

# Upgrading Extensions to Version 2

Here's a run down of the changes you might need to make to get your Clearspace customizations working on version 2. This topic looks at upgrading actions, macros, widgets, themes, and web services (client and server components), and other customizations.

[Summary of Version 2 Changes](#) (page 1)

[General Plugin Changes](#) (page 2)

[Upgrading FTL Files](#) (page 2)

[Upgrading Themes](#) (page 5)

[Upgrading Widgets](#) (page 6)

[Upgrading Macros](#) (page 7)

[Upgrading Actions](#) (page 9)

[Upgrading Web Services](#) (page 9)

[Upgrading Custom Authentication and User Data Providers](#) (page 12)

[Upgrading API Uses](#) (page 14)

## Summary of Version 2 Changes

Clearspace version 2 includes a few big changes to the conventions and frameworks on which Clearspace is built. Many of these changes were made to improve Clearspace extensibility by letting the application and extensions be more loosely coupled — and so be more durable during upgrade. Other changes that impact customizations were simply feature improvements in which some change to code is required.

Here's a high-level list of the changes that effect extension and customization code:

- Clearspace was **migrated to the Spring framework**. This has both broad and deep effects on code written to run on Clearspace; such code must use the same conventions in order to integrate well. The changes include API refactoring to support dependency injection and integration of Spring modules (for security and transaction management, for example).
- Clearspace was **migrated from WebWork2 to Struts2**. This has broad impact on plugins, but the changes are relatively small — primarily effecting syntax in FreeMarker templates and configuration files.
- **Widgets were enhanced** to support use in multiple contexts, rather than just space overview pages. Changes to widget development support this new feature.
- The **macro-related API was narrowed** to the area documented in version 1, essentially excluding an undocumented area that some people used. Also, macros must now return HTML that qualifies as well-formed XML.
- **Web services support** is now provided through CXF, an evolution of XFire (the version 1 technology). A larger change results from the migration to Spring, which makes integrating new web services a bit more complex.

- **Custom user data providers** are now based on a streamlined API. These must be rewritten, but the new model is much simpler.
- **Custom authentication providers** are now based on Spring through Acegi. The change is pretty significant, including new API and configuration conventions.
- **Myriad API changes** resulted from the migration to Spring and from an effort to streamline and modularize the Clearspace API. These include support for the projects feature, refactoring manager interfaces, content handling, and conventions such as dependency injection.

## General Plugin Changes

For those extensions and customizations deployed as plugins (widgets, actions, macros, and web services), there are a few changes in version 2 with broad effect.

- Migration from WebWork to Struts means replacing your `xwork-plugin.xml` with a `struts.xml`
- Migration to Spring means adding a `spring.xml` for certain kinds of Spring support.
- One benefit of the new model is that modifying your `plugin.xml`, `struts.xml`, or `spring.xml` will provoke Clearspace to reload your plugin. This can be handy for debugging.
- Support for a `<maxserverversion>` element was added. Similar to the `<minserverversion>` element, use the new one to specify that highest Clearspace version on which your plugin is supported. This can be useful when you want to ensure that upgrades to Clearspace prompt the plugin's users to upgrade the plugin also.
- Plugin life cycle methods changed. If you used the version 1 `com.jivesoftware.base.Plugin` interface, you'll notice that its two methods have changed in order to support Spring. Semantics for the methods is unchanged.
  - `Plugin.initialize(PluginManager, PluginMetaData)` is now `Plugin.init`, a method without parameters. Use Spring dependency injection to receive a `PluginManager` instance; you can retrieve a `PluginMetaData` instance from the manager.
  - `Plugin.destroyPlugin` is now `Plugin.destroy`.

## Upgrading FTL Files

Each new version of Clearspace includes changes to FreeMarker template (FTL) files and version 2 is no exception. Typically these changes are needed to support new or enhanced features. The FTL changelog included with a Clearspace release provides a list of the FTL files that were changed from the previous version. However, changes in version 2 have pretty much touched every template. In most cases these changes require a simple search-and-replace to fix. But some changes are more substantial.

The [information on upgrading themes](#) (page 5) suggests an incremental process for making your changes that could save you some aggravation.

The following list suggests the tips and best practices for upgrading your templates.

- When you've overridden existing Clearspace templates (as in a theme), start with new version 2 FTL files that correspond to the version 1 files in your theme. Then, working from your customized version of the version 1 template, transfer your changes to to the version 2 template. Updating the new templates might help you avoid accidentally transferring code from version 1 that no longer works on version 2. This is

especially important given that FreeMarker errors are difficult to debug; errors don't show up until run time. Weaving your version 1 changes into the version 2 templates will make the process more systematic.

As you copy your changes into the version 2 template, keep in mind the changed items described below.

- The version 1 pages whose templates were `community.ftl` and `main.ftl` — community pages and the home page — can be easily customized with widgets in version 2. Before you set out to upgrade these pages, take a look at how much of your customization work could be accomplished by using widgets on the version 2 templates. Using widgets might reduce the amount of customization you need to do and greatly reduce any work in future upgrades.
- If you've overridden `community.ftl` and want to upgrade it, note that the template has been split into multiple FTL files. This was done to divide logically what was a very large template.
- Replace WebWork directives with their Struts counterparts. Clearspace version 2 replaces WebWork conventions with Struts. This is pretty much just search-and-replace work to replace `@ww.` (for WebWork) with `@s` (for Struts). The following example shows how to update the url directive. Notice that the updated code also omits the `includeParams='none'` attribute; in Struts 2, which Clearspace version 2 uses, `none` is the default value for `includeParams`.

Version 1 (supporting WebWork):

```
<style type="text/css" media="screen">
  @import "<@ww.url value='/styles/jive-blog.css' includeParams='none' />";
</style>
```

Version 2 (supporting Struts):

```
<style type="text/css" media="screen">
  @import "<@s.url value='/styles/jive-blog.css' />";
</style>
```

- Update calls to [APIs that have been consolidated and simplified](#) (page 14) . Pay attention to the places in a template where code calls methods directly (although, as a best practice, you should avoid calling methods in FTL code and use actions instead). Here's an example in which you'd replace a call to a `JiveGlobals` methods with a call to `JiveResourceResolver`:

Version 1 (using `JiveGlobals`):

```
<@ww.text name="doc.viewer.more_recent_ver.text">
  <@ww.param><a href="<@ww.url value="{JiveGlobals.getJiveObjectURL(document)}" includeParams="none" />"></@ww.param>
  <@ww.param></a></@ww.param>
</@ww.text>
```

Version 2 (using `JiveResourceResolver`):

```
<@s.text name="doc.viewer.more_recent_ver.text">
  <@s.param><a href="<@s.url value="{JiveResourceResolver.getJiveObjectURL(document)}" />"></@s.param>
  <@s.param></a></@s.param>
</@s.text>
```

Here's another example where the change was from another class and method, but again to `JiveResourceResolver`:

### Version 1 (using CommunityUtils):

```
<a href="<@ww.url value='${CommunityUtils.getCommunityURL(community)}'
includeParams='none' />?view=documents"
class="jive-link-more"><@ww.text name="doc.main.brdrmb.documents.link" /></a>
```

### Version 2 (using JiveResourceResolver):

```
<a href="<@s.url value='${JiveResourceResolver.getJiveObjectURL(container)}' />?view=documents"
class="jive-link-more"><@s.text name="doc.main.brdrmb.documents.link" /></a>
```

- Update community references to container references. One twist on the API changes means that both projects and communities (also known as "spaces") are represented conceptually as containers. In the version 2 API, the `Community` and `Project` interfaces both inherit from `JiveContainer`. To disambiguate between the projects and communities, you'll need to pass a container *type* with your action calls. The actual disambiguation is handled by Clearspace, however, when it intercepts the call before passing it to the action. The net effect is that the action itself receives only the container ID parameter, not the container type. Here's an FTL example:

### Version 1 (specifying a community by passing its community ID)

```
<@ww.action name="community-breadcrumb" executeResult="true" ignoreContextParams="true">
  <@ww.param name="communityID" value="${community.ID?c}" />
</@ww.action>
```

### Version 2 (specifying a community by passing both its type and its ID)

```
<@s.action name="community-breadcrumb" executeResult="true" ignoreContextParams="true">
  <@s.param name="containerType" value="${container.objectType?c}" />
  <@s.param name="container" value="${container.ID?c}" />
</@s.action>
```

- Update code that handles content. This includes code that gets content for display as wiki markup, HTML, and so on. Among the API changes were [several designed to streamline and add structure to how you handle content](#) (page 18) . For example, in version 1, to get content as wiki markup you would have called methods of the message or document or comment itself. In version 2, you pass the content object to a method inherited from `JiveActionSupport`. These methods include `renderToText`, `renderToHtml`, `renderSubjectToHtml`, and so on. Here's an example using `convertToWikiSyntax`:

### Version 1 (using content object method)

```
<textarea id="comment-body-edit-${comment.ID?c}" name="body" rows="10"
style="width:100%;font-family:verdana,arial,Helvetica,sans-serif;font-size:8pt;"
class="jive-comment-textarea">${comment.unfilteredBody!html}</textarea>
```

### Version 2 (using JiveActionSupport method)

```
<textarea id="comment-body-edit-${comment.ID?c}" name="body" rows="10"
style="width:100%;font-family:verdana,arial,Helvetica,sans-serif;font-size:8pt;"
class="jive-comment-textarea">${action.convertToWikiSyntax(comment)!html}</textarea>
```

## Removed and Renamed FTL Files

Version 1 FTL File	Version 2 Change
community-document-picker.ftl	Renamed to container-document-picker.ftl
community-thread-picker.ftl	Renamed to container-thread-picker.ftl
import-callback-communitynntp.ftl	Removed.
import-directory-error.ftl	Removed.
import-directory-updatetags.ftl	Removed.
import-directory.ftl	Removed.
datepicker.ftl	Renamed to datetimepicker.ftl

## Upgrading Themes

Nearly all of your work upgrading themes will focus on [upgrading FreeMarker template \(FTL\) files](#) (page 2) . The way you build and deploy themes is unchanged from version 1.

**But wait — there's more.** Having said that, the best practice recommendation in version 2 is to use widget-customized pages wherever possible as an alternative to custom FTLs. You can use widgets in more places than in version 1; check out the [section on upgrading widgets](#) (page 6) for more.

Here's why you should use widgets:

- You can do with widgets much of what you'd do with custom FTL markup. In addition, you can write logic in Java behind the your widget UI.
- Building widgets to support new features is a good deal tidier than adding new FTL files. While custom FTL files override default FTL files, widgets are encapsulated by the plugin deployment model — essentially sandboxed.
- When it's time to upgrade Clearspace, the work needed to upgrade FTL files you've customized would likely be a good deal greater than what's needed to upgrade a widget.

## Suggested Upgrade Process

Use the following steps as a systematic way to upgrade your themes.

1. Consider which custom FTLs in your themes can be replaced by customizing the page with widgets.
2. Use `jive.devMode` Jive property for debugging. By default, Clearspace hides FreeMarker errors in themes. With dev mode on, you'll see them.
3. Disable themes while upgrading them.
4. Enable your custom FTL files one at a time. Debugging incrementally will make the process smoother.

## Upgrading Widgets

Widgets can be used much more broadly in version 2 than previously, so most of your widget-specific upgrade changes are related to this broader support. In version 1 a system or space administrator could use widgets only to assemble a space or community overview page. In version 2 both administrators and users can customize page layouts in several areas. Widgets can be used on the Clearspace instance home page (which has been rendered from `main.ftl`), a user's personal home page, a community/space overview page, a project overview page.

Note that unless your widget is extremely simple, you're likely to also need to keep in mind [API changes](#) (page 14) and [FreeMarker changes](#) (page 2) . For example, version 2 requires that widgets acquire references to Clearspace manager beans using Spring. The rest of widget development model — artifacts involved, how you deploy them, and so on — is unchanged from version 1.

When upgrading a widget (or developing a new one on version 2), you use the `@WidgetTypeMarker` annotation to specify which of the display contexts your widget is allowed in. Deciding which contexts to allow is an important part of your widget's design. For example, you might decide that a widget that takes a very individual-oriented set of property values (such as the Tag widget, which displays content associated with a particular tag) isn't useful in high-level contexts such as the instance or space home pages (where the broad set of people viewing might want views on a large variety of tags).

The `@WidgetTypeMarker` annotation supports values from the `WidgetType` enumeration. Here's an example that includes all of those values:

```
@WidgetTypeMarker({WidgetType.HOME PAGE, WidgetType.PERSONALIZEDHOME PAGE,
    WidgetType.COMMUNITY, WidgetType.PROJECT})
@propertyNames("myProperty")
public class MyWidget extends BaseWidget {
    // Implementation code omitted.
}
```

In order to reduce the tight coupling between the widget views (which are typically built using FreeMarker) and the rest of the Clearspace application, the widget context that was previously populated with a reference to the current context via a `JiveContext` instance has been modified so that the widget view no longer has access to that instance. That means that you'll need to provide everything that your widget view needs through the properties that the widget interface requires. Typically you'll create a widget class that extends `BaseWidget` and then you'll override this method:

```
protected Map<String, Object> loadProperties(WidgetContext widgetContext, ContainerSize size)
```

So in the previous version of Clearspace, you might have had something like this in the FTL file that's your widget's view:

```
`${widgetContext.jiveContext.communityManager.rootCommunity.ID?c}
```

Because the `JiveContext` instance has been removed from the `WidgetContext` class, you'll need to provide your view with the properties it needs explicitly in your widget. Here's an example:

```
public class MyWidget extends BaseWidget {

    // Declare a property variable and setter for injection by Spring.
    private CommunityManager communityManager;
```

```

public void setCommunityManager(CommunityManager cm) {
    this.communityManager = cm;
}

protected Map<String, Object> loadProperties(WidgetContext widgetContext,
    ContainerSize size)
{
    Map<String, Object> properties = super.loadProperties(widgetContext, size);

    // Implementation code omitted.
    properties.put("rootCommunityID", communityManager.getRootCommunity().getID());
    return properties;
}
}

```

Finally, because you can create a widget that exists in multiple parts of the application (the homepage, a personalized homepage, a community, a project), you'll sometimes want to change the behavior of your widget based on where the widget is being rendered. You can determine the render context of your widget by checking the type of the `WidgetContext` class that you're given in the `loadProperties` method. Here's some example code that shows how you can determine what context the widget is in:

```

public class MyWidget extends BaseWidget {
    protected Map<String, Object> loadProperties(WidgetContext widgetContext, ContainerSize size) {
        Map<String, Object> properties = super.loadProperties(widgetContext, size);
        if (widgetContext.getWidgetType() == WidgetType.COMMUNITY) {
            CommunityWidgetContext cwc = (CommunityWidgetContext)widgetContext;
            // Do something specific for the community
        }
        else if (widgetContext.getWidgetType() == WidgetType.HOMEPAGE) {
            HomepageWidgetContext hwc = (HomepageWidgetContext)widgetContext;
            // Do something specific for the homepage
        }
        else if (widgetContext.getWidgetType() == WidgetType.PERSONALIZEDHOMEPAGE) {
            PersonalizedHomepageWidgetContext phwc =
                (PersonalizedHomepageWidgetContext)widgetContext;
            // Do something specific for the personalized homepage
        }
        else if (widgetContext.getWidgetType() == WidgetType.PROJECT) {
            ProjectWidgetContext wwc = (ProjectWidgetContext)widgetContext;
            // Do something specific for the project
        }

        properties.put("rootCommunityID", communityManager.getRootCommunity().getID());
        return properties;
    }
}

```

## Upgrading Macros

Depending on what your macro does, it might need substantial upgrade changes or none at all. The main macro-specific difference arises from the fact that Clearspace content is now stored as XML, rather than plain

text. At run time Clearspace renders plain text, rich text, or HTML from stored XML. This means that the HTML you return for display when the macro is previewed or published must qualify as well-formed XML.

Note that unless your macro is extremely simple, you're likely to also need to keep in mind [API changes](#) (page 14) and [FreeMarker changes](#) (page 2) . The rest of macro development model — artifacts involved, how you deploy them, and so on — is unchanged from version 1.

In version 1, you might have written your macro class in one of two ways: implementing the `com.jivesoftware.base.Macro` interface or extending the `com.jivesoftware.base.BaseMacro` class. Implementing the Macro interface was the relatively simple (and documented) way to write a macro class. If you extended the `BaseMacro` class you were probably doing so in order to support alternate content formats for targets other than HTML — including for email, the plain text editor, and so on. In version 2 Clearspace handles conversion to these other formats for you.

The process for upgrading your macro class will differ depending on which approach you took. The following describes the two common version 1 ways to implement a macro class and describes what you can do to upgrade each.

## Your Macro Class Implements the Macro Interface

In version 1 the `Macro` interface has a single method: `String render(String, Map, MacroContext)`. Your implementation received macro content and parameters (if any), along with information about the context in which it was being used, then simply returned the HTML that should be displayed in the published content.

In version 2 this is still supported with one caveat: The HTML returned by your render method must now qualify as well-formed XML. That's because the return value is now inserted into the XML that represents the content it's being used in. Malformed XML returned by a macro won't be used by Clearspace.

To upgrade your macro class, simply ensure that the HTML your render method implementation returns is well-formed XML.

## Your Macro Class Extends the Version 1 BaseMacro Class

In version 1 the `BaseMacro` class extended `BaseFilter`, which provided all of the methods a macro needed to return content to Clearspace, whether the content was being displayed in the Clearspace UI or in a notification email. You implemented each of these methods to handle the different targets. In version 2, Clearspace handles rendering for each of these targets so your code doesn't need to.

In version 2 there's a `BaseMacro` class, but it no longer extends `BaseFilter`, implementing `RenderMacro` instead. In other words, macro classes extending the version 1 `BaseMacro` class won't compile on version 2. The version 2 `BaseMacro` class is for internal use and will almost surely change in future Clearspace releases.

To upgrade your macro class, rewrite it so that it implements the `Macro` interface rather than extending `BaseMacro`. In most cases this will mean moving your `BaseMacro.executeHtmlTarget` method implementation (along with the logic that supports it) to your `Macro.render` method implementation. Both methods return HTML as a `String`. You can discard the code designed to support the other targets (methods `executePlainTextTarget`, `executeWysiwygEditorTarget`, and so on).

## Upgrading Actions

Version 2 uses Struts rather than WebWork. Actually WebWork 2 *became* Struts 2 when the two open source communities merged. Clearspace was simply upgraded to use the new Struts, which is the latest version of WebWork. And Struts. Well, you get the idea.

In addition to FreeMarker syntax changes, you'll find migration changes described in the [Struts documentation from Apache](#). The Apache site also describes the [key differences between WebWork and Struts](#). Also, be sure to see the information on [upgrading FTLs](#) (page 2) , [upgrading API uses](#) (page 14) , and [general plugin changes](#) (page 2) .

Here are a couple of things you'll notice immediately:

- In plugins, your `xwork-plugin.xml` must be changed to `struts.xml`.
- Action support methods `doCancel()` and `doInput()` from WebWork became `cancel()` and `input()` in Struts.

**Important:** Overriding actions, which was occasionally successful in version 1, is likely to get you into trouble in version 2. Best not to do it.

## Upgrading Web Services

Several changes were made that could impact your web services code, whether the code is for a client or new service. Generally, these changes were made to make the API easier to use, to add functionality (such as support for REST, new in version 2), or to remove a little-used feature that couldn't be made to work well.

- In version 2 Clearspace web services are built on the [CXF framework](#). CXF evolved from XFire, the framework on which Clearspace version 1 web services were built. When upgrading, be sure to update your classpath to replace the XFire JAR file with the CXF JAR.
- SOAP-based web services no longer support caching; references to caching have been removed from the web service client API.
- If you're using the Clearspace web service client API (for SOAP web services), you'll need to update your references to some classes. To clarify the difference between the standard and web service client APIs, some class names were prefixed with "WS". The following table lists the renamed classes; these are all in the `com.jivesoftware.community.webservices` package. Of course, this also means that signatures for some of the methods within the API itself have changed where these types were used.

### Renamed Web Service Client Classes

Version 1 Type	Version 2 Type
Attachment	WSAttachment
BinaryBody	WSBinaryBody
Blog	WSBlog
BlogPost	WSBlogPost
BlogPostResultFilter	WSBlogPostResultFilter

BlogResultFilter	WSBlogResultFilter
BlogTagResultFilter	WSBlogTagResultFilter
Cache	<i>Removed. SOAP-based web services no longer support access to caching.</i>
Comment	WSComment
Community	WSCommunity
ContentTag	WSContentTag
Document	WSDocument
FeedbackResultFilter	WSFeedbackResultFilter
ForumMessage	WSForumMessage
ForumThread	WSForumThread
Group	WSGroup
Image	WSImage
JiveObject	WSJiveObject
Locale	WSLocale
Poll	WSPoll
PrivateMessage	WSPrivateMessage
PrivateMessageFolder	WSPrivateMessageFolder
ProfileField	WSProfileField
ProfileFieldOption	WSProfileFieldOption
Property	WSProperty
Query	WSQuery
Rating	WSRating
ResultFilter	WSResultFilter
StatusLevel	WSStatusLevel
StatusLevelScenario	WSStatusLevelScenario
TagCount	WSTagCount
TagResultFilter	WSTagResultFilter

User	WSUser
UserContentCommentResultFilter	WSUserContentCommentResultFilter
UserProfile	WSUserProfile
Watch	WSWatch

- Update some API references where arrays were used so that collections are used instead.
- The .NET client API is no longer available. If you've got a .NET client, you'll need to rewrite it using another web service client library.
- If you've written web services that expose Clearspace functionality, note that the `WSUtil.getJiveContext` method has been removed. Instead, as noted in the [description of API changes](#) (page 15) , you should use Spring properties to obtain manager references.
- Web services you've included in a plugin are no longer declared in the `plugin.xml` file. In version 2, when writing a web service for deployment in a plugin, you declare the web service in a `spring.xml` file included in your plugin. Use the [conventions described in the Apache CXF user's guide](#).

Here's an example

```
<bean id="echoServiceImpl" class="com.jivesoftware.clearspace.plugin.example.webservices.EchoServiceImpl" />

<bean id="saajInInterceptor" class="org.apache.cxf.binding.soap.saaj.SAAJInInterceptor" />
  <bean id="wss4jInInterceptor" class="org.apache.cxf.ws.security.wss4j.WSS4JInInterceptor">
    <constructor-arg>
      <map>
        <entry key="passwordCallbackClass"
          value="com.jivesoftware.community.webservices.server.xfire.PasswordHandler" />
        <entry key="action" value="UsernameToken" />
      </map>
    </constructor-arg>
  </bean>
<bean id="validationCheckInterceptor"
  class="com.jivesoftware.community.webservices.server.xfire.ValidationCheckInterceptor">
  <property name="jiveUserDetailsService" ref="jiveUserDetailsService" />
</bean>

<jaxws:endpoint id="echoService" address="/soap/EchoService">
  <jaxws:implementor>
    <ref bean="echoServiceImpl" />
  </jaxws:implementor>
  <jaxws:inInterceptors>
    <ref bean="saajInInterceptor" />
    <ref bean="wss4jInInterceptor" />
    <ref bean="validationCheckInterceptor" />
  </jaxws:inInterceptors>
</jaxws:endpoint>
```

- In version 2 WSDL operation params have friendly names. You should get new version 2 WSDL.
- In version 1 accessing web services via XML-RPC used two extra parameters for username and password; version 2 uses basic auth instead.

## Upgrading Custom Authentication and User Data Providers

As in version 1, in version 2 you can write your own components to manage user information and provide custom authentication.

### Upgrading a User Data Provider

In version 1 you created a custom user provider by implementing the `com.jivesoftware.spi.user.UserProvider` interface to interact with your user data source and implementing the `com.jivesoftware.base.User` interface to represent a user. In version 2, you implement `UserProvider` and `User`, but the interfaces have changed a bit. The version 2 model is more streamlined, removing the need to implement the lifecycle methods that the SPI framework requires. Here's a summary of the changes:

- Because you no longer implement the `ServiceProvider` interface, you don't implement support for its lifecycle methods. Clearspace handles provider life cycle through Spring.
- The `com.jivesoftware.base.User` interface is read-only (lacks setters) in version 2. Your implementation now provides the interface and logic for updating user information and returning it to Clearspace.
- The `User.authenticate` method has been removed. Use an authentication provider to authenticate a user.
- The `User` interface includes several new methods for retrieving information about the user.
- You now connect your custom provider to Clearspace using Spring conventions rather than by setting a Jive property.

Here's a high level view of the upgrade steps you'll need to consider.

- Implement the `User` interface to support whatever setters it needs. You'll also want to implement the `is*Supported` methods that indicate to Clearspace which user data is supported for setting.
- For `UserProvider` methods in which your code receives and returns a `User` instance, you'll need to rewrite a bit to return your own `User` implementation. In version 1 the `UserProvider` instance received a `UserTemplate` with setters. Because you can no longer simply call setters on the `User` instance you receive (it might not have any), you'll instead copy data from the received instance into an instance of your own implementation and return your own (such as by constructing your instance from the `User` your code receives). The methods you'll need to change include `create(User)` and `getUser(User)`.
- Implement `UserProvider.supportsUpdate` to return true if your data store supports creating or updating users from Clearspace. If you return false from this method, be sure to also throw an `UnsupportedOperationException` from the create and update methods.

### Upgrading an Authentication Provider

Authentication is substantially changed in version 2. For example, you no longer need to pass an `AuthToken` instance with method calls. Every request is guaranteed to have an authentication context. In version 2, Clearspace uses Acegi security, which designed to fit well with the Spring framework. If you're implementing your own authentication provider, your components are based on Acegi; in some cases you might be able to use the classes included with Acegi, such as `Authentication` implementations.

Be sure to read the [Acegi documentation](#), which includes information you'll find useful.

## What Your Implementation Should Include

- Implement `org.acegisecurity.AuthenticationProvider` to provide the service that authenticates users for Clearspace. This interface includes two methods: `authenticate(Authentication)` and `supports(Class)`. Your `supports` method is called by Clearspace to determine whether the Authentication approach is supported by your provider. You should return `true` if the `Authentication` class it receives is one your `authenticate` method supports.

```
public boolean supports(Class authentication) {
    return authentication == UsernamePasswordAuthenticationToken.class;
}
```

The `authenticate` method receives an `Authentication` instance containing information identifying the user. Your implementation knows how to authenticate the user and should return an `Authentication` instance that indicates whether the user is authenticated. Here's a simple `authenticate` method example in which the `checkAuth` method (not shown) communicates with the data source and returns `true` if the user is authentic:

```
public Authentication authenticate(Authentication authentication) throws AuthenticationException {
    UsernamePasswordAuthenticationToken auth = (UsernamePasswordAuthenticationToken) authentication;
    String username = String.valueOf(auth.getPrincipal());
    String password = String.valueOf(auth.getCredentials());
    if(!checkAuth(username, password)){
        throw new BadCredentialsException("Username:" + username + " was not authenticated");
    }
    return new UsernamePasswordAuthenticationToken(username, password, new GrantedAuthority[]{});
}
```

- Implement `org.acegisecurity.Authentication` to contain details of the authentication request (you can also use an existing class that implements `Authentication`). An `Authentication` class represents the user in the context of a particular authentication approach -- such as basic authentication, LDAP, X.509 certificate, and so on. An instance includes details such as principal (something identifying the user, such as a username) and credentials (such as a password). Your `isAuthenticated` method should return `true` if the user is authentic.

The `Authentication` class includes a `getAuthorities` method that is not currently supported by Clearspace. Your implementation should return an empty array:

```
public GrantedAuthority[] getAuthorities() {
    return new GrantedAuthority[0];
}
```

- Connect your authentication provider to Clearspace by adding a Spring configuration XML file to the `jiveHome/etc` directory.

```
<?xml version="1.0" encoding="UTF-8"?>
  <beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd" >
    <bean id="filterChainProxy" class="org.acegisecurity.util.FilterChainProxy">
      <property name="filterInvocationDefinitionSource">
        <value>
          CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
        </value>
      </property>
    </bean>
  </beans>
```

```

PATTERN_TYPE_APACHE_ANT
/upgrade/**=httpSessionContextIntegrationFilter, upgradeAuthenticationFilter, upgradeExceptionTranslationFilter
/post-upgrade/**=httpSessionContextIntegrationFilter, postUpgradeAuthenticationFilter, postUpgradeExceptionTranslationFilter
/admin/**=httpSessionContextIntegrationFilter, adminAuthenticationFilter, adminExceptionTranslationFilter
/rpc/xmlrpc=httpSessionContextIntegrationFilter, basicAuthenticationFilter, wsExceptionTranslator
/rpc/rest/**=httpSessionContextIntegrationFilter, basicAuthenticationFilter, wsExceptionTranslator
/**=httpSessionContextIntegrationFilter, formAuthenticationFilter, rememberMeProcessingFilter, anonymousProcessingFilter, ex
.... add your implementation ...
</value>
</property>
</bean>

```

The following stanza lists the authentication providers that Clearspace tries to use. When trying to authenticate a user, Clearspace tries with each in turn, top to bottom. Each of the beans listed here

```

<!-- A list of authentication sources that will be consulted when attempting to
authenticate the user. Each is consulted in order until a provider does
*not* return null. This chains multiple providers together
until one decides it can handle the user. -->
<bean id="authenticationManager"
class="org.acegisecurity.providers.ProviderManager">
<property name="providers">
<list>
<ref bean="jiveLdapAuthenticationProvider"/>
<ref bean="jiveLegacyAuthenticationProvider"/>
<ref bean="daoAuthenticationProvider" />
<ref bean="rememberMeAuthenticationProvider"/>
<ref bean="anonymousAuthenticationProvider"/>
</list>
</property>
</bean>

```

## Upgrading API Uses

Some of the most important changes in version 2 required that the underlying API be changed. To get things working on version 2, you'll need to change not only references to these in your Java classes, but also references to them in your FreeMarker templates.

The following describes the most significant changes.

[Inject Dependencies with Spring](#) (page 15)

[Work with Containers for Projects and Spaces/Communities](#) (page 17)

[Use Transaction Support When Updating Multiple Tables](#) (page 18)

[Use Manager Interfaces to Work with Managed Things](#) (page 18)

[Handle Content As XML](#) (page 18)

[Other API Changes](#) (page 19)

## Inject Dependencies With Spring

For version 2, the Clearspace codebase was refactored to support [Spring](#), a framework with modules for inversion of control, transaction management, authentication, and more. This has potentially the biggest impact on your code because components that extend Clearspace must use the same conventions in order to interact reliably with Clearspace. For example, as described below, dependency injection replaces version 1 conventions for using `JiveContext` and factory classes to get manager instances.

**Dependency injection** is a way to obtain a dependency, such as an object on which your code relies, by having the instance "injected" at run time — that is, the instance is set with a JavaBeans-style accessor method. In Clearspace code, manager class instances are generally now injected rather than obtained via a context or factory class method.

A class that supports injection includes a class-level variable for holding the instance it depends on and provides a `set*` accessor method through which Spring can inject the instance (in other words, your code provides a property of the interface's type). Your setter implementation assigns the injected instance to the variable, which your code can then use to access the instance. By specifying the *interface* on which your code depends — rather than an implementation of the interface — you have a loosely-coupled dependency. This lets you avoid breakage that might occur if the implementation changes.

**Note:** You can ensure that the setter you provide actually has a configured type associated with it by [annotating the setter method with `@org.springframework.beans.factory.annotation.Required`](#). If the injected type is not known to Spring configuration, then the Spring container will throw an exception at run time.

The following example illustrates the basics of dependency injection with Spring: declare private class-level variables for manager instances and declare setter methods through which Spring will inject the instances. Code for version 1 techniques has been commented out in favor of the Spring conventions.

```
// Variables to hold the manager instances.
private DocumentManager documentManager;
private FreemarkerManager freemarkerManager;

// Setters for injecting manager instances.
public void setDocumentManager(DocumentManager documentManager) {
    this.documentManager = documentManager;
}
public void setFreemarkerManager(FreemarkerManager freemarkerManager) {
    this.freemarkerManager = freemarkerManager;
}

private String getDocContent(String documentID)
{
    // ... Declare variables ...

    // Don't use JiveContext (from JiveApplication) to get a DocumentManager
    // instance. The instance has been injected through the setter above.
    // DocumentManager documentManager = JiveApplication.getContext(AuthFactory
    //     .getSystemAuthToken()).getDocumentManager();

    // Use the manager to get a document by its ID.
    document = documentManager.getDocument(documentID);
    Map properties = new HashMap();
```

```

        // ... Set FreeMarker properties from the document, then apply
        // a template with them ...
        result = applyFreemarkerTemplate(properties, FREEMARKER_HTML_FILE);

    // ... catch exceptions ...

    return result;
}

private String applyFreemarkerTemplate(Map properties,
    String templateName) {

    // ... Declare variables ...

    // Don't use getInstance to get the FreemarkerManager instance. It has
    // been injected.
    // FreemarkerManager freemarkerManager = FreemarkerManager.getInstance();

    // ... Use the manager to set up FreeMarker configuration ...
    config = freemarkerManager.getConfiguration(ServletActionContext.getServletContext());
    if (properties != null) {
        // Process the FreeMarker template and store the resulting HTML as a String
        result = applyFreemarkerTemplate(config, properties, templateName);
    }

    // ... catch exceptions ...

    return result;
}

```

While you can optionally name the specific interface implementation you want to have injected, Spring in Clearspace supports a feature known as autowiring. In autowiring, you merely include the setter method (using JavaBeans naming conventions) with a single parameter of the interface's type. To optionally specify the implementation you want injected, you include a spring.xml file that associates the implementation class with your setter.

**Remove JiveContext uses that retrieve managers.** In version 1 the `JiveContext` interface was a popular convention to get instances of the various manager interfaces — `CommunityManager`, `UserManager`, `DocumentManager`, and so on. In version 2 this convention is, depending on the case, either unavailable or unreliable. For example, if you've written a plugin that has a static initializer that tries to bootstrap with call to `JiveContext`, Clearspace startup will likely fail.

Instead, use dependency injection as described above. In fact, to stay out of trouble, the general rule in version 2 is "Don't use `JiveContext`."

**Remove getInstance calls that retrieve managers.** This includes `FreemarkerManager.getInstance`, but also all of the `*Factory.getInstance` methods you might have used in version 1. In version 2 most of the factory classes have been removed. You should replace your calls to their `getInstance` methods with a property for setting the instance from Spring as described above. Here's a list of the removed classes:

- `GroupManagerFactory`
- `UserManagerFactory`
- `AvatarManagerFactory`

- BanManagerFactory
- PollManagerFactory
- SearchQueryLoggerFactory
- StatusLevelManagerFactory
- TagManagerFactory
- WidgetDAOFactory

## Work with Containers for Projects and Spaces/Communities

Version 2 introduces *projects*, which, like spaces (known as communities in Clearspace Community), collect content such as documents, discussions, and blogs. Projects also add tasks as a content type. To organize the conceptually similar projects and spaces, the API was changed to introduce the idea of a *container*. The interfaces that represent projects and spaces — `Project` and `Community`— now inherit from a common `JiveContainer` interface.

In practical terms, this means that some methods taking an instance of the `Community` interface now take a `JiveContainer` instance instead. The following code works to get the latest document in either a project or space, for example, because they inherit from `JiveContainer` and both can contain documents. To see how your code might be affected, search the Javadoc index for occurrences of `JiveContainer` in method signatures and return types.

```
public Document getLatestDocument(JiveContainer container) {
    if (getDocumentCount(container) == 0) {
        return null;
    }
    DocumentResultFilter filter = new DocumentResultFilter();
    filter.setSortOrder(DocumentResultFilter.DESENDING);
    filter.setSortField(JiveConstants.MODIFICATION_DATE);

    // Round down the lower date boundary by 1 day.
    filter.setModificationDateRangeMin(ThreadResultFilter
        .roundDate(container.getModificationDate(), 24 * 60 * 60));
    filter.setNumResults(10);
    Iterable<Document> documents = documentManager.getDocuments(container, filter);
    if (documents.iterator().hasNext()) {
        return documents.iterator().next();
    }
    else {
        return null;
    }
}
```

By the way, this `getLatestDocument` method is now exposed by the `DocumentManager` interface; in version 1 this method was exposed by the `Document` interface. Methods such as this one were moved as part of a larger effort to migrate such functionality into managers.

## Use Transaction Support When Updating Multiple Tables

Version 2 supports [transaction management through Spring with the `@Transactional` annotation](#). When the work of your code results in updates (including deletes) to data in multiple database tables, supporting [transactional behavior](#) can help to ensure that the updates are made consistently (that is, all of the effected tables are updated or none are).

If you're explicitly supporting transactions with `@Transactional`, you'll need to add an [AspectJ](#) compilation step. The AspectJ compiler weaves into your code the support needed to include at load time your transactional method calls in a transaction context.

You'll want to add transaction support if your code updates multiple database tables as while executing a method. While the need to support transactions is fairly rare if you're doing all your work through the Clearspace API, keep in mind that your calls to API `set*` methods might result in database updates. If you're not sure whether your code's work results in database updates, it's not a bad idea probably to add the `@Transactional` annotation to the method.

Adding support for transactions is pretty easy. Here's what you do:

- Add the `@Transactional` annotation to any methods whose execution might result in updates to multiple tables. The annotation is in the package [org.springframework.transaction.annotation](#).
- Compile your code with the [AspectJ compiler](#). There are also [Ant tasks](#).

## Use Manager Interfaces to Work with Managed Things

Much of the Clearspace code was rewritten to clarify (or create) relationships between instances of manager interfaces and instances of what they manage. The result is an API that's more intuitive, but it also might mean changes to your code.

In general, methods designed to handle *types of things*— documents, attachments, discussion messages, and so on — were moved from a "containing" type to a manager type. For example, in version 1 the method `getAttachmentCount` was exposed by the interfaces `BlogPost`, `Document`, and `ForumMessage`; each of these could have attachments, and `getAttachmentCount` was how you got the count of them. In version 2, you get the count of attachments with the `AttachmentManager.getAttachmentCount(AttachmentContentResource)` method. The `AttachmentContentResource` is a marker interface extended by `BlogPost`, `Document`, `ForumMessage`, and other things that can have attachments.

So in version 2 the rule of thumb, when looking for a way to manage types of things, is to look for a manager of those things.

## Handle Content As XML

Version 2 includes significant changes in the way Clearspace handles content. Improving the rich text editor (which blogs, documents, and discussions threads all use) included completely reworking the way that content is handled in conversions between plain text (wiki markup), rich text, and HTML (for previewed and published content).

In particular, content is set and received as XML. In version 1, the content-related accessors that get and set content body, subject and so on received and returned `String` instances. In version 2, these methods handle content as instances of `org.w3c.dom.Document`. Version 2 interfaces that used content accessor methods — including `Document`, `PrivateMessage`, `ForumMessage`, `BlogPost`, and others — still include methods such as `getPlainBody` for retrieving content without markup.

For plugins that include actions, you can extend `JiveActionSupport` for several methods helpful in converting content from one format to another.

While version 2 features a model for setting and getting content as XML, there's also an exception: the `Macro` interface `render` method still returns content as a `String`. [It has to parse as well-formed XML \(page 7\)](#) , but it's still a string.

## Other API Changes

- In some cases, methods that took a `JiveObject` instance now take an instance of an interface that extends `JiveObject`. The marker interfaces `CommentContentResource` and `AttachmentContentResource` were created to indicate an object's support for comments or attachments. Look for these types in the Javadoc index to find out whether you need to update your own code.