Jetspeed-2 Security Components v.2.1.2

Project Documentation

Apache Software Foundation **15 July 2007**

Table of Contents

1.1 Overview

Overview

Jetspeed 2 security architecture provides a comprehensive suite of security services that can be used to protect a wide ranging type of portal resources. The security service implementation is fairly independent of the other portal services and can be reused outside of the portal application. At its core, Jetspeed 2 security services rely entirely on JAAS to provide authentication and authorization services to the portal:

- Authentication services are implemented through the use of JAAS login modules.
- Authorization services are implemented through the use of custom JAAS policies.

Both authentication and authorization services have been implemented with the goal of providing a direct plugin to the underlying application server security framework. Jetspeed 2 can leverage the underlying application server login module as well as through the use of JACC, the application server policy management capabilities available in J2EE 1.4 (see API Specifications).

Jetspeed 2 Security Services

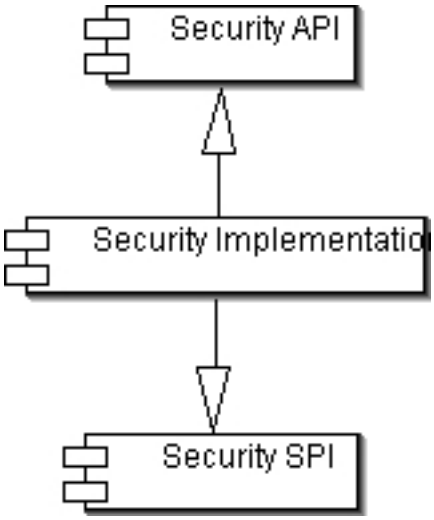
JAAS defines the contract for authentication and authorization but does not specify any guidelines for the management of the security resources. Jetspeed 2 provide a modular set of components aims at providing management functionality for the portal security components.

Leveraging Jetspeed 2 component, architecture, the security services provide a set of loosely coupled components providing specialized services:

- UserManager: Service providing user management capabilities.
- GroupManager: Service providing group management capabilities.
- RoleManager: Service providing role management capabilities.
- PermissionManager: Service providing permission management capabilities.

A Modular and Pluggable Architecture

Jetspeed 2 security components are assembled using Dependency Injection . By default, Jetspeed uses the Spring Framework as its default IoC container.



Jetspeed 2 security services are founded on a set of modular and extensible security modules exposed through an SPI model. The SPI model provides the ability to modify the behavior of the Jetspeed coarsed security services (UserManager, RoleManager, GroupManager) through the modification and configuration of specialized handlers. For instance, Jetspeed security services can be configured to retrieve user security principals through the default Jetspeed store or through an LDAP store or both.
A `SecurityProvider` exposes the configured SPI handlers to the security services. Jetspeed component assembly (based on Spring) architecture provides an easy way to reconfigure the security services to satisfy the needs of a specific implementation.

Role Based Access Control

Role based access control (RBAC) in Jetspeed 2 support multiple hierarchy resolution strategies as defined in The Uses of Hierarchy in Access Control . See Hierarchy Management Overview for more information.

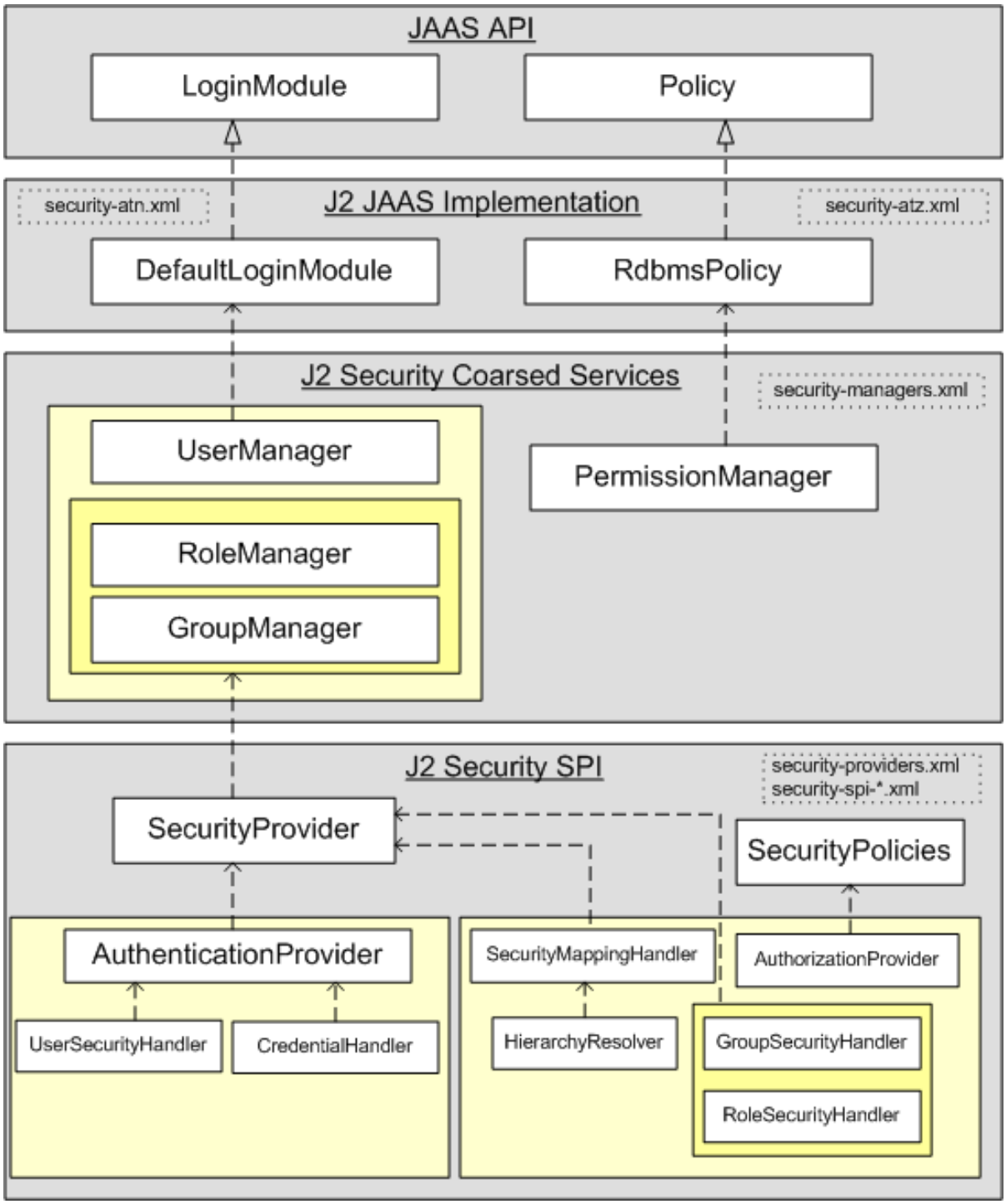
1.2 Architecture Overview

Architecture Overview

Jetspeed 2 security leverages J2EE authentication and authorization standards for both authentication and authorization through the implementation of a default `LoginModule` and a default authorization `Policy`.

Authentication establishes the identity of the user and populates the `Subject` with all the user principals. In a portal context, the populated `Subject` is added to the session in the `org.apache.jetspeed.security.SecurityValve` implementation. The `Subject` principals are then used to authorize the user's access to a given resource. It leverages JAAS authorization by checking the user's permission with the AccessController . More details on authorization are provided in the JAAS authorization section of this documentation.

The following diagram describes the high level security architecture:



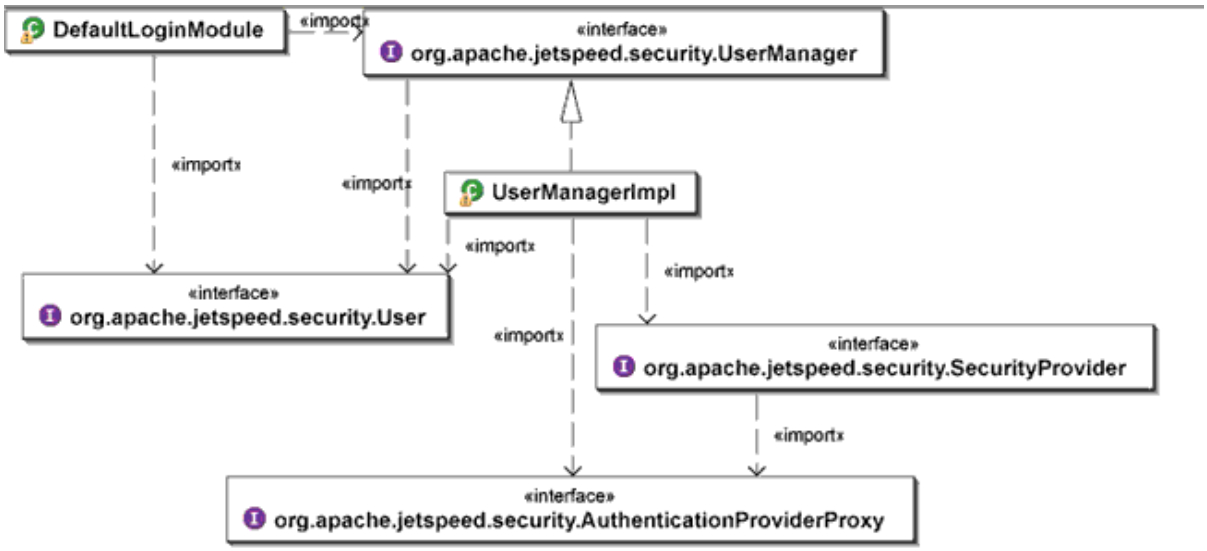
Configuration files for each component areas are specified. For more information, go to the documentation section on configuration .

Jetspeed security architecture is fully JAAS compliant. Developers can replace Jetspeed security architecture with their own `LoginModule` and `Policy` implementation. Jetspeed implementation provides management programming and user interfaces as well as an SPI model to facilitate its extension.

1.3 Authentication

Authentication Architecture Overview

For authentication, Jetspeed 2 leverages Java LoginModule architecture. It provides a DefaultLoginModule implementation and a flexible architecture to be able to authenticate user against multiple user repositories and provide user management capabilities across those repository. A `UserManager` provides a set of coarsed services for authenticating and managing users. The class diagram below illustrates how the `UserManager` provides authentication to the `DefaultLoginModule` and leverages the Authentication SPI to interact with various implementation and user stores.



The various components described above fulfill the following functions:

Component	Description
`DefaultLoginModule`	Jetspeed 2 default LoginModule implementation which leverages the `authenticate()` method of the `UserManager` to provide authentication against the various `AuthenticationProvider` implementation currently configured.
`UserManager`	Coarsed service providing authentication and user management. The `UserManager`code> leverages the various `AuthenticationProvider` implementations exposed to it through the `AuthenticationProviderProxy` through the `SecurityProvider`.
`SecurityProvider`	Provides access to the security providers exposing SPI implementation to the coarsed security services.

Component	Description
`AuthenticationProviderProxy`	A proxy to the various `AuthenticationProvider` implementations. The `AuthenticationProviderProxy` is responsible of invoking the correct `AuthenticationProvider` to authenticate or manage a specific user against a specific data store.

1.3.1 Login Module

Login Module Overview

For authentication purpose, Jetspeed 2 provide a default login module implementation. Login modules provide a standard way to expose authentication services for java application. More information about login modules can be found in the JDK LoginModule interface documentation.

Login Module Configuration

Configuration is central to JAAS authentication. By default, Jetspeed 2 is configured to use its `DefaultLoginModule` implementation. The configuration file (login.conf) for the login module ship with the `jetspeed2-security-{version}.jar` component and provide the following configuration:

```
Jetspeed {
   org.apache.jetspeed.security.impl.DefaultLoginModule required;
};
```

In order to override this configuration, you can place your own login.conf file in your web application class path under WEB-INF/classes. The location of the login.conf file is configured in the `security-providers.xml` as described below. For more information on how to configure the security providers, see the configuration section .

```
<!-- Security: Default Authentication Provider -->
<bean id="org.apache.jetspeed.security.AuthenticationProvider"
            class="org.apache.jetspeed.security.impl.AuthenticationProviderImpl"
>
    <constructor-arg index="0"><value>DefaultAuthenticator</value></constructor-arg>
          <constructor-arg index="1"><value>The default
authenticator</value></constructor-arg>
          <constructor-arg index="2"><value>login.conf</value></constructor-arg>
          <constructor-arg index="3">
              <ref bean="org.apache.jetspeed.security.spi.CredentialHandler"/>
          </constructor-arg>
          <constructor-arg index="4">
              <ref bean="org.apache.jetspeed.security.spi.UserSecurityHandler"/>
    </constructor-arg>
</bean>
```

The `AuthenticationProvider` configures the `LoginModule` to be used by the application by setting the System property `java.security.auth.login.config` to the `login.conf` specified in the component configuration.

Login Module Implementation

The `DefaultLoginModule` implementation is illustrated by the class diagram below:

The roles of the classes used to implement the DefaultLoginModule are:

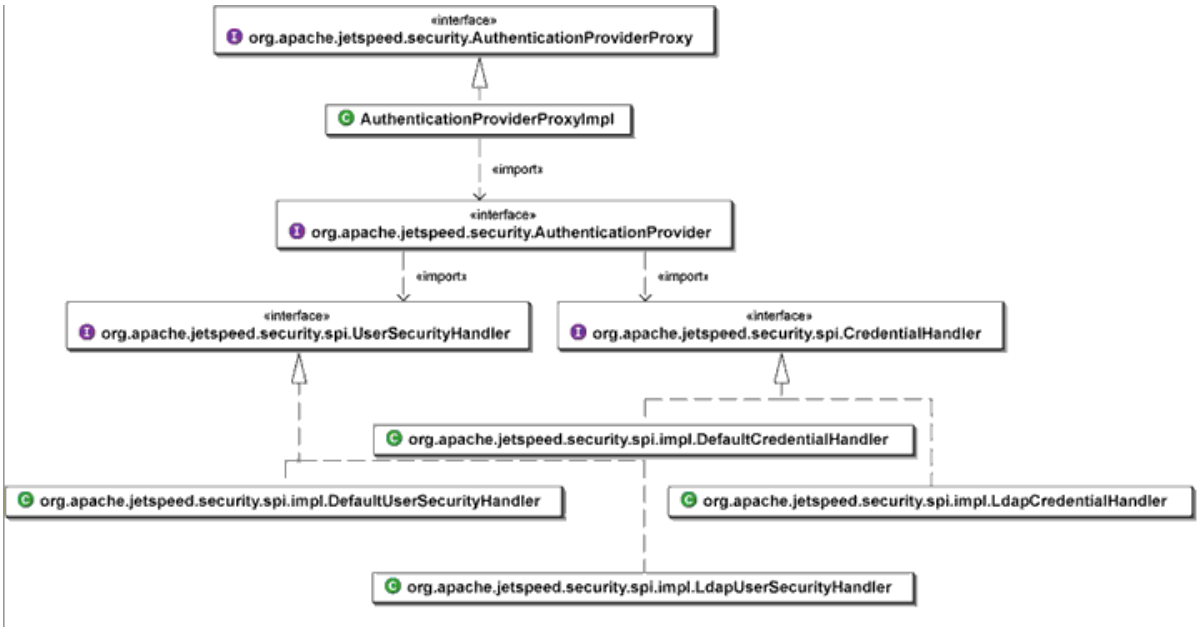
Class	Description
`org.apache.jetspeed.security.impl.DefaultLoginModule`	The `javax.security.auth.spi.LoginModule` implementation. The `DefaultLoginModule` authentication decision is encapsulated behind the `UserManager` interface which leverages the SPI implementation to decide which authenticator should be used in order to authenticate a user against a specific system of record. For more information on how to implement your own authenticator, see the authentication SPI documentation .
`org.apache.jetspeed.security.LoginModuleProxy`	A utility component used to expose the `UserManager` to the `DefaultLoginModule`.
`org.apache.jetspeed.security.User`	The `User` is an interface that holds the `javax.security.auth.Subject` and his/her `java.util.prefs.Preferences`. The `UserManager` upon user authentication populates the user subject with all user `java.security.Principal`. Jetspeed 2 implements 3 types of principals: • UserPrincipal: The principal holding the user unique identifier for the application. • RolePrincipal: The principal representing a role for the system. • GroupPrincipal: The principal representing a group for the system.
`org.apache.jetspeed.security.UserManager`	The interface exposing all user operations. This interfaces fronts the aggregates various SPI to provide developers with the ability to map users to their specific system of record.

1.3.2 Authentication SPI

Authentication SPI Overview

The authentication SPI provides the implementation for managing user principals and their credentials and provides the underlying `UserManager` coarsed service implementation.

The authentication SPI also provides a mechanism for managing users across multiple datastore. The class diagram below describes how the authentication SPI relates to the `UserManager` .



Authentication SPI Components

The authentication SPI implements the following components:

Component	Description
`AuthenticationProviderProxy`	A proxy to the various `AuthenticationProvider` implementations. The `AuthenticationProviderProxy` is responsible of invoking the correct `AuthenticationProvider` to authenticate or manage a specific user against a specific data store.
`AuthenticationProvider`	Exposes a specific authentication and user management services implementation. Jetspeed 2 provides 2 implementations: RDBMS and LDAP. Multiple authentication providers can be provided through configuration. For more information, see the security providers configuration.

Component	Description
`CredentialHandler`	See security-spi-atn.xml configuration.
`UserSecurityHandler`	See security-spi-atn.xml configuration.

1.3.3 Credentials Management

Credentials Management Overview

DefaultCredentialHandler Features

With the Jetspeed `DefaultCredentialHandler` special management of password credentials can easily be configured. Through the provided `PasswordCredentialProvider` and `InternalPasswordCredentialInterceptor` components custom logic can be plugged in for:

- providing a custom `PasswordCredential` implementation
- password encoding
  If an `CredentialPasswordEncoder` is available from the `PasswordCredentialProvider` passwords will be encoded with it before they are persisted. The provided `MessageDigestCredentialPasswordEncoder` uses `MessageDigest` hash algorithms for the password encryption, and can for example be configured to use `SHA-1` and `Base64`.
- enforcing password value rules
  If an `CredentialPasswordValidator` is available from the `PasswordCredentialProvider`, passwords will be validated with it before they are persisted. The `DefaultCredentialPasswordValidator` for example enforces non-emtpy password. And with the `SimpleCredentialPasswordValidator` a minimum length and a minum number of numeric characters can be enforced.
- intercepting `InternalCredential` lifecycle events
  If the `DefaultCredentialHandler` is provided with an `InternalPasswordCredentialInterceptor`, it will invoke this interceptor (or an arbirary set if `InternalPasswordCredentialInterceptorsProxy` is used) on:
  - after loading a credential from the persistent store
  - after authenticating a user
  - before a new credential is saved to the persistent store
  - before a new password is save for the credential

  Jetspeed already provides a basic set of interceptors, ready to be used:
  - `ValidatePasswordOnLoadInterceptor`
    This interceptor can be used to validate (pre)set passwords in the persistent store and force a required change by the user if invalid. It uses the configured `CredentialPasswordValidator` of the `PasswordCredentialProvider`, the same as used when a password is changed.
  - `EncodePasswordOnFirstLoadInterceptor`
    This interceptor can be used if passwords needs to be preset in the persistent store or migrated unencoded from a different store. With this interceptor, these cleartext password will

automatically be encoded the first time they are loaded from the database, using the `CredentialPasswordEncoder` from the `PasswordCredentialProvider`

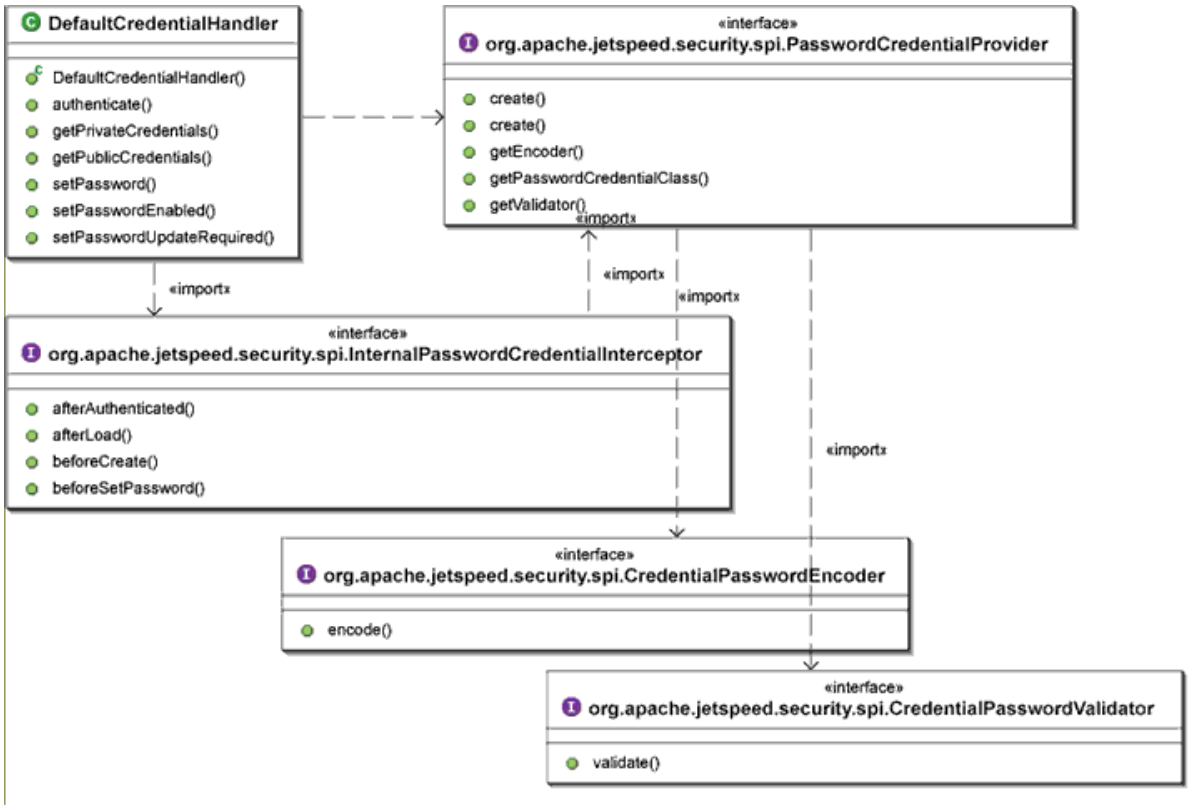
- `PasswordExpirationInterceptor`
  This interceptor can be used to enforce a maximum lifespan for passwords. It manages the `expiration_date` and `is_expired` members of the `InternalCredential` and sets the expired flag when on authentication of a user its (valid) password is expired. The authentication will then fail.
  Note: A Jetspeed pipeline Valve, the `PasswordCredentialValveImpl` can be used to request or even enforce users to change their password in time to prevent a password expiration (described further below).
- `MaxPasswordAuthenticationFailuresInterceptor`
  This interceptor can be used to prevent password hacking by enforcing a maximum number of invalid password attempts in a row. Once this number of authentication failures is reached, the credential will be disabled. On a successful authentication though, this count will automatically be reset to zero again by the `DefaultCredentialHandler`.
- `PasswordHistoryInterceptor`
  This interceptor can be used to enforce usage of unique new passwords in respect to a certain number of previous used passwords. When a new password is set, the current password is saved in a FIFO stack of used passwords. When a user itself changes its password, it must be different from all the onces thus saved, otherwise a `PasswordAlreadyUsedException` will be thrown. But setting a new password through the administrative interface still allows any password (when otherwise valid) to be set.

The `DefaultCredentialHandler` only supports one interceptor to be configured. But, with the `InternalPasswordCredentialInterceptorsProxy` , a list of interceptors can be configured which then will be invoked sequentially.

Jetspeed comes out of the box with several of these interceptors configured, and its very easy to change and extend.See the security-spi-atn.xml section in the Security Services Configuration document for a description of the default configuration. Also provided there is an example how to setup the interceptors to restore the "old" (and much more restrict) configuration provided with the 2.0-M3 release and earlier.

Credentials Management Implementation

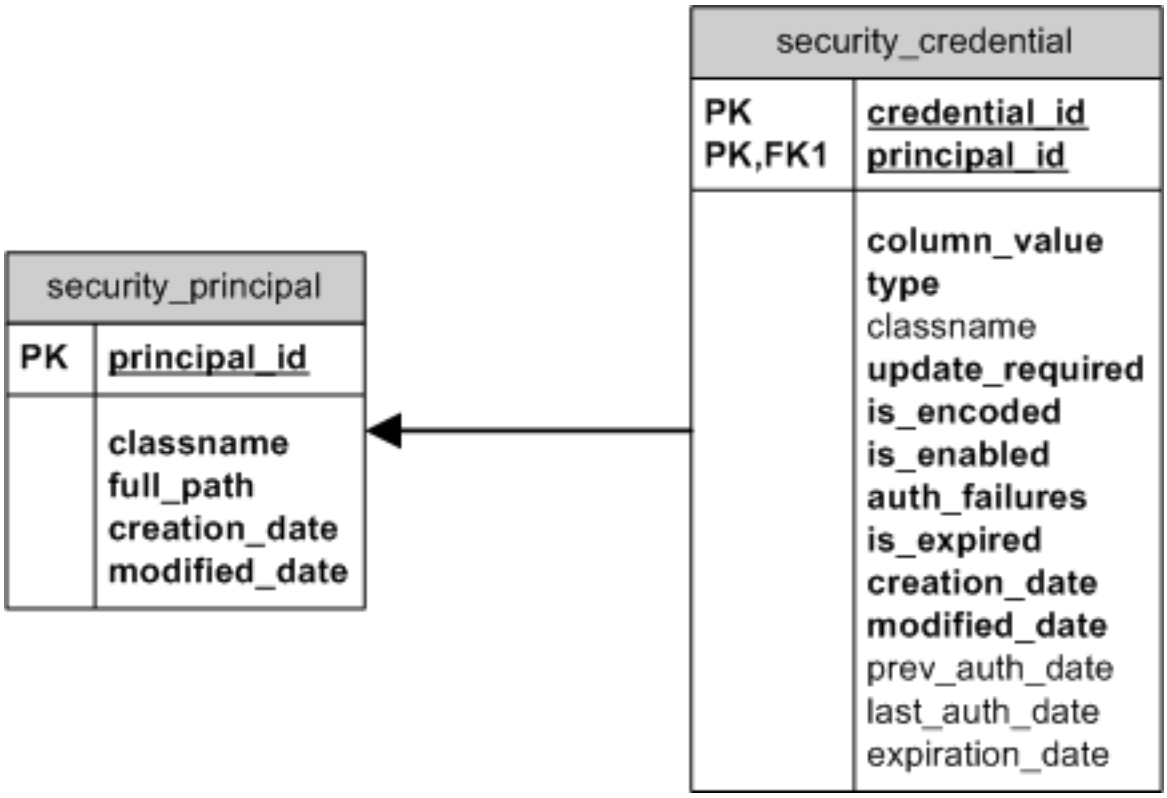
The class diagram below describes the components used for the `DefaultCredentialHandler` implementation.



The OJB mappings for the default credentials implementation are described in `security_repository.xml`:

- `InternalCredential`: Maps to the SECURITY_CREDENTIAL table.

The following database schema is used to stored credentials and their associations to principals.



User interaction

Although the `DefaultCredentialHandler` provides fine-grained management of credentials, it cannot provide direct feedback to the user like presenting a warning that the current password is soon to be expired. But, special request processing pipeline valves provided with jetspeed allow to do just that.

The configuration for these valves can be found and set in the `pipelines.xml` spring configuration file.

LoginValidationValveImpl

The `LoginValidationValveImpl` provides feedback to the user about the cause of an failed login attempt.

It retrieves the `UserPrincipal` and its current `PasswordCredential` for the specified user name, and (if found) determines an specific error code based on its state. This error code is communicated back to through the session so an appropriate error message can be presented to the user.

The following possible error codes can be returned (all defined in the `LoginConstants` interface):

1. ERROR_UNKNOWN_USER
2. ERROR_INVALID_PASSWORD
3. ERROR_USER_DISABLED
4. ERROR_FINAL_LOGIN_ATTEMPT

5. ERROR_CREDENTIAL_DISABLED
6. ERROR_CREDENTIAL_EXPIRED

Of the above error codes, the `ERROR_FINAL_LOGIN_ATTEMPT` will only be reported if the valve is configured with the same `maxNumberOfAuthenticationFailures` value as used for the related `MaxPasswordAuthenticationFailuresInterceptor` described above:

```
  <bean id="loginValidationValve"
        class="org.apache.jetspeed.security.impl.LoginValidationValveImpl"
        init-method="initialize">
    <!-- maxNumberOfAuthenticationFailures
         This value should be in sync with the value for
org.apache.jetspeed.security.spi.impl.MaxPasswordAuthenticationFailuresInterceptor
         (if used) to make sense.
         Any value < 2 will suppress the LoginConststants.ERROR_FINAL_LOGIN_ATTEMPT
         error code when only one last attempt is possible before the credential
         will be disabled after the next authentication failure.
    -->
    <constructor-arg index="0"><value>3</value></constructor-arg>
</bean>
```

PasswordCredentialValveImpl

The `PasswordCredentialValveImpl` is meant to be used together with a special Portlet on a special Portal Page (PSML) to automatically request or even require a user to change its password.

This valve evaluates `PasswordCredential.isUpdateRequired()` and optionally the `expirationDate`, `lastAuthenticationDate` and `previousAuthenticationDate` fields to determine if a user is required or just be asked to change its password.

This valve can optionally be configured with a list of `expirationWarningDays` numbers in its constructor:

```
<bean id="passwordCredentialValve"
      class="org.apache.jetspeed.security.impl.PasswordCredentialValveImpl"
      init-method="initialize">
 <constructor-arg>
   <!-- expirationWarningDays -->
   <list>
     <value>2</value>
     <value>3</value>
     <value>7</value>
   </list>
 </constructor-arg>
</bean>
```

These numbers each represent a day before the current `expirationDate` of the password credential when a user should be warned its password is soon to expire and be asked to change it. The `lastAuthenticationDate` and the `previousAuthenticationDate` are used to determine when

this should happen. It will be done only once for each configured `expirationWarningDay`. If a user logs on for the first time (after several days) with the above example configuration, 6 days before the password expires, he or she will be warned about it. And again when 3 or 2 days are left.

When a user logs on the last day before the password expires *or* when `updateRequired` is `true`, the user will be required to change the password, regardless if expirationWarningDays are configured or not.

To be able to automatically provide the user with this information and allow or require the password to be changed directly after login, a special `ProfileLocator` `SECURITY_LOCATOR` is used. The `PageProfilerValve` (which should be configed *after* this valve in the pipeline) will then use this enforced locator to be used to find the related portal page to present to the user.

For this to work, a `"security"` Profiler rule must have been setup like the default one provided by Jetspeed:

As can seen from the above image, the default page which will be presented to the user is the `/my-account.psml` located in the root.

This default page contains only one portlet, the `ChangePasswordPortlet` from the security Portlet Application.

The `ChangePasswordPortlet` works together with the `PasswordCredentialValveImpl` as it checks for the `PASSWORD_CREDENTIAL_DAYS_VALID_REQUEST_ATTR_KEY` request parameter which will be set by this valve with the number of days the password is still valid. For a required password change this will be set to Integer(0).

The default `my-account.psml` page contains *only* the `ChangePasswordPortlet` to make sure a user which is *required* to change the password cannot interact with the portal any other way then after the password is changed.

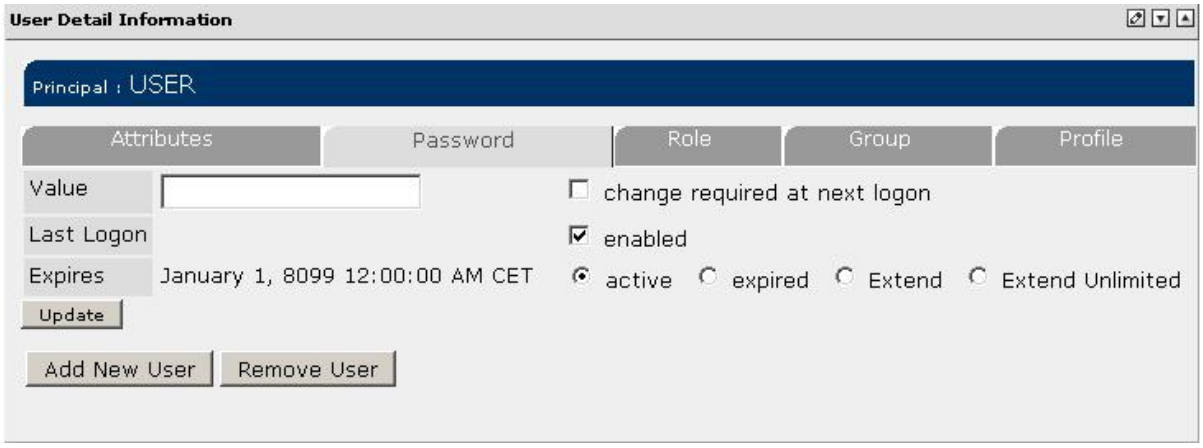
Although the user might be attempted to select a link to a different page (from a portal menu for exampl), this valve will make sure only the configured "security" locator page is returned if it is required. But, once the password is changed the then targeted page in the url will be navigated to automatically.

Managing Password Expiration

If the `PasswordExpirationInterceptor` is used, password expiration for a certain user can be directly managed through the `UserDetailPortlet` provided with the `security` portlet application.

If enabled, this portlet can display the current expiration date of a password and also allows to change its

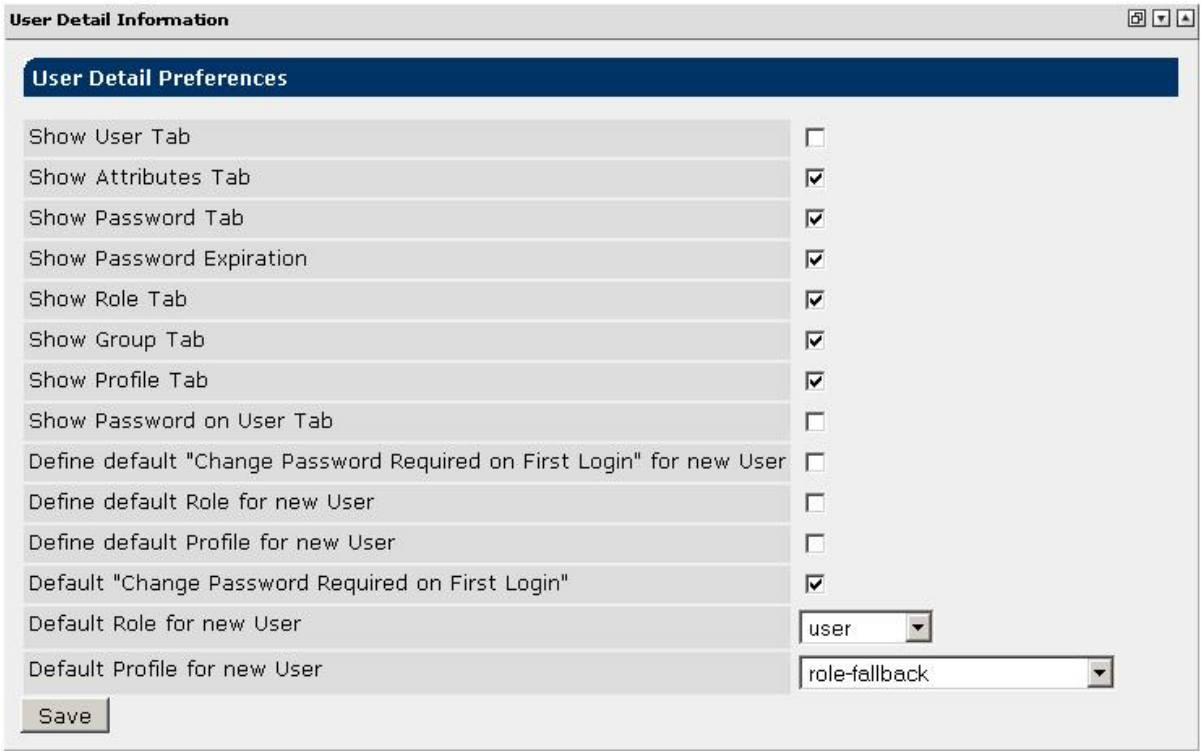
value:



As you can see, through the radio group, the password expiration date can be changed to:

Action	Expires
Expired	today
Extend	today + `maxLifeSpanInDays` as configured for the PasswordExpirationInterceptor
Extend Unlimited	January 1, 8099 (the maximum value allowed for java.sql.Date)

This feature can be enabled through the edit/preferences page of the `UserDetailsPortlet`:



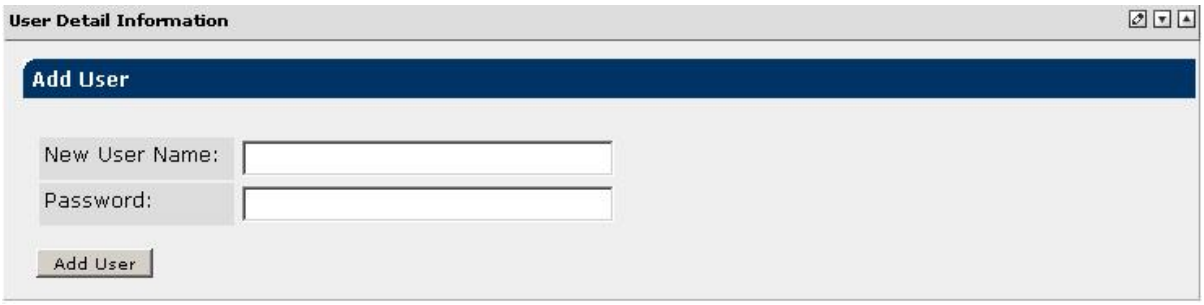
Note: when a new password value is specified selected password expiration action `Expired` will be ignored!

Setting default 'Change Password required on First Login'

Through the same `UserDetailsPortlet` preferences as show above, the default `updateRequired` property of a password credential for a new user can be configured too.

And, if you always need the same setting for all users, you can even suppress the selection box normally displayed on the `Add User` dialog.

With the preferences set as in the example shown above, the `Add User` dialog will look like this:



A user added with the example preferences set, will have the `updateRequired` property set to true, the `User` role assigned and use the `role-fallback` profiling rule.

1.4 Authorization

Authorization Overview

For auhorization, Jetspeed 2 implements its own java.security.Policy using a relation database store to manage associations between principals and permissions.

The `PermissionManager` provides access to the permissions associated to given principals.

- The JAAS Authorization provides an overview of the authorization aspect of JAAS.
- The PermissionManager Overview documents the `PermissionManager` implementation.

1.4.1 JAAS Authorization

Overview of JAAS Authorization

A good overview of JAAS authorization is provided on Sun's web site . At a high level, JAAS authorization leverages:

- Permission that associates actions to resources.
- Principal that represents an entity in the system. In Jetspeed 2, 3 principals are used to represent users, roles and groups.
- Policy that associates principals to permissions.

Jetspeed 2 provides a custom policy implemention that allow the portal to secure resources as follow:

```
grant principal o.a.j.security.UserPrincipal "theUserPrincipal" {
  permission o.a.j.security.PagePermission "mypage", "view";
  permission o.a.j.security.PortletPermission "myportlet",
"view,edit,minimize,maximize";
  permission o.a.j.security.TabPermission "mytab", "view";
};

grant principal o.a.j.security.RolePrincipal "theRolePrincipal" {
  permission o.a.j.security.PagePermission "mypage", "view";
  permission o.a.j.security.PortletPermission "myportlet",
"view,edit,minimize,maximize";
  permission o.a.j.security.TabPermission "mytab", "view";
};

grant principal o.a.j.security.GroupPrincipal "theGroupPrincipal" {
  permission o.a.j.security.PagePermission "mypage", "view";
  permission o.a.j.security.PortletPermission "myportlet",
"view,edit,minimize,maximize";
  permission o.a.j.security.TabPermission "mytab", "view";
};
```

The custom security policy provides a `java.security.Policy` implementation that stores the association between principals and permissions in a relational database as opposed to leveraging the default JDK policy. In the case of Sun's JDK, the default policy is sun.security.provider.PolicyFile a file based policy.

In the code sample above, the `UserPrincipal` identify with the `Principal.getName()` "theUserPrincipal" has permission to "view" the page called "mypage", to "view,edit,minimize,maximize" the portlet portlet called "myportlet"

The AccessController validates a `Subject` permissions. For instance, a page permission check would perform the following check:

```
PagePermission permission = new PagePermission(path, actions);
AccessController.checkPermission(permission);
```

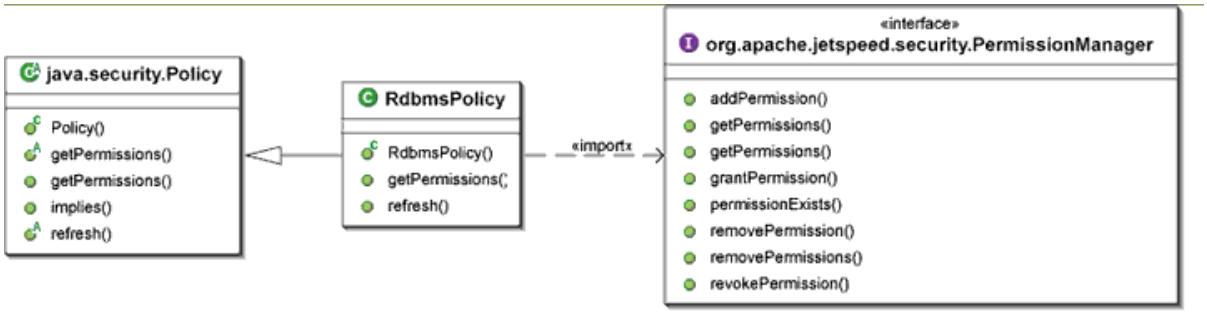
Jetspeed JAAS Policy

The `RdbmsPolicy` implements `java.security.Policy` . It leverages the `PermissionManager` to get the permissions associated with a given `Subject` principals.

```
pms.getPermissions(user.getPrincipals());
```

The class diagram below illustrate the association between the `RdbmsPolicy` and the `PermissionManager` .

A good article on custom policies implementation is available on IBM web site .



To get more detail about the implementation of the `PermissionManager` , see PermissionManager Overview .

Note: The current `RdbmsPolicy` manages the policies to apply. It applies `RdbmsPolicy` in conjunction with the default policy configured in the runtime environment. Jetspeed 2 should explore providing JACC adapters for its custom policy for specific application servers.

Authorization Provider and Policy Configuration

The `AuthorizationProvider` configures the authorization policies to be used by Jetspeed 2 and keeps the list of such policies in the `SecurityPolicies` singleton. The `RdbmsPolicy` when getting the permissions for access control will execute its policy as well as all the policies configured in `SecurityPolicies` . If the `AuthorizationProvider` was constructed with `useDefaultPolicy`

set to true, the default JDK or application server policy will be applied when getting the permissions.

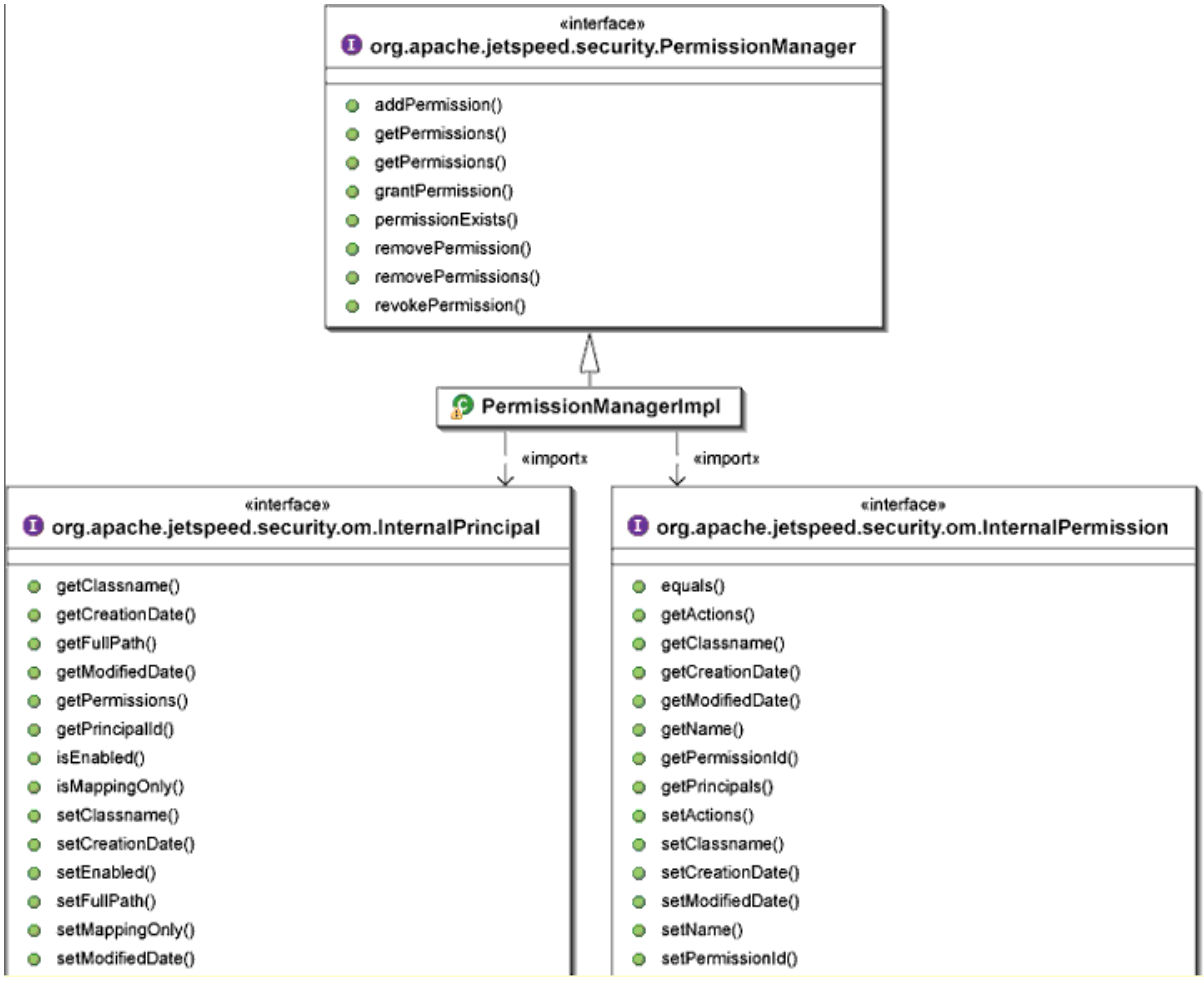
Note: The `RbmsPolicy` permission check is concerned about the principals associated to the `Subject`, therefore where performing an access control check, the check should be performed with the following call: `doAsPrivileged(theSubject, anAction, null)`. By passing a null `AccessContolContext`, the caller is essentially saying: "I don't care who called me, the only important thing is whether I have permission when associated with the given subject".

1.4.2 PermissionManager Overview

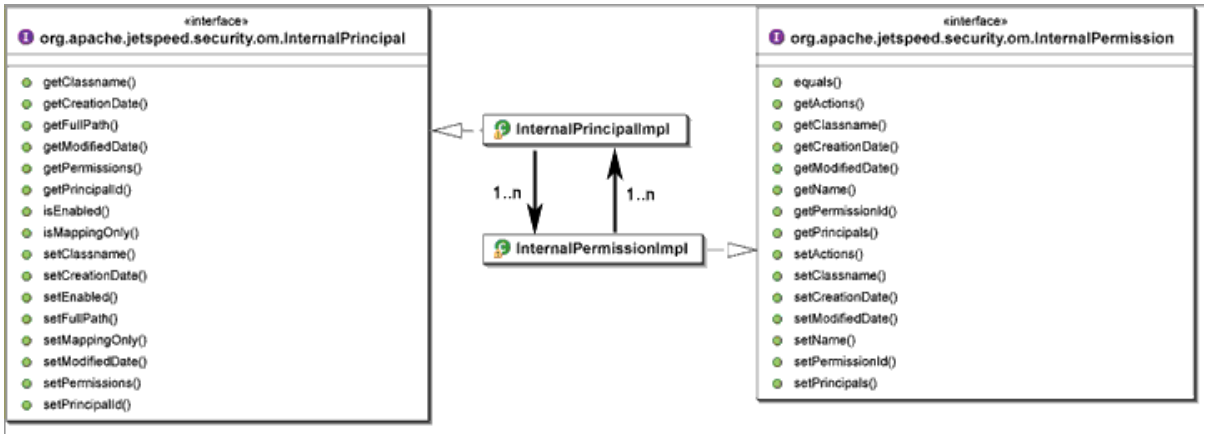
PermissionManager Overview

The `PermissionManager` is used by the `RdbmsPolicy` to get the permissions for a given user principals as presented in the Jetspeed JAAS Policy section of the documentation.

The `PermissionManager` manages the association between permissions and principals. Each permission or principal maps to a generic object model and reflexion is used to instantiate the proper permission or principal type. The class diagram below represents the interfaces representing a generic permission (`InternalPermission`) and a generic principal (`InternalPrincipal`) and their relation to the `PermissionManager`.



Each `InternalPermission` maps to one or more `InternalPrincipal` and, each `InternalPrincipal` can have one or more `InternalPermission`.



Schema and OJB Mapping

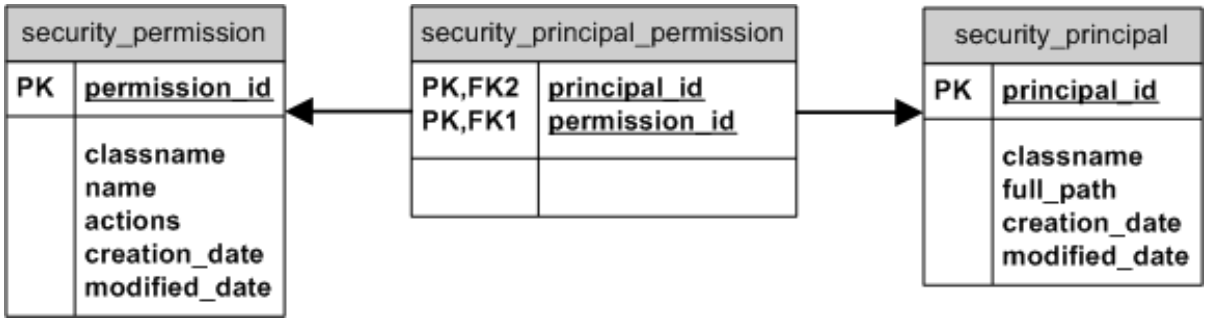
The OJB mappings for the security component are described in `security_repository.xml`:

- `InternalPrincipal`: Maps to the `SECURITY_PRINCIPAL` table.
- `InternalPermission`: Maps to the `SECURITY_PERMISSION` table.
- Associations between `InternalPrincipal` and `InternalPermission` are maintained through the indirection table `PRINCIPAL_PERMISSION`.

```
<class-descriptor
            class="org.apache.jetspeed.security.om.impl.InternalPrincipalImpl"
            proxy="dynamic"
            table="SECURITY_PRINCIPAL"
        >...</class-descriptor>

<class-descriptor
            class="org.apache.jetspeed.security.om.impl.InternalPermissionImpl"
            proxy="dynamic"
            table="SECURITY_PERMISSION"
        >...</class-descriptor>
```

The relational schema maintaining principal to permission associations is provided below:



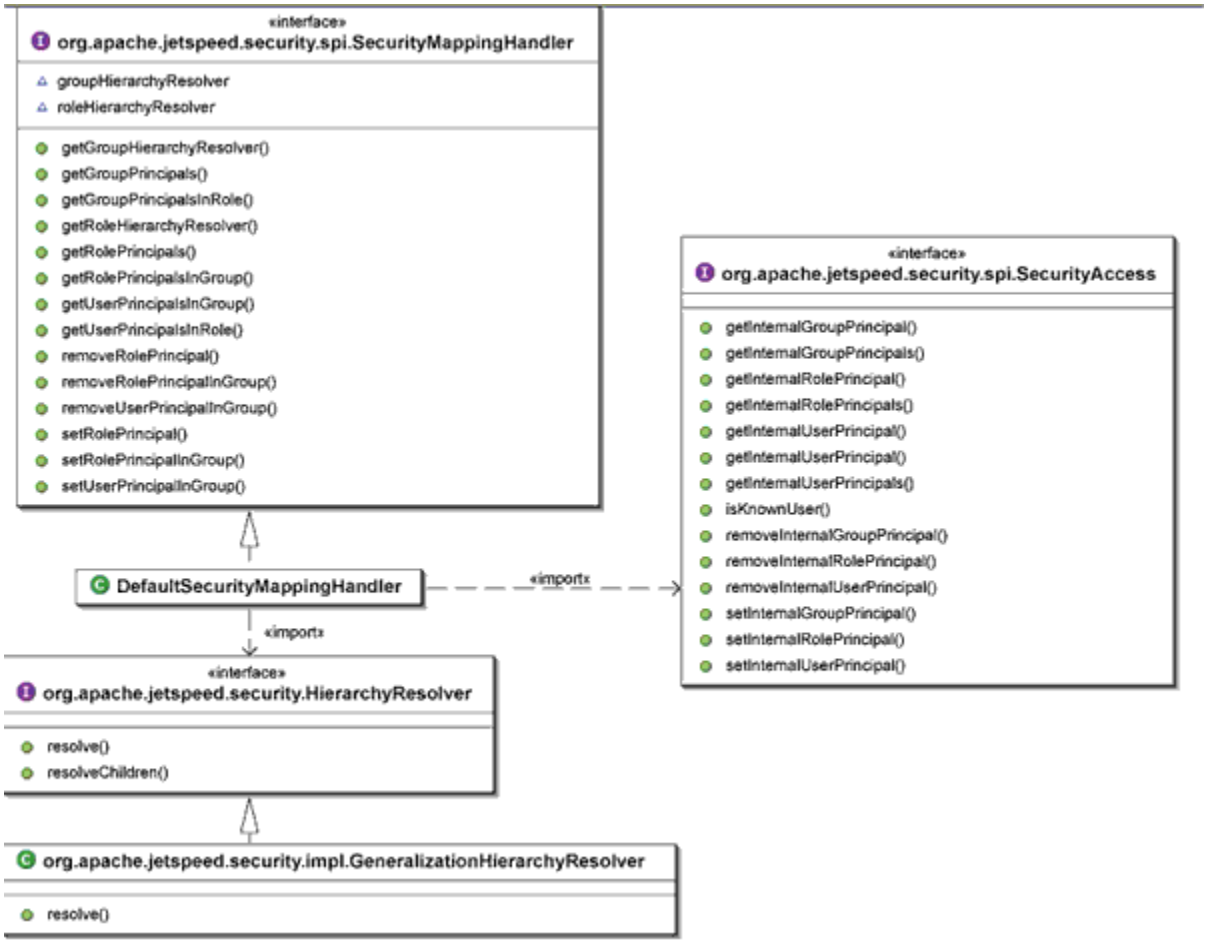
1.4.3 Authorization/Security Mapping SPI

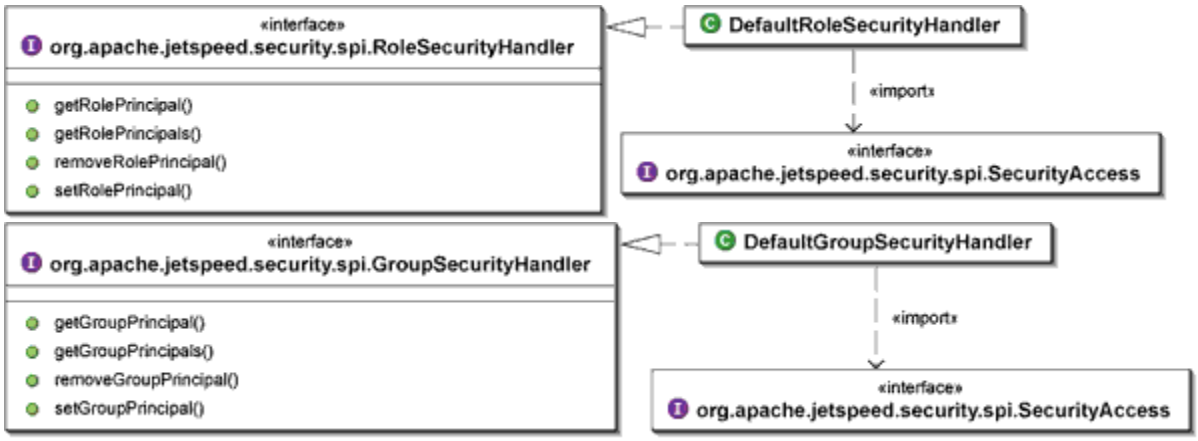
Authorization/Security Mapping SPI Overview

The authorization SPI provides the implementation to support Jetspeed 2 users, roles and groups associations and the roles/groups hierarchy policy. It provides the underlying mechanism to support the implementation of the `RoleManager` and `GroupManager` .

As described in the security overview , Jetspeed support hierarchical role based access control with configurable hierarchy policies.

First, let's have a look at a class diagram of the authorization SPI:





Authorization SPI Components

The authorization SPI implements the following components:

Component	Description
org.apache.jetspeed.security.spi.SecurityMappingHandler	See security-spi-atz.xml configuration.
org.apache.jetspeed.security.HierarchyResolver	See hierarchy management .
org.apache.jetspeed.security.spi.RoleSecurityHandler	See security-spi-atz.xml configuration.
org.apache.jetspeed.security.spi.GroupSecurityHandler	See security-spi-atz.xml configuration.

1.4.4 Hierarchy Management

Hierarchy Management Overview

Two hierarchy resolution strategies are supported for authorization decisions:

- Hierarchy resolution by Generalization: This is the default hierarchy resolution in Jetspeed. If a hierarchy uses a generalization strategy, each role is more general than the previous one. For instance, if a user has the role [roleA.roleB.roleC] then `user.getSubject().getPrincipals()` returns:
  - /role/roleA
  - /role/roleA/roleB
  - /role/roleA/roleB/roleC
- Hierarchy resolution by Aggregation: If a hierarchy uses a aggregation strategy, the higher role is responsible for a superset of the activities of the lower role. For instance, if the following roles are available:
  - roleA
  - roleA.roleB
  - roleA.roleB.roleC

  If a user has the role [roleA] then, `user.getSubject().getPrincipals()` returns:
  - /role/roleA
  - /role/roleA/roleB
  - /role/roleA/roleB/roleC

As described in the authorization SPI section , the `SecurityMappingHandler` is configured with a specific hierarchy strategy for group and role hierarchy management. See the authorization SPI configuration for a configuration example.

Leveraging Preferences to Manage Hierarchies

The default hierarchy management implementation resolves the hierarchy strategy by leveraging Jetspeed 2's java.util.prefs.Preferences implementation. The `Preferences` implementation provides the underlying structure in Jetspeed to store user attributes, and roles and groups definitions. The `Preferences` model provides a hierarchy model that is leveraged to store the base roles and groups hierarchy upon which various resolving strategies can be applied (resolution by generalization or aggregation).

See Jetspeed 2 Preferences implementation section for more information.

How does this work?

The `SecurityMappingHandler` implementation resolves the mappings between roles and groups. Let's say that we want to find out the roles mapping to a specific group name. To do so, the `SecurityMappingHandler` implements a `getRolePrincipalsInGroup(String groupFullPathName)` method. In this method, the group name is mapped to a specific `Preferences` node. According to a given hierarchy resolution strategy (see overview section), being in [group A] may mean belonging to a set of groups; the HierarchyResolver is used to do so as illustrated below:

```
public Set getRolePrincipalsInGroup(String groupFullPathName)
{
   ...
   Preferences preferences = Preferences.userRoot().node(
       GroupPrincipalImpl.getFullPathFromPrincipalName(groupFullPathName));
   String[] fullPaths = groupHierarchyResolver.resolve(preferences);
   ...
}
```

The resulting groups are then used to find all associated roles.

As a result of this implementation, the name of a role principal (`Principal` getName()) in the security layer should match the full path of that user preferences root in the preferences layer (`Preference` absolutePath() ; e.g: `/role/theRolePrincipal`).

Group and roles hierarchy are stored in the `Preferences` layer as follow (the output of exportNode() for Jetspeed's RBMS Preferences implementation):

```
<preferences EXTERNAL_XML_VERSION="1.0">
<root type="user">
<map />
    <node name="group1">
    <map />
        <node name="groupid1.1">
        <map />
            <node name="groupid1.1.1">
            <map />
            </node>
        </node>
    </node>

    <node name="role1">
    <map />
        <node name="roleid1.1">
        <map />
            <node name="roleid1.1.1">
            <map />
            </node>
        </node>
    </node>
</root>
```

This structure would define the following group and role hierarchy:

- `/group1/groupid1.1/groupid1.1.1`
- `/role1/roleid1.1/roleid1.1.1`

Additionally, in this model, the `map` element can define groups or roles custom properties. For instance, a role could have a rule custom property (or a pointer to a rule) that allow rule based role definition tied to some rule engine (Drools for instance) and is validated when the isInRole method is invoked. For groups, a portal could use group to describe organization and have custom property such as address, city, etc. associated with the organization/group.

1.5 High Level Security Services

High Level Security Services Overview

Jetspeed 2 provides the four following high level security services:

- `UserManager` : Service providing user management capabilities.
- `GroupManager` : Service providing group management capabilities.
- `RoleManager` : Service providing role management capabilities.
- `PermissionManager` : Service providing permission management capabilities.

Using High Level Security Services in Portlets

In order to access Jetspeed high level security services in your portlets, Jetspeed provide a custom extension to the `portlet.xml` metadata. All Jetspeed custom metadata is located in the `jetspeed-portlet.xml` configuration file in the `WEB-INF` folder of the portlet application. The custom `js:services` tag provides the ability to expose portal services to a portlet through the `javax.portlet.PortletContext`.

Jetspeed portal services are configured in the spring assembly file located in the portal `WEB-INF/assembly/jetspeed-services` configuration file. The UserManager for instance is configured as follow:

```
<!-- Portlet Services  -->
<bean id="PortalServices"
          class="org.apache.jetspeed.services.JetspeedPortletServices" >
   <constructor-arg>
      <map>
               ...
               <entry key="UserManager">
                        <ref bean="org.apache.jetspeed.security.UserManager"/>
                     </entry>
                     ...
      </map>
   </constructor-arg>
</bean>
```

The `UserManager` services is then available to be loaded in a specific portlet `PortletContext`. Portlet developers need to specify the portal services they would like to use. The following example shows how to expose the portal `UserManager` to a portlet application:

```
<js:services>
```

```
   <js:service name='UserManager'/>
</js:services>
```

Once a portal service is loaded in the portlet context, the portlet implementation (which typically extends `javax.portlet.GenericPortlet`) can access the service as follow:

```
PortletContext context = getPortletContext();
userManager = (UserManager)
context.getAttribute(CommonPortletServices.CPS_USER_MANAGER_COMPONENT);
```

where `CommonPortletServices.CPS_USER_MANAGER_COMPONENT = "cps:UserManager"`

1.6 Security Services Configuration

Default configuration

Jetspeed 2 default security services configuration leverages a relational database as its default persitent datastore for security information. Jetspeed 2 security service provider interface provides a mechanism to replace the default datastore configured.

3 files are involved when configuring Jetspeed 2 security SPI. All the SPI configuration files are located under *${jetspeed-source-home}/portal/src/webapp/WEB-INF/assembly/* .

security-atn.xml

This configuration file provides the login module configuration. Not everyone needs this, as some application may decide to use another login module other than the one provided.

security-atz.xml

This configuration file configures the authorization policy, in J2's case RdbmsPolicy .

security-managers.xml

This configuration file configures all the managers for security purpose.

security-providers.xml

This configuration file configures the various providers and weaves the SPI together.

- `AuthenticationProviderProxy` : Configures the list of `AuthenticationProvider` and the default authenticator.

```
<bean id="org.apache.jetspeed.security.AuthenticationProviderProxy"
   class="org.apache.jetspeed.security.impl.AuthenticationProviderProxyImpl">
   <constructor-arg >
      <list>
         <ref bean="org.apache.jetspeed.security.AuthenticationProvider"/>
      </list>
   </constructor-arg>
  <constructor-arg><value>DefaultAuthenticator</value></constructor-arg>
</bean>
```

- `AuthenticationProvider` : Configures the authentication providers for the current portal implementation. The example below configures the default authenticator that uses the RDBMS to manage/store user information.

```
<bean id="org.apache.jetspeed.security.AuthenticationProvider"
class="org.apache.jetspeed.security.impl.AuthenticationProviderImpl">
   <constructor-arg
index="0"><value>DefaultAuthenticator</value></constructor-arg>
   <constructor-arg index="1"><value>The default
authenticator</value></constructor-arg>
   <constructor-arg index="2"><value>login.conf</value></constructor-arg>
   <constructor-arg index="3">
      <ref bean="org.apache.jetspeed.security.spi.CredentialHandler"/>
   </constructor-arg>
   <constructor-arg index="4">
      <ref bean="org.apache.jetspeed.security.spi.UserSecurityHandler"/>
   </constructor-arg>
</bean>
```

- `AuthorizationProvider` : Configures the policies and instantiates the `SecurityPolicies` that are used for enforcing permissions. By default, Jetspeed 2 does not load any other security policies that may have been configured. In order to use default policies, set `useDefaultPolicy` to `true`

```
<bean id="org.apache.jetspeed.security.AuthorizationProvider"
            class="org.apache.jetspeed.security.impl.AuthorizationProviderImpl">
    <constructor-arg index="0">
        <ref bean="org.apache.jetspeed.security.impl.RdbmsPolicy"/>
    </constructor-arg>
    <!-- Does not use the default policy as a default behavior -->
    <constructor-arg index="1"><value>false</value></constructor-arg>
</bean>
```

security-spi.xml

This configuration file contains configuration that are common to the authentication and authorization SPIs.

Bean	Description
org.apache.jetspeed.security.spi.SecurityAccess	Used internally by the default OJB based SPI. Provide access to common action/methods for the various SPI implementations. The *SecurityAccess* bean is used by both the Authentication and Authorization SPIs.

security-spi-atn.xml

This configuration file contains all the configurations for configuring the authentication SPI.

Bean	Description
org.apache.jetspeed.security.spi.CredentialHandler	The *CredentialHandler* encapsulates the operations involving manipulation of credentials. The default implementation provides support for password protection as defined by the *PasswordCredentialProvider* ; as well as lifecycle management of credentials through *InternalPasswordCredentialInterceptor* which can be configured to manages parameters such as maximum number of authentication failures, maximum life span of a credential in days and how much history to retain for a given credential.
org.apache.jetspeed.security.spi.UserSecurityHandler	The *UserSecurityHandler* encapuslated all the operations around the user principals.

The following simple `CredentialHandler` configuration is currently provided by default with Jetspeed:

```
<!-- require a non-empty password -->
<bean id="org.apache.jetspeed.security.spi.CredentialPasswordValidator"
class="org.apache.jetspeed.security.spi.impl.DefaultCredentialPasswordValidator"/>

<!-- MessageDigest encode passwords using SHA-1 -->
<bean id="org.apache.jetspeed.security.spi.CredentialPasswordEncoder"
class="org.apache.jetspeed.security.spi.impl.MessageDigestCredentialPasswordEncoder">
     <constructor-arg index="0"><value>SHA-1</value></constructor-arg>
</bean>

<!-- allow multiple InternalPasswordCredentialInterceptors to be used for
DefaultCredentialHandler -->
<bean id="org.apache.jetspeed.security.spi.InternalPasswordCredentialInterceptor"
class="org.apache.jetspeed.security.spi.impl.InternalPasswordCredentialInterceptorsProxy">
     <constructor-arg index="0">
       <list>
         <!-- enforce an invalid preset password value in the persisent store is
required to be changed -->
         <bean
class="org.apache.jetspeed.security.spi.impl.ValidatePasswordOnLoadInterceptor"/>

         <!-- ensure preset cleartext passwords in the persistent store  will be
encoded on first use -->
         <bean
class="org.apache.jetspeed.security.spi.impl.EncodePasswordOnFirstLoadInterceptor"/>
       </list>
     </constructor-arg>
</bean>

<bean id="org.apache.jetspeed.security.spi.PasswordCredentialProvider"
class="org.apache.jetspeed.security.spi.impl.DefaultPasswordCredentialProvider">
     <constructor-arg index="0">
       <ref bean="org.apache.jetspeed.security.spi.CredentialPasswordValidator"/>
     </constructor-arg>
     <constructor-arg index="1">
       <ref bean="org.apache.jetspeed.security.spi.CredentialPasswordEncoder"/>
     </constructor-arg>
</bean>
```

```
<bean id="org.apache.jetspeed.security.spi.CredentialHandler"
     class="org.apache.jetspeed.security.spi.impl.DefaultCredentialHandler">
     <constructor-arg index="0">
       <ref bean="org.apache.jetspeed.security.spi.SecurityAccess"/>
     </constructor-arg>
     <constructor-arg index="1">
       <ref bean="org.apache.jetspeed.security.spi.PasswordCredentialProvider"/>
     </constructor-arg>
     <constructor-arg index="2">
       <ref
bean="org.apache.jetspeed.security.spi.InternalPasswordCredentialInterceptor"/>
     </constructor-arg>
</bean>
```

The above configuration requires not much more than that a password should not be empty and MessageDigest encode it using SHA-1.

Before the 2.0-M4 release, Jetspeed came configured with a much stricter configuration, but for first time users of the Portal this was a bit overwelming and also quite difficult to configure differently.

With the 2.0-M4 release, the previously provided, and rather complex, `InternalPasswordCredentialInterceptor` implementations are split up in single atomic interceptors which can much easier be configured indepedently.

An overview of the new interceptors and how related request processing pipeline valves can be configured to provide feedback to the user is provided in the Credentials Management document.

Since the "old" (pre 2.0-M4) interceptors are no longer provided with Jetspeed, the example below shows how to "restore" the old setup using the new interceptors:

```
<!-- require a password of minimum length 6 and at least two numeric characters -->
<bean id="org.apache.jetspeed.security.spi.CredentialPasswordValidator"
class="org.apache.jetspeed.security.spi.impl.SimpleCredentialPasswordValidator">
     <constructor-arg index="0"><value>6</value></constructor-arg>
     <constructor-arg index="1"><value>2</value></constructor-arg>
</bean>

<!-- allow multiple InternalPasswordCredentialInterceptors to be used for
DefaultCredentialHandler -->
<bean id="org.apache.jetspeed.security.spi.InternalPasswordCredentialInterceptor"
class="org.apache.jetspeed.security.spi.impl.InternalPasswordCredentialInterceptorsProxy">
     <constructor-arg index="0">
       <list>
         <!-- enforce an invalid preset password value in the persisent store is
required to be changed -->
         <bean
class="org.apache.jetspeed.security.spi.impl.ValidatePasswordOnLoadInterceptor"/>

         <!-- ensure preset cleartext passwords in the persistent store  will be
encoded on first use -->
         <bean
class="org.apache.jetspeed.security.spi.impl.EncodePasswordOnFirstLoadInterceptor"/>

         <!-- remember the last 3 passwords used and require a new password to be
```

```
different from those -->
         <bean
class="org.apache.jetspeed.security.spi.impl.PasswordHistoryInterceptor">
           <constructor-arg index="0"><value>3</value></constructor-arg>
         </bean>

         <!-- Automatically expire a password after 60 days -->
         <bean
class="org.apache.jetspeed.security.spi.impl.PasswordExpirationInterceptor">
           <constructor-arg index="0"><value>60</value></constructor-arg>
         </bean>

         <!-- Automatically disable a password after 3 invalid authentication
attempts in a row -->
         <bean
class="org.apache.jetspeed.security.spi.impl.MaxPasswordAuthenticationFailuresInterceptor">
           <constructor-arg index="0"><value>3</value></constructor-arg>
         </bean>
       </list>
     </constructor-arg>
</bean>
```

And, make sure something like the following configuration is set for the security related valves in pipelines.xml:

```
<bean id="passwordCredentialValve"
      class="org.apache.jetspeed.security.impl.PasswordCredentialValveImpl"
      init-method="initialize">
 <constructor-arg>
   <!-- expirationWarningDays -->
   <list>
     <value>2</value>
     <value>3</value>
     <value>7</value>
   </list>
 </constructor-arg>
</bean>

<bean id="loginValidationValve"
      class="org.apache.jetspeed.security.impl.LoginValidationValveImpl"
      init-method="initialize">
  <!-- maxNumberOfAuthenticationFailures
       This value should be in sync with the value for
org.apache.jetspeed.security.spi.impl.MaxPasswordAuthenticationFailuresInterceptor
       (if used) to make sense.
       Any value < 2 will suppress the LoginConststants.ERROR_FINAL_LOGIN_ATTEMPT
       error code when only one last attempt is possible before the credential
       will be disabled after the next authentication failure.
  -->
  <constructor-arg index="0"><value>3</value></constructor-arg>
</bean>
```

Also, make sure the above valves are configured in the `jetspeed-pipeline` bean.

See the User Interaction section in the Credentials Management document for a description of these valves and their relation to the interceptors configuration.

security-spi-atz.xml

This configuration file contains all the configurations for configuring the authorization SPI.

Bean	Description
org.apache.jetspeed.security.spi.RoleSecurityHandler	The *RoleSecurityHandler* encapsulates all the operations around the role principals.
org.apache.jetspeed.security.spi.GroupSecurityHandler	The *GroupSecurityHandler* encapsulates all the operations around the group principals.
org.apache.jetspeed.security.spi.SecurityMappingHandler	The *SecurityMappingHandler* encapsulates all the operations involving mapping between principals. It contains the logic managing hierarchy resolution for hierarchical principals (roles or groups). The default hierarchy resolution provided is a hierarchy by generalization (see overview for definitions). A *contructor-arg* can be added to the *SecurityMappingHandler* to change the hierarchy resolution strategy. Jetspeed 2 also support a hierarchy resolution by aggregation.

A sample `SecurityMappingHandler` configuration could be:

```
<!-- Security SPI: SecurityMappingHandler -->
<bean id="org.apache.jetspeed.security.spi.SecurityMappingHandler"
      class="org.apache.jetspeed.security.spi.impl.DefaultSecurityMappingHandler">
   <constructor-arg >
      <ref bean="org.apache.jetspeed.security.spi.SecurityAccess"/>
   </constructor-arg>
   <!-- Default role hierarchy strategy is by generalization.
        Add contructor-arg to change the strategy. -->
   <!-- Default group hierarchy strategy is by generalization.
        Add contructor-arg to change the strategy. -->
</bean>
```

1.7 LDAP Configuration

LDAP Configuration

This document attempts to document the configuration of the LDAP security module in Jetspeed. Out of the box, Jetspeed searches for user, group & role information in a relational database. However, it can also search this information in an LDAP directory.

Jetspeed stores its LDAP configuration in a Spring XML file called security-spi-ldap.xml

This XML file describes an object (used internally by Jetspeed) that contains LDAP configuration parameters. These configuration parameters are passed onto the object through constructor arguments:

```
<!-- The LDAP initial context factory. -->
<constructor-arg index="0">
  <value>com.sun.jndi.ldap.LdapCtxFactory</value>
</constructor-arg>
```

Each constructor argument contains an index to specify the correct order. The file defines the following arguments:

Index	Name	Example
0	Initial context factory	com.sun.jndi.ldap.LdapCtxFactory
1	LDAP server host	localhost
2	LDAP server port	389
3	Root context	o=sevenSeas
4	The LDAP server root dn	uid=admin,o=sevenSeas
5	The LDAP server root password	secret
6	The roles filter	(objectclass=groupOfUniqueNames))
7	The groups filter	(objectClass=groupOfNames)
8	The user filter	(objectclass=inetorgperson)
9	roleMembershipAttributes	uniqueMember
10	userRoleMembershipAttributes	
11	groupMembershipAttributes	member
12	userGroupMembershipAttributes	

Index	Name	Example
13	groupMembershipForRoleAttributes	uniqueMember
14	roleGroupMembershipForRoleAttributes	
15	defaultSearchBase	
16	roleFilterBase	ou=Roles,ou=OrgUnit1
17	groupFilterBase	ou=Groups,ou=OrgUnit1
18	userFilterBase	ou=People,ou=OrgUnit1
19	roleObjectClasses	top,groupOfUniqueNames
20	groupObjectClasses	top,groupOfNames
21	userObjectClasses	top,person,organizationalPerson,inetorgperson
22	roleIdAttribute	cn
23	groupIdAttribute	cn
24	userIdAttribute	uid
25	UidAttribute	uid
26	MemberShipSearchScope	1
27	roleUidAttribute	cn
28	groupUidAttribute	cn
29	userUidAttribute	uid
30	roleObjectRequiredAttributeClasses	uniqueMember
31	groupObjectRequiredAttributeClasses	member
32	userAttributes	sn={u},cn={u}
33	roleAttributes	sn={u}
34	groupAttributes	sn={u}
35	userPasswordAttribute	passWord
36	knownAttributes	cn,sn,o,uid,ou,objectClass,userPassword,member,uniqueMember,mem

Configuring Jetspeed 2 to Use LDAP

Configuring jetspeed for LDAP usage is simply a matter of having the proper configuration files in place. These configuration files are to be placed in the `WEB-INF/assembly` folder of the expanded jetspeed WAR.

The following files need to be copied into that directory if you want to connect Jetspeed2 to an LDAP server.

- *security-spi-ldap.xml :* Provides the configuration information for LDAP binding, explained in detail below.

- *security-spi-ldap-atn.xml :* Provides the SPI configuration for authentication. It replaces the default implementations of the *CredentialHandler* and *UserSecurityHandler* with an LDAP specific implementation.
- *security-spi-ldap-atz.xml :* Provides the SPI configuration for authorization. It replaces the default implementations of the *RoleSecurityHandler*, *GroupSecurityHandler* and *SecurityMappingHandler* with an LDAP specific implementation.

The default authentication and authorization SPI configurations (the files called **`security-spi-atn.xml`** and **`security-spi-atz.xml`**) need to be removed from that assembly directory.

In the Jetspeed source tree the examples ldap configuration files can be found in:

```
${jetspeed-source-home}/components/security/etc/
```

If your application is deployed in Tomcat, the target assembly directory is located at:

```
${tomcat-home}/webapps/jetspeed/WEB-INF/assembly/
```

Furthermore, the source tree of the Jetspeed security component provides several tests using different configurations as well as ldiff sample data for testing the ApacheDS, OpenLDAP, Domino and sunDS LDAP servers. These are located at:

```
${jetspeed-source-home}/components/security/src/test/JETSPEED-INF/directory/config/
```

We'll discuss the security-spi-ldap.xml file in detail below.

LDAP Connection properties

One of the first Jetspeed needs to know is how it to connect to the directory store.

This is done by providing the following properties:

`initialContextFactory`

The initial context factory

```
<constructor-arg index="0">
  <value>com.sun.jndi.ldap.LdapCtxFactory</value>
</constructor-arg>
```

`ldapServerName`

The name of the LDAP server

```

```

```
<constructor-arg index="1">
  <value>localhost</value>
</constructor-arg>
```

`ldapServerPort`

The port of the LDAP server

```
<constructor-arg index="2">
  <value>389</value>
</constructor-arg>
```

`rootContext`

The root context of the LDAP server

```
<constructor-arg index="3">
  <value>o=sevenSeas</value>
</constructor-arg>
```

`rootDn`

The username

```
<constructor-arg index="4">
  <value>uid=admin,ou=system</value>
</constructor-arg>
```

`rootPassword`

The password

```
<constructor-arg index="5">
  <value>secret</value>
</constructor-arg>
```

Validate the connection using an LDAP browser:

LDAP Object Filters

A directory service can store any type of object anywhere. As Jetspeed needs to work with roles, groups and users that are defined within the directory, it needs some help in finding them.

The following 3 properties define how Jetspeed will lookup Roles, Groups and Users from the directory store.

- RoleFilter
- GroupFilter
- UserFilter

Property values must be valid objectClassses that are defined in the LDAP schema.

Most LDAP vendors usually expose their schema through an LDIF file that defines every attribute and objectclass available in the directory store.

A configuration based on Lotus Domino might look like this

```
RoleFilter=(&(objectclass=groupOfUniqueNames)(!(objectClass=dominoGroup)))
GroupFilter=(objectclass=dominoGroup)
UserFilter=(objectclass=dominoPerson)
```

Domino uses the **dominoGroup** objectClass to define a group, **dominoPerson** to define a user, and **groupOfUniqueNames** to define a role. Since group also has the groupOfUniqueNames as an object class, we need to define a filter for the roles, so that it will only pick up roles. If we had defined the RoleFilter as being (objectclass=groupOfUniqueNames), then the filter would have also picked up the groups.

`RoleFilter`

This property tells Jetspeed that roles can be recognized by looking for an **objectClass** attribute with value **groupOfUniqueNames**.

```
<constructor-arg index="6">
  <value>=(objectclass=groupOfUniqueNames)</value>
</constructor-arg>
```

`GroupFilter`

This property tells Jetspeed that groups can be recognized by looking for an **objectClass** attribute with value **groupOfNames**.

```
<constructor-arg index="7">
  <value>=(objectclass=groupOfUniqueNames)</value>
</constructor-arg>
```

`UserFilter`

This property tells Jetspeed that users can be recognized by looking for an **objectClass** attribute with value **organizationalPerson**.

```
<constructor-arg index="8">
  <value>=(objectclass= organizationalPerson)</value>
</constructor-arg>
```

Alongside these filters, we can also define a filter base for each of those objects (roles, groups and users).

Group/Role membership

In LDAP there are basically 2 ways to define group & role membership (the fact that a user belongs to a group or a role):

- The user object has an attribute that specifies the groups he is a member of. This is usually done through a memberOf attribute. Microsoft Active Directory and Sun Directory Server use the memberOf and nsrole attribute on the user object.
- The group/role object contains the group membership information via a multi-valued attribute. No attributes are put on the user to specify membership. Each group/role object has a member list that contains the users belonging to the group

Jetspeed supports both models.

The primary tasks concerning membership of an LDAP are

- Determining if a user is part of a particular group/role
- Obtain a list of users belonging to a particular group/role

The 2 models we just covered have an impact on how these tasks are performed

- Attributes on user object
  - Determining if a user is part of a particular group/role:
    - lookup the membership attribute (ex: memberOf) on the user object for a particular group/role
  - Obtain a list of users belonging to a particular group/role:
    - iterate over the all users, and check their memberOf attribute values for the group
- Attributes on group/role object
  - To determine if a user is part of a particular group:
    - search the member list on the group for the user
  - To determine the users belonging to a particular group:
    - iterate over the member list on the group

We'll now discuss in detail how group/role membership can be configured.

Role membership

As already discussed, Jetspeed supports 2 models when it comes to Role membership:

1. Putting the attributes on the user
2. Putting the attributes on the role

Jetspeed requires that 1 of 2 properties is set with a value to determine the model:

- RoleMembershipAttributes
- UserRoleMembershipAttributes

`RoleMembershipAttributes`

In order to store role membership on the role, we'll set the **RoleMembershipAttributes** attribute by specifying the attribute on the role object that contains the membership information. We don't provide a value for the **UserRoleMembershipAttributes** property.

```
<constructor-arg index="9">
  <value>member</value>
</constructor-arg>
```

This will make sure that the member attribute is set on the role object, as can be seen in the following screenshot. In the next example, the RoleMembershipAttribute will be blank, so the attributes will be on the user level.

In the screenshot below, we have a Role object defined by **cn=Role3,ou=Roles,ou=OrgUnit1,o=sevenSeas**

The role contains a member attribute, listing all users belonging to that role.

A role with 2 members

The value of the member attribute is the fully qualified DN of the user (including the root context). As you can see, the user doesn't contain any attributes with regards to role membership.

A user

When this attribute is set, Jetspeed will determine the roles for a particular user by performing the following query:

```
(&(member=cn=user1,ou=people,ou=orgunit1,o=sevenSeas)(objectclass=groupOfNames))
```

This search filter will return any number of Roles in the directory. The next step for Jetspeed is to identifiy these roles internally. In order to uniquely identify a role, it will use the RoleIdAttribute.

In the example above, cn=Role1 would have been amongst the searchresult. Jetspeed will use the RoleIdAttribute to pickup the role name.

`UserRoleMembershipAttributes`

In order to store role membership on the user, we'll set the **UserRoleMembershipAttributes** attribute by specifying the attribute on the user object that contains the membership information. We don't provide a value for the **RoleMembershipAttributes** property.

```
<constructor-arg index="10">
  <value>memberOf</value>
</constructor-arg>
```

This will make sure that for each role the user belongs to, the memberOf attribute is set on the user object, as can be seen in the following screenshot:

User belonging to 4 different roles

The value of the **memberOf** attribute is the fully qualified DN of the role (including the root context). It is a multi valued attribute, so a user can have zero or more **memberOf** attribute values.

As you can see, the user belongs to a role defined by **cn=role1,ou=Roles,OrgUnit1,o=sevenSeas**.

In order to resolve role membership, Jetspeed will search the directory for roles by using the following filter:

```
# define the filters needed to search for roles/groups/users
RoleFilter=(objectclass=groupOfUniqueNames)
```

As you can see in the screenshot, cn=role1,o=sevenSeas corresponds to an object representing a role.

Notice the empty uniqueMember attribute. Most LDAP schemas force you to have a **uniqueMember** attribute on a **groupOfUniqueNames** object. Since Jetspeed needs to be able to create roles (that are empty upon creation), an empty **uniqueMember** attribute needs to be set. This is configurable by Jetspeed through the **RequiredAttributeClasses** property.

A role without any members

Group membership

As already discussed, Jetspeed supports 2 models when it comes to Group membership:

1. Putting the attributes on the user
2. Putting the attributes on the group

Jetspeed requires that 1 of 2 properties is set with a value to determine the model:

- GroupMembershipAttributes
- UserGroupMembershipAttributes

`GroupMembershipAttributes`

In order to store group membership on the group, we'll set the **GroupMembershipAttributes** attribute by specifying the attribute on the group object that contains the membership information. We don't provide a value for the **UserGroupMembershipAttributes** property.

```
<constructor-arg index="11">
  <value>uniqueMember</value>
</constructor-arg>
```

This will make sure that the **uniqueMember** attribute is set on the group object, as can be seen in the following screenshot. In the previous example, the **GroupMembershipAttributes** was blank, so instead the **UserGroupMembershipAttributes** was used on the user level:

The value of the uniquemember attribute is the fully qualified DN of the user (including the root context). As you can see, the user doesn't contain any attributes with regards to group membership.

`UserGroupMembershipAttributes`

In order to store group membership on the user, we'll set the **UserGroupMembershipAttributes** attribute by specifying the attribute on the user object that contains the membership information. We don't provide a value for the **GroupMembershipAttributes** property.

```
<constructor-arg index="12">
  <value>memberOf</value>
</constructor-arg>
```

This will make sure that the **memberOf** attribute is set on the user object, as can be seen in the following screenshot.

Only one of those parameters can be filled in. If the **GroupMemberShipAttributes** is set, Jetspeed assumes that the attribute to determine group membership is on the group object.

User belonging to 2 different roles

The value of the memberOf attribute is the fully qualified DN of the role (including the root context). It is a multi valued attribute, so a user can have zero or more memberOf attribute values. In the screenshot above, we can see that user1 belongs to 2 roles.

As you can see, the role is defined in **cn=role1,o=sevenSeas**. (notice the empty uniqueMember attribute).

Role definition

Group membership (roles)

Besides storing users in a group, Jetspeed also supports storing roles into groups.

Again, just like with the basic group membership for users, Jetspeed supports 2 models when it comes to Group membership for roles:

1. Putting the attributes on the role

2. Putting the attributes on the group

Jetspeed requires that 1 of 2 properties is set with a value to determine the model:

- GroupMembershipForRoleAttributes
- RoleGroupMembershipForRoleAttributes

`GroupMembershipForRoleAttributes`

In order to store group membership on the group, we'll set the GroupMembershipAttributes attribute by specifying the attribute on the group object that contains the membership information. We don't provide a value for the UserGroupMembershipAttributes property.

```
<constructor-arg index="13">
  <value>uniqueMember</value>
</constructor-arg>
```

This will make sure that the uniqueMember attribute is set on the group object, as can be seen in the following screenshot. In the previous example, the GroupMembershipAttributes was blank, so instead the UserGroupMembershipAttributes was used on the user level.

The value of the uniquemember attribute is the fully qualified DN of the user (including the root context). As you can see, the user doesn't contain any attributes with regards to group membership.

`RoleGroupMembershipForRoleAttributes`

In order to store group membership on the user, we'll set the **UserGroupMembershipAttributes** attribute by specifying the attribute on the user object that contains the membership information. We don't provide a value for the **GroupMembershipAttributes** property.

```
<constructor-arg index="14">
  <value>memberOf</value>
</constructor-arg>
```

This will make sure that the **memberOf** attribute is set on the user object, as can be seen in the following screenshot.

The value of the uniquemember attribute is the fully qualified DN of the user (including the root context). As you can see, the user doesn't contain any attributes with regards to group membership.

Only one of those parameters can be filled in. If the **GroupMemberShipAttributes** is set, Jetspeed assumes that the attribute to determine group membership is on the group object.

User belonging to 2 different roles

The value of the memberOf attribute is the fully qualified DN of the role (including the root context). It is a multi valued attribute, so a user can have zero or more memberOf attribute values. In the screenshot

above, we can see that user1 belongs to 2 roles.

As you can see, the role is defined in **cn=role1,o=sevenSeas**. (notice the empty uniqueMember attribute).

Role definition

DefaultSearchBase

Jetspeed allows you to define a default search base that will be used to search the directory

```
<constructor-arg index="15">
  <value></value>
</constructor-arg>
```

LDAP Object Filter base

Jetspeed allows you to define the search base that will be applied to queries for roles, groups and users.

Roles, groups and user are typically stored in well-defined containers within the LDAP structure.

- Roles can be stored in ou=Roles,ou=OrgUnit1
- Groups can be stored in ou=Groups,ou=OrgUnit1
- Users can be stored in ou=People,ou=OrgUnit1

This allows you to have the following structure in your LDAP schema. Notice how there are many organizational units within the o=sevenSeas schema. Jetspeed will limit its search scope on the LDAP to the property values defined above. This means that only roles, groups and people within OrgUnit1 will be used by Jetspeed.

So, together with the object filers (RoleFilter, GroupFilter, UserFilter), Jetspeed will be able to locate the roles, groups and users within the directory.

Using these properties, Jetspeed will also create roles, groups and users using the provided ObjectClasses.

`RoleFilterBase`

Using the property value below, Jetspeed will search for roles in the ou=Roles,ou=OrgUnit subtree.

```
<constructor-arg index="16">
  <value>ou=Roles,ou=OrgUnit1</value>
</constructor-arg>
```

`GroupFilterBase`

Using the property value above, Jetspeed will search for groups in the ou=Groups,ou=OrgUnit subtree.

```
<constructor-arg index="17">
  <value>ou=Groups,ou=OrgUnit1</value>
</constructor-arg>
```

UserFilterBase

Using the property value above, Jetspeed will search for users in the ou=People,ou=OrgUnit subtree.

```
<constructor-arg index="18">
  <value>ou=People,ou=OrgUnit1</value>
</constructor-arg>
```

LDAP Object classes

Jetspeed allows you to define the ObjectClasses that are needed to create roles, groups and users through the following properties

- RoleObjectClasses
- GroupObjectClasses
- UserObjectClasses

Through the administrative interface, Jetspeed allows an administrator to create roles, groups and users. Each directory server has its own way of defining a role, group or user. Some of the LDAP vendors use proprietary ObjectClasses to define these objects (for example Domino LDAP server uses an dominoGroup objectClass to define a group).

Using these properties, Jetspeed will create roles, groups and users using the provided ObjectClasses.

RoleObjectClasses

```
<constructor-arg index="19">
  <value>top,groupOfNames</value>
</constructor-arg>
```

Using the settings above, roles will be created like this

Notice how all of the objectClasses defined by the RoleObjectClasses attribute have been created in the LDAP

GroupObjectClasses

```
```

```
<constructor-arg index="20">
  <value>top,groupOfUniqueNames</value>
</constructor-arg>
```

Using the settings above, groups will be created like this

Notice how all of the objectClasses defined by the GroupObjectClasses attribute have been created in the LDAP

`UserObjectClasses`

```
<constructor-arg index="21">
  <value>top,groupOfUniqueNames</value>
</constructor-arg>
```

Using the settings above users will be created like this

Notice how all of the objectClasses defined by the UserObjectClasses attribute have been created in the LDAP

Naming Attributes

- RoleIdAttribute
- GroupIdAttribute
- UserIdAttribute

The attributes above allow you to define the naming attribute for roles / groups and users. When an object is created in the directory, a naming attribute needs to be specified. The naming attribute is the attribute that uniquely defines the object within its subdirectory.

In the screenshot below, you can see that the admin user in OrgUnit1/People is defined by **cn=admin**.

cn is the naming attribute for the user object, as no 2 admin users can exist in the OrgUnit1/People subdirectory

By changing the property, you can control the way Jetspeed creates user objects.

`RoleIdAttribute`

```
<constructor-arg index="22">
  <value>cn</value>
</constructor-arg>
```

GroupIdAttribute

```
<constructor-arg index="23">
  <value>cn</value>
</constructor-arg>
```

UserIdAttribute

```
<constructor-arg index="24">
  <value>uid</value>
</constructor-arg>
```

In the screenshot below, users have the **uid** attribute as their naming attribute

UserId Attribute

When Jetspeed attempts to find a user, it does so based on the userId provided by the user in the login screen. This userId needs to be defined on the object through a specific attribute. Most LDAP servers have a uid attribute that defines the username of the user in the LDAP.

When Jetspeed builds a userPrincipal internally, it will use the attribute corresponding to the value of the userUidAttribute.

userUidAttribute

```
<constructor-arg index="25">
  <value>cn</value>
</constructor-arg>
```

This property is used in conjunction with the UidAttribute

```
UserIdAttribute=cn
UidAttribute=uid
```

membershipSearchScope

Jetspeed allows you to customize the search scope when it comes to membership

```
<constructor-arg index="26">
  <value>cn</value>
</constructor-arg>
```

RequiredAttributeClasses

Some ObjectClasses force you to add specific attributes on the object before storing it in the directory. Jetspeed allows you to specify these attributes for roles and groups through the following properties

- roleObjectRequiredAttributeClasses
- roleObjectRequiredAttributeClasses

For example, most LDAP schemas force you to have a **uniqueMember** attribute on a **groupOfUniqueNames** object.

Since Jetspeed needs to be able to create empty roles through the administrative console, an empty **uniqueMember** attribute needs to be set upon role creation.

This is handled internally by Jetspeed and can be customized by setting the **groupObjectRequiredAttributeClasses** property.

`roleObjectRequiredAttributeClasses`

The following property specifies that if a role is created, an empty **member** attribute will be created on the role object in order to comply with the LDAP schema.

```
<constructor-arg index="30">
  <value>member</value>
</constructor-arg>
```

`groupObjectRequiredAttributeClasses`

The following property specifies that if a group is created, an empty **uniqueMember** attribute will be created on the group object in order to comply with the LDAP schema.

```
<constructor-arg index="31">
  <value>uniqueMember</value>
</constructor-arg>
```

LDAP Object attributes

Jetspeed has an administrative console that allows an administrator to create groups, roles and users in the directory. The Jetspeed LDAP configuration has 3 properties that can manipulate the creation of

those objects

- userAttributes
- roleAttributes
- groupAttributes

Each property accepts a comma separated list of attributes. Placeholders can be used in the attribute value.

`userAttributes`

For example, the following **userAttributes** value will make sure that when Jetspeed creates a user in the directory, the **sn, cn and uid** attribute will be created containing the username of the user.

```
<constructor-arg index="32">
  <value>sn={u},cn={u}</value>
</constructor-arg>
```

`roleAttributes`

For example, the following **roleAttributes** value will make sure that when Jetspeed creates a user in the directory, the **cn** attribute will be created containing the username of the user.

```
<constructor-arg index="33">
  <value>cn={u}</value>
</constructor-arg>
```

`groupAttributes`

For example, the following **groupAttributes** value will make sure that when Jetspeed creates a user in the directory, the **cn** attribute will be created containing the username of the user.

```
<constructor-arg index="34">
  <value>cn={u}</value>
</constructor-arg>
```

LDAP Password attributes

During runtime, Jetspeed needs to read the password that is associated with a user. Jetspeed needs to know the attribute on the user object that contains the password. The **userPasswordAttribute** property defines the LDAP attribute that contains the password of the user

```
```

```
<constructor-arg index="35">
  <value>cn={u}</value>
</constructor-arg>
```

Known Attributes

When Jetspeed performs LDAP queries, we need to specify the set of attributes that we want to return. This is done by specifying a comma separated value of LDAP attributes in the **knowAttributes** property

```
<constructor-arg index="36">
<value>cn,sn,o,uid,ou,objectClass,userPassword,member,uniqueMember,memberOf</value>
</constructor-arg>
```

2.1 Tasks

Tasks

Currently this is just a listing of tasks.

- Remove unused classes.