Simple Web Applications

One of the major benefits of using Jython is the ability to make use of Java platform capabilities programming in the Python programming language instead of Java. In the Java world today, the most widely used web development technique is the Java servlet. Now in JavaEE, there are techniques and frameworks used so that we can essentially code HTML or other markup languages as opposed to writing pure Java servlets. However, sometimes writing a pure Java servlet still has its advantages. We can use Jython to write servlets and this adds many more advantages above and beyond what Java has to offer because now we can make use of Python language features as well. Similarly, we can code web start applications using Jython instead of pure Java to make our lives easier. Coding these applications in pure Java has proven sometimes to be a difficult and sometimes grueling task. We can use some of the techniques available in Jython to make our lives easier. We can even code WSGI applications with Jython making use of the *modjy* integration in the Jython project.

In this chapter, we will cover three techniques for coding simple web applications using Jython: servlets, web start, and WSGI. We'll get into details on using each of these different techniques here, but we will discuss deployment of such solutions in Chapter 17.

Servlets

Servlets are a Java platform technology for building web-based applications. They are a platform- and server-independent technology for serving content to the web. If you are unfamiliar with Java servlets, it would be worthwhile to learn more about them. An excellent resource is wikipedia (http://en.wikipedia.org/wiki/Java_Servlet); however, there are a number of other great places to find out more about Java servlets. Writing servlets in Jython is a very productive and easy way to make use of Jython within a web application. Java servlets are rarely written using straight Java anymore. Most Java developers make use of Java Server Pages (JSP), Java Server Faces (JSF), or some other framework so that they can use a markup language to work with web content as opposed to only working with Java code. However, in some cases it is still quite useful to use a pure Java servlet. For these cases we can make our lives easier by using Jython instead. There are also great use-cases for JSP; similarly, we can use Jython for implementing the logic in our JSP code. The latter technique allows us to apply a model-view-controller (MVC) paradigm to our programming model, where we separate our front-end markup from any implementation logic. Either technique is rather easy to implement, and you can even add this functionality to any existing Java web application without any trouble.

Another feature offered to us by Jython servlet usage is dynamic testing. Because Jython compiles at runtime, we can make code changes on the fly without recompiling and redeploying our web application. This can make it very easy to test web applications, because usually the most painful part of web application development is the wait time between deployment to the servlet container and testing.

Configuring Your Web Application for Jython Servlets

Very little needs to be done in any web application to make it compatible for use with Jython servlets. Jython contains a built-in class named *PyServlet* that facilitates the creation of Java servlets using Jython source files. We can make use of PyServlet quite easily in our application by adding the necessary XML configuration into the application's web.xml descriptor such that the *PyServlet* class gets loaded at runtime and any file that contains the *.py* suffix will be passed to it. Once this configuration has been added to a web application, and *jython.jar* has been added to the CLASSPATH then the web application is ready to use Jython servlets. See Listing 13-1.

Listing 13-1. Making a Web Application Compatible with Jython

```
<servlet>
    <servlet-name>PyServlet</servlet-name>
    <servlet-class>org.python.util.PyServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>PyServlet</servlet-name>
    <url-pattern>*.py</url-pattern>
</servlet-mapping>
```

Any servlet that is going to be used by a Java servlet container also needs to be added to the *web.xml* file as well, since this allows for the correct mapping of the servlet via the URL. For the purposes of this book, we will code a servlet named *NewJythonServlet* in the next section, so the following XML configuration will need to be added to the web.xml file. See Listing 13-2.

Listing 13-2. Coding a Jython Servlet

```
<servlet>
    <servlet-name>NewJythonServlet</servlet-name>
    <servlet-class>NewJythonServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>NewJythonServlet</servlet-name>
    <url-pattern>/NewJythonServlet</url-pattern>
</servlet-mapping>
```

Writing a Simple Servlet

In order to write a servlet, we must have the *javax.servlet.http.HttpServlet* abstract Java class within our CLASSPATH so that it can be extended by our Jython servlet to help facilitate the code. This abstract class, along with the other servlet implementation classes, is part of the *servlet-api.jar* file. According to the abstract class, there are two methods that we should override in any Java servlet, those being *doGet* and *doPost*. The former performs the HTTP

GET operation while the latter performs the HTTP POST operation for a servlet. Other commonly overridden methods include *doPut*, *doDelete*, and *getServletInfo*. The first performs the HTTP PUT operation, the second performs the HTTP DELETE operation, and the last provides a description for a servlet. In the following example, and in most use-cases, only the *doGet* and *doPost* are used.

Let's first show the code for an extremely simple Java servlet. This servlet contains no functionality other than printing its name along with its location in the web application to the screen. Following that code we will take a look at the same servlet coded in Jython for comparison (Listing 13-3).

Listing 13-3. NewJavaServlet.java

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class NewJavaServlet extends HttpServlet {
    protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
    throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet NewJavaServlet Test</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>Servlet NewJavaServlet at " +
            request.getContextPath () + "</h1>");
            out.println("</body>");
            out.println("</html>");
        } finally {
            out.close();
        }
    }

    @Override
    protected void doGet(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {
        processRequest(request, response);
    }

    @Override
    protected void doPost(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {
        processRequest(request, response);
```

```
    }

    @Override
    public String getServletInfo() {
        return "Short description";
    }
}
```

All commenting has been removed from the code in an attempt to make the code a bit shorter. Now, Listing 13-4 is the equivalent servlet code written in Jython.

Listing 13-4.

```
from javax.servlet.http import HttpServlet

class NewJythonServlet (HttpServlet):
    def doGet(self,request,response):
        self.doPost (request,response)

    def doPost(self,request,response):
        toClient = response.getWriter()
        response.setContentType ("text/html")
        toClient.println ("<html><head><title>Jython Servlet Test</title>"
        +
        "<body><h1>Servlet Jython Servlet at" +
        request.getContextPath() + "</h1></body></html>")

    def getServletInfo(self):
        return "Short Description"
```

Not only is the concise code an attractive feature, but also the easy development lifecycle for working with dynamic servlets. As stated previously, there is no need to redeploy each time you make a change because of the compile at runtime that Jython offers. Simply change the Jython servlet, save, and reload the webpage to see the update. If you begin to think about the possibilities you'll realize that the code above is just a basic example, you can do anything in a Jython servlet that you can with Java and even most of what can be done using the Python language as well.

To summarize the use of Jython servlets, you simply include *jython.jar* and *servlet-api.jar* in your CLASSPATH. Add necessary XML to the web.xml, and then finally code the servlet by extending the javax.servlet.http.HttpServlet abstract class.

Using JSP with Jython

Harnessing Jython servlets allows for a more productive development lifecycle, but in certain situations Jython code may not be the most convenient way to deal with front-facing web code. Sometimes using a markup language such as HTML works better for developing sophisticated front-ends. For instance, it is easy enough to include JavaScript code within a Jython servlet. However, all of the JavaScript code would be written within the context of a String. Not only

does this eliminate the usefulness of an IDE for situations such as semantic code coloring and auto completion, but it also makes code harder to read and understand. Cleanly separating such code from Jython or Java makes code more clear to read, and easier to maintain in the long run. One possible solution would be to choose from one of the Python template languages such as Django, but using Java Server Pages (JSP) technology can also be a nice solution.

Using a JSP allows one to integrate Java code into HTML markup in order to generate dynamic page content. We are not fans of JSP. There, we said it: JSP can make code a living nightmare if the technology is not used correctly. Although JSP can make it very easy to mix JavaScript, HTML, and Java into one file, it can make maintenance very difficult. Mixing Java code with HTML or JavaScript is a bad idea. The same would also be true for mixing Jython and HTML or JavaScript.

The Model-View-Controller (MVC) paradigm allows for clean separation between logic code, such as Java or Jython, and markup code such as HTML. JavaScript is always gets grouped into the same arena as HTML because it is a client-side scripting language. In other words, JavaScript code should also be separated from the logic code. In thinking about MVC, the controller code would be the markup and JavaScript code used to capture data from the end-user. Model code would be the business logic that manipulates the data. Model code is contained within our Jython or Java. The view would be the markup and JavaScript displaying the result.

Clean separation using MVC can be achieved successfully by combining JSP with Jython servlets. In this section we will take a look at a simple example of how to do so. As with many of the other examples in this text it will only brush upon the surface of great features that are available. Once you learn how to make use of JSP and Jython servlets you can explore further into the technology.

Configuring for JSP

There is no real configuration above and beyond that of configuring a web application to make use of Jython servlets. Add the necessary XML to the web.xml deployment descriptor, include the correct JARs in your application, and begin coding. What is important to note is that the *.py* files that will be used for the Jython servlets must reside within your CLASSPATH. It is common for the Jython servlets to reside in the same directory as the JSP web pages themselves. This can make things easier, but it can also be frowned upon because this concept does not make use of packages for organizing code. For simplicity sake, we will place the servlet code into the same directory as the JSP, but you can do it differently.

Coding the Controller/View

The view portion of the application will be coded using markup and JavaScript code. Obviously, this technique utilizes JSP to contain the markup, and the JavaScript can either be embedded directly into the JSP or reside in separate *.js* files as needed. The latter is the preferred method

in order to make things clean, but many web applications embed small amounts of JavaScript within the pages themselves.

The JSP in this example is rather simple, there is no JavaScript in the example and it only contains a couple of input text areas. This JSP will include two forms because we will have two separate submit buttons on the page. Each of these forms will redirect to a different Jython servlet, which will do something with the data that has been supplied within the input text. In our example, the first form contains a small textbox in which the user can type any text that will be redisplayed on the page once the corresponding submit button has been pressed. Very cool, eh? Not really, but it is of good value for learning the correlation between JSP and the servlet implementation. The second form contains two text boxes in which the user will place numbers; hitting the submit button in this form will cause the numbers to be passed to another servlet that will calculate and return the sum of the two numbers. Listing 13-5 is the code for this simple JSP.

Listing 13-5. JSP Code for a Simple Controller/Viewer Application

testJSP.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Jython JSP Test</title>
    </head>
    <body>
        <form method="GET" action="add_to_page.py">
            <input type="text" name="p">
            <input type="submit">
        </form>
        <br/>
        <p>${page_text}</p>
        <br/>
        <form method="GET" action="add_numbers.py">
            <input type="text" name="x">
            +
            <input type="text" name="y">
            =
            ${sum}
            <br/>
            <input type="submit" title="Add Numbers">
        </form>
    </body>
</html>
```

In this JSP example, you can see that the first form redirects to a Jython servlet named *add_to_page.py, which plays the role of the controller*. In this case, the text that is

contained within the input textbox named *p* will be passed into the servlet, and redisplayed in on the page. The text to be redisplayed will be stored in an attribute named *page_text*, and you can see that it is referenced within the JSP page using the ${} notation. Listing 13-6 is the code for*add_to_page.py*.

Listing 13-6. A Simple Jython Controller Servlet

```
######################################################################
# add to page.py
#
# Simple servlet that takes some text from a web page and redisplays
# it.
######################################################################
import java, javax, sys

class add_to_page(javax.servlet.http.HttpServlet):
    def doGet(self, request, response):
        self.doPost(request, response)

    def doPost(self, request, response):
        addtext = request.getParameter("p")
        if not addtext:
            addtext = ""
        request.setAttribute("page_text", addtext)
        dispatcher = request.getRequestDispatcher("testJython.jsp")
        dispatcher.forward(request, response)
```

Quick and simple, the servlet takes the request and obtains value contained within the parameter *p*. It then assigns that value to a variable named *addtext*. This variable is then assigned to an attribute in the request named *page_text* and forwarded back to the *testJython.jsp* page. The code could just as easily have forwarded to a different JSP, which is how we'd go about creating a more in-depth application.

The second form in our JSP takes two values and returns the resulting sum to the page. If someone were to enter text instead of numerical values into the text boxes then an error message would be displayed in place of the sum. While very simplistic, this servlet demonstrates that any business logic can be coded in the servlet, including database calls, and so on. See Listing 13-7.

Listing 13-7. Jython Servlet Business Logic

```
######################################################################
# add_numbers.py
#
# Calculates the sum for two numbers and returns it.
######################################################################
import javax

class add_numbers(javax.servlet.http.HttpServlet):
    def doGet(self, request, response):
```

```
        self.doPost(request, response)

    def doPost(self, request, response):
        x = request.getParameter("x")
        y = request.getParameter("y")
        if not x or not y:
            sum = "<font color='red'>You must place numbers in each value
box</font>"
        else:
            try:
                sum = int(x) + int(y)
            except ValueError, e:
                sum = "<font color='red'>You must place numbers only in each
value box</font>"
        request.setAttribute("sum", sum)

        dispatcher = request.getRequestDispatcher("testJython.jsp")
        dispatcher.forward(request, response)
```

If we add the JSP and the servlets to the web application we created in the previous Jython Servlet section, then this example should work out-of-the-box.

It is also possible to embed code into Java Server Pages by using various template tags known as scriptlets to enclose the code. In such cases, the JSP must contain Java code unless a special framework such as the Bean Scripting Framework (http://jakarta.apache.org/bsf/) is used along with JSP. For more details on using Java Server Pages, please take a look at the Sun Microsystems JSP documentation (http://java.sun.com/products/jsp/docs.html) or pick up a book such as *Beginning JSP, JSF and Tomcat Web Development: From Novice to Professional* from Apress.

Applets and Java Web Start

At the time of this writing, applets in Jython 2.5.0 are not yet an available option. This is because applets must be statically compiled and available for embedding within a webpage using the <*applet*> or <*object*> tag. The static compiler known as *jythonc***has been removed in Jython 2.5.0 in order to make way for better techniques. Jythonc was good for performing certain tasks, such as static compilation of Jython applets, but it created a disconnect in the development lifecycle as it was a separate compilation step that should not be necessary in order to perform simple tasks such as Jython and Java integration. In a future release of Jython, namely 2.5.1 or another release in the near future, a better way to perform static compilation for applets will be included.

For now, in order to develop Jython applets you will need to use a previous distribution including *jythonc* and then associate them to the webpage with the<*applet*> or <*object*> tag. In Jython, applets are coded in much the same fashion as a standard Java applet. However, the resulting lines of code are significantly smaller in Jython because of its sophisticated syntax. GUI development in general with Jython is a big productivity boost compared to developing a

Java Swing application for much the same reason. This is why coding applets in Jython is a viable solution and one that should not be overlooked.

Another option for distributing GUI-based applications on the web is to make use of the Java Web Start technology. The only disadvantage of creating a web start application is that it cannot be embedded directly into any web page. A web start application downloads content to the client's desktop and then runs on the client's local JVM. Development of a Java Web Start application is no different than development of a standalone desktop application. The user interface can be coded using Jython and the Java Swing API, much like the coding for an applet user interface. Once you're ready to deploy a web start application then you need to create a Java Network Launching Protocol (JNLP) file that is used for deployment and bundle it with the application. After that has been done, you need to copy the bundle to a web server and create a web page that can be used to launch the application.

In this section we will develop a small web start application to demonstrate how it can be done using the object factory design pattern and also using pure Jython along with the standalone Jython JAR file for distribution. Note that there are probably other ways to achieve the same result and that these are just a couple of possible implementations for such an application.

Coding a Simple GUI-Based Web Application

The web start application that we will develop in this demonstration is very simple, but they can be as advanced as you'd like in the end. The purpose of this section is not to show you how to develop a web-based GUI application, but rather, the process of developing such an application. You can actually take any of the Swing-based applications that were discussed in the GUI chapter and deploy them using web start technology quite easily. As stated in the previous section, there are many different ways to deploy a Jython web start application. We prefer to make use of the object factory design pattern to create simple Jython Swing applications. However, it can also be done using all .py files and then distributed using the Jython stand-alone JAR file. We will discuss each of those techniques in this section. We find that if you are mixing Java and Jython code then the object factory pattern works best. The JAR method may work best for you if developing a strictly Jython application.

Object Factory Application Design

The application we'll be developing in this section is a simple GUI that takes a line of text and redisplays it in JTextArea. We used Netbeans 6.7 to develop the application, so some of this section may reference particular features that are available in that IDE. To get started with creating an object factory web start application, we first need to create a project. We created a new Java application in Netbeans named *JythonSwingApp* and then added *jython.jar* and*plyjy.jar* to the classpath.

First, create the *Main.java* class which will really be the driver for the application. The goal for Main.java is to use the Jython object factory pattern to coerce a Jython-based Swing application into Java. This class will be the starting point for the application and then the Jython code will

perform all of the work under the covers. Using this pattern, we also need a Java interface that can be implemented via the Jython code, so this example also uses a very simple interface that defines a *start()* method which will be used to make our GUI visible. Lastly, the Jython class named below is the code for our *Main.java* driver and the Java interface. The directory structure of this application is as shown in Listing 13-8.

Listing 13-8. Object Factory Application Code

JythonSwingApp

JythonSimpleSwing.py

jythonswingapp

Main.java

jythonswingapp.interfaces

JySwingType.java

Main.java

```
package jythonswingapp;

import jythonswingapp.interfaces.JySwingType;
import org.plyjy.factory.JythonObjectFactory;

public class Main {
    JythonObjectFactory factory;

    public static void invokeJython(){
        JySwingType jySwing = (JySwingType) JythonObjectFactory
        .createObject(JySwingType.class, "JythonSimpleSwing");
        jySwing.start();
    }

    public static void main(String[] args) {
        invokeJython();
    }
}
```

As you can see, *Main.java* doesn't do much else except coercing the Jython module and invoking the *start()* method. In Listing 13-9, you will see the*JySwingType.java* interface along with the implementation class that is obviously coded in Jython.

Listing 13-9. JySwingType.java Interface and Implementation

JySwingType.java

```
package jythonswingapp.interfaces;

public interface JySwingType {
    public void start();
}
```

JythonSimpleSwing.py

```
import javax.swing as swing
import java.awt as awt
from jythonswingapp.interfaces import JySwingType
import add_player as add_player
import Player as Player

class JythonSimpleSwing(JySwingType, object):
    def __init__(self):
        self.frame=swing.JFrame(title="My Frame", size=(300,300))
        self.frame.defaultCloseOperation=swing.JFrame.EXIT_ON_CLOSE
        self.frame.layout=awt.BorderLayout()
        self.panel1=swing.JPanel(awt.BorderLayout())
        self.panel2=swing.JPanel(awt.GridLayout(4,1))
        self.panel2.preferredSize = awt.Dimension(10,100)
        self.panel3=swing.JPanel(awt.BorderLayout())
        self.title=swing.JLabel("Text Rendering")
        self.button1=swing.JButton("Print Text",
actionPerformed=self.printMessage)
        self.button2=swing.JButton("Clear Text",
actionPerformed=self.clearMessage)
        self.textField=swing.JTextField(30)
        self.outputText=swing.JTextArea(4,15)

        self.panel1.add(self.title)
        self.panel2.add(self.textField)
        self.panel2.add(self.button1)
        self.panel2.add(self.button2)
        self.panel3.add(self.outputText)

        self.frame.contentPane.add(self.panel1,
awt.BorderLayout.PAGE_START)
        self.frame.contentPane.add(self.panel2, awt.BorderLayout.CENTER)
        self.frame.contentPane.add(self.panel3, awt.BorderLayout.PAGE_END)

    def start(self):
        self.frame.visible=1

    def printMessage(self,event):
        print "Print Text!"
        self.text = self.textField.getText()
        self.outputText.append(self.text)

    def clearMessage(self, event):
        self.outputText.text = ""
```

If you are using Netbeans, when you clean and build your project a JAR file is automatically generated for you. However, you can easily create a JAR file at the command-line or terminal by ensuring that the *JythonSimpleSwing.py* module resides within your classpath and using the *java -jar* option. Another nice feature of using an IDE such as Netbeans is that you can make this into a web-start application by going into the project properties and checking a couple of boxes. Specifically, if you go into the project properties and select *Application - Web Start* from the left-hand menu, then check the *Enable Web Start* option then the IDE will take care of generating the necessary files to make this happen. Netbeans also has the option to self sign the JAR file which is required to run most applications on another machine via web start. Go ahead and try it out, just ensure that you clean and build your project again after making the changes.

To manually create the necessary files for a web start application, you'll need to generate two additional files that will be placed outside of the application JAR. Create the JAR for your project as you would normally do, and then create a corresponding JNLP file which is used to launch the application, and an HTML page that will reference the JNLP. The HTML page obviously is where you'd open the application if running it from the web. Listing 13-10 is some example code for generating a JNLP as well as embedding in HTML.

Listing 13-10. JNLP Code for Web Start

launch.jnlp

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<jnlp codebase="file:/path-to-jar/" href="launch.jnlp" spec="1.0+">
    <information>
        <title>JythonSwingApp</title>
        <vendor>YourName</vendor>
        <homepage href=""/>
        <description>JythonSwingApp</description>
        <description kind="short">JythonSwingApp</description>
    </information>
    <security>
        <all-permissions/>
    </security>
    <resources>
        <j2se version="1.5+"/>
        <jar eager="true" href="JythonSwingApp.jar" main="true"/>
        <jar href="lib/PlyJy.jar"/>
        <jar href="lib/jython.jar"/>
    </resources>
    <application-desc main-class="jythonswingapp.Main">
    </application-desc>
</jnlp>
```

launch.html

```
<html>
    <head>
```

```
        <title>Test page for launching the application via JNLP</title>
    </head>
    <body>
        <h3>Test page for launching the application via JNLP</h3>
        <a href="launch.jnlp">Launch the application</a>
        <!-- Or use the following script element to launch with the
        Deployment Toolkit -->
        <!-- Open the deployJava.js script to view its documentation -->
        <!--
        <script src="http://java.com/js/deployJava.js"></script>
        <script>
        var url="http://[fill in your URL]/launch.jnlp"
        deployJava.createWebStartLaunchButton(url, "1.6")
        </script>
        -->
    </body>
</html>
```

In the end, Java web start is a very good way to distribute Jython applications via the web.

Distributing via Standalone JAR

It is possible to distribute a web start application using the Jython standalone JAR option. To do so, you must have a copy of the Jython standalone JAR file, explode it, and add your code into the file, then JAR it back up to deploy. The only drawback to using this method is that you may need to ensure files are in the correct locations in order to make it work correctly, which can sometimes be tedious.

In order to distribute your Jython applications via a JAR, first download the Jython standalone distribution. Once you have this, you can extract the files from the *jython.jar* using a tool to expand the JAR such as Stuffit or 7zip. Once the JAR has been exploded, you will need to add any of your *.py* scripts into the *Lib*directory, and any Java classes into the root. For instance, if you have a Java class named *org.jythonbook.Book*, you would place it into the appropriate directory according to the package structure. If you have any additional JAR files to include with your application then you will need to make sure that they are in your classpath. Once you've completed this setup, JAR your manipulated standalone Jython JAR back up into a ZIP format using a tool such as those noted before. You can then rename the ZIP to a JAR. The application can now be run using the java "-jar" option from the command line using an optional external*.py* file to invoke your application.

```
$ java -jar newStandaloneJar.jar {optional .py file}
```

This is only one such technique used to make a JAR file for containing your applications. There are other ways to perform such techniques, but this seems to be the most straight forward and easiest to do.

WSGI and Modjy

WSGI, also known as the *Web Server Gateway Interface*, is a low-level API that provides communication between a web server and a web application. Actually, WSGI is a lot more than that and you can actually write complete web applications using WSGI. However, WSGI is more of a standard interface to call python methods and functions. Python PEP 333 specifies the proposed standard interface between web servers and Python web applications or frameworks, to promote web application portability across a variety of web servers.

This section will show you how to utilize WSGI to create a very simple "Hello Jython" application by utilizing *modjy*. Modjy is an implementation of a WSGI compliant gateway/server for Jython, built on Java/J2EE servlets. Taken from the modjy website (http://opensource.xhaus.com/projects/modjy/wiki), modjy is characterized as follows:

Note

Jython WSGI applications run inside a Java/J2EE container and incoming requests are handled by the servlet container. The container is configured to route requests to the modjy servlet. The modjy servlet then creates an embedded Jython interpreter inside the servlet container, and loads a configured Jython web application. For instance, a Django application can be loaded via modjy. The modjy servlet then delegates the requests to the configured WSGI application or framework. Lastly, the WSGI response is routed back to the client through the servlet container.

Running a Modjy Application in Glassfish

To run a modjy application in any Java servlet container, the first step is to create a Java web application that will be packaged up as a WAR file. You can create an application from scratch or use an IDE such as Netbeans 6.7 to assist. Once you've created your web application, ensure that *jython.jar* resides in the CLASSPATH as modjy is now part of Jython as of 2.5.0. Lastly, you will need to configure the modjy servlet within the application deployment descriptor (web.xml). In this example, we took the modjy sample application for Google App Engine and deployed it in my local Glassfish environment.

To configure the application deployment descriptor with modjy, we simply configure the modjy servlet, provide the necessary parameters, and then provide a servlet mapping. In the configuration file shown in Listing 13-11, note that the modjy servlet class is *com.xhaus.modjy.ModjyServlet*. The first parameter you will need to use with the servlet is named *python.home*. Set the value of this parameter equal to your Jython home. Next, set the parameter*python.cachedir.skip* equal to true. The *app_filename* parameter provides the name of the application callable. Other parameters will be set up the same for each modjy application you configure. The last piece of the web.xml that needs to be set up is the servlet mapping. In the example, we set up all URLs to map to the modjy servlet.

Listing 13-11. Configuring the Modjy Servlet

web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <display-name>modjy demo application</display-name>
    <description>
        modjy WSGI demo application
    </description>

    <servlet>
        <servlet-name>modjy</servlet-name>
        <servlet-class>com.xhaus.modjy.ModjyJServlet</servlet-class>

        <init-param>
            <param-name>python.home</param-name>
            <param-value>/Applications/jython/jython2.5.0/</param-value>
        </init-param>

        <init-param>
            <param-name>python.cachedir.skip</param-name>
            <param-value>true</param-value>
        </init-param>
        <!--
        There are two different ways you can specify an application to
        modjy
        1. Using the app_import_name mechanism
        2. Using a combination of
        app_directory/app_filename/app_callable_name
        Examples of both are given below
        See the documentation for more details.
        http://modjy.xhaus.com/locating.html#locating_callables
        -->
        <!--
        This is the app_import_name mechanism. If you specify a value
        for this variable, then it will take precedence over the other
        mechanism
        <init-param>
            <param-name>app_import_name</param-name>
            <param-
value>my_wsgi_module.my_handler_class().handler_method</param-value>
        </init-param>
        -->
        <!--
        And this is the app_directory/app_filename/app_callable_name
        combo
        The defaults for these three variables are
        ""/application.py/handler
        So if you specify no values at all for any of app_* variables,
        then modjy
        will by default look for "handler" in "application.py" in the
        servlet
        context root.
        <init-param>
```

```
            <param-name>app_directory</param-name>
            <param-value>some_sub_directory</param-value>
        </init-param>
        -->
        <init-param>
            <param-name>app_filename</param-name>
            <param-value>demo_app.py</param-value>
        </init-param>
        <!--
        Supply a value for this parameter if you want your application
        callable to have a different name than the default.
        <init-param>
            <param-name>app_callable_name</param-name>
            <param-value>my_handler_func</param-value>
        </init-param>
        -->
        <!-- Do you want application callables to be cached? -->
        <init-param>
            <param-name>cache_callables</param-name>
            <param-value>1</param-value>
        </init-param>
        <!-- Should the application be reloaded if it's .py file changes?
        -->
        <!-- Does not work with the app_import_name mechanism -->
        <init-param>
            <param-name>reload_on_mod</param-name>
            <param-value>1</param-value>
        </init-param>
        <init-param>
            <param-name>log_level</param-name>
            <param-value>debug</param-value>
            <!-- <param-value>info</param-value> -->
            <!-- <param-value>warn</param-value> -->
            <!-- <param-value>error</param-value> -->
            <!-- <param-value>fatal</param-value> -->
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>modjy</servlet-name>
    <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

The demo_app should be coded as shown in Listing 13-12. As part of the WSGI standard, the application provides a function that the server calls for each request. In this case, that function is named *handler*. The function must take two parameters, the first being a dictionary of CGI-defined environment variables. The second is a callback that returns the HTTP headers. The callback function should also be called as follows *start_response(status, response_headers, exx_info=None)*, where status is an HTTP status, response_headers is a list of HTTP headers, and exc_info is for exception handling. Let's take a look at the*demo_app.py* application and identify the features we've just discussed.

Listing 13-12.

```
import sys
import string

def escape_html(s):
    return s.replace('&', '&amp;').replace('<', '&lt;').replace('>',
'&gt;')

def cutoff(s, n=100):
    if len(s) > n: return s[:n]+ '.. cut ..'
    return s

def handler(environ, start_response):
    writer = start_response("200 OK", [ ('content-type', 'text/html') ])
    response_parts = '''<html><head>
        <title>Modjy demo WSGI application running on Local
Server!</title>
        </head>
        <body>
        <p>Modjy servlet running correctly:
        jython $version on $platform:
        </p>
        <h3>Hello jython WSGI on your local server!</h3>
        <h4>Here are the contents of the WSGI environment</h4>'''
    environ_str = "<table border='1'>"
    keys = environ.keys()
    keys.sort()
    for ix, name in enumerate(keys):
        if ix % 2:
            background='#ffffff'
        else:
            background='#eeeeee'
        style = " style='background-color:%s;'" % background
        value = escape_html(cutoff(str(environ[name]))) or ' '
        environ_str = "%s\\n<tr><td%s>%s</td><td%s>%s</td></tr>" % \\
            (environ_str, style, name, style, value)
    environ_str = "%s\\n</table>" % environ_str
    response_parts = response_parts + environ_str + '</body></html>\\n'
    response_text = string.Template(response_parts)
    return [response_text.substitute(version=sys.version,
platform=sys.platform)]
```

This application returns the environment configuration for the server on which you run the application. As you can see, the page is quite simple to code and really resembles a servlet.

Once the application has been set up and configured, simply compile the code into a WAR file and deploy it to the Java servlet container of your choice. In this case, we used Glassfish V2 and it worked nicely. However, this same application should be deployable to Tomcat, JBoss, or the like.

Summary

There are various ways that we can use Jython for creating simple web-based applications. Jython servlets are a good way to make content available on the web, and you can also utilize them along with a JSP page which allows for a Model-View-Controller situation. This is a good technique to use for developing sophisticated web applications, especially those mixing some JavaScript into the action because it really helps to organize things. Most Java web applications use frameworks or other techniques in order to help organize applications in such a way as to apply the MVC concept. It is great to have a way to do such work with Jython as well.

This chapter also discussed creation of WSGI applications in Jython making use of modjy. This is a good low-level way to generate web applications as well, although modjy and WSGI are usually used for implementing web frameworks and the like. Solutions such as Django use WSGI in order to follow the standard put forth for all Python web frameworks with PEP 333. You can see from the section in this chapter that WSGI is also a nice quick way to write web applications, much like writing a servlet in Jython.

Source: http://www.jython.org/jythonbook/en/1.0/SimpleWebApps.html