Building Actions

You can build plugins that enhance Jive SBS with links to new functionality, additions to the admin console, and additions to the end user UI.

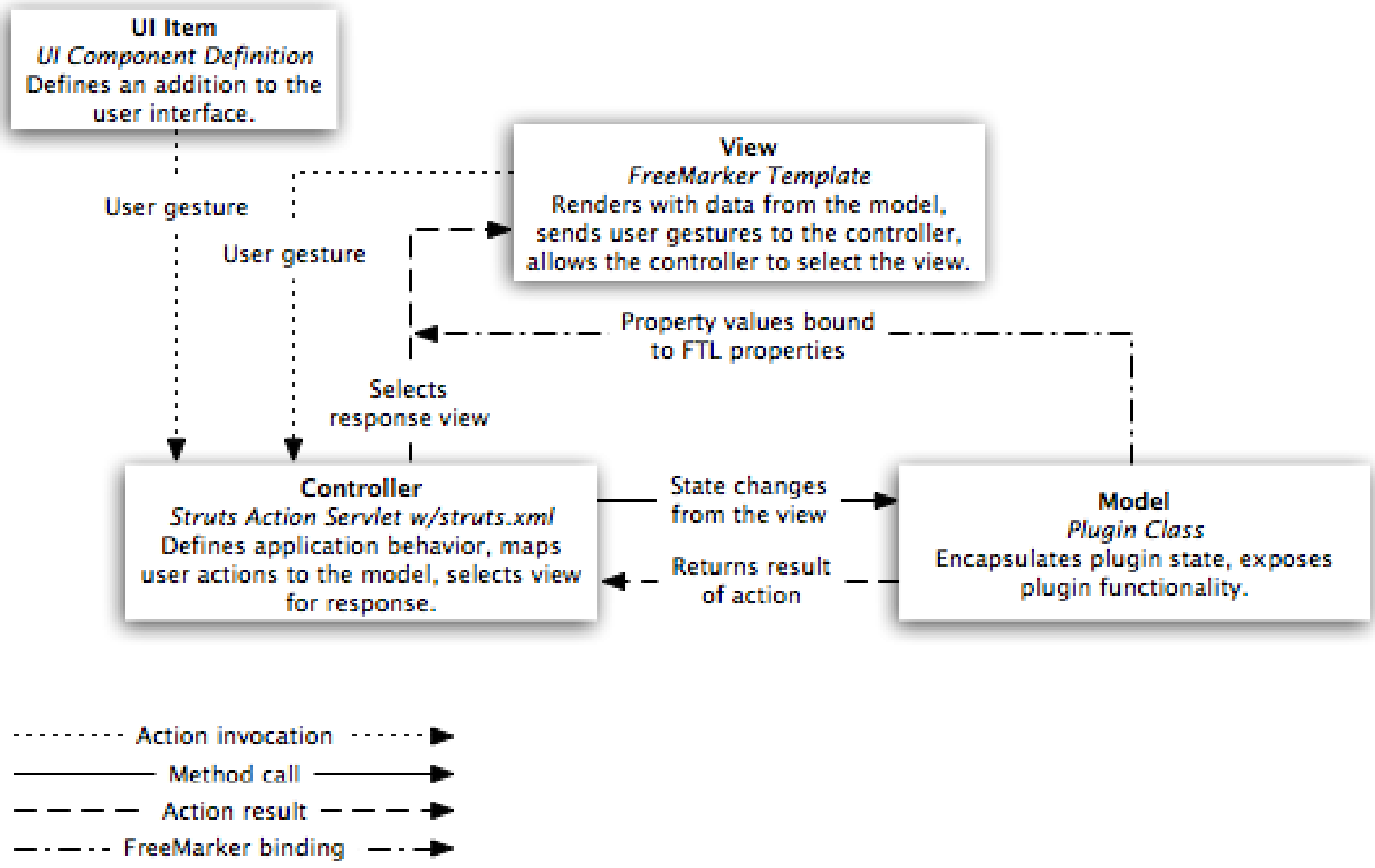
Action plugins incorporate Struts actions, in which a user gesture (such as clicking a link) directs processing to an action class whose code executes, then typically routes processing back to the user interface to display some result.

The simple plugin described here displays a link that, when clicked, displays a new page showing a greeting. There are simpler ways to do this, of course, but hopefully you'll see how much more you can do in your action class and how useful it is to separate that code from your user interface.

Action Plugin Basics

In the model-view-controller (MVC) architecture that the application is based on, the idea is to separate code that manages logic around data (the model) from code that presents the data as user interface (the view) from code that controls interactions between the two (the controller).

Action plugins use the same model on a smaller scale. The following illustrates how it works in a plugin:



Here's a brief overview of what happens among the pieces.

1. Things get started when someone acts on a UI component you've defined in your plugin.xml file (one part of the view). For example, they might click a link defined in a <component> stanza.
2. The <component> stanza usually connects the user action to a URL that displays a page or invokes an action defined through your controller. (You configure this in your struts.xml file, but the controller is technically a Struts action servlet beneath your code.) If the URL is simply a page of UI (such as an FTL or JSP page,
3. The controller (struts.xml file) knows that your plugin class corresponds to the action. It interacts with the model by calling a method of the action class, passing in any arguments defined as parameters of the request.
4. Logic in the model (plugin action class) executes, sending an appropriate response back to the controller based on the data it received.
5. The controller (struts.xml file) describes which of your views (FreeMarker files) to display based on the plugin class's response. It invokes FreeMarker to display the view, which uses action class properties (getter methods) to retrieve values to display in the rendered page.
6. With the new view (user interface page) before them, the person who started everything interacts with the UI, such as by clicking links. These gestures in the updated view invoke actions you've defined in the controller (struts.xml file). For each action the user invokes by doing something in your UI, the cycle will pick up again at step 3.

The code used in this topic is a simple action that adds a menu item to the Browse menu of the user bar (that row of menus along the top of every application page). That "Say Hello" menu item will simply navigate to a new page that displays a simple greeting. Here are the pieces described:

- A SimpleAction Java class that will do the work of returning the greeting to present. This is the action class.
- A simple-template.ftl FreeMarker template that will present the greeting.
- A struts.xml file that will map the interaction, telling the application to look at SimpleAction for the data needed and at simple-template.ftl for presenting it.
- The plugin.xml file, where you'll include descriptive info for this new functionality.

Create a plugin.xml File for Configuration

Every plugin has one. This file tells what's in the plugin — in short, what features you're extending — and where to find code and other supporting files your plugin needs (although not necessarily all of them, as you'll see in the next section).

The url element specifies the URL that points to the action you'll create. This is your hook from your plugin's user interface into the action that provides its functionality.

```
<plugin xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://www.jivesoftware.com/schemas/clearspace/2_5/plugin.xsd">
    <name>SimpleExamples</name>
    <description>Simple macro and plugin examples.</description>
    <author>Me</author>
    <version>1.0.0</version>
    <minServerVersion>1.3</minServerVersion>

    <!-- The <components> element contains additions or customizations to
     components in the user interface. Each <component> child element
     represents a different UI piece, with its id attribute value
     specifying which piece is being customized.

     Here, you're customizing the user bar (the menu bar at the top of
     each page) so that its Browse menu gets a new entry
     item, "Say Hello". The <url> element here specifies the URL that
     will be loaded when the user clicks the item. In this
     case, the application will execute the sayhello action (which is defined
     in the struts.xml file).
    -->
    <components>
        <component id="user-bar">
            <tab id="jiveUserMenu5">
                <item id="sayhello" name="Say Hello">
```

```
            <url><![CDATA[<@ww.url action="sayhello" />]]></url>
          </item>
        </tab>
      </component>
   </components>
</plugin>
```

Pretty simple, really. The `<plugin>` element's children include information about where the plugin is coming from (you), its version (in case you revise it for upgrade), and so on. The `<minServerVersion>` element specifies the minimum version that the plugin will run on (it won't be deployed on earlier versions). The code here tells the application to add a new "Say Hello" link to the Browse menu on the user bar. It also says which action (as defined in the Struts file you'll create in a moment) should be executed when the user clicks the link.

Create a Java Class for the Action's Logic

A Java action class is the plugin's "model." This is where the plugin gets the data it needs, in this case a simple greeting. A FreeMarker template you create in a moment will display the data.

```
package com.example.clearspace.plugin.action;

import com.jivesoftware.community.action.JiveActionSupport;

/**
 * A "Hello World" plugin that merely receives or returns a greeting.
 * The JavaBeans-style accessors in this class are mapped to
 * property names in resources/simple-template.ftl, which is the FreeMarker
 * template the provides UI for displaying the data. The mapping is
 * done by Struts after the class-to-FTL mapping in the
 * struts.xml file.
 *
 * In other words, this class represents the plugin's data "model,"
 * the FTL file provides its "view," and the code in the struts.xml
 * file provides its "controller."
 */
public class SimpleAction extends JiveActionSupport {

   private static final long serialVersionUID = 1L;

   private String message = "Hello World";

   /**
    * Gets the greeting message. This method is mapped by Struts to the
    * $message property used in simple-template.ftl. In other words,
    * in rendering the user interface, the app (via Struts) maps
    * the $message property to a "getter" name of the form
    * get<property_name> -- this getMessage method.
    *
```

```
 * @return The greeting text.
 */
public String getMessage() {
    return message;
}

/**
 * Sets the greeting message. The FTL file doesn't provide a way
 * for the user to set the message text. But if it did, this is
 * the method that would be called.
 *
 * @param message The greeting text.
 */
public void setMessage(String message) {
    this.message = message;
}
}
```

Create a FreeMarker Template to Provide UI for the Action's Results

Your action has a "view," or user interface. One important thing to notice is that the plugin class you just created and the view code you're about to create are in a way connected by a naming convention. In other words, the getMessage method in the class is matched up with the message variable in the code below through a convention in which the app knows that removing the method name's "get" and lower-casing the first letter "m" creates a match with the variable. (Although, actually Struts does all that behind the scenes.) You'll enable that mapping through the Struts file you'll create in a moment.

Note: The Struts documentation has introductory information on using FreeMarker with Struts actions.

```
<html>
  <head>
     <!-- Create a FreeMarker variable for the page title bar text, then
          use that variable in the <title> element. -->
    <#assign pageTitle="Hello World" />
    <title>${pageTitle}</title>
    <content tag="pagetitle">${pageTitle}</content>
  </head>
  <body>
     <!-- Have the message appear a little down on the page and
          in the center, so it's easier to find. -->
     <br/>
    <p align="center">${message}</p>
  </body>
```

```
</html>
```

Create a Struts File to Map the Action to the Logic Class

This is where you'll connect the pieces. The code below defines an action that is associated with the SimpleAction action class. A "success" result returned by the class (something it does by default in this case) tells the application to go get the simple-template.ftl template and process it by merging in the data that it retrieved from the class.

```
<!DOCTYPE struts PUBLIC -//Apache Software Foundation//DTD Struts Configuration 2.0//EN
    http://struts.apache.org/dtds/struts-2.0.dtd>
<struts>
  <package name="example-actions" namespace="" extends="community-custom">
    <!-- Map the action name, sayhello, to the action class, SimpleAction. -->
    <action name="sayhello" class="com.example.clearspace.plugin.action.SimpleAction">
      <!-- Specify the FTL file that should be used to present the data in the case of
          a "success" result (the default for an action class). -->
      <result name="success">/plugins/simpleexamples/resources/simple-template.ftl</result>
    </action>
  </package>
</struts>
```

That's it for this introduction to action plugins. As you can imagine, your action class could do much more — retrieve data from an external source (or from the application database using its API), perform calculations on data entered by the user, and so on. You could have multiple FTL files to provide different user interface responses to your plugin's state, such as what is returned by your action class.