Providing Custom Support for Authentication and User Data Stores

You can bypass the database Clearspace uses to perform authentication and to store user data, using your own data store instead. You do this by building a service provider that implements interfaces in the Clearspace API. Clearspace calls methods of your implementation to authenticate users and retrieve information about them.

You can provide classes to integrate Clearspace with your custom user data store, LDAP directory and custom authentication system. Through your classes, Clearspace queries your external system about users and authenticates registered users before granting them access to Clearspace.

The API is divided into two groups of interfaces: those for implementing custom authentication and those for implementing interaction with a user data store. It isn't necessary for you to implement both of these together. You can implement only one if that's all you need. For example, if you implement support for a provider that interacts with an external user data store, Clearspace will use its default authentication mechanism.

This topic includes the following sections:

[Implementing an Authentication Mechanism](#)

[Implementing a User Data Provider](#)

Implementing an Authentication Mechanism

To provide custom authentication, you implement two Clearspace interfaces: `com.jivesoftware.base.AuthFactory` and `com.jivesoftware.base.AuthToken`. You put your classes into a JAR file on the Clearspace classpath, then tell Clearspace to use your custom class by adding a Jive property that specifies your class name.

Implement the following `AuthFactory` interface methods:

- `AuthToken createAuthToken(String username, String password)` — Creates an `AuthToken` instance using the specified username and password. If your authentication mechanism doesn't support authentication via a username and password (for example, when single sign-on is used), your method should always return `null`.
- `AuthToken createAuthToken(HttpServletRequest request, HttpServletResponse response)` — This is optional. It creates an `AuthToken` instance using `Servlet` request and response objects. The default implementation of this method checks for the Jive autologin encrypted cookie and uses that information to log in the user if it is found. If you'd like to preserve that functionality, don't extend this method when you create your custom `AuthFactory` class. If you'd like to disable this functionality, extend this method and always return `null`. This method can also be used to integrate with external single sign-on systems. Typically, single sign-on information is stored as either a session object or cookie value. So, you would take that information and create the corresponding AuthToken.

- AuthToken createAnonymousAuthToken() — creates an anonymous auth token. An anonymous auth token is used for the guest user — a user who is not logged into the system.

The default skin will attempt to authenticate by:

1. Checking for an existing AuthToken object in the user's session. If one is not found, go to the next step.
2. Authenticating via the createAuthToken(request, response) method. This check allows single sign-on to work. If this check fails, go to the next step.
3. At this point, it's assumed the user is a guest so create an anonymous auth token.

Using the login form in the default skin will perform authentication using the createAuthToken(username, password) method and then store the resulting AuthToken in the user's session.

Once you've created and compiled your custom authentication classes, you should put them in a JAR file and then add the JAR to the classpath of the Clearspace web application. For example, you might add it to the WEB-INF/lib directory. Next, you must tell Clearspace to use your custom AuthFactory class by adding a Jive property. To do this, start the admin console and navigate to System > Management > System Properties. The property name is "AuthFactory.className" and the value is the fully-qualified name of your custom AuthFactory class. For example, set "AuthFactory.className to "com.foo.CustomAuthFactory" (without the quotes, of course).

Implementing a User Data Provider

Any user data store that you can access programmatically can be integrated with Clearspace. This includes relational databases via the JDBC API, directory services such as LDAP, or custom stores that are accessible via a Java API.

To provide a service for interacting with your user data store, you implement interfaces that handle the service lifecycle, including service initialization, configuration and destruction. You also implement methods through which Clearspace can retrieve objects representing users and retrieve data about each user. The provider you implement is a service provider in the standard sense.

Note: A sample implementation of User Provider API is available as part of the Clearspace source distribution. You receive this distribution with your purchase of Clearspace.

Handling the Service Lifecycle

Implementing the ServiceProvider Interface

This is a base interface that all service providers have to implement. Through your implementation, Clearspace configures and initializes your provider. This interface has the following methods:

- Map<String, String> getConfigurationMap() — Clearspace calls this method during the setup process to obtain the all of the configuration values that need to be set up for your service provider. Your implementation must return a Map<String, String> with configuration keys and their default

values. These keys and default values will then be displayed to an administrator in the setup tool. During setup, they can change these values and Clearspace will save them for future use.
- `void initialize(Map<String, String> configuration)` — Clearspace will call this method with all of the configuration parameters/values that are associated with your service provider. During the setup process, the service provider is queried (through the `getConfigurationMap()` method) for configuration parameters and their default values. These values can then be changed during the setup process and are stored in Clearspace for future use. Once the setup is complete, Clearspace will use the stored configuration values and pass them to the service provider by calling this initialize() method.
- `void destroy()` — Clearspace will call this method to shut down your service provider. Your implementation should release any resources or connections that the service provider might have allocated.

Providing Data About Users

The meat of the provider work is done through your implementation of the `User` and `UserProvider` interfaces, as described below.

Implementing the `UserProvider` Interface

Clearspace uses this interface to query for users. Your implementation must provide support for the following methods:

- `User getUser(String)` — Called to get a User instance identified by the username. Implementations will usually query the user store for the user with the username as a key. If no user exists for the given username, your class must throw a `NotFoundException`. This method must never return `null`; this can cause unexpected errors in Clearspace.

- `User getUser(UserTemplate)` — Called to query for the user using an email address and/or user id. In certain situations, a username is not available but email or user id is. In such situations, Clearspace will construct a partial user object(UserTemplate) with email and/or user id set and query the user provider with it. It is entirely up to the implementations to use or disregard either email or user id. For example, an implementation of this method could look like this:

```
User user = null;

if (userTemplate.getEmail() != null) {
    user = findUserByEmail(userTemplate.getEmail());
} else if (userTemplate.getUserID() != null) {
    user = findUserByUserId(userTemplate.getID());
}else if (userTemplate.getUsername() != null) {
    user = findUserByUsername(userTemplate.getUsername());
}
if(user == null) {
    throw new NotFoundException();
}
return user;
```

- `boolean exists(String)/exists(UserTemplate)` — Called to determine whether a user exists in the data store. This method will usually be called before calling one of the `getUser()` methods.
- `Iterable<User> getUsers()` — Called to get a list of users that are in the user store. For stores that contain large numbers of users, your implementation should consider certain performance optimizations like lazy loading of result sets. In some cases, implementations might find it necessary to cache results to provide the best performance.
- `Iterable<String> getUsernames()` — Called to get a list of user names in the user store. For

stores that contain large numbers of users, implementation should consider certain performance optimizations like lazy loading of result sets. In some cases, implementations might find it necessary to cache results to provide optimum performance.

- `boolean supportsUpdate()` — Implementations that support update to the user stores can enable create/delete methods by returning true from this method. Clearspace will then allow creation and deletion of users from the admin console inside Clearspace. When the update functionality is available, administrators can create and delete users using the Clearspace administration console. The application checks to see if the installed UserProvider supports updates to the user store (by calling this method) and then enables access to create and delete user pages of the administration console. The following methods become available for use in Clearspace when this method returns true:
    - `User create(UserTemplate)` — Called to create a user in the user store. If a user already exists that matches the template, the method should throw an `AlreadyExistsException`.
    - `delete(User)` — Called to delete a user from the store.
- `boolean supportsPagination()` — Implementations that support pagination/browsing functionality can enable pagination support by returning true from this method. This will allow Clearspace to intelligently load users as required. It will do so by first checking the total number of users (with `getCount()` methods) and then requesting users in small batches (`getUsers(startIndex, size)`). Clearspace will keep track of its location (startIndex) in the result set and ask the provider for the next "size" block of users. The following methods become available for user when `supportsPagination()` returns true:
    - `int getCount()` — Called to get a count of total number of users in the store.
    - `Iterable<User> getUsers(startIndex, size)` — Called to load a block of users from the user store, beginning at `startIndex`. The number of users returned should be less than or equal to "size".

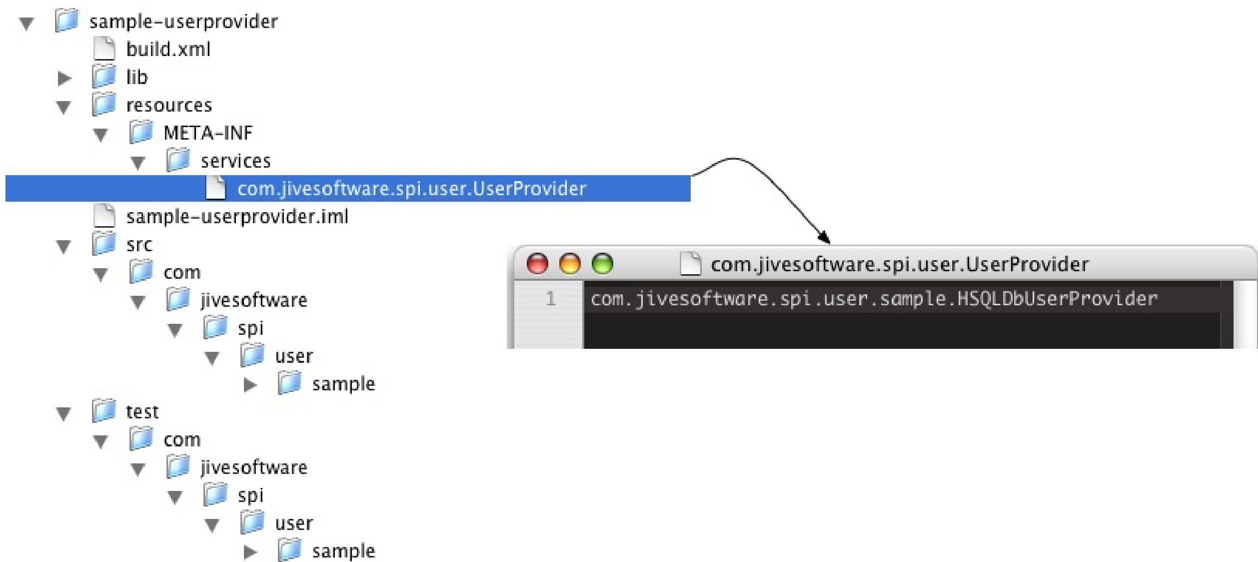
Implementing the `User` Interface

This interface provide access to information about a user's identity (username, email and userid properties). Also, this interface will be used by Clearspace when authenticating users. Your implementation must provide support for the following methods:

- `String getUsername()` — Returns the name of the user. User names are assumed to unique and case insensitive.
- `String getID()` — Returns the id of the user. In most providers this will be a numeric value, often a database-generated sequence number.
- `String getEmail()` — Email address associated with a user.
- `boolean authenticate(char[] password`) - This method will be called when authenticating users with a password.

Building a User Provider

The process of building a standard Service provider is almost exactly the same as building any other JAR archive you might be familiar with. The only difference is the requirement of the service provider configuration file. This file tells the application about the class that implements a particular service. The configuration file must be located in the META-INF/services directory and must be called "com.jivesoftware.spi.user.UserProvider". Note that the configuration file has no extension. Inside the file should be a list of fully qualified class names of all the classes that implement the `UserProvider` interface. In most cases, there is only one class that is implementing the `UserProvider` interface, so the configuration file will have only one entry. For example, a user provider implementation that used an HSQL database to store users could be set up as follows:



During a the build process, the files in the META-INF directory are copied over and made part of the user provider JAR (see the included build.xml in the sample-userprovider for an example). Again, using the above example, the JAR that was generated from this process would look like: