2. Go to *dms-system:/exo:ecm/exo:taxonomyTrees/definition/*.

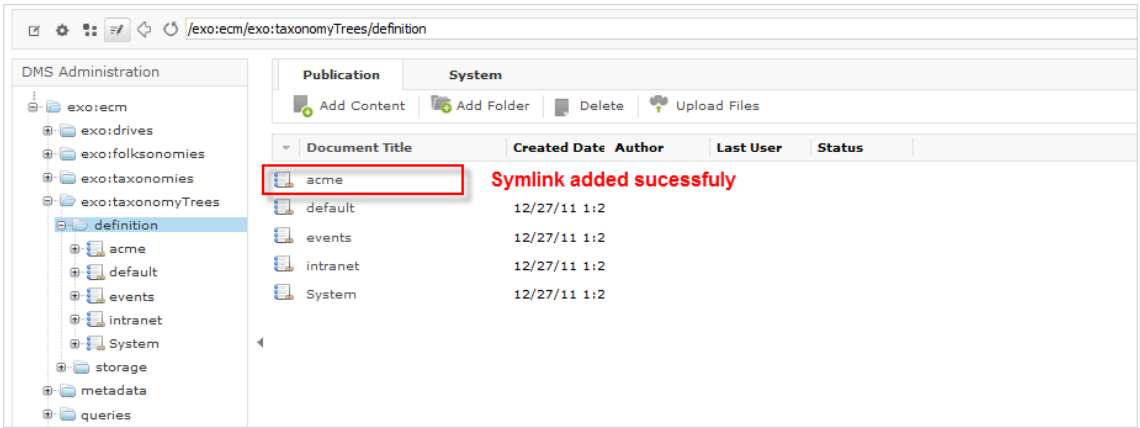
3. Add a symlink to the **Taxonomy Tree Root Node**, for example *collaboration:/sites content/live/acme/categories/acme*.



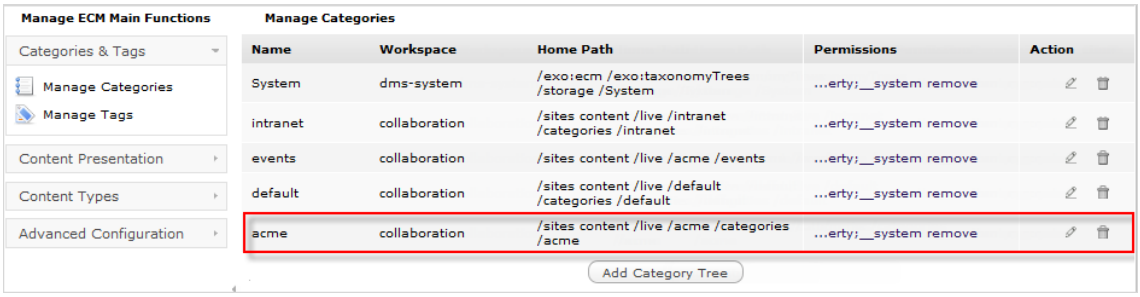
The name of symlink is displayed as "acme".



The symlink will be generated as below:



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



Step 6. Copy metadata templates.

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.

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.

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 select box in a dynamic way.

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

Step 9. Copy drive configurations.

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.

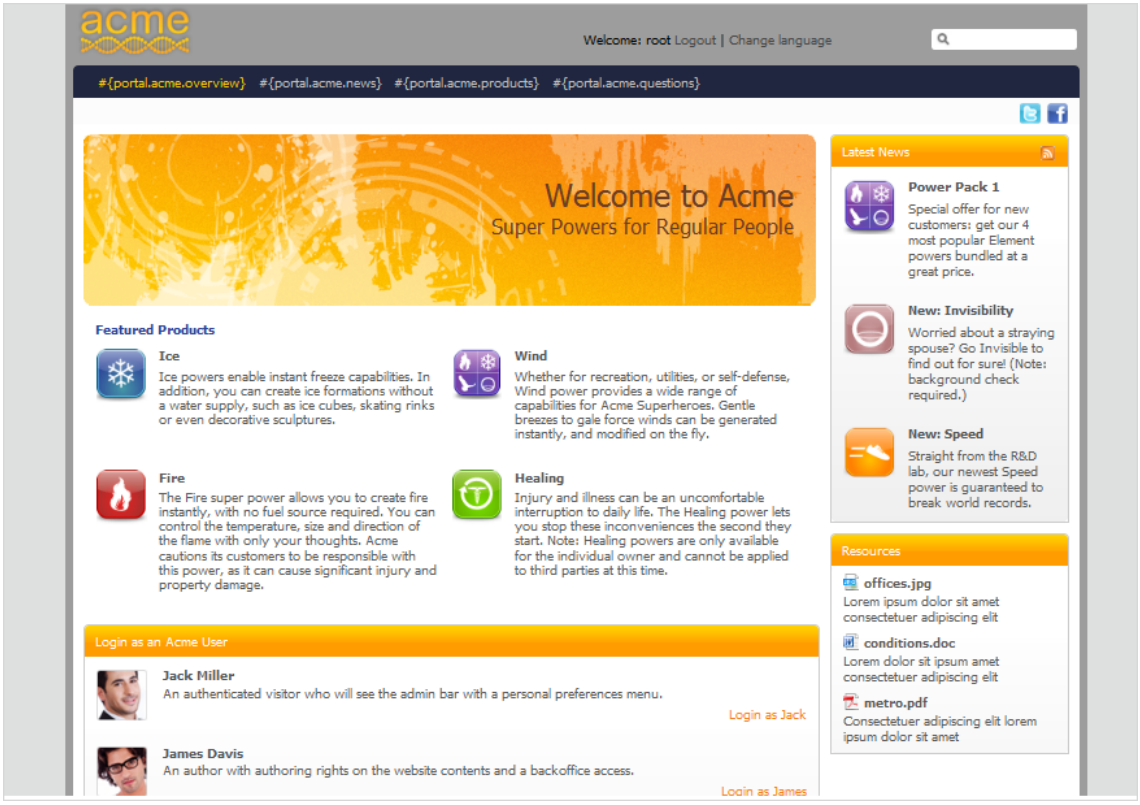
1. Open the drive that points into the **Portal-System** Workspace.

2. Go to your gadget by following the *portal-system:/production/app:gadgets/{gadget_name}* path.

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 may look quite broken, so you need to restart the server first. After the server is restarted, the site will look like:


acme
Welcome: root Logout | Change language
#{portal.acme.overview}
#{portal.acme.news}
#{portal.acme.products}
#{portal.acme.questions}
Welcome to Acme
Super Powers for Regular People
Featured Products
Ice
Ice powers enable instant freeze capabilities. In addition, you can create ice formations without a water supply, such as ice cubes, skating rinks or even decorative sculptures.
Wind
Whether for recreation, utilities, or self-defense, Wind power provides a wide range of capabilities for Acme Superheroes. Gentle breezes to gale force winds can be generated instantly, and modified on the fly.
Fire
The Fire super power allows you to create fire instantly, with no fuel source required. You can control the temperature, size and direction of the flame with only your thoughts. Acme cautions its customers to be responsible with this power, as it can cause significant injury and property damage.
Healing
Injury and illness can be an uncomfortable interruption to daily life. The Healing power lets you stop these inconveniences the second they start. Note: Healing powers are only available for the individual owner and cannot be applied to third parties at this time.
Login as an Acme User
Jack Miller
An authenticated visitor who will see the admin bar with a personal preferences menu.
Login as Jack
James Davis
An author with authoring rights on the website contents and a backoffice access.
Login as James
Latest News
Power Pack 1
Special offer for new customers: get our 4 most popular Element powers bundled at a great price.
New: Invisibility
Worried about a straying spouse? Go Invisible to find out for sure! (Note: background check required.)
New: Speed
Straight from the R&D lab, our newest Speed power is guaranteed to break world records.
Resources
offices.jpg
Lorem ipsum dolor sit amet consectetuer adipiscing elit
conditions.doc
Lorem dolor sit ipsum amet consectetuer adipiscing elit
metro.pdf
Consectetuer adipiscing elit lorem ipsum dolor sit amet

Upgrade eXo Platform

Upgrading eXo Platform from 3.0 to 3.5 is generally a procedure which includes 2 steps:

Step 1. Upgrade the project extensions.

Step 2. Upgrade the project and the product data.

This chapter aims at instructing you how to archive Step 1 [129] via the following topics:

- **Prerequisites**

  Questions that help developers to gather all requirements for their upgraded project, required preparations of an extension project, and steps which need to be adpated in eXo Platform.
- **Update Maven dependencies, configurations and components**

  How to update Maven dependencies, configurations and components.
- **Update extensions**

  How to update your custom eXo Platform extensions, including updating Kernel XML Schema, portal and APIs.

Note

Upgrading data can be strongly dependent on both your deployments and environments, so it is recommended that you contact http://support.exoplatform.com to bootstrap the Step 2 [129].

8.1. Prerequisites

Before upgrading, you first need to understand how to create a project which is based on eXo Platform 3.5. For more details, refer to Create Your Own Portal.

And now, you need to gather all requirements for your upgraded project by clarifying the following questions:

- Do you actually need to migrate your project sources?
- Have you created any new portal containers in your system?
- Have you made any lifetime-related changes on eXo Platform? Would you like to keep these changes?
- Have you developed a custom application? Where have you stored your application data?
- Have you configured a dedicated JCR workspace?
- Which eXo APIs are you using in your code?
- Have you overridden any files of eXo Platform?
- Have you disabled any services of eXo Platform?
- Which Kernel configurations have you customized?

Prepare for your extension project

You need to create a new branch for eXo Platform 3.5 extension project, then clone your source code of eXo Platform 3.0 with that of eXo Platform 3.5.

The way to package your project customization called extensions has been introduced in eXo Platform 3.0. Therefore, by using the WAR extension configuration approach, you can overload eXo Platform's default files defined in WARs, such as *gtmpl, favicon, xml, jsp*. For more details on how to create your extension project, refer to the Create Your Own Portal section.

Because there are not any changes on the eXo Platform 3.5 extension mechanism, your extensions should be mostly compatible. However, if your extension was overriding either of the eXo Platform 3.0 built-in files, you should check that the file has not been changed in eXo Platform 3.5 and update it if any.

What needs to be adapted in eXo Platform extension

There are 4 main steps which need to be defined for updating eXo Platform extension from 3.0 to 3.5:

- Update your project Maven dependencies in the *POM* files.
- Update your eXo Platform 3.5 configurations.
- Update components in your eXo Platform extension to use components of Platform 3.5
- Update your eXo Platform extension which are customised features/components (were added new/ modified/ delete in eXo Platform 3.0 extension), and make sure they can work well in eXo Platform 3.5 environment.

See also

- Update Maven dependencies, configurations and components
- Update extensions

8.2. Update Maven dependencies, configurations and components

After gathering all requirements, and preparing for your extension project and adaptions in eXo Platform extension, you then need to update:

- Maven dependencies [130]
- Configurations [132]
- Components [132]

Maven dependencies

eXo Platform includes many components, so if you want to update your eXo Platform extension to use eXo Platform 3.5, you need to update the version of all dependencies in the *POM* files, and select the version of dependencies which are compatible with eXo Platform 3.5. For example, eXo Platform 3.5.1 needs to have the following main dependencies:

```
<!-- kernel -->
<dependency>
    <groupId>org.exoplatform.kernel</groupId>
    <artifactId>kernel-parent</artifactId>
    <version>2.3.5-GA</version>
</dependency>
<!-- core -->
<dependency>
  <groupId>org.exoplatform.core</groupId>
  <artifactId>core-parent</artifactId>
  <version>2.4.5-GA</version>
</dependency>
<!-- ws -->
<dependency>
  <groupId>org.exoplatform.ws</groupId>
  <artifactId>ws-parent</artifactId>
  <version>2.2.5-GA</version>
</dependency>

<!-- jcr -->
```

```
<dependency>
  <groupId>org.exoplatform.jcr</groupId>
  <artifactId>jcr-parent</artifactId>
  <version>1.14.5-GA</version>
</dependency>

<!-- jcr service -->
<dependency>
  <groupId>org.exoplatform</groupId>
  <artifactId>exo-jcr-services</artifactId>
  <version>1.14.5-GA</version>
</dependency>

<!-- commons -->
<dependency>
  <groupId>org.exoplatform.commons</groupId>
  <artifactId>exo.platform.commons</artifactId>
  <version>1.1.5</version>
</dependency>

<!-- exogtn -->
<dependency>
   <groupId>org.exoplatform.portal</groupId>
   <artifactId>exo.portal.parent</artifactId>
   <version>3.2.2-PLF</version>
</dependency>

<!-- webos -->
<dependency>
  <groupId>org.exoplatform.webos</groupId>
  <artifactId>webos-parent</artifactId>
  <version>2.0.2</version>
</dependency>

<!-- ecms -->
<dependency>
  <groupId>org.exoplatform.ecms</groupId>
  <artifactId>exo-ecms</artifactId>
  <version>2.3.5</version>
</dependency>

<!-- ide -->
<dependency>
   <groupId>org.exoplatform.ide</groupId>
   <artifactId>exo-ide-parent</artifactId>
   <version>1.1.4</version>
</dependency>

<!-- cs -->
<dependency>
  <groupId>org.exoplatform.cs</groupId>
  <artifactId>cs</artifactId>
  <version>2.2.7</version>
</dependency>

<!-- ks -->
<dependency>
  <groupId>org.exoplatform.ks</groupId>
  <artifactId>ks</artifactId>
  <version>2.2.7</version>
</dependency>

<!-- social -->
<dependency>
  <groupId>org.exoplatform.social</groupId>
  <artifactId>social-project</artifactId>
  <version>1.2.7</version>
</dependency>

<!-- integration -->
<dependency>
  <groupId>org.exoplatform.integration</groupId>
  <artifactId>integ</artifactId>
  <version>1.0.5</version>
```

```
</dependency>
```

Configurations

After updating Maven dependencies, you need to update configurations via answering 2 questions:

- In your eXo Platform extension, did you override anything in *$PLATFORM_TOMCAT_HOME/gatein/conf*?
  - If NO, you can skip this step, and simply keep all configurations in *$PLATFORM_TOMCAT_HOME/gatein/conf*.
  - If YES, you first need to keep all configurations of eXo Platform 3.5, then define which configurations you want to add/remove/update. After that, you can put them into *$PLATFORM_TOMCAT_HOME/gatein/conf*.

Note

There are some new configurations in *$PLATFORM_TOMCAT_HOME/gatein/conf/* which are related to how to deploy the eXo Platform clustering, and JBoss Cache configuration.

- In your eXo Platform extension, did you override anything in *$PLATFORM_TOMCAT_HOME/bin*?
  - If NO, you can skip this step, and simply keep all configurations in *$PLATFORM_TOMCAT_HOME/bin*.
  - If YES, you first need to keep all eXo Platform 3.5's origin configurations, then add/remove/update your extension configuration into it.

Note

$PLATFORM-TOMCAT-HOME/bin/gatein.sh and *$PLATFORM_TOMCAT_HOME/bin/gatein.bat* are no longer used. Therefore, you should add your customized configurations in a more official way to set environment variables via *setenv.sh* or *setenv.bat*.

Components

There are many components in eXo Platform, so if you update your eXo Platform extension to eXo Platform 3.5, you need to update the version of all eXo Platform 3.5's components. To do so, simply configure the Maven POM file that allows you to use the artifacts (eXo Platform 3.5 and its components). For more details, see the Update project Maven dependencies [130] section. All eXo Platform 3.5's components will be automatically included in your eXo Platform extension.

There are some new components in eXo Platform 3.5 that have not been existing in eXo Platform 3.0, including Webos, Wiki, and Gadget-pack. To use these components, simply activate their profiles. For more details, see the Profiles of eXo Platform section.

In eXo Platform 3.5, the "intranet" portal is defined in the *acme-intranet.war* file (not in the *office-portal.war* file of eXo Platform 3.0). So if you have overridden anything in the "intranet" portal, you need to synchronize your extension with the *acme-intranet.war* file of eXo Platform-3.5.

See also

- Prerequisites
- Update extensions

8.3. Update extensions

The next procedure of upgrading eXo Platform is to update your custom eXo Platform extensions, including:

- Kernel XML Schema [133]
- Portal [133]

- APIs [134]

Kernel XML Schema

- Simply upgrade the version from *kernel_1_0.xsd* to *kernel_1_2.xsd* in all *.xml* configurations files.

```
<configuration xmlns="http://www.exoplatform.org/xml/ns/kernel_1_2.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.exoplatform.org/xml/ns/kernel_1_2.xsd http://www.exoplatform.org/xml/ns/kernel_1_2.xsd">
   .....
</configuration>
```

Portal

- **Portal data import**

Note

It is recommended you have the knowledge of the import mechanism by referring to the Data Import Strategy section.

Processing the portal data import includes the following tasks:

- Improve the current data import strategy so that it becomes more flexible in various scenarios.
- Deploy an extension after you have initialized the portal that allows the extension to contribute its entities to an existing site.
- Reimport data by overwriting an existing entire configuration at each startup.

What's new?

Previous behavior	New behavior
The previous mode for navigations is defined using the overridden configuration. • **Merge**: Merge data from the XML descriptor to the existing data. • **Override**: Destroy any existing data, and create new ones. The previous mode has two main issues: • The behavior for instance disables the "always merge data" usecase. It means that it is always impossible to merge new navigations nodes from XML to MOP. • The behavior is configured by the overridden parameter.	The new behavior which is configured at the import level with a field parameter would have the following values: • **conserve**: Import data. However, if the navigation has been existing, it will be kept. • **insert**: Insert the missing descriptor data by adding new nodes without any modifications on the existing nodes. • **merge**: Merge the descriptor data, add missing nodes and update the nodes of the same name. • **overwrite**: Always destroy the previous data and create new ones.

For example, if you want to merge the new portal navigation from the configuration file and the existing portal navigation into the current eXo Platform, you can configure as below:

```
<object-param>
  <name>portal.configuration</name>
  <object type="org.exoplatform.portal.config.NewPortalConfig">
    <field name="predefinedOwner">
      <collection type="java.util.HashSet">
```

```
      <value><string>classic</string></value>
    </collection>
   </field>
   <field name="ownerType">
    <string>portal</string>
   </field>
   <field name="templateLocation">
    <string>war:/conf/portal</string>
   </field>
   <field name="importMode">
    <string>merge</string>
   </field>
  </object>
</object-param>
```

Note

When a mode is not specified, the default mode will be **conserve**.

- **Authentication**: eXo Platform 3.5 uses a new authentication mechanism so-called the WCI authentication mechanism. This change from GateIn into the WCI module is because:
  - Authentication is commonly part of Java EE and happens at the Servlet Container layer, while WCI is the layer of integration between our stack and the Servlet Container.
  - WCI provides an easier integration of SSO providers that allows the SSO module to depend on the WCI module instead of integrating directly with the GateIn module. Its main benefit is to improve the quality of SSO and to decouple SSO from GateIn.
  - The GateIn stack wants to support and leverage specifications of both Servlet 3.0 and Servlet 2.5 that are not supported by the GateIn module. Meanwhile, WCI can serve both.
  - The Servlet 3.0 provides a programmatic login feature which can be used to implement WCI.
  - WCI provides the stronger guaranty of the authentication quality, meanwhile the authentication of GateIn was not unit tested. By moving the GateIn to WCI module, the test can be made against several implementations.

If there are other new portal containers in your eXo Platform extension, you need to update the authentication mechanism in the *$PLATFORM-TOMCAT-HOME/conf/jaas.conf* file. For example, in case you have added a new portal container named "company" to eXo Platform, you need to configure the new authentication module for the new portal container as follows:

```
<!--this configuration for defaul portal container with name "portal"-->
gatein-domain {
  org.gatein.wci.security.WCILoginModule optional;
  org.exoplatform.services.security.jaas.SharedStateLoginModule required;
  org.exoplatform.services.security.j2ee.TomcatLoginModule required;
};

<!--this configuration for new portal container with name "company"-->
gatein-domain-company {
  org.gatein.wci.security.WCILoginModule optional
    portalContainerName="company"
   realmName="gatein-domain-company";
  org.exoplatform.services.security.jaas.SharedStateLoginModule required
    portalContainerName="company"
   realmName="gatein-domain-company";
  org.exoplatform.services.security.j2ee.TomcatLoginModule required
    portalContainerName="company"
   realmName="gatein-domain-company";
};
```

For more details, see the Change the JAAS realm section.

APIs

eXo Platform 3.5 has many changes on the data model and APIs. Therefore, if there are any new customized components in your eXo Platform extension that use eXo Platform 3.0 APIs, you need to check if they can work well in eXo Platform 3.5 or not. For more details, see the eXo Platform 3.5 APIs section.

Finally, you need to update your customized components if needed.

See also

- Prerequisites
- Update Maven dependencies, configurations and components