Reference Guide

Exported from JBoss Community Documentation Editor at 2013-06-17 00:27:31 EDT

Table of Contents

GateIn Portal 3.5 Reference Guide

1 Configuration

1.1 Database Configuration

GateIn Portal3.5 has two different database dependencies. One is the identity service configuration, which depends on Hibernate. The other is Java Content Repository (JCR) service, which depends on JDBC API, and can integrate with any existing datasource implementation.

When you change the database configuration for the first time, GateIn Portal will automatically generate the proper schema (assuming that the database user has the appropriate permissions).

GateIn Portal assumes the default encoding for your database is `latin1`. You may need to change this parameter for your database in order for GateIn Portal3.5 to work properly.

1.1.1 Configuring the databases on JBoss AS7

On JBoss the configuration of JCR, and IDM services is set to use a container provided datasources, configured through AS7 configuration.

See https://docs.jboss.org/author/display/AS7/DataSource+configuration for instructions on how to configure a DataSource on AS7.

Datasource configuration can be found in:
`$JBOSS_HOME/standalone/configuration/standalone.xml`

Configuring the database for JCR

JCR uses the datasource bound to JNDI under: `java:/jdbcjcr_portal`.

```
<datasource jndi-name="java:/jdbcjcr_portal" pool-name="JCRPortalDS" enabled="true"
use-java-context="true">


<connection-url>jdbc:h2:file:${jboss.server.data.dir}/gatein/portal/jdbcjcr_portal;DB_CLOSE_DELAY=
<driver>h2</driver>

    <security>
        <user-name>sa</user-name>
        <password>sa</password>
    </security>

</datasource>
```

If additional portals are deployed, additional datasources have to be configured and bound in JNDI under `java:/jdbcjcr_PORTAL-NAME` for each additional portal deployed.

Make sure the user has rights to create tables on jdbcjcr_portal, and to update them as they will automatically be created during the first startup.

Configuring the database for the default identity store

IDM datasource configuration is analogous to JCR datasource configuration.

IDM uses the datasource bound to JNDI under: `java:/jdbcidm_portal`.

```
<datasource jndi-name="java:/jdbcidm_portal" pool-name="IDMPortalDS" enabled="true"
use-java-context="true">

<connection-url>jdbc:h2:file:${jboss.server.data.dir}/gatein/portal/jdbcidm_portal;DB_CLOSE_DELAY=
<driver>h2</driver>
    <security>
        <user-name>sa</user-name>
        <password>sa</password>
    </security>
</datasource>
```

1.1.2 Configuring the databases on Tomcat

On Tomcat the configuration of JCR, and IDM services is set to use application provided datasources. It means that GateIn application itself configures and creates the datasources.

Configuring the database for JCR

To configure the database used by JCR you will need to edit the following files:

- `$TOMCAT_HOME/gatein/conf/configuration.properties`

And edit the values of driver, url, username and password with the values for your JDBC connection. Refer to your database JDBC driver documentation for more details).

```
gatein.jcr.datasource.driver=org.hsqldb.jdbcDriver
gatein.jcr.datasource.url=jdbc:hsqldb:file:${gatein.db.data.dir}/data/jdbcjcr_${name}
gatein.jcr.datasource.username=sa
gatein.jcr.datasource.password=
```

By default, the name of the database is "jdbcjcr_${name}" - ${name} should be a part of the database name, as it is dynamically replaced by the name of the portal container extension.

In the case of HSQL, the databases are created automatically. For any other database, you will need to create a database named jdbcjcr_portal (and if you have additional portals deployed you'll have to create additional databases named "jdbcjcr_PORTAL-NAME" for each additional portal deployed - note that some databases don't accept '-' in the database name, so you may have to rename your additional portals accordingly).

Make sure the user has rights to create tables on jdbcjcr_portal, and to update them as they will be automatically created during the first startup.

Also, add your database's JDBC driver into the classpath - you can put it in $TOMCAT_HOME/lib.

Let's configure our JCR to store data in MySQL. Let's pretend we have a user named "gateinuser" with a password "gateinpassword". We would create a database "mygateindb_portal" (remember that _portal is required), and assign our user the rights to create tables.

Then we need to add MySQL's JDBC driver to the classpath, and finally edit `$TOMCAT_HOME/gatein/conf/configuration.properties` to contain the following:

MySQL

```
gatein.jcr.datasource.driver=com.mysql.jdbc.Driver
gatein.jcr.datasource.url=jdbc:mysql://localhost:3306/mygateindb${container.name.suffix}
gatein.jcr.datasource.username=gateinuser
gatein.jcr.datasource.password=gateinpassword
```

Configuring the database for the default identity store

By default, users are stored in a database. To change the database in which to store users, you will need to edit the same file as for JCR:

- `$TOMCAT_HOME/gatein/conf/configuration.properties`

Configuration is analogous to the one for JCR datasource, just that it uses different property keys:

```
gatein.idm.datasource.driver=org.hsqldb.jdbcDriver
gatein.idm.datasource.url=jdbc:hsqldb:file:${gatein.db.data.dir}/data/jdbcidm_${name}
gatein.idm.datasource.username=sa
gatein.idm.datasource.password=
```

1.2 Email Service Configuration

GateIn Portal 3.5 includes an email sending service that needs to be configured before it can function properly. This service, for instance, is used to send emails to users who forgot their password or username.

1.2.1 Configure the outgoing email account

The email service can use any SMTP account configured in:

- For JBoss: `$JBOSS_HOME/standalone/configuration/gatein/configuration.properties`
- For Tomcat: `$TOMCAT_HOME/gatein/conf/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 preconfigured for GMail, so that any GMail account can easily be used (simply use the full GMail address with username and password.

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

1.3 HTTPS Configuration

GateIn Portal3.5 default run on the HTTP mode. However, for security purpose, you can config GateIn Portal to run on the HTTPS mode. This section show you how to config GateIn with HTTPS mode.

1.3.1 Generate your key

If you haven't your own X.509 certificate, you can make a simple certificate using the `keytool` command:

```
keytool -genkey -alias serverkeys -keyalg RSA -keystore server.keystore -storepass 123456
-keypass 123456 -dname "CN=localhost, OU=MYOU, O=MYORG, L=MYCITY, ST=MYSTATE, C=MY"
```

Now, your key is stored in `server.keystore`.

You need to import your key into the Sun JDK keystore (This is required to help running gadget features)

```
keytool -importkeystore -srckeystore server.keystore -destkeystore
$JAVA_HOME/jre/lib/security/cacerts
```

Note

On OS X `cacerts` file is located at `$JAVA_HOME/lib/security/cacerts`.

Also, since your Sun JDK keystore has a different password than the one used for the key you created in the first step you have to change your key password to match the new keystore password (probably it's the default JDK trustore pasword: 'changeit')

```
keytool -keypasswd -alias serverkeys --keystore $JAVA_HOME/jre/lib/security/cacerts
```

1.3.2 Setup JBoss configuration to use your key

1. Edit `$JBOSS_HOME/standalone/configuration/standalone.xml` by adding https connector to web subsystem configuration (change `certificate-key-file` and `password` to values appropriate for your keystore - here we assume the keystore password is 'changeit'):

```
<subsystem xmlns="urn:jboss:domain:web:1.2" default-virtual-server="default-host"
native="false">
    ...
    <connector name="https" protocol="HTTP/1.1" socket-binding="https" scheme="https"
secure="true">
        <ssl name="https" key-alias="serverkeys" password="changeit"
certificate-key-file="${java.home}/jre/lib/security/cacerts"/>
    </connector>
    ...
</subsystem>
```

You can now access the portal by going to https://localhost:8443/portal.

1.3.3 Setup Tomcat configuration to use your key

1. Edit `server.xml` from `tomcat/conf` folder by commenting the lines:

```
<Connector port="8080" protocol="HTTP/1.1"
 maxThreads="150" connectionTimeout="20000"
 redirectPort="8443" URIEncoding="UTF-8"
 emptySessionPath="true"/>
```

1. Uncomment lines and add keystoreFile and keystorePass values:

```
<Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true"
 maxThreads="150" scheme="https" secure="true"
 clientAuth="false" sslProtocol="TLS"
 keystoreFile="${java.home}/jre/lib/security/cacerts"
 keystorePass="changeit" />
```

1. Restart GateIn. If your configuration is correct, you can access to GateIn via `https://<ServerAddress>:8443/portal`.

1.4 Configuration of custom data validators

GateIn Portal 3.5 includes a user-configurable validator that can be applied to input fields of different bundled portlets. Currently, this validator is only used to configure the validation of username formats in the user account, user registration and group membership portlets, though the architecture allows for configurable validation to be used in different contexts if needed.

The validator can be configured via properties in the `configuration.properties` file found in the GateIn configuration directory. By default, this directory is found at `$JBOSS_HOME/standalone/configuration/gatein/` if you are using JBoss Application Server or `$TOMCAT_HOME/gatein/conf/` if you are using Tomcat.

The architecture supports several configurations that can be activated and associated to specific instances of the user-configurable validator when they are created and assigned to fields in portlets. We will only concern ourselves with the currently supported use cases, which are creation/modification of a username during registration/modification of a user and group membership assignments.

1.4.1 Validator configuration

A configuration is created by adding an entry to `configuration.properties` using the `gatein.validators.` prefix followed by the name of the configuration, a period '.' and the name of the validation aspect you want to configure. The user-configurable validator currently supports four different aspects per configuration, as follows, where {`configuration`} refers to the configuration name:

- `gatein.validators.{configuration}.length.min`: Minimal length of the validated field.
- `gatein.validators.{configuration}.length.max`: Maximal length of the validated field.
- `gatein.validators.{configuration}.regexp`: Regular expression to which values of the validated field must conform.
- `gatein.validators.{configuration}.format.message`: Information message to display when the value of the validated field does not conform to the specified regular expression.

Only two configurations are currently supported by GateIn:

- `username` which configures the validation of usernames when they are created/modified.
- `groupmembership` which configures the validation of usernames in the context of group memberships.

If you want to make sure that your users use an email address as their usernames, you could use the following configuration:

```
# validators
gatein.validators.username.regexp=^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-za-z]{2,4}$
gatein.validators.username.format.message=Username must be a valid email address.
```

If you do not change the configuration of the validator, the username will be validated as follows:

- Length must be between 3 and 30 characters.
- Only lowercase letters, numbers, undescores (_) and period (.) can be used.
- No consecutive undescores (_) or period (.) can be used.
- Must start with a letter.
- Must end with a letter or number.

Some components that leverage GateIn depend on usernames being all lowercase. Therefore, it is strongly recommended that you only accept the lowercase usernames.

1.4.2 Developer information

The user-configurable validator is implemented by the `org.exoplatform.webui.form.validator.UserConfigurableValidator` class. Please refer to its documentation for more details.

To use a specific validator configuration to validate a given field value, add the validator to the field where `configurationName` is a `String` representing the name of the configuration to use:

```
addValidator(UserConfigurableValidator.class, configurationName))
```

The validator instance can then be configured by adding the relevant information in `configuration.properties`, for example:

```
# validators
gatein.validators.configurationName.length.min=5
gatein.validators.configurationName.length.max=10
gatein.validators.configurationName.regexp=^u\d{4,9}$
gatein.validators.configurationName.format.message=Username must start with ''u'' and be
followed by 4 to 9 digits.
```

Alternatively, a resource key can also be passed to the `addValidator` method to specify which localized message should be used in case a validation error occurs, for example `configurationName` as follows:

```
addValidator(UserConfigurableValidator.class, UserConfigurableValidator.GROUPMEMBERSHIP,
UserConfigurableValidator.GROUP_MEMBERSHIP_LOCALIZATION_KEY);
```

1.5 Rememberme password encryption

1.5.1 Introduction

Automatic login feature of GateIn Portal employs token mechanism to authenticate returning users without asking their explicit logins.

For the moment, token storage contains a security hole as user passwords are persisted in plain form. The high risk from such unsecured implementation boost us to find an encryption mechanism which:

1. Bases on secured algorithm.
2. Functions with secret factors created/maintained by customers.
3. Generates not-too-long encrypted data.

1.5.2 based encryption

We decided to build a symmetric encryption over JCA - Java Cryptography Architecture library whose default algorithm is AES

Configuration

Default configuration entry of JCA-based encryption is declared in configuration.properties file

```
gatein.codec.builderclass=org.exoplatform.web.security.codec.JCASymmetricCodecBuilder
gatein.codec.config=${gatein.conf.dir}/codec/jca-symmetric-codec.properties
```

Detailed parameters for encryptions whose builder is *org.exoplatform.web.security.codec.JCASymmetricCodecBuilder* are referred in the file jca-symmetric-codec.properties

```
# Defailed information on JCA standard names could be found at
#
# http://docs.oracle.com/javase/6/docs/technotes/guides/security/StandardNames.html#KeyStore
#
# The file key.txt is generated via keytool util in JDK
#
# keytool -genseckey -alias "gtnKey" -keypass "gtnKeyPass" -keyalg "AES" -keysize 128 -keystore
"key.txt" -storepass "gtnStorePass" -storetype "JCEKS"
#
#
gatein.codec.jca.symmetric.alias=gtnKey
gatein.codec.jca.symmetric.keypass=gtnKeyPass
gatein.codec.jca.symmetric.keyalg=AES
gatein.codec.jca.symmetric.keystore=key.txt
gatein.codec.jca.symmetric.storepass=gtnStorePass
gatein.codec.jca.symmetric.storetype=JCEKS
```

Customization

A crucial point of our encryption is that secret factors (algorithm, key storage, key size,...) are created/maintained on customer side, hence keep it private to them.

Below are steps to customize those secret factors in products using JCASymmmetricCodecBuilder.

Generate secret key via keytool

```
$JAVA_HOME/bin/keytool -genseckey -alias "customAlias" -keypass "customKeyPass" -keyalg
"customAlgo" -keystore "customStore" -storepass "customStorePass" -storetype "customStoreType"
```

The above keytool command generates secret key stored in a file named *customStore*. Let's copy the file to the directory gatein/conf/codec.

NOTEs:

* The list of standard algorithms could be found here
* Extra params for keytool might be required for special algorithms.
* In JCA, only JCEKS storetype supports symmetric key.

Updates jca-symmetric-codec.properties

Remain work is updating the file jca-symmetric-codec.properties with parameters used in previous step.

```
gatein.codec.jca.symmetric.alias=customAlias
gatein.codec.jca.symmetric.keypass=customKeyPass
gatein.codec.jca.symmetric.keyalg=customAlgo
gatein.codec.jca.symmetric.keystore=customStore
gatein.codec.jca.symmetric.storepass=customStorePass
gatein.codec.jca.symmetric.storetype=customStoreType
```

1.6 Send mail to administrator when new user is registered

It may be useful for a portal administrator to be notified when a new user registers with the portal, particularly if the portal provides a public registration page, which is the case with GateIn Portal. This feature is disabled by default. To configure the new user email service, perform the following configuration:

1. Configure your SMTP server - You will need to configure correctly your SMTP server as described in Email Service Configuration
2. Configure the service used for sending emails - There is new component **PostRegistrationService**, which can be configured in `GATEIN_HOME/gatein/gatein.ear/portal.war/WEB-INF/conf/admin/admin-configura`. There are descriptions in this file for each of the available parameters. Most importantly, the two parameters that need to be configured at the minimum are:
    - **sendMailAfterRegistration** - The value needs to be changed to `true` (It's false by default so sending emails after registration is disabled)
    - **mailTo** - You need to add your email address here.
3. Test - Restart the server and register a new user via GateIn Portal user interface. An email will be sent notifying you that a new user has been registered.

1.7 Clustering configuration

Let's assume that you want to setup 2-nodes cluster of GateIn Portal servers on JBoss AS7 and one loadbalancer server, which will use Apache HTTPD+Mod_jk

1.7.1 Portal cluster setup

1. Unzip your GateIn Portal bundle (assuming directory $GATEIN_HOME where bundle is unzipped) and create a copy for each cluster node:

```
$ cp -r $GATEIN_HOME/ node1
$ cp -r $GATEIN_HOME/ node2
```

   Let's assume that you have 2 available IP addresses *192.168.210.101* (here node1 will be executed) and *192.168.210.102* (here node2 will be executed). You can use either:
   - Two separate physical servers. In this case you need to have folder "node1" available on first server and folder "node2" needs to be available on second server.
   - One physical server, which is using two virtual IP addresses (Consult documentation of your OS on how to setup virtual IP addresses)
2. All portal servers in cluster need to share same database. You need to choose one of two approaches:
   1. If you have setup with one physical server and two virtual IP addresses, you can use preconfigured H2 database without need to setup your own database. You can run the DB server (from GATEIN_HOME directory)

```
$ java -cp modules/com/h2database/h2/main/h2-1.3.168.jar org.h2.tools.Server
```

   2. For production setup with real cluster, it's recommended to install and setup other database and configure both nodes to use it. More info about database configuration via JBoss datasources is in Database Configuration chapter.

> ⚠ You need to edit file `standalone-ha.xml` instead of `standalone.xml` because it's clustered setup

3. Run the servers. From the **node1** directory:

```
$ ./bin/standalone.sh --server-config=standalone-ha.xml -Djboss.node.name=node1 -b
192.168.210.101 -u 239.23.42.2 -Djboss.bind.address.management=192.168.210.101
```

   And from the **node2** directory:

```
$ ./bin/standalone.sh --server-config=standalone-ha.xml -Djboss.node.name=node2 -b
192.168.210.102 -u 239.23.42.2 -Djboss.bind.address.management=192.168.210.102
```

   Now you can directly access node1 on http://192.168.210.101:8080/portal and node2 on http://192.168.210.102:8080/portal

1.7.2 Setting up mod_jk

1. Install the apache server and mod_jk module
   On Fedora, the package containing Apache HTTP Server is named **httpd**. You will probably need to build and install mod_jk from sources, in which case the package **httpd-devel** might be useful. Verify that the file `mod_jk.so` is present in `/etc/httpd/modules` .
   On Ubuntu the packages are named **apache2** and **libapache2-mod-jk**.
2. Setup apache to use mod_jk
   - When using Fedora and recent version of Apache (2.2+), put file `mod-jk.conf` into `/etc/httpd/conf.d`. Don't forget to load the module by appending the following line to `/etc/httpd/conf/httpd.conf` .

     ```
     LoadModule jk_module modules/mod_jk.so
     ```

     File `mod-jk.conf` needs to look like this:

     ```
     # Where to find workers.properties
     JkWorkersFile workers.properties

     # Where to put jk logs
     JkLogFile /var/log/apache2/mod_jk.log

     # Set the jk log level [debug/error/info]
     JkLogLevel debug

     # Select the log format
     JkLogStampFormat "[%a %b %d %H:%M:%S %Y]"

     # JkOptions indicates to send SSK KEY SIZE
     JkOptions +ForwardKeySize +ForwardURICompat -ForwardDirectories

     # JkRequestLogFormat
     #JkRequestLogFormat "%w %V %T"

     JkMountFile uriworkermap.properties

     # Add shared memory.
     # This directive is present with 1.2.10 and
     # later versions of mod_jk, and is needed for
     # for load balancing to work properly
     JkShmFile /var/log/apache2/jk.shm

     # Add jkstatus for managing runtime data
     <Location /jkstatus/>
         JkMount status
     </Location>
     ```

   - When using Ubuntu, you can create file `mod-jk.conf` with above content into `/etc/apache2/mods-enabled`

3. Setup workers
   in `/etc/httpd/` (or `/etc/apache2/` in case of Ubuntu) create `workers.properties` file. And make sure that balanced workers have same names as the name of *jboss.node.name+ attribute used to run both nodes, which has been configured in previous steps. In our case, we used names _node1* and *node2* in *server.xml*. This means that particular balanced workers in `workers.properties` file also need to have same names *node1* and *node2* . File `workers.properties` in our case can look like this:

```
# Define list of workers that will be used
# for mapping requests
worker.list=loadbalancer,status

# modify the host as your host IP or DNS name
worker.node1.port=8009
worker.node1.host=192.168.210.101
worker.node1.type=ajp13
worker.node1.lbfactor=1

## modify the host as your host IP or DNS name
worker.node2.port=8009
worker.node2.host=192.168.210.102
worker.node2.type=ajp13
worker.node2.lbfactor=1

# Load-balancing behaviour
worker.loadbalancer.type=lb
worker.loadbalancer.method=Session
worker.loadbalancer.balance_workers=node1,node2
worker.loadbalancer.sticky_session=1

#worker.list=loadbalancer
worker.status.type=status
```

   And file `uriworkermap.properties` which should be like this

```
/portal=loadbalancer
/portal/*=loadbalancer
/eXo*=loadbalancer
/eXoResources*/*=loadbalancer
/exo*=loadbalancer
/exo*/*=loadbalancer
/web=loadbalancer
/web/*=loadbalancer
/integration=loadbalancer
/integration/*=loadbalancer
/dashboard=loadbalancer
/dashboard/*=loadbalancer
/rest=loadbalancer
/rest/*=loadbalancer
/jpp_branding_skin|/*=loadbalancer
/jpp-branding-skin|/*=loadbalancer
/jpp-branding-extension|/*=loadbalancer
/status=status
/status/*=status
```

4. Problems & Solutions
    1. You've configured everything properly, yet you're unable to access the portal via Apache.
        - **Description:** When accessing EPP via Apache, you're getting "503 Service Temporarily Unavailable" response.
          The cluster itself is working and you can access individual EPP nodes directly.
          According to `mod_jk.log`, mod_jk is working, but Tomcat is reported as likely not to be running on the specified port.
        - **Possible cause:**
          If you're using Fedora, SELinux might be stopping httpd from accessing something important, e.g. jk.shm. Check SELinux alerts to see if that's the case.
        - **Solution:**
          Ideally, you would create a policy to deal with this. A quick workaround is to temporarily disable SELinux to allow Apache to initialize the `mod_jk` connector properly. You can do this using the following command:

          ```
          setenforce 0
          ```

2 Portal Development

2.1 Skinning the portal

GateIn Portal3.5 provides robust skinning support for the entire portal User Interface (UI). This includes support for skinning all of the common portal elements as well as being able to provide custom skins and window decorations for individual portlets. It is designed for common graphic resource reuse and ease of development.

2.1.1 Skin Components

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

- *Portal Skin*

  > *The portal skin contains the CSS styles for the portal and its various UI components. This should include all the UI components, except for the window decorators and portlet specific styles.*

- *Window Styles*

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

- *Portlet Skins*

  > *The portlet skins affects how portlets are rendered on the page via one of the following ways:*
  >
  > - *Portlet Specification CSS Classes*

  The portlet specification defines a set of CSS classes that should be available to portlets. GateIn Portal3.5 provides these classes as part of the portal skin. This allows each portal skin to define its own look and feel for these default values.

  > - *Portlet Skins*

  GateIn Portal3.5 provides a means for portlet CSS files to be loaded, based on the current portal skin. This allows a portlet to provide different CSS styles to better match the current portal look and feel. Portlet skins provide a much more customizable CSS experience than just using the portlet specification CSS classes.

  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 part of the overall portal skin. The portal skin must include these components´ CSS classes or they will not be displayed correctly.
  A portlet skin does not need to be included as part of the portal skin and can be included within the portlet web application.

2.1.2 Skin Selection

Skin Selection Through the User Interface

There are a few means for you to select the display skin. The easiest way to change the skin is to select it through the user interface. Administrators can change the default skin for the portal, or users who logged in can select their desired display skins.

Please see the User Guide for information on how to change the skin using the user interface.

Setting the Default Skin within the Configuration Files

The default skin can also be configured through the portal configuration files if you do not want to use the admin user interface. This will allow the portal to have the new default skin ready when GateIn Portal is initially started.

The default skin of the portal is called `Default`. To change this value, add a `skin` tag to the `portal.war/WEB-INF/conf/portal/portal/classic/portal.xml` configuration file.

To change the skin to `MySkin`, you would make the following changes:

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

2.1.3 Skins in Page Markups

A GateIn Portal skin contains CSS styles for the portal's components but also shares components that may be reused in portlets. When GateIn Portal3.5 generates a portal page markup, it inserts stylesheet links in the page's `head` tag.

There are two main types of CSS links that will appear in the `head` tag: a link to the portal skin CSS file and a link to the portlet skin CSS files.

- *Portal Skin*

  > *The portal skin will appear as a single link to a CSS file. This link contains content from all the portal skin classes merged into one file. This allows the portal skin to be transfered more quickly as a single file instead of multiple smaller files. All pages of the portal have the same skin defined in the CSS file.*

- *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.*

  In the code fragment below, you can see the 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>
```

Both window styles and portlet specification CSS classes are included in the portal skin.

2.1.4 Skin Service

The skin service of GateIn Portal3.5 manages various types of skins. This service is responsible for discovering and deploying the skins into the portal.

Skin configuration

GateIn Portal3.5 automatically discovers web archives that contain a file descriptor for skins (`WEB-INF/gatein-resources.xml`). This file is reponsible for specifying the portal, portlet and window decorators to be deployed into the skin service.

The full schema can be found in the lib directory: `exo.portal.component.portal.jar/gatein_resources_1_0.xsd`.

Here is an example where a skin (MySkin) with its CSS location is defined, and a few window decorator skins are specified:

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

  <!-- 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>
    ...
```

Resource Request Filter

Because of the Right-To-Left support, all CSS files need to be retrieved through a Servlet filter and the web application needs to be configured to activate this filter. This is already done for `eXoResources.war` web application which contains the default skin.

Any new web applications containing skinning CSS files will need to have the following added to their `web.xml`:

```
<filter>
      <filter-name>ResourceRequestFilter</filter-name>
      <filter-class>org.exoplatform.portal.application.ResourceRequestFilter</filter-class>
      </filter>

      <filter-mapping>
      <filter-name>ResourceRequestFilter</filter-name>
      <url-pattern>*.css</url-pattern>
      </filter-mapping>
```

The `display-name` element also needs to be specified in `web.xml` for the skinning service to work properly with the web application.

2.1.5 Default Skin

The default skin for GateIn Portal3.5 is located as part of the `eXoResources.war`. The main files associated with the skin are shown below:

```
WEB-INF/gatein-resources.xml
WEB-INF/web.xml
skin/Stylesheet.css
```

Define the skin setup to use.

Contain the resource filter and has the display-name set.

Contain the CSS class definitions for this skin.

- `gatein-resources.xml`

  > *For the default portal skin, this file contains definitions for the portal skin, the window decorations and defines some Javascript resources which are not related to the skin. The default portal skin does not directly define portlet skins which should be provided by the portlets themselves.*

- `web.xml`

  > *For the default portal skin, `web.xml` of the `eXoResources.war` contains a lot of information which is mostly irrelevant to the portal skinning. The areas of interest in this file is `ResourceRequestFilter` and the fact that `display-name` is set.*

- `Stylesheet.css`

  > *This file contains the main stylesheet of the portal skin. The file is the main entry point to the CSS class definitions for the skin. The content of this file is shown below:*
  >
  > ```
  > @import url(DefaultSkin/portal/webui/component/UIPortalApplicationSkin.css);
  > @import url(DefaultSkin/webui/component/Stylesheet.css);
  > @import url(PortletThemes/Stylesheet.css);
  > @import url(Portlet/Stylesheet.css);
  > ```
  >
  > *Skin for the main portal page.*
  > *Skins for various portal components.*
  > *Window decoration skins.*
  > *The portlet specificiation CSS classes.*
  > *Instead of defining all the CSS classes in this file, you can import other CSS stylesheet files, some of which may also import other CSS stylesheets. The CSS classes are split up between multiple files to make it 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 would need to use this import:*
  >
  > ```
  > @import url(/eXoResources/skin/Portlet/Stylesheet.css);
  > ```
  >
  > 
  > *When the portal skin is added to the page, it merges all the CSS stylesheets into a single file.*

2.1.6 Creating New Skins

Creating a New Portal Skin

A new portal skin needs to be added to the portal through the skin service. The web application which contains the skin will need to be properly configured for the skin service to discover them. This means properly configuring `ResourceRequestFilter` and `gatein-resources.xml`.

Portal Skin Configuration

The `gatein-resources.xml` file specifies the new portal skin, including its name and where to locate its CSS stylesheet file and whether to overwrite an existing portal theme with the same name.

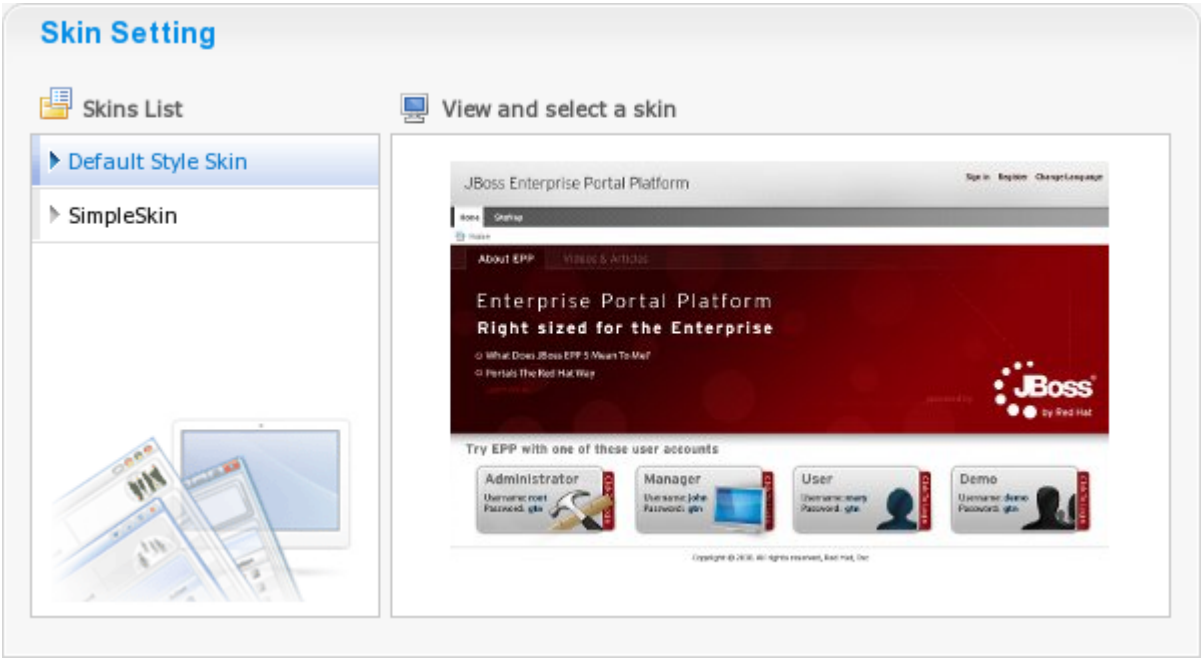
```
<gatein-resources>
  <portal-skin>
    <skin-name>MySkin</skin-name>
    <css-path>/skin/myskin.css</css-path>
    <overwrite>false</overwrite>
  </portal-skin>
</gatein-resources>
```

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

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

Portal Skin Preview Icon

When selecting a skin, it is possible to preview it. 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.



The portal skin preview icon is specified through the CSS of the portal skin. In order for the current portal skin to be able to display the preview, it must specify a specific CSS class and set the icon as the background.

For a portal named *MySkin*, the CSS class must be defined as follows:

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

In order for the default skin to know about the skin icon for a new portal skin, the preview screenshot needs to be placed in:

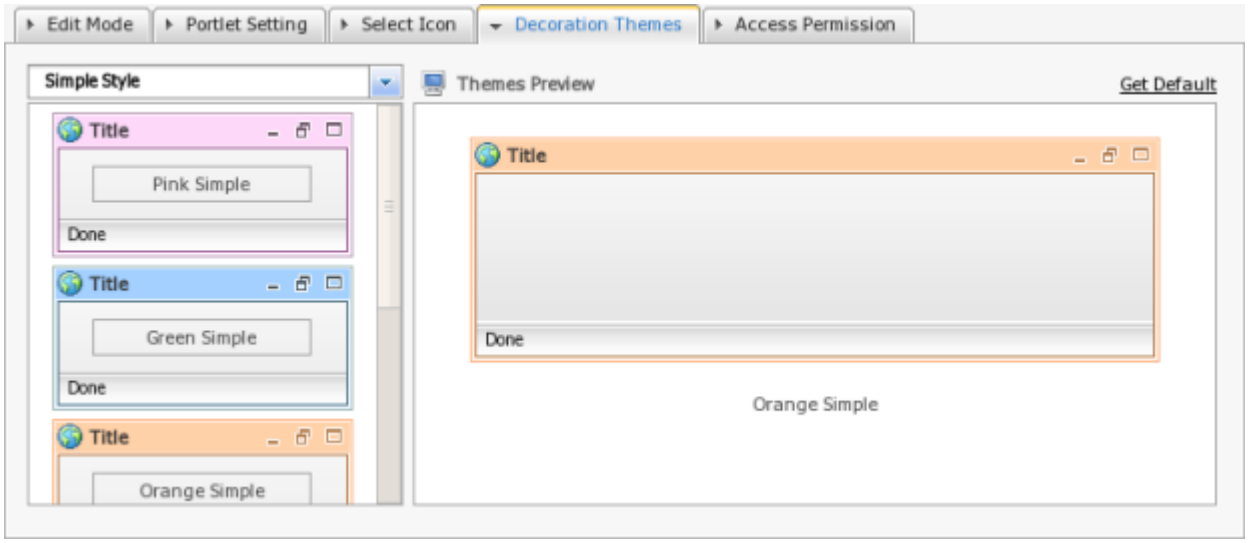
{{eXoResources.war:/skin/DefaultSkin/portal/webui/component/customization/UIChangeSkinForm/backgroun
CSS stylesheet for the default portal needs to have the following updated with the preview icon CSS class. For a skin named *MySkin* then the following needs to be updated:

`eXoResources.war:/skin/DefaultSkin/portal/webui/component/customization/UIChange`

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

Creating a New Window Style

Window styles are the CSS applied to the window decoration. When an administrator chooses a new application to add to a page, he can decide which style of decoration would go around the window if any.



Window Style Configuration

Window Styles are defined within the `gatein-resources.xml` file which is used by the skin service to deploy the window style into the portal. Window styles can belong in with a window style category, this category and the window styles need to be specified in resources file.

The following `gatein-resource.xml` fragment adds 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 configuration for the default skin is configured in: `eXoResources.war/WEB-INF/gatein-resources.xml`.

When a window style is defined in the `gatein-resources.xml` file, it will be available to all portlets whether the current portal skin supports the window decorator or not. When a new window decorator is added, it is recommended you add it to all portal skins or that portal skins share a common stylesheet for window decorators.

Window Style CSS

To display the window decorators for the skin service, it must have the CSS classes with specific naming related to the window style name. The service will try and display the CSS based on this naming. The CSS class must be included as part of the current portal skin for displaying the window decorators.

The location of the window decorator CSS classes for the default portal theme is located at: `eXoResources.war/skin/PortletThemes/Stylesheet.css`.

Create the CSS file:

```
/*---- MyTheme ----*/
.MyTheme .WindowBarCenter .WindowPortletInfo {
 margin-right: 80px; /* orientation=lt */
 margin-left: 80px; /* orientation=rt */
}

.MyTheme .WindowBarCenter .ControlIcon {
 float: right; /* orientation=lt */
 float: left; /* orientation=rt */
 width: 24px;
 height: 17px;
 cursor: pointer;
 background-image: url('background/MyTheme.png');
}

.MyTheme .ArrowDownIcon {
 background-position: center 20px;
}

.MyTheme .OverArrowDownIcon {
 background-position: center 116px;
}

.MyTheme .MinimizedIcon {
 background-position: center 44px;
}

.MyTheme .OverMinimizedIcon {
 background-position: center 140px;
}

.MyTheme .MaximizedIcon {
 background-position: center 68px;
}

.MyTheme .OverMaximizedIcon {
 background-position: center 164px;
}

.MyTheme .RestoreIcon {
 background-position: center 92px;
}

.MyTheme .OverRestoreIcon {
 background-position: center 188px;
}

.MyTheme .NormalIcon {
 background-position: center 92px;
}
```

```
.MyTheme .OverNormalIcon {
 background-position: center 188px;
}

.MyTheme .Information {
 height: 18px; line-height: 18px;
 vertical-align: middle; font-size: 10px;
 padding-left: 5px; /* orientation=lt */
 padding-right: 5px; /* orientation=rt */
 margin-right: 18px; /* orientation=lt */
 margin-left: 18px; /* orientation=rt */
}

.MyTheme .WindowBarCenter .WindowPortletIcon {
 background-position: left top; /* orientation=lt */
 background-position: right top; /* orientation=rt */
 padding-left: 20px; /* orientation=lt */
 padding-right: 20px; /* orientation=rt */
 height: 16px;
 line-height: 16px;
}

.MyTheme .WindowBarCenter .PortletName {
 font-weight: bold;
 color: #333333;
 overflow: hidden;
 white-space: nowrap;
}

.MyTheme .WindowBarLeft {
 padding-left: 12px;
 background-image: url('background/MyTheme.png');
 background-repeat: no-repeat;
 background-position: left -148px;
}

.MyTheme .WindowBarRight {
 padding-right: 11px;
 background-image: url('background/MyTheme.png');
 background-repeat: no-repeat;
 background-position: right -119px;
}

.MyTheme .WindowBarCenter {
 background-image: url('background/MyTheme.png');
 background-repeat: repeat-x;
 background-position: left -90px;
 height: 21px;
 padding-top: 8px;
}

.MyTheme .MiddleDecoratorLeft {
 padding-left: 12px;
 background: url('background/MMyTheme.png') repeat-y left;
}

.MyTheme .MiddleDecoratorRight {
 padding-right: 11px;
```

```
 background: url('background/MMyTheme.png') repeat-y right;
}

.MyTheme .MiddleDecoratorCenter {
 background: #ffffff;
}

.MyTheme .BottomDecoratorLeft {
 padding-left: 12px;
 background-image: url('background/MyTheme.png');
 background-repeat: no-repeat;
 background-position: left -60px;
}

.MyTheme .BottomDecoratorRight {
 padding-right: 11px;
 background-image: url('background/MyTheme.png');
 background-repeat: no-repeat;
 background-position: right -30px;
}

.MyTheme .BottomDecoratorCenter {
 background-image: url('background/MyTheme.png');
 background-repeat: repeat-x;
 background-position: left top;
 height: 30px;
}
```

Set Default Window Style

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

You do not need to specify `DefaultTheme` in `gatein-resources.xml`.

Create New Portlet skins

Portlets often require additional styles that may not be defined by the portal skin. GateIn Portal3.5 allows portlets to define additional stylesheets for each portlet and will append the corresponding `link` tags to the `head`.

The link ID will be of the form: {`portletAppName`}{`PortletName`}. For example, `ContentPortlet` in `content.war` will give `id="contentContentPortlet"`.

To define a new CSS file to include whenever a portlet is available on a portal page, the following fragment needs to be added to `gatein-resources.xml` as follows:

```
<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 will load `DefaultStylesheet.css` when the Default skin is used and the `OtherSkinStylesheet.css` when the OtherSkin is used.

If the current portal skin is not defined as part of the supported skins, then the portlet CSS class will not be loaded. It is recommended you update portlet skins 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 the portlet webapplication:

- {{skin/DefaultSkin/portletIcons/}}icon_name.png
  To use the icon correctly, it must be named after the portlet.

The icon for an account portlet named AccountPortlet would be located at: {{skin/DefaultSkin/portletIcons/}}AccountPortlet.png.

You must use `skin/DefaultSkin/portletIcons/` for the directory to store the portlet icon regardless of what skin is going to be used.

Create New Portlet Specification CSS Classes

The portlet specification defines a set of default CSS classes that should be available for portlets. These classes are included as part of the portal skin. Please see the portlet specification for a list of the default classes that should be available.

For the default portal skin, the portlet specification CSS classes are defined in: `eXoResources.war/skin/Portlet/Stylesheet.css`.

2.1.7 Tips and Tricks

Easier CSS debugging

By default, CSS files are cached and their imports are merged into a single CSS file at the server side. This reduces the number of HTTP requests from the browser to the server.

The optimization code is quite simple as all the CSS files are parsed at the server startup time and all the @import and url(...) references are rewritten to support a single flat file. The result is stored in a cache directly used from `ResourceRequestFilter`.

Although the optimization is useful for production environments, it may be easier to deactivate this optimization while debugging stylesheets. To do so, set the java system property `exo.product.developing` to `true`.

For example, the property can be passed as a JVM parameter with the `-D` option when running GateIn Portal.

```
sh $JBOSS_HOME/bin/standalone.sh -Dexo.product.developing=true
```

This option may cause display bugs with certain browsers, such as Internet Explorer.

Some CSS techniques

It is recommended you have some experiences with CSS before studying GateIn Portal CSS.

GateIn Portal3.5 relies heavily on CSS to create the layout and effects for the UI. Some common techniques for customizing GateIn Portal CSS are explained below.

Decorator pattern

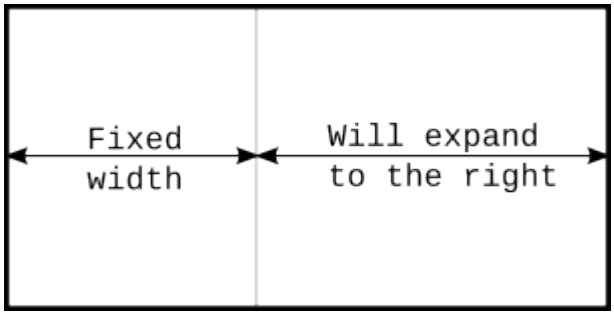
The decorator is a pattern to create a contour or a curve around an area. To achieve this effect, you need to create 9 cells. The BODY is the central area for you to decorate. The other 8 cells are distributed around the BODY cell. You can use the width, height and background image properties to achieve any desired decoration effects.

TopLeft	TopCenter	TopRight
CenterLeft	BODY	CenterRight
BottomLeft	BottomCenter	BottomRight

```
<div class="Parent">
  <div class="TopLeft">
    <div class="TopRight">
      <div class="TopCenter"><span></span></div>
    </div>
  </div>
  <div class="CenterLeft">
    <div class="CenterRight">
      <div class="CenterCenter">BODY</div>
    </div>
  </div>
  <div class="BottomLeft">
    <div class="BottomRight">
      <div class="BottomCenter"><span></span></div>
    </div>
  <div>
</div>
```

Left margin left pattern

Left margin left pattern is a technique to create 2 blocks side by side. The left block has a fixed size and the right block will take the rest of the available space. When the user resizes the browser, the added or removed space will be taken from the right block.




```
<div class="Parent">
 <div style="float: left; width: 100px">
 </div>
 <div style="margin-left: 105px;">
 <div>
 <div style="clear: left"><span></span></div>
</div>
```

2.2 Portal Lifecycle

This section describes the portal lifecycle from the application server start to its stop and how requests are handled.

2.2.1 Application Server start and stop

A portal instance is simply a web application deployed as a WAR in an application server. Portlets are also part of an enhanced WAR called a portlet application.

GateIn Portal does not require any particular setup for your portlet in most common scenarios and the `web.xml` file can remain without any GateIn Portal specific configuration.

During deployment, GateIn Portal will automatically and transparently inject a servlet into the portlet application to be able to interact with it. This feature is dependent on the underlying servlet container but will work out of the box on the proposed bundles.

2.2.2 Command Servlet

The servlet is the main entry point for incoming requests, it also includes some init codes when the portal is launched. This servlet (`org.gatein.wci.command.CommandServlet`) is automatically added during deployment and mapped to `/tomcatgateinservlet`.

This is equivalent to adding the following to `web.xml`.

As the servlet is already configured, this example is for information only.

```
<servlet>
  <servlet-name>TomcatGateInServlet</servlet-name>
  <servlet-class>org.gatein.wci.command.CommandServlet</servlet-class>
  <load-on-startup>0</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>TomcatGateInServlet</servlet-name>
  <url-pattern>/tomcatgateinservlet</url-pattern>
</servlet-mapping>
```

It is possible to filter on the CommandServlet by filtering the URL pattern used by the Servlet mapping.

The example below would create a servlet filter that calculates the time of execution of a portlet request.

The filter class:

```
package org.example;

import java.io.IOException;

import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;

public class MyFilter implements javax.servlet.Filter {

  public void doFilter(ServletRequest request, ServletResponse response,
      FilterChain chain) throws IOException, ServletException
  {
    long beforeTime = System.currentTimeMillis();
    chain.doFilter(request, response);
    long afterTime = System.currentTimeMillis();
    System.out.println("Time to execute the portlet request (in ms): " + (afterTime -
beforeTime));
  }

  public void init(FilterConfig config) throws ServletException
  {
  }

  public void destroy()
  {
  }

}
```

The Java EE web application configuration file (`web.xml`) of the portlet on which we want to know the time to serve a portlet request. As mentioned above, GateIn Portal does not require anything special to be included, only the URL pattern to set has to be known.

```
<?xml version="1.0"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.5">

  <filter>
    <filter-name>MyFilter</filter-name>
    <filter-class>org.example.MyFilter</filter-class>
  </filter>

  <filter-mapping>
    <filter-name>MyFilter</filter-name>
    <url-pattern>/tomcatgateinservlet</url-pattern>
    <dispatcher>INCLUDE</dispatcher>
  </filter-mapping>

</web-app>
```

INCLUDE dispatcher

It is important to set `INCLUDE` as a dispatcher as the portal always hits the CommandServlet through a request dispatcher. Without this, the filter will not be triggered, unless the direct access to a resource, such as an image.

2.3 Default Portal Configuration

GateIn Portal 3.5 default home page URL is `[http://]{hostname}:{port}/portal/`. There may be multiple independent portals deployed in parallel at any given time, each of which has its root context (for example, `[http://]{hostname}:{port}/sample-portal/`). Each portal is internally composed of one or more, what we again call 'portals'. It is required to have at least one such portal - the default one called 'classic'. When accessing the GateIn Portal 3.5 default homepage URL, you are automatically redirected to the 'classic' portal.

The default portal performs another important task. When starting up GateIn Portal 3.5 for the first time, its JCR database will be empty (that is where portals keep their runtime-configurable settings). It is the default portal used to detect this, and to trigger automatic data initialization.

2.3.1 Configuration

The following example configuration can be found at: "`portal.war:/WEB-INF/conf/portal/portal-configuration.xml`".

```
<component>
    <key>org.exoplatform.portal.config.UserPortalConfigService</key>
    <type>org.exoplatform.portal.config.UserPortalConfigService</type>
    <component-plugins>
      <component-plugin>
        <name>new.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>classic</value>
          </value-param>
          ...
        </init-params>
      </component-plugin>
    </component-plugins>
  </component>
```

In this example, the *classic* portal has been set as the default.

Components, component-plugins, and init-params are explained in Foundations chapter. For now just note how `NewPortalConfigListener` component-plugin is used to add configuration to `UserPortalConfigService`, which is designed in this way to allow other components to add configuration to it.

2.3.2 Tips

Delete Portals Definition by Configuration

In some cases, some portal definitions are defined but not used any more. If you want to delete them, you can add some configurations to `portal.war/WEB-INF/conf/portal/portal-configuration.xml`.

To delete a portal definition or a portal template definition, you need to define a component plug-in as the example below:

```
<external-component-plugins>
  <target-component>org.exoplatform.portal.config.UserPortalConfigService</target-component>
  <component-plugin>
    <name>new.portal.config.user.listener</name>
    <set-method>deleteListenerElements</set-method>
    <type>org.exoplatform.portal.config.NewPortalConfigListener</type>
    <description>this listener delete some predefined portal and templates
configuration</description>
    <init-params>
      <object-param>
        <name>site.templates.location</name>
        <description>description</description>
        <object type="org.exoplatform.portal.config.SiteConfigTemplates">
          <field name="portalTemplates">
            <collection type="java.util.HashSet">
              <value>
                <string>basic</string>
              </value>
              <value>
                <string>classic</string>
              </value>
            </collection>
          </field>
        </object>
      </object-param>
      <object-param>
        <name>portal.configuration</name>
        <description>description</description>
        <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>
        </object>
      </object-param>
    </init-params>
  </component-plugin>
</external-component-plugins>
```

Set Info bar

You can set the info bar shown by default for portlets of a portal by adding a property to the `portal-config` configuration of the `portal.xml` file.

```
<properties>
                      <entry key="showPortletInfo">1</entry>
                    </properties>
```

There are two values for "showPortletInfo": 0 and 1. If the value is 1, the info bar of portlets is shown by default. If the value is 0, it is hidden.

2.4 Portal Default Permission Configuration

The default permission configuration for the portal is defined through the `org.exoplatform.portal.config.UserACL` component configuration in the `portal.war:/WEB-INF/conf/portal/portal-configuration.xml` file.

It defines 8 permissions types:

- *super.user*

  > *The super-user as `root` has all the rights on the the platform.*

- *portal.administrator.groups*

  > *Any member of those groups are considered administrators. The default value is `/platform/administrators`.*

- *portal.administrator.mstype*

  > *Any user with that membership type would be considered administrator or the associated group with the `manager` by default.*

- *portal.creator.groups*

  > *This list defines all groups that will be able to manage the different portals. Members of this group also have the permission to create new portals. The format is `membership:/group/subgroup`.*

- *navigation.creator.membership.type*

  > *Defines the membership type of group managers. The group managers have the permission to create and edit group pages and they can modify the group navigation.*

- *guests.group*

  > *Any anonymous user automatically becomes a member of this group when they enter the public pages.*

- *mandatory.groups*

  > *Groups that cannot be deleted.*

- *mandatory.mstypes*

  > *Membership types that cannot be deleted.*

```
<component>
  <key>org.exoplatform.portal.config.UserACL</key>
  <type>org.exoplatform.portal.config.UserACL</type>
  <init-params>
    <value-param>
      <name>super.user</name>
      <description>administrator</description>
      <value>root</value>
    </value-param>

    <value-param>
      <name>portal.creator.groups</name>
      <description>groups with membership type have permission to manage
portal</description>
      <value>*:/platform/administrators,*:/organization/management/executive-board</value>
    </value-param>

    <value-param>
      <name>navigation.creator.membership.type</name>
      <description>specific membership type have full permission with group
navigation</description>
      <value>manager</value>
    </value-param>
    <value-param>
      <name>guests.group</name>
      <description>guests group</description>
      <value>/platform/guests</value>
    </value-param>
    <value-param>
      <name>access.control.workspace</name>
      <description>groups with memberships that have the right to access the User Control
Workspace</description>
      <value>*:/platform/administrators,*:/organization/management/executive-board</value>
    </value-param>
  </init-params>
</component>
```

2.4.1 Overwrite Portal Default Permissions

When creating the custom portals and portal extensions, it is possible to override the default configuration by using `org.exoplatform.portal.config.PortalACLPlugin`, configuring it as an external-plugin of `org.exoplatform.portal.config.UserACL` service:

```
<external-component-plugins>
    <target-component>org.exoplatform.portal.config.UserACL</target-component>
    <component-plugin>
      <name>addPortalACLPlugin</name>
      <set-method>addPortalACLPlugin</set-method>
      <type>org.exoplatform.portal.config.PortalACLPlugin</type>
      <description>setting some permission for portal</description>
      <init-params>
        <values-param>
          <name>access.control.workspace.roles</name>
          <value>*:/platform/administrators</value>
          <value>*:/organization/management/executive-board</value>
        </values-param>
        <values-param>
          <name>portal.creation.roles</name>
          <value>*:/platform/administrators</value>
          <value>*:/organization/management/executive-board</value>
        </values-param>
      </init-params>
    </component-plugin>
  </external-component-plugins>
```

2.5 Portal Navigation Configuration

There are three navigation types available to portal users:

- Portal Navigation
- Group Navigation
- User Navigation

These navigations are configured using the standard XML syntax in the `portal.war:/WEB-INF/conf/portal/portal-configuration.xml` file.

```
<component>
   <key>org.exoplatform.portal.config.UserPortalConfigService</key>
   <type>org.exoplatform.portal.config.UserPortalConfigService</type>
   <component-plugins>
      <component-plugin>
         <name>new.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>classic</value>
            </value-param>
            <value-param>
               <name>page.templates.location</name>
               <description>the path to the location that contains Page templates</description>
               <value>war:/conf/portal/template/pages</value>
            </value-param>
            <value-param>
               <name>override</name>
               <description>The flag parameter to decide if portal metadata is overriden on
restarting server
            </description>
               <value>false</value>
            </value-param>
            <object-param>
               <name>site.templates.location</name>
               <description>description</description>
               <object type="org.exoplatform.portal.config.SiteConfigTemplates">
                  <field name="location">
                     <string>war:/conf/portal</string>
                  </field>
                  <field name="portalTemplates">
                     <collection type="java.util.HashSet">
                        <value><string>basic</string></value>
                        <value><string>classic</string></value>
                     </collection>
                  </field>
                  <field name="groupTemplates">
                     <collection type="java.util.HashSet">
                        <value><string>group</string></value>
                     </collection>
                  </field>
                  <field name="userTemplates">
                     <collection type="java.util.HashSet">
                        <value><string>user</string></value>
                     </collection>
                  </field>
               </object>
            </object-param>
            <object-param>
               <name>portal.configuration</name>
               <description>description</description>
               <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>conserve</string>
               </field>
            </object>
         </object-param>
         <object-param>
            <name>group.configuration</name>
            <description>description</description>
            <object type="org.exoplatform.portal.config.NewPortalConfig">
               <field name="predefinedOwner">
                  <collection type="java.util.HashSet">
                     <value><string>/platform/administrators</string></value>
                     <value><string>/platform/users</string></value>
                     <value><string>/platform/guests</string></value>
                     <value><string>/organization/management/executive-board</string></value>
                  </collection>
               </field>
               <field name="ownerType">
                  <string>group</string>
               </field>
               <field name="templateLocation">
                  <string>war:/conf/portal</string>
               </field>
               <field name="importMode">
                  <string>conserve</string>
               </field>
            </object>
         </object-param>
         <object-param>
            <name>user.configuration</name>
            <description>description</description>
            <object type="org.exoplatform.portal.config.NewPortalConfig">
               <field name="predefinedOwner">
                  <collection type="java.util.HashSet">
                     <value><string>root</string></value>
                     <value><string>john</string></value>
                     <value><string>mary</string></value>
                     <value><string>demo</string></value>
                     <value><string>user</string></value>
                  </collection>
               </field>
               <field name="ownerType">
                  <string>user</string>
               </field>
               <field name="templateLocation">
                  <string>war:/conf/portal</string>
               </field>
               <field name="importMode">
                  <string>conserve</string>
               </field>
            </object>
         </object-param>
      </init-params>
   </component-plugin>
</component-plugins>
</component>
```

This XML configuration defines where in the portal's war to look for configuration, and which portals, groups, and user specific views to include in `portal/group/user` navigation. Those files will be used to create an initial navigation when the portal is launched in the first time. That information will then be stored in the JCR content repository, and can then be modified and managed from the portal UI.

Each portal, groups and users navigation is indicated by a configuration paragraph, for example:

```
<object-param>
   <name>portal.configuration</name>
   <description>description</description>
   <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>conserve</string>
      </field>
   </object>
</object-param>
```

`predefinedOwner`: Define the navigation owner. The portal will look for the configuration files in folder with this name. If there is no suitable folder, a default portal will be created with name is this value.

`ownerType`: Define the type of portal navigation. It may be a portal, group or user.

`templateLocation`: Define the classpath where all portal configuration files are contained.

`importMode`: Define the mode for navigation import. There are 4 types of import mode:

- `conserve`: Import data when it does not exist, otherwise do nothing.
- `insert`: Import data when it does not exist, otherwise perform a strategy that adds new data only.
- `merge`: Import data when it does not exist, otherwise update data when it exists.
- `overwrite`: Overwrite data whatsoever.

Based on these parameters, portal will look for the configuration files and create a relevant portal navigation, pages and data import strategy.

- The portal configuration files are stored in folders with the {`templateLocation`}/{`ownerType`}/{`predefinedOwner`} path.
- All navigations are defined in the `navigation.xml` file.
- Pages are defined in `pages.xml`.
- Portal configuration is defined in {`ownerType`}`.xml`.
  For example, with the above configuration, the portal will look for all configuration files from the `war:/conf/portal/portal/classic` path.

2.5.1 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 *classic* portal is configured by four XML files in the `portal.war:/WEB-INF/conf/portal/portal/classic` directory:

- `portal.xml`

> *This file describes the layout and portlets that will be shown on all pages. The layout usually contains the banner, footer, menu and breadcrumbs portlets. GateIn 3.2 is extremely configurable as every view element (even the banner and footer) is a portlet.*

```
<portal-config
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.gatein.org/xml/ns/gatein_objects_1_2
http://www.gatein.org/xml/ns/gatein_objects_1_2"
   xmlns="http://www.gatein.org/xml/ns/gatein_objects_1_2">
   <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>
```

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.
Use the `page-body` tag to define where GateIn 3.2 should render the current page.
The defined classic portal is accessible to "Everyone" (at `/portal/public/classic`) but only members of the `/platform/administrators` group can edit it.

- `navigation.xml`

> *This file defines all the navigation nodes of the portal. The syntax is simple using the nested node tags. Each node refers to a page defined in the `pages.xml` file.*
> *If the administrators want to create node labels for each language, they will have to use the `xml:lang` attribute in the label tag with value of `xml:lang` which is the relevant locale. Otherwise, if they want the node label to be localized by the 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 literal property name is replaced by a localized value taken from the associated properties file matching the current locale.*

```
<node-navigation
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.gatein.org/xml/ns/gatein_objects_1_2
http://www.gatein.org/xml/ns/gatein_objects_1_2"
   xmlns="http://www.gatein.org/xml/ns/gatein_objects_1_2">
   <priority>1</priority>
   <page-nodes>
      <node>
         <name>home</name>
         <label xml:lang="en">Home</label>
         <page-reference>portal::classic::homepage</page-reference>
      </node>
      <node>
         <name>sitemap</name>
         <label xml:lang="en">SiteMap</label>
         <visibility>DISPLAYED</visibility>
         <page-reference>portal::classic::sitemap</page-reference>
      </node>
      ..........
   </page-nodes>
</node-navigation>
```

> *This navigation tree can have multiple views inside portlets (such as the breadcrumbs portlet) that render the current view node, the sitemap or the menu portlets.*

- `pages.xml`

This configuration file structure is very similar to `portal.xml` 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.

```
<page-set
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.gatein.org/xml/ns/gatein_objects_1_2
http://www.gatein.org/xml/ns/gatein_objects_1_2"
   xmlns="http://www.gatein.org/xml/ns/gatein_objects_1_2">
   <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>
```

2.5.2 Group Navigation

Group navigations are dynamically added to the user navigation at login. This allows users to see the menu of all pages assigned to any groups they belong to.

The group navigation menu is configured by two XML files (`navigation.xml` and `pages.xml`). The syntax used in these files is the same as those covered in Portal Navigation.

They are also located in the `{templateLocation}/{ownerType}/{predefinedOwner}` directory where `ownerType` is `group` and `predefinedOwner` is the path to the group. For example, `portal.war/WEB-INF/conf/portal/group/platform/administrators/`.

2.5.3 User Navigation

User navigation is a set of nodes and pages that are owned by the user. They are part of the user's dashboard.

Two files configure the user navigation (`navigation.xml` and `pages.xml`). They are located in the `{templateLocation}/{ownerType}/{predefinedOwner}` directory where `ownerType` is `user` and `predefinedOwner` is name of user who wants to create the navigation. For example, if the administrator wants to create navigation for the user named `root`, he has to locate the configuration files in `portal.war/WEB-INF/conf/portal/user/root`.

2.6 Data Import Strategy

In the Portal extension mechanism, developers can define an extension that Portal data can be customized by configurations in the extension. There are several cases which an extension developer wants to define how to customize the Portal data, for example modifying, overwriting or just inserting a bit into the data defined by the portal. Therefore, GateIn also defines several modes for each case and the only thing which a developer has to do is to clarify the usecase and reasonably configure extensions.

This section shows you how data is changed for each mode.

2.6.1 Import Mode

In this section, the following modes for the import strategy are introduced:

- `CONSERVE`
- `MERGE`
- `INSERT`
- `OVERWRITE`

Each mode indicates how the Portal data is imported. The import mode value is set whenever NewPortalConfigListener is initiated. If the mode is not set, the default value will be used in this case. The default value is configurable as a `UserPortalConfigService` initial param. For example, the bellow configuration means that default value is `MERGE`.

```
<component>
  <key>org.exoplatform.portal.config.UserPortalConfigService</key>
  <type>org.exoplatform.portal.config.UserPortalConfigService</type>
  <component-plugins>
  ............
  </component-plugins>
  <init-params>
    <value-param>
      <name>default.import.mode</name>
      <value>merge</value>
    </value-param>
  </init-params>
</component>
```

The way that the import strategy works with the import mode will be clearly demonstrated in next sections for each type of data.

2.6.2 Data Import Strategy

The 'Portal Data' term which has been referred in the previous sections can be classified into three types of object data: Portal Config, Page Data and Navigation Data; each of which has some differences in the import strategy.

Navigation Data

The navigation data import strategy will be processed to the import mode level as the followings:

- `CONSERVE`: If the navigation exists, leave it untouched. Otherwise, import data.
- `INSERT`: Insert the missing description data, but add only new nodes. Other modifications remains untouched.
- `MERGE`: Merge the description data, add missing nodes and update same name nodes.
- `OVERWRITE`: Always destroy the previous data and recreate it. In the GateIn navigation structure, each navigation can be referred to a tree which each node links to a page content. Each node contains some description data, such as label, icon, page reference, and more. Therefore, GateIn provides a way to insert or merge new data to the initiated navigation tree or a sub-tree.

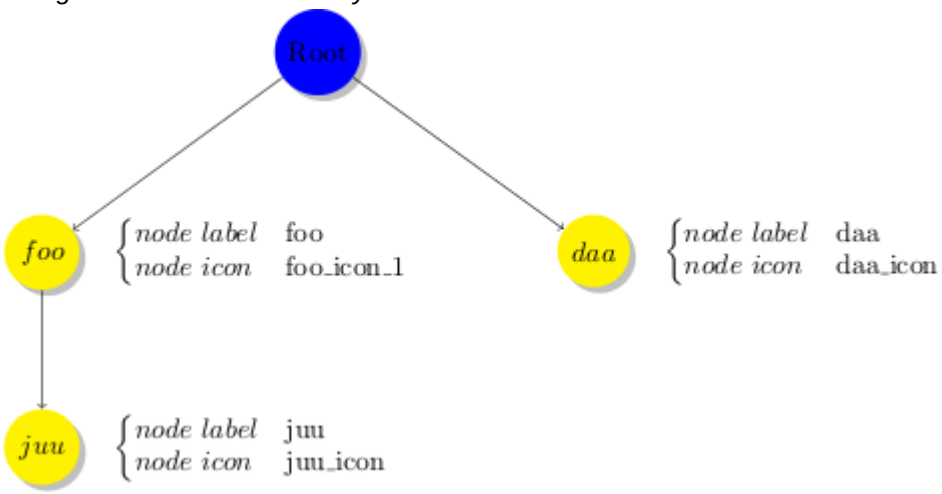
The merge strategy performs the recursive comparison of child nodes between the existing persistent nodes of a navigation and the transient nodes provided by a descriptor:

- Start with the root nodes (which is the effective root node or another node if the parent URI is specified).
- Compare the set of child nodes and insert the missing nodes in the persistent nodes.
- Proceed recursively for each child having the same name.

Let's see the example with two navigation nodes in each import mode. In this case, there are 2 navigation definitions:

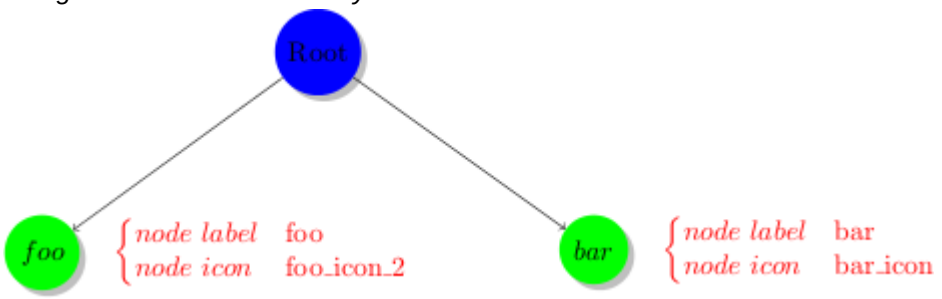
```
<node-navigation>
  <page-nodes>
    <node>
      <name>foo</name>
      <icon>foo_icon_1</icon>
      <node>
        <name>juu</name>
        <icon>juu_icon</icon>
      </node>
    </node>
    <node>
      <name>daa</name>
      <icon>daa_icon</icon>
    </node>
  </page-nodes>
</node-navigation>
```

- *Navigation node tree hierarchy*



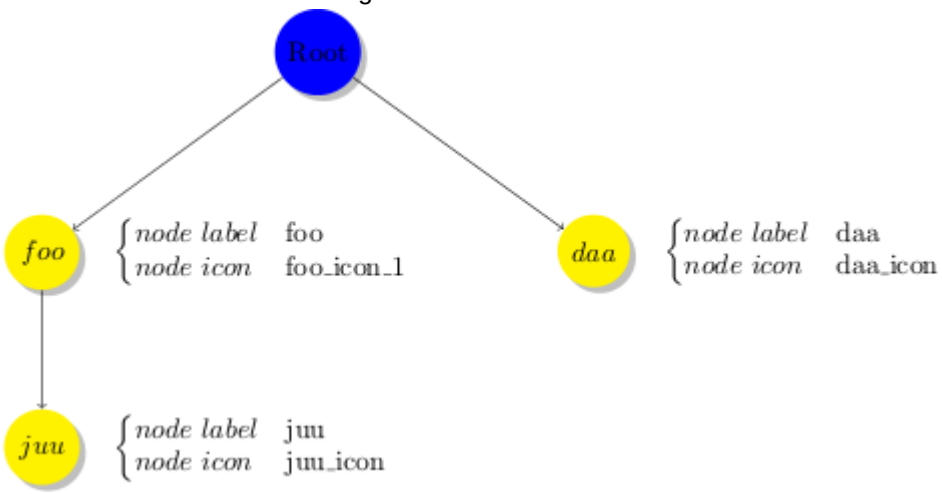

```
<node-navigation>
  <page-nodes>
    <node>
      <name>foo</name>
      <icon>foo_icon_2</icon>
    </node>
    <node>
      <name>bar</name>
      <icon>bar_icon</icon>
    </node>
  </page-nodes>
</node-navigation>
```

- *Navigation node tree hierarchy*

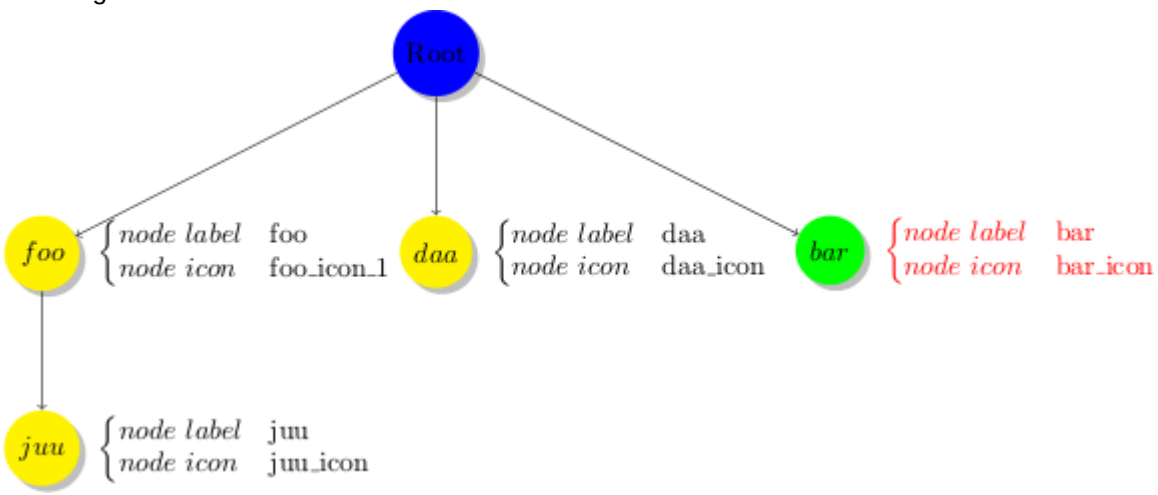


For example, the *navigation1* is loaded before *navigation2*. The Navigation Importer processes on two navigation definitions, depending on the Import Mode defined in portal configuration.

- *Case 1: Import mode is* `CONSERVE`
  With the `CONSERVE` mode, data are only imported when they do not exist. So, if the navigation has been created by the *navigation1* definition, the *navigation2* definition does not affect anything on it. We have the result as following

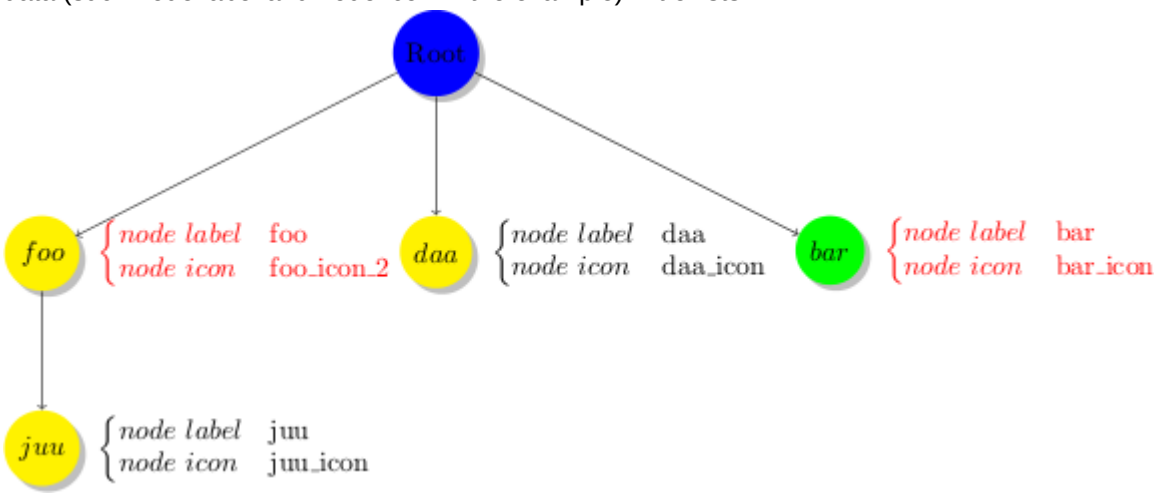


- *Case 2: Import mode is* `INSERT`
  If a node does not exist, the importer will add new nodes to the navigation tree. You will see the following result:

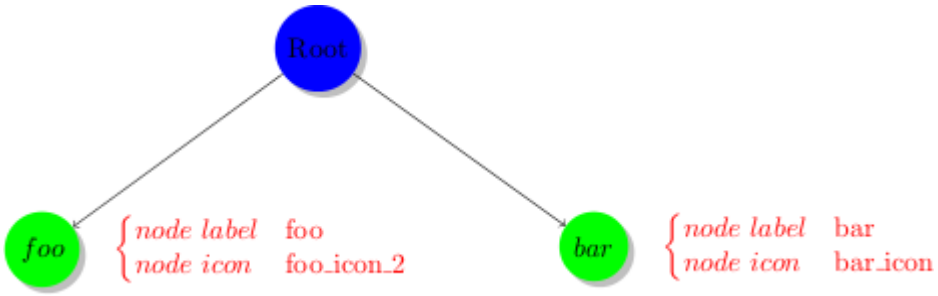


  Hereafter, the node 'bar' is added to the navigation tree, because it does not exist in the initiated data. Other nodes are kept in the import process.

- *Case 3: Import mode is* `MERGE`
  The `MERGE` mode indicates that a new node is added to the navigation tree, and updates the node data (such node label and node icon in the example) if it exists.



- *Case 4: Import mode is* `OVERWRITE`
  Everything will be destroyed and replaced with new data if the `OVERWRITE` mode is used.



Portal Config

PortalConfig defines the portal name, permission, layout and some properties of a site. These information are configured in the *portal.xml*, *group.xml* or *user.xml*, depending on the site type. The PortalConfig importer performs a strategy that is based on the mode defined in NewPortalConfigListener, including `CONSERVE`, `INSERT`, `MERGE` or `OVERWRITE`. Let's see how the import mode affects in the process of portal data performance:

- `CONSERVE`: There is nothing to be imported. The existing data will be kept without any changes.
- `INSERT`: When the portal config does not exist, create the new portal defined by the portal config definition. Otherwise, do nothing.
- `MERGE` and `OVERWRITE`: The same behavior. The new portal config will be created if it does not exist or update portal properties defined by the portal config definition.

Page Data

The import mode affects the page data import as the same as Portal Config.

If the Import mode is `CONSERVE` or `INSERT`, the data import strategy always performs as the `MERGE` mode in the first data initialization of the Portal.

2.7 Internationalization Configuration

Assumed Knowledge

GateIn Portal 3.5 is fully configurable for internationalization; however, users should have a general knowledge of Internationalization in Java products before attempting these configurations.

Sun Java hosts a comprehensive guide to internationalize Java products at http://java.sun.com/docs/books/tutorial/i18n/TOC.html.

All GateIn Portal 3.5 applications contain property files for various languages. They are packaged with the portlets applications in a `WEB-INF/classes/locale/` directory.

These files are located in the `classes` folder of the WEB-INF directory to be loaded by the class loader.

All resource files are in a subfolder named `locale`.

For example, the translations for the *NavigationPortlet* are located in `web.war/WEB-INF/classes/locale/portlet/portal`.

```
NavigationPortlet_de.properties
NavigationPortlet_en.properties
NavigationPortlet_es.properties
NavigationPortlet_fr.properties
NavigationPortlet_nl.properties
NavigationPortlet_ru.properties
NavigationPortlet_uk.properties
NavigationPortlet_ar.xml
```

Those files contain typical `key=value` Java EE properties. For example, the French one:

```
javax.portlet.title=Portlet Navigation
```

There are also properties files in the portal itself. They form the *portal resource bundle*.

From a portlet, you can then access translations from the portlet itself or shared at the portal level, both are aggregated when you need them.

Translation in XML format

It is also possible to use a proprietary XML format to define translations. This is a more convenient way for languages, such as Japanese, Arabic or Russian. Property files have to be ISO 8859-1 encoded with Unicode escape sequences, while the XML file can define its encoding. As a result, it is easier for you to read or edit a translation in XML instead of handling the Unicode escape sequences in property files.

For more information, refer to XML Resources Bundles.

2.7.1 Locales configuration

Various languages are available in the portal package. The configuration below will define which languages shown in the "Change Language" section and made available to users.

The `portal.war:/WEB-INF/conf/common/common-configuration.xml` file of your installation contains the following section:

```
<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>
```

This configuration points to the locale configuration file (`portal.war:/WEB-INF/conf/common/locales-config.xml`) which contains the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<locales-config>
  <locale-config>
    <locale>en</locale>
    <output-encoding>UTF-8</output-encoding>
    <input-encoding>UTF-8</input-encoding>
    <description>Default configuration for english locale</description>
  </locale-config>

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

  <locale-config>
    <locale>ar</locale>
    <output-encoding>UTF-8</output-encoding>
    <input-encoding>UTF-8</input-encoding>
    <description>Default configuration for the arabic locale</description>
    <orientation>rt</orientation>
  </locale-config>
</locales-config>
```

`locale`: This has to be defined, such as http://ftp.ics.uci.edu-pub-ietf-http-related-iso639.txt. In this example, "*ar*" is Arabic.

`output-encoding`: This deals with the character encoding. It is recommended that *UTF-8* be used.

`input-encoding`: In the Java implementation, the encoding parameters will be used for the request response stream. The input-encoding parameter will be used for requesting setCharacterEncoding(..).

`description`: Brief description of the language.

`orientation`: The default orientation of text and images is Left-To-Right. GateIn Portal 3.5 supports *Right-To-Left* orientation. Modifying the text orientation is explained in RTL (Right To Left) Framework.

2.7.2 ResourceBundleService

The resource bundle service is configured in `portal.war:/WEB-INF/conf/common/common-configuration.xml`.

```
<component>
  <key>org.exoplatform.services.resources.ResourceBundleService</key>
  <type>org.exoplatform.services.resources.impl.SimpleResourceBundleService</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>Initiate the following resources during the first launch</description>
      <value>locale.portal.expression</value>
      <value>locale.portal.services</value>
      <value>locale.portal.webui</value>
      <value>locale.portal.custom</value>
      <value>locale.navigation.portal.classic</value>
      <value>locale.navigation.group.platform.administrators</value>
      <value>locale.navigation.group.platform.users</value>
      <value>locale.navigation.group.platform.guests</value>
      <value>locale.navigation.group.organization.management.executive-board</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.expression</value>
      <value>locale.portal.services</value>
      <value>locale.portal.webui</value>
      <value>locale.portal.custom</value>
    </values-param>
  </init-params>
</component>
```

`classpath.resources`: This is discussed in the later section.

`init.resources`: Initiate resources related to portal, group, user resource bundles.

`portal.resource.names`: Define all resources that belong to the *Portal Resource Bundle*.

These resources are merged into a single resource bundle which is accessible from anywhere in GateIn Portal 3.5. All these keys are located in the same bundle, which is separated from the navigation resource bundles.

2.7.3 Navigation Resource Bundles

There is a resource bundle for each navigation. A navigation can exist for user, groups and portal.

The previous example shows bundle definitions for the navigation of the classic portal and of four different groups. Each of these resource bundles occupies a different sphere, they are independent of each other and they are not included in the `portal.resource.names` parameter.

The properties for a group must be in the `WEB-INF/classes/locale/navigation/group/` folder. For example, `/WEB-INF/classes/locale/navigation/group/organization/management/executive-board_` .

The folder and file names must correspond to the group hierarchy. The group name "`executive-board`" is followed by the ISO 639 code.

Each language defined in `LocalesConfig` must have a resource file defined. If the name of a group is changed the name of the folder and/or files of the correspondent navigation resource bundles must also be changed.

Content of `executive-board_en.properties`:

```
organization.title=Organization
organization.newstaff=New Staff
organization.management=Management
```

This resource bundle is only accessible for the navigation of the `organization.management.executive-board` group.

2.7.4 Portlets

Portlets are independent applications and deliver their own resource files.

All shipped portlet resources are located in the *locale/portlet* subfolder. The `ResourceBundleService` parameter called *classpath.resources* defines this subfolder.

Add a Spanish translation to the GadgetPortlet

1. Create the `GadgetPortlet_es.properties` file in `WEB-INF/classes/locale/portlet/gadget/GadgetPortlet`.
2. Add `Spanish` as a *supported-locale* to `portlet.xml` ('es' is the 2 letters code for Spanish). The *resource-bundle* is already declared and is the same for all languages:

   ```
   <supported-locale>en</supported-locale>
   <supported-locale>es</supported-locale>
   <resource-bundle>locale.portlet.gadget.GadgetPortlet</resource-bundle>
   ```

   See the portlet specification for more details about the portlet internationalization.

Standard portlet resource keys

The portlet specifications define three standard keys: Title, Short Title and Keywords. Keywords are formatted as a comma-separated list of tags.

```
javax.portlet.title=Breadcrumbs Portlet
 javax.portlet.short-title=Breadcrumbs
 javax.portlet.keywords=Breadcrumbs, Breadcrumb
```

Debugging resource bundle usage

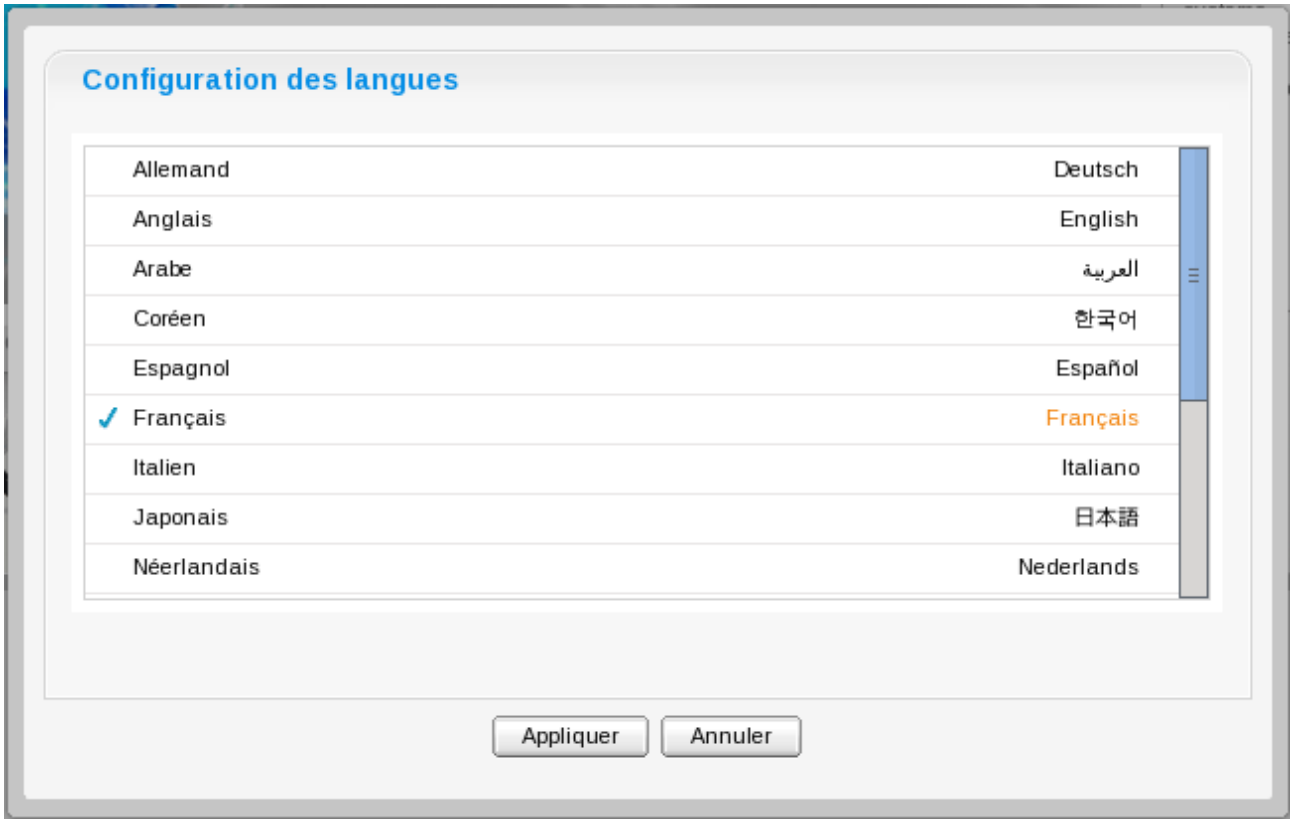
When translating an application, it can sometimes be importamt to find out which key underlies some given given label in the user interface. GateIn Portal offers *Magic locale* to handle such situations.

With *Magic locale* the portal visitor sees the keys themselves in the graphical user interface. For example, the translated value of the "`organization.title`" key is "`organization.title`" itself.

To be able to select *Magic locale* from the list of available locales, you need to start the portal in development mode. This can be done through setting the java system property `exo.product.developing` to `true`. For example, the property can be passed as a JVM parameter on portal start up:

```
$JBOSS_HOME/bin/standalone.sh -Dexo.product.developing=true
```

2.7.5 Translate the language selection form



When choosing a language as on the screenshot above, the user is presented with a list of languages on the left side in the current chosen language and on the right side, the same language translated into its own language. Those texts are obtained from the JDK API `java.util.Locale.getDisplayedLanguage()` and `java.util.Locale.getDisplayedCountry()` (if needed) and all languages may not be translated and can also depend on the JVM currently used. It is still possible to override those values by editing the `locale.portal.webui` resource bundle, to do so edit the `gatein.ear/portal.war/WEB-INF/classes/locale/portal/webui_xx_yy.properties` file where *xx_yy* represents the country code of the language in which you want to translate a particular language. In that file, add or modify a key such as `Locale.xx_yy` with the value being the translated string.

First, edit `gatein.ear/portal.war/WEB-INF/classes/locale/portal/webui_fr.properties` where *ne* is the country code for French, and add the following key into it:

```
Locale.zh_TW=Chinois traditionnel
```

After a restart, the language will be updated in the user interface when a user is trying to change the current language.

2.8 Pluggable Locale Policy

Every request processed by every portlet is invoked within a context of current `Locale`. Current `Locale` can be retrieved by calling the `getLocale()` method of the `javax.portlet.PortletRequest` interface.

The exact algorithm for determining the current `Locale` is not specified by Portlet Specification, and is left to portlet containers to implement the way they deem most appropriate.

In GateIn 3.2, each portal instance has a default language which can be used to present content for new users. Another option is to use each user's browser language preference, provided it matches one of the available localizations that GateIn 3.2 supports, and only fallback to portal default language if no match is found. Every user, while visiting a portal, has an option to change the language of the user interface by using a Language chooser. The choice can be remembered for the duration of the session, or it can be remembered for a longer period using a browser cookie, or - for registered and logged-in users - it can be saved into user's profile.

So, we can see that there is more than one way to determine the `Locale` to be used for displaying a portal page to the user. For this reason the mechanism for determining the current `Locale` of the request is pluggable in GateIn 3.2, so the exact algorithm can be customized.

2.8.1 LocalePolicy API

Customization is achieved by using LocalePolicy API, which is a simple API consisting of one interface, and one class:

- `org.exoplatform.services.resources.LocalePolicy` interface
- `org.exoplatform.services.resources.LocaleContextInfo` class
  The `LocalePolicy` interface defines a single method that is invoked on the installed `LocalePolicy` service implementation:

```
public interface LocalePolicy
{
   public Locale determineLocale(LocaleContextInfo localeContext);
}
```

`Locale` returned by the `determineLocale()` method is `Locale` that will be returned to portlets when they call `javax.portlet.PortletRequest.getLocale()` method.

The returned `Locale` has to be one of the locales supported by portal, otherwise it will fallback to portal-default `Locale`.

The supported locales are listed in `gatein.ear/portal.war/WEB-INF/conf/common/locales-config.xml` file as described in sect-Reference_Guide-Internationalization_Configuration-Locales_configuration.

The `determineLocale()` method takes a parameter of `LocaleContextInfo` type, which represents a compilation of preferred locales from different sources, including user's profile, portal default, browser language settings, current session, browser cookie, and more. All these different sources of `Locale` configuration or preference are used as input to the `LocalePolicy` implementation that decides which `Locale` should be used.

2.8.2 Default LocalePolicy

By default, `org.exoplatform.portal.application.localization.DefaultLocalePolicyService`- an implementation of `LocalePolicy`- is installed to provide the default behaviour. This, however, can easily be extended and overridden. A completely new implementation can also be written from scratch.

`DefaultLocalePolicyService` treats logged-in users slightly differently than anonymous users. Logged-in users have a profile that can contain language preference, while anonymous users do not.

Here is an algorithm used for anonymous users.

An algorithm for anonymous users

1. **
   Iterate over the `LocaleContextInfo` properties in the following order:
   - `cookieLocales`
   - `sessionLocale`
   - `browserLocales`
   - `portalLocale`
2. **
   Get each property's value - if it's a collection, get the first value.

- If value is one of the supported locales return it as a result.
- If value is not in the supported locales set, try to remove country information, and check if a language matching locale is in the list of supported locales. If so, return it as a result.

1. **
   Otherwise, continue with the next property.
   If no supported locale is found, the return locale eventually defaults to `portalLocale`.

The algorithm for logged-in users is virtually the same except that the first `Locale` source checked is user's profile.

An algorithm for logged-in users

1. **
   Iterate over `LocaleContextInfo` properties in the following order:
   - `userProfile`
   - `cookieLocales`
   - `sessionLocale`
   - `browserLocales`
   - `portalLocale`
2. **
   Do the next steps which are the same as for anonymous users.

2.8.3 Custom LocalePolicy

The easiest way to customize the `LocalePolicy` is to extend `DefaultLocalePolicyService`. The study of its source code will be required. There is sample JavaDoc that provides thorough information. Most customizations will involve simply overriding one or more of its protected methods.

An example of a customization is an already provided `NoBrowserLocalePolicyService`. By overriding just one method, it skips any use of browser language preference.

```
public class NoBrowserLocalePolicyService extends DefaultLocalePolicyService
{
   /**
    * Override super method with no-op.
    *
    * @param context locale context info available to implementations in order to determine
appropriate Locale
    * @return null
    */
   @Override
   protected Locale getLocaleConfigFromBrowser(LocaleContextInfo context)
   {
      return null;
   }
}
```

2.8.4 LocalePolicy Configuration

The `LocalePolicy` framework is enabled for portlets by configuring the `LocalizationLifecycle` class in portal's webui configuration file: `gatein.ear/portal.war/WEB-INF/webui-configuration.xml`.

```
<application-lifecycle-listeners>
...
 <listener>org.exoplatform.portal.application.localization.LocalizationLifecycle</listener>
</application-lifecycle-listeners>
```

The default `LocalePolicy` implementation is installed as GateIn Kernel portal service via `gatein.ear/portal.war/WEB-INF/conf/portal/web-configuration.xml`. Here you can change it to a different value according to your needs.

The following fragment is responsible for installing the service:

```
<component>
   <key>org.exoplatform.services.resources.LocalePolicy</key>
   <type>org.exoplatform.portal.application.localization.DefaultLocalePolicyService</type>
</component>
```

Besides implementing `LocalePolicy`, the service class also needs to implement `org.picocontainer.Startable` interface in order to get installed.

2.8.5 Keeping non-bridged resources in synchronization with current Locale

In portals, all the resources that are not portlets themselves but are accessed through portlets - reading data through `PortletRequest`, and writing to `PortletResponse`- are referred to as 'bridged'. Any resources that are accessed directly, bypassing portal filters and servlets, are referred to as 'non-bridged'.

Non-bridged servlets, and .jsps have no access to `PortalRequest`. They do not use `PortletRequest.getLocale()` to determine current `Locale`. Instead, they use `ServletRequest.getLocale()` which is subject to precise semantics defined by Servlet specification - it reflects browser's language preference.

In other words, non-bridged resources do not have a notion of current `Locale` in the same sense that portlets do. The result is that when mixing portlets and non-bridged resources there may be a localization mismatch - an inconsistency in the language used by different resources composing your portal page.

This problem is addressed by `LocalizationFilter`. This is a filter that changes the behaviour of `ServletRequest.getLocale()` method so that it behaves the same way as `PortletRequest.getLocale()`. That way even localization of servlets, and .jsps accessed in a non-bridged manner can stay in synchronization with the portlet localization.

`LocalizationFilter` is installed through the `gatein.ear/portal.war/WEB-INF/web.xml` file of the portal.

```
<filter>
 <filter-name>LocalizationFilter</filter-name>
 <filter-class>org.exoplatform.portal.application.localization.LocalizationFilter</filter-class>
</filter>
...

<filter-mapping>
   <filter-name>LocalizationFilter</filter-name>
   <url-pattern>*.jsp</url-pattern>
   <dispatcher>INCLUDE</dispatcher>
   <dispatcher>FORWARD</dispatcher>
   <dispatcher>REQUEST</dispatcher>
   <dispatcher>ERROR</dispatcher>
</filter-mapping>
```

One tiny limitation to this mechanism is that it is unable to determine the current portal, and consequently its default language. As a result, the `portalLocale` defaults to `English`, but can be configured to something else by using filter's `PortalLocale` init param. For example:

```
<filter>
 <filter-name>LocalizationFilter</filter-name>
 <filter-class>org.exoplatform.portal.application.localization.LocalizationFilter</filter-class>
 <init-param>
    <param-name>PortalLocale</param-name>
    <param-value>fr_FR</param-value>
 </init-param>
</filter>
```

By default, `LocalizationFilter` is applied to *.jsp, which is considered the minimum required by GateIn 3.2 to properly keep its non-bridged resources in synchronization with the rest of the portal. Additionally, deployed portlets and portal applications may need broader mapping to cover their non-bridged resources.

Avoid using `/`, `/public/`, `/private/*`, and similar broad mappings as `LocalizationFilter` sometimes adversely interacts with the processing of portlet requests. Use multiple filter-mappings instead to specifically target non-bridged resources.

Keeping the mapping limited to only non-bridged resources will minimize any impact on performance as well.

2.9 RTL (Right To Left) Framework

The text orientation depends on the current locale setting. The orientation is a Java 5 enum that provides a set of functionalities:

```
LT, // Western Europe
RT, // Middle East (Arabic, Hebrew)
TL, // Japanese, Chinese, Korean
TR; // Mongolian
public boolean isLT() { ... }
public boolean isRT() { ... }
public boolean isTL() { ... }
public boolean isTR() { ... }
```

The object defining the Orientation for the current request is the `UIPortalApplication`. However, it should be accessed at runtime using the `RequestContext` that delegates to the `UIPortalApplication`.

In case of `PortalRequestContext`, it directly delegates as the `PortalRequestContext` that has a reference to the current `UIPortalApplication`.

In case of a different context, such as the `PortletRequestContext`, it delegates to the parent context given the fact that the root `RequestContext` is always a `PortalRequestContext`.

2.9.1 Groovy templates

Orientation is defined by implicit variables in the Groovy binding context:

- *Orientation*

  > *The current orientation as an Orientation.*

- *isLT*

  > *The value of orientation.isLT().*

- *isRT*

  > *The value of orientation.isRT().*

- *dir*

  > *The string 'ltr' if the orientation is LT or the string 'rtl' if the orientation is RT.*

2.9.2 Stylesheet

The skin service handles stylesheet rewriting to accommodate the orientation. It works by appending -lt or -rt to the stylesheet name.

For instance: `/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.

In this example, the orientation is used 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;
```

In this example, you need to modify the padding according to 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 */
```

2.9.3 Images

Sometimes, it is necessary to create the RT version of an image that will be used from a template or from a stylesheet. However, symmetric images can be automatically generated avoiding the necessity to create a mirrored version of an image and furthermore avoiding maintenance cost.

The web resource filter uses the same naming pattern as the skin service. When an image ends with the -rt suffix, the portal will attempt to locate the original image and create a mirror of it. For example, when `/GateInResources/skin/DefaultSkin/webui/component/UITabSystem/UITabs/background/I` is requested, a mirror of `/GateInResources/skin/DefaultSkin/webui/component/UITabSystem/UITabs/background/I` is returned.

It is important to consider whether the image to be mirrored is symmetrical as this will impact its final appearance.

Combine stylesheet and images:

```
line-height: 24px;
background: url('background/NavigationTab.gif') no-repeat right top; /* orientation=lt */
background: url('background/NavigationTab-rt.gif') no-repeat left top; /* orientation=rt */
padding-right: 2px; /* orientation=lt */
padding-left: 2px; /* orientation=rt */
```

2.9.4 Client side JavaScript

The `eXo.core.I18n` object provides the following parameters for orientation:

- `getOrientation()`

  > *Return either the string lt or rt.*

- `getDir()`

  > *Return either the string ltr or rtl.*

- `isLT()`

  > *Return true for LT.*

- `isRT()`

  > *Return true of RT.*

2.10 XML Resources Bundles

Resource bundles are usually stored in property files. However, as property files are plain files, issues with the encoding of the file may arise. The XML resource bundle format has been developed to provide an alternative to property files.

- The XML format declares the encoding of the file. This avoids use of the native2ascii program which can interfere with encoding.
- Property files generally use the ISO 8859-1 character encoding which does not cover the full unicode charset. As a result, languages, such as Arabic, would not be natively supported.
- Tooling for XML files is better supported than the tooling for Java property files; thus, the XML editor copes well with the file encoding.

2.10.1 XML format

The XML format is very simple and has been developed based on the *DRY* (Don't Repeat Yourself) principle. The resource bundle keys are hierarchically defined and we can leverage the hierarchic nature of the XML for that purpose. Here is an example of turning a property file into an XML resource bundle file:

```
UIAccountForm.tab.label.AccountInputSet = ...
UIAccountForm.tab.label.UIUserProfileInputSet = ...
UIAccountForm.label.Profile = ...
UIAccountForm.label.HomeInfo= ...
UIAccountForm.label.BusinessInfo= ...
UIAccountForm.label.password= ...
UIAccountForm.label.Confirmpassword= ...
UIAccountForm.label.email= ...
UIAccountForm.action.Reset= ...
```

```
<?xml version="1.0" encoding="UTF-8"?>
<bundle>
  <UIAccountForm>
    <tab>
      <label>
        <AccountInputSet>...</AccountInputSet>
        <UIUserProfileInputSet>...</UIUserProfileInputSet>
      </label>
    </tab>
    <label>
      <Profile>...</Profile>
      <HomeInfo>...</HomeInfo>
      <BusinessInfo>...</BusinessInfo>
      <password>...</password>
      <Confirmpassword>...</Confirmpassword>
      <email>...</email>
    </label>
    <action>
      <Reset>...</Reset>
    </action>
  </UIAccountForm>
</bundle>
```

2.10.2 Portal support

To be loaded by the portal at runtime (actually the resource bundle service), the name of the file must be the same as a property file and it must use the *.xml* suffix.

For example, for the Account Portlet to be displayed in Arabic, the resource bundle would be _ AccountPortlet_ar.xml_ rather than *AccountPortlet_ar.properties*.

2.11 Upload Component

In this section, you will learn how to configure the *Upload service* that is defined by the `org.exoplatform.upload.UploadService` class.

This can be configured with the following XML code:

```
<component>
     <type>org.exoplatform.upload.UploadService</type>
     <init-params>
       <value-param>
         <name>upload.limit.size</name>
         <description>Maximum size of the file to upload in MB</description>
         <value>10</value>
       </value-param>
     </init-params>
   </component>
```

This code allows uploading files with the default size limit (10MB). The default value unit is in Megabytes.

This limitation will be used by default by all applications if no application-specific limit is set. Setting a different limit for applications is discussed in a later section.

If the value is set to `0`, the upload size is unlimited.

2.11.1 Use the upload component

1. Create an `org.exoplatform.webui.form.input.UIUploadInput` object type by using one of three following constructors:
    - The default constructor that allows uploading the file with the size of 10 MB.

      ```
      public UIUploadInput(String name, String bindingExpression, int limitFile)
      ```

    - This constructor allows you to customize the size limit of uploaded files by using the `limitSize` parameter. The default value unit is in Megabytes.

      ```
      public UIUploadInput(String name, String bindingExpression,int limitFile, int
      limitSize)
      ```

    - This constructor allows you to customize the size limit and the value unit by using the `limitSize` and `unit` parameters respectively.
      In GateIn, you can set the value unit to Megabytes (MB), Kilobytes (KB) or Gigabytes (GB).

      ```
      public UIUploadInput(String name, String bindingExpression, int limitFile, int
      limitSize, UploadUnit unit)
      ```

      The following is an example using the third form:

      ```
      PortletRequestContext pcontext =
      (PortletRequestContext)WebuiRequestContext.getCurrentInstance();
      PortletPreferences portletPref = pcontext.getRequest().getPreferences();
      int limitFile = Integer.parseInt(portletPref.getValue("uploadFileLimit",
      "1").trim());
      int limitSize = Integer.parseInt(portletPref.getValue("uploadFileSizeLimit",
      "").trim());
      UploadUnit limitUnit =
      UploadUnit.valueOf(portletPref.getValue("uploadFileLimitUnit", "MB").trim());
      UIUploadInput uiInput = new UIUploadInput("upload", "upload", limitFile, limitSize,
      limitUnit);
      ```

2. Obtain the limit from the XML configuration by adding the following code to either `portlet.xml` or `portlet-preferences.xml`:

```
<! The number of files are uploaded -->
 <preference>
  <name>uploadFileLimit</name>
  <value>3</value>
  <read-only>false</read-only>
 </preference>
<! The size limit -->
 <preference>
  <name>uploadFileSizeLimit</name>
  <value>300</value>
  <read-only>false</read-only>
 </preference>
<! The unit limit -->
 <preference>
  <name>uploadFileLimitUnit</name>
  <value>KB</value>
  <read-only>false</read-only>
 </preference>
```

The `0` value means the upload size is unlimited, and the value unit is set to MegaBytes.

3. Use the `getUploadDataAsStream()` method to get the uploaded data:

```
UIUploadInput input = (UIUploadInput)uiForm.getUIInput("upload");
InputStream[] inputStreams = input.getUploadDataAsStreams();
...
```

The upload service stores a temporary file on the file system during the upload process. When the upload is finished, the service must be cleaned to:

- Delete the temporary file.
- Delete the classes used for the upload.

4. Use the `removeUploadResource(String uploadId)` method defined in the upload service to purge the file:

```
UploadService uploadService = uiForm.getApplicationComponent(UploadService.class);
UIUploadInput uiChild = uiForm.getChild(UIFormUploadInput.class);
for(String uploadId : uiChild.getUploadIds()) {
    uploadService.removeUpload(uploadId);
}
```

Saving the uploaded file

Ensure the file is saved *before* the service is cleaned.

2.12 Deactivation of the Ajax Loading Mask Layer

The loading mask layer is deployed after an ajax-call. It aims at blocking the GUI to prevent further user actions until the the ajax-request has been completed.

However, the mask layer may need to be deactivated in instances where the portal requires user instructions before the previous instructions have been carried out.

Deactivate the ajax-loading mask

Simply enerate a script to make an asynchronous ajax-call. Use the `uicomponent.doAsync()` method rather than the `uicomponent.event()` method. For example:

```
<a href="<%=uicomponent.doAsync(action, beanId, params)%>" alt="">Asynchronous</a>
```

- The `doAsync()` method automatically adds the following new parameter to the parameters list: `asyncparam = new Parameter(AJAX_ASYNC,"true"); (AJAX_ASYNC == "ajax_async")` .
- This request is asynchronous and the ajax-loading mask will not deployed.

> ⚠ An asynchronous request can still be made using the `uicomponent.event()`. When using this method, the asyncparam must be added manually.

The GUI will be blocked to ensure that the user can only request one action at one time and while the request seems to be synchronous, all ajax requests are always asynchronous. For further information, refer to sect-Reference_Guide-Deactivation_of_the_Ajax_Loading_Mask_Layer_-Synchronous_issue .

2.12.1 Synchronous issue

Most web browsers support ajax requests in two modes: *Synchronous* and *Asynchronous*. This mode is specified with a boolean `bAsync` parameter.

```
var bAsync = false; // Synchronous
request.open(instance.method, instance.url, bAsync);
```

However, to work with browsers that do not support the *Synchronous* requests, `bAsync` is always set to `true` (The Ajax request will always be asynchronous).

```
// Asynchronous request
request.open(instance.method, instance.url, true);
```

2.13 Javascript Configuration

Managing Javascript in an application like GateIn 3.2 is a critical part of the configuration work. Configuring the scripts correctly will result in the faster response time from the portal.

Every portlet can have its own Javascript code but in many cases, it is more convenient to reuse some existing shared libraries. For that reason, GateIn 3.2 has a mechanism to easily register the libraries that will be loaded when every page is rendered.

To do so, every WAR deployed in GateIn 3.2 can register the `.js` files with the `gatein-resources.xml` configuration file.

The example code snippet below is found in `gatein-resources.xml` of `eXoResources.war` .

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

<!-- CORE Javascripts -->
<javascript>
  <param>
    <js-module>eXo.core.Utils</js-module>
    <js-path>/javascript/eXo/core/Util.js</js-path>
    <js-priority>1</js-priority>
  </param>
  <param>
    <js-module>eXo.core.DOMUtil</js-module>
    <js-path>/javascript/eXo/core/DOMUtil.js</js-path>
    <js-priority>1</js-priority>
  </param>
  <param>
    <js-module>eXo.core.Browser</js-module>
    <js-path>/javascript/eXo/core/Browser.js</js-path>
    <js-priority>2</js-priority>
  </param>
  <param>
    <js-module>eXo.core.MouseEventManager</js-module>
    <js-path>/javascript/eXo/core/MouseEventManager.js</js-path>
  </param>
  <param>
    <js-module>eXo.core.UIMaskLayer</js-module>
    <js-path>/javascript/eXo/core/UIMaskLayer.js</js-path>
  </param>
  <param>
    <js-module>eXo.core.Skin</js-module>
    <js-path>/javascript/eXo/core/Skin.js</js-path>
  </param>
  <param>
    <js-module>eXo.core.DragDrop</js-module>
    <js-path>/javascript/eXo/core/DragDrop.js</js-path>
  </param>
  <param>
    <js-module>eXo.core.DragDrop2</js-module>
    <js-path>/javascript/eXo/core/DragDrop2.js</js-path>
  </param>
</javascript>
```

Note that registered Javascript files will be merged into a single `merged.js` file when the server loads. This reduces the number of HTTP calls as shown in the homepage source code:

```
<script type="text/javascript" src="/portal/javascript/merged.js"></script>
```

Although this optimization is useful for a production environment, it may be easier to deactivate this optimization while debugging Javascript problems.

To do this, set the Java system property `exo.product.developing` to `true`. GateIn provides two startup scripts that define this property in `gatein-dev.sh` (for Linux, Mac) and `gatein-dev.bat` (for Windows).

To generate the `merged.js` file, set this property to `false`. If the property is not set, the default value is false.

The property can be passed as a JVM parameter with the *-D* option in your `gatein.sh` or `gatein.bat` startup script.

Every Javascript file is associated with a module name which acts as a namespace.

Inside the associated Javascript files, the eXo Javascript objects are exposed as global variables named after the module.

For example:

```
eXo.core.DragDrop = new DragDrop();
```

It is also possible to use the `eXo.require()` method to lazy load and evaluate some Javascript codes. This is quite useful for the portlet or gadget applications that will use this Javascript only once. Otherwise, if the library is reusable in several places, it is better to define it in the `gatein-resources.xml` file.

2.14 Navigation Controller

The navigation controller is a major enhancement of GateIn that has several goals:

- Provide non-ambiguous URLs for resources managed by the portal, such as navigation. Previously, different resources were possible for a single URL, even worse, the set of resources available for an URL depends on private navigations (groups and dashboard).
- Decouple the HTTP request from the portal request. Previously, both were tightly coupled, for instance, the URL for a site had to begin with */public/{sitename}* or */private/{sitename}* .The navigation controller provides a flexible and configurable mapping.
- Provide a more friendly URL and let portal administrator configure how the HTTP request should look like.

2.14.1 Controller in Action

Controller

The `WebAppController` is the component of GateIn that processes HTTP invocations and transforms them into a portal request. It has been improved with the addition of a request mapping engine (*controller*) whose role is to make the HTTP request decouple and create a portal request. The mapping engine makes two essential tasks:

- Create a `Map<QualifiedName, String>` from an incoming HTTP request.
- Render a `Map<QualifiedName, String>` as an HTTP URL.
  The goal of the controller (mapping engine) is to *decouple* the request processed by GateIn from the incoming HTTP request. Indeed, a request contains data that determine how the request will be processed and such data can be encoded in various places in the request, such as the request path, or a query parameter. The controller allows GateIn to route a request according to a set of parameters (a map) instead of the servlet request.

The controller configuration is declarative in an `.xml` file named `controller.xml`, allowing easy reconfiguration of the routing table and it is processed into an internal data structure that is used to perform resolution (routing or rendering).

The controller data cannot be modified by using the portlet interface, but can be still changed at runtime by modifying in the `controller.xml` file, then calling the `WebAppController.reloadConfiguration()` method.

Building controller

The controller configuration that contains the routing rules is loaded from the `controller.xml` file retrieved in the GateIn configuration directory. Its location is determined by the `gatein.controller.config` property.

`WebAppController` loads and initializes the mapping engine.

```
<!-- conf/portal/controller-configuration.xml of portal.war -->
<component>
  <type>org.exoplatform.web.WebAppController</type>
    <init-params>
      <value-param>
        <name>controller.config</name>
        <value>${gatein.portal.controller.config}</value>
      </value-param>
    </init-params>
</component>
```

GateIn's extension project can define its own routing table, thanks to the extension mechanism.

The `controller.xml` file can be changed and reloaded at runtime. This helps the test of different configurations easily (configuration loading operations) and provides more insight into the routing engine (the findRoutes operation). See *Rebuiding controller* below for more details.

- *ReBuilding controller*
  `WebAppController` is annotated with the `@Managed` annotations and is bound under the `view=portal,service=controller` JMX name and under the "portalcontroller" REST name.

It provides the following attributes and operations:

- Attribute configurationPath: the "read-only" configuration path of the `controller.xml` file.
- Operation loadConfiguration: load a new configuration file from a specified XML path.
- Operation reloadConfiguration: reload the configuration file.
- Operation findRoutes: route the request argument through the controller and returns a list of all parameter map resolutions. The argument is a request URI, such as */g/:platform:administrators/administration/registry*. It returns a string representation (`List<Map>`) of the matched routes.

Controller Configuration (controller.xml)

Most of the controller configuration defines rules (Routing table - contains routes object) that will drive the resolution. Routes are processed during the controller initialization to give a tree of node.

- Each node is related to its parent with a matching rule that can either be an *exact string matching* or a *regular expression matching*.
- Each node is associated with a set of parameters.
  A parameter is defined by a qualified name and there are three kinds of parameters explained in the sections below.

Route parameters

Route parameters define a fixed value which is associated with a qualified name.

- *Routing*: Route parameters which allow the controller to distinguish branches easily and route the request accordingly.
- *Rendering*: The system will select a route to render an URL if all route parameters are always matched.

```
<route path="/foo">
  <route-param qname="gtn:handler">
    <value>portal</value>
  </route-param>
</route>
```

This configuration matches the "*/foo*" request path to the map (gtn:handler=portal). Conversely, it renders the map (gtn:handler=portal) as the "*/foo*" URL. This example shows two concepts:

- Exact path matching ("/foo").
- Route parameters ("gtn:handler").

Path parameters - _Regular expression support _

Path parameters allow associating a portion of the request path with a parameter. Such parameter will match any non empty portions of text except the / character (that is the [^/]+ regular expression). Otherwise, they can be associated with a regular expression for matching specific patterns. Path parameters are mandatory for matching since they are a part of the request path, however it is allowed to write regular expression matching an empty value.

- Routing: Route is accepted if the regular expression is matched.
- Rendering: The system will select a route to render an URL if all route parameters are always matched.
  Encoding

Path parameters may contain the '/' character which is a reserved char for the URI path. This case is specially handled by the navigation controller by using a special character to replace the '/' literals. By default, the character is the colon (*":"*) and can be changed to other possible values (see controller XML schema for possible values) to give a greater amount of flexibility.

This encoding is applied only when the encoding is performed for parameters having a mode set to the `default-form` value, for instance, it does not happen for navigation node URI (for which / are encoded literally). The separator escape char can still be used but under it is percent escaped form, so by default, a path parameter value containing the colon *":"* would be encoded as %3A and conversely the %3A value will be decoded as the colon *":"*.

```
<route path="/{gtn:path}">
</route>
```

As a result, routing and rendering are as below:

```
Routing and Rendering
Path "/foo"      <--> the map (gtn:path=foo)

Path "/foo:bar"  <--> the map (gtn:path=foo/bar)
```

If the request path contains another "/" char, it will not work. The default encoding mode is `default-form`. In the example above, "/foo/bar" is not matched, so the system returns an empty parameter map.

However, this problem could be solved with the following configuration:

```
<route path="/{gtn:path}">
  <path-param encoding="preserve-path" qname="gtn:path">
    <pattern>.*</pattern>
  </path-param>
</route>
```

- The ".*" declaration allows matching any char sequence.
- The "*preserve-path*" encoding tells the engine that the "/" chars should be handled by the path parameter itself as they have a special meaning for the router. Without this special encoding, "/" would be rendered as the ":" character and conversely the ":" character would be matched as the "/" character.

Request parameters

Request parameters are matched from the request parameters (GET or POST). The match can be optional as their representation in the request allows it.

- *Routing*:
  - Route is accepted when a required parameter is present and matched in the request.
  - Route is accepted when an optional parameter is absent or matched in the request.
- *Rendering*:
  - For required parameters, the system will select a route to render an URL when the parameter is present and matched in the map.
  - For optional parameters, the system will select a route to render an URL when the parameter is absent or matched in the map.

```
<route path="/">
  <request-param name="path" qname="gtn:path"/>
</route>
```

Request parameters are declared by a `request-param` element and will match any value by default. A request like "/?path=foo" is mapped to the map (gtn:path=foo). The `name` attribute of the `request-param` tag defines the request parameter value. This element accepts more configuration:

- A `value` or a `pattern` element that is a child element used to match a constant or a pattern.
- A `control-mode` attribute with the `optional` or `required` value indicates if matching is mandatory or not.
- A `value-mapping` attribute with the possible values, such as `canonical`, `never-empty`, `never-null` can be used to filter values after matching is done. For instance, a parameter configured with `value-mapping="never-empty"` and matched with the empty string value will not put the empty string in the map.

Route precedence

The order of route declaration is important as it affects how rules are matched. Sometimes, the same request could be matched by several routes and the routing table is ambiguous.

```
<route path="/foo">
  <route-param qname="gtn:handler">
    <value>portal</value>
  </route-param>
</route>
<route path="/{gtn:path}">
  <path-param encoding="preserve-path" qname="gtn:path">
    <pattern>.*</pattern>
  </path-param>
</route>
```

In that case, the request path "/foo" will always be matched by the first rule before the second rule. This can be misleading since the map (gtn:path=foo) would be rendered as "/foo" and would not be matched by the first rule. Such ambiguity can happen, it can be desirable or not.

Route nesting

Route nesting is possible and often desirable as it helps to:

- Factor common parameters in a common rule.
- Perform more efficient matching as the match of the common rule is done once for all the sub routes.

```
<route path="/foo">
  <route-param qname="gtn:handler">
    <value>portal</value>
  </route-param>
  <route path="/bar">
    <route-param qname="gtn:path">
      <value>bar</value>
    </route-param>
  </route>
  <route path="/juu">
    <route-param qname="gtn:path">
      <value>juu</value>
    </route-param>
  </route>
</route>
```

- The request path "/foo/bar" is mapped to the (gtn:handler=portal,gtn:path=bar) map.
- The request path "/foo/juu" is mapped to the (gtn:handler=portal,gtn:path=juu) map.
- The request path "/foo" is not mapped as non leaf routes do not perform matches.

2.14.2 Integrate to GateIn WebUI framework

Routing

GateIn defines a set of parameters in its routing table, for each client request, the mapping engine processes the request path and returns the defined parameters with their values as a `Map<QualifiedName, String>`.

gtn:handler

The `gtn:handler` name is one of the most important qualified names as it determines which handler will take care of the request processing just after the controller has determined the parameter map. The handler value is used to make a lookup in the handler map of the controller. The handler is a class that extends the `WebRequestHandler` class and implements the `execute(ControllerContext)` method. Several handlers are available by default:

- `portal`: Process aggregated portal requests.
- `upload/download`: Process file upload and download.
- `standalone`: Process standalone portal requests.
- `legacy`: Handle legacy URL redirection (see sect-Reference_Guide-Navigation_Controller-Legacy_handler).
- `default`: HTTP redirection to the default portal of the container.
- `staticResource`: Serve static resources like image, CSS or JavaScript and more in *portal.war* (see sect-Reference_Guide-Navigation_Controller-Static_resource_handler).
  gtn:sitetype / gtn:sitename / gtn:path

Those qualified names drives a request for the portal handler. They are used to determine which site to show and which path to resolve against a navigation. For instance, the (gtn:sitetype=portal,gtn:sitename=classic,gtn:path=home) instruct the portal handler to show the home page of the classic portal site.

gtn:lang

This parameter shows which language used in the URL for the portal handler. This is a new feature offered, now language can be specified on URL. It means that users can bookmark that URL (with the information about language) or he can changed the language simply by modifying the URL address.

gtn:componentid / gtn:action / gtn:objectid

The webui parameters used by the portal handler for managing webui component URLs for portal applications (but not for portlet applications).

Rendering

The *controller* is designed to render a Map<QualifiedName, String> as an HTTP URL according to its routing table. However, to integrate it for easy usage in WebUI Framework of GateIn, you need some more components:

- PortalURL
- NodeURL
- ComponentURL
- Portlet URLs
- Webui URLBuilder
- Groovy Templates

PortalURL

`PortalURL` plays a similar role at the portal level. Its main role is to abstract the creation of an URL for a resource managed by the portal.

```
public abstract class PortalURL<R, U extends PortalURL<U>>
{
   ...
}
```

The `PortalURL` declaration may seem a bit strange at first sight with two generic types: `U` and `R`.

- The `R` generic type represents the type of the resource managed by the portal.
- The `U` generic type is also described as *self bound generic type*. This design pattern allows a class to return subtypes of itself in the class declaring the generic type. Java Enums are based on this principle (`class Enum<E extends Enum<E>>`).
  A portal URL has various methods but certainly the most important method is the `toString()` method that generates an URL targeting to the resource. The remaining methods are `getter` and `setter` used to mutate the URL configuration, those options will affect the URL representation when it is generated.

- `resource`: The mandatory resource associated with the URL.
- `locale`: The optional locale used in the URL allowing the creation of bookmarkable URL containing a language.
- `confirm`: The optional confirmation message displayed by the portal in the context of the portal UI.
- `ajax`: The ajax option allowing an ajax invocation of the URL.
  Obtaining a PortalURL

`PortalURL` objects are obtained from `RequestContext` instance, such as `PortalRequestContext`, or `PortletRequestContext`. Usually, those are obtained thanks to the `getCurrentInstance` method of the `RequestContext` class:

```
RequestContext ctx = RequestContext.getCurrentInstance();
```

`PortalURL` is created via the `createURL` method that takes an input as a resource type. The resource type is usually a constant and type-safe object that allows retrieving the `PortalURL` subclasses:

```
RequestContext ctx = RequestContext.getCurrentInstance();
PortalURL<R, U> url = ctx.createURL(type);
```

In reality, you will use a concrete type constant and have instead more concrete code like:

```
RequestContext ctx = RequestContext.getCurrentInstance();
NodeURL url = ctx.createURL(NodeURL.TYPE);
```

The `NodeURL.TYPE` is actually declared as `new ResourceType<NavigationResource, NodeURL>()` that can be described as a `type-literal` object emulated by a Java anonymous inner class. Such literal was introduced by Neil Gafter as Super Type Token and popularized by Google Guice as Type Literal. It is an interesting way to create a literal representing a kind of Java type.

NodeURL

The `NodeURL` class is one of the subclass of `PortalURL` that is specialized in navigation node resources:

```
public class NodeURL extends PortalURL<NavigationResource, NodeURL>
{
   ...
}
```

The `NodeURL` class does not carry any generic types of its super class, which means that a `NodeURL` is type-safe and you do not have to worry about generic types.

Using a `NodeURL` is pretty straightforward:

```
NodeURL url = RequestContext.getCurrentInstance().createURL(NodeURL.TYPE);
url.setResource(new NavigationResource("portal", "classic, "home"));
String s = url.toString();
```

The `NodeURL` subclass contains the specialized `setter` methods to make its usage even easier:

```
UserNode node = ...;
NodeURL url = RequestContext.getCurrentInstance().createURL(NodeURL.TYPE);
url.setNode(node);
String s = url.toString();
```

ComponentURL

The `ComponentURL` subclass is another specialization of `PortalURL` that allows the creation of WebUI components URLs. `ComponentURL` is commonly used to trigger WebUI events from client side:

```
<% def componentURL = uicomponent.event(...); /*or uicomponent.url(...) */ %>
<a href=$componentURL>Click me</a>
```

Normally, you should not have to deal with it as the WebUI framework has already an abstraction for managing URL known as `URLBuilder`. The `URLBuilder` implementation delegates URL creation to `ComponentURL` objects.

Portlet URLs

Portlet URLs API implementation delegates to the portal `ComponentURL` (via the portlet container SPI). It is possible to control the language in the URL from a `PortletURL` object by setting the `gtn:lang` property:

- When the property value is set to a value returned by the `Locale#toString()` method for locale objects having a non null language value and a null variant value, the URL generated by the `PortletURL#toString()` method will contain the locale in the URL.
- When the property value is set to an empty string, the generated URL will not contain a language. If the incoming URL was carrying a language, this language will be erased.
- When the property value is not set, it will not affect the generated URL.

```
PortletURL url = resp.createRenderURL();
url.setProperty("gtn:lang", "fr");
writer.print("<a href='" + url + "'>French</a>");
```

Webui URLBuilder

This internal API used to create URL works as usual and delegates to the `PortletURL` API when the framework is executed in a portlet, and delegates to a `ComponentURL` API when the framework is executed in the portal context. The API has been modified to take in account the language in URL with two properties on the builder:

- locale: A locale for setting on the URL.
- removeLocale: A boolean for removing the locale present on the URL.

Groovy Templates

In a Groovy template, the mechanism to create an URL is the same as the way of APIs above, however a splash of integration has been done to make creation of NodeURL simpler. A closure is bound under the `nodeurl` name and is available for invocation anytime. It will simply create a NodeURL object and return it:

```
UserNode node = ...;
NodeURL url = nodeurl();
url.setNode(node);
String s = url.toString();
```

The `nodeurl` closure is bound to Groovy template in `WebuiBindingContext`.

```
// Closure nodeurl()
put("nodeurl", new Closure(this)
{
  @Override
  public Object call(Object[] args)
  {
    return context.createURL(NodeURL.TYPE);
  }
});
```

2.14.3 Changes and migration from GateIn 3.1.x

The navigation controller implies a migration of the client code that is coupled to several internal APIs of GateIn. The major impact is related to anything dealing with URL:

- Creation of an URL representing a resource managed by the portal: navigation node or UI component.
- Using HTTP request related information.

Migration of navigation node URL

Using free form node

The previous code for creating navigation node was:

```
String uri = Util.getPortalRequestContext().getPortalURI() + "home";
```

The new code will look like:

```
PortalURL nodeURL = nodeurl();
NavigationResource resource = new NavigationResource(SiteType.PORTAL, pcontext.getPortalOwner(),
"home");
String uri = nodeURL.setResource(resource).toString();
```

Using UserNode object

The previous code for creating navigation node was:

```
UserNode node = ...;
String uri = Util.getPortalRequestContext().getPortalURI() + node.getURI()";
```

The new code will look like:

```
UserNode node = ...;
PortalURL nodeURL = nodeurl();
String uri = nodeURL.setNode(node).toString();
```

Security changes

Security configuration needs to be changed to keep the flexibility added by the navigation controller. In particular, the authentication does not depend anymore on path specified in `web.xml` but relies on the security mandated by the underlying resource instead. Here are the noticeable changes for security:

- Authentication is now triggered on the "`/login`" URL when it does not have a username or a password specified. Therefore, the URL `/login?initialURI=/classic/home` is (more or less) equivalent to `/private/classic/home`.
- When a resource cannot be viewed due to security constraint.
    - If the user is not logged, the authentication will be triggered.
    - Otherwise, a special page (the usual one) will be displayed.

Default handler

Redirection to the default portal used to be done by the `index.jsp` JSP page. This is not the case anymore, the `index.jsp` file has been removed and the welcome file in `web.xml` was removed too. Instead a specific handler in the routing table has been configured, the sole role of this handler is to redirect the request to the default portal when no other request has been matched previously:

```
<controller>
  ...
  <route path="/">
    <route-param qname="gtn:handler">
      <value>default</value>
    </route-param>
  </route>
</controller>
```

Legacy handler

Legacy URLs, such as `/public/...` and `/private/...` are now emulated to determine the best resource with the same resolution algorithm, but instead of displaying the page, it will make an HTTP 302 redirection to the correct URL. This handler is present in the controller configuration. There is a noticeable difference between two routes.

- The public redirection attempts to find a node with the legacy resolution algorithm without authentication, which means that secured nodes will not be resolved and the redirection of a secured node will likely redirect to another page. For instance, resolving the URL `/public/classic/administration/registry` path will likely resolve to another node if the user is not authenticated and is not in the platform administrator group.
- The private redirection first performs an authentication before doing the redirection. In that case, the `/private/classic/administration/registry` path will be redirected to the `/portal/groups/:platform:administrators/administration/registry` page if the user has the sufficient security rights.

Static resource handler

The "/" mapping for the "default" servlet is now replaced by mapping for the `org.exoplatform.portal.application.PortalController` servlet. It means that you need a handler (`org.exoplatform.portal.application.StaticResourceRequestHandler`) to serve static resources like image, CSS or JavaScript files in `portal.war`. And it should be configured, and extended easily thanks to the `controller.xml` file. This file can be overridden and can be changed and reloaded at runtime (`WebAppController` is MBean with some operations, such as `reloadConfiguration()`).

Declare `StaticResourceHandler` in `controller.xml`.

```
<route path="/{gtn:path}">
  <route-param qname="gtn:handler">
    <value>staticResource</value>
  </route-param>
  <path-param encoding="preserve-path" qname="gtn:path">
    <pattern>.*\.(jpg|png|gif|ico|css)</pattern>
  </path-param>
</route>
```

And you do not need these kinds of the following mapping in `web.xml` in `portal.war` anymore.

```
<servlet-mapping>
  <servlet-name>default</servlet-name>
  <url-pattern>*.jpg</url-pattern>
</servlet-mapping>
...
```

portal.war's web.xml changes

Declare `DoLoginServlet`:

```
<servlet>
  <servlet-name>DoLoginServlet</servlet-name>
  <servlet-class>org.exoplatform.web.login.DoLoginServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>DoLoginServlet</servlet-name>
  <url-pattern>/dologin</url-pattern>
</servlet-mapping>
```

Delare *portal servlet* as the default servlet:

```
<servlet-mapping>
   <servlet-name>portal</servlet-name>
   <url-pattern>/</url-pattern>
</servlet-mapping>
```

Some mapping declarations for portal servlet are unused, so you should remove them: *_/private/* /public/* /admin/* /upload/* /download/*_*

Add some security constraints.

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>user authentication</web-resource-name>
      <url-pattern>/dologin</url-pattern>
      <url-pattern>/standalone/*</url-pattern>
      <url-pattern>/groups/*</url-pattern>
      <url-pattern>/users/*</url-pattern>
...
  </web-resource-collection>
</security-constraint>
```

You can remove the `index.jsp` file, and its declaration in the `web.xml` file thanks to the default request handler:

```
<welcome-file-list>
  <welcome-file>/index.jsp</welcome-file>
</welcome-file-list>
```

Dashboard changes

There are several important changes you need to take into account:

- Dashboard are now bound to a single URL (/users/root by default) and dashboard pages are leaf of this path.
- Dashboard lifecycle can be decoupled (create or destroy) from the identity creation in a configurable manner in `UserPortalConfigService` and exposed in `configuration.properties` under `gatein.portal.idm.createuserportal` and `gatein.portal.idm.destroyuserportal`.
- By default, dashboard are not created when a user is registered.
- A dashboard is created when the user accesses his dashboard URL.

Remove unused files

- `portal-unavailable.jsp`: This file is presented before if a user goes to a non-available portal. But now the server sends a 404 status code instead.
- `portal-warning.jsp`: This file is not used in any places.

3 Portlet Development

3.1 Portlet Primer

3.1.1 168 and JSR-286 overview

The Java Community Process (*JCP*) uses Java Specification Requests (JSRs) to define proposed specifications and technologies designed for the Java platform.

The Portlet Specifications aim at defining portlets that can be used by any JSR-168 (Portlet 1.0) or JSR-286 (Portlet 2.0) portlet container.

Most Java EE (Enterprise Edition) portals include at least one compliant portlet container, and GateIn 3.2 is no exception. In fact, GateIn 3.2 includes a container that supports both versions.

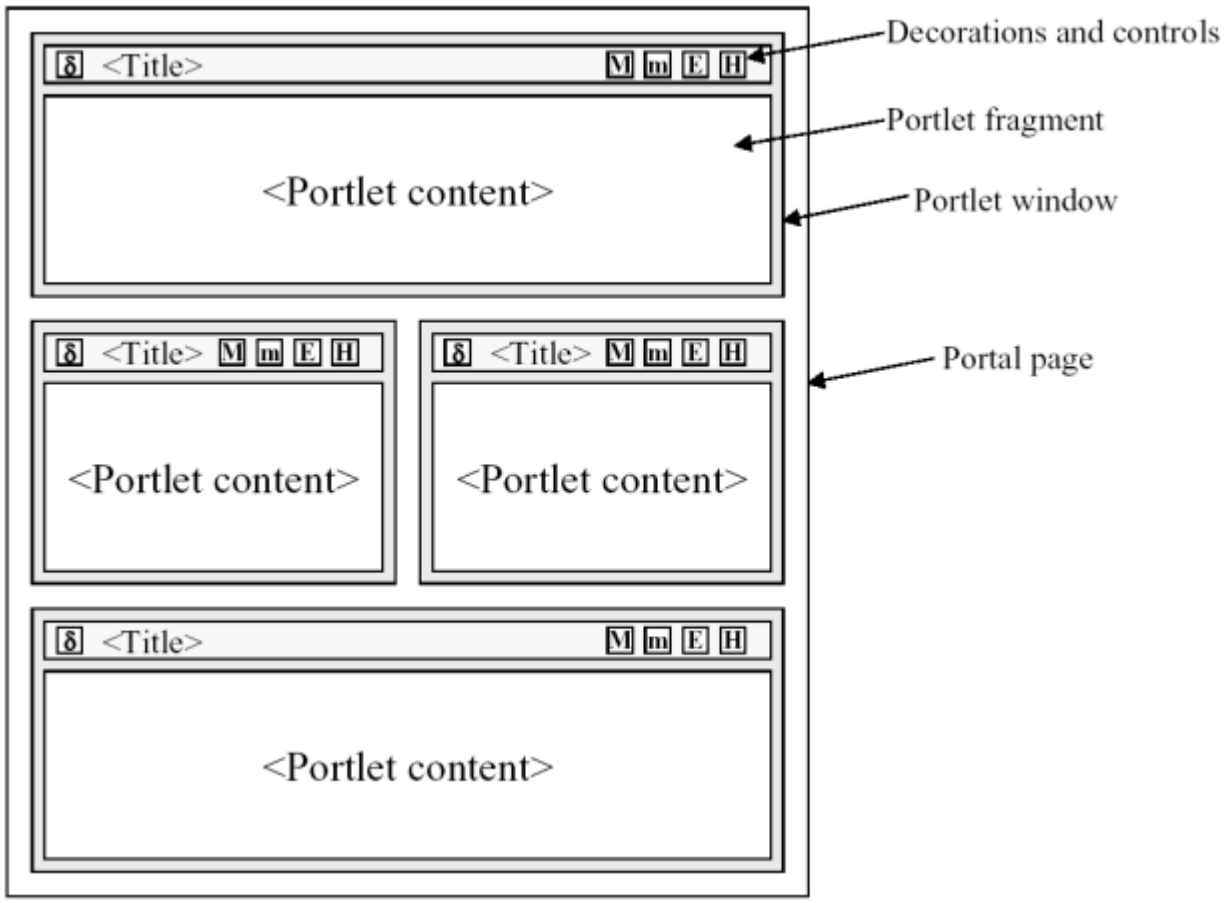
This section gives a brief overview of the Portlet Specifications, but portlet developers are strongly encouraged to read the JSR-286 Portlet Specification.

GateIn 3.2 is fully JSR-286 compliant. Any JSR-168 or JSR-286 portlet operates as it is mandated by the respective specifications inside the portal.

Portal Pages

A portal can be considered as a series of web pages with different *areas* within them. Those areas contain different *windows* and each *window* contains portlet:

The diagram below visually represents this nesting:



Rendering Modes

A portlet can have different view modes. Three modes are defined by the JSR-286 specification:

- *View*

  > *Generate the markup reflecting the current state of the portlet.*

- *Edit*

  > *Allow you to customize the behavior of the portlet.*

- *Help*

  > *Provide information to the user and how to use the portlet.*

Window States

Window states are indicators of how much space is consumed on any given page by a portlet. The three states defined by the JSR-168 specification are:

- *Normal*

  > *A portlet shares this page with other portlets.*

- *Minimized*

  > *A portlet may show very little information, or none at all.*

- *Maximized*

  > *A portlet may be the only portlet displayed on this page.*

3.1.2 Tutorials

The tutorials contained in this section are targeted towards portlet developers. It is also recommended that developers read and understand the JSR-286 Portlet Specification.

Maven

This example is using Maven to compile and build the web archive. Maven versions can be downloaded from maven.apache.org.

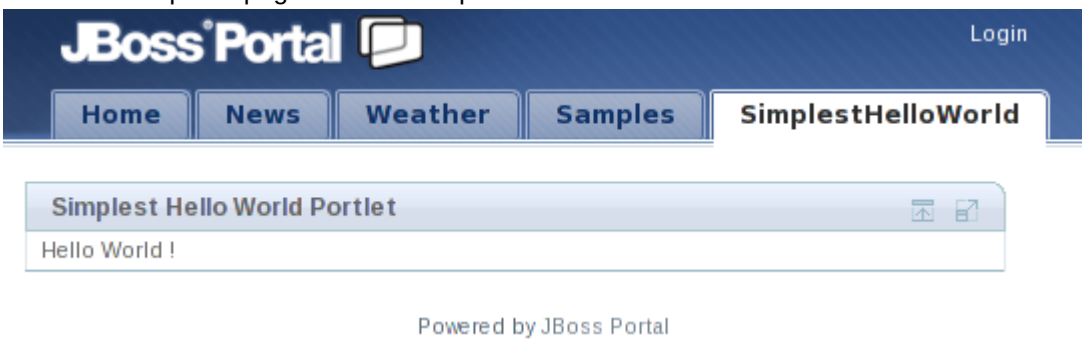
Deploying your first portlet

This section describes how to deploy a portlet in GateIn 3.2. A sample portlet called *SimplestHelloWorld* is located in the `examples` directory at the root of your GateIn 3.2 binary package. This sample is used in the following cases.

Compiling

To compile and package the application:

1. Navigate to the *SimplestHelloWorld* directory and execute: `mvn package`.
   If the compile is successfully packaged, the result will be available in: `SimplestHelloWorld/target/SimplestHelloWorld-0.0.1.war`.
2. Copy the package file into `JBOSS_HOME/server/default/deploy`.
3. Start the JBoss Application Server (if it is not already running).
4. Create a new portal page and add the portlet to it.



Package Structure

Like other Java EE applications, GateIn 3.2 portlets are packaged in the `.war` files. A typical portlet `.war` file can include servlets, resource bundles, images, HTML, JavaServer Pages (JSP), and other static or dynamic files.

The following is an example of the directory structure of the *SimplestHelloWorld* portlet:

```
|-- SimplestHelloWorld-0.0.1.war
|   `-- WEB-INF
|       |-- classes
|       |   `-- org
|       |       `-- gatein
|       |           `-- portal
|       |               `-- examples
|       |                   `-- portlets
|       |                       `-- SimplestHelloWorldPortlet.class
|       |-- portlet.xml
|       `-- web.xml
```

The compiled Java class implements `javax.portlet.Portlet` (through `javax.portlet.GenericPortlet`).

This is the mandatory descriptor files for portlets. It is used during deployment.

This is the mandatory descriptor for web applications.

Portlet Class

Below is the Java source:

- {{SimplestHelloWorldPortlet/src/main/java/org/gatein/portal/examples/portlets/SimplestHelloWorldPortl

```
package org.gatein.portal.examples.portlets;

import java.io.IOException;
import java.io.PrintWriter;

import javax.portlet.GenericPortlet;
import javax.portlet.RenderRequest;
import javax.portlet.RenderResponse;

public class SimplestHelloWorldPortlet extends GenericPortlet
{
   public void doView(RenderRequest request,
                      RenderResponse response) throws IOException
   {
      PrintWriter writer = response.getWriter();
      writer.write("Hello World !");
      writer.close();
   }
}
```

  All portlets must implement the `javax.portlet.Portlet` interface. The portlet API provides a convenient implementation of this interface.

The `javax.portlet.Portlet` interface uses the `javax.portlet.GenericPortlet` class which implements the `Portlet render` method to dispatch to abstract mode-specific methods. This makes it easier to support the standard portlet modes.

`Portlet render` also provides a default implementation for the `processAction`, `init` and `destroy` methods. It is recommended to extend `GenericPortlet` for most cases.

If only the `view` mode is required, only the `doView` method needs to be implemented. The `GenericPortletrender` implementation calls our implementation when the `view` mode is requested.

Use the `RenderResponse` to obtain a writer to be used to produce content.

Write the markup to display.

Close the writer.

> **Markup Fragments**
>
> Portlets are responsible for generating markup fragments, as they are included on a page and are surrounded by other portlets. This means that a portlet outputing HTML must not output any markup that cannot be found in a `<body>` element.

Application Descriptors

GateIn 3.2 requires certain descriptors to be included in a portlet WAR file. These descriptors are defined by the Jave EE (`web.xml`) and Portlet Specification (`portlet.xml`).

Below is an example of the `SimplestHelloWorldPortlet/WEB-INF/portlet.xml` file. This file must adhere to its definition in the JSR-286 Portlet Specification. More than one portlet application may be defined in this file:

```
<portlet-app xmlns="http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd
                                       http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd"
   version="2.0">
   <portlet>
      <portlet-name>SimplestHelloWorldPortlet</portlet-name>
      <portlet-class>
         org.gatein.portal.examples.portlets.SimplestHelloWorldPortlet
      </portlet-class>
      <supports>
        <mime-type>text/html</mime-type>
      </supports>
      <portlet-info>
          <title>Simplest Hello World Portlet</title>
      </portlet-info>
   </portlet>
</portlet-app>
```

Define the portlet name. It does not have to be the class name.

The Fully Qualified Name (`FQN`) of your portlet class must be declared here.

The `<supports>` element declares all of the markup types that a portlet supports in the `render` method. This is accomplished via the `<mime-type>` element, which is required for every portlet.

The declared MIME types must match the capability of the portlet. It allows administrators to pair which modes and window states are supported for each markup type.

This does not have to be declared as all portlets must support the `view` portlet mode.

Use the `<mime-type>` element to define which markup type the portlet supports. In the example above, this is `text/html`. This section tells the portal to only output HTML.

When rendered, the portlet's title is displayed as the header in the portlet window, unless it is overridden programmatically. In the example above, the title would be `Simplest Hello World Portlet`.

JavaServer Pages Portlet Example

This section discusses:

1. Add more features to the previous example.
2. Use a JSP page to render the markup.
3. Use the portlet tag library to generate links to the portlet in different ways.
4. Use the other standard portlet modes.

The example used in this section can be found in the `JSPHelloUser` directory.

1. **

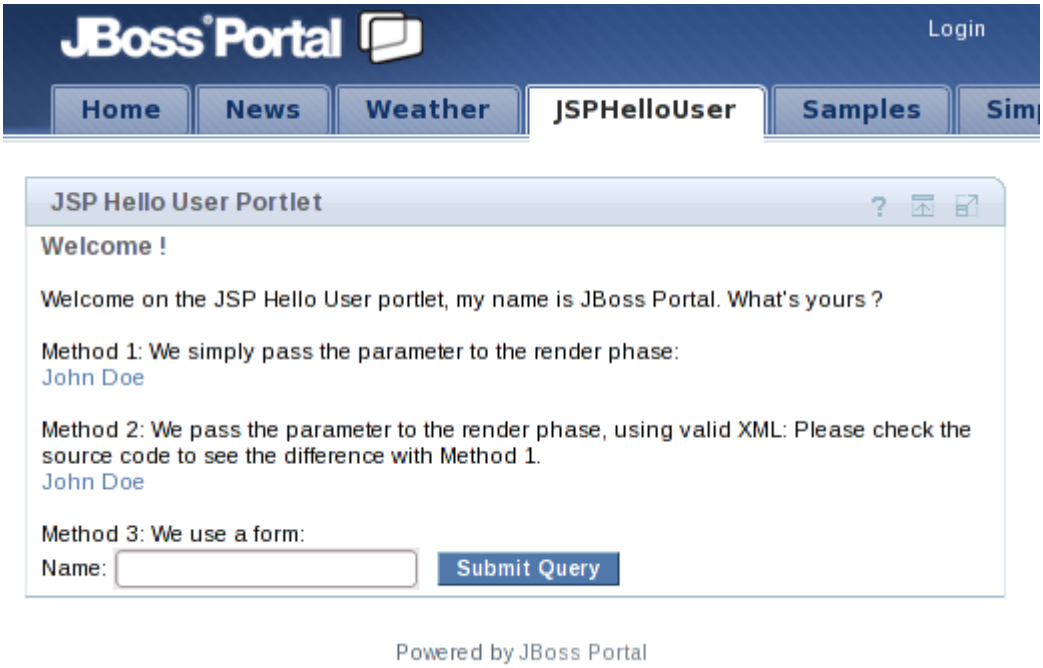
Execute `mvn package` in this directory.

2. **

Copy `JSPHelloUser/target/JSPHelloUser-0.0.1.war` to the `deploy` directory of JBoss Application Server.

3. **

Select the new `JSPHelloUser` tab in your portal.



The `EDIT` button only appears with logged-in users, which is not the case in the screenshot.

Package Structure

The package structure in this tutorial does not much differ from the previous example, with the exception of adding some JSP files detailed later.

The JSPHelloUser portlet contains the mandatory portlet application descriptors. The following is an example of the directory structure of the JSPHelloUser portlet:

```
JSPHelloUser-0.0.1.war
    |-- META-INF
    |   |-- MANIFEST.MF
    |-- WEB-INF
    |   |-- classes
    |   |   `-- org
    |   |       `-- gatein
    |   |           `-- portal
    |   |               `-- examples
    |   |                   `-- portlets
    |   |                       `-- JSPHelloUserPortlet.class
    |   |-- portlet.xml
    |   `-- web.xml
    `-- jsp
        |-- edit.jsp
        |-- hello.jsp
        |-- help.jsp
        `-- welcome.jsp
```

Portlet Class

The code below is from the Java source:

- `JSPHelloUser/src/main/java/org/gatein/portal/examples/portlets/JSPHelloUserI`

It is split in different pieces.

```
package org.gatein.portal.examples.portlets;

import java.io.IOException;

import javax.portlet.ActionRequest;
import javax.portlet.ActionResponse;
import javax.portlet.GenericPortlet;
import javax.portlet.PortletException;
import javax.portlet.PortletRequestDispatcher;
import javax.portlet.RenderRequest;
import javax.portlet.RenderResponse;
import javax.portlet.UnavailableException;

public class JSPHelloUserPortlet extends GenericPortlet
{

   public void doView(RenderRequest request, RenderResponse response)
       throws PortletException, IOException
   {
      String sYourName = (String) request.getParameter("yourname");
      if (sYourName != null)
      {
         request.setAttribute("yourname", sYourName);
         PortletRequestDispatcher prd =
            getPortletContext().getRequestDispatcher("/jsp/hello.jsp");
         prd.include(request, response);
      }
      else
      {
         PortletRequestDispatcher prd =
getPortletContext().getRequestDispatcher("/jsp/welcome.jsp");
         prd.include(request, response);
      }
   }
...
```

Override the `doView` method (as in the first tutorial).

This entry attempts to obtain the value of the render parameter named `yourname`. If defined, it should redirect to the `hello.jsp` JSP page or to the `welcome.jsp` JSP page.

Get a request dispatcher on a file located within the web archive.

Perform the inclusion of the markup obtained from the JSP.

Like the `VIEW` portlet mode, the specification defines two other modes: `EDIT` and `HELP`.

These modes need to be defined in the `portlet.xml` descriptor. This enables the corresponding buttons on the portlet's window.

The generic portlet that is inherited dispatches different views to the methods: `doView`, `doHelp` and `doEdit`.

```
...
   protected void doHelp(RenderRequest rRequest, RenderResponse rResponse) throws
PortletException, IOException,
         UnavailableException
   {
      rResponse.setContentType("text/html");
      PortletRequestDispatcher prd = getPortletContext().getRequestDispatcher("/jsp/help.jsp");
      prd.include(rRequest, rResponse);
   }

   protected void doEdit(RenderRequest rRequest, RenderResponse rResponse) throws
PortletException, IOException,
         UnavailableException
   {
      rResponse.setContentType("text/html");
      PortletRequestDispatcher prd = getPortletContext().getRequestDispatcher("/jsp/edit.jsp");
      prd.include(rRequest, rResponse);
   }
...
```

Portlet calls happen in one or two phases: when the portlet is rendered and when the portlet is actioned , *then* rendered.

An action phase is a phase which contains some state changes. The render phase will have access to render parameters that will be passed each time the portlet is refreshed (with the exception of caching capabilities).

The code to be executed during an action has to be implemented in the `processAction` method of the portlet.

```
...
         public void processAction(ActionRequest aRequest, ActionResponse aResponse) throws
PortletException, IOException,
         UnavailableException
   {
      String sYourname = (String) aRequest.getParameter("yourname");
      aResponse.setRenderParameter("yourname", sYourname);
   }
...
```

`processAction` is the method from `GernericPorlet` to override for the `action` phase.

Here the parameter is retrieved through an `action URL`.

The value of `yourname` is kept to make it available in the rendering phase. The previous line simply copies action parameters to a render parameter for this example.

JSP files and the Portlet Tag Library

The `help.jsp` and `edit.jsp` files are very simple. Note that CSS styles are used as defined in the portlet specification. This ensures that the portlet will render successfully within the theme and across portal vendors.

```
<div class="portlet-section-header">Help mode</div>
<div class="portlet-section-body">This is the help mode where you can find useful
information.</div>
```

```
<div class="portlet-section-header">Edit mode</div>
<div class="portlet-section-body">This is the edit mode where you can change your portlet
preferences.</div>
```

The landing page contains the links and form to call the portlet:

```
<%@ taglib uri="http://java.sun.com/portlet" prefix="portlet" %>

<div class="portlet-section-header">Welcome !</div>

<br/>

<div class="portlet-font">Welcome on the JSP Hello User portlet,
my name is GateIn Portal. What's yours ?</div>

<br/>

<div class="portlet-font">Method 1: We simply pass the parameter to the render phase:<br/>
<a href="<portlet:renderURL><portlet:param name="yourname" value="John Doe"/>
                </portlet:renderURL>">John Doe</a></div>

<br/>

<div class="portlet-font">Method 2: We pass the parameter to the render phase, using valid XML:
Please check the source code to see the difference with Method 1.
<portlet:renderURL var="myRenderURL">
    <portlet:param name="yourname" value='John Doe'/>
</portlet:renderURL>
<br/>
<a href="<%= myRenderURL %>">John Doe</a></div>

<br/>

<div class="portlet-font">Method 3: We use a form:<br/>

<portlet:actionURL var="myActionURL"/>
<form action="<%= myActionURL %>" method="POST">
         <span class="portlet-form-field-label">Name:</span>
         <input class="portlet-form-input-field" type="text" name="yourname"/>
         <input class="portlet-form-button" type="Submit"/>
</form>
</div>
```

The portlet taglib which needs to be declared.

The first method shown here is the simplest one. `portlet:renderURL` will create a URL that calls the render phase of the current portlet and append the result at the place of the markup (within a tag). A parameter is also added directly to the URL.

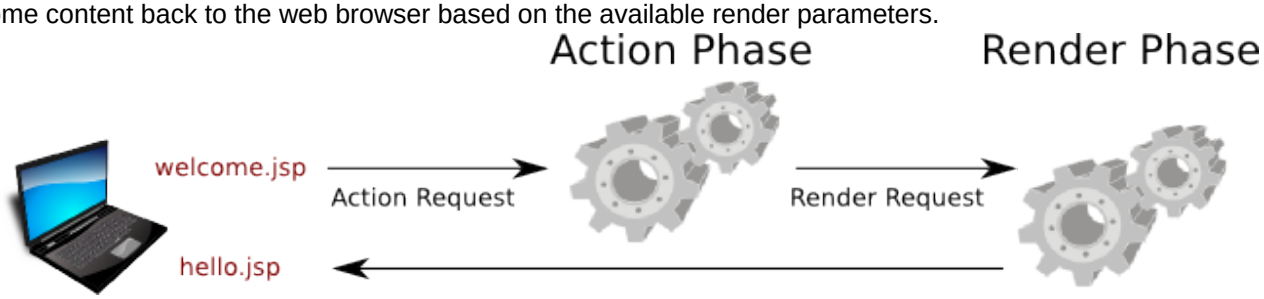
In this method, the `var` attribute is used. This avoids having one XML tag within another. Instead of printing the URL, the `portlet:renderURL` tag will store the result in the referenced variable (`myRenderURL`).

The variable `myRenderURL` is used like any other JSP variable.

The third method mixes the form submission and action request. Again, a temporary variable is used to put the created URL into.

The action URL is used in the HTML form.

In the third method, the action phase is triggered first, then the render phase is triggered, which outputs some content back to the web browser based on the available render parameters.



JSF example using the JBoss Portlet Bridge

To write a portlet using JSF, it is required to have a 'bridge'. This software allows developers to write a portlet application as if it was a JSF application. The bridge then negotiates the interactions between the two layers.

An example of the JBoss Portlet Bridge is available in `examples/JSFHelloUser`. The configuration is slightly different from a JSP application. This example can be used as a base to configure instead of creating a new application.

As in any JSF application, the `faces-config.xml` file is required. It must contain the following information:

```
<faces-config>
...
    <application>
      <view-handler>org.jboss.portletbridge.application.PortletViewHandler</view-handler>
      <state-manager>org.jboss.portletbridge.application.PortletStateManager</state-manager>
   </application>
...
</faces-config>
```

The portlet bridge libraries must be available and are usually bundled with the `WEB-INF/lib` directory of the web archive.

The other differences as compared to a regular portlet application can be found in the portlet descriptor. All details about it can be found in the JSR-301 specification that the JBoss Portlet Bridge implements.

```
<?xml version="1.0" encoding="UTF-8"?>
<portlet-app xmlns="http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd
                                          http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd"
   version="2.0">
   <portlet>
      <portlet-name>JSFHelloUserPortlet</portlet-name>
      <portlet-class>javax.portlet.faces.GenericFacesPortlet</portlet-class>
      <supports>
         <mime-type>text/html</mime-type>
         <portlet-mode>view</portlet-mode>
         <portlet-mode>edit</portlet-mode>
         <portlet-mode>help</portlet-mode>
      </supports>
      <portlet-info>
         <title>JSF Hello User Portlet</title>
      </portlet-info>

      <init-param>
         <name>javax.portlet.faces.defaultViewId.view</name>
         <value>/jsf/welcome.jsp</value>
      </init-param>

      <init-param>
         <name>javax.portlet.faces.defaultViewId.edit</name>
         <value>/jsf/edit.jsp</value>
      </init-param>

      <init-param>
         <name>javax.portlet.faces.defaultViewId.help</name>
         <value>/jsf/help.jsp</value>
      </init-param>

   </portlet>
</portlet-app>
```

All JSF portlets define `javax.portlet.faces.GenericFacesPortlet` as the portlet class. This class is part of the JBoss Portlet Bridge.

This is a mandatory parameter to define what is the default page to display.

This parameter defines which page to display on the 'edit' mode.

This parameter defines which page to display on the 'help' mode.

3.2 Global portlet.xml file

3.2.1 Global portlet.xml usecase

The Portlet Specification introduces PortletFilter as a standard approach to extend the behaviors of portlet objects. For example, a filter can transform the content of portlet requests and portlet responses. According to the Portlet Specification, normally there are three steps in setting up a portlet filter:

1. Implement a PortletFilter object.
2. Define the filter in portlet application deployment descriptor.
3. Define the filter mapping in portlet definitions.
   Two first steps are quite simple and easy to be done; however, at the step 3, developers/administrators need to replicate the filter mapping in many portlet definitions that makes work erroneous and tedious in several usecases. The global portlet feature is designed to compensate such limitation.

3.2.2 Global metadata

The Global metadata is declared in the `portlet.xml` file conforming with Portlet 2.0 's XSD.

```
<portlet-app version="1.0" xmlns="http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd
http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd">

</portlet-app>
```

Location

The path to the global `portlet.xml` is value of `gatein.portlet.config` in the `configuration.properties` file and varied by hosting application servers.

- For Tomcat: `$TOMCAT_HOME/gatein/conf/portlet.xml`
- For JBoss: `$JBOSS_HOME/standalone/configuration/gatein/portlet.xml`

Global metadata elements

The global `portlet.xml` file conforms to the schema of the portlet deployment descriptor defined in the Portlet Specification with some restrictions. In this file, the following elements are supported:

- Portlet Filter
- Portlet Mode and Window State

Portlet filter

Portlet filter mappings declared in the global `portlet.xml` file are applied across portlet applications. With the XML configuration below, the *ApplicationMonitoringFilter* filter involves in request handling on any deployed portlet.

```
<filter>
    <filter-name>org.exoplatform.portal.application.ApplicationMonitoringFilter</filter-name>
    <filter-class>org.exoplatform.portal.application.ApplicationMonitoringFilter</filter-class>
    <lifecycle>ACTION_PHASE</lifecycle>
    <lifecycle>RENDER_PHASE</lifecycle>
    <lifecycle>EVENT_PHASE</lifecycle>
    <lifecycle>RESOURCE_PHASE</lifecycle>
</filter>
```

Application Monitoring Filter supports four lifecycle phases as the order below: *ACTION_PHASE/ EVENT_PHASE/ RENDER_PHASE/ RESOURCE_PHASE* and records statistic information on deployed portlets. The filter alternates actual monitoring mechanism in WebUI Framework.

Portlet Mode and Window State

The global `portlet.xml` file is considered as an alternative place to declare custom Portlet Modes and Window States.

4 Gadget Development

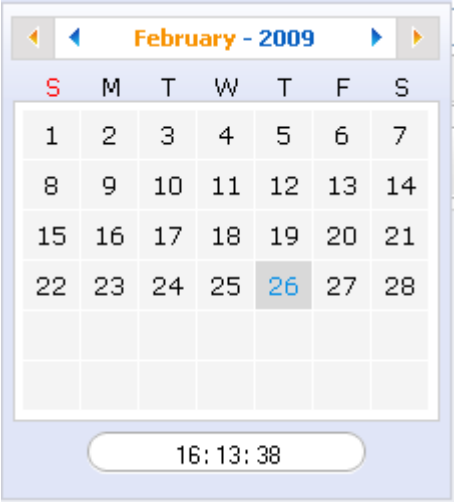
4.1 Gadgets

A gadget is a mini web application, embedded in a web page and running on an application server platform. These small applications help users perform various tasks.

GateIn 3.2 supports gadgets, such as *Todo*, *Calendar*, *Calculator*, *Weather Forecasts* and *RSS Reader*.

4.1.1 Default Gadgets:

- Calendar

> *The calendar gadget allows you to switch easily between daily, monthly and yearly views. Also, this gadget is customizable to match your portal's theme.*
>
> 
>

- ToDo

> *This application helps you organize your day and work group. It is designed to keep track of your tasks in a convenient and transparent way. Tasks can be highlighted with different colors.*
>
> 
>

- Calculator

> *This mini-application lets you perform the most basic arithmetic operations and can be themed to match the rest of your portal.*
>
> 
>

- RSS Reader

> *An RSS reader, or aggregator collects content from various, user-specified feed sources and displays them in one location. This content can include, but is not limited to, news headlines, blog posts or email. The RSS Reader gadget displays this content in a single window on your Portal page.*

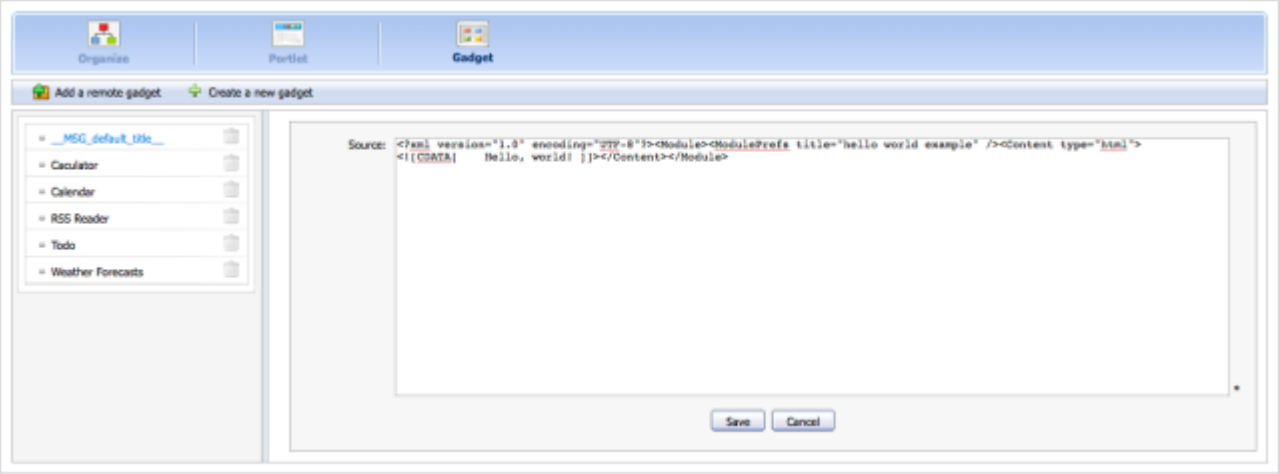
- More Gadgets

> *Further gadgets can be obtained from the Google Gadget site. GateIn 3.2 is compatible with most of the gadgets available here.*

> The following sections require more textual information.

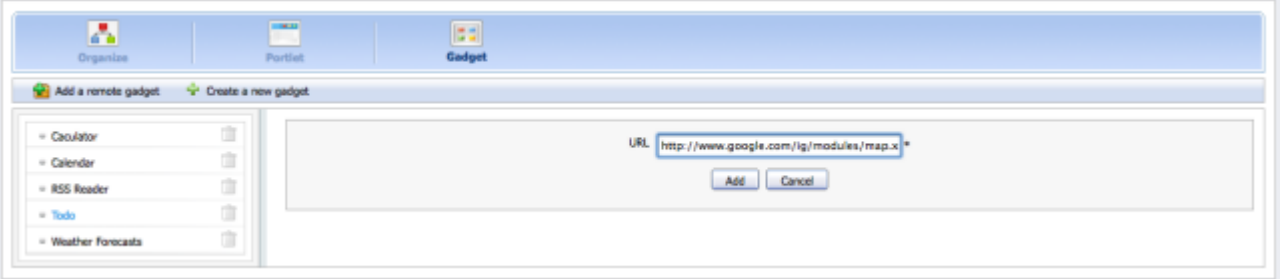
4.1.2 Existing Gadgets

4.1.3 Create a new Gadget



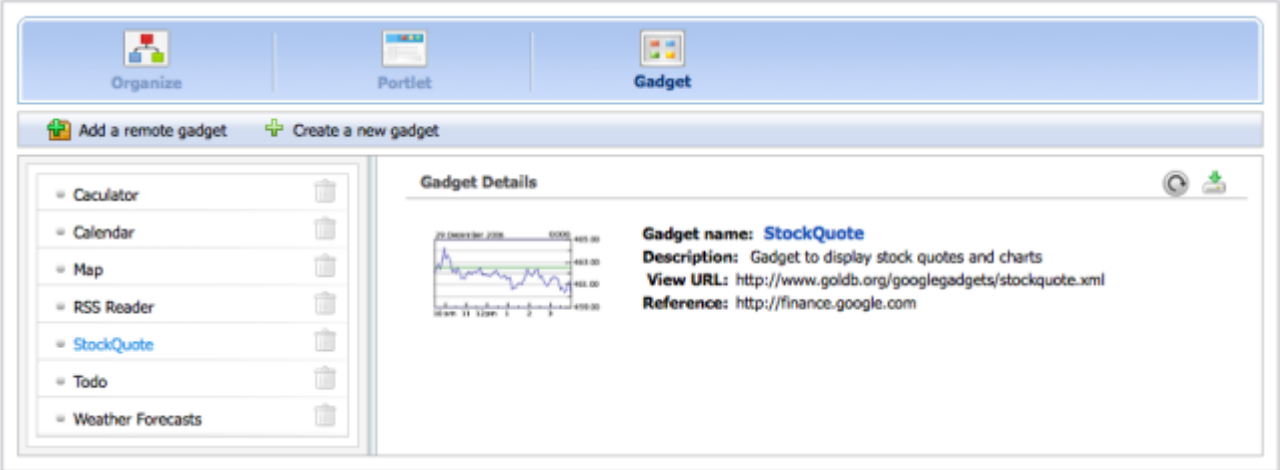
4.1.4 Remote Gadget

This is the reference to a remote gadget (stock one).



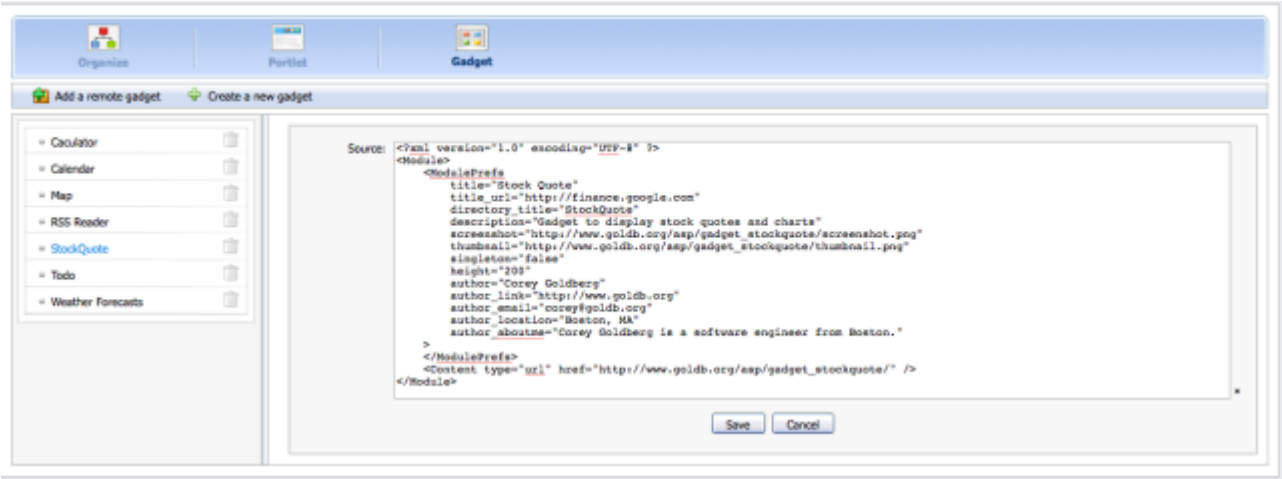
4.1.5 Gadget Importing

After referencing the gadget successfully, you can import it into the local repository.



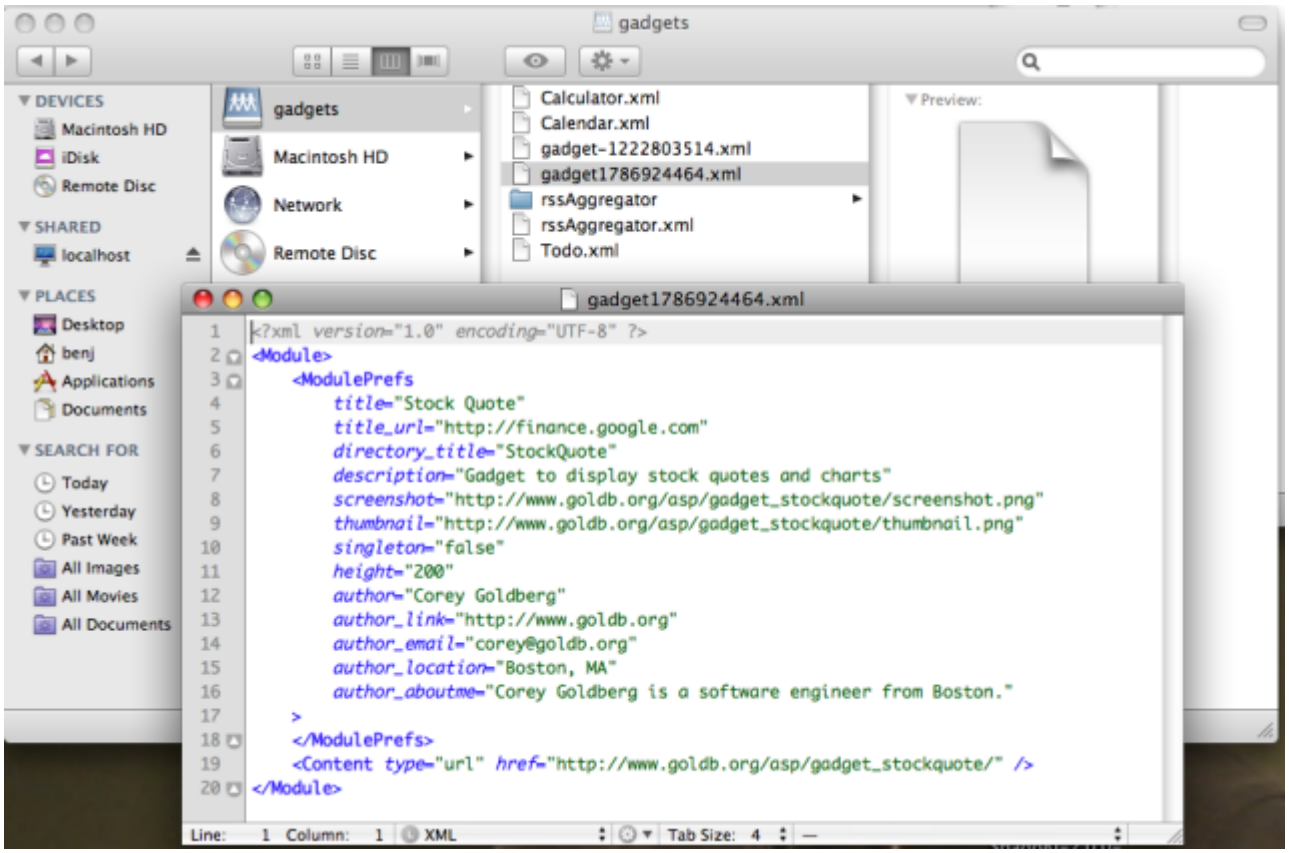
4.1.6 Gadget Web Editing

Modify it from the Web where the Gadget was imported:



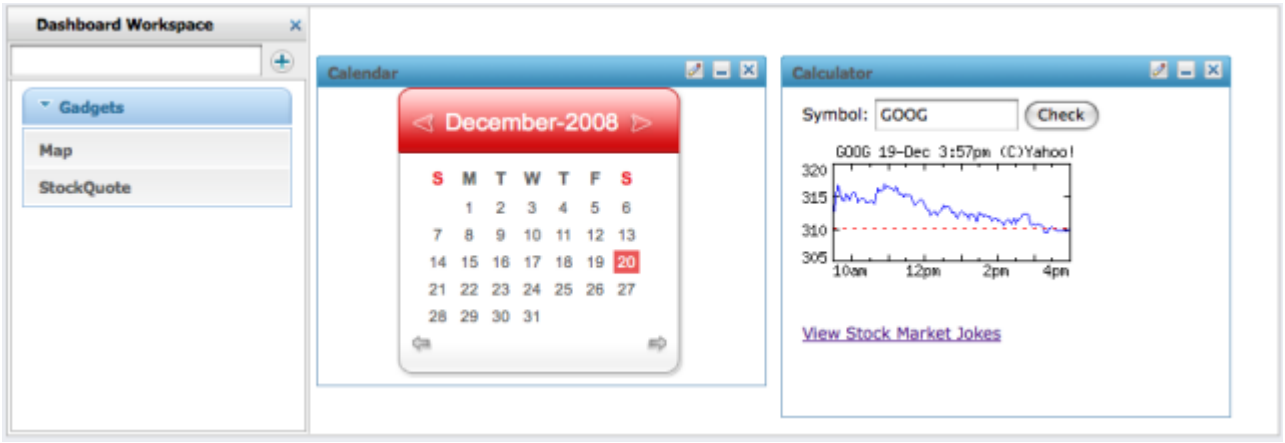
4.1.7 Gadget IDE Editing

Edit it from your IDE thanks to the WebDAV protocol:



4.1.8 Dashboard Viewing

View it from the Dashboard when you drag and drop the Gadget from listing to the dashboard.



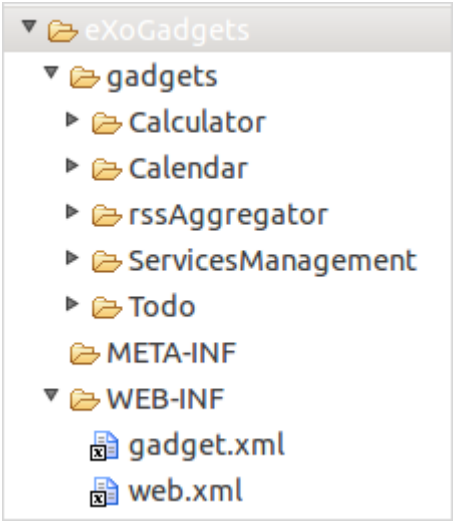
4.1.9 Standard WebApp for Gadget importer

In GateIn 3.2, you can create a WebApp folder to contain the configuration of the list of gadgets which are automatically added to the Application Registry.

This section shows you how to create a WebApp which has a standard structure processed by the Gadget importer.

Structure of WebApp

A WebApp should be organized as the following structure:



Necessary files

The WebApp structure consists of 2 necessary files: `web.xml` and `gadget.xml`.

In the `web.xml` file, there are 2 requirements:

- `org.exoplatform.portal.application.ResourceRequestFilter` filter: Handle requests to map into GateIn.
- `org.gatein.wci.api.GateInServlet` servlet: Help to register this WebApp to GateIn.

```
<web-app>
  <display-name>eXoGadgets</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>/*</url-pattern>
  </filter-mapping>

  <servlet>
    <servlet-name>GateInServlet</servlet-name>
    <servlet-class>org.gatein.wci.api.GateInServlet</servlet-class>
    <load-on-startup>0</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>GateInServlet</servlet-name>
    <url-pattern>/gateinservlet</url-pattern>
  </servlet-mapping>

</web-app>
```

The `gadget.xml` file is used to locate the configuration of gadgets that you want to add to the Application Registry.

```
<gadgets
    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/gadgets_1_0"
    xmlns="http://www.gatein.org/xml/ns/gadgets_1_0">

 <gadget name="To-do">
  <path>/gadgets/Todo/Todo.xml</path>
 </gadget>

 <gadget name="Calendar">
  <path>/gadgets/Calendar/Calendar.xml</path>
 </gadget>

 <gadget name="Calculator">
  <path>/gadgets/Calculator/Calculator.xml</path>
 </gadget>

 <gadget name="rssAggregator">
  <path>/gadgets/rssAggregator/rssAggregator.xml</path>
 </gadget>

  <gadget name="Currency">
    <url>http://www.donalobrien.net/apps/google/currency.xml</url>
  </gadget>

  <gadget name="ServicesManagement">
    <path>/gadgets/ServicesManagement/ServicesManagement.xml</path>
  </gadget>
</gadgets>
```

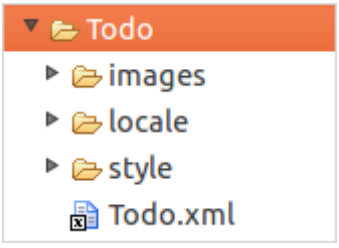
This file has 3 important tags:

- `<gadget>`: Every gadget will be configured in the `<gadget>` tag which has a "`name`" attribute to indicate the gadget name that Application Registry manages. The `<gadget>` tag can contain a child tag, either `<path>`, or `<url>`.
- `<path>`: This tag is the child of the `<gadget>` tag. It indicates the path to the gadget and helps the Gadget service find exactly where the gadget source and resources are located. It is known as the local path of the server.
- `<url>`: if the gadget source is located from the external server, the `<url>` tag should be configured to indicate the URL of the gadget.

Local gadget resources

For local gadgets, their source and resources should be located in the same folder.

See the example about the *To-do* gadget:



The Gadget importer service will find a folder that contains the gadget source, then find all resources in the folder and store them into JCR as the resources of the gadget. Therefore, putting the gadget resources in the folder different from the gadget source can cause many unnecessary files to be stored as the resource files of the gadget.

4.2 Set up a Gadget Server

4.2.1 Virtual servers for gadget rendering

GateIn 3.2 recommends using two virtual hosts for security. If the gadget is running on a different domain than the container (the website that 'contains' the app), it is unable to interfere with the portal by modifying code or cookies.

An example would hosting the portal from `http://www.sample.com` and the gadgets from `http://www.samplemodules.com`.

To do this, configure a parameter called `gadgets.hostName`. The value is the `path/to/gadgetServer` in `GadgetRegisteryService`:

```
<component>
  <key>org.exoplatform.application.gadget.GadgetRegistryService</key>
  <type>org.exoplatform.application.gadget.jcr.GadgetRegistryServiceImpl</type>
  <init-params>
    <value-param>
      <name>gadgets.hostName</name>
      <description>Gadget server url</description>
      <value>http://localhost:8080/GateInGadgetServer/gadgets/</value>
    </value-param>
  </init-params>
</component>
```

It is also possible to have multiple rendering servers. This helps to balance the rendering load across multiple servers.

When deploying on the same server, ensure the gadget initiates before anything that calls it (for example, the webapp `GateInGadgets` which uses `org.exoplatform.application.gadget.GadgetRegister`).

4.2.2 Configuration

Security key

In GateIn, the gadget container is using three security files for authentication and authorization gadgets:

- `key.txt`
- `oauthkey.pem`
- `oauthkey_pub.pem`
  By default, they are located in:

- For JBoss: `$JBOSS_HOME/standalone/configuration/gatein/gadgets`. This folder is configured by system variables in `$JBOSS_HOME/standalone/configuration/gatein/configuration.properties`.
- For Tomcat: `$TOMCAT_HOME/gatein/conf/gadgets`. This folder is configured by system variables in `$TOMCAT_HOME/gatein/conf/configuration.properties`.

```
gatein.gadgets.securitytokenkeyfile=$\{gatein.conf.dir\}/gadgets/key.txt
gatein.gadgets.signingkeyfile=$\{gatein.conf.dir\}/gadgets/oauthkey.pem
```

In case you have other files, you can change these variables to point to them.

The `key.txt` file contains a secret key used to encrypt the security token used for the user authentication. When starting GateIn, this file is read via the `gatein.gadgets.securitytokenkeyfile` path. In case the `key.txt` file is not found, GateIn automatically generates a new `key.txt` one and save it to the `gatein.gadgets.securitytokenkeyfile` path.

`oauthkey.pem` and `oauthkey_pub.pem` are a key pair of RSA cryptography standard. `oauthkey.pem` is known as a private key and `oauthkey_pub.pem` is a public key. They are the default keys of the gadget container which OAuth gadgets will use to authorize with external service providers.

Gadget proxy and concat configuration

These servers have to be on the same domain as the gadget server. You can configure the container in `eXoGadgetServer:/WEB-INF/classes/containers/default/container.js`.

```
"gadgets.content-rewrite" : {
  "include-urls": ".*",
  "exclude-urls": "",
  "include-tags": ["link", "script", "embed", "img", "style"],
  "expires": "86400",
  "proxy-url": "http://localhost:8080/eXoGadgetServer/gadgets/proxy?url=",
  "concat-url": "http://localhost:8080/eXoGadgetServer/gadgets/concat?"
},
```

Proxy

To allow external gadgets when the server is behind a proxy, add the following code to the beginning of the JVM:

```
-Dhttp.proxyHost=proxyhostURL -Dhttp.proxyPort=proxyPortNumber -Dhttp.proxyUser=someUserName
-Dhttp.proxyPassword=somePassword
```

5 JavaScript Development

5.1 JavaScript Modularity

5.1.1 The Rise of JavaScript

JavaScript has become more and more popular over the last few years, without any doubts its usage will continue to increase. GateIn is an aggregation platform focusing on aggregating markup in the browser with portlets, but also on integrating JavaScript in the same browser: each application comes with markup and JavaScript, more and more JavaScript. GateIn JavaScript integration until version 3.2 was designed a long time ago and was not able to answer the most recent needs required by modern web applications.

Since version 3.3, we started an effort to bring GateIn to support the most recent use cases of JavaScript integration. GateIn 3.3 and 3.4 are transitional versions that improved JavaScript support but were not able to answer all the use case we had in mind. GateIn 3.5 finally met the goals and reached maturity. Performance is the most important concern, GateIn provides now on demand, flexible and parallel loading of JavaScript resources. Support for modularity is fully addressed, it allows to integrate better JavaScript by providing a natural isolation mechanism between applications. Last but not least, GateIn internal JavaScript was rewritten on top of the jQuery library running in total isolation of the portlet runtimes, while it may seem an implementation detail, it deserves to be mentioned.

5.1.2 JavaScript modules

GateIn JavaScript improvements is built on top of the notion of JavaScript module. JavaScript does not provide a natural way for namespacing, the notion of module was designed to solve this problem. This natural lack of namespacing can be perceived as a lack, instead it should be seen as an advantage as modules provide namespacing and more: indeed the module pattern allows to create dependencies between modules and resolve them at runtime enabling on demand and parallel loading of JavaScript resources.

This guide will not explain modules because we haven't designed a module system for GateIn. Instead GateIn uses the RequireJS library and integrates. Therefore the best documentation you can read about modules is the RequireJS documentation you can also read the excellent article written by Ben Cherry about modules in depth.

Module in Ecmascript 6

Ecmascript 6 will provide native modules, you can read more about in this article

5.1.3 Introduction to modules

In the essence the notion of module can be viewed as:

- An identifier or name
- A list of dependencies on the modules required by the module to work properly
- The code packaged usually expressed as a self-executing function
- The product which is an object produced by the module that is usually consumed by other modules

At runtime the dependency system defines a graph of function to execute, the product of each module being injected in the other modules. It can be seen as a simple dependency injection system able to load modules in an asynchronous and parallel fashion providing parallel loading, namespacing and dependency management.

GateIn 3.5 provides support for different module format:

- GateIn Module Definition (GMD): the most appropriate way to define a JavaScript module in GateIn. The module is expressed as a simple function, the list of dependencies being expressed in the `gatein-resources.xml` file. This format clearly decouples the module code in a JavaScript file from the module dependencies expressed in a single XML file and is the best trade off for integrating JavaScript in GateIn.
- Asynchronous Module Definition (AMD): the native module format supported by RequireJS.
- Adapted Module: this format is the most flexible one as it allows to adapt to other module format such as CommonJS modules.

Previously we said that modules provide more than just namespacing, this section explains the other benefits of modules. Let's write a module consuming the jQuery library:

```
(function($) {
  $("body").on("click", ".mybutton", function(event) {
    alert("Button clicked");
  };
})($)
```

The JavaScript side of our module is a mere function that takes as argument the *jQuery* $ object:

- The jQuery object is provided as a module dependency expressed as a function argument $. The module code can use the name it likes for jQuery, in our module we use the $ name
  $ jQuery can be used without needing to tuse the `$.ready` function because jQuery will be already loaded and initialized when the function is executed
- The module is scoped to the entire page and this code will react to any click on the .mybutton selector.

Let's now integrate this module in GateIn, for this matter we declare it in the `gatein-resources.xml` file of the war file:

```
<gatein-resources
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.gatein.org/xml/ns/gatein_resources_1_3
http://www.gatein.org/xml/ns/gatein_resources_1_3"
  xmlns="http://www.gatein.org/xml/ns/gatein_resources_1_3">
  <portlet>
    <name>MyPortlet</name>
    <module>
      <script>
        <path>/mymodule.js</path>
      </script>
      <depends>
        <module>jquery</module>
        <as>$</as>
      </depends>
    </module>
  </portlet>
</<gatein-resources>
```

The module tag declares our module and its dependencies, in this case we declare that jQuery is the only dependency and it should be named $ when our module is executed. We have created a dependency relationship between our module and the jQuery module:

- Our module will be loaded only when the portlet is displayed
- GateIn will first load the jQuery module before loading our module
- Several different versions of jQuery can coexist in the browser without conflicting

This introductory example is simple but outlines the important features provided by modules: performance, isolation, dependency management and ease of use.

5.1.4 Script support

Modules should be used when possible but it is not always possible, for example when an existing portlet uses an inline script it will access the global namespace to find functions or objects. Although it is advocated to use modules, GateIn 3.5 provides supports this traditional way of using JavaScript with a simple script declaration in the head section of a web page. Dependencies between such script can be created to order the scripts in the head section and resolve dependency problems as much as possible.

5.2 JavaScript in GateIn

5.2.1 Modules in GateIn

GateIn Module Definition (GMD)

Declaring a module

We will first explain how a module can be integrated and consume dependencies, i.e other modules. A GateIn module consist in the declaration of a JavaScript self-executing function and its declaration in the `gatein-resources.xml` descriptor of the web application, the descriptor defines the module and its dependencies. At runtime GateIn will build a graph of resouces and their dependencies, when a client needs a particular module it will invoke a JS function that call *RequireJS* to resolve dependencies.

For instance if we have a *foo* module that uses the jQuery dependency:

foo.js

```
(function ($) {
  // Do something with jQuery
})(jQuery)
```

Our module is declared as a *self-executing function* in the `foo.js` file. This file is then declared in the `gatein-resources.xml` file:

gatein-resources.xml

```
...
<module>
  <name>foo</name>
  <script>
    <path>/foo.js</path>
  </script>
  <depends>
    <module>jquery</module>
    <as>jQuery</as>
  </depends>
</module>
...
```

The self-executing function declares

- parameter for the function: `$`
- arguments of the function invocation: `jQuery`

The self-executing function argument must match the dependencies

- Function do no need to match XML dependencies order
- Function can use parameter subset: an dependency can be declared in XML but not consumed as an argument
- Parameters must be aliased in dependencies with the <as> tag XML

The self executing function argument is a JavaScript construct. The purpose of this construct is to pass arguments to the function and let the function name the arguments like they want and override the current scope of execution. For example jQuery uses it:

```
(function(window, undefined) { .... })(window);
```

Resources are related by the concept of dependency relationship that specifies how scripts are related to each other and how the modular system should be honored. In our example, the foo module needs to use jQuery, we say that they are in relationship: the module foo depends on jQuery. When the module is loaded, the jQuery module must be available to the module.

Translation to the AMD system

GateIn will translate into an AMD module:

```
define("SHARED/foo", ["SHARED/jquery"], function(jQuery) {
  return (function($) {
    // Do something with jQuery
  })(jQuery);
});
```

- logical identifiers are translated to AMD identifiers
  - the *foo* module is translated to the `SHARED/foo` AMD module identifier, the *SHARED* prefix is added by GateIn for scoping the name, there are other existing scopes such as *PORTLET* and *PORTAL*
  - the dependency on the *jquery* module is translated to a `["SHARED/jquery"]` AMD module dependency
- the module function is wrapped two times
  - a first time by the GateIn module wrapper that delegates to the module function, the goal of this function is to provide a lazy evaluation of the module self-executing function.
  - a second time by the `define` AMD function that takes care of loading the module properly
- the self-executing function module arguments must match the module dependencies expressed in the XML declaration
  - the *jquery* dependency is aliased to the `jQuery` name thanks to the XML `as` tag, as a consequence the GateIn function wrapper parameter is named `jQuery`
  - the module self-executing function argument is named `jQuery` to match the declared alias
  - the module self-executing function parameter is named `$` and thus redefines the *jquery* depenendency to `$` locally

At runtime the following happens:

- the define function is invoked and declares the dependency
- when the dependency is resolved (i.e the jquery module is loaded and available)
  - the module wrapper is invoked with the `jQuery` argument containing the *jquery* dependency
  - the module wrapper evaluates the self-executing function that resolves the `jQuery` argument to the *jquery* dependency
  - the self-executing function is invoked and the *jquery* dependency is available under the name `$`

Producing dependencies

In our example we have seen that a module is able to consume dependencies as arguments, now we need to explain how a module can produce a dependency. When a module self-executing function is evaluated it can return an object, this precise object is what the module itself. Let's modify our previous example to return a value:

foo.js

```
(function ($) {
  // Do something with jQuery
  return {hello:"world"}
})($)
```

In this example we return a simple JavaScript object, this object will be stored by the dependency system of AMD and provided as arguments of modules that want to consume the *foo* module. Pretty simple isn't it ?

Module scopes

We have seen previously that the name of a logical GateIn module translates into an AMD name with a prefix among `SHARED`, `PORTLET` or `PORTAL`. Thus a module is fully identified by its logical name and its scope. Scopes ensure that a module (and therefore the underlying web resource) is loaded at the right time when it is effectively required.

Shared scope

The shared scope does not related to a specific GateIn entity, instead a shared module should be consumed by other modules, it is declared in `gatein-resources.xml` with the top level `module` tag:

```
<module>
  <name>core</name>
  <script>
    <path>/core.js</path>
  </script>
  <depends>
    <module>base</module>
  </depends>
</module>
```

Portal scope

The module is related to a GateIn portal and should be loaded when the related portal is accessed:

```
<portal>
  <name>classic</name>
  <module>
    <script>
      <path>/classic.js</path>
    </script>
    <depends>
      <module>core</module>
    </depends>
  </module>
</portal>
```

Portlet scope

The module will be loaded when the corresponding portlet is accessed:

```
<portlet>
  <name>sitemap</name>
  <module>
    <script>
      <path>/sitemap.js</path>
    </script>
    <depends>
      <module>core</module>
    </depends>
  </module>
</portlet>
```

A module can only depend on a shared module, therefore any `depends` tag implicitly refers to a shared module.

Portlet dynamic module

As seen previous, portlet dependencies can be expressed in the `gatein-resources.xml` file, it forces to declare the portlet dependencies at packaging time when the corresponding war file is created.

The JSR286 specification provides a mechanism for modifying portal headers that can be used to load JavaScript files. Although this mechanism is portable, it has severe drawbacks:

- A script can be loaded multiple times, specially if two portlets in two different war files load the same scripts since the only way to identify a script is by its URL
- Scripts have to be loaded by the head section of the portal impacting front end performances

GateIn allows to provide the best of both worlds and create dynamic dependencies of a portlet at runtime, when the render phase of the portlet occurs:

```
public void render(RenderRequest req, RenderResponse resp) throws PortletException, IOException
{
  resp.setProperty("org.gatein.javascript.dependency", "base");
  resp.addProperty("org.gatein.javascript.dependency", "common");
}
```

This code is equivalent to the following declaration:

```
<portlet>
  <name>MyPortlet</name>
  <module>
    <depends>
      <module>base</module>
    </depends>
    <depends>
      <module>common</module>
    </depends>
  </module>
</portlet>
```

Aliases

When a module is defined, the module name will be used in a JavaScript self-executing function argument:

```
<module>
  <name>foo</name>
  ...
</module>
<module>
  <name>bar</name>
  ...
  <depends>
    <module>foo</module>
  </depends>
</module>
```

The corresponding module *foo* code is:

```
(function(foo) {
})(foo)
```

Aliases allow to change this name and provide a different name for arguments. Aliasing is done thanks to the `as` XML tag:

Module alias

Other modules that depends on this will refer to it by the module alias defined in gatein-resources.xml.

```
<module>
  <name>foo</name>
  <as>Foo</as>
  ...
</module>
<module>
  <name>bar</name>
  ...
  <depends>
    <module>foo</module>
  </depends>
</module>
```

The corresponding module *foo* code is:

```
(function(Foo) {
})(Foo)
```

Dependency alias

The same as tag can be used in the depends tags providing a local alias:

```
<module>
  <name>foo</name>
  ...
</module>
<module>
  <name>bar</name>
  ...
  <depends>
    <module>foo</module>
    <as>Foo</as>
  </depends>
</module>
```

Custom adapters

JavaScript does not have a standard module until Ecmascript 6, several proposal exist for defining modules all revolving around the same module pattern. GateIn decided to use the AMD format because it is really adapted to the web and its asynchronous nature.

As consequence sometimes you can have to deal with included script that do not match the self-executing function declaration format or requirejs format expected by GateIn and is RequireJS integration. Custom code is required for adapting the script to the expected format. To provide this bit of flexibility it is possible to declare an adapter that will wrap the adapted script.

The jQuery library is a good example of how a custom adapter is useful, thanks to the adapter we can reuse the jQuery without any change, easing the integration and maintenance of this library in GateIn. jQuery uses the following construct for defining itself:

```
(function(window, undefined) {
})(window);
```

The main issue with this construct is that it will bind jQuery to the window but most importantly it will not return any value as expected by the dependency system. Thanks to the custom adapter we can integrate it trivially:

```
<module>
 <name>jquery</name>
 <script>
   <adapter>
     (function() {
       <include>/jquery.js</include>
       return jQuery.noConflict(true);
     })();
   </adapter>
 </script>
</module>
```

The `adapter` tag can contains mixed content and the `include` tag will perform a mere inclusion (as C language includes) of the original jQuery script in the resulting module:

```
define("SHARED/jquery", [], function() {
  return (function() {
    (function(window, undefined) {
    })(window);
   return jQuery.noConflict(true);
  })();
});
```

Module resources

The dependency notion is quite trivial, until now we have covered dependencies between modules. However AMD allows to define dependencies onto a resource loaded by a module based on the loader plugin.

Loader plugin are interesting because they can load resources thanks to the AMD loading mechanism and benefit from the same performances and parallel loading.

There are some useful loader plugin:

- The text plugin for loading text resources such as a stylesheet or a template
- The i18n plugin for internationalized bundle support

Using configuration in GateIn for AMD loader plugin's target resource is straightforward. For example, we want to build some HTML by Javascript, the `text.js` AMD loader plugin can help with this issue. The plugin will load the template (plugin's resource) and pass it into your module definition callback as a parameter.

First, we need to declare *text* as a GateIn module, it was written as a native module that depend on a predefined module of RequireJS *module*. Thanks to the native support mechanism, this configuration works transparently without modifying the 3rd party library.

```
<module>
  <name>text</name>
  <script>
    <path>/requirejs/js/plugins/text.js</path>
  </script>
  <depends>
    <!-- here module means RequireJS itself -->
    <module>module</module>
  </depends>
</module>
```

Now let's define our *foo* module that need the *text* plugin to load the template via the `resource` XML tag:

```
<module>
  <name>foo</name>
  ...
  <depends>
    <module>text</module>
    <as>myTemplate</as>
    <resource>/path/to/template.tmpl</resource>
  </depends>
</module>
```

Finally, the *foo* module will have its template loaded by the *text* plugin:

```
(function(myTemplate) {
  //parse the 'myTemplate' then append to DOM
})(myTemplate);
```

Load groups

Until now we focused on the logical definition of modules, i.e as a self-executing function and an XML descriptor. When those modules loaded by the browser, GateIn serves them as web resource. By default a module will be served as a single resource, however this can be an issue in the production system and the load group feature allows to decouple the logical module and the JavaScript resource serving:

- The granularity of the module system should have a small impact on the performance, specially when there are many small modules
- The front end performance should have a small impact on how the JavaScript modularity

The `load-group` XML tag can be used to group modules together in the same web resources. When a module of a load-group is requested, the web resource containing the modules of the load group is loaded, for instance if we have the *foo* and *bar* modules grouped by *foobar* :

```
<module>
  <name>foo</name>
  <load-group>foobar</load-group>
  ...
</module>
<module>
  <name>bar</name>
  <load-group>foobar</load-group>
  ...
</module>
```

When the *foo* is loaded the AMD loader will load the *foobar* group instead of only *foo*, consequently the *bar* module is also available now. If there is any request for *bar* module, no more request is made as it is already loaded.

Note: Don't use group name that follow by dash and 2 character, for example: **forbar-as**. This may cause unexpected parsing group name on the server (on gatein 3.5.0.Final)

Localisation

GateIn can localise modules on the server: the JavaScript of each module is rewritten and replace keys by values borrowed from resource bundles. Each script will be available under a different URL for each localisation to provide maximum caching of the resource.

The localisation of a script is activated with the `resource-bundle` XML tag along with the `path` tag. Resources bundles are declared by the `supported-locale` tag and must be available in the web application classpath:

```
<module>
  <name>greetings</name>
  <supported-locale>de</supported-locale>
  <supported-locale>en</supported-locale>
  <supported-locale>fr</supported-locale>
  <supported-locale>vi</supported-locale>
  <supported-locale>ru</supported-locale>
  <script>
    <path>/path/to/greetings.js</path>
    <resource-bundle>greetings_bundle</resource-bundle>
  </script>
  ...
</module>
```

GateIn will escape any key of the form `${key}` and will look up the value in the resource bundles. For instance one can build easily a JSON object containing localised values:

greetings.js

```
(function() {
  return {
    "hello": "${hello}",
    "goodbye": "${goodbye}"
  };
})();
```

At runtime different versions of this script will be served by GateIn with the properly escaped values.

5.2.2 Scripts in GateIn

Whenever possible, JavaScript should be written as AMD module (remember that GateIn also support custom adapter mechanism to adapt traditional script into GateIn module). This is the best practice, it helps to manage dependencies, organize code, not pollute the global scope and do parallel loading.

However we do acknowledge that in some specific case, people may want to use the old way: load the scripts as non-AMD module, GateIn still support this, the good thing is that we can still manage those js dependencies:

```
<scripts>
  <name>foo</name>
  <script>
    <path>/path/to/foo.js</path>
  </script>
  <depends>
    <scripts>bar</scripts>
  <depends>
</scripts>
```

- Traditional script must be declared in `scripts` tag instead of `module`
- Dependencies can be declared using `depends` but it can only depends on other scripts, not modules. Tthis also true at module side, a module can only depends on other modules
- Javascript declared in `scripts` will be push on the html head, that means those scripts will be loaded and executed before the DOM is created and will likely have an impact on front end performances

We can also use a script provided on the internet by using the `url` tag instead of the `script` tag:

```
<scripts>
  <name>foo</name>
  <url>http://path.to/foo.js</url>
</scripts>
```

5.3 JavaScript Cookbook

We have seen in the previous chapter how to use GateIn module system and the script integration. The goal of this section is to provide a list of common use case and how they can be addressed by GateIn. The section is two parts: the module cookbook and the script cookbook.

5.3.1 Module Cookbook

Declare a GateIn module

Let's study how to declare a GateIn module, the integration of the Highlight.js library. This example can be found among the GateIn examples in the `amd-js.war`. The Highlight.js is actually a jQuery plugin, jQuery plugins are perfect examples as they are natural GateIn modules and follow the self-invoking pattern that consumes the *jquery* dependency as $. Here is an overview of the Highlight.js source code:

```
(function($) {
  ...
}(jQuery)
```

It is integrated using the XML declaration:

```
<module>
  <name>highlight</name>
  <script>
    <path>/highlight/highlight.js</path>
  </script>
  <depends>
    <module>jquery</module>
    <as>jQuery</as>
  </depends>
</module>
```

- The module is named `highlight` and uses the `/highlight/highligh.js` source code bundled in the war file
- The `depends` tag creates a dependency on the *jquery* module. The dependency is aliased as `jQuery` using the `as` tag to match the Highlight.js self-executing function argument $.

The *jquery* plugins comes out of the box with GateIn and one should not worry about declaring it. The *jquery* module is simply named `jquery` and provides the jQuery 1.7.1 .

Declare a AMD module

In the previous section we mentioned that GateIn can integrate native AMD modules since GateIn modules are translated to AMD modules. Asynchronous module declaration is well explained in the RequireJS documentation and you should read it if you want to be familiar with the format.

AMD modules follow the pattern:

```
define("module", ["dependency1",...,"dependencyN"], function(dep1,...,depN) {
});
```

GateIn can use such module out of the box, however some parts will be overridden by the XML declaration:

- The `"module"` name will be ignored and replaced by the declared module name
- The module dependencies from `"dependency1"` to `"dependencyN"` have to be declared with the same name in XML (we will see later that we can use the `as` tag to override the dependency name)

Assuming that the dependencies `dependency1` to `dependencyN` have been declared in XML, such module definition can be declared with the following XML:

```
<module>
  <name>MyModule</name>
  ...
  <depends>
    <module>dependency1</module>
  </depends>
  ...
  <depends>
    <module>dependencyN</module>
  </depends>
</module>
```

Use GateIn jQuery module

GateIn provides the jQuery library 1.7.1 as a *jquery* module, the configuration of this module can be found in the `eXoResources.war` file. To reuse this jQuery version one just has to declare a dependency over it:

```
<portlet>
  <name>RequireJSPortlet</name>
  <module>
    <depends>
      <module>jquery</module>
    </depends>
</portlet>
```

The default *jquery* module alias is $ so if you are using it, it should be named $ in the self-executing function:

```
(function($) {
  ...
})($);
```

If your library use a different name such as `jQuery` the XML `as` tag should be used:

```
<portlet>
  <name>RequireJSPortlet</name>
  <module>
    <depends>
      <module>jquery</module>
      <as>jQuery</as>
    </depends>
</portlet>
```

With the following self-executing function:

```
(function($) {
  ...
}(jQuery);
```

Use a custom jQuery version

If you are not satisfied by the jQuery version provided by GateIn you can integrate the version you like. It is not uncommon that products built over GateIn depends on third party JavaScript frameworks depending on other versions of jQuery libraries, so deploying other jQuery libraries is unavoidable at some point. Having multiple jQuery instances within a web page is prone to conflict over global variables. The beauty of the module system is that you can use such library with no hassles.

Consider an example with a *jQueryPortlet* using jQuery version 1.6.4, what we need to do is to configure the two modules properly:

```
<module>
  <name>jquery-1.6.4</name>
  <script>
    <adapter>
(function() {
  <include>/javascript/jquery-1.6.4.js</include>
  return jQuery.noConflict(true);
})();
    </adapter>
  </script>
</module>
<portlet>
  <name>jQueryPortlet</name>
  ...
  <depends>
    <module>jquery-1.6.4</module>
  </depends>
</portlet>
```

jQuery plugins

In this section, we are going to see how to configure a jQuery plugin and how to use it in jQueryPluginPortlet portlet. The plugin code is a minimal plugin:

```
(function($) {
 $.fn.doesPluginWork = function() {
    alert('YES, it works!');
 };
})(jQuery);
```

The plugin is then declared as a module:

```
<module>
  <name>jquery-plugin</name>
  <as>jqPlugin</as>
  <script>
    <path>/jqueryPlugin/jquery-plugin.js</path>
  </script>
  <depends>
    <module>jquery</module>
    <as>jQuery</as>
  </depends>
</module>
```

Finally we use this plugin in our portlet:

```
<portlet>
  <name>jQueryPluginPortlet</name>
  <module>
    <script>
      <path>/jqueryPlugin/jqueryPluginPortlet.js</path>
    </script>
    <depends>
      <module>jquery</module>
      <as>$</as>
    </depends>
    <depends>
      <module>jquery-plugin</module>
    </depends>
  </module>
</portlet>
```

One important point to have in mind is that our portlet module should depend on the *jquery* and the *jquery-plugin* modules even it will not use the plugin itself:

- declaring the dependency on *jquery* allows to use the jQuery object
- delcaring the dependency on *jquery-plugin* ensures that the plugin will be loaded in the *jquery* dependency before it is injected in the portlet module

Overriding the dependency of a native AMD module

We have seen previously how to declare a native AMD module and said that the module dependency names must match the AMD dependencies declared in the `define` function arguments. When there is a mismatch between a module declared in the native module and the module system of GateIn the `as` tag can be used for renaming the dependencies.

Let's say that there is a `foo.js` file defining an AMD module named `foo` with two dependencies `"dep1", "dep2"` as following :

```
define("foo", ["dep1", "dep2"], function(a1, a2) {
  // The module
});
```

Now let's suppose that the dependencies are declared as *module1* and *module2* in GateIn, as we can see the names don't match. To override this we can use the `as` tag to rename the dependencies:

```
<module>
  <name>foo</name>
  <script>
    <path>/path/to/foo.js</path>
  </script>
  <depends>
    <module>module1</module>
    <as>dep1</as>
  </depends>
 <depends>
   <module>module2</module>
   <as>dep2</as>
 </depends>
</module>
```

CommonJS modules

CommonJS defines its own module format, although it is not supported natively by GateIn, the `adapter` format can be used to adapt CommonJS modules to work nicely in GateIn.

Here are two simple CommonJS modules:

math.js

```
exports.add = function() {
  var sum = 0, i = 0, args = arguments, l = args.length;
  while (i < l) {
    sum += args[i++];
  }
  return sum;
};
```

increment.js

```
var add = require('math').add;
exports.inc = function(val) {
  return add(val, 1);
};
```

CommonJS modules use its `require` function which is conflict with RequireJS same function. So in order to make it works in AMD
enabled environment, these modules need to be wrapped and injected *predefined* modules: *require*, *exports* and *module* provided by Requirejs (details here). The good news is that developers don't need to do this themselves, GateIn will wrap the code for you based on configuration using the adapter format:

```
<module>
  <name>math</name>
  <script>
    <adapter>
      define(["require", "exports"], function(require, exports) {
      <include>/commonjs/math.js</include>
      });
    </adapter>
  </script>
  <depends>
    <module>require</module>
  </depends>
  <depends>
    <module>exports</module>
  </depends>
</module>
<module>
  <name>increment</name>
  <script>
    <adapter>
      define(["require", "exports", "math"], function(require, exports) {
      <include>/commonjs/increment.js</include>
      });
    </adapter>
  </script>
  <depends>
    <module>require</module>
  </depends>
  <depends>
    <module>exports</module>
  </depends>
  <depends>
    <module>math</module>
  </depends>
</module>
```

Mustache.js module

Mustache.js is a popular JavaScript template engine. Mustache is written to be executed in several kind of environment : as a global object, as a CommonJS module or as a native AMD module. If "module", "exports" dependencies are available Mustache will register it as a CommonJS module. It can be adapted to GateIn thanks to the adapter format:

```
<module>
  <name>mustache</name>
  <script>
    <adapter>
define(["require", "exports", "module"], function(require, exports, module) {
  <include>/requirejs/js/plugins/mustache.js</include>
});
    </adapter>
  </script>
  <depends>
    <module>require</module>
  </depends>
  <depends>
    <module>exports</module>
  </depends>
  <depends>
    <module>module</module>
  </depends>
</module>
```

We need to use `adapter` tag here and declare the *require*, *exports* and *module* dependencies of the CommonJS module. Now any module can have Mustache instance injected just by declaring it in its dependencies list:

```
<module>
  <name>foo</name>
  ...
  <depends>
    <module>mustache</module>
  </depends>
</module>
```

```
(function(mustache){
  //code that use Mustache
  mustache.render(template);
})(mustache);
```

Resource loading with Text.js

RequireJS provides support for loader plugin allowing a module to be a plugin and use the AMD system for loading web resources in an efficient manner.

Mustache.js is a javascript template engine and the engine need a template file to parse and generate html. When there are many templates or the template has a large size, embedding template in the page is not a good choice for front end performance reason. It would be better to use Text.js to load the separate template files and inject them as dependencies.

Text.js is a native AMD module, it also depends on the *module* dependency predefined dependency provided by the AMD loader. Thanks to the native AMD support of GateIn, it is straight foward to declare and use Text.js in GateIn:

```
<module>
  <name>text</name>
  <script>
    <path>/requirejs/js/plugins/text.js</path>
  </script>
  <depends>
    <module>module</module>
  </depends>
</module>
```

Now we can use the *mustache* and *text* modules to load templates and render them in our own module:

```
<portlet>
  <name>foo</name>
  <module>
    ...
    <depends>
      <module>mustache</module>
    </depends>
    <depends>
      <module>text</module>
      <as>tmpl</as>
      <resource>/path/to/template.html</resource>
    </depends>
  </module>
</portlet>
```

We have the *text* module in dependency list with a <resource> tag, Text.js will load that resource template and inject it with the name `tmpl`. Here is the javascript of the portlet:

```
function(mustache, tmpl) {
  var html = mustache.render(tmpl);
  //append rendered html to DOM
})(mustache, tmpl);
```

5.3.2 Script cookbook

Accessing a module from a script

Sometimes it is required to access a module from a script, RequireJS provides such capability by using the `require` function to execute a function in the managed context:

```
require([“SHARED/ModuleA”], function(a) {
  // Codes of interacting with module A
  a.doSomething();
});
```

In such situation we need to use the AMD module name of the module we need to depend on, `PORTLET/ModuleA` in this case. The prefix in upper case is the module scope among `SHARED`, `PORTLET` and `PORTAL`.

Expose GateIn version of jQuery globally

As explained previously, the built-in jQuery is currently declared as an AMD module. By default jQuery will not be available in the window object of the browser. This cookbook recipe shows how to make it available so one can write code like in a plain script.

The following script will make jQuery available by mounting the jQuery object in the window object:

```
require( [“SHARED/jquery”], function($) {
  // the ‘$’ in window.$ is alias, you can make the other for yourself.
  window.$ = $;
});
```

This script must be integrated as a shared script:

```
<scripts>
  <name>imediatejs</name>
  <script>
    <path>/myfolder/imediateJScript.js</path>
  </script>
</scripts>
```

A portlet can then provide its own script that depends on this script:

```
<portlet>
  <name>foo</name>
  <script>
    <name>portletjs</name>
    <path>/myfolder/portlet.js</path>
  </script>
  <depends>
    <scripts>imediatejs</scripts>
  </depends>
</scripts>
```

With the following JavaScript:

```
$("#foo").html("<h1>hello global jQuery</h1>");
```

Define a custom global jQuery

We can define a globally available custom jQuery quite easily. For this we reuse existing recipes seen previously and combine them:

- Use a custom jQuery version
- Expose GateIn version of jQuery globally

Use a global jQuery plugin

There are a few ways to implement this recipe. However, the most important point to keep in mind is to make sure that the global jQuery is available before the global jQuery plugin is loaded.

We have seen before how we can scope a module to a portlet, the module will be loaded when the portlet is on a page using the `PORTLET` scope. In this recipe we will change and use instead a `PORTAL` scope, the main difference is that the loading of our plugin will be triggered on a specific portal instead of a specific portlet.

First step we create our jQuery plugin as a script named `myPlugin.js` and we integrate our plugin:

```
require([“SHARED/jquery”], function($) {
  $.fn.myPluginFunction = function() {
    // Your work here;
  };
});
```

Second step we bind the script in our portal and also reuse the `immediatejs` script seen before:

```
<portal>
  <name>classic</name>
  <scripts>
    <script>
      <name>myPlugin</name>
      <path>/myfolder/myPlugin.js</path>
    </script>
    <script>
      <name>imediatejs</name>
      <path>/myfolder/imediateJScript.js</path>
    </script>
  </scripts>
</portal>
```

Now, our plugin is globally available and we can use it:

```
<script type=”text/javascript”>
$(‘#foo’).myPluginFunction();
</script>
```

6 Authentication and Identity

6.1 Authentication and Authorization intro

Authentication in GateIn Portal is based on JAAS and by default it is standard J2EE FORM based authentication. However, the authentication workflow is not so easy and straightforward, because GateIn Portal supports many different authentication usecases, so that you can leverage authentication process according to your needs.

In GateIn Portal the following types of authentication are supported:

- J2EE FORM based authentication.
- *RememberMe* authentication (user checks *Remember my login* checkbox in login form).
- SSO servers integration (CAS, JOSSO, OpenSSO) - more information in Single-Sign-On (SSO).
- SPNEGO authentication with Kerberos ticket - more information in SPNEGO.
- SAML2 based authentication - more information in SAML2
- Cluster authentication with loadbalancer or with JBoss SSO valve - more information in Clustered SSO setup
  Authentication workflow consists of more HTTP requests and redirects with couple of handshakes in it. Currently we support only Servlet 3.0 containers, so authentication is triggered programmatically from Servlet API .

First you can see in *JBOSS_HOME/gatein/gatein.ear/portal.war/WEB-INF/web.xml* that authentication can be triggered by accessing secured URL */dologin* :

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>user authentication</web-resource-name>
      <url-pattern>/dologin</url-pattern>
      <http-method>POST</http-method>
      <http-method>GET</http-method>
    </web-resource-collection>
    <auth-constraint>
      <role-name>users</role-name>
    </auth-constraint>
    <user-data-constraint>
      <transport-guarantee>NONE</transport-guarantee>
    </user-data-constraint>
  </web-resource-collection>
</security-constraint>
```

This means that access to URL like http://localhost:8080/portal/dologin will directly trigger the J2EE authentication in case the user is not logged. Access to this URL also means that user needs to be in JAAS group *users*, otherwise he can authenticate but he will have HTTP error like *403 Forbidden*.

In next part of the file, you can see that authentication is FORM based and it starts by redirection to */login* URL, which is actually mapped to `LoginServlet`.

```
<login-config>
  <auth-method>FORM</auth-method>
  <realm-name>gatein-domain</realm-name>
  <form-login-config>
    <form-login-page>/login</form-login-page>
    <form-error-page>/login</form-error-page>
  </form-login-config>
</login-config>
```

`LoginServlet` simply redirects user to login page placed in `gatein/gatein.ear/portal.war/login/jsp/login.jsp`.

So if you want to change somehow the look and feel of this login page, you can do it in this JSP file. Alternatively you can create extension and override this page via extension if you don't want to edit it directly. You can also change/override image or CSS placed in *gatein/gatein.ear/login/skin*.

After the user has submitted his login form, he will be redirected to login URL, which looks like http://localhost:8080/portal/login?username=root&password=gtn&initialURI=/portal/classic, which is again mapped to `LoginServlet`. Now LoginServlet will trigger WCI login, which delegates to Servlet API (method `HttpServletRequest.login(String username, String password)` available in Servlet 3.0) and additionaly it triggers WCI Authentication listeners. Login through Servlet API will delegate to JAAS.

6.1.1 Login modules

So from WCI servlet API login, you are redirected to the JAAS authentication. GateIn Portal is using its own security domain *gatein-domain* with a set of predefined login modules. Login module configuration for gatein-domain is in the `JBOSS_HOME/standalone/configuration/standalone.xml` in JBoss AS7 and in `TOMCAT_HOME/conf/jaas.conf` in Tomcat 7. By default you can see this login modules stack:

```
<security-domain name="gatein-domain" cache-type="default">
  <authentication>
    <login-module code="org.gatein.sso.integration.SSODelegateLoginModule" flag="required">
      <module-option name="enabled" value="${gatein.sso.login.module.enabled}" />
      <module-option name="delegateClassName" value="${gatein.sso.login.module.class}" />
      <module-option name="portalContainerName" value="portal" />
      <module-option name="realmName" value="gatein-domain" />
      <module-option name="password-stacking" value="useFirstPass" />
    </login-module>
    <login-module code="org.exoplatform.services.security.j2ee.JBossAS7LoginModule"
flag="required">
      <module-option name="portalContainerName" value="portal"/>
      <module-option name="realmName" value="gatein-domain"/>
    </login-module>
  </authentication>
</security-domain>
```

You are free to add some new login modules or completely replace existing login modules with some of your own.

Authentication starts with invoke of the `login` method on each login module. After all `login` methods are invoked, the authentication is continued by invoke of the `commit` method on each login module. Both `login` and `commit` methods can throw `LoginException`. If it happens, then the whole authentication ends unsuccessfully, which in next turn invokes the `abort` method on each login module. By returning "false" from method login, you can ensure that login module is ignored. This is not specific to GateIn Portal but it is generic to JAAS. See here for more information about login modules in general.

Existing login modules

Here is some brief description of existing login modules:

- `SSODelegateLoginModule` - It's useful only if SSO authentication is enabled (disabled by default. It can be enabled through properties in `configuration.properties` file and in this case it delegates the work to another real login module for SSO integration. If SSO is disabled, SSODelegateLoginModule is simply ignored. See Central Authentication Service (CAS)#Configuration properties details for more details.
  If SSO is used and SSO authentication succeed, the special `Identity` object will be created and saved into shared state map (Map, which is shared between all login modules), so that this Identity object can be used by `JBossAS7LoginModule` or other login modules in the JAAS chain.
- `JBossAS7LoginModule` - Most important login module, which is normally used to perform whole authentication by itself. First it checks if `Identity` object has been already created and saved into sharedState map by previous login modules (like `SSODelegateLoginModule`, `CustomMembershipLoginModule` or `SharedStateLoginModule`). If not, it triggers real authentication of user with usage of `Authenticator` interface and it will use `Authentication.validateUser(Credential[] credentials)` which performs real authentication of username and password against `OrganizationService` and portal identity database. See Authenticator and RolesExtractor for details about `Authenticator` and about `Identity` objects.
  In the `JbossAS7LoginModule.commit` method, the `Identity` object is registered to `IdentityRegistry`, which will be used later for authorization. Also some JAAS principals (`UserPrincipal` and `RolesPrincipal`) and assigned to our authenticated Subject. This is needed for JBoss AS server, so that it can properly recognize name of logged user and his roles on JBoss AS level.

There is couple of other login modules, which are not active by default, but you can add them if you find them useful.

- `CustomMembershipLoginModule`- special login module, which can be used to add user to some existing groups during the successful login of this user. The group name is configurable and by default is `/platform/users` group. The login module is not used because in normal environment, users are already in the `/platform/users` group. It is useful only for some special setups like read-only LDAP, where groups of ldap user are taken from ldap tree so that users may not be in the `/platform/users` group, which is needed for successful authorization.
  Note that CustomMembershipLoginModule can't be first login module in LM chain because it assumes that Identity object is already available in shared state. So there are those possible cases:
  - For non-SSO case, you may need to chain this LM with other login modules, which can be used to establish Identity and add it into shared state. Those LM can be `InitSharedStateLoginModule` and `SharedStateLoginModule`. See below.
  - For SSO case, you can simply add CustomMembershipLoginModule between `SSODelegateLoginModule` and `JBossAS7LoginModule`.

Configuration example with CustomMembershipLoginModule and disabled SSO:

```
<login-module code="org.exoplatform.web.security.InitSharedStateLoginModule" flag="required">
  <module-option name="portalContainerName" value="portal"/>
  <module-option name="realmName" value="gatein-domain"/>
</login-module>
<login-module code="org.exoplatform.services.security.jaas.SharedStateLoginModule"
flag="required">
  <module-option name="portalContainerName" value="portal"/>
  <module-option name="realmName" value="gatein-domain"/>
</login-module>
<login-module code="org.exoplatform.services.organization.idm.CustomMembershipLoginModule"
flag="required">
  <module-option name="portalContainerName" value="portal"/>
  <module-option name="realmName" value="gatein-domain"/>
  <module-option name="membershipType" value="member" />
  <module-option name="groupId" value="/platform/users" />
</login-module>
<login-module code="org.exoplatform.services.security.j2ee.JBossAS7LoginModule" flag="required">
  <module-option name="portalContainerName" value="portal"/>
  <module-option name="realmName" value="gatein-domain"/>
</login-module>
```

And now configuration example with enabled SSO:

```
<login-module code="org.gatein.sso.integration.SSODelegateLoginModule" flag="required">
  <module-option name="enabled" value="${gatein.sso.login.module.enabled}" />
  <module-option name="delegateClassName" value="${gatein.sso.login.module.class}" />
  <module-option name="portalContainerName" value="portal" />
  <module-option name="realmName" value="gatein-domain" />
  <module-option name="password-stacking" value="useFirstPass" />
</login-module>
<login-module code="org.exoplatform.services.organization.idm.CustomMembershipLoginModule"
flag="required">
  <module-option name="portalContainerName" value="portal"/>
  <module-option name="realmName" value="gatein-domain"/>
  <module-option name="membershipType" value="member" />
  <module-option name="groupId" value="/platform/users" />
</login-module>
<login-module code="org.exoplatform.services.security.j2ee.JBossAS7LoginModule" flag="required">
  <module-option name="portalContainerName" value="portal"/>
  <module-option name="realmName" value="gatein-domain"/>
</login-module>
```

- `InitSharedStateLoginModule` - It can read credentials from JAAS callback and add them into shared state. It's intended to be used in JAAS chain before `SharedStateLoginModule`
- `SharedStateLoginModule`- It reads username and password from sharedState map from attributes `javax.security.auth.login.name` and `javax.security.auth.login.password` . Then it calls `Authenticator.validateUser(Credential[] credentials)`, to perform authentication of username and password against OrganizationService and portal identity database. Result of successful authentication is object `Identity`, which is saved to sharedState map .

Creating your own login module

Before creating your own login module, it is recommended you study source code of existing login modules to better understand the whole JAAS authentication process. You need to have good knowledge so that you can properly decide where your login module should be placed and if you need to replace some existing login modules or simply attach your own module to existing chain.

There are actually two levels of authentication and the overall result of JAAS authentication should properly handle both these cases:

- Authentication at application server level.
- Authentication at GateIn Portal level.

Authentication at application server level

Application server needs to properly recognize that user is successfully logged and it has assigned his JAAS roles. Unfortunately this part is not standardized and is specific for each AS. For example in JBoss AS, you need to ensure that JAAS Subject has assigned principal with username (UserPrincipal) and also RolesPrincipal, which has name "Roles" and it contains list of JAAS roles. This part is actually done in `JbossLoginModule.commit()`. In Tomcat, this flow is little different, which means Tomcat has it is own `TomcatLoginModule`.

After successful authentication, the user needs to be at least in JAAS role *users* because this role is declared in `web.xml` as you saw above. The JAAS roles are extracted by the special algorithm from GateIn Portal memberships. See below in section with RolesExtractor.

Authentication at portal level

Login process needs to create special object `org.exoplatform.services.security.Identity` and register this object into `IdentityRegistry` component of GateIn Portal. This `Identity` object should encapsulate username of authenticated user, memberships of this user and JAAS roles. Identity object can be easily created with the `Authenticator` interface as shown below.

Authenticator and RolesExtractor

`Authenticator` is an important component in the authentication process. Actually interface `org.exoplatform.services.security.Authenticator` looks like this:

```
public interface Authenticator
{

   /**
    * Authenticate user and return userId.
    *
    * @param credentials - list of users credentials (such as name/password, X509
    *          certificate etc)
    * @return userId
    */
   String validateUser(Credential[] credentials) throws LoginException, Exception;

   /**
    * @param userId.
    * @return Identity
    */
   Identity createIdentity(String userId) throws Exception;

}
```

The `validateUser` method is used to check whether given credentials (username and password) are really valid. So it performs real authentication. It returns the username if credentials are correct. Otherwise, `LoginException` is thrown.

The `createIdentity` method is used to create instance of the `org.exoplatform.services.security.Identity` object, which encapsulates all important information about single user like:

- Username.
- Set of Memberships (`MembershipEntry` objects) which the user belongs to. `Membership` is object, which contains information about `membershipType` (manager, member, validator, and more) and about `group` (/platform/users, /platform/administrators, /partners, /organization/management/executiveBoard, and more).
- Set of Strings with JAAS roles of given user. JAAS roles are simple Strings, which are mapped from MembershipEntry objects. There is another special component `org.exoplatform.services.security.RolesExtractor`, which is used to map JAAS roles from MembershipEntry objects. RolesExtractor interface looks like this:

```
public interface RolesExtractor
{

   /**
    * Extracts J2EE roles from userId and|or groups the user belongs to both
    * parameters may be null
    *
    * @param userId
    * @param memberships
    */
   Set<String> extractRoles(String userId, Set<MembershipEntry> memberships);
}
```

  The default implementation named `DefaultRolesExtractorImpl` is based on special algorithm, which uses name of role from the root of the group (for example you have JAAS role "organization" for the "/organization/management/something" role). The only exception is "platform" group where the second level is used as the group name. For example, from "/platform/users" group, you have the JAAS role "users".

Assuming that you have user `root`, which has memberships `member:/platform/users`, `manager:/platform/administrators`, `validator:/platform/managers`, `member:/partners`, `member:/customers/acme`, `member:/organization/management/board`. In this case, you will have JAAS roles: `users`, `administrators`, `managers`, `partners`, `customers`, `organization`.

Default implementation of Authenticator is `OrganizationAuthenticatorImpl`, which is implementation based on `OrganizationService`. See Organization API .

You can override the default implementation of mentioned interfaces (`Authenticator` and `RolesExtractor`) if the default behavior is not suitable for your needs.

6.1.2 Different authentication workflows

RememberMe authentication

In default login dialog, you can notice that there is "Remember my login" checkbox, which users can use to persist their login on his workstation. Default validity period of RememberMe cookie is 1 day (it is configurable), and so user can be logged for whole day before he needs to reauthenticate again with his credentials.

How does it work

- The user checks the "Remember my login" checkbox on login screen of GateIn Portal, then submits the form.
- HTTP request, such as `http://localhost:8080/portal/login?initialURI=/portal/classic&username=root`, is sent to server.
- Request is processed by the `LoginController` servlet. The servlet obtains instance of `RemindPasswordTokenService` and save user credentials into JCR. It generates and returns special token (key) for later use. Then it creates cookie called `rememberme` and use returned token as value of cookie.

Reauthentication

- After some time, user wants to reauthenticate. It is assumed that his HTTP Session is already expired but his `RememberMe` cookie is still active.
- The user sends the HTTP request to some portal pages (for example, `http://localhost:8080/portal/classic`).
- There is special HTTP filter named `RememberMeFilter` configured in `web.xml`, which checks `rememberme` cookie and then it retrieves credentials of user from `RemindPasswordTokenService`. Now the filter redirects request to `PortalLoginController` and authentication process goes in same way as for normal FORM based authentication.

RemindPasswordTokenService

This is a special service used during the RememberMe authentication workflow. It is configurable in the `GATEIN_HOME/gatein/gatein.ear/portal.war/WEB-INF/conf/common/remindpwd-configura` file. See Authentication Token Configuration for more details.

Another thing is that you can encrypt passwords before store them into JCR. More info is in section Password Encryption.

6.1.3 Authorization

In the previous section, you have learned about JAAS authentication and about login modules. So you know that result of authentication, including:

- JAAS Subject with principals for username (`UserPrincipal`) and for JAAS roles (`RolesPrincipal`).
- Identity object, which encapsulates username, all memberships and all JAAS roles. This Identity object is bound to `IdentityRegistry` component.
  Authorization in GateIn Portal actually happens on two levels:

Servlet container authorization

First round of authorization is servlet container authorization based on secured URL from `web.xml`. You can see above in the `web.xml` snippet that secured URL are accessible only for users from the `users` role:

```
<auth-constraint>
  <role-name>users</role-name>
</auth-constraint>
```

This actually means that your user needs to be in GateIn Portal role `/platform/users` (See Authenticator and RolesExtractor for details). In other words, if the authentication is successful but your user is not in the `/platform/users` group, it means that he is not in JAAS role `users`, which in next turn means that he will have authorization error named *403 Forbidden* thrown by the servlet container. For example in LDAP setup, your users may not be in `/platform/users` by default, but you can use `CustomMembershipLoginModule` to fix this problem. For details see Login modules

You can change the behaviour and possibly add some more `auth-constraint` elements into `web.xml`. However, this protection of resources based on `web.xml` is not standard GateIn Portal way and it is mentioned here mainly for illustration purposes.

Portal level authorization

Second round of authorization is based on the `UserACL` component (See Portal Default Permission Configuration). You can declare access and edit permissions for portals, pages and/or portlets. UserACL is then used to check if our user has particular permissions to access or edit specified resource. Important object with information about roles of our user is mentioned `Identity` object created during the JAAS authentication.

Authorization on portal level looks like this:

- The user sends the HTTP request to some URLs in portal.
- The HTTP request is processed through `SetCurrentIdentityFilter`, which is declared in `gatein/gatein.ear/portal.war/WEB-INF/web.xml`.
- `SetCurrentIdentityFilter` reads username of current user from `HttpServletRequest.getRemoteUser()`. Then it looks for Identity of this user in `IdentityRegistry`, where Identity has been saved during authentication. The found Identity is then encapsulated into the `ConversationState` object and bound into the `ThreadLocal` variable.
- `UserACL` is able to obtain Identity of current user from the `UserACL.getIdentity()` method, which simply calls `ConversationState.getCurrent().getIdentity()` for finding the current Identity bound to `ThreadLocal`. Now `UserACL` has identity of user so that it can perform any security checks.

6.2 Password Encryption

6.2.1 Hashing and salting of passwords in Picketlink IDM

GateIn Portal is using Picketlink IDM framework to store information about identity objects (users/groups/memberships) and more info about this is in PicketLink IDM integration . For better security, Picketlink IDM does not save user passwords into database in plain-text, but it uses `CredentialEncoder`, which encode password and save the encoded form into Picketlink IDM database.
Later when user want to authenticate, he needs to provide his password in plain-text via web login form. Provided password is then encoded and compared with encoded password from Picketlink IDM database. GateIn Portal is then able to authenticate user based on this comparison.

Default implementation of `CredentialEncoder` is using password hashing with MD5 algorithm and storing those MD5 hashes in database. It does not use any salting of passwords. This is not safest solution, but it's backward compatible with previous releases of GateIn Portal before version 3.5, where MD5 password hashing was only possible encoding form. So if you migrate from older release of GateIn Portal, your users will be still able to authenticate.

However if you are starting from fresh database (no migration from previous GateIn Portal release), you may increase security by using better hashing algorithm and especially by enable password salting. See below for details.

Choosing CredentialEncoder implementation

The implementation of CredentialEncoder is configured in file `GATEIN_HOME/gatein/gatein.ear/portal.war/WEB-INF/conf/organization/picketlink-idm`. Usually the most important are options of realm `idm_portal` starting with prefix `credentialEncoder.`. Possible implementations are:

HashingEncoder

This is the default choice. It uses only hashing of passwords with MD5 algorithm without salting. As mentioned previously, it's not safest solution but it's backward compatible with previous GateIn Portal releases, so there are no issues with database migration from previous release. Configuration looks like this:

```
<option>
  <name>credentialEncoder.class</name>
  <value>org.picketlink.idm.impl.credential.HashingEncoder</value>
</option>
<option>
  <name>credentialEncoder.hashAlgorithm</name>
  <value>MD5</value>
</option>
```

DatabaseReadingSaltEncoder

This implementation provides salting of password in addition to hashing. The salt is unique for each user, so it's much more complicated to decrypt password via brute force, if some attacker steal encoded passwords from your database. The salt is generated randomly for each user and stored in Picketlink IDM database as attribute. Random generation of salt ensure that all users have different salts, so even if two users have same password, the encoded password in database will be different for them. Here is configuration example, which is using SHA-256 algorithm for hashing (more secure than MD5) and algorithm SHA1PRNG for generation of random salts.

```
<option>
  <name>credentialEncoder.class</name>
  <value>org.picketlink.idm.impl.credential.DatabaseReadingSaltEncoder</value>
</option>
<option>
  <name>credentialEncoder.hashAlgorithm</name>
  <value>SHA-256</value>
</option>
<option>
  <name>credentialEncoder.secureRandomAlgorithm</name>
  <value>SHA1PRNG</value>
</option>
```

FileReadingSaltEncoder

It also uses hashing and salting, so it's similar like previous encoder. But it's theoretically even more secure, because salts are not stored in Picketlink IDM database together with passwords. Salt of each user is generated from *saltPrefix* and user's username. And *saltPrefix* is read from some file in your filesystem. Configuration can look like this:

```
<option>
  <name>credentialEncoder.class</name>
  <value>org.picketlink.idm.impl.credential.FileReadingSaltEncoder</value>
</option>
<option>
  <name>credentialEncoder.hashAlgorithm</name>
  <value>SHA-256</value>
</option>
<option>
  <name>credentialEncoder.fileLocation</name>
  <value>/salt/mysalt.txt</value>
</option>
```

Please note that specified file `/salt/mysalt.txt` must exist and must be readable by user, which executed GateIn Portal. But file should be properly secured to not be readable by every user of your OS. The file can have some random content phrase, for example *a4564dac2aasddsklklkajdgnioiow* .

So the `FileReadingSaltEncoder` is probably most secure of all options, but in addition to `DatabaseReadingSaltEncoder` you need to set the file with salt.

Important

The `CredentialEncoder` from above is actually used only for encoding of passwords in Picketlink IDM database. It's not used for LDAP. Picketlink IDM LDAP implementation (`LDAPIdentityStore`) is sending passwords to LDAP server in plain form, because password encoding is usually provided by LDAP server itself. For example OpenDS 2 is using SHA1 based hashing of passwords with random generation of user salt (so actually something similar to our `DatabaseReadingSaltEncoder` implementation).

6.2.2 Password encryption of rememberme passwords

Username and passwords stored in clear text

The *Remember Me* feature of GateIn Portal uses a token mechanism to be able to authenticate returning users without requiring an explicit login. However, to be able to authenticate these users, the token needs to store the username and password in clear text in JCR.

Administrators have two options available to ameliorate this risk:

1. The *Remember Me* feature can be disabled by removing the corresponding checkbox in:
    - `JBOSS_HOME/gatein/gatein.ear/portal.war/login/jsp/login.jsp` and
    - `JBOSS_HOME/gatein/gatein.ear/portal.war/groovy/portal/webui/UILoginFor`
2. Passwords can be encoded prior to being saved to the JCR. This option requires administrators to provide a custom subclass of `org.exoplatform.web.security.security.AbstractCodec` and set up a codec implementation with `CookieTokenService`:

3. Create a Java class similar to:

```
package org.example.codec;

import org.exoplatform.container.xml.InitParams;
import org.exoplatform.web.security.security.AbstractCodec;
import org.exoplatform.web.security.security.CookieTokenService;
import org.picocontainer.Startable;

public class ExampleCodec extends AbstractCodec implements Startable
{
   private String simpleParam;
   private CookieTokenService cookieTokenService;

   public ExampleCodec(InitParams params, CookieTokenService cookieTokenService)
   {
      simpleParam = params.getValueParam("encodingParam").getValue();
      this.cookieTokenService = cookieTokenService;
   }

   public void start()
   {
      cookieTokenService.setupCodec(this);
   }

   public void stop()
   {
   }

   /**
    * Very simple encoding algorithm used only for demonstration purposes.
    * You should use stronger algorithm in real production environment!!!
    */
   public String encode(String plainInput)
   {
      return plainInput + simpleParam;
   }

   public String decode(String encodedInput)
   {
      return encodedInput.substring(0, encodedInput.length() - simpleParam.length());
   }

}
```

ExampleCodec is using very simple encoding algorithm used only for demonstration purposes. You should use stronger algorithm in real production environment!!!

4. Compile the class and package it into a `.jar` file. For this example, you will call the `codec-example.jar` file.

5. Create a `conf/portal/configuration.xml` file within the `codec-example.jar` similar to the example below. This allows the portal kernel to find and use the new codec implementation.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<configuration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.exoplaform.org/xml/ns/kernel_1_2.xsd
http://www.exoplaform.org/xml/ns/kernel_1_2.xsd"
xmlns="http://www.exoplaform.org/xml/ns/kernel_1_2.xsd">

<component>
   <key>org.example.codec.ExampleCodec</key>
   <type>org.example.codec.ExampleCodec</type>
   <init-params>
      <value-param>
         <name>encodingParam</name>
         <value>aaa</value>
      </value-param>
   </init-params>
</component>

</configuration>
```

6. Deploy `codec-example.jar` into your `JBOSS_HOME/modules/org/gatein/lib/main` directory and update file `module.xml` of this directory (new jar needs to be added to AS7 modules)
7. Start (or restart) your GateIn Portal.
   Any passwords written to the JCR will now be encoded and not plain text.

6.3 Predefined User Configuration

To specify the initial Organization configuration, the content of `JBOSS_HOME/gatein/gatein.ear/portal.war/WEB-INF/conf/organization/organization-c` should be edited. This file uses the portal XML configuration schema. It lists several configuration plugins.

6.3.1 Plugin for adding users, groups and membership types

The plugin of type `org.exoplatform.services.organization.OrganizationDatabaseInitializer` is used to specify the list of membership types/groups/users to be created.

The `checkDatabaseAlgorithm` initialization parameter determines how the database update is performed.

If its value is set to `entry`, it means that each user, group and membership listed in the configuration is checked each time GateIn Portal is started. If the entry does not exist in the database yet, it is created. If the `checkDatabaseAlgorithm` parameter value is set to `empty`, the configuration data will be updated to the database only if the database is empty.

6.3.2 Membership types

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

See `organization-configuration.xml` for the full content.

```
<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>
     <value>
       <object type="org.exoplatform.services.organization.OrganizationConfig$MembershipType">
         <field name="type">
           <string>validator</string>
         </field>
         <field name="description">
           <string>validator membership type</string>
         </field>
       </object>
     </value>
   </collection>
</field>
```

6.3.3 Groups

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

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

6.3.4 Users

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>
```

6.3.5 Plugin for monitoring user creation

The plugin of type `org.exoplatform.services.organization.impl.NewUserEventListener` specifies which groups all the newly created users should become members of. It specifies the groups and the memberships to use (while the group is just a set of users, a membership type represents a user's role within a group). It also specifies a list of users that should not be processed (for example, administrative users like 'root').

The terms 'membership' and 'membership type' refer to the same thing, and are used interchangeably.

```
<component-plugin>
  <name>new.user.event.listener</name>
  <set-method>addListenerPlugin</set-method>
  <type>org.exoplatform.services.organization.impl.NewUserEventListener</type>
  <description>this listener assign group and membership to a new created user</description>
  <init-params>
    <object-param>
      <name>configuration</name>
      <description>description</description>
      <object type="org.exoplatform.services.organization.impl.NewUserConfig">
        <field  name="group">
          <collection type="java.util.ArrayList">
            <value>
              <object type="org.exoplatform.services.organization.impl.NewUserConfig$JoinGroup">
                <field name="groupId"><string>/platform/users</string></field>
                <field name="membership"><string>member</string></field>
              </object>
            </value>
          </collection>
        </field>
        <field  name="ignoredUser">
          <collection type="java.util.HashSet">
            <value><string>exo</string></value>
            <value><string>root</string></value>
            <value><string>company</string></value>
            <value><string>community</string></value>
          </collection>
        </field>
      </object>
    </object-param>
  </init-params>
</component-plugin>
```

6.4 Authentication Token Configuration

Token Service is used in authentication. The token system prevents user account information being sent in the clear text mode within inbound requests. This increases authentication security.

The token service allows administrators to create, delete, retrieve and clean tokens as required. The service also defines a validity period of any given token. The token becomes invalid once this period expires.

6.4.1 Implement the Token Service API

All token services used in the GateIn Portal authentication must be implemented by subclassing an `AbstractTokenService` abstract class. The following `AbstractTokenService` methods represent the contract between authentication runtime, and a token service implementation.

```
public Token getToken(String id) throws PathNotFoundException, RepositoryException;
public Token deleteToken(String id) throws PathNotFoundException, RepositoryException;
public String[] getAllTokens();
public long getNumberTokens() throws Exception;
public String createToken(Credentials credentials) throws
IllegalArgumentException,NullPointerException;
public Credentials validateToken(String tokenKey, boolean remove) throws NullPointerException;
```

6.4.2 Configure token services

The token services configuration includes specifying the token validity period. The token service is configured as a portal component (in the portal scope, as opposed to the root scope - See Foundations for more information).

In the example below, `CookieTokenService` is a subclass of `AbstractTokenService`, so it has a property which specifies the validity period of the token.

The token service will initialize this validity property by looking for an `init-param` named `service.configuration`.

This property must have three values.

```
<component>
  <key>org.exoplatform.web.security.security.CookieTokenService</key>
  <type>org.exoplatform.web.security.security.CookieTokenService</type>
  <init-params>
   <values-param>
      <name>service.configuration</name>
      <value>jcr-token</value>
      <value>7</value>
      <value>DAY</value>
    </values-param>
  </init-params>
</component>
```

Service name

Amount of time

Unit of time

In this case, the service name is `jcr-token` and the token expiration time is one week.

GateIn Portal supports four time units:

- `SECOND`
- `MINUTE`
- `HOUR`
- `DAY`

6.5 PicketLink IDM integration

GateIn Portal uses the PicketLink IDM component to keep the necessary identity information (users, groups, memberships, and more). While the legacy interfaces are still used (org.exoplatform.services.organization) for identity management, there is a wrapper implementation that delegates to the PicketLink IDM framework. This section does not provide information about PicketLink IDM and its configuration. Refer to http://jboss.org/picketlink/IDM.html) for further information.

It is important to fully understand the concepts behind this framework design before changing the default configuration.

The identity model represented in '*org.exoplatform.services.organization*' interfaces and the one used in *PicketLink IDM* have some major differences.

PicketLink IDM provides greater abstraction. It is possible for groups in *IDM* framework to form memberships with many parents (which requires recursive ID translation), while GateIn Portal model allows only pure tree-like membership structures.

Additionally, GateIn Portal *membership* concept needs to be translated into the IDM *Role* concept. Therefore, the *PicketLink IDM* model is used in a limited way. All these translations are applied by the integration layer.

6.5.1 Configuration files

The main configuration file is `JBOSS_HOME/gatein/gatein.ear/portal.war/WEB-INF/conf/organization/idm-configurat` :

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

   <component>
        <key>org.exoplatform.services.organization.idm.PicketLinkIDMService</key>
      <type>org.exoplatform.services.organization.idm.PicketLinkIDMServiceImpl</type>
      <init-params>
         <value-param>
            <name>config</name>
            <value>war:/conf/organization/idm-config.xml</value>
         </value-param>
         <value-param>
            <name>portalRealm</name>
            <value>realm${container.name.suffix}</value>
         </value-param>
       </init-params>
```

```
   </component>


   <component>
      <key>org.exoplatform.services.organization.OrganizationService</key>

<type>org.exoplatform.services.organization.idm.PicketLinkIDMOrganizationServiceImpl</type>
      <init-params>
      <object-param>
        <name>configuration</name>
        <object type="org.exoplatform.services.organization.idm.Config">
          <field name="useParentIdAsGroupType">
            <boolean>true</boolean>
          </field>

          <field name="forceMembershipOfMappedTypes">
            <boolean>true</boolean>
          </field>

          <field name="pathSeparator">
            <string>.</string>
          </field>

          <field name="rootGroupName">
            <string>GTN_ROOT_GROUP</string>
          </field>

          <field name="groupTypeMappings">
            <map type="java.util.HashMap">
              <entry>
                <key><string>/</string></key>
                <value><string>root_type</string></value>
              </entry>

              <!-- Sample mapping -->
              <!--
              <entry>
                <key><string>/platform/*</string></key>
                <value><string>platform_type</string></value>
              </entry>
              <entry>
                <key><string>/organization/*</string></key>
                <value><string>organization_type</string></value>
              </entry>
              -->

            </map>
          </field>

          <field name="associationMembershipType">
            <string>member</string>
          </field>

          <field name="ignoreMappedMembershipType">
            <boolean>false</boolean>
          </field>
        </object>
      </object-param>
```

```
    </init-params>


   </component>

</configuration>
```

PicketlinkIDMServiceImpl

The `org.exoplatform.services.organization.idm.PicketLinkIDMServiceImpl` service has the following options:

- `config` (value-param) The PicketLink IDM configuration file.
- `hibernate.properties` (properties-param) A list of hibernate properties used to create SessionFactory that will be injected to JBoss Identity IDM configuration registry.
- `hibernate.annotations` A list of annotated classes that will be added to Hibernate configuration.
- `hibernate.mappings` A list of `.xml` files that will be added to hibernate configuration as mapping files.
- `jndiName` (value-param) If the 'config' parameter is not provided, this parameter will be used to perform JNDI lookup for `IdentitySessionFactory`.
- `portalRealm` (value-param) The realm name that should be used to obtain proper `IdentitySession`. The default is `'PortalRealm'`.
- `apiCacheConfig` (value-param) The infinispan configuration file with cache configuration for Picketlink IDM API. It's different for cluster and non-cluster because infinispan needs to be replicated in cluster environment.
- `storeCacheConfig` (value-param)
  The infinispan configuration file with cache configuration for Picketlink IDM IdentityStore. Actually it's used only for LDAP store (not used with default DB configuration). It's different for cluster and non-cluster because infinispan needs to be replicated in cluster environment.

PicketlinkIDMOrganizationServiceImpl

The `org.exoplatform.services.organization.idm.PicketLinkIDMOrganizationServiceImpl` key is a main entrypoint implementing `org.exoplatform.services.organization.OrganizationService` and is dependent on `org.exoplatform.services.organization.idm.PicketLinkIDMService`.

The `org.exoplatform.services.organization.idm.PicketLinkIDMOrganizationServiceImpl` service has the following options defined as fields of object-param of the `org.exoplatform.services.organization.idm.Config` type:

- `defaultGroupType` The name of the PicketLink IDM GroupType that will be used to store groups. The default is `'GTN_GROUP_TYPE'`.
- `rootGroupName` The name of the PicketLink IDM Group that will be used as a root parent. The default is `'GTN_ROOT_GROUP'`.
- `passwordAsAttribute` This parameter specifies if a password should be stored using PicketLink IDM Credential object or as a plain attribute. The default is `false`.
- `useParentIdAsGroupType` This parameter stores the parent ID path as a group type in PicketLink IDM for any IDs not mapped with a specific type in 'groupTypeMappings'. If this option is set to `false` , and no mappings are provided under 'groupTypeMappings', then only one group with the given name can exist in the portal group tree.
- `pathSeparator` When 'userParentIdAsGroupType is set to `true`, this value will be used to replace all "/" characters in IDs. The "/" character is not allowed to be used in group type name in PicketLink IDM.
- `associationMembershipType` If this option is used, then each Membership, created with MembrshipType that is equal to the value specified here, will be stored in PicketLink IDM as simple Group-User association.
- `groupTypeMappings` This parameter maps groups added with portal API as children of a given group ID, and stores them with a given group type name in PicketLink IDM.
  If the parent ID ends with "/*", then all child groups will have the mapped group type. Otherwise, only direct (first level) children will use this type.
  This can be leveraged by LDAP if LDAP DN is configured in PicketLink IDM to only store a specific group type. This will then store the given branch in portal group tree, while all other groups will remain in the database.
- `forceMembershipOfMappedTypes` Groups stored in PicketLink IDM with a type mapped in 'groupTypeMappings' will automatically be members under the mapped parent. Group relationships linked by PicketLink IDM group association will not be necessary.
  This parameter can be set to false if all groups are added via portal APIs. This may be useful with LDAP configuration as, when set to true, it will make every entry added to LDAP appear in portal. This, however, is not true for entries added via GateIn Portal management UI.
- `ignoreMappedMembershipType` If "associationMembershipType" option is used, and this option is set to true, then Membership with MembershipType configured to be stored as PicketLink IDM association will not be stored as PicketLink IDM Role.

Additionally, *PicketlinkIDMOrganizationServiceImpl* uses those defaults to perform identity management operations.

- GateIn Portal User interface properties fields are persisted in Picketlink IDM using those attributes names: firstName, lastName, email, createdDate, lastLoginTime, organizationId, password (if password is configured to be stored as attribute).
- GateIn Portal Group interface properties fields are persisted in Picketlink IDM using those attributes names: label, description.
- GateIn Portal MembershipType interface properties fields are persisted in JBoss Identity IDM using those RoleType properties: description, owner, create_date, modified_date.
  A sample *PicketLink IDM* configuration file is shown below. To understand all the options it contains, please refer to the PicketLink IDM Reference Guide.

```
<jboss-identity xmlns="urn:jboss:identity:idm:config:v1_0_beta"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:schemaLocation="urn:jboss:identity:idm:config:v1_0_alpha
identity-config.xsd">
    <realms>
        <realm>
            <id>PortalRealm</id>
            <repository-id-ref>PortalRepository</repository-id-ref>
            <identity-type-mappings>
                <user-mapping>USER</user-mapping>
            </identity-type-mappings>
        </realm>
    </realms>
    <repositories>
        <repository>
            <id>PortalRepository</id>
            <class>org.jboss.identity.idm.impl.repository.WrapperIdentityStoreRepository</class>
            <external-config/>
            <default-identity-store-id>HibernateStore</default-identity-store-id>
            <default-attribute-store-id>HibernateStore</default-attribute-store-id>
        </repository>
    </repositories>
    <stores>
        <attribute-stores/>
        <identity-stores>
            <identity-store>
                <id>HibernateStore</id>

<class>org.jboss.identity.idm.impl.store.hibernate.HibernateIdentityStoreImpl</class>
                <external-config/>
                <supported-relationship-types>
                    <relationship-type>JBOSS_IDENTITY_MEMBERSHIP</relationship-type>
                    <relationship-type>JBOSS_IDENTITY_ROLE</relationship-type>
                </supported-relationship-types>
                <supported-identity-object-types>
                    <identity-object-type>
                        <name>USER</name>
                        <relationships/>
                        <credentials>
                            <credential-type>PASSWORD</credential-type>
                        </credentials>
                        <attributes/>
                        <options/>
                    </identity-object-type>
                </supported-identity-object-types>
                <options>
                    <option>
                        <name>hibernateSessionFactoryRegistryName</name>
                        <value>hibernateSessionFactory</value>
                    </option>
                    <option>
                        <name>allowNotDefinedIdentityObjectTypes</name>
                        <value>true</value>
                    </option>
                    <option>
                        <name>populateRelationshipTypes</name>
```

```
                    <value>true</value>
                </option>
                <option>
                    <name>populateIdentityObjectTypes</name>
                    <value>true</value>
                </option>
                <option>
                    <name>allowNotDefinedAttributes</name>
                    <value>true</value>
                </option>
                <option>
                    <name>isRealmAware</name>
                    <value>true</value>
                </option>
            </options>
        </identity-store>
    </identity-stores>
</stores>
</jboss-identity>
```

6.6 LDAP integration

For ease of readability the following section uses the notational device `ID_HOME` to represent the file path `GATEIN_HOME/gatein/gatein.ear/portal.war/WEB-INF/conf/organization/`, as this directory is the root of all GateIn Portal's identity-related configuration.

LDAP (Lightweight Directory Access Protocol) is a set of open protocols used to access centrally stored information over a network. It is based on the X.500 standard for directory sharing, but is less complex and resource-intensive.

Using a client/server architecture, LDAP provides a reliable means to create a central information directory accessible from the network. When a client attempts to modify information within this directory, the server verifies the user has permission to make the change, and then adds or updates the entry as requested. To ensure the communication is secure, the Secure Sockets Layer (*SSL*) or Transport Layer Security (*TLS*) cryptographic protocols can be used to prevent an attacker from intercepting the transmission.

LDAP provides the protocols required to manage the data stored in a Directory Server. A Directory Server contains information about resources available (user accounts and printers for example) and their location on the network.

The following table is a list of Directory Servers that are supported and certified in GateIn Portal.

6.6.1 Supported and Certified Directory Servers

Directory Server	Version
OpenDS	1.2 and 2.0
OpenLDAP	2.4
Red Hat Directory Server	7.1
Sun Java System Directory Server	6.1
Microsoft Active Directory	Windows Server 2008

Examples

GateIn Portal includes several example LDAP configuration **.xml** files and **.ldif** (LDAP Data Interchange Format) data files.
These examples are in the `ID_HOME/picketlink-idm/examples` directory and can be deployed in a testing environment to assist in configuring LDAP.

6.6.2 LDAP Set Up

1. Install your **LDAP** server by following the installation instructions provided for the product you are using.
   If you are installing the **Red Hat Directory Server**, you should refer to the Installation Guide at http://docs.redhat.com/docs/en-US/Red_Hat_Directory_Server/index.html.
   If you are using a third party directory server (**OpenDS**, **OpenLDAP** or **Microsoft Active Directory**), refer the appropriate documentation for that product.
   The following values provide an example of working configuration settings for the different Directory Servers:

Directory Server	root user Distinguished Name (DN)	Password	Port	Admin Port	Base DN	
RHDS and OpenDS	cn=Directory Manager	password	1389	4444	dc=example,dc=com	
MSAD	CN=Users					
OpenLDAP	cn=Manager,dc=example,dc=com	secret	1389		dc=example,dc=com	

   These, and other appropriate settings, should be adjusted to suit your circumstances.
2. **Optional**: Import an ldif file and populate the Directory Server.
3. Start the Directory Server.

6.6.3 LDAP in Read-only Mode

This section will show you how to add LDAP in read-only mode. This means that user data entries (both pre-existing, and newly added through the GateIn Portal User Interface) will be consumed though the Directory Server and LDAP services, but written to the underlying database. The only exception is that passwords updated via the UI will also be propagated into the appropriate LDAP entry.

Set up LDAP read-only Mode

1. Open the ID_HOME/idm-configuration.xml file.
   GateIn Portal uses the PicketLink IDM framework as the underlying identity storage system, hence all the configurations use dedicated Picketlink settings.
2. Comment out the default Picketlink config value:
   `<value>war:/conf/organization/picketlink-idm/picketlink-idm-config.xml</valu`
3. Uncomment the appropriate sample configuration values as described below, depending on which Directory Server you are implementing:

- **Red Hat Directory Server or OpenDS**
  1. Uncomment the line under *Read Only "ACME" LDAP Example*:

```
<!--Read Only "ACME" LDAP Example-->
<value>war:/conf/organization/picketlink-idm/examples/picketlink-idm-LDAP-acme-c
```

  2. Uncomment the groupTypeMappings under *Uncomment for ACME LDAP example*:

```
<!-- Uncomment for ACME LDAP example -->
<entry>
  <key><string>/acme/roles/*</string></key>
  <value><string>acme_roles_type</string></value>
</entry>
<entry>
  <key><string>/acme/organization_units/*</string></key>
  <value><string>acme_ou_type</string></value>
</entry>
```

  Refer to Read Only groupTypeMappings for more information about how these groupTypeMappings operate.
- **Microsoft Active Directory**
  1. Uncomment the line under *MSAD Read Only "ACME" LDAP Example*:

```
<!--MSAD Read Only "ACME" LDAP Example-->
<value>war:/conf/organization/picketlink-idm/examples/picketlink-idm-msad-reador
```

  2. Uncomment the groupTypeMappings under *Uncomment for MSAD ReadOnly LDAP example*:

```
<!-- Uncomment for MSAD ReadOnly LDAP example -->
<entry>
  <key><string>/acme/roles/*</string></key>
  <value><string>msad_roles_type</string></value>
</entry>
```

  Refer to Read Only groupTypeMappings for more information about how these groupTypeMappings operate.

- **OpenLDAP**
  1. If you have not done so already, install your LDAP server. Refer to LDAP Set Up for some assistance.
  2. Uncomment the line under *OpenLDAP ReadOnly "ACME" LDAP Example*:

```
<!--OpenLDAP ReadOnly "ACME" LDAP Example-->
<value>war:/conf/organization/picketlink-idm/examples/picketlink-idm-openLDAP-ac
```

  3. Uncomment the groupTypeMappings under *Uncomment for ACME LDAP example*:

```
<!-- Uncomment for ACME LDAP example -->
<entry>
  <key><string>/acme/roles/*</string></key>
  <value><string>acme_roles_type</string></value>
</entry>
<entry>
  <key><string>/acme/organization_units/*</string></key>
  <value><string>acme_ou_type</string></value>
</entry>
```

     Refer to Read Only groupTypeMappings for more information about how these groupTypeMappings operate.

4. To use a different LDAP server or directory data, edit the DS-specific .xml file you uncommented in **Substep 3a** above and change the values to suit your requirements.
   Refer to the list in LDAP configuration for some examples or refer to the product-specific documentation for more information.
5. Start the server.
6. Navigate to the portal homepage (http://localhost:8080/portal) and log in as an administrator.
7. Navigate to **Group Organization Users and groups management**.
   1. Create a new group called *acme* under the root node.
      - **For RHDS, OpenDS and OpenLDAP**:
        Create two sub-groups called *roles* and *organization_units*.
      - **For MSAD:**
        Create a subgroup called *roles*.

Users defined in LDAP should be visible in "*Users and groups management*" and groups from LDAP should be present as children of */acme/roles* and */acme/organization_units*.
More information about configuration can be found in PicketLink IDM integration and in the PicketLink project Reference Guide.

In read-only LDAP setup, your LDAP users are usually not member of group */platform/users* by default. This means that they are not authorized to see non-public content of portal (like top address bar). To address this issue, we have special login module **CustomMembershipLoginModule**, which automatically adds each user to group */platform/users* after his successful login. See Existing login modules for details about setup of this login module.

Another option is to use **CoreOrganizationInitializer** plugin which will enforce running OrganizationService listeners, as one of the listeners is automatically adding users into group */platform/users* . See https://github.com/gatein/gatein-toolbox/tree/master/CoreOrganizationInitializer and especially it's README.txt file for more info.

6.6.4 LDAP as Default Store

Follow the procedure below to set LDAP up as the default identity store for GateIn Portal. All default accounts and some of groups that comes with GateIn Portal will be created in the LDAP store.

The LDAP server will be configured to store part of the GateIn Portal group tree. This means that groups under specified part of the tree will be stored in directory server while all others will be stored in database.

Set up LDAP as Default Indentity Store

1. If you have not done so already, install your LDAP server. Refer to LDAP Set Up for some assistance.
2. Open the ID_HOME/idm-configuration.xml file.
   GateIn Portal uses the PicketLink IDM framework as the underlying identity storage system, hence all the configurations use dedicated Picketlink settings.
3. Comment out the default Picketlink config value:
   war:/conf/organization/picketlink-idm/picketlink-idm-config.xml

4. Uncomment the appropriate LDAP configuration entry depending on your LDAP server:
    - **For RHDS and OpenDS**
        1. Expose the entry under *Sample LDAP config*:

```
<!--Sample LDAP config-->
<value>war:/conf/organization/picketlink-idm/examples/picketlink-idm-ldap-config
```

    - **For MSAD**
        1. Expose the entry under *MSAD LDAP Example*:

```
<!--MSAD LDAP Example-->
<value>war:/conf/organization/picketlink-idm/examples/picketlink-idm-msad-config
```

            - **To use SSL encryption with MSAD:**
            - Open the `ID_HOME/picketlink-idm/examples/picketlink-idm-msad-config.xml`
            - Ensure the following entries are uncommented and that the path to the truststore file and password are correct:

```
<option>
  <name>customSystemProperties</name>
  <value>javax.net.ssl.trustStore=/path/to/truststore</value>
  <value>javax.net.ssl.trustStorePassword=password</value>
</option>
```

            You can import a custom certificate by replacing the certificate and truststore details in the following command:

```
keytool -import -file  certificate -keystore truststore
```

    - **For OpenLDAP**
        1. Expose the entry under *OpenLDAP LDAP config*:

```
<!--OpenLDAP LDAP config-->
<value>war:/conf/organization/picketlink-idm/examples/picketlink-idm-openLDAP-co
```

5. Uncomment the **groupTypeMappings** under *Uncomment for sample LDAP configuration*:

```
<entry>
  <key><string>/platform/*</string></key>
  <value><string>platform_type</string></value>
</entry>
<entry>
  <key><string>/organization/*</string></key>
  <value><string>organization_type</string></value>
</entry>
```

   Refer to Default groupTypeMappings for more information about how these groupTypeMappings operate.
6. Uncomment **ignoreMappedMembershipTypeGroupList** under *Uncomment for sample LDAP config*

```
<value>
  <string>/platform/*</string>
</value>
<value>
  <string>/organization/*</string>
</value>
```

   Usually you need to uncomment same roles, which you are using for LDAP mapping. The point is that user memberships under those groups will be created in Picketlink IDM only as relationships and not as roles. Without uncommenting, memberships will be used as both relationships and roles, which could cause that you may see some duplicated records in UI.
7. To use a different LDAP server or directory data, edit the DS-specific **.xml** file you uncommented in **Step 4** above and change the values to suit your requirements.
   Refer to the list in LDAP configuration for some examples or refer to the product-specific documentation for more information.

Now after server startup, all GateIn Portal groups under `/platform` and under `/organization` groups (for example `/platform/users` , `/platform/administrators` , `/organization/management/executive-board` etc.) will be mapped to LDAP tree. The location of groups in LDAP tree is configurable via parameter `ctxDNs` in Picketlink IDM configuration file as described below.

6.6.5 Examples

LDAP configuration

The following settings are stored in the Picketlink configuration file that is nominated in the `idm-configuration.xml` file of your deployment (under the `config` parameter of the `PicketLinkIDMService` component):
This file could be:

- The default `picketlink-idm-config.xml`.
- One of the three example configuration files discussed in LDAP Set Up :
  - `picketlink-idm-LDAP-acme-config.xml`
  - `picketlink-idm-msad-readonly-config.xml`
  - `picketlink-idm-openLDAP-acme-config.xml`
- A custom file created by modifying one of the above files.

Configuration options

- **ctxDNs** - This is the DN that will be used as context for IdentityObject searches. More than one value can be specified.Some examples are:
  - ou=People,o=acme,dc=example,dc=com
  - ou=Roles,o=acme,dc=example,dc=com
  - ou=OrganizationUnits,o=acme,dc=example,dc=com
  - **MSAD:** CN=Users,DC=test,DC=domain (in two places)
- **providerURL** - The LDAP server connection URL. Formatted as "LDAP://localhost:<PORT>". The default setting is:LDAP://localhost:1389.
  MSAD: Should use SSL connection (LDAPs://xxx:636) for password update or creation to work.
- **adminDN** - The LDAP entry used to connect to the server. Some possible values are:
  - **RHDS or OpenDS:** cn=Directory Manager
  - **OpenLDAP:** cn=Manager,dc=my-domain,dc=com
  - **MSAD:** TEST\Administrator
- **adminPassword** - The password associated with the **adminDN**.
- **customSystemProperties** - This option defines the values needed to use SSL encryption with LDAP.

Read Only groupTypeMappings

The groupTypeMappings exposed in the `idm-configuration.xml` file correspond to identity-object-type values defined in the DS-specific configuration file (referenced in *Sub-step 3a* of the DS-specific procedure above).

For RHDS, OpenDS and OpenLDAP the `picketlink-idm-LDAP-acme-config.xml` and `picketlink-idm-openLDAP-acme-config.xml` files contain the following values:

```
<repository>
   <id>PortalRepository</id>
   <class>org.picketlink.idm.impl.repository.FallbackIdentityStoreRepository</class>
   <external-config/>
   <default-identity-store-id>HibernateStore</default-identity-store-id>
   <default-attribute-store-id>HibernateStore</default-attribute-store-id>
   <identity-store-mappings>
     <identity-store-mapping>
       <identity-store-id>PortalLDAPStore</identity-store-id>
       <!-- Comment #1 -->
       <identity-object-types>
         <identity-object-type>USER</identity-object-type>
         <identity-object-type>acme_roles_type</identity-object-type>
         <identity-object-type>acme_ou_type</identity-object-type>
       </identity-object-types>
       <!-- Comment #2 -->
       <options>
         <option>
           <name>readOnly</name>
           <value>true</value>
         </option>
       </options>
     </identity-store-mapping>
   </identity-store-mappings>
   <options>
     <option>
       <name>allowNotDefinedAttributes</name>
       <value>true</value>
     </option>
   </options>
</repository>
```

Comment 1: The PicketLink IDM configuration file dictates that users and those two group types be stored in LDAP.
Comment 2: An additional option defines that nothing else (except password updates) should be written there.

All groups under **/acme/roles** will be stored in PicketLink IDM with the **acme_roles_type** group type name and groups under **/acme/organization_units** will be stored in PicketLink IDM with **acme_ou_type group** type name.

For MSAD, the identity-object-types values in `picketlink-idm-msad-readonly-config.xml` change to:

```
<identity-store-id>PortalLDAPStore</identity-store-id>
 <identity-object-types>
   <identity-object-type>USER</identity-object-type>
   <identity-object-type>msad_roles_type</identity-object-type>
 </identity-object-types>
```

The difference is that this configuration maps only one group type and points to the same container in LDAP for both users and mapped groups.

Default groupTypeMappings

The groupTypeMappings exposed in the `idm-configuration.xml` file correspond to identity-object-type values defined in the DS-specific configuration file (referenced in *Sub-step 3a* of the DS-specific procedure above).

All of the supported LDAP configurations use the following values when implemented as the default identity store:

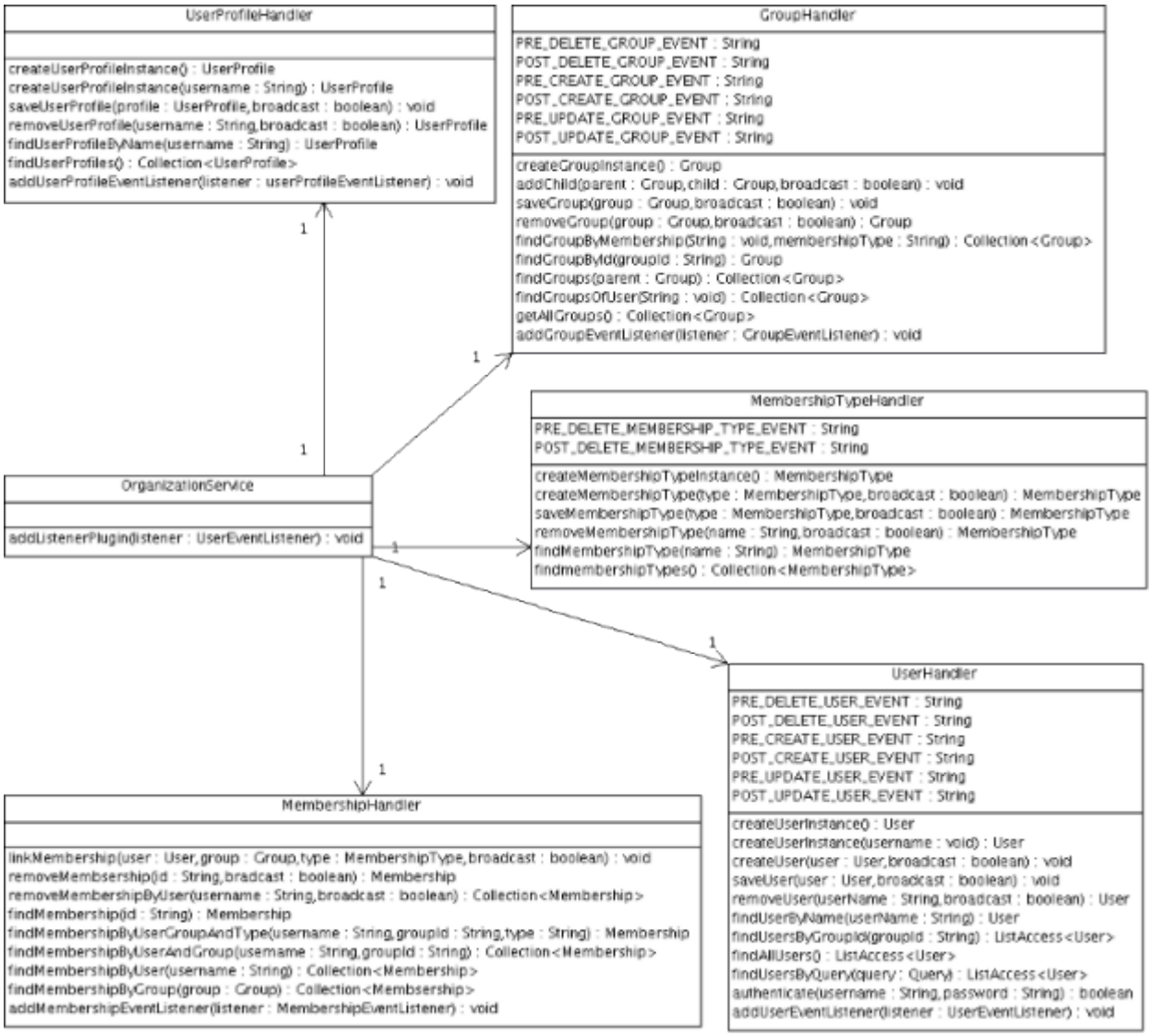
```
<repository>
   <id>PortalRepository</id>
   <class>org.picketlink.idm.impl.repository.FallbackIdentityStoreRepository</class>
   <external-config/>
   <default-identity-store-id>HibernateStore</default-identity-store-id>
   <default-attribute-store-id>HibernateStore</default-attribute-store-id>
   <identity-store-mappings>
     <identity-store-mapping>
       <identity-store-id>PortalLDAPStore</identity-store-id>
       <!-- Comment #1 -->
       <identity-object-types>
         <identity-object-type>USER</identity-object-type>
         <identity-object-type>platform_type</identity-object-type>
         <identity-object-type>organization_type</identity-object-type>
       </identity-object-types>
       <options/>
     </identity-store-mapping>
   </identity-store-mappings>
   <options>
     <option>
       <name>allowNotDefinedAttributes</name>
       <value>true</value>
     </option>
   </options>
 </repository>
 <repository>
   <id>DefaultPortalRepository</id>
   <class>org.picketlink.idm.impl.repository.WrapperIdentityStoreRepository</class>
   <external-config/>
   <default-identity-store-id>HibernateStore</default-identity-store-id>
   <default-attribute-store-id>HibernateStore</default-attribute-store-id>
 </repository>
```

Comment 1: The groupTypeMappings define that all groups under **/platform** should be stored in PicketLink IDM with the *platform_type* group type name and groups under **/organization** should be stored in PicketLink IDM with *organization_type* group type name.

The PicketLink IDM configuration file repository maps users and those two group types as stored in LDAP.

6.7 Organization API

The `exo.platform.services.organization` package has five main components: user, user profile, group, membership type and membership. There is an additional component that serves as an entry point into Organization API - `OrganizationService` component that provides the handling functionality for the five components.



The `User` component contains basic information about the user, such as username, password, first name, last name, and email. The `User Profile` component contains extra information about a user, such as user's personal information, and business information. You can also add additional information about a user if your application requires it. The `Group` component contains a group graph. The `Membership Type` component contains a list of predefined membership types. Finally, the `Membership` component connects a User, a Group and a Membership Type.

A user can have one or more memberships within a group, for example: the user A can have two memberships: 'member' and 'admin' in the group /user. A user belongs to a group if he has at least one membership in that group.

Exposing the Organization API to developers who have access to the handler objects provided by the OrganizationService component. These handler objects are used to manage each of the five components, including `UserHandler`, `UserProfileHandler`, `GroupHandler`, `MembershipTypeHandler`, and `MembershipHandler`.

The five central API components are really designed like persistent entities, and handlers are really specified like data access objects (DAO).

Organization API simply describes a contract, meaning it is not a concrete implementation. The described components are interfaces, allowing different concrete implementations. Practically, it means that you can replace the existing implementation with a different one.

6.8 Access User Profile

The following code retrieves the details for a logged-in user:

```
// Alternative context: WebuiRequestContext context = WebuiRequestContext.getCurrentInstance() ;
PortalRequestContext context = PortalRequestContext.getCurrentInstance() ;
// Get the id of the user logged
String userId = context.getRemoteUser();

// Retrieve OrganizationService but it works only from WebUI code. See variants below in
documentation
OrganizationService orgService = getApplicationComponent(OrganizationService.class) ;

// Request the information from OrganizationService:
if (userId != null)
{
  User user = orgService.getUserHandler().findUserByName(userId) ;
  if (user != null)
  {
    String firstName = user.getFirstName();
    String lastName = user.getLastName();
    String email = user.getEmail();
  }
}
```

Below are two alternatives for retrieving the Organization Service:

- The first alternative

```
OrganizationService service = (OrganizationService)

ExoContainerContext.getCurrentContainer().getComponentInstanceOfType(OrganizationService.cla
```

- The second alternative

```
OrganizationService service = (OrganizationService)
     PortalContainer.getInstance().getComponentInstanceOfType(OrganizationService.class);
```

Both alternatives are probably better then `OrganizationService orgService = getApplicationComponent(OrganizationService.class)` because you can use them from your own portlets or servlet/portlet filters. Variant with `getApplicationComponent` variant works only from WebUI.

6.9 Single-Sign-On (SSO)

GateIn Portal provides some forms of Single Sign On (SS0) as an integration and aggregation platform.

When logging into the portal, users gain access to many systems through portlets using a single identity. In many cases, however, the portal infrastructure must be integrated with other SSO enabled systems. There are many different Identity Management solutions available. In most cases, each SSO framework provides a unique way to plug into a Java EE application.

6.9.1 Prerequisites

In this tutorial, the SSO server (like CAS, OpenAM or JOSSO) is usually installed in a Tomcat installation. Tomcat can be obtained from http://tomcat.apache.org. GateIn Portal is usually running on JBoss AS 7. All the details about particular SSO solutions are available in individual subpages.

All the packages required for setup can be found in a latest zip file located under this directory. At this moment, latest version is here. In this document, GATEIN_SSO_HOME is called as the directory where the ZIP file is extracted.

Users are advised to not run any portal extensions that could override the data when manipulating the `gatein.ear` file directly.

6.9.2 Central Authentication Service (CAS)

This Single Sign On plugin enables seamless integration between GateIn Portal and the CAS Single Sign On Framework. Details about CAS can be found here.

The integration consists of two parts:

- The first consists of installing or configuring a CAS server.
- The second consists of setting up the portal to use the CAS server.

How the integration works

Login workflow

The whole authentication process with CAS integration works like this:

1. Non-authenticated user is on GateIn Portal page. He wants to authenticate and so he clicks *Sign in* link. Normally this will show GateIn Portal login dialog, but when SSO integration is enabled, it will redirect user to special marker URL like: http://localhost:8080/portal/sso
2. There is special interceptor (Servlet filter) **LoginRedirectFilter** on GateIn Portal side, which is able to handle */portal/sso* URL and it redirects user to CAS server page.
3. So user is redirected to CAS login page http://localhost:8888/cas/login . He is asked to fill his credentials and authenticate on CAS server side .
   It's up to CAS administrator how he configure authentication on CAS side to prove user credentials. For example you can configure CAS to obtain credentials from some external DB or LDAP server. However GateIn Portal SSO component provides special authentication plugin, which can be deployed and configured on CAS server. This plugin is verifying user credentials by connect to existing GateIn Portal instance via back channel with usage of REST over HTTP protocol. In this case, GateIn Portal will serve Rest authentication callback request and it will verify user identity against GateIn Portal's own identity storage provided by portal *OrganizationService* (typically based on Picketlink IDM database) and send response to CAS server with the result.
   Please note that using of this Authentication plugin is not mandatory for integration with GateIn Portal. You can use whatever you want for authentication on CAS side (like external DB or LDAP) as already mentioned. But using of GateIn Portal SSO authentication plugin has one important advantage, that you need to have only single identity storage for storing users/groups/roles. So if you add new user via GateIn Portal UI, you are immediately able to authenticate this user with CAS server. More info about how to configure CAS to use Authentication plugin is in Authentication plugin setup.
4. Once user is authenticated on CAS side, he is redirected back to GateIn Portal to URL like http://localhost:8080/portal/initiatelogin . There is another filter **InitiateLoginFilter**, which intercepts */portal/initiatelogin* URL. This filter will obtain CAS ticket attached in HTTP request inside parameter *ticket* and it delegates validation of this ticket to configured **CASAgent** component. CAS agent then validates ticket by sending validation request to CAS server via back channel. Response from CAS contains username of the user who authenticated on CAS side in step 3.
5. After SSO validation, *InitiateLoginFilter* redirects user to portal login URL like http://localhost:8080/portal/login, which performs JAAS authentication. **SSOLoginModule** will recognize if user has been successfully validated by *CASAgent* and if it's the case, it will obtain data about user (groups, memberships) from *OrganizationService* and encapsulate them into **Identity** object. Another Login module **JBossAS7LoginModule** (or **TomcatLoginModule**) is able to finish authentication by establish JAAS *Subject* and save *Identity* object to *IdentityRegistry* (See Authentication and Authorization intro#Login modules for more details).
   So you can notice that CAS is used only for authenticating user, but informations about roles and group memberships are still obtained from GateIn Portal OrganizationService (Picketlink IDM database)
6. After successful JAAS authentication is user redirected to GateIn Portal and he is properly authenticated and authorized now.

Logout workflow

1. Authenticated user clicks to *Sign out* link.
2. There is interceptor **CASLogoutFilter**, which recognize logout request and it redirects user to CAS logout page http://localhost:8888/cas/logout
3. CAS server will logout user on it's side and invalidate CAS cookie *CASTGC* . Then it redirects user back to GateIn Portal (Proper redirection to portal requires another change in CAS server configuration as mentioned in Logout redirection setup).
4. Request to GateIn Portal will finish logout process on it's side and user will be redirected to GateIn Portal anonymous page. So note that if *CASLogoutFilter* is enabled, user is logged out from both GateIn Portal and CAS server.

CAS server

For simplification purpose, we assume that CAS 3.5 will be deployed on Tomcat 7 server, which will listen on *localhost:8888* . GateIn Portal will be deployed on JBoss AS 7, which will listen on *localhost:8080* .

First, set up the server to authenticate against the portal via REST callback.

Obtaining CAS

CAS can be downloaded from http://www.jasig.org/cas/download. The tested version which should work with these instructions is *CAS 3.5*. However, other versions can also work.

Extract the downloaded file into a suitable location. This location will be referred to as `CAS_HOME` in the following instructions.

Modifying the CAS server

To configure the web archive as desired, the simplest way is to make the necessary changes directly in the CAS codebase.

> **Important**
>
> To complete these instructions, and perform the final build step, you will need the Apache Maven 3. You can get it here.

Authentication plugin setup

As already mentioned, this is not mandatory and is needed only if you want setup CAS to make secure authentication callbacks to a RESTful service installed on the remote GateIn Portal

In order for the plugin to function correctly, it needs to be properly configured to connect to this service. This configuration is done via the `cas.war/WEB-INF/deployerConfigContext.xml` file.

1. Open `CAS_HOME/cas-server-webapp/src/main/webapp/WEB-INF/deployerConfigContext.xml`.
2. Replace

```
<!--
 | This is the authentication handler declaration that every CAS deployer will need to
change before deploying CAS
 | into production.  The default SimpleTestUsernamePasswordAuthenticationHandler
authenticates UsernamePasswordCredentials
 | where the username equals the password.  You will need to replace this with an
AuthenticationHandler that implements your
 | local authentication strategy.  You might accomplish this by coding a new such handler
and declaring
 | edu.someschool.its.cas.MySpecialHandler here, or you might use one of the handlers
provided in the adaptors modules.
+-->
<bean

class="org.jasig.cas.authentication.handler.support.SimpleTestUsernamePasswordAuthentication
/>
```

   With the following that is also available in `GATEIN_SSO_HOME/cas/plugin/WEB-INF/deployerConfigContext.xml` (Make sure you set the host, port and context with the values corresponding to your portal). GATEIN_SSO_HOME refers to directory with GateIn Portal SSO artifacts as mentioned in Single-Sign-On (SSO) .

```
<!--
 | This is the authentication handler declaration that every CAS deployer will need to
change before deploying CAS
 | into production.  The default SimpleTestUsernamePasswordAuthenticationHandler
authenticates UsernamePasswordCredentials
 | where the username equals the password.  You will need to replace this with an
AuthenticationHandler that implements your
 | local authentication strategy.  You might accomplish this by coding a new such handler
and declaring
 | edu.someschool.its.cas.MySpecialHandler here, or you might use one of the handlers
provided in the adaptors modules.
+-->
<bean class="org.gatein.sso.cas.plugin.AuthenticationPlugin">
  <property name="gateInProtocol"><value>http</value></property>
  <property name="gateInHost"><value>localhost</value></property>
  <property name="gateInPort"><value>8080</value></property>
  <property name="gateInContext"><value>portal</value></property>
  <property name="httpMethod"><value>POST</value></property>
</bean>
```

3. Copy all jars from `GATEIN_SSO_HOME/cas/plugin/WEB-INF/lib/` into the `CAS_HOME/cas-server-webapp/src/main/webapp/WEB-INF/lib` created directory.

Logout redirection setup

By default on logout the CAS server will display the CAS logout page with a link to return to the portal. To make the CAS server redirect to the portal page after a logout, modify `CAS_HOME/cas-server-webapp/src/main/webapp/WEB-INF/cas-servlet.xml` to include the `followServiceRedirects="true"` parameter:

```
<bean id="logoutController"  class="org.jasig.cas.web.LogoutController"
  p:centralAuthenticationService-ref="centralAuthenticationService"
  p:logoutView="casLogoutView"
  p:warnCookieGenerator-ref="warnCookieGenerator"
  p:ticketGrantingTicketCookieGenerator-ref="ticketGrantingTicketCookieGenerator"
  p:followServiceRedirects="true"/>
```

CAS SSO cookie configuration

CAS server is using special cookie *CASTGC*, which is useful for SSO scenarios with more Service applications. For example, you have single CAS server and 2 GateIn Portal servers configured to use it (let's mark those GateIn Portal instances as **accounts** and **services**). So if you login against CAS server with **accounts** GateIn Portal instance, you don't need to reauthenticate again when you access CAS with **services** GateIn Portal instance. You will be automatically authenticated when click *Sign in* on **services** instance. This is real SSO and it works thanks to CASTGC cookie, which automatically creates new ticket for **services** instance if it recognize that user is already authenticated.

Thing is that CASTGC cookie is secured by default (aka. available only from *https* connections). So by default the "real" SSO doesn't work and you need to reauthenticate when accessing CAS from your **services** instance. So if you really want to support this scenario with more nodes, you have 2 possibilities:

- Use *https* protocol to access your CAS server. This will ensure that secure CASTGC cookie can be viewable by browser. This is recommended approach for production environments. See https://wiki.jasig.org/display/CASUM/Securing+Your+New+CAS+Server for more details.
- Easier workaround (but not recommended in production environment) is to switch CASTGC cookie to be non-secure (ie. cookie won't need secure access through *https* but can be accessed from *http* as well). To achieve this you need to configure in CAS side in file `CAS_HOME/cas-server-webapp/src/main/webapp/WEB-INF/spring-configuration/tic` and switch attribute `cookieSecure` to false. Configuration of cookie generator in this file should look like this:

  ```
  <bean id="ticketGrantingTicketCookieGenerator"
    p:cookieSecure="false"
    p:cookieMaxAge="-1"
    p:cookieName="CASTGC"
    p:cookiePath="/cas" />
  ```

SSO server setup

1. Get an installation of Tomcat 7 from http://tomcat.apache.org/download-70.cgi and extract it into a suitable location (which will be called `TOMCAT_HOME` for these instructions).
   Change the default port to avoid a conflict with the default GateIn Portal (for testing purposes). Edit `TOMCAT_HOME/conf/server.xml` and replace the 8080 port with 8888.

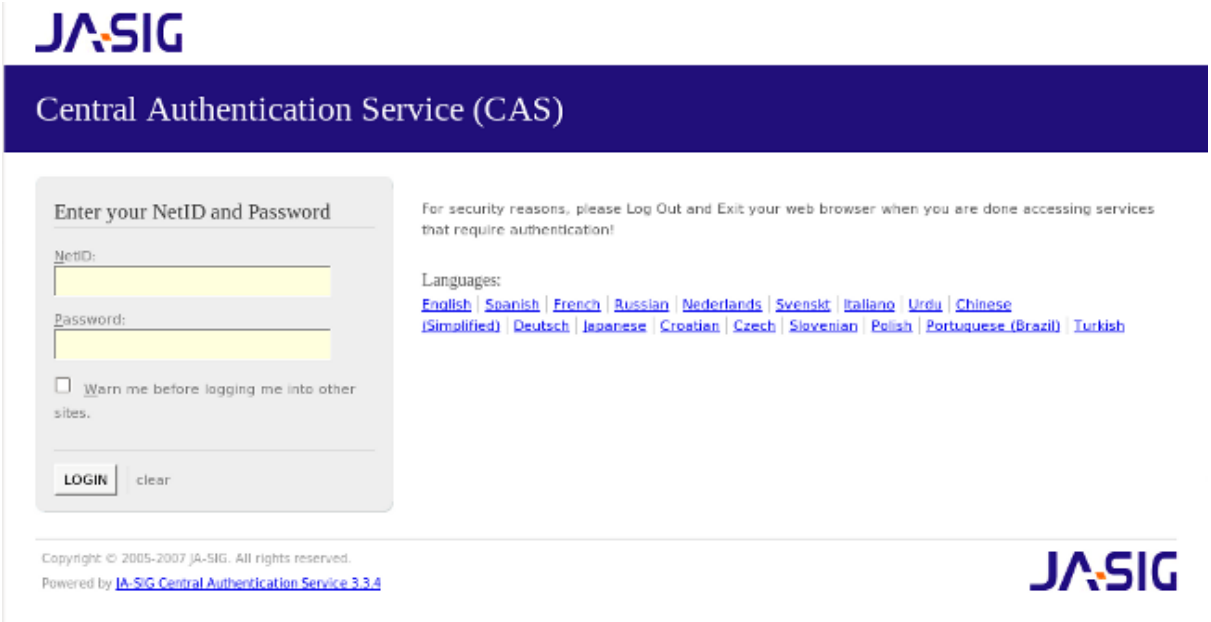
   > **Important**
   >
   > If GateIn Portal is running on the same machine as Tomcat, other ports need to be changed in addition to 8080 to avoid port conflicts. They can be changed to any free port. For example, you can change admin port from 8005 to 8805, and AJP port from 8009 to 8809.

2. Go to `CAS_HOME/cas-server-webapp` and execute the command:

   ```
   mvn clean install -Dmaven.test.skip=true
   ```

3. Copy `CAS_HOME/cas-server-webapp/target/cas.war` into `TOMCAT_HOME/webapps`.
4. Tomcat should start and be accessible at http://localhost:8888/cas.

   > **Important**
   >
   > The login will not be available at this stage if you configure authentication against Authentication plugin as GateIn Portal is not started yet.



Setup the portal

Assuming you have GateIn Portal bundled with JBoss AS7, all you need to do on portal side is to configure couple of configuration properties in file `GATEIN_HOME/standalone/configuration/gatein/configuration.properties` . Find SSO section in this file and change/add properties so that it looks like this:

```
# SSO
gatein.sso.enabled=true
gatein.sso.callback.enabled=${gatein.sso.enabled}
gatein.sso.login.module.enabled=${gatein.sso.enabled}
gatein.sso.login.module.class=org.gatein.sso.agent.login.SSOLoginModule
gatein.sso.server.url=http://localhost:8888/cas
gatein.sso.portal.url=http://localhost:8080
gatein.sso.filter.logout.class=org.gatein.sso.agent.filter.CASLogoutFilter
gatein.sso.filter.logout.url=${gatein.sso.server.url}/logout
gatein.sso.filter.login.sso.url=${gatein.sso.server.url}/login?service=${gatein.sso.portal.url}/@@
```

In previous versions of GateIn Portal, there were much more changes needed in various configuration file but it's not the case anymore. Now all JARS are available in AS7 module `org.gatein.sso` so you don't need to manually add any JAR files. If you are interested in technical details about single properties and configuration, you can see next section with configuration properties details.

Configuration properties details

- `gatein.sso.enabled` - This option will generally enable SSO integration and informs GateIn Portal about that. So for example now when you click to *Sign in* link, you won't see login dialog, but will be redirected to */portal/sso* URL
- `gatein.sso.callback.enabled` - This will enable REST callback authentication handler, which is needed if you want CAS server to use SSO Authentication plugin for CAS own authentication. See Login workflow point 3 for details. By default, callback handler is enabled if option `gatein.sso.enabled` is true. You can switch it to false if you don't want to use Authentication Plugin on CAS server side.
- `gatein.sso.login.module.enabled` - There is special login module configured for `gatein-domain` in `GATEIN_HOME/standalone/configuration/standalone.xml` called `SSODelegateLoginModule` . If SSO is disabled, this SSODelegateLoginModule is simply ignored during authentication process. But if SSO is enabled by this property, it delegates the work to another login module configured via next option `gatein.sso.login.module.class` . SSODelegateLoginModule will also resend all it's options to it's delegate. In case of CAS server, we will use `org.gatein.sso.agent.login.SSOLoginModule` as delegate. The point of this architecture is, that people don't need to manually change any login module configurations in standalone.xml file.
- `gatein.sso.login.module.class` - See previous paragraph

The main GateIn Portal configuration file for SSO integration is `GATEIN_HOME/gatein/gatein.ear/portal.war/WEB-INF/conf/sso/security-sso-configura` . All needed SSO components like agents and SSO interceptors (former servlet filters) are configured in this file. The idea is that you never need to manually edit this file as most of the options are configurable via configuration.properties . But in case that something is really not suitable for your use-case or you need to add another custom interceptor or something else, you can manually edit it here. All the additional configuration properties are used especially for substitute values in this security-sso-configuration.xml file.

- `gatein.sso.server.url` - Here you need to configure where your CAS server is deployed
- `gatein.sso.portal.url` - Here is URL for access to your GateIn Portal server (actually server you are just configuring)
- `gatein.sso.filter.logout.class` - Class of logout filter, which needs to be set to `org.gatein.sso.agent.filter.CASLogoutFilter` . This filter is able to redirect to CAS server and performs logout on CAS side.
- `gatein.sso.filter.logout.url` - CAS server logout URL, which will be used for redirection by logout filter

  If you want to disable logout on CAS side, you can simply disable this logout interceptor by adding option `gatein.sso.filter.logout.enabled` with value `false`. This will cause that click to *Sign out* on portal side will logout user from GateIn Portal but not from CAS server. In this case both options `gatein.sso.filter.logout.class` and `gatein.sso.filter.logout.url` will be ignored.

- `gatein.sso.filter.login.sso.url` - CAS server login URL, which will be used by LoginRedirectFilter for redirection to CAS server login page.

  The string @@**portal.container.name**@@ will be dynamically replaced by correct name of portal container, where it will be executed. GateIn Portal SSO component will do it. So in configuration, you should really use string @@`portal.container.name`@@ instead of some hardcoded portal container name (like `portal` or `sample-portal`)

Once these changes have been made, all links to the user authentication pages will redirect to the CAS centralized authentication form. And on CAS you will be able to authenticate with portal credentials (like john/gtn) thanks to Authentication plugin.

Setup with portal on Tomcat

If you have GateIn Portal on Tomcat7 and you want to use CAS for authentication, you can change file `GATEIN_HOME/gatein/conf/configuration.properties` and configure them exactly same like for GateIn Portal on JBoss AS7. There is only one additional thing you need to do:

In file `GATEIN_HOME/conf/server.xml` you need to add special `ServletAccessValve` under *Host* element:

```
<Host name="localhost" appBase="webapps"
  unpackWARs="true" autoDeploy="true">

  <Valve className="org.gatein.sso.agent.tomcat.ServletAccessValve" />

  <!-- SingleSignOn valve, share authentication between web applications
  ...
```

6.9.3 JOSSO

This Single-Sign-On plugin enables the seamless integration between GateIn Portal and the JOSSO Single-Sign-On Framework. Details about JOSSO can be found here.

Setting up this integration consists of two steps: installing/configuring a JOSSO server, and setting up the portal to use the JOSSO server.

We are supporting JOSSO versions 1.8 and 2.2. Integration steps are slightly different among these 2 versions. So we will separately describe integration for JOSSO 1.8 and for JOSSO 2.2.

Login workflow

Login workflow for JOSSO si quite similar like for CAS, so you can look Central Authentication Service (CAS)#Login workflow for more details. Basically after click to *Sign in* is user redirected to JOSSO login screen where he provides his credentials and then he is redirected back to GateIn Portal. InitiateLoginFilter will then delegate validation of JOSSO ticket, which is in request parameter *josso_assertion_id* to **JOSSOAgent** component, which performs validation of ticket with JOSSO server via back channel. Actually there is communication via web services between JOSSO agent and JOSSO server. After successful validation is user identity successfully established and user is logged in GateIn Portal.

For logout part, there is also **JOSSOLogoutFilter**, which performs logout on both GateIn Portal and JOSSO server (similarly like for CAS).

For JOSSO authentication part, we again support authentication plugin, which is able to send REST request to GateIn Portal and then obtain response from GateIn Portal and authenticate user on JOSSO side based on it. However this authentication plugin is supported only for JOSSO 1.8 (not supported for JOSSO 2.2 at this moment)

Integration with JOSSO 1.8

In this sample setup, we will assume again that GateIn Portal will be running on JBoss AS7 and on *localhost:8080* and JOSSO server will be running on Tomcat on *localhost:8888* .

There are also differences between various JOSSO minor versions (especially betweeen JOSSO versions 1.8.1 and 1.8.2) so instructions will be slightly different between various versions. This will be pointed in text in more details.

JOSSO server

This section describes how to set up the JOSSO server to authenticate against the GateIn Portal with usage of REST authentication plugin.

In this example, the JOSSO server will be installed on Tomcat.

Obtaining JOSSO

JOSSO can be downloaded from http://sourceforge.net/projects/josso/files/. Use the package that embeds Apache Tomcat.

Once downloaded, extract the package into what will be called `JOSSO_HOME` in this example.

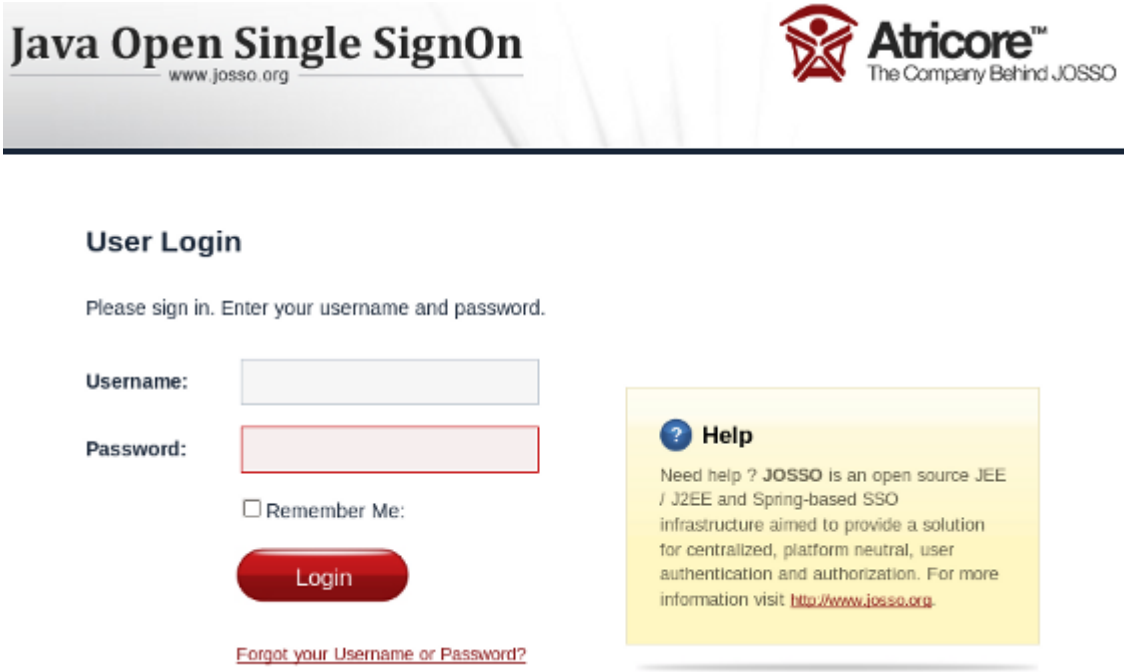
Modifying the JOSSO server

1. For using SSO authentication plugin with JOSSO (not-mandatory but recommended. See Login workflow for details)
    - If you have JOSSO 1.8.1, then copy the files from `GATEIN_SSO_HOME/josso/josso-181/plugin` into the Tomcat directory (`JOSSO_HOME`). (GATEIN_SSO_HOME points to directory with GateIn Portal as mentioned in Single-Sign-On (SSO))
    - If you have JOSSO 1.8.2 or newer, then copy the files from `GATEIN_SSO_HOME/josso/josso-182/plugin` into the Tomcat directory (`JOSSO_HOME`). This action should replace or add some JAR files to the `JOSSO_HOME/webapps/josso/WEB-INF/lib` directory and also the files: `JOSSO_HOME/lib/josso-gateway-config.xml` `JOSSO_HOME/lib/josso-gateway-gatein-stores.xml` `JOSSO_HOME/webapps/josso/WEB-INF/classes/gatein.properties` - This file may need to be reconfigured according to your GateIn Portal environment (you need to use host and port where your GateIn Portal is running as this will be used by Authentication plugin to send REST request over HTTP)

2. Edit `TOMCAT_HOME/conf/server.xml` and replace the 8080 port to 8888 to change the default Tomcat port and avoid a conflict with the default GateIn Portal port (for testing purposes).

> **Port Conflicts**
>
> If GateIn Portal is running on the same machine as Tomcat, other ports need to be changed in addition to 8080 to avoid port conflicts. They can be changed to any free port. For example, you can change the admin port from 8005 to 8805, and AJP port from 8009 to 8809.

3. Start Tomcat now that allows access to http://localhost:8888/josso/signon/login.do. However, if you are using SSO Authentication plugin, the login will not be available at this stage as your GateIn Portal is not set yet



Setup the JOSSO client

1. You need to configure couple of configuration properties in file `GATEIN_HOME/standalone/configuration/gatein/configuration.properties` . Find SSO section in this file and change/add properties so that it looks like this:

```
#SSO
gatein.sso.enabled=true
gatein.sso.callback.enabled=${gatein.sso.enabled}
gatein.sso.login.module.enabled=${gatein.sso.enabled}
gatein.sso.login.module.class=org.gatein.sso.agent.login.SSOLoginModule
gatein.sso.josso.agent.config.file=sso/josso/1.8/josso-agent-config.xml
gatein.sso.josso.properties.file=file:${jboss.home.dir}/standalone/configuration/gatein/conf
```

   Most of the properties were already described in Central Authentication Service (CAS)#Configuration properties details . For JOSSO some of the properties are different, especially URL for redirection to login/logout to JOSSO server and Logout filter class is now `org.gatein.sso.agent.filter.JOSSOLogoutFilter` . Important property is `gatein.sso.josso.host`, which points to location of JOSSO server. Also `gatein.sso.portal.url` needs to be changed if you expect GateIn Portal to be accessed on different URL than *localhost:8080*.
   Property `gatein.sso.josso.agent.config.file` points to location of Agent configuration file, which is relative to classpath. So the agent file location is actually in `GATEIN_HOME/gatein/gatein.ear/portal.war/WEB-INF/classes/sso/josso/1.8/joss` but normally you won't need to change anything here.
2. Update SSO module. JOSSO has some specific dependencies, which differ between various JOSSO versions. So for JOSSO setup, we are replacing original `org.gatein.sso` AS7 module with specific module for proper JOSSO version, which can be found in SSO packaging. You need to:
   - Delete directory `GATEIN_HOME/modules/org/gatein/sso`
   - If you have JOSSO 1.8.1 or older, then copy directory `GATEIN_SSO_HOME/josso/gatein-josso-181/modules/org/gatein/sso into GATEIN_HOME/modules/org/gatein/`
   - If you have JOSSO 1.8.2 or newer, then copy directory `GATEIN_SSO_HOME/josso/gatein-josso-182/modules/org/gatein/sso into GATEIN_HOME/modules/org/gatein/`
     So actually original `org.gatein.sso` module is replaced with the new one specific for JOSSO.

From now on, all links redirecting to the user authentication pages will redirect to the JOSSO centralized authentication form. If you set Authentication plugin for JOSSO, you can login with GateIn Portal credentials (like john/gtn) on JOSSO side.

Integration with JOSSO 2.2

JOSSO 2.2 has different approach than JOSSO 1.8. The idea is that even non-technical people are able to create their own SSO environment by model it in flash web application called **atricore-console**. But now it's much harder to plug our own Authentication plugin as it's not easily possible to configure existing JOSSO 2.2 environment via Spring XML files similarly like it was with JOSSO 1.8.

So actually we don't support plugging our own AuthenticationPlugin into JOSSO 2.2 similarly like for other SSO solutions. Important is that we still support agent part, so nothing is changed from portal perspective.

JOSSO 2.2 server setup

You can downloaded JOSSO 2.2.0 from JOSSO site and follow the instructions from JOSSO 2 quickstart in http://www.josso.org/confluence/display/JOSSO1/JOSSO2+Quick+start . I expect that after unzip and running the JOSSO, you can access atricore console on http://server.local.network:8081/atricore-console (*server.local.network* is my virtual host defined in `/etc/hosts`)

Then follow other instructions from quickstart in flash console together with instructions here:

1. Login as *admin/admin*
2. Create new empty *Identity appliance*:
   Name: *MYFIRSTIA*
   Realm name: *com.mycompany.myrealm*
   Appliance location: *http://server.local.network:8081*
3. Create new Identity provider named *AcmeIDP* (let all options default)

Identity Provider configuration

4. Create Identity vault *IDPUsers* and connect it with *AcmeIDP* via *Identity lookup* connection
5. Create Service provider called *SP1* but let the hosts to be on *server.local.network:8081*

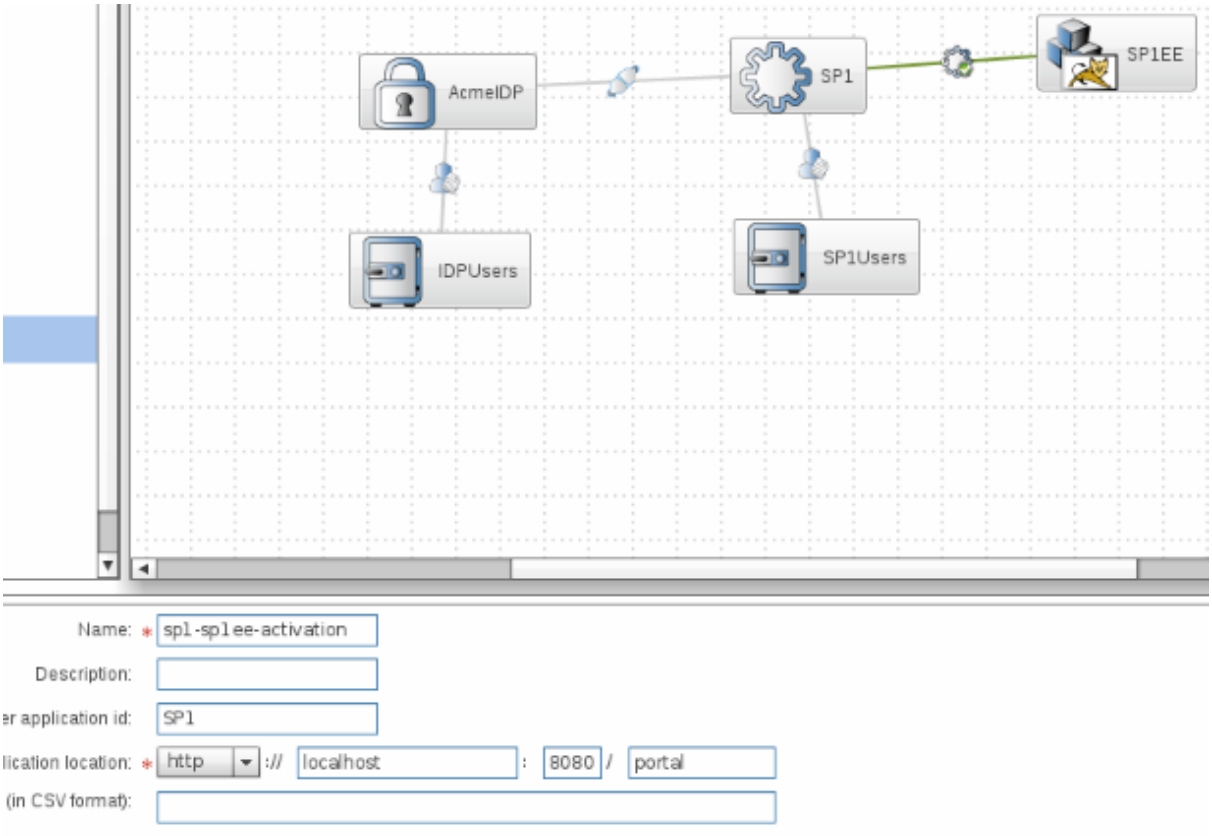
Service Provider configuration

6. Create Identity vault *SP1Users* and wire it with SP1 via *Identity lookup* connection

7. Create empty temporary directory `/tmp/tomcat7` on your filesystem and then in atricore console create new Execution environment of type *Tomcat* with params:
   Name: *SP1EE*
   Version: *7.0.x*
   Target host: *Local*
   Install home: */tmp/tomcat7* (Directory `/tmp/tomcat7` must exists, but it could be empty directory without any tomcat presented as we don't overwrite existing setup or install demo apps)

SP Execution Environment configuration

8. Wire *SP1* and *SP1EE* via connection of type *Activation*. All parameters of this new connection can have default values except for parameter *Partner application location*, which needs to be changed to http://localhost:8080/portal .
9. Wire *SP1* and *AcmeIDP* via connection of type *Federated connection*
10. Click *Save* and save this model



Whole picture overview with SP connection detail

11. Go to tab *Identity appliance lifecycle management* and go through lifecycle of Identity appliance (Saved => Staged => Deployed => Started) as suggested in quickstart

12. Go to tab *Account & Entitlement management* and create some users. We need to add users this way, because REST callbacks to GateIn Portal are not supported at this moment. So let's create users:
    "john" with password "password"
    "root" with password "password"
    "demo" with password "password"

JOSSO client setup

1. Assuming again that you have GateIn Portal on JBoss AS7, you need to change properties in `GATEIN_HOME/standalone/configuration/gatein/configuration.properties` in SSO sections. Let's ssume that JOSSO 2.2 is running on `server.local.network:8081`, name of JOSSO Identity appliance is `MYFIRSTIA`, name of created service provider is `SP1` and name of execution environment is `SP1EE`. Everything was described in JOSSO 2.2 server setup . Note that `gatein.sso.filter.logout.url` is empty now as logout URL will be obtained from JOSSO agent configuration in file `GATEIN_HOME/gatein/gatein.ear/portal.war/WEB-INF/classes/sso/josso/2.2/josso`

```
# SSO
gatein.sso.enabled=true
gatein.sso.callback.enabled=${gatein.sso.enabled}
gatein.sso.login.module.enabled=${gatein.sso.enabled}
gatein.sso.login.module.class=org.gatein.sso.agent.login.SSOLoginModule
gatein.sso.filter.initiatelogin.enabled=false
gatein.sso.filter.initiatelogin.josso2.enabled=true
gatein.sso.josso.agent.config.file=sso/josso/2.2/josso-agent-config.xml
gatein.sso.josso.properties.file=file:${jboss.home.dir}/standalone/configuration/gatein/conf
```

2. Update SSO module in AS7 similarly like for JOSSO 1.8.2. So you need to:
    - Delete directory `GATEIN_HOME/modules/org/gatein/sso`
    - Copy directory `GATEIN_SSO_HOME/josso/gatein-josso-182/modules/org/gatein/sso into GATEIN_HOME/modules/org/gatein/`
3. Start GateIn Portal, then access http://localhost:8080/portal/ and click *Sign in*. Now you will be redirected to JOSSO, but you will need to login with username/password created via JOSSO console (aka. john/password) as REST callbacks are not supported. After successful login to JOSSO, you will be redirected to GateIn Portal as john.

Setup with portal on Tomcat

If you have GateIn Portal on Tomcat 7 and you want to configure it for SSO against JOSSO, you additionally need to add `ServletAccessValve` into `server.xml` similarly like for CAS setup. See Central Authentication Service (CAS)#Setup with portal on Tomcat for more details.

Also you need to copy the jars for proper JOSSO version from `GATEIN_SSO_HOME/josso/gatein-josso-18X/modules/org/gatein/sso/main into GATEIN_HOME/lib/` . Instead of `gatein-josso-18X` use:

- `gatein-josso-181` if you are on JOSSO 1.8.1 or older
- `gatein-josso-182` if you are on JOSSO 1.8.2 or newer or on JOSSO 2.2

6.9.4 OpenAM

Setting up this integration involves two steps. The first step is to install or configure an OpenAM server, and the second is to set up the portal to use the OpenAM server. In this sample setup, we will assume again that GateIn Portal will be running on JBoss AS7 and on *localhost:8080* and JOSSO server will be running on Tomcat on *localhost:8888* .

Login and logout workflow

Login workflow for OpenAM si quite similar like for CAS, so you can look to Central Authentication Service (CAS)#Login workflow for more details. Basically after click to *Sign in* on GateIn Portal page is user redirected to OpenAM login screen where he provides his credentials. Then he is redirected back to GateIn Portal. *InitiateLoginFilter* will then delegate validation of OpenAM ticket, which is in cookie *iPlanetDirectoryPro* to *OpenSSOAgent* component, which performs validation of ticket with OpenAM server via back channel. Actually agent is using OpenAM REST API to verify ticket. After successful validation is user identity successfully established and user is logged in GateIn Portal.

For logout part, there is *OpenSSOLogoutFilter*, which performs logout on both GateIn Portal and OpenAM server (similarly like for CAS).

For OpenAM authentication part, we again support authentication plugin, which is able to send REST request to GateIn Portal and then obtain response from portal and authenticate user on OpenAM side based on it.

OpenAM server setup

This section details setting up of OpenAM server to authenticate against the GateIn Portal with usage of REST callback. So similarly like for CAS and JOSSO, we support Authentication plugin, which can be set on OpenAM to authenticate against portal. But note again that it's not mandatory to use this callback, you can use whatever you want for authentication on OpenAM side.

In this example, the OpenAM server will be installed on Tomcat.

Obtaining OpenAM

OpenAM must be purchased from Forgerock. Assumption is that you download ZIP distribution.

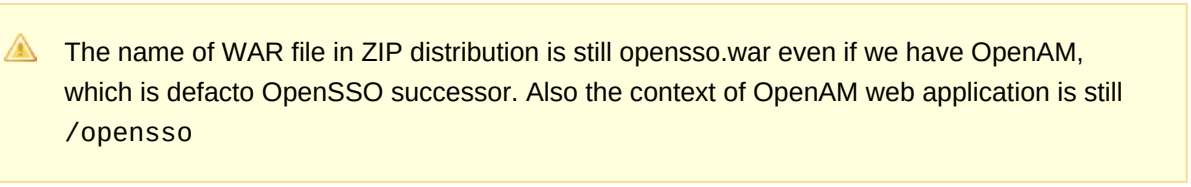
For GateIn Portal integration, we support and test OpenAM of versions 9.5 or 10.0. However other versions can still work (also OpenAM predecessor OpenSSO can still work even if we stop to officially support it)

Once downloaded, extract the package into a suitable location. This location will be referred to as `OPENAM_HOME` in this example.

Modifying the OpenAM server

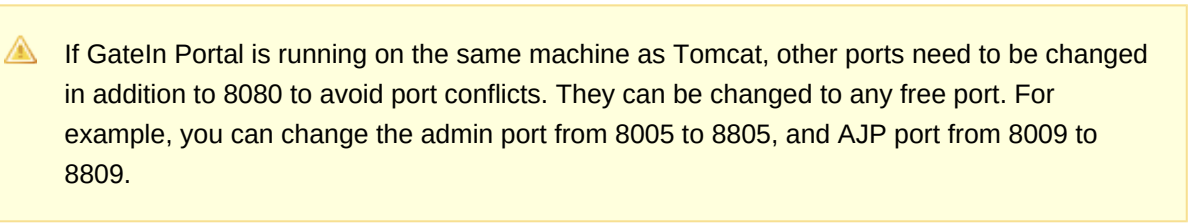
The first step is to deploy OpenAM to Tomcat 7 server and properly configure it. Recommended (but not mandatory) step is to add the GateIn Portal SSO Authentication Plugin. The plugin makes secure authentication callbacks to a RESTful service installed on the remote GateIn Portal server to authenticate a user. In these instructions we will setup it.

1. Obtain a copy of Tomcat 7 and extract it into a suitable location (this location will be referred to as `TOMCAT_HOME` in this example).
2. Deploy OpenAM to tomcat by copying `OPENAM_HOME/opensso/deployable-war/opensso.war` to `TOMCAT_HOME/webapps` .



The name of WAR file in ZIP distribution is still opensso.war even if we have OpenAM, which is defacto OpenSSO successor. Also the context of OpenAM web application is still `/opensso`

3. Change the default port to avoid a conflict with the default GateIn Portal (for testing purposes) by editing `TOMCAT_HOME/conf/server.xml` and replacing the 8080 port with 8888.



If GateIn Portal is running on the same machine as Tomcat, other ports need to be changed in addition to 8080 to avoid port conflicts. They can be changed to any free port. For example, you can change the admin port from 8005 to 8805, and AJP port from 8009 to 8809.

Authentication plugin setup

1. Copy content of `GATEIN_SSO_HOME/opensso/plugin` to `TOMCAT_HOME/webapps/opensso` (GATEIN_SSO_HOME points to location with extracted GateIn Portal SSO packaging as described in Single-Sign-On (SSO))
   This will add:
   - `TOMCAT_HOME/webapps/opensso/config/auth/default/AuthenticationPlugin.xml` file, which looks like this:

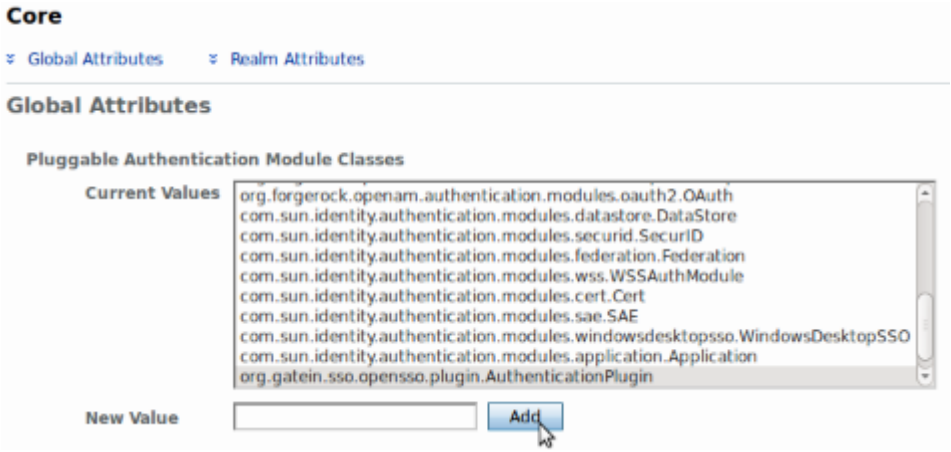
```
<?xml version='1.0' encoding="UTF-8"?>
<!DOCTYPE ModuleProperties PUBLIC "=//iPlanet//Authentication Module Properties XML
Interface 1.0 DTD//EN"
          "jar://com/sun/identity/authentication/Auth_Module_Properties.dtd">

<ModuleProperties moduleName="AuthenticationPlugin" version="1.0" >
  <Callbacks length="2" order="1" timeout="60"
             header="GateIn OpenAM Login" >
    <NameCallback>
      <Prompt>
        Username
      </Prompt>
    </NameCallback>
    <PasswordCallback echoPassword="false" >
      <Prompt>
        Password
      </Prompt>
    </PasswordCallback>
  </Callbacks>
</ModuleProperties>
```

   - Some JARS like `sso-opensso-plugin-<VERSION>.jar` and `commons-httpclient-<VERSION>.jar` and few others to directory `TOMCAT_HOME/webapps/opensso/WEB-INF/lib` .
   - File `TOMCAT_HOME/webapps/opensso/WEB-INF/classes/gatein.properties`. You may need to configure the host, port, protocol or other properties in this file according to your GateIn Portal location. This will be used by Authentication plugin for REST connection to portal over HTTP protocol.
2. Start Tomcat to be able to access http://localhost:8888/opensso.

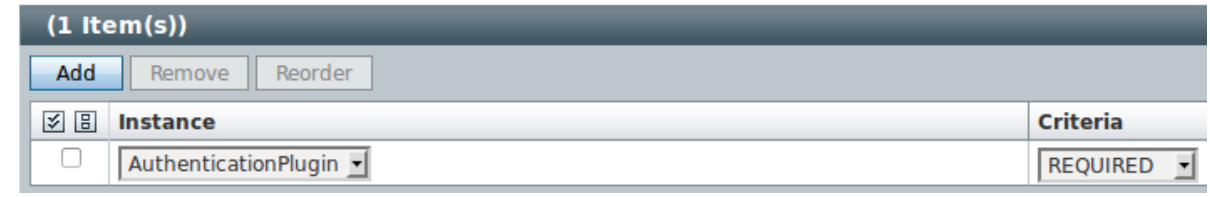
Configure realm in OpenAM UI

1. Direct your browser to http://localhost:8888/opensso.
2. Create the default configuration.

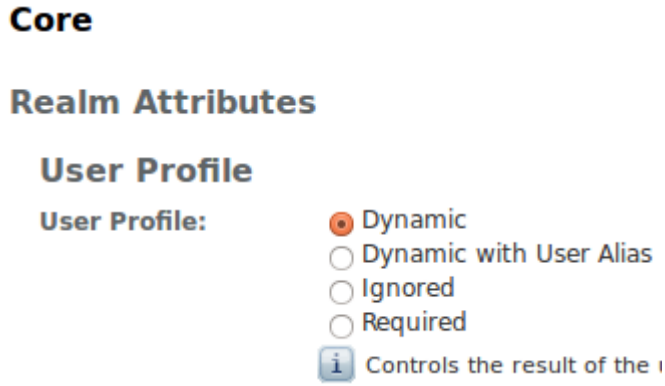
3. Login as `amadmin` and then go to `Configuration -> Authentication`. Next, select the *Core* link, and add new value and fill in the `org.gatein.sso.opensso.plugin.AuthenticationPlugin` class name and finally click "Save". This step is important for setup of GateIn Portal SSO `AuthenticationPlugin` to be available among other OpenAM authentication modules.



4. Go to the *Access control* tab and create a new realm called `gatein`.
5. Go to the *gatein* realm, and click on the *Authentication* tab. At the bottom in the *Authentication chaining* section, click on *ldapService*. Here change the selection from *Datastore*, which is the default module in the authentication chain, to *AuthenticationPlugin*. This enables authentication of the *gatein* realm by using GateIn Portal REST service instead of the OpenAM LDAP server.



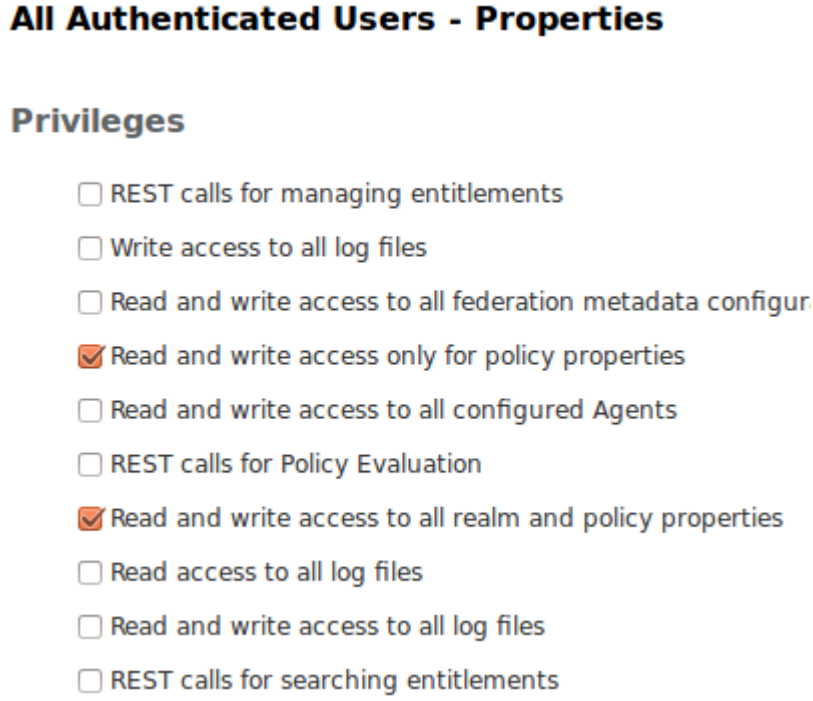
6. When you are on *Authentication* tab of *gatein* realm, click to button *All Core Settings* . Then change UserProfile from *Required* to *Dynamic*. This step is needed because GateIn Portal users are not in OpenAM Datastore (LDAP server), so their profiles can not be obtained if "Required" is active. By using "Dynamic", all new users are automatically created in OpenAM datastore after successful authentication.



7. Increase the user privileges to allow REST access. Go to `Access control -> Top level realm -> Privileges -> All authenticated users`, and check the checkboxes:

- `Read and write access only for policy properties`
- `Read and write access to all realm and policy properties`

Repeat previous step with increasing privileges for the *gatein* realm as well.



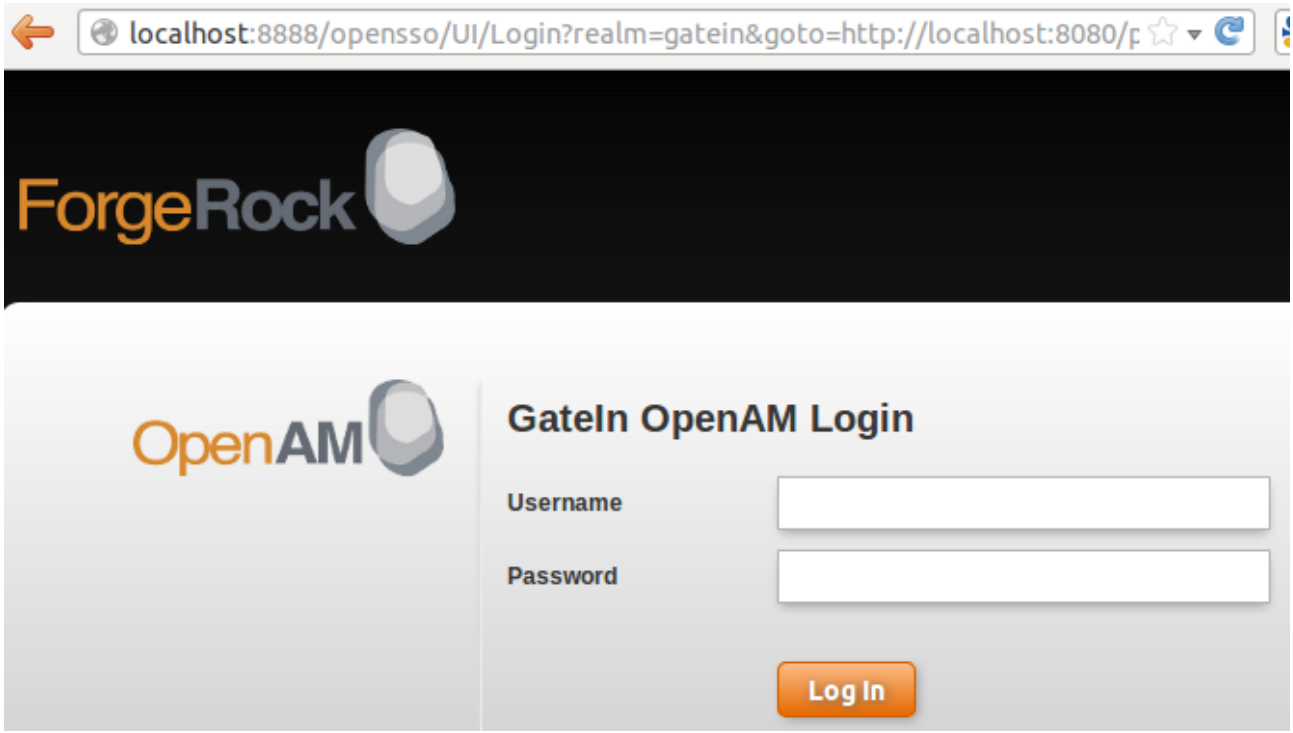
Setup the OpenAM client

We assume that GateIn Portal, which will act as OpenAM client, will be running on JBoss AS7. So you need to configure couple of configuration properties in file `GATEIN_HOME/standalone/configuration/gatein/configuration.properties` . Find SSO section in this file and change/add properties to look like this:

```
# SSO
gatein.sso.enabled=true
gatein.sso.callback.enabled=${gatein.sso.enabled}
gatein.sso.login.module.enabled=${gatein.sso.enabled}
gatein.sso.login.module.class=org.gatein.sso.agent.login.SSOLoginModule
gatein.sso.server.url=http://localhost:8888/opensso
gatein.sso.openam.realm=gatein
gatein.sso.portal.url=http://localhost:8080
gatein.sso.filter.logout.class=org.gatein.sso.agent.filter.OpenSSOLogoutFilter
gatein.sso.filter.logout.url=${gatein.sso.server.url}/UI/Logout
gatein.sso.filter.login.sso.url=${gatein.sso.server.url}/UI/Login?realm=${gatein.sso.openam.realm}
```

Most of the properties were already described in Central Authentication Service (CAS)#Configuration properties details . For OpenAM some of the properties are different, especially URL for redirection to login/logout to OpenAM server and Logout filter class is now `org.gatein.sso.agent.filter.OpenSSOLogoutFilter`. Very important property is `gatein.sso.server.url`, which points to location of OpenAM server. Also `gatein.sso.portal.url` will need to be changed if you expect GateIn Portal to be accessed on different URL than *http://localhost:8080* . *Property* `gatein.sso.openam.realm` points to realm, which needs to be available in OpenAM server. We assume that name of the realm is *gatein* as configured in previous section Configure realm in OpenAM UI .

From now on, all links redirecting to the user authentication pages will redirect to the OpenAM centralized authentication form. On OpenAM side, you can login with GateIn Portal credentials (for example john/gtn) to *gatein* realm if you properly configured Authentication plugin.



Setup with portal on Tomcat

If you have GateIn Portal on Tomcat 7 and you want to configure it for SSO against OpenAM, you additionally need to add `ServletAccessValve` into `server.xml` similarly like for CAS setup. See Central Authentication Service (CAS)#Setup with portal on Tomcat for more details.

Cross-domain authentication with OpenAM

The authentication scenario described in previous parts assumes that GateIn Portal and OpenAM are deployed on the same server or in same DNS domain (like OpenAM on *opensso.shareddomain.com* and GateIn Portal on *portal.shareddomain.com*).

After successful authentication in OpenAM console, OpenAM will add special cookie *iPlanetDirectoryPro* for DNS domain *shareddomain.com* and then it redirects to portal agent. Portal OpenSSOAgent can read SSO token from this cookie because cookie is in same DNS domain, so it can perform validation of token. In other words, exchange of secret token between OpenAM and GateIn Portal is done through this shared cookie.

This approach cannot work in situations, when GateIn Portal server and OpenAM server are in different domains and cannot share cookie. For this scenario, OpenAM provides special servlet *CDCServlet*. Authenticated user can send request to this servlet and servlet will send him encoded SAML message with SSO token and other informations. Portal agent is then able to parse and validate this message, obtain SSO token and establish *iPlanetDirectoryPro* cookie for server where portal is deployed. Once OpenAM agent on portal side has token, it can perform other validations of this token and successfully finish authentication of user.

You can follow this link for more technical information about CDCServlet.

Cross-domain authentication configuration

1. Assume that your OpenAM server is deployed on host *opensso.mydomain.com* and GateIn Portal on *portal.yourdomain.com*. If you are on single machine, you can simply simulate this scenario by using virtual hosts. On linux you can edit the `/etc/hosts` file and add records similar to those (change according to IP addresses in your environment):

   ```
   opensso.mydomain.com 192.168.2.7
   portal.yourdomain.com 10.11.12.13
   ```

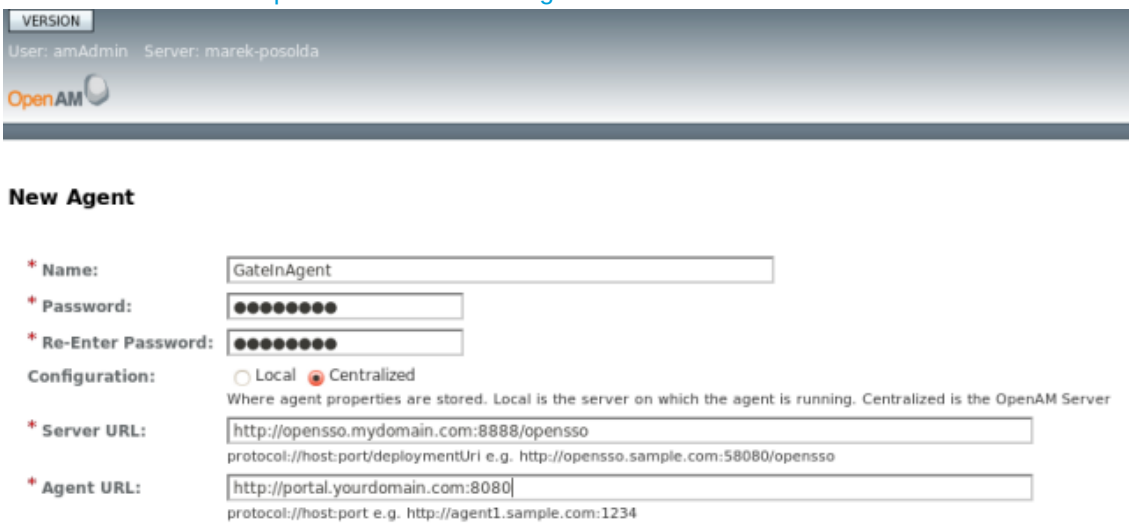
2. SSO configuration in file `configuration.properties` on GateIn Portal side needs to look like this:

   ```
   # SSO
   gatein.sso.enabled=true
   gatein.sso.callback.enabled=${gatein.sso.enabled}
   gatein.sso.login.module.enabled=${gatein.sso.enabled}
   gatein.sso.login.module.class=org.gatein.sso.agent.login.SSOLoginModule
   gatein.sso.server.url=http://opensso.mydomain.com:8888/opensso
   gatein.sso.openam.realm=gatein
   gatein.sso.portal.url=http://portal.yourdomain.com:8080
   gatein.sso.filter.logout.class=org.gatein.sso.agent.filter.OpenSSOLogoutFilter
   gatein.sso.filter.logout.url=${gatein.sso.server.url}/UI/Logout
   gatein.sso.filter.login.enabled=false
   gatein.sso.filter.login.openamcdc.enabled=true
   gatein.sso.filter.login.sso.url=${gatein.sso.server.url}/cdcservlet
   ```

   The important part are last 3 properties (We needs to redirect to OpenAM cdcservlet, so we are using modified version of `LoginRedirectFilter`. That's why `gatein.sso.filter.login.openamcdc.enabled` is changed to true and `gatein.sso.filter.login.enabled` is false. Also `gatein.sso.filter.login.sso.url` now points to URL of OpenAM cdcservlet)

3. On OpenAM side, it is mandatory to create agent for GateIn Portal server. This agent is required by *CDCServlet* to work properly. You can create agent in OpenAM UI by performing these steps:
    - Go to http://opensso.mydomain.com:8888/opensso and login as *amadmin*.
    - Go to `Access Control -> Realm "gatein" -> Agents -> Web`.
    - Create new web agent through the wizard. You can use these properties:
    - Name: GateInAgent
    - Password: Whatever you want.
    - Configuration: Centralized
    - Server URL: http://opensso.mydomain.com:8888/opensso
    - Agent URL: http://portal.yourdomain.com:8080

      If you have more portal servers on different hosts, you may want to create agent for each of them. Please look at OpenAM administration guide for more details.



6.9.5 SPNEGO

SPNEGO (Simple and Protected GSSAPI Negotiation Mechanism) is used to authenticate transparently through the web browser after the user has been authenticated when logging-in his session.

A typical use case is the following:

1. User logs into his desktop (Such as a Windows machine).
2. The desktop login is governed by Active Directory domain.
3. User then uses his browser (IE/Firefox) to access a web application (that uses JBoss Negotiation) hosted on GateIn Portal.
4. The Browser transfers the desktop sign on information to the web application.
5. GateIn Portal uses background GSS messages with the Active Directory (or any Kerberos Server) to validate the Kerberos ticket from user.
6. The User has seamless SSO into the web application.

GateIn Portal provides SPNEGO integration via GateIn Portal SSO library, which leverages JBoss Negotiation library for this purpose.

SPNEGO Server Configuration

In this section, there are some necessary steps for setting up the Kerberos server on Linux. This server will then be used for SPNEGO authentication against GateIn Portal.

If you do not have Linux but you are using the Windows and Active Directory domain, the information is not important for you and you may jump to the Portal configuration . Please note that Kerberos setup is also dependent on your Linux distribution, so steps can be slightly different in your environment.

1. Correct the setup of network on the machine. For example, if you are using the "server.local.network" domain as your machine where Kerberos and GateIn Portal are located, add the line containing the machine's IP address to the */etc/hosts* file.

```
192.168.1.88  server.local.network
```

It is not recommended to use loopback addresses.

2. Install Kerberos with these packages: `krb5-admin-server`, `krb5-kdc`, `krb5-config`, `krb5-user`, `krb5-clients`, and `krb5-rsh-server`.
3. Edit the Kerberos configuration file at `/etc/krb5.conf`, including:
    - Uncomment on these lines:

```
default_tgs_enctypes = rc4-hmac
default_tkt_enctypes = rc4-hmac
permitted_enctypes = rc4-hmac
```

- Add *local.network* as a default realm and it is also added to the list of realms and remove the remains of realms. The content looks like:

```
[libdefaults]
    default_realm = LOCAL.NETWORK

# The following krb5.conf variables are only for MIT Kerberos.
    krb4_config = /etc/krb.conf
    krb4_realms = /etc/krb.realms
    kdc_timesync = 1
    ccache_type = 4
    forwardable = true
    proxiable = true

# The following encryption type specification will be used by MIT Kerberos
# if uncommented.  In general, the defaults in the MIT Kerberos code are
# correct and overriding these specifications only serves to disable new
# encryption types as they are added, creating interoperability problems.
#
# Thie only time when you might need to uncomment these lines and change
# the enctypes is if you have local software that will break on ticket
# caches containing ticket encryption types it doesn't know about (such as
# old versions of Sun Java).

    default_tgs_enctypes = rc4-hmac
    default_tkt_enctypes = rc4-hmac
    permitted_enctypes = rc4-hmac

# The following libdefaults parameters are only for Heimdal Kerberos.
    v4_instance_resolve = false
    v4_name_convert = {
        host = {
            rcmd = host
            ftp = ftp
        }
        plain = {
            something = something-else
        }
    }
    fcc-mit-ticketflags = true

[realms]
    LOCAL.NETWORK = {
        kdc = server.local.network
        admin_server = server.local.network
    }

[domain_realm]
    .local.network = LOCAL.NETWORK
    local.network = LOCAL.NETWORK

[login]
    krb4_convert = true
    krb4_get_tickets = false
```

4. Edit the KDC configuraton file at `/etc/krb5kdc/kdc.conf` that looks like.

```
[kdcdefaults]
    kdc_ports = 750,88

[realms]
    LOCAL.NETWORK = {
        database_name = /var/lib/krb5kdc/principal
        admin_keytab = FILE:/etc/krb5.keytab
        acl_file = /etc/krb5kdc/kadm5.acl
        key_stash_file = /etc/krb5kdc/stash
        kdc_ports = 750,88
        max_life = 10h 0m 0s
        max_renewable_life = 7d 0h 0m 0s
        master_key_type = rc4-hmac
        supported_enctypes = rc4-hmac:normal
        default_principal_flags = +preauth
    }

[logging]
        kdc = FILE:/tmp/kdc.log
        admin_server = FILE:/tmp/kadmin.log
```

5. Next, create a KDC database using the command:

```
sudo krb5_newrealm
```

6. Start the KDC and Kerberos admin servers using these commands:

```
sudo /etc/init.d/krb5-kdc restart
sudo /etc/init.d/krb5-admin-server restart
```

7. Add Principals and create Keys.
    - Start an interactive 'kadmin' session and create the necessary Principals using the command:

```
sudo kadmin.local
```

    - Add the GateIn Portal machine and keytab file that need to be authenticated using the commands:

```
addprinc -randkey HTTP/server.local.network@LOCAL.NETWORK
ktadd HTTP/server.local.network@LOCAL.NETWORK
```

    - Add the default GateIn Portal user accounts and enter the password for each created user that will be authenticated using the commands:

```
addprinc john
addprinc demo
addprinc root
```

8. Test your changed setup by command:

```
kinit -A demo
```

    - If the setup works well, you are required to enter the password created for this user in previous step. Without `-A`, the kerberos ticket validation involved reverse DNS lookups, which can get very cumbersome to debug if your network's DNS setup is not great. This is a production level security feature, which is not necessary in this development setup. In production environment, it will be better to avoid `-A` option.
    - After successful login to Kerberos, you can see your Kerberos ticket when using the `klist` command.
    - If you want to logout and destroy your ticket, use the `kdestroy` command.

Browser configuration

After performing all configurations above, you need to enable the Negotiate authentication of Firefox in client machines so that clients could be authenticated by GateIn Portal as follows:

1. Start Firefox, then enter the `about:config` command into the address field.
2. Enter *network.negotiate-auth* and set the value as below:

```
network.negotiate-auth.allow-proxies = true
network.negotiate-auth.delegation-uris = .local.network
network.negotiate-auth.gsslib (no-value)
network.negotiate-auth.trusted-uris = .local.network
network.negotiate-auth.using-native-gsslib = true
```

Consult the documentation of your OS or web browser if using different browser than Firefox.

Portal configuration

SPNEGO setup works only with GateIn Portal deployed on JBoss AS 7 because it uses JBoss Negotiation library specific to JBoss. This library leverages custom Tomcat authenticator Valve and custom JAAS login module to provide SPNEGO and Kerberos integration.

1. You will need to change SSO section in the file `GATEIN_HOME/standalone/configuration/gatein/configuration.properties` to be like this:

```
# SSO
gatein.sso.enabled=true
gatein.sso.callback.enabled=false
gatein.sso.skip.jsp.redirection=false
gatein.sso.login.module.enabled=false
gatein.sso.filter.login.sso.url=/@@portal.container.name@@/dologin
gatein.sso.filter.logout.enabled=false
gatein.sso.filter.initiatelogin.enabled=false
gatein.sso.valve.enabled=true
gatein.sso.valve.class=org.gatein.sso.spnego.GateInNegotiationAuthenticator
```

Description of individual properties:

- `gatein.sso.enabled` - General property used to inform GateIn Portal that click to *Sign in* will redirect user to URL ending with `/portal/dologin` . This is described in more details in Central Authentication Service (CAS)#Configuration properties details
- `gatein.sso.callback.enabled` - This could be false as we don't need REST callback for SPNEGO integration
- `gatein.sso.skip.jsp.redirection` - This should be false for SPNEGO, especially if you want fallback to FORM authentication to work properly (More details in Disable fallback to FORM authentication)
- `gatein.sso.login.module.enabled` - This can be false as we need slightly different set of login modules for SPNEGO integration. More details below where will be detailed login modules configuration
- `gatein.sso.filter.login.sso.url` - Click to *Sign in* should redirect us to URL */portal/dologin* , which is secured URL (declared in *security-constraint* section of `GATEIN_HOME/gatein/gatein.ear/portal.war/WEB-INF/web.xml` file), which will allow `GateInNegotiationAuthenticator` valve to intercept this HTTP request
- `gatein.sso.filter.logout.enabled`, `gatein.sso.filter.initiatelogin.enabled` - Logout filter and InitiateLoginFilter are not needed for SPNEGO integration
- `gatein.sso.valve.enabled` - This option points that SPNEGO integration requires enabling of custom Tomcat valve, which is used to intercept HTTP request for secured URL (/portal/dologin). The `SSODelegateValve` is defined in file `GATEIN_HOME/gatein/gatein.ear/portal.war/WEB-INF/jboss-web.xml` and is used only if the option `gatein.sso.valve.enabled` is true. The purpose of SSODelegateValve is to delegate the real work to another valve declared in option `gatein.sso.valve.class` . The point is that you don't need to edit anything in file `jboss-web.xml` as everything can be configured via properties.
- `gatein.sso.valve.class` - So for SPNEGO integration the delegate valve is `org.gatein.sso.spnego.GateInNegotiationAuthenticator`, which is used to establish identity of client by exchanging few handshakes with client's browser. The valve is able to obtain and parse SPNEGO token and resend it to JAAS layer, where login modules (especially `SPNEGOLoginModule`) are able to verify validity of wrapped Kerberos token and establish user identity at GateIn Portal layer.

2. **JAAS configuration** - Configuration of login modules for SPNEGO integration is not trivial and so it's not simply configurable only via `configuration.properties`, but it require changing of JAAS stack. You will need to do those changes in file `GATEIN_HOME/standalone/configuration/standalone.xml` in configuration of security-domains:

1. New configuration for security-domain *host* needs to be added:

```
<security-domain name="host">
  <authentication>
    <login-module code="com.sun.security.auth.module.Krb5LoginModule"
flag="required">
      <module-option name="storeKey" value="true" />
      <module-option name="useKeyTab" value="true" />
      <module-option name="principal"
value="HTTP/server.local.network@LOCAL.NETWORK" />
      <module-option name="keyTab" value="/etc/krb5.keytab" />
      <module-option name="doNotPrompt" value="true" />
      <module-option name="debug" value="true" />
    </login-module>
  </authentication>
</security-domain>
```

   It's used internally by JBoss negotiation to verify server principal from Kerberos keytab. Important options, which you may need to change are:
   - **principal** -- This points to HTTP server principal from your kerberos keytab. In SPNEGO Server Configuration we used Kerberos realm LOCAL.NETWORK and host *server.local.network* and so we added principal *HTTP/server.local.network@LOCAL.NETWORK* to our keytab.
   - **keyTab** -- This points to file with your Kerberos keytab. In Kerberos configuration we used location */etc/krb5.keytab* . Please note that this file should be readable by user, under which is GateIn Portal executed .

   Generally Kerberos keytab files shouldn't be readable by normal OS users as they contain encrypted keys of server principals. Consult Kerberos documentation for more details

   - **debug** -- Enabling this option will add more STDOUT messages into GateIn Portal console and log file. It's good to have it enabled during setup, but in production it may be good to disable it to avoid huge server logs.
2. Existing `gatein-domain` needs to be renamed to `gatein-form-auth-domain`
3. New configuration for `gatein-domain` needs to be added:

```
<security-domain name="gatein-domain" cache-type="default">
  <authentication>
    <login-module code="org.gatein.sso.spnego.SPNEGOLoginModule" flag="requisite">
      <module-option name="password-stacking" value="useFirstPass"/>
      <module-option name="serverSecurityDomain" value="host"/>
      <module-option name="removeRealmFromPrincipal" value="true"/>
      <module-option name="usernamePasswordDomain"
value="gatein-form-auth-domain"/>
    </login-module>
    <login-module code="org.gatein.sso.agent.login.SPNEGORolesModule"
flag="required">
      <module-option name="password-stacking" value="useFirstPass"/>
      <module-option name="portalContainerName" value="portal"/>
      <module-option name="realmName" value="gatein-domain"/>
    </login-module>
  </authentication>
</security-domain>
```

So with all 3 changes the configuration in *standalone.xml* file should be similar to this:

```
<security-domain name="gatein-form-auth-domain" cache-type="default">
  <authentication>
    <login-module code="org.gatein.sso.integration.SSODelegateLoginModule"
flag="required">
      <module-option name="enabled" value="${gatein.sso.login.module.enabled}" />
      <module-option name="delegateClassName"
value="${gatein.sso.login.module.class}" />
      <module-option name="portalContainerName" value="portal" />
      <module-option name="realmName" value="gatein-domain" />
      <module-option name="password-stacking" value="useFirstPass" />
    </login-module>
    <login-module code="org.exoplatform.services.security.j2ee.JBossAS7LoginModule"
flag="required">
      <module-option name="portalContainerName" value="portal"/>
      <module-option name="realmName" value="gatein-domain"/>
    </login-module>
  </authentication>
</security-domain>

<security-domain name="host">
  <authentication>
    <login-module code="com.sun.security.auth.module.Krb5LoginModule"
flag="required">
      <module-option name="storeKey" value="true" />
      <module-option name="useKeyTab" value="true" />
      <module-option name="principal"
value="HTTP/server.local.network@LOCAL.NETWORK" />
      <module-option name="keyTab" value="/etc/krb5.keytab" />
      <module-option name="doNotPrompt" value="true" />
      <module-option name="debug" value="true" />
    </login-module>
  </authentication>
</security-domain>

<security-domain name="gatein-domain" cache-type="default">
  <authentication>
    <login-module code="org.gatein.sso.spnego.SPNEGOLoginModule" flag="requisite">
      <module-option name="password-stacking" value="useFirstPass"/>
      <module-option name="serverSecurityDomain" value="host"/>
      <module-option name="removeRealmFromPrincipal" value="true"/>
      <module-option name="usernamePasswordDomain"
value="gatein-form-auth-domain"/>
    </login-module>
    <login-module code="org.gatein.sso.agent.login.SPNEGORolesModule"
flag="required">
      <module-option name="password-stacking" value="useFirstPass"/>
      <module-option name="portalContainerName" value="portal"/>
      <module-option name="realmName" value="gatein-domain"/>
    </login-module>
  </authentication>
</security-domain>
```

3. So assuming that your kerberos realm is *LOCAL.NETWORK* and your kerberos KDC is on host *server.local.network* and you will run GateIn Portal on same host *server.local.network*, you will need to use this command to start your GateIn Portal (Don't forget again that it needs to be executed under user with read permission to kerberos keytab file as mentioned above):

```
./standalone.sh -Djava.security.krb5.realm=LOCAL.NETWORK
-Djava.security.krb5.kdc=server.local.network -b server.local.network
```

4. Testing SPNEGO - Log into Kerberos with the *kinit -A demo* command (on linux).
   You should be able to go to http://server.local.network:8080/portal and click the 'Sign In' link on the GateIn Portal. Then the "demo" user should be automatically logged into GateIn Portal.
5. Testing FORM fallback - Let's destroy the kerberos ticket with command *kdestroy*. Then try to logout from GateIn Portal and login again. You will now be placed to GateIn Portal login screen because you do not have active Kerberos ticket. You can log in with predefined account and password "demo"/"gtn" . It works this way because you have SPNEGO integration enabled and integrated with your kerberos, but if user doesn't have active Kerberos ticket, it will fallback to default *FORM* based authentication, so user has possibility to fill his credentials classically via GateIn Portal login screen.

Disable fallback to FORM authentication

As mentioned in previous paragraph, the fallback to FORM authentication is automatically enabled. This means that user can authenticate either via SPNEGO handshake with his kerberos ticket or via providing his credentials in GateIn Portal login form, which will perform verification of credentials against Picketlink IDM database.

For some reason you may want to enforce authentication only via SPNEGO and disable possibility to authenticate with GateIn Portal login FORM. In this case you will need to edit configuration of `SPNEGOLoginModule` in `GATEIN_HOME/standalone/configuration/standalone.xml` and comment the `usernamePasswordDomain` option:

```
<login-module
    code="org.gatein.sso.spnego.SPNEGOLoginModule" flag="requisite">
    <module-option name="password-stacking" value="useFirstPass"/>
    <module-option name="serverSecurityDomain" value="host"/>
    <module-option name="removeRealmFromPrincipal" value="true"/>
<!--<module-option name="usernamePasswordDomain" value="gatein-form-auth-domain"/>-->
</login-module>
```

Tips & Tricks

- **Logging** - Like for other SSO solutions, it may be good to enable trace logging for `org.gatein.sso` etc but in case of SPNEGO, it's good also to enable logging for `org.jboss.security.negotiation`, which is base package for JBoss Negotiation library. You can add these categories to `JBOSS_HOME/standalone/configuration/standalone.xml` :

```
<logger category="org.gatein.sso">
  <level name="TRACE"/>
</logger>
<logger category="org.jboss.security.negotiation">
  <level name="TRACE"/>
</logger>
```

  For Kerberos troubleshooting, you can enable increased logging to STDOUT by enable option *debug* of Krb5LoginModule as already described in section with login module configuration. Very helpful could be enabling system property for increased Kerberos logging by adding this to command line when running server:

```
-Dsun.security.krb5.debug=true
```

- **Kerberos documentation**
  - Kerberos FAQ: http://www.cmf.nrl.navy.mil/CCS/people/kenh/kerberos-faq.html
  - Kerberos documentation for your linux distribution: Install package *krb5-doc* and enjoy documentation in directory `/usr/share/doc/krb5-doc` . There should be admin, install and user guide for Kerberos on your linux distribution.
    Note that Kerberos setup is OS specific and it's different for Windows with AD domain.

6.9.6 SAML2

SAML (Security Assertion Markup Language) is Oasis standard for exchanging authentication and authorization data between security domains. SAML 2.0 is an XML-based protocol that uses security tokens containing assertions to pass information about a principal (usually an end user) between an identity provider and a web service. SAML 2.0 enables web-based authentication and authorization scenarios including single sign-on (SSO).

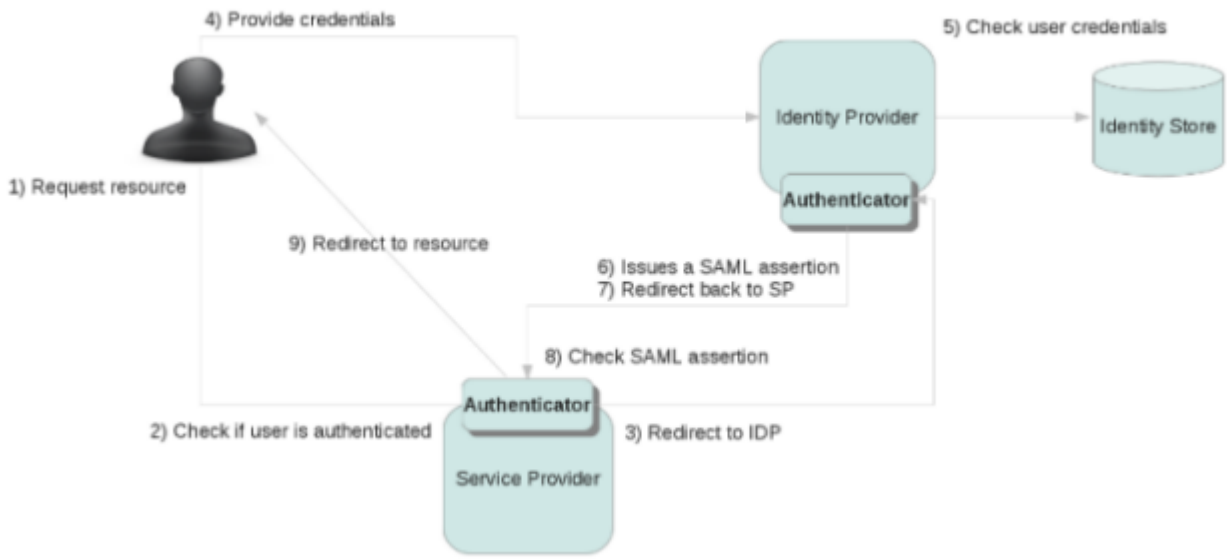
SAML2 standard is described in set of specifications, which provides exact format of XML messages and context how these messages are exchanged between Identity Provider (IDP, Web application, which acts as SSO provider and users are authenticated against it) and Service Provider (SP, Web application, which is used by client who wants to authenticate). See http://docs.oasis-open.org/security/saml/v2.0/ for more information about specifications.

The SAML2-based authentication is provided in GateIn Portal SSO component. We support scenarios with GateIn Portal acting as Service Provider (SP) or Identity Provider (IDP).

SAML2 Overview and authentication workflow

For GateIn Portal and SAML2 integration, we are using JBoss project Picketlink Federation, which provides solution for most important parts of SAML2 specification. Especially it supports SSO authentication with SAML2 HTTP Redirect Binding and SAML2 HTTP Post Binding and it supports SAML2 Single Logout feature.

SSO authentication is based on circle of trust between SP and IDP.



The authentication works as follows (flow with GateIn Portal as SAML2 SP):

1. User sends request to secured resource like http://localhost:8080/portal/dologin
2. GateIn Portal will check if user is already authenticated and if yes, grant access to resource. Otherwise continue with flow below.

3. There is special Tomcat valve, which needs to be configured for portal context. This Valve will create SAML Request, which is basically XML message. Example of message:

```
<samlp:AuthnRequest AssertionConsumerServiceURL="http://localhost:8080/portal/dologin"
ID="ID_101dcb5e-f432-4f45-87cb-47daff92edef" IssueInstant="2012-04-12T17:53:27.294+01:00"
ProtocolBinding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST" Version="2.0">
   <saml:Issuer>http://localhost:8080/portal/dologin</saml:Issuer>
   <samlp:NameIDPolicy AllowCreate="true"
Format="urn:oasis:names:tc:SAML:2.0:nameid-format:transient"/>
</samlp:AuthnRequest>
```

   Valve will encapsulate SAML request into HttpResponse and it redirects it to IDP. Picketlink Federation supports SAML Redirect Binding, which basically means that SAML XML Request message is Base64 encoded and URL encoded and it is appended as URL parameter to GET request, which will be send to IDP. PL Fed also supports SAML POST Binding where is message encoded into Base64 and sent in the body of POST request.
4. IDP parses XML with SAML request and it sends login screen back to client. Now client (user) needs to authenticate himself. SAML specification does not mandate how exactly should be authentication of client on IDP side performed.
5. User fills his credentials into IDP FORM and submits request for JAAS authentication. GateIn Portal SSO component provides login module `SAML2IdpLoginModule`, which will authenticate user by sending callback request via REST API back to GateIn Portal. This is similar approach like authentication with other SSO providers like CAS, which are also leveraging this REST service.

Portal administrators are free to use their own login module stack instead of our REST callback based login module. However they need to make sure that authenticated users also need to exist in GateIn Portal database. Otherwise their users may have authorization errors with 403 response when they try to access portal.

6. So after successful authentication, IDP will create SAML assertion ticket and it creates SAML Response message with this ticket. Message can looks like this:

```
<samlp:Response ID="ID_5291c49e-5450-4b3b-9f99-f76606db9929" Version="2.0"
IssueInstant="2012-04-12T17:53:59.237+01:00"
Destination="http://localhost:8080/portal/dologin"
InResponseTo="ID_101dcb5e-f432-4f45-87cb-47daff92edef">
   <saml:Issuer>http://localhost:8080/idp/</saml:Issuer>
   <samlp:Status>
      <samlp:StatusCode Value="urn:oasis:names:tc:SAML:2.0:status:Success"/>
   </samlp:Status>

   <saml:Assertion ID="ID_ebe89398-1e27-4257-9413-c3c17c40c9df" Version="2.0"
IssueInstant="2012-04-12T17:53:59.236+01:00">
      <saml:Issuer
Format="urn:oasis:names:tc:SAML:2.0:nameid-format:persistent">root</saml:Issuer>
      <saml:Subject>
         <saml:NameID
Format="urn:oasis:names:tc:SAML:2.0:nameid-format:persistent">root</saml:NameID>
         <saml:SubjectConfirmation Method="urn:oasis:names:tc:SAML:2.0:cm:bearer">
            <saml:SubjectConfirmationData
InResponseTo="ID_101dcb5e-f432-4f45-87cb-47daff92edef"
NotBefore="2012-04-12T17:53:59.236+01:00" NotOnOrAfter="2012-04-12T17:54:06.236+01:00"
Recipient="http://localhost:8080/portal/dologin"/>
         </saml:SubjectConfirmation>
      </saml:Subject>
      <saml:Conditions NotBefore="2012-04-12T17:53:57.236+01:00"
NotOnOrAfter="2012-04-12T17:54:06.236+01:00"/>
      <saml:AuthnStatement AuthnInstant="2012-04-12T17:53:59.237+01:00">
         <saml:AuthnContext>

<saml:AuthnContextClassRef>urn:oasis:names:tc:SAML:2.0:ac:classes:Password</saml:AuthnContex
</saml:AuthnContext>
      </saml:AuthnStatement>
      <saml:AttributeStatement>
         <saml:Attribute Name="Role">
            <saml:AttributeValue xsi:type="xs:string">users</saml:AttributeValue>
         </saml:Attribute>
         <saml:Attribute Name="Role">
            <saml:AttributeValue xsi:type="xs:string">administrators</saml:AttributeValue>
         </saml:Attribute>
      </saml:AttributeStatement>
   </saml:Assertion>
</samlp:Response>
```

7. Message is then encapsulated into HttpResponse and redirected back to SP (GateIn Portal).
8. On GateIn Portal side is SAML response message decoded again by the Tomcat Valve and if assertion from response is valid, then username and his roles are added into ThreadLocal context variable. Valve then triggers JAAS authentication. GateIn Portal component will provide login module `SAML2IntegrationLoginModule`, which will parse authenticated username and it will perform GateIn Portal specific operations, like creating Identity object and registering it into IdentityRegistry. Now user is successfully authenticated.

9. User is redirected back to the secure resource http://localhost:8080/portal/dologin, which in next turn will redirect him to GateIn Portal as authenticated user. If the user wants to authenticate against different SP application within same browser session (GateIn Portal on different host or completely different web application), then he does not need to provide credentials again on IDP side because he has been already authenticated against IDP. So he has automatic authentication thanks to SSO.

Previous scenario covered the case with GateIn Portal as SAML2 SP and different web application acting as SAML2 IDP. Scenario with GateIn Portal as SAML2 IDP is little different (for example there is no usage of SAML2IdpLoginModule). Both cases will be covered by example below.

In next sections, we will go through various scenarios, which describes how you can leverage SAML2 in GateIn Portal and there is description of all needed configuration changes.

Portal as SAML2 SP and SAML2 IDP

In this scenario we will use GateIn Portal as SAML2 SP and another instance of GateIn Portal as SAML2 IDP. SP will be executed on virtual host *www.sp.com* and IDP will be on virtual host *www.idp.com* . Both GateIn Portal instances will be on JBoss AS7. The idea is that users, who want to authenticate against GateIn Portal on *www.sp.com* will be redirected to *www.idp.com* where they need to authenticate and after successful authentication on IDP the *SAMLResponse* will be send back to *www.sp.com* and user will be redirected back.

You can look at video http://vimeo.com/45841256, which is showing GateIn Portal and SAML2 in practice

Configuration

1. We assume that you have two virtual hosts running on single physical machine. So you will need to add virtual hosts to the `/etc/hosts` file. On Linux, it can be done by adding some entries like those (use real IP addresses according to your environment):

```
192.168.2.7 www.sp.com
10.11.11.141 www.idp.com
```

2. We assume that GateIn Portal, which will act as SAML2 SP is in directory *GATEIN_SP_HOME*. You need to configure properties in SSO section of file *GATEIN_SP_HOME/standalone/configuration/gatein/configuration.properties* like this:

```
# SSO
gatein.sso.enabled=true
gatein.sso.callback.enabled=${gatein.sso.enabled}
gatein.sso.login.module.enabled=${gatein.sso.enabled}
gatein.sso.login.module.class=org.gatein.sso.agent.login.SAML2IntegrationLoginModule
gatein.sso.filter.login.sso.url=/@@portal.container.name@@/dologin
gatein.sso.filter.logout.enabled=true
gatein.sso.filter.logout.class=org.gatein.sso.agent.filter.SAML2LogoutFilter
gatein.sso.filter.initiatelogin.enabled=false
gatein.sso.valve.enabled=true
gatein.sso.valve.class=org.picketlink.identity.federation.bindings.tomcat.sp.ServiceProvider
WARNING: This bundled keystore is only for testing purposes. You should generate and use
your own keystore!
gatein.sso.picketlink.keystore=/sso/saml/jbid_test_keystore.jks
```

Most of the properties were already described in Central Authentication Service (CAS)#Configuration properties details or in SPNEGO#Portal configuration . The ones, which are specific/different for SAML2 SP are:

- `gatein.sso.login.module.class` - For GateIn Portal integration as SAML2 SP, we need to use login module class *org.gatein.sso.agent.login.SAML2IntegrationLoginModule*, which will act as JAAS delegate from SSODelegateLoginModule
- `gatein.sso.filter.logout.class` - Specific filter class *org.gatein.sso.agent.filter.SAML2LogoutFilter* is needed to enable support for SAML2 single logout. SAML2 Single logout is special SAML profile, which support logout from all SAML2 SP applications and from SAML2 IDP application at once. Currently it means that when you are logged in GateIn Portal and you click "Sign out", you will be redirected by *SAML2LogoutFilter* to your SAML2 IDP application where you will be logged out and also IDP will take care of logout from all other SP applications. Finally you will be redirected back to GateIn Portal and logged out here. See Disable SAML2 Single logout if you don't want SAML2 Single logout enabled.
- `gatein.sso.valve.class` - There is special valve *org.picketlink.identity.federation.bindings.tomcat.sp.ServiceProviderAuthenticator* provided by Picketlink federation library. This valve will take care of create and parse SAMLRequest and SAMLResponse messages, so it's one of the most important part for successful SAML2 integration.
- `gatein.sso.saml.config.file` - This points to location of SAML2 SP configuration file. Path is relative to portal WAR application, so real location of file is *GATEIN_SP_HOME/gatein/gatein.ear/portal.war/WEB-INF/conf/sso/saml/picketlink-sp.xml* . For this simple scenario, you won't need to change anything in this file. But in production environment, you will likely need to generate and use your own keystore and in this case, you need to configure it in this file *picketlink-sp.xml*. More details in Generate and configure your own keystore .
- `gatein.sso.idp.host` - This points to host with SAML2 IDP.
- `gatein.sso.idp.url` - This points to URL of IDP application. We assume that you will use another instance of GateIn Portal as SAML2 IDP, so request path will be */portal/dologin* in this case.
- `gatein.sso.sp.url` - This points to URL of this GateIn Portal, which will be used as SAML2 SP.
- `gatein.sso.picketlink.keystore` - This points to location of keystore file. It's relative to classpath of portal WAR application, so real location is *GATEIN_SP_HOME/gatein/gatein.ear/portal.war/WEB-INF/classes/sso/saml/jbid_test_keystore*

This bundled keystore *jbid_test_keystore.jks* is intended to be used only for testing purposes. In production environment, you should generate and configure your own keystore. See Generate and configure your own keystore for details.

3. We assume that GateIn Portal, which will act as SAML2 IDP is in directory *GATEIN_IDP_HOME*. You need to configure properties in SSO section of file *GATEIN_IDP_HOME/standalone/configuration/gatein/configuration.properties* like this:

```
# SSO
gatein.sso.enabled=false
gatein.sso.valve.enabled=true
gatein.sso.valve.class=org.gatein.sso.saml.plugin.valve.PortalIDPWebBrowserSSOValve
gatein.sso.saml.config.file=/WEB-INF/conf/sso/saml/picketlink-idp.xml
gatein.sso.idp.url=http://www.idp.com:8080/portal/dologin
gatein.sso.idp.listener.enabled=true
gatein.sso.sp.domains=sp.com
gatein.sso.sp.host=www.sp.com
# WARNING: This bundled keystore is only for testing purposes. You should generate and use
your own keystore in production!
gatein.sso.picketlink.keystore=/sso/saml/jbid_test_keystore.jks
```

Details about properties:

- `gatein.sso.enabled` - You can see that it's false now. It's because GateIn Portal IDP won't use any external SSO provider, but now it will act itself as SSO provider (accurately SAML2 Identity Provider) for GateIn Portal SP.
- `gatein.sso.valve.class` - For IDP we need to use *org.gatein.sso.saml.plugin.valve.PortalIDPWebBrowserSSOValve* valve, which is able to handle SAML request/response messages from SP applications and react on them.
- `gatein.sso.saml.config.file` - Location of configuration file is again relative to portal WAR, so it's *GATEIN_IDP_HOME/gatein/gatein.ear/portal.war/WEB-INF/conf/sso/saml/picketlink-idp.xml*. For this simple scenario, you won't need to change anything in this file. But in production environment, you will likely need to generate and use your own keystore and in this case, you need to configure it in this file. More details Generate and configure your own keystore . You will also to manually add validating aliases in keystore section if you have more different SP applications on different hosts.
- `gatein.sso.idp.url` - URL of this GateIn Portal, which will act as SAML2 IDP
- `gatein.sso.idp.listener.enabled` - This will enable special session listener for cleanup records about SAML tickets of expired hosts
- `gatein.sso.sp.domains` - Comma separated list of all SP domains, which will be trusted by this IDP
- `gatein.sso.sp.host` - Host of application with GateIn Portal SP. If you want more SP applications, you will need to manually edit *picketlink-idp.xml* file and add *ValidatingAlias* element for each of them.
- `gatein.sso.picketlink.keystore` - Keystore in *GATEIN_IDP_HOME/gatein/gatein.ear/portal.war/WEB-INF/classes/sso/saml/jbid_test_keystor*

This bundled keystore *jbid_test_keystore.jks* is intended to be used only for testing purposes. In production environment, you should generate and configure your own keystore. See Generate and configure your own keystore for details.

4. Start SP with:

```
cd GATEIN_SP_HOME/bin
./standalone.sh -b www.sp.com
```

5. Start IDP with:

```
cd GATEIN_IDP_HOME/bin
./standalone.sh -b www.idp.com
```

6. Test it - You can go to URL http://www.sp.com:8080/portal/classic and click to *Sign in*. At this moment, SAML request will be sent to *www.idp.com* and you will be redirected to login screen of GateIn Portal IDP on http://www.idp.com:8080/portal/dologin . After successful login, SAML response will be sent back to *www.sp.com* and you will be redirected to GateIn Portal on *www.sp.com* as logged user. Now you can go to *www.idp.com:8080/portal* and you will see that you are logged due to SSO because you already logged in this host.
   Now let's go back to *www.sp.com:8080/portal* and click to *Sign out* . Now SAML2 single logout will take place and will logged you from both GateIn Portal on IDP and SP. You can check that you are logged out from both *www.sp.com:8080/portal* and *www.idp.com:8080/portal*

Disable SAML2 Single logout

SAML2 Single logout is very powerful feature of SAML2 protocol, because trigger logout from any SP application will enforce "global" logout also from IDP and all other SP applications. It makes sense that you may not want this feature to be enabled, so trigger logout from GateIn Portal (click to *Sign out* link) will logout user only from GateIn Portal on *www.sp.com*, but user will be still logged on GateIn Portal IDP on *www.idp.com* and on all other SP applications.

Disable this feature is as easy as switch option in file `GATEIN_SP_HOME/standalone/configuration/gatein/configuration.properties` :

```
gatein.sso.filter.logout.enabled=false
```

Now when you click *Sign out* in *www.sp.com*, you will be still logged on GateIn Portal on *www.idp.com* . There won't be any SAML communication between SP and IDP during logout from *www.sp.com* .

Generate and configure your own keystore

In this procedure, you will generate and use your own Keystores. This will add more safety into trusted communication between GateIn Portal and IDP because default packaging is using prepackaged keystore `jbid_test_keystore.jks`. For secure and trusted communication, you will need your own keystores with your own keys. Default jbid_test_keystore is useful only for testing purpose, but **it really can't be used in production!!!**

1. Go to `GATEIN_SP_HOME/gatein/gatein.ear/portal.war/WEB-INF/classes/sso/saml` directory and run command:

```
keytool -genkey -alias secure-key -keyalg RSA -keystore secure-keystore.jks
```

   You need to choose keystore password and private key password. Other values doesn't matter. Let's assume that your keystore password is *keystorepass* and a private key password is *keypass* .
2. For simplification purposes, we will use same keystore for SP and IDP. So copy the generated keystore file also to IDP to directory `GATEIN_IDP_HOME/gatein/gatein.ear/portal.war/WEB-INF/classes/sso/saml` .
3. Configure your new keystore in file `GATEIN_SP_HOME/gatein/gatein.ear/portal.war/WEB-INF/conf/sso/saml/picketlin` and replace existing KeyProvider definition with:

```
<KeyProvider ClassName="org.picketlink.identity.federation.core.impl.KeyStoreKeyManager">
  <Auth Key="KeyStoreURL" Value="/sso/saml/secure-keystore.jks"/>
  <Auth Key="KeyStorePass" Value="keystorepass"/>
  <Auth Key="SigningKeyPass" Value="keypass"/>
  <Auth Key="SigningKeyAlias" Value="secure-key"/>
  <ValidatingAlias Key="${gatein.sso.idp.host}" Value="secure-key"/>
</KeyProvider>
```

4. Similarly configure keystore in IDP file `GATEIN_IDP_HOME/gatein/gatein.ear/portal.war/WEB-INF/conf/sso/saml/picketli` :

```
<KeyProvider ClassName="org.picketlink.identity.federation.core.impl.KeyStoreKeyManager">
  <Auth Key="KeyStoreURL" Value="/sso/saml/secure-keystore.jks"/>
  <Auth Key="KeyStorePass" Value="keystorepass"/>
  <Auth Key="SigningKeyPass" Value="keypass"/>
  <Auth Key="SigningKeyAlias" Value="secure-key"/>
  <ValidatingAlias Key="${gatein.sso.sp.host}" Value="secure-key"/>
</KeyProvider>
```

5. Restart both SP and IDP. Now it will use your new keystore instead of default `jbid_test_keystore.jks`

It may be slightly better to use certificates signed by certification authority. But for our purpose, it is fine to use self-signed certificates. For more information, you can check additional sources like http://docs.oracle.com/javase/tutorial/security/sigcert/index.html.

Setup with external IDP using REST callback

In previous section, you saw how to configure GateIn Portal to act as SAML2 SP or as SAML2 IDP. Note that you can use GateIn Portal IDP with any other SP web applications. And on the opposite site, you can configure your GateIn Portal SP to authenticate against some external IDP application. You can consult documentation of Picketlink Federation for more details about setup.

But in addition we provide modified version of *idp-sig* quickstart application from Picketlink Federation. Our modified version is again using REST callback to authenticate against existing GateIn Portal instance. So it's similar like for other SSO scenarios and details are in Central Authentication Service (CAS)#Login workflow . In this case, we need to configure special login module `SAML2IdpLoginModule`, which is able to connect to GateIn Portal via REST and verify against portal database if provided credentials are valid.

So we assume that you have instance of JBoss AS7 where we deploy our *idp-sig.war* application. This server will run again on *www.idp.com*. GateIn Portal will be configured as SP on *www.sp.com* and it will use this IDP from *www.idp.com* to authenticate.

1. GateIn Portal SP setup - You need to configure properties in SSO section of file `GATEIN_SP_HOME/standalone/configuration/gatein/configuration.properties` like this:

   ```
   gatein.sso.enabled=true
   gatein.sso.callback.enabled=${gatein.sso.enabled}
   gatein.sso.login.module.enabled=${gatein.sso.enabled}
   gatein.sso.login.module.class=org.gatein.sso.agent.login.SAML2IntegrationLoginModule
   gatein.sso.filter.login.sso.url=/@@portal.container.name@@/dologin
   gatein.sso.filter.logout.enabled=true
   gatein.sso.filter.logout.class=org.gatein.sso.agent.filter.SAML2LogoutFilter
   gatein.sso.filter.initiatelogin.enabled=false
   gatein.sso.valve.enabled=true
   gatein.sso.valve.class=org.picketlink.identity.federation.bindings.tomcat.sp.ServiceProvider
   WARNING: This bundled keystore is only for testing purposes. You should generate and use
   your own keystore!
   gatein.sso.picketlink.keystore=/sso/saml/jbid_test_keystore.jks
   ```

   There are some differences from previous setup. Now `gatein.sso.callback.enabled` is true because `idp-sig.war` application need to handle REST callback requests from idp-sig application. Also `gatein.sso.idp.url` is changed as idp will run on URI */idp-sig/* .
2. Start SP with:

   ```
   cd GATEIN_SP_HOME/bin
   ./standalone.sh -b www.sp.com
   ```

3. IDP setup:
    1. Take another instance of GateIn Portal , which will be used for IDP setup. The directory with this instance will be referred as AS7_IDP_HOME.

        We won't use GateIn Portal itself for IDP setup, but we will use independent web application `idp-sig` . So theoretically you can use plain AS7 without GateIn Portal deployed. But note that AS7 instance with GateIn Portal contains some additional modules, which are needed for our `idp-sig.war` application. So it's still recommended to use GateIn Portal instead of plain AS7.

    2. Copy `GATEIN_SSO_HOME/saml/idp-sig.war` into `AS7_IDP_HOME/standalone/deployments` . GATEIN_SSO_HOME points to directory with SSO packaging. See Single-Sign-On (SSO) for more details.
    3. Create empty file `AS7_IDP_HOME/standalone/deployments/idp-sig.war.dodeploy` (It's needed because `idp-sig.war` is exploded archive right now)
    4. New IDP security domain needs to be added to `AS7_IDP_HOME/standalone/configuration/standalone.xml`

        ```
        <security-domain name="idp" cache-type="default">
           <authentication>
              <login-module code="org.gatein.sso.saml.plugin.SAML2IdpLoginModule"
        flag="required">
                 <module-option name="rolesProcessing" value="STATIC"/>
                 <module-option name="staticRolesList" value="manager,employee,sales"/>
                 <module-option name="gateInURL" value="http://www.sp.com:8080/portal"/>
              </login-module>
           </authentication>
        </security-domain>
        ```

        Note especially option `gateInURL`, which contains base URL for sending authentication callback to GateIn Portal, so this needs to point to URL where your portal is running.
    5. Run IDP with:

        ```
        cd AS7_IDP_HOME/bin
        ./standalone.sh -b www.idp.com -Dsp.host=www.sp.com -Dsp.domains=sp.com
        -Dpicketlink.keystore=/jbid_test_keystore.jks
        ```

        All these properties are used in picketlink configuration file of `idp-sig.war` application in `AS7_IDP_HOME/standalone/deployments/idp-sig.war/WEB-INF/picketlink.xml`
4. Test - you can go to http://www.sp.com:8080/portal and click *Sign in*. You will be redirected to idp-sig application on http://www.idp.com:8080/idp-sig/. Now you can login with GateIn Portal credentials like "john"/"gtn" because REST callback will be sent to portal on *www.sp.com* and it will take care of user authentication. After authentication you will be redirected to portal on *www.sp.com* and logged as john.

Make sure that in production environment, you also need to use your own keystore for this second scenario instead of the prebundled `jbid_test_keystore.jks`, which is in idp-sig application in `WEB-INF/classes/`

Integration with Salesforce and Google Apps

GateIn Portal SSO component contains the support for Salesforce and Google Apps integration for SAML2 based SSO. Supported scenarios are:

- GateIn Portal as SAML Identity Provider, Salesforce as SAML Service Provider
- GateIn Portal as Identity Provider, Google Apps as SAML Service Provider
- GateIn Portal as SAML Service Provider, Salesforce as SAML Identity Provider

In next sections, you can see detailed description of all scenarios with needed configuration changes.

You can look at video http://vimeo.com/45895919, which is showing how the integration with Salesforce and Google Apps works in practice.

Portal as SAML IDP, Salesforce as SAML SP

- **Base configuration of** GateIn Portal **as IDP** - first you need to configure GateIn Portal as SAML IDP as described in Portal as SAML2 SP and SAML2 IDP .
- **Common steps for Salesforce integration** are described in Picketlink documentation and they are quite similar for GateIn Portal like for another web application. Here we describe only steps different for GateIn Portal, which can be divided into configuration on Salesforce side and configuration on GateIn Portal side:

Salesforce configuration

- In **Salesforce setup in SSO configuration**, you need to configure *Issuer*, *Identity Provider Login URL* and *Identity Provider Logout URL* to value http://www.idp.com:8080/portal/dologin (Assuming that you are using virtual host *www.idp.com* as described in Portal as SAML2 SP and SAML2 IDP)

- **Certificate** - you need to import correct certificate into Salesforce. It needs to be certificate, which is used for signing of SAML messages from GateIn Portal. You can export certificate from directory **GATEIN_IDP_HOME/gatein/gatein.ear/portal.war/WEB-INF/classes/sso/saml** with command:

```
keytool -export -file portal-idp.crt -keystore secure-keystore.jks -alias secure-key
```

This certificate then needs to be imported into Salesforce as described in Picketlink documentation . We assumed that you followed steps in Generate and configure your own keystore and your certificate is in file `secure-keystore.jks` with alias name `secure-key` . Otherwise change command according to name of your keystore and signing key aliases.

- **Create some users** in your Salesforce domain, which are available in GateIn Portal as well. You can use inverse approach as well and create users in GateIn Portal, which are present in Salesforce. Important is mapping - **username in portal** is mapped to **Federation ID in Salesforce**. So for example GateIn Portal user *mary* is mapped to Salesforce user in your domain, which has Federation ID *mary* .

Portal configuration

- Make sure you import Salesforce client certificate into your GateIn Portal keystore. Command for doing it can looks like:

```
keytool -import -keystore secure-keystore.jks -file /tmp/proxy-salesforce-com.123 -alias
salesforce-cert
```

- Configure ValidatingAlias and Metadata similarly like for base Picketlink tutorial . Note that main Picketlink configuration file where you need to do changes is `GATEIN_IDP_HOME/gatein/gatein.ear/portal.war/WEB-INF/conf/sso/saml/picketli` and assumption is that you will add your metadata into `GATEIN_IDP_HOME/gatein/gatein.ear/portal.war/WEB-INF/sp-metadata.xml` .
- Trusted domain are configured through configuration.properties . So property `gatein.sso.sp.domains` in file `GATEIN_IDP_HOME/standalone/configuration/gatein/configuration.properties` should look like this:

```
gatein.sso.sp.domains=sp.com,idp.com,salesforce.com
```

Test the integration - After configure both IDP and SP, you can test that after accessing URL of your Salesforce domain like https://yourdomain.my.salesforce.com , SAML Request will be sent to GateIn Portal and you will be redirected to your virtual host to GateIn Portal login screen on http://www.idp.com:8080/portal/dologin . You can login with some GateIn Portal user (like *mary*) and you should be redirected back to Salesforce and automatically logged here as user *mary* thanks to SAML SSO.

Portal as SAML IDP, Google Apps as SAML SP

Again, we assume that you followed generic instructions for portal as IDP in Portal as SAML2 SP and SAML2 IDP .

The common steps for Google Apps integration with classic web application are again described in Picketlink guide, so we will pay attention only to steps specific for GateIn Portal.

Google configuration

- **Create some users** in Google Apps, which are available in GateIn Portal as well (or vice-versa). **Username from portal** is mapped to **nick in Google Apps**. For example GateIn Portal user *mary* is standardly mapped to Google Apps user mary@yourgoogledomain.com , which normally has nick *mary* .
- In **SSO configuration of Google Apps domain**, you need to change all Sign-In and Sign-Out URL to value http://www.idp.com:8080/portal/dologin instead of http://localhost:8080/idp-sig/ , which is used in Picketlink guide (Assuming that you are using virtual host *www.idp.com* as described in Portal as SAML2 SP and SAML2 IDP)
- You again need to export certificate from your GateIn Portal keystore into some file (for example `portal-idp.crt` as described in Portal as SAML2 SP and SAML2 IDP) and then import it from this file into Google Apps through Google Apps UI.

Portal configuration

- **Metadata configuration** - you need to configure metadata in similar way like in Picketlink tutorial . Assumption is that metadata are in file `GATEIN_IDP_HOME/gatein/gatein.ear/portal.war/WEB-INF/sp-metadata.xml` .

- **Trusted domains** are configured through configuration.properties . You need to add *google.com* domain like described in Picketlink tutorial. So property `gatein.sso.sp.domains` in file `GATEIN_IDP_HOME/standalone/configuration/gatein/configuration.properties` should look like this:

```
gatein.sso.sp.domains=sp.com,idp.com,salesforce.com,google.com
```

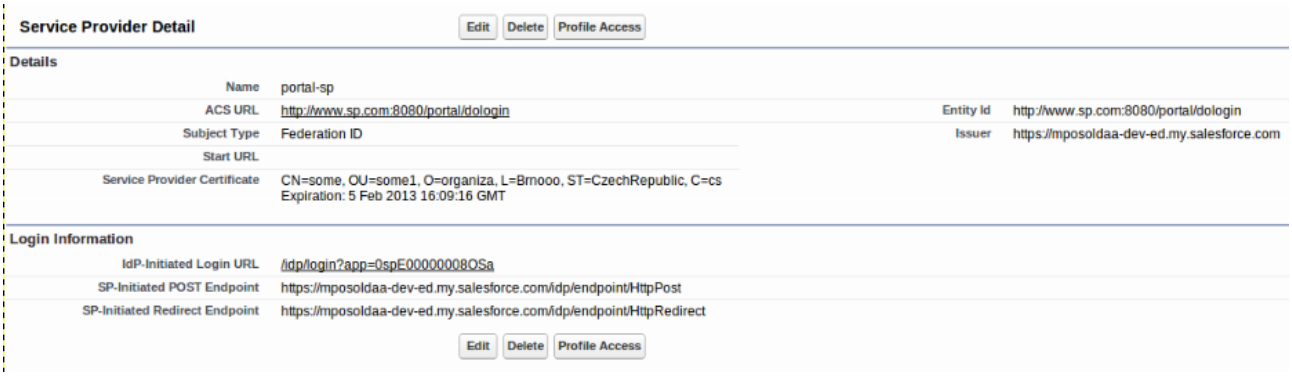
Test the integration - when you access your Google Apps domain, SAML Request will be sent to GateIn Portal on your host like http://www.idp.com:8080/portal/dologin and after login with portal username, you will be again redirected to Google Apps with SAML Response and you should be automatically logged to Google Apps thanks to SAML SSO.

Portal as SAML SP, Salesforce as SAML IDP

Again, we assume that you followed generic instructions for portal as SP in Portal as SAML2 SP and SAML2 IDP and you configured GateIn Portal on *www.sp.com* to act as SAML2 SP. Also the common steps for Salesforce integration with classic web application are again described in Picketlink guide, so here we will pay attention especially on steps specific to GateIn Portal.

Salesforce configuration

- **Disable SSO in Salesforce** - Like in generic Picketlink guide, you need to disable SSO in Salesforce, because now we don't want Salesforce to act as SP but as IDP.
- **Create Service Provider** - It is also necessary to add GateIn Portal Service Provider in configuration of Identity Provider in Salesforce. In Picketlink guide is described how to do it. Configuration for GateIn Portal SP can look similarly like this (Assuming that portal SP will be executed on virtual host www.sp.com):



- As certificate, you may need to export certificate from your GateIn Portal keystore file in `GATEIN_SP_HOME/gatein/gatein.ear/portal.war/WEB-INF/classes/saml/sso/secure` (you can simply use certificate `portal-idp.crt` if you went through previous sections of this tutorial and you're using same certificate for SP like for IDP. See sections Generate and configure your own keystore and Portal as SAML IDP, Salesforce as SAML SP for more details).

Portal configuration

- **Certificate** - Similarly like in Picketlink tutorial, we need to import certificate created by Salesforce into GateIn Portal keystore in file `GATEIN_SP_HOME/gatein/gatein.ear/portal.war/WEB-INF/classes/saml/sso/secure`. You can use command like this (assuming certificate downloaded from Salesforce is in `/tmp/salesforce_idp_cert.cer`):

```
keytool -import -file /tmp/salesforce_idp_cert.cer -keystore secure-keystore.jks -alias
salesforce-idp
```

- **URL configuration** - In **GATEIN_SP_HOME/standalone/configuration/gatein/configuration.properties** you need to change URLs to values corresponding to Salesforce:

```
gatein.sso.idp.url=https:/yourdomain.my.salesforce.com/idp/endpoint/HttpPost
gatein.sso.sp.url=http://www.sp.com:8080/portal/dologin
```

- **ValidatingAlias** needs to be added to file In **GATEIN_SP_HOME/gatein/gatein.ear/portal.war/WEB-INF/conf/sso/saml/picketlink-sp.xml** (assuming that you imported salesforce certificate into your keystore under alias *salesforce-idp*):

```
<ValidatingAlias Key="yourdomain.my.salesforce.com" Value="salesforce-idp" />
```

- **For roles-mapping**, you don't need to configure nothing on GateIn Portal because GateIn Portal SP is configured by default to obtain roles from Picketlink IDM database and not from IDP SAML Response. So no changes needed.
- **Test the integration** - after startup of GateIn Portal on your host (assuming virtual host *www.sp.com*), you can go to http://www.sp.com:8080/portal and click *Sign in* link. Then GateIn Portal will send SAML Request to Salesforce and you will be redirected to Salesforce login screen. After login into Salesforce, you will be redirected to your GateIn Portal and logged as the user which you logged in Salesforce . Again, username from GateIn Portal is mapped to Federation ID in Salesforce, so GateIn Portal user *john* is mapped to Salesforce user with Federation ID *john*)

Tips & Tricks

- **Videos** - You can follow some videos on GateIn Portal vimeo channel, where you can see SAML2 integration in action. The videos are:
  1. http://vimeo.com/45841256 - Video with basic usecases showing GateIn Portal as SAML2 SP and as SAML2 IDP.
  2. http://vimeo.com/45895919 - Video showing integration with Salesforce and Google Apps
- **SAML tracer** is cool plugin to Firefox browser, which allows you to monitor HTTP requests and see wrapped SAML messages in XML format. It could be useful especially for troubleshooting. You can download plugin here
- **Logging** - Like for other SSO solutions, it may be good to enable trace logging for `org.gatein.sso` etc but in case of SAML, it's good also to enable logging for `org.picketlink.identity.federation`, which is base package for picketlink federation library. You can add these categories to `GATEIN_HOME/standalone/configuration/standalone.xml` :

```
<logger category="org.gatein.sso">
  <level name="TRACE"/>
</logger>
<logger category="org.picketlink.federation">
  <level name="TRACE"/>
</logger>
<logger category="org.jboss.security">
  <level name="TRACE"/>
</logger>
<logger category="org.picketbox">
  <level name="TRACE"/>
</logger>
<logger category="org.exoplatform.services.security">
  <level name="TRACE"/>
</logger>
```

6.9.7 Clustered SSO setup

The JBoss SSO is useful to authenticate a user on one GateIn Portal node in a cluster and have that authentication automatically carry across to other nodes in the cluster.

This authentication can also be used in any other web applications which may require authentication, provided that these applications use same roles as the main portal instance. Attempting to use an SSO authentication in an application that uses different roles may create authorization errors (403 errors, for example).

This behavior is coming from the fact that the same JAAS principal is added by the SSO valve to all HTTP requests, even to other web applications. So the same roles are required because of it. There is alternative that you can configure SSO valve with the parameter reauthenticate="true", which will force SSO valve to perform reauthentication with saved credentials in each HTTP request against security domain of particular web application where the request is coming. This will enforce that new principal for that web application will be created with updated roles for that web application. In other words, when requireReauthentication is false (default state), you need to have same roles among web applications. When requireReauthentication is true, you need to have the same username and passwords.

Default configuration

In file `GATEIN_HOME/standalone/configuration/standalone-ha.xml` you can see that SSO is enabled by default as there is this line in configuration of JBoss web subsystem:

```
<sso cache-container="web" cache-name="sso" reauthenticate="false" />
```

This means that you don't need to do nothing if you access two GateIn Portal servers via loadbalancer as you're using same URL for access servers (which is actually URL of Apache loadbalancer). In this case you have automatic SSO when one node fail and loadbalancer needs to redirect client request to second node (sticky session).

Clustered SSO in shared DNS domain

This scenario is helpful if you're not using loadbalancer, but you directly access different GateIn Portal servers in different URLs on same DNS domain. In this example, we will use two GateIn Portal servers. First will be running on host `machine1.yourdomain.com` and second on host `machine2.yourdomain.com` .

1. If you are on Linux, you can configure the `/etc/hosts` file which contains these lines. This will allow you to test the scenario with two GateIn Portal servers on one physical machine:

```
192.168.2.7 machine1.yourdomain.com
10.11.45.78 machine2.yourdomain.com
```

2. Let's configure your domain via attribute `domain` of sso configuration in file `GATEIN_HOME/standalone/configuration/standalone-ha.xml`

```
<sso cache-container="web" cache-name="sso" reauthenticate="false" domain="yourdomain.com"
/>
```

   This needs to be configured on both servers. This configuration means that cookie *JSESSIONIDSSO* will be scoped to particular domain *yourdomain.com* (by default it's scoped only to host where authentication happened)
3. You need to configure same datasource on both servers to ensure that both servers will share same database (shared database is general requirement for cluster setup). You can look to chapter Database Configuration to see some general info about datasource setup. Note that you need to configure it in file `standalone-ha.xml` instead of `standalone.xml` because you are using clustered setup. By default, standalone-ha.xml is configured to use shared H2 database, which is intended to be used only for testing purpose.
4. Run first server with command:

```
./standalone.sh -b machine1.yourdomain.com -c standalone-ha.xml -Djboss.node.name=node1
```

5. Run second server with command:

```
./standalone.sh -b machine2.yourdomain.com -c standalone-ha.xml -Djboss.node.name=node2
```

6. Go to http://machine1.yourdomain.com:8080/portal and log in as a user john. Then go to http://machine2.yourdomain.com:8080/portal and you will see that you're logged here as john as well. Let's logout on one node and you will be logged from second node as well.

7 Web Services for Remote Portlets (WSRP)

The Web Services for Remote Portlets specification defines a web service interface for accessing and interacting with interactive presentation-oriented web services. It has been produced through the efforts of the Web Services for Remote Portlets (WSRP) OASIS Technical Committee. It is based on the requirements gathered and on the concrete proposals made to the committee.

Scenarios that motivate the WSRP functionality include:

- Content hosts, such as portal servers, providing Portlets as presentation-oriented web services that can be used by aggregation engines.
- Aggregating frameworks, including portal servers, consuming presentation-oriented web services offered by content providers and integrating them into the framework.

More information on WSRP can be found on the official website for WSRP. It is suggested that you read the primer for a good, and albeit technical overview of WSRP.

7.1 Level of support

The WSRP Technical Committee defined WSRP Use Profiles to help with the WSRP interoperability. You can refer to terms defined in that document in this section.

GateIn Portal provides a Simple level of support for our WSRP Producer except that out-of-band registration is not currently handled. We support in-band registration and persistent local state (which are defined at the Complex level).

On the Consumer side, GateIn Portal provides a Medium level of support for WSRP, except that we only handle HTML markup (as GateIn Portal itself doesn't handle other markup types). We do support explicit portlet cloning and we fully support the PortletManagement interface.

As far as caching goes, we have Level 1 Producer and Consumer. We support Cookie handling properly on the Consumer and our Producer requires initialization of cookies (as we have found that it improved interoperabilty with some consumers). We don't support custom window states or modes, as GateIn Portal doesn't either. We do, however, support CSS on both the Producer (though it's more a function of the portlets than inherent Producer capability) and Consumer.

While we provide a complete implementation of WSRP 1.0, we do need to go through the Conformance statements and perform more interoperability testing (an area that needs to be better supported by the WSRP Technical Committee and Community at large).

GateIn Portal supports WSRP 2.0 with a complete implementation of the non-optional features. The only features that we have not implemented is support for lifetimes and leasing support.

As of version 3.5 of GateIn Portal, WSRP is only activated and supported when GateIn Portal is deployed on the JBoss Application Server.

7.2 Deploying GateIn's WSRP services

GateIn Portal provides a complete support of WSRP 1.0 and 2.0 standard interfaces and offers both consumer and producer services. Starting with version 2.1.0-GA of the component, WSRP is packaged as a GateIn Portal extension and is now self-contained in an easy to install package named `$GATEIN_HOME/extensions/gatein-wsrp-integration.ear` where `$GATEIN_HOME` refers to the `gatein` directory found at the root of the JBoss AS install directory.

To use WSRP in GateIn Portal, make sure that the `gatein-wsrp-integration.ear` is located in `$GATEIN_HOME/extensions`. You shouldn't need to make any modifications to that file. If you are not going to use WSRP, it will not adversely affect your installation to leave it as-is. Otherwise, you can just remove the `gatein-wsrp-integration.ear` file from `$GATEIN_HOME/extensions`.

Configuration files for WSRP, which will be detailed in their respective sections, are found in `$JBOSS_HOME/standalone/configuration/gatein/wsrp` where `$JBOSS_HOME` is the root of your JBoss AS install directory.

7.2.1 WSRP use when running GateIn Portal on a non-default port or hostname

JBoss WS (the web service stack that GateIn Portal uses) should take care of the details of updating the port and host name used in WSDL. See the JBoss WS user guide on that subject for more details.

Of course, if you have modified the host name and port on which your server runs, you will need to update the configuration for the consumer used to consume GateIn Portal's own producer. Please refer to the Consuming remote WSRP portlets in GateIn to learn how to do so.

7.3 Securing WSRP

7.3.1 Securing WSRP

There are two main ways to secure the communication between a producer and consumer:

1. Securing the Transport Layer
   This requires using SSL and a HTTPS endpoint. By using this, the communication between the consumer and producer will be encrypted.
2. Securing the Contents of the SOAP message
   This option requires using ws-security to handle parts of the SOAP message. With this option you can specify things like encryption, signing, timestamps, etc as well as passing across user credentials to perform a login on the producer side. WS-Security is more powerful and has more options, but is requires more complex configurations.

Depending on requirements, an HTTPs endpoint or/and ws-security can be used.

7.3.2 WSRP over SSL with HTTPS endpoints

It is possible to use WSRP over SSL for a secure exchange of data. Since GateIn Portal does not come initially configured for HTTPS connectors, we will need to configure the producer's server for this first. This is a global configuration change to JBoss AS and will affect more than just GateIn Portal and WSRP. Please see the JBoss AS documentation for how to configure HTTPS connectors for the server.

Once the producer is configured for HTTPS connections, on the consumer you will just need to modify the URL for the WSRP endpoint to point to the new https based url. This will require either manually updating the value in the WSRP admin application, or by specifying it using the *wsrp-consumers-config.xml* configuration file before the server is first started.

Sample Configuration For Enabling SSL With WSRP

This is just a simple, test configuration to be used as an example as to how its possible to setup the https/ssl with wsrp. It is not meant to show best practices for configuring https with JBoss AS and does things which should not be used in a production server (such as self-signed certificates). Please see the JBoss AS documentation for full configuration options.

Configure the Producer to Use HTTPS

First we will need to configure the producer's server to use https. This is handled in the same manner that you would configure any JBoss AS server for HTTPS.

1. Generate the keystore for the producer

```
keytool -genkey -alias tomcat -keyalg RSA -keystore producerhttps.keystore -dname
"cn=localhost" -keypass changeme -storepass changeme
```

2. Configure the server to add an https connection. This requires modifying the *standalone/configuration/standalone.xml* file with the following content in bold:

```
...
<subsystem xmlns="urn:jboss:domain:web:1.1" default-virtual-server="default-host"
native="false">
   <connector name="http" protocol="HTTP/1.1" scheme="http" socket-binding="http"/>
   <connector name="https" protocol="HTTP/1.1" scheme="https" socket-binding="https"
secure="true">
      <ssl certificate-key-file="/path/to/producerhttps.keystore" password="changeme"/>
   </connector>
<virtual-server name="default-host" enable-welcome-root="true">
   <alias name="localhost"/>
   <alias name="example.com"/>
</virtual-server>
...
```

3. Start the server and verify that https://localhost:8443/portal is accessible. Note that since you are using a self-signed certificate that your browser will give a warning that the certificate cannot be trusted.

In this example case we are accessing the portal using 'localhost' hence why we are using "cn=localhost" in the keytool command. If you are using this across another domain, you will need to make the necessary change.

Configure the Consumer to Access the WSRP Endpoint over HTTPS

Ideally we should be able to just change the URL for the producer in the wsrp admin to use https, but we need to tell the consumer's server to trust our self-signed certificate first.

1. Export the producer's public key from the producer's keystore

```
keytool -export -alias tomcat -file producerkey.rsa -keystore producerhttps.keystore
-storepass changeme
```

2. Import the producer's public key into a new keystore for the consumer

```
keytool -import -alias tomcat -file producerkey.rsa -keystore consumerhttps.keystore
-storepass changeme -noprompt
```

3. Configure the *bin/standalone.conf* file to add the following line at the end of the file:

```
JAVA_OPTS="$JAVA_OPTS -Djavax.net.ssl.trustStore=/path/to/consumerhttps.keystore
-Djavax.net.ssl.trustStorePassword=changeme"
```

4. Start the consumer and change the selfv2 producer url to https://localhost:8443/wsrp-producer/v2/MarkupService?wsdl and verify that the consumer can access the producer.

It is also possible to modify the *wsrp-consumers-config.xml* configuration file to change the URL instead of modifying it in the admin gui

7.3.3 WSRP and WS-Security

Portlets may present different data or options depending on the currently authenticated user. For remote portlets, this means having to propagate the user credentials from the consumer back to the producer in a safe and secure manner. The WSRP specification does not directly specify how this should be accomplished, but delegates this work to the existing WS-Security standards. The WS-Security standards can also be used to secure the soap message, such as encryption and signing the message.

Web Container Compatibility

WSRP and WS-Security is currently only supported on GateIn Portal when running on top of JBoss AS 7.

Encryption

You will want to encrypt the credentials, and potentially the content and data, being sent between the consumer and producer, otherwise they will be sent in plain text and could be easily intercepted. You can either configure WS-Security to encrypt and sign the SOAP messages being sent, or secure the transport layer by using an https endpoint. Failure to encrypt the soap message or transport layer will result in the username and password being sent in plain text. *Use of encryption is strongly recommended.*

Credentials

When the consumer sends the user credentials to the producer, it is sending the credentials for the currently authenticated user in the consumer. This makes signing in to remote portlets transparent to end users, but also requires that the producer and consumer use the same credentials. This means that the username and password must be the same and valid on both servers.

The recommended approach for this situation would be to use a common ldap configuration. Please see the user guide on how to configure ldap for use with GateIn Portal.

Security Configuration

Introduction

JBoss AS7 uses a different web service implementation than the previous versions: it is now uses the JBossWS CXF Stack instead of the JBossWS Native Stack. Due to these changes, the way we configure WS-Security for WSRP with GateIn Portal on JBossAS 7 has changed.

We only support one ws-security configuration option for the producer. All consumers accessing the producer will have to conform to this security constraint. This means if the producer requires encryption, all consumers will be required to encrypt their messages when accessing the producer.

We only support one ws-security configuration option to be used by all the consumers. A consumer has the option to enable or disable ws-security, which allows for one or more consumers to use ws-security while the others do not.

Overview

CXF uses interceptors to extend and configure its behaviour. There are two main types of interceptors: inInterceptors and outInterceptors. InInterceptors are invoked for communication coming into the client or server, while outInterceptors are invoked when the client or server sends a message.

So for the WSRP case, the communication from the consumer to the producer is governed by the consumer's OutInterceptor and the producer's InIntereceptor. The communication from the producer to the consumer is governed by the producer's OutInterceptor and the consumer's InInterceptor. This may mean having to configure 4 Interceptors.

When dealing with WS-Security, there are some things to consider here:

1. When dealing with user propagation, only the consumer sends the user credentials to the producer. So Username Tokens only need to be configured for the consumer's OutInterceptor and the producer's InInterceptor.

1. When dealing with things like encryption, you will most likely want to encrypt the message from the consumer to the producer and also the message from the producer to the consumer. This means that encryption properties must be configured for all 4 interceptors.

Please see the CXF Documentation for more details on interceptors and their types: http://cxf.apache.org/docs/interceptors.html

To support ws-security, GateIn Portal uses CXF's WSS4J Interceptors which handle all ws-security related tasks. Please see the CXF Documentation for more details: http://cxf.apache.org/docs/ws-security.html

WSS4J Interceptors and WSRP

The WSS4J Interceptors are configured using properties, and as such can be configured using simple property files.

WSRP looks for specific property files to know whether or not in/out interceptors must be added and configured for either consumers or producer. Theses files are located in the `standalone/configuration/gatein/wsrp/cxf/ws-security` directory of your the JBoss AS 7 home directory. Consumer-specific files are in the `consumer` subdirectory while producer-specific files should be located in the `producer` subdirectory. To add and configure a WSS4J interceptor, you just need to add the proper configuration file in the proper directory. If no configuration file is found for a specific interceptor type, then no such interceptor will be added. "In" interceptors are configured using `WSS4JInInterceptor.properties` files while "out" interceptors are configured using `WSS4JOutInterceptor.properties` files.

Putting things together, here are the files you need to add to configure each interceptor types for each WSRP side:

Side	Interceptor type	Configuration file
Consumer	IN	standalone/configuration/gatein/wsrp/cxf/ws-security/consumer/WSS4JInIntercept
	OUT	standalone/configuration/gatein/wsrp/cxf/ws-security/consumer/WSS4JOutIntercep
Producer	IN	standalone/configuration/gatein/wsrp/cxf/ws-security/producer/WSS4JInIntercepto
	OUT	standalone/configuration/gatein/wsrp/cxf/ws-security/producer/WSS4JOutIntercep

Please refer to the CXF or WSS4J documentation for instructions and options available for each type of interceptors.

User Propagation

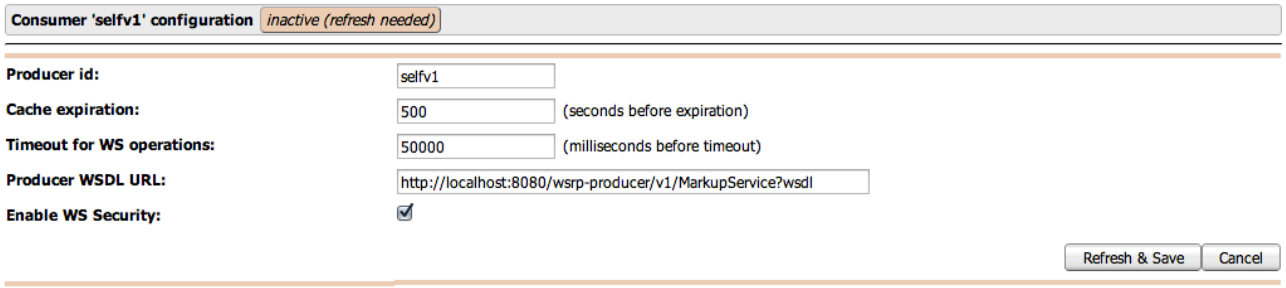
User propagation can be configured to be used over WSRP with ws-security. What this means is that a user logged into a consumer can have their credentials propagated over to the producer. This allows the producer to authenticate the user and any portlet on the producer (a remote portlet from the consumer's perspective) will view the user as being properly authenticated. This allows for remote portlets to access things like user information.

This only works if the user's credentials on the producer and consumer are the same. This may require using a common authentication mechanism, such as LDAP.

This requires some special options when configuring the producer and server.

Consumer Configuration

In order to configure ws-security on the consumer side, you will have to configure the WSS4J Interceptors as seen above. This will require having to configure the WSS4JInInterceptor and/or WSS4JOutInterceptor. You will also need to check the 'Enable WS-Security' checkbox on the WSRP Admin Portlet for the consumer configuration to take effect.



Special GateIn Portal Configuration Options for User Propagation

In order to handle user propagation in GateIn Portal across ws-security, a couple of special configuration options have been created which should be applied to the consumer's WSS4JOutInterceptor.

Custom 'user' option:

```
user=gtn.current.user
```

What this option will do is it will set the 'user' property to the currently authenticated user on the consumer.

Custom 'action' option

```
action=gtn.UsernameToken.ifCurrentUserAuthenticated
```

If a user is currently authenticated, it will replace the 'gtn.UsernameToken.ifCurrentUserAuthenticated' with 'UsernameToken'. If the current user is an unauthenticated user, 'gtn.UsernameToken.ifCurrentUserAuthenticated' will be removed from the action list. If no other actions are specified, then the WSS4J interceptor will not be added to the consumer. This allows you to only use ws-security when dealing with authenticated users, and not for anonymous users.

This requires that the user option is set to 'gtn.current.user'

Custom PasswordCallbackClass

To set the password for the username token, we need to specify the password in a callback class. See the cxf ws-security documentation for more details (http://cxf.apache.org/docs/ws-security.html).

A special callback class has already been created which handles this for you: CurrentUserPasswordCallback. This class will retrieve the currently authenticated user's password and set this as the password in the callback object.

```
passwordCallbackClass=org.gatein.wsrp.wss.cxf.consumer.CurrentUserPasswordCallback
```

Producer Configuration

The configuration of the producer is similar to that of the consumer. It also requires having to configure the WSS4JInInterceptor and/or WSS4JOutInterceptor.

Special GateIn Portal Configuration Options for User Propagation

To properly propagate user information on the producer-side, you will need to use GTNSubjectCreatingInterceptor instead of a regular WSS4JInInterceptor. This GateIn Portal specific "in" interceptor is an extension of the traditional WSS4JInInterceptor and therefore can be configured similarly and accept the same configuration properties. To specify that you want to use the GTNSubjectCreatingInterceptor, please create a property file at `standalone/configuration/gatein/wsrp/cxf/ws-security/producer/GTNSubjectCreating` instead of the regular WSS4JInInterceptor.properties file.

This Interceptor will handle the ws-security headers and retrieve the users credentials. It will then use these credentials to perform a login on the producer site, thus authenticating the user on the producer and makes the user available to remote portlets.

This class also extends org.jboss.wsf.stack.cxf.security.authentication.SubjectCreatingInterceptor and can accept the same properties this class normally accepts. See the JBossWS documentation for options and more information.

Custom 'action' option

```
action=gtn.UsernameToken.ifAvailable
```

When this option is activated, the interceptor will set the action to 'UsernameToken' when the received SOAP message contains ws-security headers. If no ws-security header is included in the message, then no action is taken and the interceptor is not run. This is useful for dealing with authenticated and unauthentcated users trying to access the producer.

Sample Configuration using the UsernameToken and User Propagation

This example configuration does not encrypt the message. This means the username and password will be sent between the producer and consumer in plain text. This is a security concern and is only being shown as a simple example. It is up to administrators to properly configure the WSS4J Interceptors to encrypt messages or to only use https communication between the producer and consumer.

Producer Setup

1. create the following file: `standalone/configuration/gatein/wsrp/cxf/ws-security/producer/GTNSubjectCrea`
2. set the content of `GTNSubjectCreatingInterceptor.properties` created in step 1 to:

```
action=gtn.UsernameToken.ifAvailable
```

3. start the producer server

Consumer Setup

1. create the following file: `standalone/configuration/gatein/wsrp/cxf/ws-security/consumer/WSS4JOutInterc`
2. set the content of the `WSS4JOutInterceptor.properties` created in step 1 to:

```
passwordType=PasswordText
user=gtn.current.user
action=gtn.UsernameToken.ifCurrentUserAuthenticated
passwordCallbackClass=org.gatein.wsrp.wss.cxf.consumer.CurrentUserPasswordCallback
```

3. start the consumer server
4. in the WSRP admin portlet, click the 'enable ws-security' checkbox
5. access a remote portlet (for example, the user identity portlet included as an example portlet in GateIn Portal) and verify that the authenticated user is the same as the one on the consumer

Sample Configuration Securing the Endpoints using Encryption and Signing

The following steps outline how to configure the producer and consumer to encrypt and sign SOAP messages passed between the producer and consumer. This example only deals with SOAP messages being sent between the producer and consumer, and not with user propagation.

Some of the configuration options specified here are based on the content at http://cxf.apache.org/docs/ws-security.html and http://www.jroller.com/gmazza/entry/cxf_x509_profile More information may be available at these sites.

Password Callback Class

WSS4J uses a Java class to specify the password when performing any security related actions. For the purpose of these encryption and signing examples, we will use the same password for the producer's and consumer's keystore (wsrpAliasPassword). This simplifies things a bit as it means we can use just one password callback class for both the producer and consumer.

Example `test.TestCallbackHandler` class:

```
package test;

import java.io.IOException;

import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;

import org.apache.ws.security.WSPasswordCallback;
import org.gatein.wsrp.wss.cxf.consumer.CurrentUserPasswordCallback;

public class TestCallbackHandler implements CallbackHandler
{

    @Override
    public void handle(Callback[] callbacks) throws IOException,
            UnsupportedCallbackException
    {

        //First check if we have any user name token call backs to add.
        //NOTE: only needed if using username tokens, and you want the currently authenticated
users password added
        CurrentUserPasswordCallback currentUserPasswordCallback = new
CurrentUserPasswordCallback();
        currentUserPasswordCallback.handle(callbacks);

        for (Callback callback: callbacks)
        {
            if (callback instanceof WSPasswordCallback)
            {
                WSPasswordCallback wsPWCallback = (WSPasswordCallback)callback;
                // since the CurrentUserPasswordCallback already handles the USERNAME_TOKEN
case, we don't want to set it in this case
                if (wsPWCallback.getUsage() != WSPasswordCallback.USERNAME_TOKEN)
                {
                    wsPWCallback.setPassword("wsrpAliasPassword");
                }
            }
        }
    }
}
```

CallbackHandler implementations are provided to GateIn Portal using the standard Java ServiceLoader infrastructure. As such, CallbackHandler implementations need to be bundled in a jar containing a file `META-INF/services/javax.security.auth.callback.CallbackHandler` specifying the fully qualified name of the CallbackHandler implementation class. This jar then needs to be put in the `gatein/extensions` directory of your GateIn Portal installation.

You can see a working example of a CallbackHandler implentation at https://github.com/gatein/gatein-wsrp/tree/master/examples/wss-callback

Configuring the Keystores

In this example we are making it a bit easier by specifying the same keystore password for both the producer and consumer, as they can use the same password callback class.

1. Generate the producer's private encryption keys

```
keytool -genkey -alias producerAlias -keypass wsrpAliasPassword -keystore producer.jks
-storepass keyStorePassword -dname "cn=producerAlias" -keyalg RSA
```

2. Export the producer's public key

```
keytool -export -alias producerAlias -file producerkey.rsa -keystore producer.jks
-storepass keyStorePassword
```

3. Generate the consumer's private encryption keys

```
keytool -genkey -alias consumerAlias -keypass wsrpAliasPassword -keystore consumer.jks
-storepass keyStorePassword -dname "cn=consumerAlias" -keyalg RSA
```

4. Export the consumer's public key

```
keytool -export -alias consumerAlias -file consumerkey.rsa -keystore consumer.jks
-storepass keyStorePassword
```

5. Import the consumer's public key into the producer's keystore

```
keytool -import -alias consumerAlias  -file consumerkey.rsa -keystore producer.jks
-storepass keyStorePassword -noprompt
```

6. Import the producer's public key into the consumer's keystore

```
keytool -import -alias producerAlias  -file producerkey.rsa -keystore consumer.jks
-storepass keyStorePassword -noprompt
```

7. Copy the `producer.jks` file to the `standalone/configuration/gatein/wsrp/cxf/ws-security/producer` directory on the producer
8. Copy the `consumer.jks` file to the `standalone/configuration/gatein/wsrp/cxf/ws-security/consumer` directory on the consumer

Configuring the Producer

1. Create `standalone/configuration/gatein/wsrp/cxf/ws-security/producer/WSS4JInterc` with the following content. This will configure the incoming message between the producer and the consumer

```
action=Signature Encrypt Timestamp
signaturePropFile=producer-security.properties
decryptionPropFile=producer-security.properties
passwordCallbackClass=test.TestCallbackHandler
```

2. Create `standalone/configuration/gatein/wsrp/cxf/ws-security/producer/WSS4JOutInter` with the following content. This will configure the outgoing message between the producer and the consumer

```
action=Signature Encrypt Timestamp
signaturePropFile=producer-security.properties
encryptionPropFile=producer-security.properties
passwordCallbackClass=test.TestCallbackHandler
user=producerAlias
encryptionUser=consumerAlias
signatureUser=producerAlias
```

3. Create `standalone/configuration/gatein/wsrp/cxf/ws-security/producer/producer-secu` with the following content:

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=keyStorePassword
org.apache.ws.security.crypto.merlin.file=producer.jks
```

4. The `passwordCallbackClass` property in these configuration files needs to match the fully qualified name of your CallbackHandler implementation class. In our case, it is `test.TestCallbackHandler`.

Configuring the Consumer

1. Create standalone/configuration/gatein/wsrp/cxf/ws-security/consumer/WSS4JOutInterceptor.prop with the following content. This will configure the outgoing message between the consumer and the producer

```
action=Signature Encrypt Timestamp
signaturePropFile=consumer-security.properties
encryptionPropFile=consumer-security.properties
passwordCallbackClass=test.TestCallbackHandler
user=consumerAlias
encryptionUser=producerAlias
signatureUser=consumerAlias
```

2. Create standalone/configuration/gatein/wsrp/cxf/ws-security/consumer/WSS4JInInterceptor.prope with the following content. This will configure the incoming message between the consumer and the producer

```
action=Signature Encrypt Timestamp
signaturePropFile=consumer-security.properties
decryptionPropFile=consumer-security.properties
passwordCallbackClass=test.TestCallbackHandler
```

3. Create standalone/configuration/gatein/wsrp/cxf/ws-security/consumer/consumer-security.properties with the following content:

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=keyStorePassword
org.apache.ws.security.crypto.merlin.file=consumer.jks
```

4. The `passwordCallbackClass` property in these configuration files needs to match the fully qualified name of your CallbackHandler implementation class. In our case, it is `test.TestCallbackHandler`.

Sample Configuration using UsernameToken, Encryption and Signing with User Propagation

The following setps outline how to configure the producer and consumer to encrypt and sign the soap message as well as use user propagation between the producer and consumer.

Configure the Producer

Follow the steps outlined in the Sample Configuration Securing the Endpoints using Encryption and Signing section but make the following changes:

1. rename the `WSS4JInInterceptor.properties` file to `GTNSubjectCreatingInterceptor.properties`
2. set the action property in `GTNSubjectCreatingInterceptor.properties` as:

```
action= gtn.UsernameToken.ifAvailable Signature Encrypt Timestamp
```

3. set the passwordType in `GTNSubjectCreatingInterceptor.properties` as:

```
passwordType=PasswordText
```

Configure the Consumer

Follow the steps outlined in the Sample Configuration Securing the Endpoints using Encryption and Signing section but make the following changes:

1. set the action property in `WSS4JOutInterceptor.properties` as:

```
action=gtn.UsernameToken.ifCurrentUserAuthenticated Signature Encrypt Timestamp
```

2. set the user in the `WSS4JOutInterceptor.properties` as:

```
user=gtn.current.user
```

3. set the passwordType in the `WSS4JOutInterceptor.properties` as:

```
passwordType=PasswordText
```

7.4 Making a portlet remotable

Only JSR-286 (Portlet 2.0) portlets can be made remotable as the mechanism to expose a portlet to WSRP relies on a JSR-286-only functionality.

GateIn does *NOT*, by default, expose local portlets for consumption by remote WSRP consumers. In order to make a portlet remotely available, it must be made "remotable" by marking it as such in the associated `portlet.xml`. This is accomplished by using a specific `org.gatein.pc.remotable` `container-runtime-option`. Setting its value to `true` makes the portlet available for the remote consumption, while setting its value to `false` will not publish it remotely. As specifying the remotable status for a portlet is optional, you do not need to do anything if you do not need your portlet to be available remotely.

```
<?xml version="1.0" standalone="yes"?>
<portlet-app xmlns="http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd
http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd"
             version="2.0">
<portlet-app>
   <portlet>
      <portlet-name>BasicPortlet</portlet-name>

      ...

      <container-runtime-option>
         <name>org.gatein.pc.remotable</name>
         <value>true</value>
      </container-runtime-option>
   </portlet>
</portlet-app>
```

It is also possible to specify that all the portlets declared within a given portlet application to be remotable by default. This is done by specifying `container-runtime-option` at the `portlet-app` element level. Individual portlets can override that value to not be remotely exposed.

```
<?xml version="1.0" standalone="yes"?>
<portlet-app xmlns="http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd
http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd"
             version="2.0">
<portlet-app>

   <portlet>
      <portlet-name>RemotelyExposedPortlet</portlet-name>
      ...
   </portlet>
   <portlet>
      <portlet-name>NotRemotelyExposedPortlet</portlet-name>
      ...
      <container-runtime-option>
         <name>org.gatein.pc.remotable</name>
         <value>false</value>
      </container-runtime-option>
   </portlet>

   <container-runtime-option>
      <name>org.gatein.pc.remotable</name>
      <value>true</value>
   </container-runtime-option>
</portlet-app>
```

In the example above, two portlets are defined. The `org.gatein.pc.remotable container-runtime-option` is set to `true` at the `portlet-app` level, meaning that all portlets defined in this particular portlet application are exposed remotely by GateIn's WSRP producer. Note, however, that it is possible to override the default behavior: specifying a value for the `org.gatein.pc.remotable container-runtime-option` at the `portlet` level will take precedence over the default. In the example above, `RemotelyExposedPortlet` inherits the remotable status defined at the `portlet-app` level since it does not specify a value for the `org.gatein.pc.remotable container-runtime-option`. `NotRemotelyExposedPortlet`, however, overrides the default behavior and is not remotely exposed. Note that in the absence of a top-level `org.gatein.pc.remotable container-runtime-option` value set to `true`, portlets are NOT remotely exposed.

7.5 Consuming GateIn's WSRP portlets from a remote Consumer

WSRP Producers vary a lot as far as how they are configured. Most of them require you to specify the URL for the Producer's WSDL definition. Please refer to the remote producer's documentation for specific instructions. For instructions on how to do so in GateIn, refer to Consuming remote WSRP portlets in GateIn.

GateIn's Producer is automatically set up when you deploy a portal instance with the WSRP service. You can access the WSDL file at `http://<hostname>:<port>/wsrp-producer/v2/MarkupService?wsdl`. If you wish to use only the WSRP 1 compliant version of the producer, please use the WSDL file found at `http://<hostname>:<port>/wsrp-producer/v1/MarkupService?wsdl`. The default hostname is `localhost` and the default port is `8080`.

7.6 Consuming remote WSRP portlets in GateIn

To consume the WSRP portlets exposed by a remote producer, GateIn's WSRP consumer needs to know how to access that remote producer. One can configure access to a remote producer using the provided configuration portlet. Alternatively, it is also possible to configure access to remote producers using an XML descriptor, though it is recommended (and easier) to do so via the configuration portlet.

Once a remote producer has been configured, the portlets that it exposes are then available in the Application Registry to be added to categories and then to pages.

As a way to test the WSRP producer service and to check that the portlets that you want to expose remotely are correctly published via WSRP, two default consumers `selfv1` and `selfv2` have been configured to consume the portlets exposed by GateIn's WSRP 1 and WSRP 2 producer, respectively.

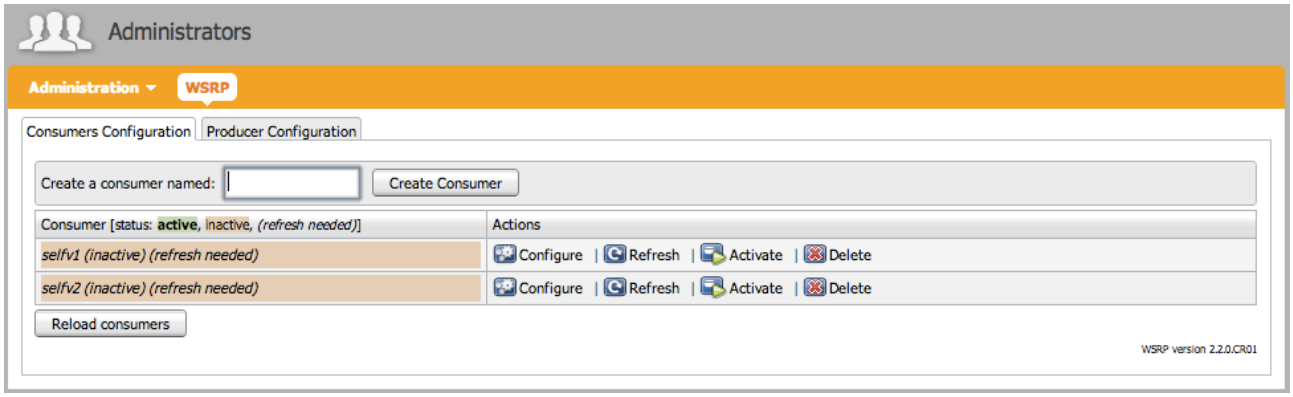
7.6.1 Configuring a remote producer using the configuration portlet

Let's work through the steps of defining access to a remote producer using the configuration portlet so that its portlets can be consumed within GateIn. We will configure access to NetUnity's public WSRP producer.

Some WSRP producers do not support chunked encoding that is activated by default by JBoss WS. If your producer does not support chunked encoding, your consumer will not be able to properly connect to the producer. This will manifest itself with the following error: `Caused by: org.jboss.ws.WSException: Invalid HTTP server response [503] - Service Unavailable`. Please see this GateIn's wiki page for more details.
/!\ Check if this is still needed

GateIn provides a portlet to configure access (among other functions) to remote WSRP Producers graphically. Starting with 3.2, the WSRP configuration portlet is installed by default. You can find it at http://localhost:8080/portal/login?initialURI=%2Fportal%2Fprivate%2Fclassic%2FwsrpConfigurationp&userna

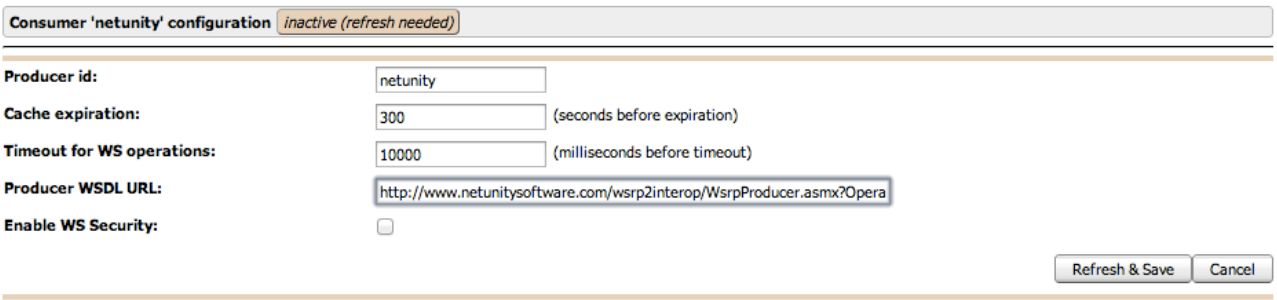
You should see a screen similar to:



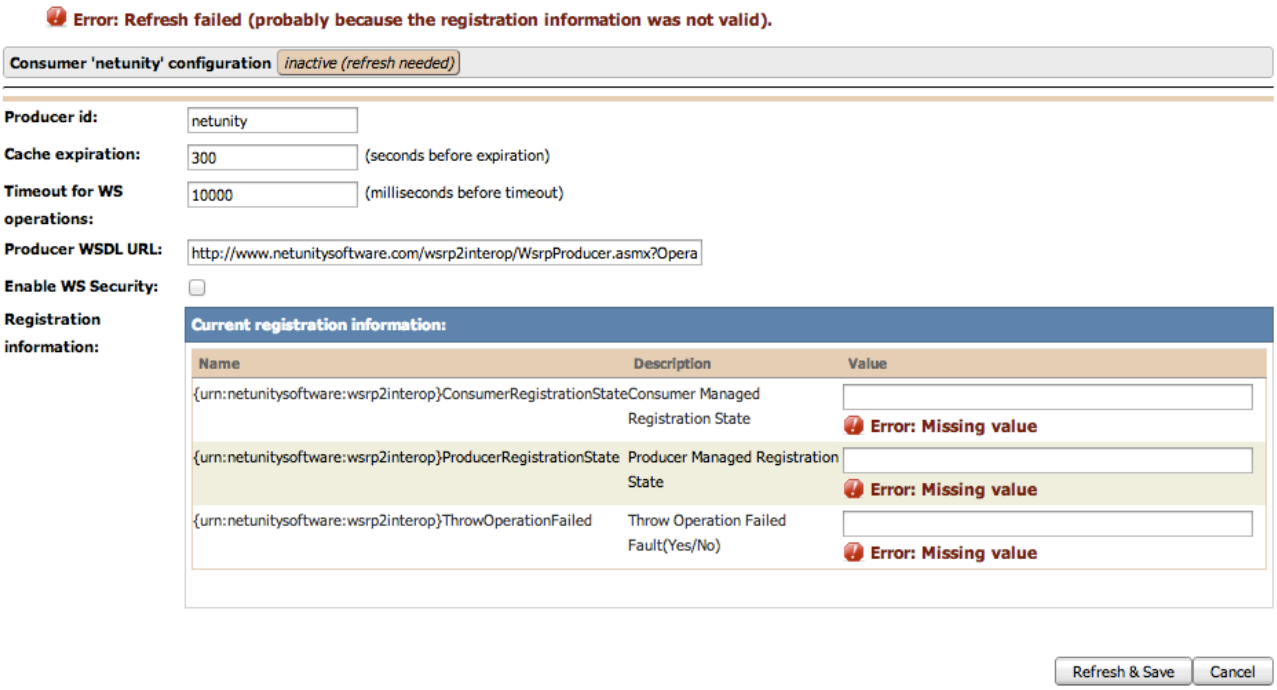
This screen presents all the configured Consumers associated with their status and possible actions on them. A Consumer can be active or inactive. Activating a Consumer means that it is ready to act as a portlet provider. Note also that a Consumer can be marked as requiring refresh meaning that the information held about it might not be up to date and refreshing it from the remote Producer might be a good idea. This can happen for several reasons: the service description for that remote Producer has not been fetched yet, the cached version has expired or modifications have been made to the configuration that could potentially invalidate it, thus requiring re-validation of the information.

Next, we create a new Consumer which we will call `netunity`. Type "`netunity`" in the "Create a consumer named:" field then click on "Create consumer":

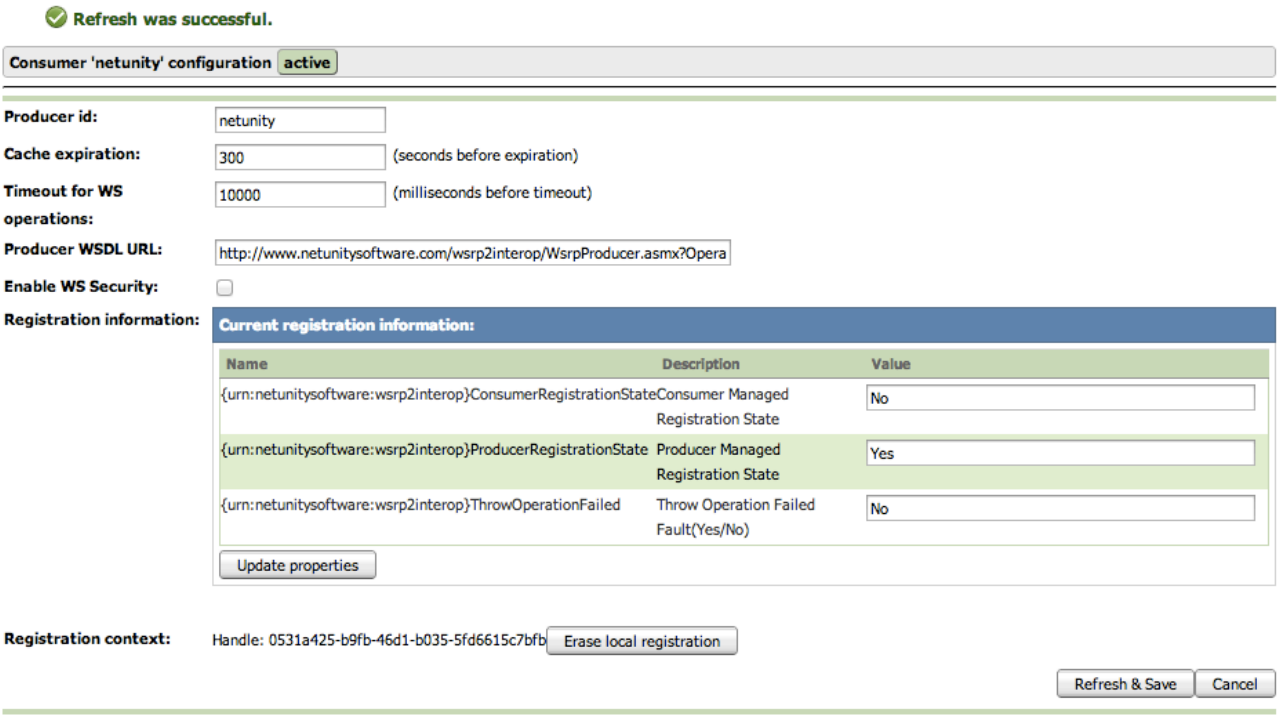
You should now see a form allowing you to enter/modify the information about the Consumer. Set the cache expiration value to 300 seconds, leave the default timeout value for web services (WS) operations and enter the WSDL URL `http://www.netunitysoftware.com/wsrp2interop/WsrpProducer.asmx?Operation=WSDL&Ws` for the producer in the text field and press the "Refresh & Save" button:



This will retrieve the service description associated with the Producer which WSRP interface is described by the WSDL file found at the URL you just entered. In our case, querying the service description will allow us to learn that the Producer requires registration, requested three registration properties and that we are missing values for these properties:

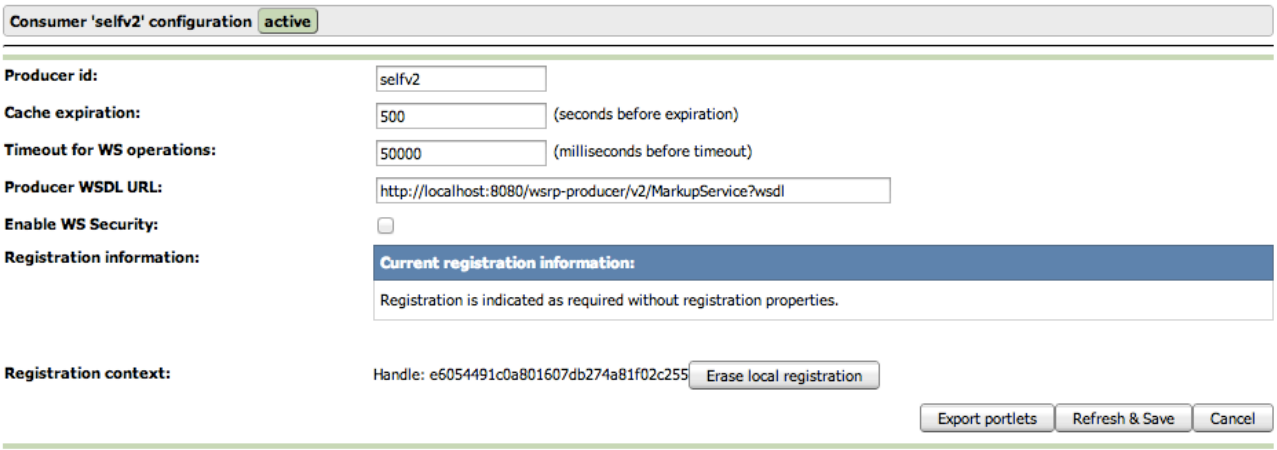


This particular producer requests simple `Yes` or `No` values for the three registration properties. Entering `No`, `Yes` and `No` (in that order) for the values and then pressing the "Refresh & Save" button should result in:



At this point, there is no automated way to learn about which possible values (if any) are expected by the remote Producer. Sometimes, the possible values will be indicated in the registration property description but this is not always the case. Please refer to the specific Producer's documentation for more details.

If we had been dealing with a producer which required registration but didn't require any registration properties, as is the case for the `selfv2` consumer (the consumer that accesses the portlets made remotely available by GateIn's producer via WSRP 2), we'd have seen something similar to the screenshot below, after pressing the "Refresh & Save" button:



7.6.2 Configuring access to remote producers via XML

While we recommend you use the WSRP Configuration portlet to configure Consumers, we provide an alternative way to configure consumers by adding an XML file called `wsrp-consumers-config.xml` in the `$JBOSS_HOME/standalone/configuration/gatein/wsrp` directory.

An XML Schema defining which elements are available to configure Consumers via XML can be found in `$JBOSS_HOME/modules/org/gatein/wsrp/main/wsrp-integration-api-$WSRP_VERSION`

It is important to note that once the XML configuration file for consumers has been read upon the WSRP service first start, the associated information is put under control of JCR (Java Content Repository). Subsequent launches of the WSRP service will use the JCR-stored information and ignore the content of the XML configuration file.

Required configuration information

Let's now look at which information needs to be provided to configure access to a remote producer.

First, we need to provide an identifier for the producer we are configuring so that we can refer to it afterwards. This is accomplished via the mandatory `id` attribute of the `<wsrp-producer>` element.

GateIn also needs to learn about the remote producer's endpoints to be able to connect to the remote web services and perform WSRP invocations. This is accomplished by specifying the URL for the WSDL description for the remote WSRP service, using the `<endpoint-wsdl-url>` element.

Both the `id` attribute and `<endpoint-wsdl-url>` elements are required for a functional remote producer configuration.

Optional configuration

It is also possible to provide additional configuration, which, in some cases, might be important to establish a proper connection to the remote producer.

One such optional configuration concerns caching. To prevent useless roundtrips between the local consumer and the remote producer, it is possible to cache some of the information sent by the producer (such as the list of offered portlets) for a given duration. The rate at which the information is refreshed is defined by the `expiration-cache` attribute of the `<wsrp-producer>` element which specifies the refreshing period in seconds. For example, providing a value of 120 for expiration-cache means that the producer information will not be refreshed for 2 minutes after it has been somehow accessed. If no value is provided, GateIn will always access the remote producer regardless of whether the remote information has changed or not. Since, in most instances, the information provided by the producer does not change often, we recommend that you use this caching facility to minimize bandwidth usage.

It is also possible to define a timeout after which WS operations are considered as failed. This is helpful to avoid blocking the WSRP service, waiting forever on the service that doesn't answer. Use the `ws-timeout` attribute of the `<wsrp-producer>` element to specify how many milliseconds the WSRP service will wait for a response from the remote producer before timing out and giving up.

To use WS-Security-secured interactions with the producer, you will need to set the `use-wss` attribute of the `<wsrp-producer>` element to `true`. This attribute is optional and defaults to `false` if no value is provided. For more details on how to use WS-Security, please refer to Securing WSRP.

Additionally, some producers require consumers to register with them before authorizing them to access their offered portlets. If you know that information beforehand, you can provide the required registration information in the producer configuration so that the consumer can register with the remote producer when required.

At this time, though, only simple String properties are supported and it is not possible to configure complex registration data. This should, however, be sufficient for most cases.

Registration configuration is done via the `<registration-data>` element. Since GateIn can generate the mandatory information for you, if the remote producer does not require any registration properties, you only need to provide an empty `<registration-data>` element. Values for the registration properties required by the remote producer can be provided via `<property>` elements. See the example below for more details. Additionally, you can override the default consumer name automatically provided by GateIn via the `<consumer-name>` element. If you choose to provide a consumer name, please remember that this should uniquely identify your consumer.

Examples

Here is the configuration of the `selfv1` and `selfv2` consumers as found in:

- {{$JBOSS_PROFILE_HOME/deploy/gatein-wsrp-integration.ear/lib/extension-component-$WSRP_VE a cache expiring every 500 seconds and with a 50 second timeout for web service operations.

> ⚠ This file contains the default configuration and you shouldn't need to edit it. If you want to make modifications to it, we recommend that you follow the procedure detailed in consumer_gui.

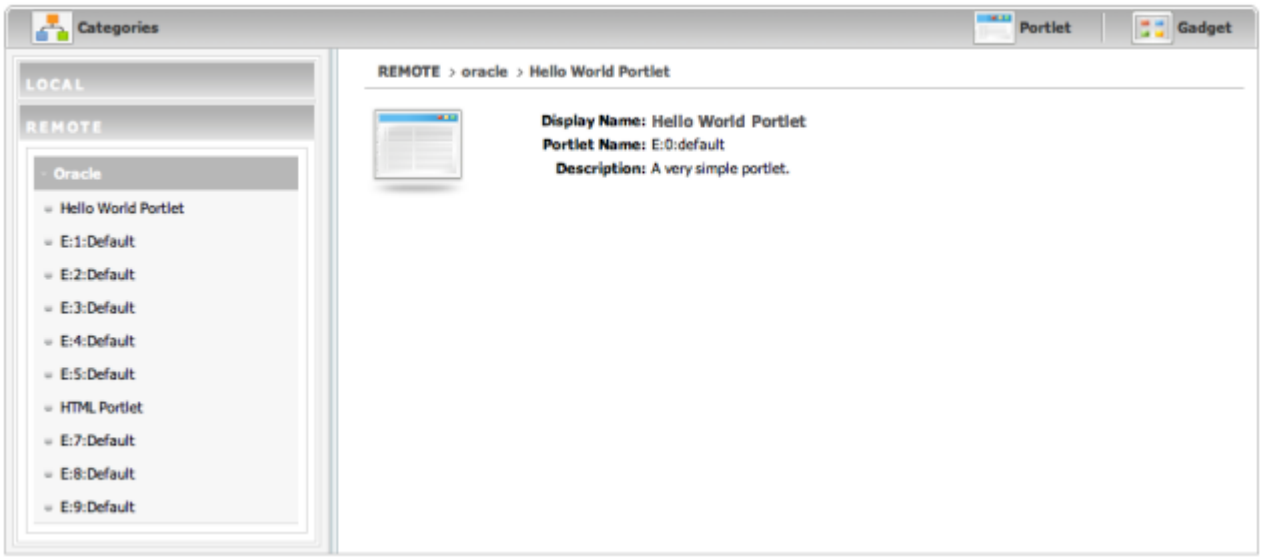
```
<?xml version='1.0' encoding='UTF-8' ?>
<deployments xmlns="http://www.gatein.org/xml/ns/gatein_wsrp_consumer_1_0"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://www.gatein.org/xml/ns/gatein_wsrp_consumer_1_0
http://www.jboss.org/portal/xsd/gatein_wsrp_consumer_1_0.xsd">
   <deployment>
      <wsrp-producer id="selfv1" expiration-cache="500" ws-timeout="50000">

<endpoint-wsdl-url>http://localhost:8080/wsrp-producer/v1/MarkupService?wsdl</endpoint-wsdl-url>
         <registration-data/>
      </wsrp-producer>
   </deployment>
   <deployment>
      <wsrp-producer id="selfv2" expiration-cache="500" ws-timeout="50000">

<endpoint-wsdl-url>http://localhost:8080/wsrp-producer/v2/MarkupService?wsdl</endpoint-wsdl-url>
         <registration-data/>
      </wsrp-producer>
   </deployment>
</deployments>
```

Here is an example of a WSRP descriptor with registration data and cache expiring every minute:

```
<?xml version='1.0' encoding='UTF-8' ?>
<deployments xmlns="http://www.gatein.org/xml/ns/gatein_wsrp_consumer_1_0"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://www.gatein.org/xml/ns/gatein_wsrp_consumer_1_0
http://www.jboss.org/portal/xsd/gatein_wsrp_consumer_1_0.xsd">
<deployments>
   <deployment>
      <wsrp-producer id="AnotherProducer" expiration-cache="60">
         <endpoint-wsdl-url>http://example.com/producer/producer?WSDL</endpoint-wsdl-url>
         <registration-data>
            <property>
               <name>property name</name>
               <lang>en</lang>
               <value>property value</value>
            </property>
         </registration-data>
      </wsrp-producer>
   </deployment>
</deployments>
```

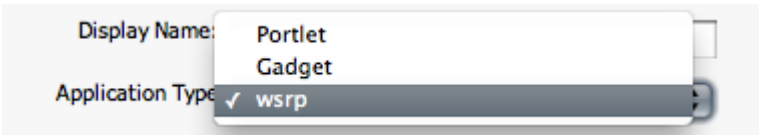
7.6.3 Adding remote portlets to categories

If we go to the Application Registry and examine the available portlets by clicking on the Portlet link, you will now be able to see remote portlets if you click on the REMOTE tab in the left column:



These portlets are available to be used. For example, regular portlets can be used in categories and added to pages. If you use the Import Applications functionality, they will also be automatically imported in categories based on the keywords they define.

More specifically, if you want to add a WSRP portlet to a category, you can access these portlets by selecting `wsrp` in the Application Type drop-down menu:



7.6.4 Adding remote portlets to pages

Since remote portlets can be manipulated just like regular portlets, you can add them to pages just like you would do for a regular portlet. Please refer to the appropriate section of the documentation for how to do so.

However, it is possible to also add a remote portlet to a `pages.xml` configuration file. This is accomplished using the `<wsrp>` element instead of the `<portlet>` element in your `pages.xml` document. While `<portlet>` references a local portlet using the name of the application in which the portlet is contained and the portlet name itself to identify which portlet to use, `<wsrp>` references a remote portlet using a combination of the consumer identifier for the producer publishing the portlet and the portlet handle identifying the portlet within the context of the producer. For more details on `pages.xml`, please refer to Portal Navigation Configuration.

The format for such a reference to a remote portlet is as follows: first, the identifier of the consumer that accesses the remote producer publishing the remote portlet, then a separator (currently a period (`.`)) and finally the portlet handle for that portlet, which is a string provided by the producer to identify the portlet.

Since there currently is no easy way to determine the correct portlet handle, we recommend that you use the graphical user interface to add remote portlets to pages instead of using `pages.xml`.

For remote portlets published by GateIn's WSRP producer, the portlet handles are, at the time of this writing, following the `/<portlet application name>.<portlet name>` format.

Example

In the following example, we define 2 portlets for a page named `Test` in the `pages.xml` configuration file. They are actually references to the same portlet, albeit one accessed locally and the other one accessing it via the `selfv2` consumer which consumes GateIn's WSRP producer. You can see that the first one is local (the `<portlet-application>` with the '`Added locally`' title) and follows the usual declaration. The second portlet (the one with the '`Added from selfv2 consumer`' title) comes from the `selfv2` consumer and uses the `<wsrp>` element instead of `<portlet>` element and follows the format for portlets coming from the GateIn's WSRP producer.

```
<page>
   <name>Test</name>

   ...

   <portlet-application>
      <portlet>
         <application-ref>richFacesPortlet</application-ref>
         <portlet-ref>richFacesPortlet</portlet-ref>
      </portlet>
      <title>Added locally</title>

      ...

   </portlet-application>

   <portlet-application>
      <wsrp>selfv2./richFacesPortlet.richFacesPortlet</wsrp>
      <title>Added from selfv2 consumer</title>

      ...

   </portlet-application>
</page>
```

7.7 Consumers maintenance

7.7.1 Modifying a currently held registration

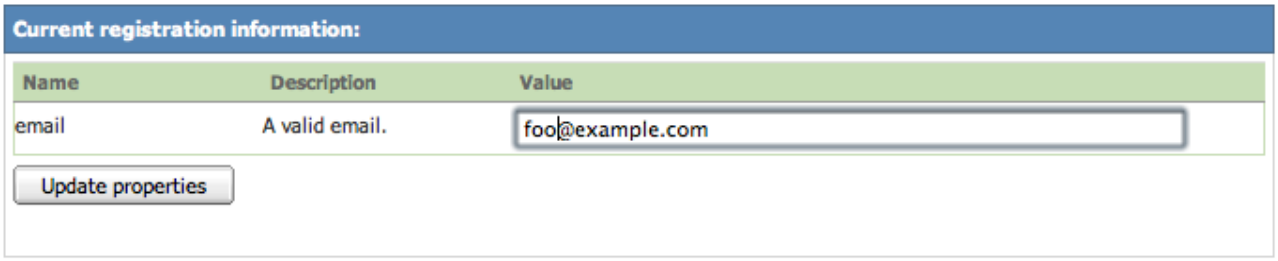
Registration modification for service upgrade

Producers often offer several levels of service depending on consumers' subscription levels (for example). This is implemented at the WSRP level with the registration concept: producers can assert which level of service to provide to consumers based on the values of given registration properties.

There might also be cases where you just want to update the registration information because it has changed. For example, the producer required you to provide a valid email and the previously email address is not valid anymore and needs to be updated.

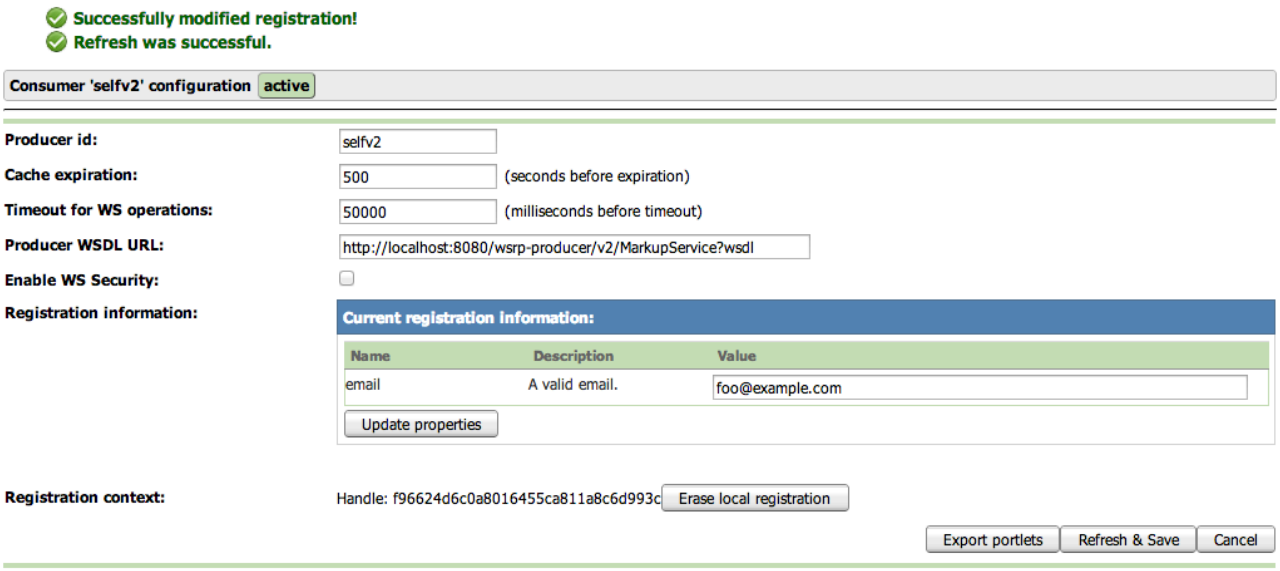
It is therefore sometimes necessary to modify the registration that concretizes the service agreement between a consumer and a producer. Let's take the example of a producer requiring a valid email (via an `email` registration property) as part of its required information that consumers need to provide to be properly registered.

Suppose now that we would like to update the email address that we provided to the remote producer when we first registered. We will need to tell the producer that our registration data has been modified. Let's see how to do this. Select the consumer for the remote producer in the available consumers list to display its configuration. Assuming you want to change the email you registered with to `foo@example.com`, change its value in the field for the `email` registration property:



Now click on "Update properties" to save the change. A "Modify registration" button should now appear to let you send this new data to the remote producer:

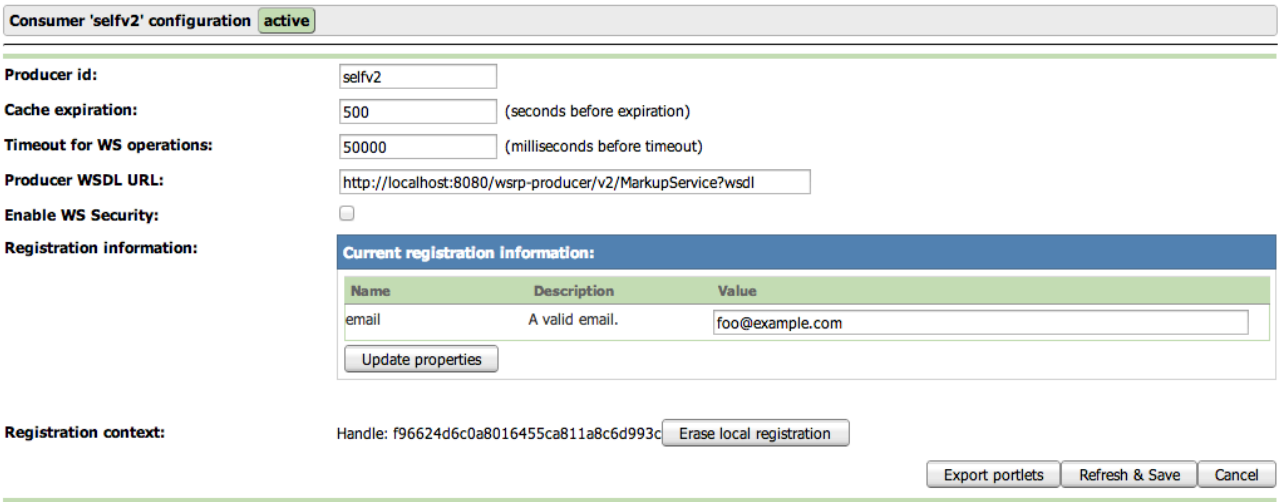
Click on this new button and, if everything went well and your updated registration has been accepted by the remote producer, you should see something similar to:



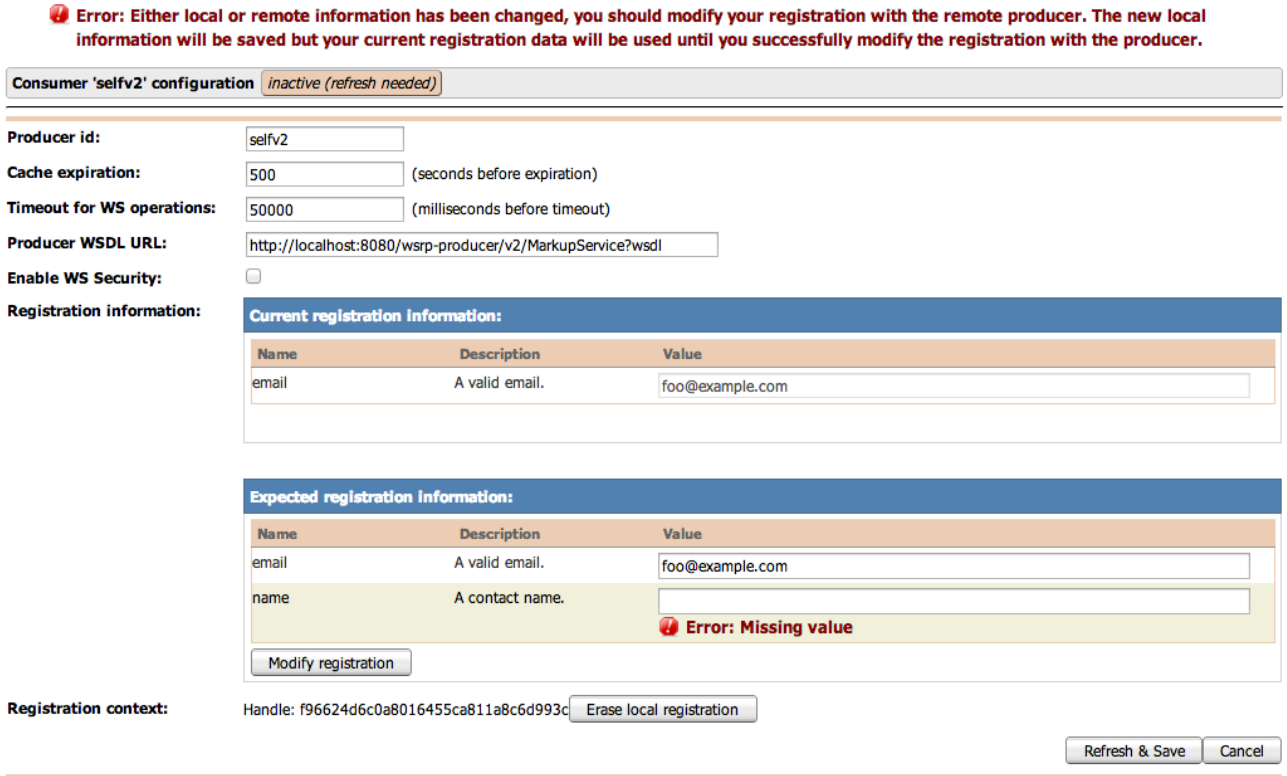
Registration modification on producer error

It can also happen that a producer administrator decided to change its requirement for registered consumers. GateIn will attempt to help you in this situation. Let's walk through an example using the `selfv2` consumer.

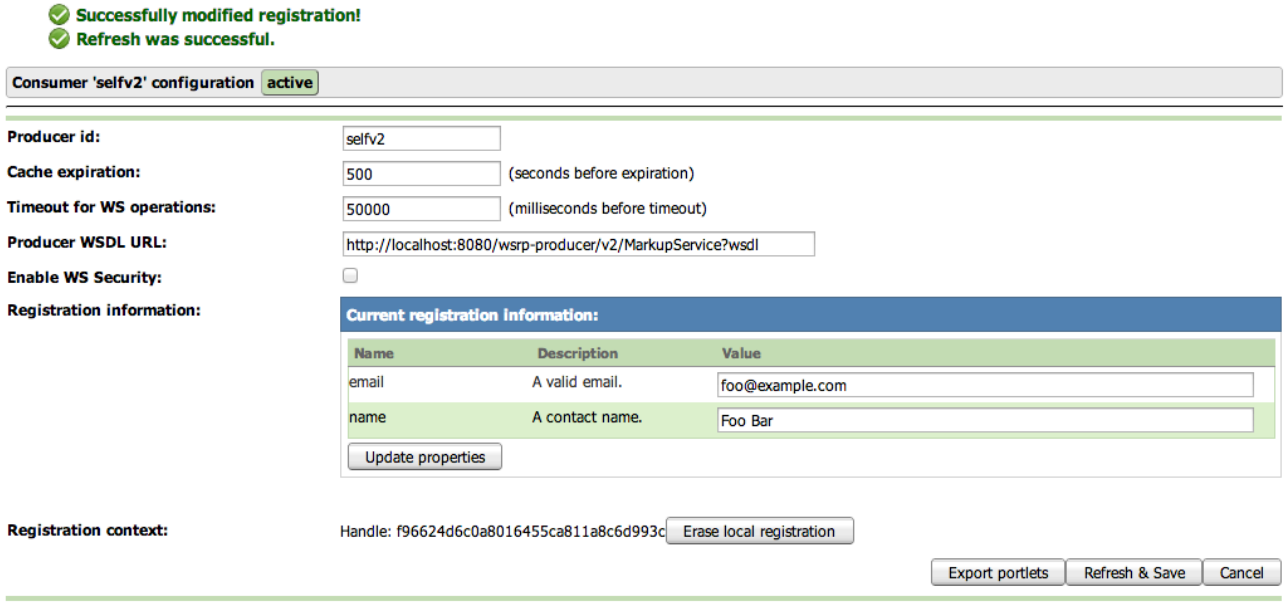
Let's assume that registration is requiring a valid value for an `email` registration property. If you go to the configuration screen for this consumer, you should see:



Now suppose that the administrator of the producer now additionaly requires a value to be provided for a `name` registration property. We will actually see how to do perform this operation in GateIn when we examine how to configure GateIn's producer in Configuring GateIn's WSRP Producer. Operations with this producer will now fail. If you suspect that a registration modification is required, you should go to the configuration screen for this remote producer and refresh the information held by the consumer by pressing "Refresh & Save":



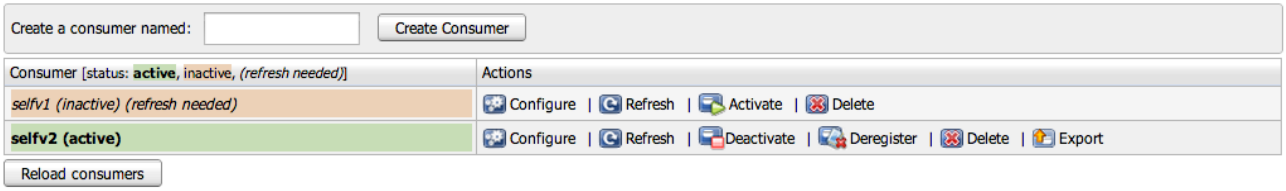
As you can see, the configuration screen now shows the currently held registration information and the expected information from the producer. Enter a value for the `name` property and then click on "Modify registration". If all went well and the producer accepted your new registration data, you should see something similar to:



WSRP 1 makes it rather difficult to ascertain for sure what caused an OperationFailedFault as it is the generic exception returned by producers if something didn't quite happen as expected during a method invocation. This means that OperationFailedFault can be caused by several different reasons, one of them being a request to modify the registration data. Please take a look at the log files to see if you can gather more information as to what happened. WSRP 2 introduces an exception that is specific to a request to modify registrations thus reducing the ambiguity that exists when using WSRP 1.

7.7.2 Consumer operations

Several operations are available from the consumer list view of the WSRP configuration portlet:



The available operations are:

- Configure: displays the consumer details and allows user to edit them
- Refresh: forces the consumer to retrieve the service description from the remote producer to refresh the local information (offered portlets, registration information, etc.)
- Activate/Deactivate: activates/deactivates a consumer, governing whether it will be available to provide portlets and receive portlet invocations
- Register/Deregister: registers/deregisters a consumer based on whether registration is required and/or acquired
- Delete: destroys the consumer, after deregistering it if it was registered
- Export: exports some or all of the consumer's portlets to be able to later import them in a different context
- Import: imports some or all of previously exported portlets

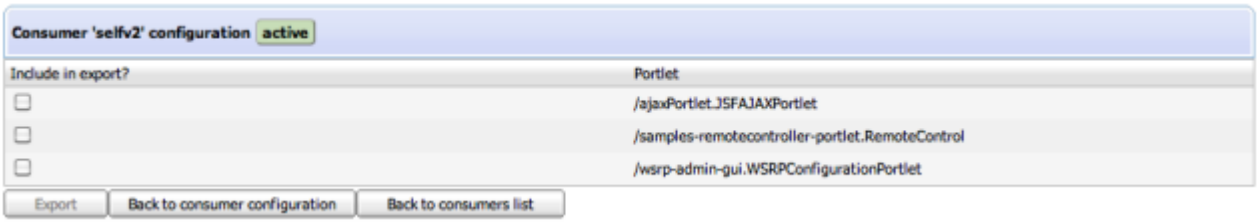
Import/Export functionality is only available to WSRP 2 consumers of producers that support this optional functionality. Import functionality is only available if portlets have previously been exported.

7.7.3 Importing and exporting portlets

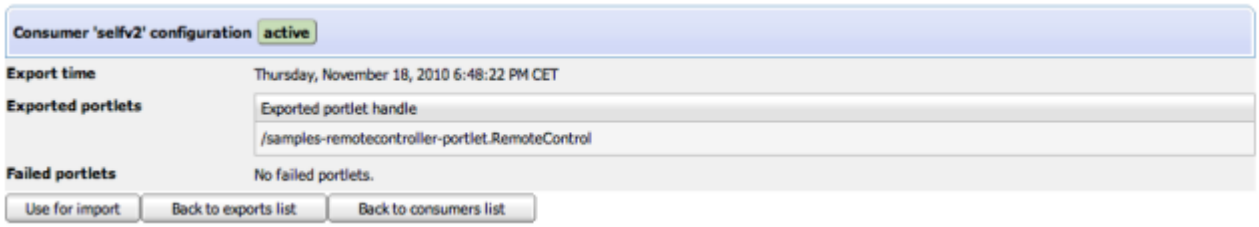
Import and export are new functionalities added in WSRP 2. Exporting a portlet allows a consumer to get an opaque representation of the portlet which can then be use by the corresponding import operation to reconstitute it. It is mostly used in migration scenarios during batch operations. Since GateIn does not currently support automated migration of portal data, the functionality that we provide as part of WSRP 2 is necessarily less complete than it could be with full portal support.

The import/export implementation in GateIn (available since 3.1) allows users to export portlets from a given consumer. These portlets can then be used to replace existing content on pages. This is accomplished by assigning previously exported portlets to replace the content displayed by windows on the portal's pages. Let us walk through an example to make things clearer.

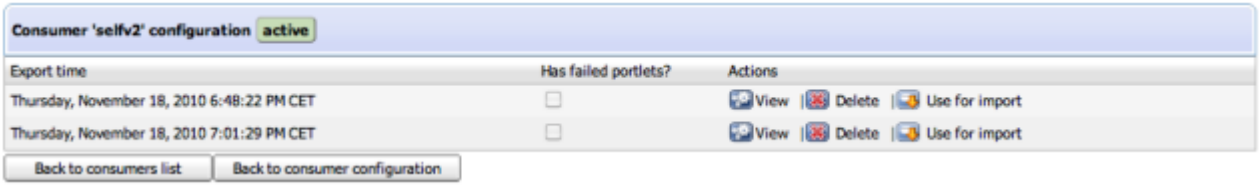
Clicking on the "Export" action for a given consumer will display the list of portlets currently made available by this specific consumer. An example of such a list is shown below:



Once portlets have been selected, they can be exported by clicking on the "Export" button thus making them available for later import:



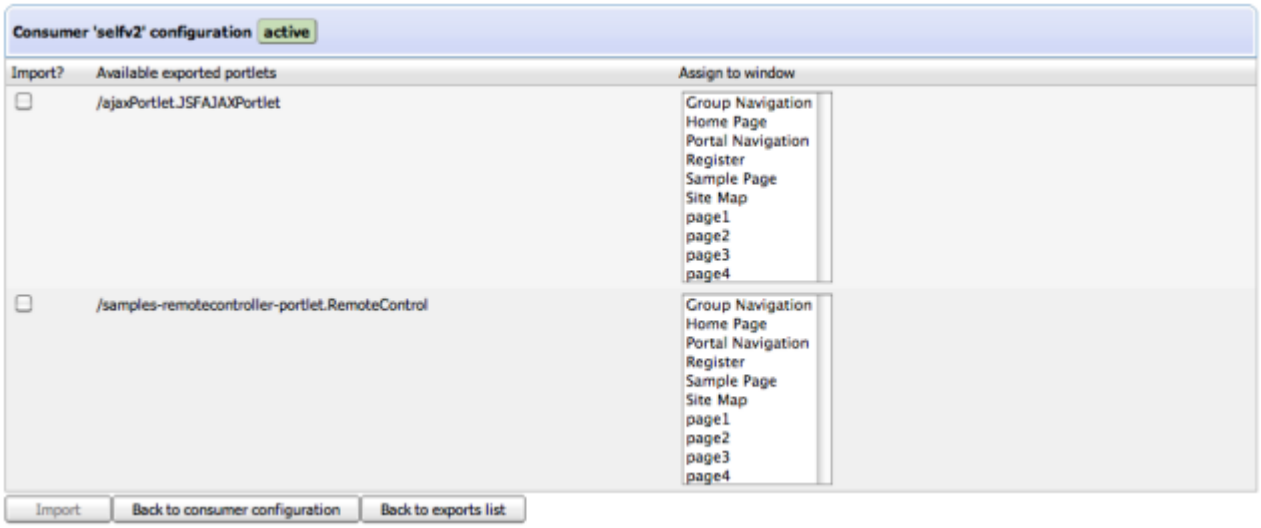
You can re-import the portlets directly by pressing the "Use for import" button or, on the Consumers list page, using the "Import" action for a given consumer. Let's assume that you used that second option and that you currently have several available sets of previously exported portlets to import from. After clicking the action link, you should see a screen similar to the one below:



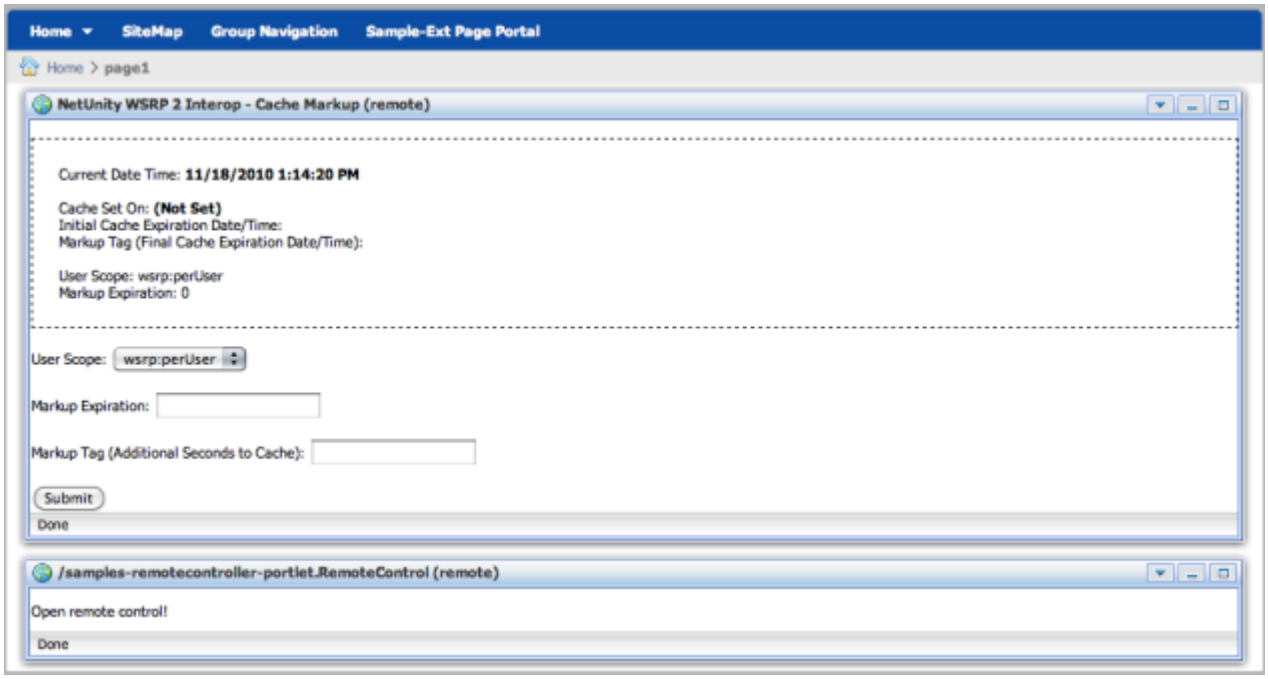
As you can see this screen presents the list of available exports with available operations for each.

- View: displays the export details as previously seen when the export was first performed
- Delete: deletes the selected export, asking you for confirmation first
- Use for import: selects the export to import portlets from

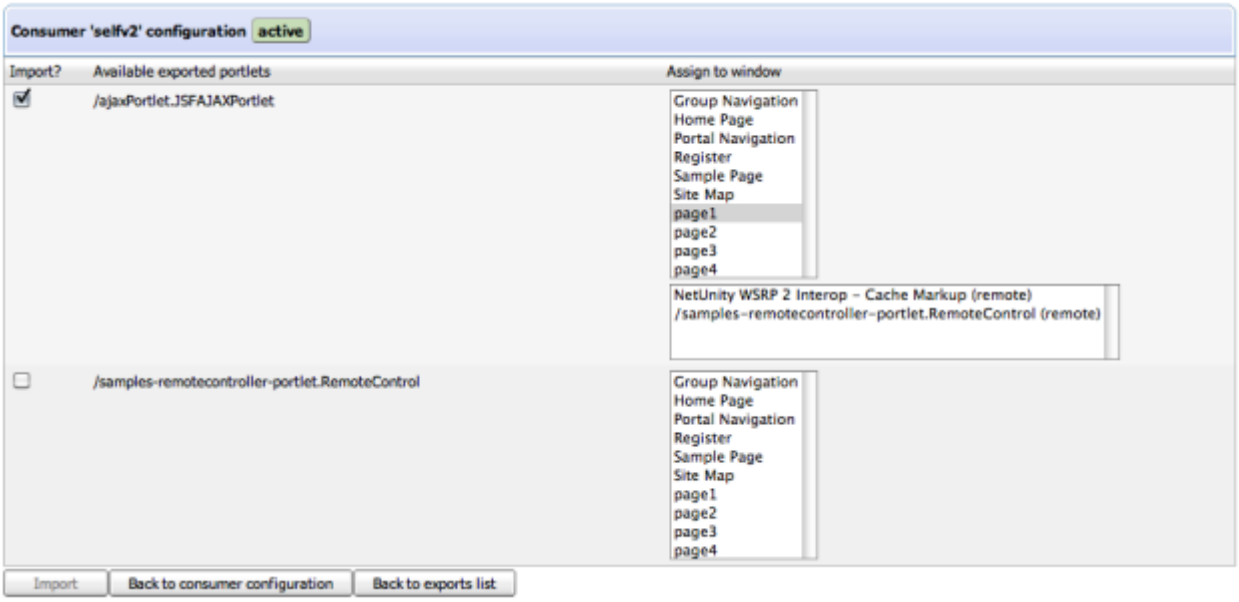
Once you've selected an export to import from, you will see a screen similar to the one below:



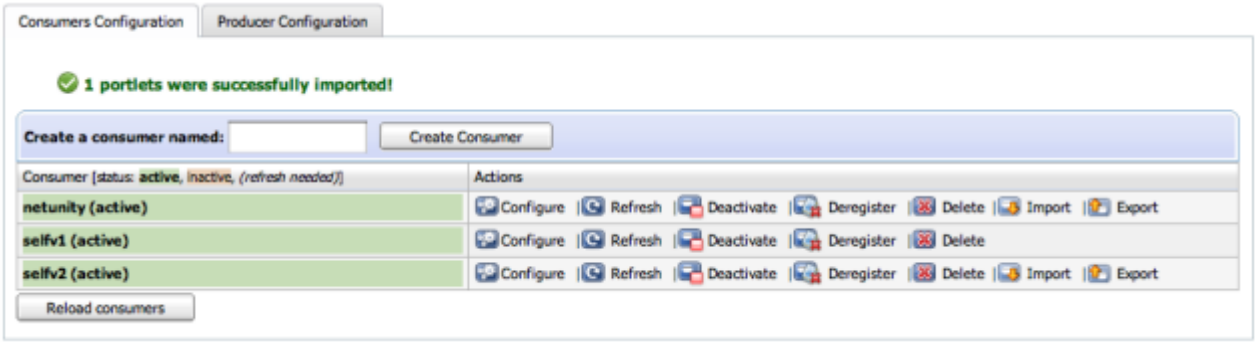
The screen displays the list of available exported portlets for the previously selected export. You can select which portlet you want to import by checking the checkbox next to its name. Next, you need to select the content of which window the imported portlet will replace. This process is done in three steps. Let's assume in this example that you have the following page called `page1` and containing two windows called `NetUnity WSRP 2 Interop - Cache Markup (remote)` and `/samples-remotecontroller-portlet.RemoteControl (remote)` as shown below:



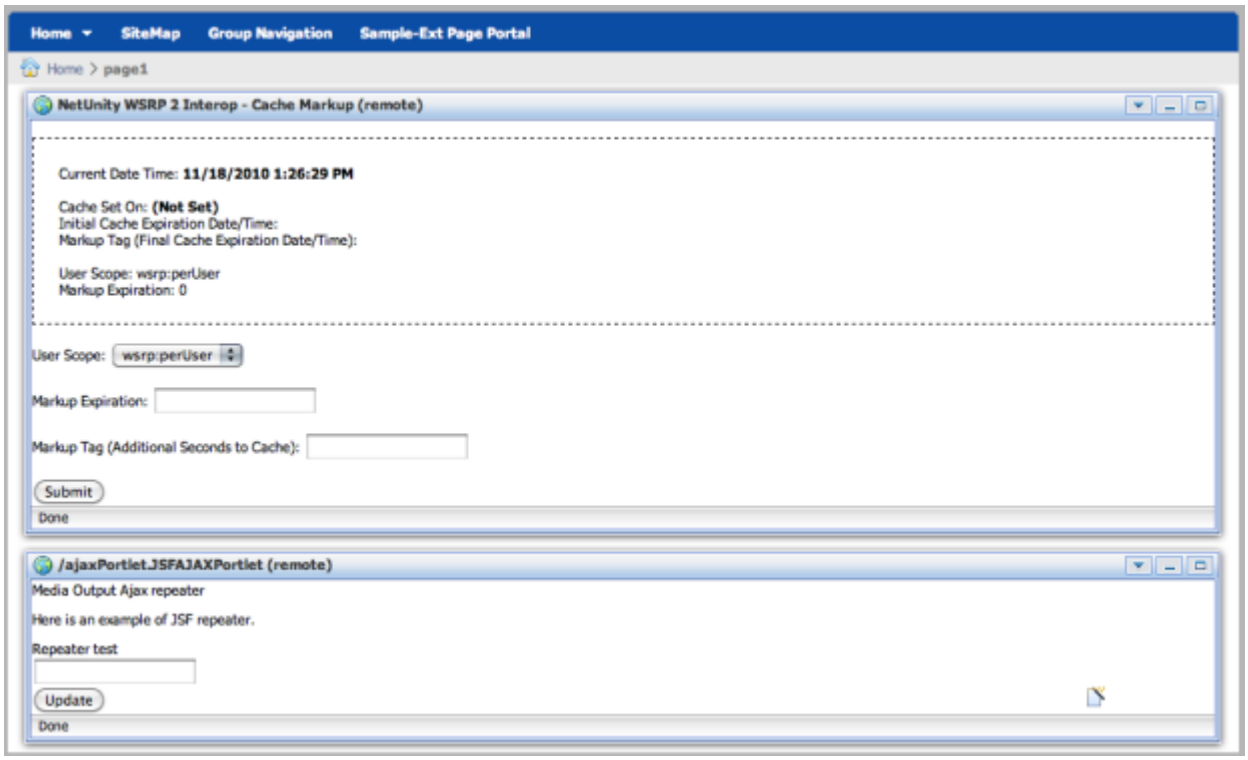
In this example, we want to replace the content of the `/samples-remotecontroller-portlet.RemoteControl (remote)` by the content of the `/ajaxPortlet.JSFAJAXPortlet` portlet that we previously exported. To do so, we will check the checkbox next to the `/ajaxPortlet.JSFAJAXPortlet` portlet name to indicate that we want to import its data and then select the `page1` in the list of available pages. The screen will then refresh to display the list of available windows on that page, similar to the one seen below:



Note that, at this point, we still need to select the window which content we want to replace before being able to complete the import operation. Let's select the `/samples-remotecontroller-portlet.RemoteControl (remote)` window, at which point the "Import" button will become enabled, indicating that we now have all the necessary data to perform the import. If all goes well, pressing that button should result in a screen similar to the one below:

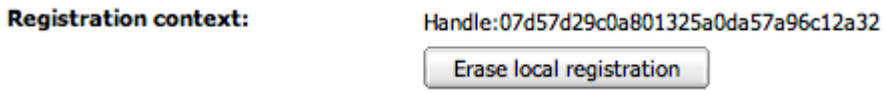


If you now take a look at the `page1` page, you should now see that the content `/samples-remotecontroller-portlet.RemoteControl (remote)` window has been replaced by the content of the `/ajaxPortlet.JSFAJAXPortlet` imported portlet and the window renamed appropriately:

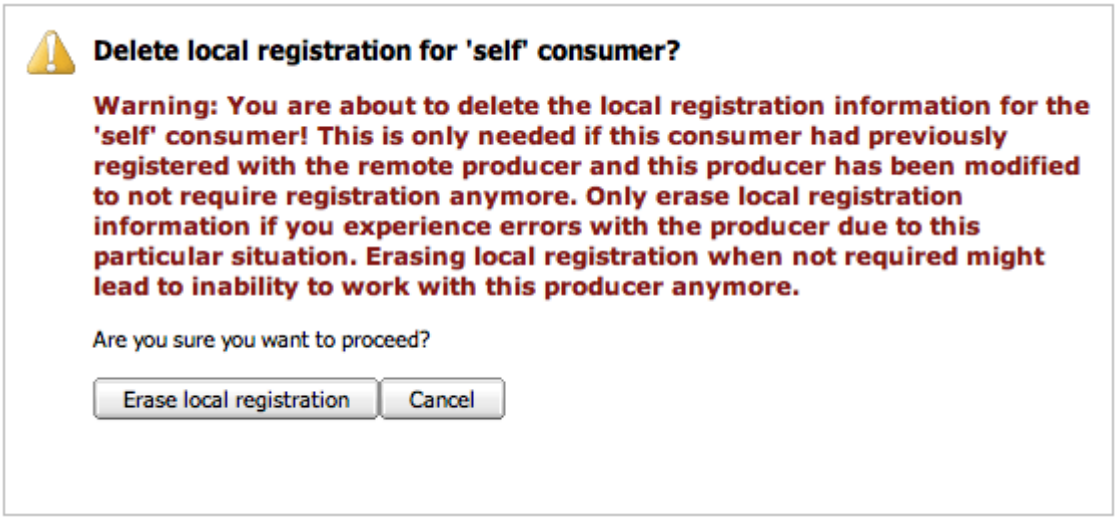

Home
SiteMap
Group Navigation
Sample-Ext Page Portal
Home > page1
NetUnity WSRP 2 Interop - Cache Markup (remote)
Current Date Time: 11/18/2010 1:26:29 PM
Cache Set On: (Not Set)
Initial Cache Expiration Date/Time:
Markup Tag (Final Cache Expiration Date/Time):
User Scope: wsrp:perUser
Markup Expiration: 0
User Scope: wsrp:perUser
Markup Expiration:
Markup Tag (Additional Seconds to Cache):
Submit
Done
/ajaxPortlet.JSFAJAXPortlet (remote)
Media Output Ajax repeater
Here is an example of JSF repeater.
Repeater test
Update
Done

7.7.4 Erasing local registration data

There are rare cases where it might be required to erase the local information without being able to deregister first. This is the case when a consumer is registered with a producer that has been modified by its administrator to not require registration anymore. If that ever was to happen (most likely, it won't), you can erase the local registration information from the consumer so that it can resume interacting with the remote producer. To do so, click on "Erase local registration" button next to the registration context information on the consumer configuration screen:



This operation is dangerous as it can result in inability to interact with the remote producer if invoked when not required. A warning screen will be displayed to give you a chance to change your mind:



7.8 Configuring GateIn's WSRP Producer

The preferred way to configure the behavior of Portal's WSRP Producer is via the WSRP configuration portlet. Alternatively, it is possible to add an XML file called `wsrp-producer-config.xml` in the `$JBOSS_HOME/standalone/configuration/gatein/wsrp` directory. Several aspects can be modified with respects to whether registration is required for consumers to access the Producer's services.

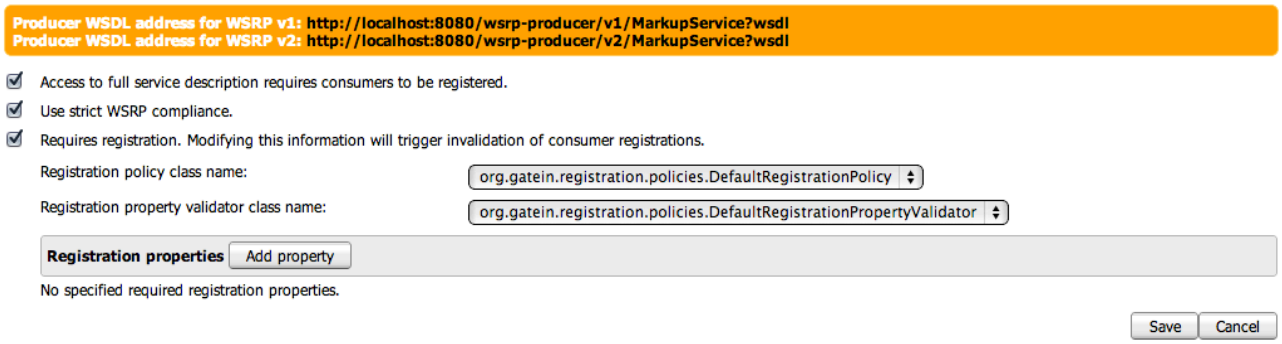
An XML Schema defining which elements are available to configure Consumers via XML can be found in:
`$JBOSS_HOME/modules/org/gatein/wsrp/main/wsrp-integration-api-$WSRP_VERSION`

It is important to note that once the XML configuration file for the producer has been read upon the WSRP service first start, the associated information is put under control of JCR (Java Content Repository). Subsequent launches of the WSRP service will use the JCR-stored information and ignore the content of the XML configuration file.

7.8.1 Default configuration

The default producer configuration requires that consumers register with it before providing access to its services but does not require any specific registration properties (apart from what is mandated by the WSRP standard). It does, however, require consumers to be registered before sending them a full service description. This means that our WSRP producer will not provide the list of offered portlets and other capabilities to unregistered consumers. The producer also uses the default `RegistrationPolicy` paired with the default `RegistrationPropertyValidator`. We will look into property validators in greater detail later in Registration configuration. Suffice to say for now that this allows users to customize how Portal's WSRP Producer decides whether a given registration property is valid or not.

GateIn provides a web interface to configure the producer's behavior. You can access it by clicking on the "Producer Configuration" tab of the "WSRP" page of the "admin" portal. Here's what you should see with the default configuration:

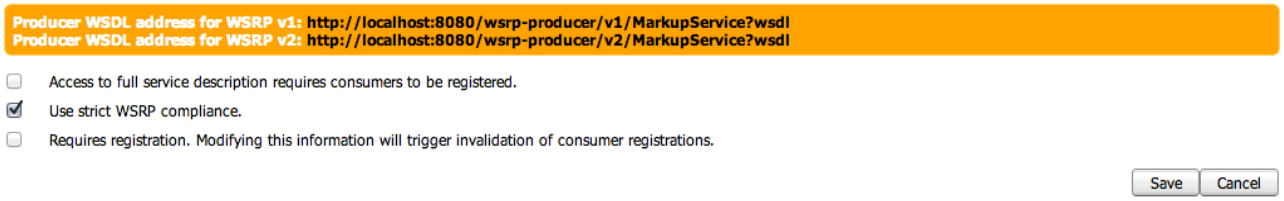


As would be expected, you can specify whether or not the producer will send the full service description to unregistered consumers, and, if it requires registration, which `RegistrationPolicy` to use (and, if needed, which `RegistrationPropertyValidator`), along with required registration property description for which consumers must provide acceptable values to successfully register.

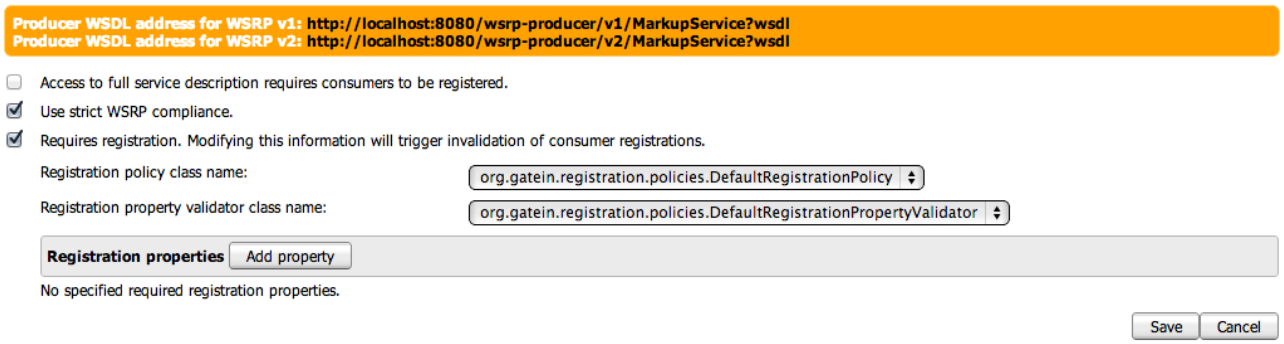
We also display the WSDL URLs to access GateIn's WSRP producer either in WSRP 1 or WSRP 2 mode.

7.8.2 Registration configuration

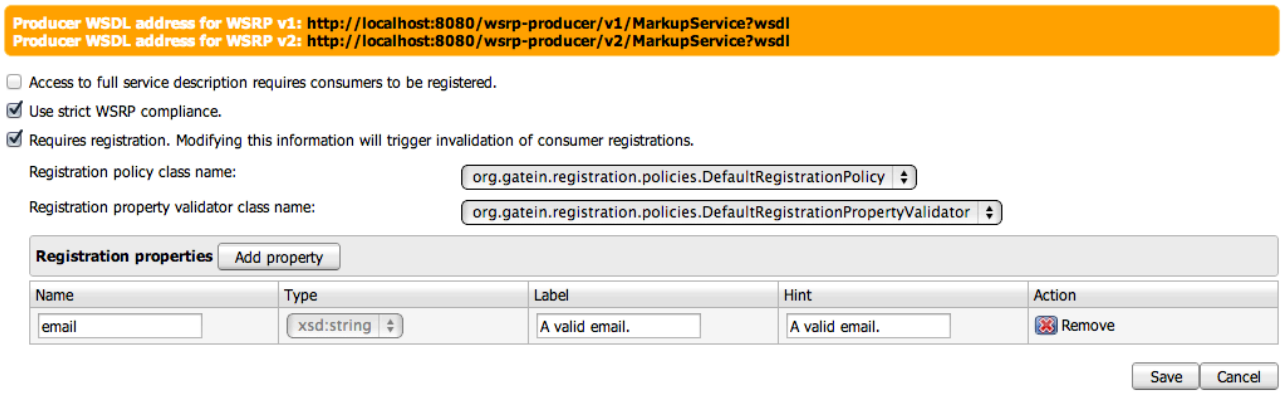
In order to require consumers to register with Portal's producer before interacting with it, you need to configure Portal's behavior with respect to registration. Registration is optional, as are registration properties. The producer can require registration without requiring consumers to pass any registration properties as is the case in the default configuration. Let's configure our producer starting with a blank state:



We will allow unregistered consumers to see the list of offered portlets so we leave the first checkbox ("Access to full service description requires consumers to be registered.") unchecked. We will, however, specify that consumers will need to be registered to be able to interact with our producer. Check the second checkbox ("Requires registration. Modifying this information will trigger invalidation of consumer registrations."). The screen should now refresh and display:



Dropdown menus display available implementations of `RegistrationPolicy` and `RegistrationPropertyValidator`. We will keep the default value for now as we will examine these concepts in more details in Customization of Registration handling behavior for more details. Let's add, however, a registration property called `email`. Click "Add property" and enter the appropriate information in the fields, providing a description for the registration property that can be used by consumers to figure out its purpose:



Press "Save" to record your modifications.

At this time, only String (`xsd:string`) properties are supported. If your application requires more complex properties, please let us know.

If consumers are already registered with the producer, modifying the configuration of required registration information will trigger the invalidation of held registrations, requiring consumers to modify their registration before being able to access the producer again. We saw the consumer side of that process in the Registration modification on producer error section of the Consumers maintenance chapter.

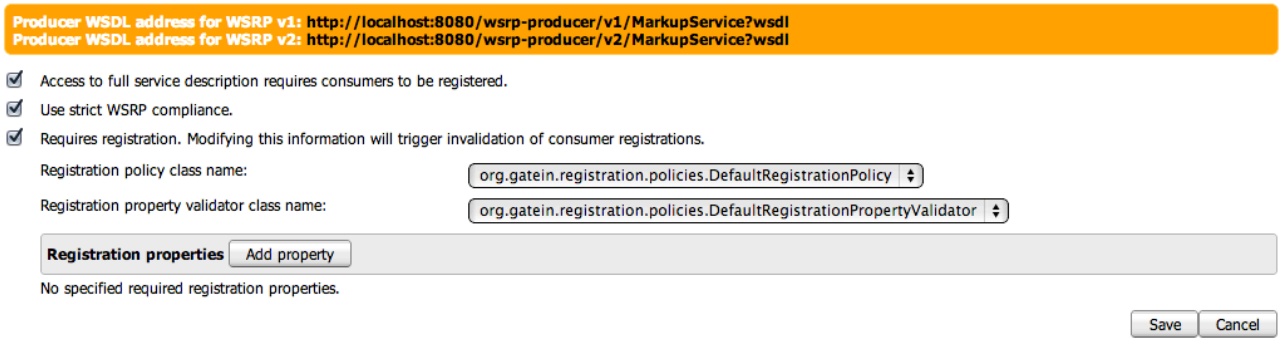
Customization of Registration handling behavior

Registration handling behavior can be customized by users to suit their Producer needs. This is accomplished by providing an implementation of the `RegistrationPolicy` interface. This interface defines methods that are called by Portal's Registration service so that decisions can be made appropriately. A default registration policy that provides basic behavior is provided and should be enough for most user needs.

While the default registration policy provides default behavior for most registration-related aspects, there is still one aspect that requires configuration when using this default registration policy: whether a given value for a registration property is acceptable by the WSRP Producer. This is accomplished by plugging a `RegistrationPropertyValidator` in the default registration policy. This allows users to define their own validation mechanism of registration properties that consumer pass to the producer.

Please refer to the Javadoc for `org.gatein.registration.RegistrationPolicy` and `org.gatein.registration.policies.RegistrationPropertyValidator` for more details on what is expected of each method.

Defining a registration policy is required for the producer to be correctly configured. If you don't provide one, the `DefaultRegistrationPolicy` associated to the `DefaultRegistrationPropertyBehavior` will be used:



GateIn Portal can automatically detect deployed implementations of `RegistrationPolicy` and `RegistrationPropertyValidator`, assuming they conform to the following rules:

- The implementations **must** follow the Java ServiceLoader architecture. In essence, they must be packaged as JARs and contain a special `META-INF/services/org.gatein.registration.RegistrationPolicy` file for `RegistrationPolicy` implementations or `META-INF/services/org.gatein.registration.policies.RegistrationPropertyVali` for `RegistrationPropertyValidator` implementations containing a line with the fully qualified name of the implementation class. Note that you should be able to package several implementations in the same JAR file, provided that each implementation class is referenced on its own line in the appropriate service definition file. To make things easier, we provide an example project of a `RegistrationPolicy` implementation and packaging at https://github.com/gatein/gatein-wsrp/tree/master/examples/policy.
- The implementation classes **must** provide a default, no-argument constructor for dynamic instantiation.
- The implementation JARs **must** be deployed in `$JBOSS_HOME/gatein/extensions`.
- The implementation JARs **should** use the `.wsrp.jar` extension. This is not mandatory but helps process the file faster by marking it as a WSRP extension.

If you have deployed the example `RegistrationPolicy` that we provide as detailed above, `registration-policy-example.wsrp.jar` in `$JBOSS_HOME/gatein/extensions`, it should appear in the list of available policies in the producer configuration screen as follows:

7.8.3 WSRP validation mode

The lack of conformance kit and the wording of the WSRP specification leaves room for differing interpretations, resulting in interoperability issues. It is therefore possible to encounter issues when using consumers from different vendors. We have experienced such issues and have introduced a way to relax the validation that our WSRP producer performs on the data provided by consumers to help with interoperability by accepting data that would normally be invalid. Note that we only relax our validation algorithm on aspects of the specification that are deemed harmless such as invalid language codes.

By default, the WSRP producer is configured in strict mode. If you experience issues with a given consumer, you might want to try to relax the validation mode. This is accomplished by unchecking the "Use strict WSRP compliance." checkbox on the Producer configuration screen.

7.9 Working with WSRP extensions

7.9.1 Overview

The WSRP specifications allows for implementations to extend the protocol using Extensions. GateIn Portal, as of its WSRP implementation version 2.2.0, provides a way for client code (e.g. portlets) to interact with such extensions in the form of several classes and interfaces gathered within the `org.gatein.wsrp.api.extensions` package, the most important ones being `InvocationHandlerDelegate`, `ConsumerExtensionAccessor` and `ProducerExtensionAccessor`.

To be able to use this API, you will need to include the `wsrp-integration-api-$WSRP_VERSION.jar` file to your project, where `$WSRP_VERSION` is the version of the GateIn Portal WSRP implementation you wish to use, 2.2.2.Final being the current one. This can be done by adding the following dependency to your maven project:

```
<dependency>
  <groupId>org.gatein.wsrp</groupId>
  <artifactId>wsrp-integration-api</artifactId>
  <version>$WSRP_VERSION</version>
</dependency>
```

InvocationHandlerDelegate infrastructure

Using the `InvocationHandlerDelegate` infrastructure, custom behavior can now be inserted on either consumer or producer sides to enrich WSRP applications before and/or after portlet requests and/or responses. Please refer to the Javadoc for `org.gatein.wsrp.api.extensions.InvocationHandlerDelegate` for more details on this interface and how to implement it.

Since `InvocationHandlerDelegate` is a very generic interface, it could potentially be used for more than simply working with WSRP extensions. Moreover, since it has access to internal GateIn Portal classes, it is important to be treat access to these internal classes as **read-only** to prevent any un-intentional side-effects.

Injecting InvocationHandlerDelegate implementations

Implementations of `InvocationHandlerDelegate` **must** follow the same constraints as `RegistrationPolicy` implementations as detailed in Customization of Registration handling behavior section of the Configuring GateIn's WSRP Producer chapter, in essence, they **must** follow the Java `ServiceLoader` architectural pattern and be deployed in the appropriate `$JBOSS_HOME/gatein/extensions` directory.

You can specify only one `InvocationHandlerDelegate` implementation per side: one implementation for the consumer and another one for the producer. This is accomplished by passing the fully classified class name to the appropriate system property when the portal is started:

WSRP side	System property
consumer	org.gatein.wsrp.consumer.handlers.delegate
producer	org.gatein.wsrp.producer.handlers.delegate

Example

```
./standalone.sh
-Dorg.gatein.wsrp.consumer.handlers.delegate=com.example.FooInvocationHandlerDelegate
```

will inject the `com.example.FooInvocationHandlerDelegate` class on the consumer side, assuming that class implements the `org.gatein.wsrp.api.extensions.InvocationHandlerDelegate` interface and is packaged and deployed appropriately as explained above.

Accessing extensions from client code

You can access extensions from client code using `ConsumerExtensionAccessor` and `ProducerExtensionAccessor` on the consumer and producer, respectively. Each interface provides several methods but you should only have to ever call two of them on each, as shown in the following table:

Interface	Relevant methods
`ConsumerExtensionAccessor`	• `public void addRequestExtension(Class targetClass, Object extension)` • `public List<UnmarshalledExtension> getResponseExtensionsFrom(Class responseClass)`
`ProducerExtensionAccessor`	• `List<UnmarshalledExtension> getRequestExtensionsFor(Class targetClass)` • `void addResponseExtension(Class wsrpResponseClass, Object extension)`

Please refer to the Javadoc for these classes for more details.

We currently only support adding and accessing extensions from a core subset of WSRP classes pertaining to markup, interaction, resource or event requests and responses, as follows:

Request classes	Response classes
`org.oasis.wsrp.v2.InteractionParams`	`org.oasis.wsrp.v2.MarkupResponse`
`org.oasis.wsrp.v2.EventParams`	`org.oasis.wsrp.v2.BlockingInteractionR`
`org.oasis.wsrp.v2.MarkupParams`	`org.oasis.wsrp.v2.HandleEventsResponse`
`org.oasis.wsrp.v2.ResourceParams`	`org.oasis.wsrp.v2.ResourceResponse`

We **strongly** recommend that you use `org.w3c.dom.Element` values as extensions for interoperability purposes.

7.9.2 Example implementation

We also provide a complete, end-to-end example to get you started, which you can find at https://github.com/gatein/gatein-wsrp/tree/master/examples/invocation-handler-delegate. This example shows how `ExampleConsumerInvocationHandlerDelegate`, a consumer-side `InvocationHandlerDelegate` implementation, can add information extracted from the consumer and pass it along to the producer, working in conjunction with `ExampleProducerInvocationHandlerDelegate`, the associated producer-side `InvocationHandlerDelegate`, to establish a round-trip communication channel outside of the standard WSRP protocol, implementing the following scenario:

- `ExampleConsumerInvocationHandlerDelegate` attaches to the consumer to add the current session id as an extension to render requests sent to the producer.
- `ExampleProducerInvocationHandlerDelegate` provides the counterpart of `ExampleConsumerInvocationHandlerDelegate` on the producer. It checks incoming render requests for potential extensions matching what `ExampleConsumerInvocationHandlerDelegate` sends and adds an extension of its own to the render response so that the consumer-side delegate can know that the information it passed was properly processed.

To activate the InvocationHandlerDelegates on both the consumer and producer, start your GateIn Portal instance as follows:

```
./standalone.sh
-Dorg.gatein.wsrp.consumer.handlers.delegate=org.gatein.wsrp.examples.ExampleConsumerInvocationH
-Dorg.gatein.wsrp.producer.handlers.delegate=org.gatein.wsrp.examples.ExampleProducerInvocationH
```

8 Advanced Development

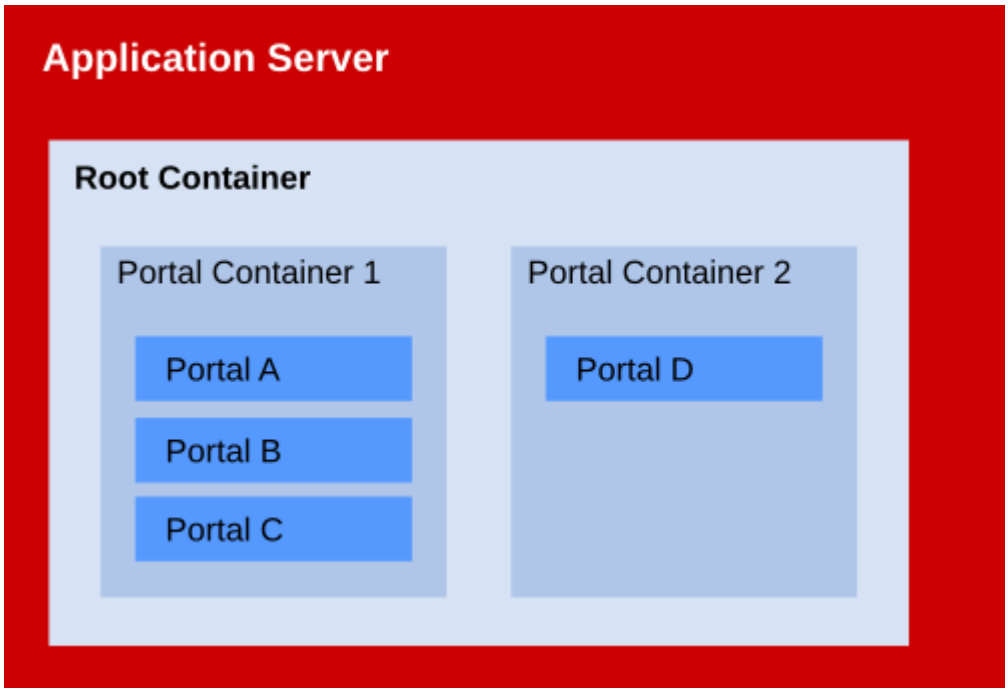
8.1 Foundations

8.1.1 GateIn Kernel

GateIn 3.2 is built as a set of services on top of a dependency injection kernel. The kernel provides configuration, lifecycle handling, component scopes, and some core services.

Service components exist in two scopes. First scope is represented by *RootContainer*- it contains services that exist independently of any portal, and can be accessed by all portals.

Second scope is portal-private in the form of *PortalContainer*. Each portal lives in an instance of PortalContainer. This scope contains services that are common for a set of portals, and services which should not be shared by all portals.



Whenever a specific service is looked up through PortalContainer, and the service is not available, the lookup is delegated further up to RootContainer. We can therefore have default instance of a certain component in RootContainer, and portal specific instances in some or all PortalContainers, that override the default instance.

Whenever your portal application has to be integrated more closely with GateIn services, the way to do it is by looking up these services through PortalContainer. Be careful though - only officially documented services should be accessed this way, and used according to documentation, as most of the services are an implementation detail of GateIn, and subject to change without notice.

8.1.2 Configuring services

GateIn Kernel uses dependency injection to create services based on `configuration.xml` configuration files. The location of the configuration files determines if services are placed into RootContainer scope, or into PortalContainer scope. All `configuration.xml` files located at `conf/configuration.xml` in the classpath (any directory, or any jar in the classpath) will have their services configured at RootContainer scope. All configuration.xml files located at `conf/portal/configuration.xml` in the classpath will have their services configured at PortalContainer scope. Additionally, `portal extensions` can contain configuration in `WEB-INF/conf/configuration.xml`, and will also have their services configured at `PortalContainer` scope.

Portal extensions are described later on.

8.1.3 Configuration syntax

Components

A service component is defined in `configuration.xml` by using `<component>` element.

There is only one required information when defining a service - the service implementation class, specified using `<type>`.

Every component has a `<key>` that identifies it. If not explicitly set, a key defaults to the value of `<type>`. If key can be loaded as a class, a Class object is used as a key, otherwise a String is used.

The usual approach is to specify an interface as a key.

Example of service component configuration:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<configuration
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.exoplaform.org/xml/ns/kernel_1_2.xsd
                          http://www.exoplaform.org/xml/ns/kernel_1_2.xsd"
      xmlns="http://www.exoplaform.org/xml/ns/kernel_1_2.xsd">
   <component>
      <key>org.exoplatform.services.database.HibernateService</key>
      <type>org.exoplatform.services.database.impl.HibernateServiceImpl</type>

      ...

   </component>
</configuration>
```

External Plugins

GateIn Kernel supports non-component objects that can be configured, instantiated, and injected into registered components, using method calls. The mechanism is called 'plugins', and allows portal extensions to add additional configurations to core services.

External plugin is defined by using <*external-component-plugins*> wrapper element which contains one or more <*component-plugin*> definitions. <external-component-plugins> uses <*target-component*> to specify a target service component that will receive injected objects.

Every <component-plugin> defines an implementation type, and a method on target component to use for injection (<*set-method*>).

A plugin implementation class has to implement *org.exoplatform.container.component. ComponentPlugin* interface.

In the following example *PortalContainerDefinitionPlugin* implements ComponentPlugin:

PortalContainerDefinitionPlugin

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.exoplaform.org/xml/ns/kernel_1_2.xsd
                          http://www.exoplaform.org/xml/ns/kernel_1_2.xsd"
      xmlns="http://www.exoplaform.org/xml/ns/kernel_1_2.xsd">

   <external-component-plugins>

<target-component>org.exoplatform.container.definition.PortalContainerConfig</target-component>
      <component-plugin>
         <!-- The name of the plugin -->
         <name>Add PortalContainer Definitions</name>

         <!-- The name of the method to call on the PortalContainerConfig
              in order to register the PortalContainerDefinitions -->
         <set-method>registerPlugin</set-method>

         <!-- The fully qualified name of the PortalContainerDefinitionPlugin -->
         <type>org.exoplatform.container.definition.PortalContainerDefinitionPlugin</type>

         ...

      </component-plugin>
   </external-component-plugins>
</configuration>
```

Includes, and special URLs

It is possible to break the `configuration.xml` file into many smaller files, that are then included into a 'master' configuration file. The included files are complete configuration xml documents by themselves - they are not fragments of text.

An example configuration.xml that 'outsources' its content into several files:

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

   <import>war:/conf/sample-ext/jcr/jcr-configuration.xml</import>
   <import>war:/conf/sample-ext/portal/portal-configuration.xml</import>

</configuration>
```

We see a special URL being used to reference another configuration file. URL schema *'war:'* means, that the path that follows is resolved relative to current PortalContainer's servlet context resource path, starting at *WEB-INF* as a root.

Current PortalContainer is really a newly created PortalContainer, as war: URLs only make sense for PortalContainer scoped configuration.

Also, thanks to extension mechanism, the servlet context used for resource loading is a *unified servlet context* (as explaned in a later section).

To have include path resolved relative to current classpath (context classloader), use *'jar:'* URL schema.

Special variables

Configuration files may contain a *special variable* reference *${container.name.suffix}*. This variable resolves to the name of the current portal container, prefixed by underscore (_). This facilitates reuse of configuration files in situations where portal specific unique names need to be assigned to some resources (i.e. JNDI names, Database / DataSource names, JCR repository names, etc ...).

This variable is only defined when there is a current PortalContainer available - only for PortalContainer scoped services.

A good example for this is *HibernateService*:

HibernateService using variables

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<configuration
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.exoplaform.org/xml/ns/kernel_1_2.xsd
                       http://www.exoplaform.org/xml/ns/kernel_1_2.xsd"
   xmlns="http://www.exoplaform.org/xml/ns/kernel_1_2.xsd">

   <component>
      <key>org.exoplatform.services.database.HibernateService</key>
      <jmx-name>database:type=HibernateService</jmx-name>
      <type>org.exoplatform.services.database.impl.HibernateServiceImpl</type>
      <init-params>
         <properties-param>
            <name>hibernate.properties</name>
            <description>Default Hibernate Service</description>
            <property name="hibernate.show_sql" value="false" />
            <property name="hibernate.cglib.use_reflection_optimizer" value="true" />
            <property name="hibernate.connection.url"
                           value="jdbc:hsqldb:file:../temp/data/exodb${container.name.suffix}"
/>
            <property name="hibernate.connection.driver_class" value="org.hsqldb.jdbcDriver" />
            <property name="hibernate.connection.autocommit" value="true" />
            <property name="hibernate.connection.username" value="sa" />
            <property name="hibernate.connection.password" value="" />
            <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect" />
            <property name="hibernate.c3p0.min_size" value="5" />
            <property name="hibernate.c3p0.max_size" value="20" />
            <property name="hibernate.c3p0.timeout" value="1800" />
            <property name="hibernate.c3p0.max_statements" value="50" />
         </properties-param>
      </init-params>
   </component>
</configuration>
```

8.1.4 InitParams configuration object

`InitParams` is a configuration object that is essentially a map of key-value pairs, where key is always a `String`, and value can be any type that can be described using kernel configuration xml.

Service components that form the GateIn 3.2 insfrastructure use `InitParams` object to configure themselves. A component can have one instance of InitParams injected at most. If the service component's constructor takes InitParams as any of the parameters it will automatically be injected at component instantiation time. The xml configuration for a service component that expects `InitParams` object must include `<init-params>` element (even if an empty one).

Let's use an example to see how the kernel xml configuration syntax looks for creating `InitParams` instances.

InitParams - properties-param

```
<component>
   <key>org.exoplatform.services.naming.InitialContextInitializer</key>
   <type>org.exoplatform.services.naming.InitialContextInitializer</type>
   <init-params>
      <properties-param>
         <name>default-properties</name>
         <description>Default initial context properties</description>
         <property name="java.naming.factory.initial"
                   value="org.exoplatform.services.naming.SimpleContextFactory" />
      </properties-param>
   </init-params>
</component>
```

The `InitParams` object description begins with `<init-params>` element. It can have zero or more children elements each of which is one of `<value-param>`, `<values-param>`, `<properties-param>`, or `<object-param>`. Each of these child elements takes a <name> that serves as a map entry key, and an optional `<description>`. It also takes a type-specific value specification.

For `<properties-param>`, the value specification is in the form of one or more `<property>` elements, each of which specifies two strings - a property name, and a property value. Each `<properties-params>` defines one `java.util.Properties` instance. Also see sect-Reference_Guide-Foundations-Configuration_syntax-Special_vars-Example for an example.

InitParams - value-param

```
<component>
   <key>org.exoplatform.services.transaction.TransactionService</key>
   <type>org.exoplatform.services.transaction.impl.jotm.TransactionServiceJotmImpl</type>
   <init-params>
      <value-param>
         <name>timeout</name>
         <value>5</value>
      </value-param>
   </init-params>
</component>
```

For `<value-param>`, the value specification is in the form of `<value>` element, which defines one `String` instance.

InitParams - values-param

```
<component>
  <key>org.exoplatform.services.resources.ResourceBundleService</key>
  <type>org.exoplatform.services.resources.impl.SimpleResourceBundleService</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.test.resources.test</value>
      </values-param>

      <values-param>
        <name>portal.resource.names</name>
        <description>The properties files of  the portal ,  those file will be merged
          into one ResourceBundle properties </description>
        <value>local.portal.portal</value>
        <value>local.portal.custom</value>
      </values-param>
    </init-params>
</component>
```

For `<values-param>`, the value specification is in the form of one or more <value> elements, each of which represents one `String` instance, where all the `String` values are then collected into a `java.util.List` instance.

InitParams - object-param

```
<component>
   <key>org.exoplatform.services.cache.CacheService</key>
   <jmx-name>cache:type=CacheService</jmx-name>
   <type>org.exoplatform.services.cache.impl.CacheServiceImpl</type>
   <init-params>
      <object-param>
         <name>cache.config.default</name>
         <description>The default cache configuration</description>
         <object type="org.exoplatform.services.cache.ExoCacheConfig">
            <field name="name">
               <string>default</string>
            </field>
            <field name="maxSize">
               <int>300</int>
            </field>
            <field name="liveTime">
               <long>300</long>
            </field>
            <field name="distributed">
               <boolean>false</boolean>
            </field>
            <field name="implementation">
               <string>org.exoplatform.services.cache.concurrent.ConcurrentFIFOExoCache</string>
            </field>
         </object>
      </object-param>
   </init-params>
</component>
```

For `<object-param>` in our case, the value specification comes in a form of `<object>` element, which is used for POJO style object specification (you specify an implementation class - `<type>`, and property values - `<field>`).

Also see sect-Reference_Guide-Foundations-Configuring_portal_Container_declaration_example for an example of specifying a field of `Collection` type.

The `InitParams` structure - the names and types of entries is specific for each service, as it is the code inside service components's class that decides what entry names to look up and what types it expects to find.

8.1.5 Configuring a portal container

A *portal container* is defined by several attributes.

First, there is a *portal container name*, which is always equal to URL context to which the current portal is bound.

Second, there is a *REST context name*, which is used for REST access to portal application - every portal has exactly one (unique) REST context name.

Then, there is a *realm name* which is the name of security realm used for authentication when users log into the portal.

Finally, there is a list of *Dependencies*- other web applications, whose resources are visible to current portal (via extension mechanism described later), and are searched in the specified order.

Portal container declaration example

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.exoplaform.org/xml/ns/kernel_1_2.xsd
                       http://www.exoplaform.org/xml/ns/kernel_1_2.xsd"
   xmlns="http://www.exoplaform.org/xml/ns/kernel_1_2.xsd">

   <external-component-plugins>
      <!-- The full qualified name of the PortalContainerConfig -->

<target-component>org.exoplatform.container.definition.PortalContainerConfig</target-component>

      <component-plugin>
         <!-- The name of the plugin -->
         <name>Add PortalContainer Definitions</name>

         <!-- The name of the method to call on the PortalContainerConfig
              in order to register the PortalContainerDefinitions -->
         <set-method>registerPlugin</set-method>

         <!-- The full qualified name of the PortalContainerDefinitionPlugin -->
         <type>org.exoplatform.container.definition.PortalContainerDefinitionPlugin</type>

         <init-params>
            <object-param>
               <name>portal</name>
               <object type="org.exoplatform.container.definition.PortalContainerDefinition">
                  <!-- The name of the portal container -->
                  <field name="name"><string>portal</string></field>

                  <!-- The name of the context name of the rest web application -->
                  <field name="restContextName"><string>rest</string></field>

                  <!-- The name of the realm -->
                  <field name="realmName"><string>exo-domain</string></field>

                  <!-- All the dependencies of the portal container ordered by loading priority
-->
                  <field name="dependencies">
                     <collection type="java.util.ArrayList">
                        <value>
                           <string>eXoResources</string>
                        </value>
                        <value>
                           <string>portal</string>
                        </value>
                        <value>
                           <string>dashboard</string>
```

```
                        </value>
                        <value>
                           <string>exoadmin</string>
                        </value>
                        <value>
                           <string>eXoGadgets</string>
                        </value>
                        <value>
                           <string>eXoGadgetServer</string>
                        </value>
                        <value>
                           <string>rest</string>
                        </value>
                        <value>
                           <string>web</string>
                        </value>
                        <value>
                           <string>wsrp-producer</string>
                        </value>
                        <!-- The sample-ext has been added at the end of the dependency list
                             in order to have the highest priority -->
                        <value>
                           <string>sample-ext</string>
                        </value>
                     </collection>
                  </field>
               </object>
            </object-param>
         </init-params>
      </component-plugin>
   </external-component-plugins>
</configuration>
```

Dependencies are part of the extension mechanism.

Every *portal container* is represented by a *PortalContainer instance*, which contains:

- associated *ExoContainerContext*, which contains information about the portal.
- *unified servlet context*, for web-archive-relative resource loading.
- *unified classloader*, for classpath based resource loading.
- methods for retrieving services.

Unified servlet context, and *unified classloader* are part of the *extension mechanism* (explained in next section), and provide standard API (ServletContext, ClassLoader) with specific resource loading behavior - visibility into associated web application archives, configured with Dependencies property of PortalContainerDefinition. Resources from other web applications are queried in the order specified by Dependencies. The later entries in the list override the previous ones.

8.1.6 GateIn Extension Mechanism, and Portal Extensions

Extension mechanism is a functionality that makes it possible to override portal resources in an almost plug-and-play fashion - just drop in a .war archive with the resources, and configure its position on the portal's classpath. This way any customizations of the portal don't have to involve unpacking and repacking the original portal .war archives. Instead, you create your own .war archive with changed resources, that override the resources in the original archive.

A web archive packaged in a way to be used through extension mechanism is called *portal extension*.

There are two steps necessary to create a portal extension.

First, declare *PortalConfigOwner* servlet context listener in web.xml of your web application.

Example of a portal extension called sample-ext:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE web-app PUBLIC -//Sun Microsystems, Inc.//DTD Web Application 2.3//EN
     http://java.sun.com/dtd/web-app_2_3.dtd>
<web-app>

   <display-name>sample-ext</display-name>

   <listener>
      <listener-class>org.exoplatform.container.web.PortalContainerConfigOwner</listener-class>
   </listener>

   ...

</web-app>
```

Then, add the servlet context name of this web application in proper place in the list of Dependencies of the PortalContainerDefinition of all the portal containers that you want to have access to its resources.

After this step, your web archive will be on portal's unified classpath, and unified servlet context resource path. The later in the Dependencies list your application is, the higher priority it has when resources are loaded by portal.

See 'Configuring a portal' section for example of PortalContainerDefinition, that has sample-ext at the end of its list of Dependencies.

8.1.7 Running Multiple Portals

It is possible to run several independent portal containers - each bound to a different URL context - within the same JVM instance. This kind of setup is very efficient from administration and resource consumption aspect. The most elegant way to *reuse* configuration for different coexisting portals is by way of extension mechanism - by *inheriting* resources and configuration from existing web archives, and just *adding* extra resources to it, and *overriding* those that need to be changed by including modified copies.

In order for a portal application to correctly function when deployed in multiple portals, the application may have to dynamically query the information about the current portal container. The application should not make any assumptions about the name, and other information of the current portal, as there are now multiple different portals in play.

At any point during request processing, or lifecycle event processing, your application can retrieve this information through *org.exoplatform.container. ExoContainerContext*. Sometimes your application needs to make sure that the proper *PortalContainer*- the source of *ExoContainerContext*- is associated with the current call.

If you ship servlets or servlet filters as part of your portal application, and if you need to access portal specific resources at any time during the processing of the servlet or filter request, then you need to make sure the servlet/filter is associated with the current container.

The proper way to do that is to make your servlet extend *org.exoplatform.container.web. AbstractHttpServlet* class. This will not only properly initialize current *PortalContainer* for you, but will also set the current thread's context classloader to one that looks for resources in associated web applications in the order specified by *Dependencies* configuration (as explained in Extension mechanism section).

Similarly for filters, make sure your filter class extends *org.exoplatform.container.web. AbstractFilter*. Both *AbstractHttpServlet*, and *AbstractFilter* have a method *getContainer()*, which returns the current *PortalContainer*. If your servlet handles the requests by implementing a *service()* method, you need to rename that method to match the following signature:

```
/**
 * Use this method instead of Servlet.service()
 */
protected void onService(ExoContainer container, HttpServletRequest req,
      HttpServletResponse res) throws ServletException, IOException;
```

The reason is that AbstractHttpServlet implements service() to perform its interception, and you don't want to overwrite (by overriding) this functionality.

You may also need to access portal information within your `HttpSessionListener`. Again, make sure to extend the provided abstract class - `org.exoplatform.container.web.AbstractHttpSessionListener`. Also, modify your method signatures as follows:

```
/**
 * Use this method instead of HttpSessionListener.sessionCreated()
 */
protected void onSessionCreated(ExoContainer container, HttpSessionEvent event);

/**
 * Use this method instead of HttpSessionListener.sessionDestroyed()
 */
protected void onSessionDestroyed(ExoContainer container, HttpSessionEvent event);
```

There is another method you have to implement in this case:

```
/**
 * Method should return true if unified servlet context,
 * and unified classloader should be made available
 */
protected boolean requirePortalEnvironment();
```

If this method returns true, current thread's context classloader is set up according to *Dependencies* configuration, and availability of the associated web applications. If it returns false, the standard application separation rules are used for resource loading (effectively turning off the extension mechanism). This method exists on `AbstractHttpServlet` and `AbstractFilter` as well, where there is a default implementation that automatically returns true, when it detects there is a current PortalContainer present, otherwise it returns false.

We still have to explain how to properly perform `ServletContextListener` based initialization, when you need access to current `PortalContainer`.

GateIn has no direct control over the deployment of application archives (.war, .ear files) - it is the application server that performs the deployment. For *extension mechanism* to work properly, the applications, associated with the portal via *Dependencies* configuration, have to be deployed before the portal, that depends on them, is initialized. On the other hand, these applications may require an already initialized PortalContainer to properly initialize themselves - we have a recursive dependency problem. To resolve this problem, a mechanism of *initialization tasks*, and *task queues*, was put in place. Web applications that depend on current PortalContainer for their initialization have to avoid performing their initialization directly in some ServletContextListener executed during their deployment (before any PortalContainer was initialized). Instead, a web application should package its initialization logic into an init task of appropriate type, and only use ServletContextListener to insert the init task instance into the proper init tasks queue.

An example of this is Gadgets application which registers Google gadgets with the current PortalContainer:

```
public class GadgetRegister implements ServletContextListener
{
   public void contextInitialized(ServletContextEvent event)
   {
      // Create a new post-init task
      final PortalContainerPostInitTask task = new PortalContainerPostInitTask() {

         public void execute(ServletContext context, PortalContainer portalContainer)
         {
            try
            {
               SourceStorage sourceStorage =
               (SourceStorage) portalContainer.getComponentInstanceOfType(SourceStorage.class);
               ...
            }
            catch (RuntimeException e)
            {
               throw e;
            }
            catch (Exception e)
            {
               throw new RuntimeException("Initialization failed: ", e);
            }
         }
      };

      // Add post-init task for execution on all the portal containers
      // that depend on the given ServletContext according to
      // PortalContainerDefinitions (via Dependencies configuration)
      PortalContainer.addInitTask(event.getServletContext(), task);
   }
}
```

The above example uses `PortalContainerPostInitTask`, which gets executed after the portal container has been initialized. In some situations you may want to execute initialization after portal container was instantiated, but before it was initialized - use `PortalContainerPreInitTask` in that case. Or, you may want to execute initialization after all the post-init tasks have been executed - use `PortalContainerPostCreateTask` in that case.

One more area that may need your attention are `LoginModules`. If you use custom LoginModules, that require current ExoContainer, make sure they extend `org.exoplatform.services.security.jaas.AbstractLoginModule` for proper initialization. `AbstractLoginModule` also takes care of the basic configuration - it recognizes two initialization options - `portalContainerName`, and `realmName` whose values you can access via protected fields of the same name.

9 Server Integration

Integration into a specific application server / web container involves several aspects of GateIn configuration, operation, and features availability.

9.1 JBoss AS7 Integration

JBoss AS7 is a very compact, modular, and extensible application server, based on concurrent, and isolated classloading layer called *JBoss Modules*, and on fully parallel kernel called *Modular Service Container (MSC)*. On top of these there is a standalone server with a notion of subsystems, and centralized configuration management.

Most notable features of AS7 are parallel deployment of application archives, and finegrained class isolation. The result is a combination of great performance, and multitenant coexistence of different applications without cross-influences between them.

9.1.1 GateIn Subsystem

GateIn integrates into JBoss AS7 via a custom subsystem. The main JBoss AS7 configuration file where subsystems are configured is located at `$JBOSS_HOME/standalone/configuration/standalone.xml`.

GateIn subsystem is activated with the following xml snippet:

```
<subsystem xmlns="urn:jboss:domain:gatein:1.0">
          <portlet-war-dependencies>
            <dependency name="org.gatein.wci"/>
            <dependency name="org.gatein.pc"/>
            <dependency name="javax.portlet.api"/>
          </portlet-war-dependencies>
        </subsystem>
```

The `<portlet-war-dependencies>` element specifies a list of libraries (available as modules under `$JBOSS_HOME/modules` directory) that are automatically available to all your portlet applications. The listed modules are required, and should not be removed. It's possible to add additional modules, which might allow you to deploy some existing portlet applications without repackaging them.

JBoss AS7 provides mechanisms for configuring which modules are visible to a specific deployment archive. One mechanism involves adding additional attribute to deployment archive's `MANIFEST.MF` - `Dependencies` attribute in `MANIFEST.MF`. Another one involves adding a JBoss AS7 specific descriptor file `jboss-deployment-structure.xml` to your deployment archive. You can read more about it in JBoss AS7 Developer Guide - Class Loading in AS7.

When *gatein* subsystem is active it takes care of boot-time GateIn initialization. Boottime initialization expects to find `gatein.ear` deployment archive at `$JBOSS_HOME/gatein/gatein.ear`, and it looks for any additional deployment archives in `$JBOSS_HOME/gatein/extensions`. These additional deployment archives can have any names, but have to be either one of .ear, .war, or .jar.

Archives placed in `$JBOSS_HOME/gatein/extensions` are not hot-deployable. They are treated as extensions of default `gatein.ear` - meaning that this is a place for archives that integrate with GateIn configuration management system as they install additional GateIn kernel services, override default services configuration, or add/override default portal resources. All GateIn custom skins, custom portals, and custom extensions should be deployed to this directory.

GateIn's Extension mechanism is described in Developer's Guide - Portal Containers and Extensions, and in Reference Guide - Foundations.

9.1.2 Standalone Mode

JBoss AS7 supports two different running modes. It can run as a standalone server (a *standalone mode*), or it can run as part of a server domain (a *domain mode*).

The difference between the two modes is in configuration management. Domain mode provides central configuration management for multiple servers, while in standalone mode every server maintains its own local configuration. There is no difference in available server features between the two modes, and advanced setups like clustering with session replication, and single sign-on can be configured in both standalone, and domain mode.

GateIn 3.5 only supports *standalone mode*.

9.1.3 Directories and files of interest

There are several locations in JBoss AS7 where GateIn adds code or configuration:

- `$JBOSS_HOME/gatein`

... contains `gatein.ear` deployment archive, which contains all the portal resources packaged as multiple web application archives. The libraries are available as modules under `$JBOSS_HOME/modules`, and are not part of the deployment archive. When implementing your portal by starting from scratch and replacing significant parts of the default portal with your own functionality, you can make changes directly in this directory to completely redefine many aspects of the portal.

- `$JBOSS_HOME/gatein/extensions`

... contains custom extensions, custom portals, and custom skins. Initially this directory is empty. When building your portal as a customization of existing default portal, or when adding additional custom portals or custom skins, you would create your own portal extension archives and place them in this directory.

- `$JBOSS_HOME/modules`

... contains classes packaged as isolated modules. It contains both modules that come with JBoss AS7, and additional modules added by GateIn. Developers can add their own modules here. Libraries that are to be shared between different application archives should be packaged as jboss modules, and placed here.

- `$JBOSS_HOME/standalone/configuration/gatein`

... contains gatein configurations - i.e. the master configuration.properties file.

- `$JBOSS_HOME/standalone/data/gatein`

... contains JCR Lucene indexes.

- `$JBOSS_HOME/standalone/deployments`

... a directory used by JBoss AS7 deployment scanner to auto-detect and hot-deploy application archives. Your portlet applications, web applications, and Java EE applications can be placed here in order to be deployed. You should not place in this directory custom portal extensions, custom portals, or custom skins. Those archives deeply integrate with GateIn kernel, and are not hot-deployable. They should be placed in `$JBOSS_HOME/gatein/extensions` in order to be deployed as part of GateIn boot-time initialization.

- `$JBOSS_HOME/standalone/configuration/standalone.xml`

This is a master JBoss AS7 configuration file. It contains *gatein* subsystem configuration, configuration of datasources, configuration of security realms required by deployed portals, and configuration for all the other JBoss AS7 subsystems.

9.1.4 JBoss AS7 specific services

There are a few GateIn services that are only available with JBoss AS7:

- SSO
- Clustered mode
- WSRP

9.2 Tomcat Integration

TODO