

Building Plugins

By building plugins, you can add new features to the application. This includes new widgets and other UI features, as well as back-end components such as web services or custom authentication providers.

Getting Sample Code

You can use Jive's public [Subversion](https://svn.jivesoftware.com/svn/dev/repos/jive/) repository to get sample code. Check out the code at the following URL: <https://svn.jivesoftware.com/svn/dev/repos/jive/>

What You Can Do With Plugins

With a plugin, you can add new features. The plugin framework supports several kinds of components.

- With **widgets**, users can customize the application's overview pages to display the content they want. You can build widgets that give the specialized windows into content and other data.
- **Macros** provide easy ways to enhance content while editing it. In the content editor, the Insert menu displays macros. (Note: This release doesn't support adding new macros. You'll be able to develop your own in a coming release.)
- **Actions** can add new links and pages to the user interface.
- Many features are implemented as Spring beans. You can override these or provide your own. One example is to override security-related beans. For more information, see [Authentication and Authorization](#).

What You Can Do *Without* Plugins

If you're writing a web service client, you don't need a plugin. For more information, see the [REST Web Services Reference](#) and the [SOAP Web Services Developer Guide](#).

Requirements

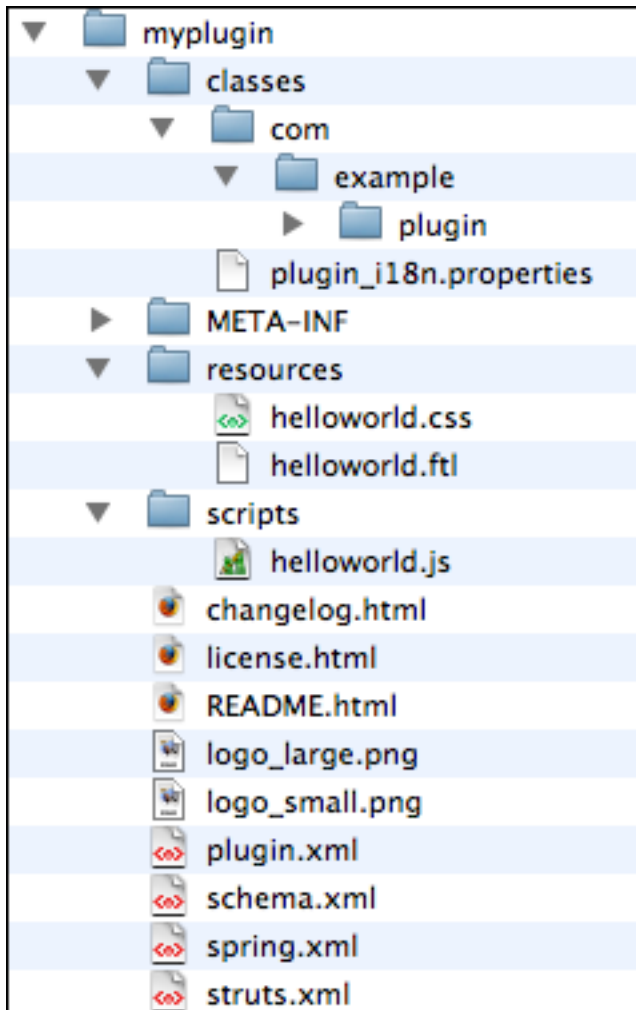
To write plugins you'll need the following:

- An instance to test and debug against, deployed on a supported environment. Take a look at the [installation guide](#) for details on supported environments. For example, you'll need to use Java 6 (JDK 1.6).

- (Optional) Building plugins is easier if you use [Maven](#), a Java build tool. Like [Ant](#), Maven provides a way to account for dependencies and to compile your source files.
- (Optional) A Java IDE. Jive Software recommends development environments such as [JetBrain's IntelliJ IDEA](#) or [Eclipse](#). You can use Maven to quickly create plugin projects for use with these tools. (Of course, plain text editors like [vi](#) or [Emacs](#) will work just fine as well.)

Packaging and Deploying a Plugin

However you get your plugin to its finished state, you'll deploy it as a JAR file. When you deploy, the application expects the contents of your JAR file to be in a particular hierarchy. Here's a snapshot of how the contents might look:



Artifact	Description
classes directory	Required. Java classes for your plugin.
META-INF directory	Directory for standard JAR descriptor file.
resources directory	Plugin user interface files, including FTL files, CSS files, and so on go here.
scripts directory	If you've got Javascript or Perl or some such, put it here.
README.html, changelog.html, and license.html	Your plugin's users (including system administrators) will appreciate having information about how the plugin should be deployed, configured and used. As you upgrade the plugin, you can also include information about changes you make from version to version.
logo_small.png, logo_large.png	Images to represent the plugin (logo_small must be 16x16 pixels).
plugin.xml	Required. Configuration for the plugin. Lists components included in the plugin, along with components added to the UI. See Plugin XML Reference and UI Components XML Reference for more information.
schema.xml	If your plugin creates tables in the database, define them here as SQLGen XML. Tables you define in this file will be automatically created when the plugin is deployed. For more on integrating database tables, see Accessing the Database from a Plugin .
spring.xml	If the plugin creates or overrides Spring beans, configure them here. For an example in the context of security, see Authentication and Authorization .
struts.xml	If the plugin includes any Struts actions, use this standard struts configuration file to map actions to action classes and results to FTL files.

Configuring a Plugin

Your plugin.xml file contains nearly all of the high-level information about what's included in your plugin. What's not described here could include any Spring beans you're overriding or database tables you're adding.

You'll find references for plugin.xml file elements in [Plugin XML Reference](#) and [UI Components XML Reference](#).

```
<plugin>
  <!--
    Top-level information: overall plugin name and description,
    its version, and the earliest application version on which
```

Building Plugins

```
    it's supported.
-->
<name>helloworld</name>
<description>Hello World</description>
<author>ACME Plugins</author>
<version>1.0.0</version>
<minServerVersion>2.1.0</minServerVersion>

<!--
    Widgets are user interface components that people can
    use to customize parts of the UI. The class attribute
    points to the Java class that provides the widget's logic.
-->
<widget class="com.example.sbs.widgets.CoolWidget"/>

<!--
    You can write web services that expose parts of the application
    not available as web services by default.
-->
<webservice class="com.example.sbs.webservices.HandyService"/>

<!--
    Specifies that this plugin includes a CSS stylesheet that
    should be used throughout the application. This is good
    way to add a global CSS class or override one already
    included with the application.
-->
<css src="/acme_styles.css"/>

<!--
    Key and value for indicating whether database changes are
    needed on upgrade.
-->
<databaseKey>helloworld</databaseKey>
<databaseVersion>1</databaseVersion>

<!--
    Aside from widgets, you can add elements to the user interface
    with Struts actions. You integrate into the UI here, then define
    the action itself
-->
<components>
  <!--
    <component> elements define user interface elements
    to be integrated. For a description of these, see
    Integrating Plugin UI. This one adds
    an Action box link to a user profile; this will be
    seen by visiting users, rather than the profile's owner.
  -->
  <component id="profile-actions">
    <tab id="profile-actions-tab">
      <item id="profile-actions-link" name="Example profile action">
        <url>
```

```

        <![CDATA[<@s.url value="/example-profile.jspa?userID=${targetUser.ID}"/>]]></url>
    </item>
</tab>
</component>
</components>

<!-- A class that handles lifecycle events. -->
<class>com.jivesoftware.plugins.MyPluginLifecycle</class>

</plugin>

```

Handling Lifecycle Events

You can write a class that handles events from the application lifecycle. Your lifecycle class implements the interface `com.jivesoftware.base.plugin.Plugin`, which has two methods: `init` and `destroy`. Use the `init` method to perform actions (such as create connections to resources) that your plugin will likely need throughout its life. Use the `destroy` method to release resources and perform actions that are the last things your plugin should do.

You specify the presence of a lifecycle class with the `plugin.xml` class element:

```
<class>com.example.plugin.MyPluginLifecycle</class>
```

Adding Database Access

You add database tables to support your plugin. When adding database support, you plug into the Spring context by creating data access object (DAO) Spring beans. You add your tables through a `schema.xml` file.

For more on adding database access to your plugin, see [Accessing the Database from a Plugin](#).

Deploying Plugins

The best way to deploy a plugin is by [using the admin console](#). In the console, go to System > Plugins > Add Plugin.

Note that whenever you add or remove a plugin, you'll need to restart the application server.

