# Deployment Targets

Deployment of Jython applications varies from container to container. However, they are all very similar and usually allow deployment of WAR file or exploded directory web applications. Deploying to “the cloud” is a different scenario all together. Some cloud environments have typical Java application servers available for hosting, while others such as the Google App Engine run a bit differently. In this chapter, we’ll discuss how to deploy web-based Jython applications to a few of the more widely used Java application servers. We will also cover deployment of Jython web applications to the Google App Engine and mobile devices. Although many of the deployment scenarios are quite similar, this chapter will walk through some of the differences from container to container.

In the end, one of the most important things to remember is that we need to make Jython available to our application. There are different ways to do this: either by ensuring that the *jython.jar* file is included with the application server, or by packaging the JAR directly into each web application. This chapter assumes that you are using the latter technique. Placing the *jython.jar* directly into each web application is a good idea because it allows the web application to follow the Java paradigm of “deploy anywhere.” You do not need to worry whether you are deploying to Tomcat or Glassfish because the Jython runtime is embedded in your application.

Lastly, this section will briefly cover some of the reasons why mobile deployment is not yet a viable option for Jython. While a couple of targets exist in the mobile world, namely Android and JavaFX, both environments are still very new and Jython has not yet been optimized to run on either.

## Application Servers

As with any Java web application, the standard web archive (WAR) files are universal throughout the Java application servers available today. This is good because it makes things a bit easier when it comes to the “write once run everywhere” philosophy that has been brought forth with the Java name. The great part of using Jython for deployment to application servers is just that, we can harness the technologies of the JVM to make our lives easier and deploy a Jython web application to any application server in the WAR format with very little tweaking.

If you have not yet used Django or Pylons on Jython, then you may not be aware that the resulting application to be deployed is in the WAR format. This is great because it leaves no assumption as to how the application should be deployed. All WAR files are deployed in the same manner according to each application server. This section will

discuss how to deploy a WAR file on each of the three most widely used Java application servers. Now, all application servers are not covered in this section mainly due to the number of servers available today. Such a document would take more than one section of a book, no doubt. However, you should be able to follow similar deployment instructions as those discussed here for any of the application servers available today for deploying Jython web applications in the WAR file format.

## Tomcat

Arguably the most widely used of all Java application servers, Tomcat offers easy management and a small footprint compared to some of the other options available. Tomcat will plug into most IDEs that are in use today, so you can manage the web container from within your development environment. This makes it handy to deploy and undeploy applications on-the-fly. For the purposes of this section, we've used Netbeans 6.7, so there may be some references to it.

To get started, download the Apache Tomcat server from the site at http://tomcat.apache.org/. Tomcat is constantly evolving, so we'll note that when writing this book the deployment procedures were targeted for the 6.0.20 release. Once you have downloaded the server and placed it into a location on your hard drive, you may have to change permissions. We had to use the *chmod +x* command on the entire apache-tomcat-6.0.20 directory before we were able to run the server. You will also need to configure an administrative account by going into the */conf/tomcat-users.xml* file and adding one. Be sure to grant the administrative account the "manager" role. This should look something like the following once completed.

Listing 17-1. tomcat-users.xml

```
<tomcat-users>
   <user username="admin" password="myadminpassword" roles="manager"/>
</tomcat-users>
```

After this has been done, you can add the installation to an IDE environment of your choice if you'd like. For instance, if you wish to add to Netbeans 6.7 you will need to go to the "Services" tab in the navigator, right-click on servers, choose "Tomcat 6.x" option, and then fill in the appropriate information pertaining to your environment. Once complete, you will be able to start, stop, and manage the Tomcat installation from the IDE.

## Deploying Web Start

Deploying a web start application is as easy as copying the necessary files to a location on the web server that is accessible via the web. In the case of Tomcat, you will need to copy the contents of your web start application to a single directory contained within the "<tomcat-root>/webapps/ROOT" directory. For instance, if you have a web-start application entitled , then you would package the JAR file along with the JNLP and HTML file for the application into a directory entitled and then place that directory into the "<tomcat-root>/webapps/ROOT" directory.

Once the application has been copied to the appropriate locations, you should be able to access it via the web if Tomcat is started. The URL should look something like the following: *http://your-server:8080/JythonWebStart/launch.jnlp*. Of course, you will need to use the server name and the port that you are using along with the appropriate JNLP name for your application.

## Deploying a WAR or Exploded Directory Application

To deploy a web application to Tomcat, you have two options. You can either use a WAR file including all content for your entire web application, or you can deploy an exploded directory application which is basically copy-and-paste for your entire web application directory structure into the "<tomcat-root>/webapps/ROOT" directory. Either way will work the same, and we will discuss each technique in this section.

For manual deployment of a web application, you can copy either your exploded directory web application or your WAR file into the "<tomcat-root>/webapps" directory. By default, Tomcat is setup to "autodeploy" applications. This means that you can have Tomcat started when you copy your WAR or exploded directory into the "webapps" location. Once you've done this, you should see some feedback from the Tomcat server if you have a terminal open (or from within the IDE). After a few seconds the application should be deployed successfully and available via the URL. The bonus to deploying exploded directory applications is that you can take any file within the application and change it at will. Once you are done with the changes, that file will be redeployed when you save it. . .this really saves on development time!

If you do not wish to have autodeploy enabled (perhaps in a production environment), then you can deploy applications on startup of the server. This process is basically the same as "autodeploy," except any new applications that are copied into the "webapps" directory are not deployed until the server is restarted. Lastly, you can always make use of the Tomcat manager to deploy web applications as well. To do this, open your web browser to the index of Tomcat, usually http://localhost:8080/index.html, and then click on the "Manager" link in the left-hand menu. You will need to authenticate at that point using your administrator password, but once you are in the console deployment is quite easy. In an effort to avoid redundancy, we will once again redirect you to the Tomcat

documentation for more information on deploying a web application via the Tomcat manager console.

## Glassfish

At the time of writing, the Glassfish V2 application server was mainstream and widely used. The Glassfish V3 server was still in preview mode, but showed a lot of potential for Jython application deployment. In this section, we will cover WAR and web start deployment to Glassfish V2, because it is the most widely used version. We will also discuss deployment for Django on Glassfish V3, because this version has added support for Django (and more Python web frameworks soon). Glassfish is very similar to Tomcat in terms of deployment, but there are a couple of minor differences which will be covered in this section.

To start out, you will need to download a glassfish distribution from the site at https://glassfish.dev.java.net/. Again, we recommend downloading V2, because it is the most widely used at the time of this writing. Installation is quite easy, but a little more involved than that of Tomcat. The installation of Glassfish will not be covered in this text, because it varies depending upon which version you are using. There are detailed instructions for each version located on the Glassfish website, so we will redirect you there for more information.

Once you have Glassfish installed, you can utilize the server via the command-line or terminal, or you can use an IDE just like Tomcat. To register a Glassfish V2 or V3 installation with Netbeans 6.7, just go to the “Services” tab in the Netbeans navigator and right-click on “Servers” and then add the version you are planning to register. Once the “Add Server Instance” window appears, simply fill in the information depending upon your environment.

There is an administrative user named “admin” that is set up by default with a Glassfish installation. In order to change the default password, it is best to startup Glassfish and log into the administrative console. The default administrative console port is 4848.

## Deploying Web Start

Deploying a web start application is basically the same as any other web server, you simply make the web start JAR, JNLP, and HTML file accessible via the web. On Glassfish, you need to traverse into your “domain” directory and you will find a “docroot” inside. The path should be similar to “<glassfish-install-loc>/domains/domain1/docroot”. Anything placed within the docroot area is visible to the web, so of course this is where you will place any web-start application directories. Again, a typical web start application will consist of your application JAR file, a JNLP file, and an HTML page used to open

the JNLP. All of these files should typically be placed inside a directory appropriately named per your application, and then you can copy this directory into docroot.

## WAR File and Exploded Directory Deployment

Again, there are a variety of ways to deploy an application using Glassfish. Let's assume that you are using V2, you have the option to "hot deploy" or use the Glassfish Admin Console to deploy your application. Glassfish will work with either an exploded directory or WAR file deployment scenario. By default, the Glassfish "autodeploy" option is turned on, so it is quite easy to either copy your WAR or exploded directory application into the autodeploy location to deploy. If the application server is started, it will automatically start your application (if it runs without issues). The autodeploy directory for Glassfish V2 resides in the location "<glassfish-install-loc>/domains/domain1/autodeploy."

## Glassfish v3 Django Deployment

The Glassfish V3 server has some capabilities built into it to help facilitate the process of deploying a Django application. In the future, there will also be support for other Jython web frameworks such as Pylons.

## Other Java Application Servers

If you have read through the information contained in the previous sections, then you have a fairly good idea of what it is like to deploy a Jython web application to a Java application server. There is no difference between deploying Jython web applications and Java web applications for the most part. You must be sure that you include as mentioned in the introduction, but for the most part deployment is the same. However, we have run into cases with some application servers such as JBoss where it wasn't so cut-and-dry to run a Jython application. For instance, we have tried to deploy a Jython servlet application on JBoss application server 5.1.0 GA and had lots of issues. For one, we had to manually add to the application because we were unable to compile the application in Netbeans without doing so...this was not the case with Tomcat or Glassfish. Similarly, we had issues trying to deploy a Jython web application to JBoss as there were several errors that had incurred when the container was scanning for some reason.

All in all, with a bit of tweaking and perhaps an additional XML configuration file in the application, Jython web applications will deploy to *most* Java application servers. The bonus to deploying your application on a Java application server is that you are in complete control of the environment. For instance, you could embed the *jython.jar* file into the application server lib directory so that it was loaded at startup and available for

all applications running in the environment. Likewise, you are in control of other necessary components such as database connection pools and so forth. If you deploy to another service that lives in “the cloud,” you have very little control over the environment. In the next section, we’ll study one such environment by Google which is known as the Google App Engine. While this “cloud” service is an entirely different environment than your basic Java web application server, it contains some nice features that allow one to test applications prior to deployment in the cloud.

## Google App Engine

The new kid on the block, at least for the time of this writing, is the Google App Engine. Fresh to the likes of the Java platform, the Google App Engine can be used for deploying applications written in just about any language that runs on the JVM, Jython included. The App Engine went live in April of 2008, allowing Python developers to begin using its services to host Python applications and libraries. In the spring of 2009, the App Engine added support for the Java platform. Along with support of the Java language, most other languages that run on the JVM will also deploy and run on the Google App Engine, including Jython. It has been mentioned that more programming languages will be supported at some point in the future, but at the time of this writing Python and Java were the only supported languages.

The App Engine actually runs a slightly slimmed-down version of the standard Java library. You must download and develop using the Google App Engine SDK for Java in order to ensure that your application will run in the environment. You can download the SDK by visiting this link:http://code.google.com/appengine/downloads.html along with viewing the extensive documentation available on the Google App Engine site. The SDK comes complete with a development web server that can be used for testing your code before deploying, and several demo applications ranging from easy JSP programs to sophisticated demos that use Google authentication. No doubt about it, Google has done a good job at creating an easy learning environment for the App Engine so that developers can get up and running quickly.

In this section you will learn how to get started using the Google App Engine SDK, and how to deploy some Jython web applications. You will learn how to deploy a Jython servlet application as well as a WSGI application utilizing modjy. Once you’ve learned how to develop and use a Jython Google App Engine program using the development environment, you will learn a few specifics about deploying to the cloud. If you have not done so already, be sure to visit the link mentioned in the previous paragraph and download the SDK so that you can follow along in the sections to come.

n **Note** The Google App Engine is a very large topic. Entire books could be written on the subject of developing Jython applications to run on the App Engine. With that said,

we will cover the basics to get you up and running with developing Jython applications for the App Engine. Once you've read through this section, we suggest going to the Google App Engine documentation for further details.

## Starting With an SDK Demo

We will start by running the demo application known as "guestbook" that comes with the Google App Engine SDK. This is a very simple Java application that allows one to sign in using an email address and post messages to the screen. In order to start the SDK web server and run the "guestbook" application, open up a terminal and traverse into the directory where you expanded the Google App Engine .zip file and run the following command:

```
<app-engine-base-directory>/bin/dev_appserver.sh demos/guestbook/war
```

Of course, if you are running on windows there is a corresponding .bat script for you to run that will start the web server. Once you've issued the preceding command it will only take a second or two before the web server starts. You can then open a browser and traverse to *http://localhost:8080* to invoke the "guestbook" application. This is a basic JSP-based Java web application, but we can deploy a Jython application and use it in the same manner as we will see in a few moments. You can stop the web server by pressing "CTRL+C".

## Deploying to the Cloud

Prior to deploying your application to the cloud, you must of course set up an account with the Google App Engine. If you have another account with Google such as GMail, then you can easily activate your App Engine account using that same username. To do so, go to the Google App Engine link:http://code.google.com/appengine/ and click "Sign Up." Enter your existing account information or create a new account to get started.

After your account has been activated you will need to create an application by clicking on the "Create Application" button. You have a total of 10 available application slots to use if you are making use of the free App Engine account. Once you've created an application then you are ready to begin deploying to the cloud. In this section of the book, we create an application known as *jythongae*. This is the name of the application that you must create on the App Engine. You must also ensure that this name is supplied within the *appengine-web.xml* file.

## Working With a Project

The Google App Engine provides project templates to get you started developing using the correct directory structure. Eclipse has a plug-in that makes it easy to generate Google App Engine projects and deploy them to the App Engine. If interested in making use of the plug-in, please visithttp://code.google.com/appengine/docs/java/tools/eclipse.html to read more information and download the plug-in. Similarly, Netbeans has an App Engine plug-in that is available on the Kenai site appropriately named (http://kenai.com/projects/nbappengine). In this text we will cover the use of Netbeans 6.7 to develop a simple Jython servlet application to deploy on the App Engine. You can either download and use the template available with one of these IDE plug-ins, or simply create a new Netbeans project and make use of the template provided with the App Engine SDK (<app-engine-base-directory/demos/new_project_template>) to create your project directory structure.

For the purposes of this tutorial, we will make use of the *nbappengine* plug-in. If you are using Eclipse you will find a section following this tutorial that provides some Eclipse plug-in specifics.

In order to install the nbappengine plug-in, you add the 'App Engine' update center to the Netbeans plug-in center by choosing the Settings tab and adding the update center using http://deadlock.netbeans.org/hudson/job/nbappengine/lastSuccessfulBuild/artifact/build/updates/updates.xml.gz as the URL. Once you've added the new update center you can select the Available Plugins tab and add all of the plug-ins in the "Google App Engine" category, then choose Install. After doing so, you can add the "App Engine" as a server in your Netbeans environment using the "Services" tab. To add the server, point to the base directory of your Google App Engine SDK. Once you have added the App Engine server to Netbeans, it will become an available deployment option for your web applications.

Create a new Java web project and name it JythonGAE. For the deployment server, choose "Google App Engine," and you will notice that when your web application is created an additional file will be created within the WEB-INF directory named appengine-web.xml. This is the Google App Engine configuration file for the JythonGAE application. Any of the .py files that we wish to use in our application must be mapped in this file so that they will not be treated as static files by the Google App Engine. By default, Google App Engine treats all files outside of the WEB-INF directory as static unless they are JSP files. Our application is going to make use of three Jython servlets, namely NewJythonServlet.py, AddNumbers.py and AddToPage.py. In our appengine-web.xml file we can exclude all .py files from being treated as static by adding the suffix to the exclusion list as follows.

Listing 17-2. appengine-web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<appengine-web-app xmlns="http://appengine.google.com/ns/1.0">
    <application>jythongae</application>
    <version>1</version>
    <static-files>
        <exclude path="/**.py"/>
    </static-files>
    <resource-files/>
    <ssl-enabled>false</ssl-enabled>
    <sessions-enabled>true</sessions-enabled>
</appengine-web-app>
```

At this point we will need to create a couple of additional directories within our WEB-INF project directory. We should create a *lib* directory and place*jython.jar* and *appengine-api-1.0-sdk-1.2.2.jar* into the directory. Note that the App Engine JAR may be named differently according to the version that you are using. We should now have a directory structure that resembles the following:

Listing 17-3.

```
JythonGAE
    WEB-INF
        lib
            jython.jar
            appengine-api-1.0-sdk-1.2.2.jar
        appengine-web.xml
        web.xml
    src
    web
```

Now that we have the application structure set up, it is time to begin building the actual logic. In a traditional Jython servlet application we need to ensure that the *PyServlet* class is initialized at startup and that all files ending in *.py* are passed to it. As we've seen in Chapter 13, this is done in the *web.xml*deployment descriptor. However, we have found that this alone does not work when deploying to the cloud. We found some inconsistencies while deploying against the Google App Engine development server and deploying to the cloud. For this reason, we will show you the way that we were able to get the application to function as expected in both the production and development Google App Engine environments. In Chapter 12, the object factory pattern for coercing Jython classes into Java was discussed. If this same pattern is applied to Jython servlet applications, then we can use the factories to coerce our Jython servlet into Java byte code at runtime. We then map the resulting coerced class to a servlet mapping in the application's web.xml deployment descriptor. We can

also deploy our Jython applets and make use of *PyServlet* mapping to the *.py* extension in the *web.xml*. We will comment in the source where the code for the two implementations differs.

## Object Factories with App Engine

In order to use object factories to coerce our code, we must use an object factory along with a Java interface, and once again we will use the PlyJy project to make this happen. Please note that if you choose to not use the object factory pattern and instead use PyServlet you can safely skip forward to the next subsection. The first step is to add to the directory that we created previously to ensure it is bundled with our application. There is a Java servlet contained within the PlyJy project named , and what this Java servlet does is essentially use the class to coerce a named Jython servlet and then invoke its resulting and methods. There is also a simple Java interface named in the project, and it must be implemented by our Jython servlet in order for the coercion to work as expected.

## Using PyServlet Mapping

When we use the PyServlet mapping implementation, there is no need to coerce objects using factories. You simply set up a servlet mapping within and use your Jython servlets directly with the .py extension in the URL. However, we've seen issues while using PyServlet on the App Engine in that this implementation will deploy to the development App Engine server environment, but when deployed to the cloud you will receive an error when trying to invoke the servlet. It is because of these inconsistencies that we chose to implement the object factory solution for Jython servlet to App Engine deployment.

## Example Jython Servlet Application for App Engine

The next piece of the puzzle is the code for our application. In this example, we'll make use of a simple servlet that displays some text as well as the same example that was used in Chapter 13 with JSP and Jython. The following code sets up three Jython servlets. The first servlet simply displays some output, the next two perform some mathematical logic, and then there is a JSP to display the results for the mathematical servlets.

Listing 17-4. NewJythonServlet.py

```
from javax.servlet.http import HttpServlet
from org.plyjy.interfaces import JythonServletInterface

```

```
class NewJythonServlet (JythonServletInterface, 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
Using Object Factory</title>" +
                                                    "<body><h1>Jython Servlet
Test for GAE</h1></body></html>")

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

Listing 17-5. AddNumbers.py

```
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)
```

Listing 17-6. AddToPage.py

```
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)
```

Listing 17-7. testjython.jsp

Note that this implementation differs if you plan to make use of the object factory technique. Instead of using and as your actions, you would utilize the servlet instead, namely and

```
<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>
        <% Object page_text = request.getAttribute("page_text");
           Object sum = request.getAttribute("sum");
           if(page_text == null){
               page_text = "";
           }
           if(sum == null){
               sum = "";
           }
        %>
        <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>
```

As mentioned previously, it is important that all of the Jython servlets reside within your classpath somewhere. If using Netbeans, you can either place the servlets into the source root of your project (not inside a package), or you can place them in the web folder that contains your JSP files. If doing the latter, we have found that you may have to tweak your CLASSPATH a bit by adding the web folder to your list of libraries from

within the project properties. Next, we need to ensure that the deployment descriptor includes the necessary servlet definitions and mappings for the application. Now, if you are using the object factory implementation and the *JythonServletFacade* servlet, you would have noticed that there is a variable named *PyServletName* which the JythonObjectFactory is using as the name of our Jython servlet. Well, within the *web.xml* we must pass an *<init-param>* using *PyServletName* as the*<param-name>* and the name of our Jython servlet as the *<param-value>*. This will basically pass the name of the Jython servlet to the *JythonServletFacade*servlet so that it can be used by the object factory.

Listing 17-8. web.xml

```
<web-app>
    <display-name>Jython Google App Engine</display-name>

    <!-- Used for the PyServlet Implementation -->
    <servlet>
        <servlet-name>PyServlet</servlet-name>
        <servlet-class>org.python.util.PyServlet</servlet-class>
    </servlet>

    <!-- The next three servlets are used for the object factory
implementation only.
        They can be excluded in the PyServlet implementation -->
    <servlet>
        <servlet-name>NewJythonServlet</servlet-name>
        <servlet-class>org.plyjy.servlets.JythonServletFacade</servlet-class>
        <init-param>
            <param-name>PyServletName</param-name>
            <param-value>NewJythonServlet</param-value>
        </init-param>
    </servlet>
    <servlet>
        <servlet-name>AddNumbers</servlet-name>
        <servlet-class>org.plyjy.servlets.JythonServletFacade</servlet-class>
        <init-param>
            <param-name>PyServletName</param-name>
            <param-value>AddNumbers</param-value>
        </init-param>
    </servlet>
    <servlet>
        <servlet-name>AddToPage</servlet-name>
        <servlet-class>org.plyjy.servlets.JythonServletFacade</servlet-class>
        <init-param>
            <param-name>PyServletName</param-name>
            <param-value>AddToPage</param-value>
        </init-param>
    </servlet>

    <!-- The following mapping should be used for the PyServlet
implementation -->
```

```
    <servlet-mapping>
        <servlet-name>PyServlet</servlet-name>
        <url-pattern>*.py</url-pattern>
    </servlet-mapping>

    <!-- The following three mappings are used in the object factory
implementation -->

    <servlet-mapping>
        <servlet-name>NewJythonServlet</servlet-name>
        <url-pattern>/NewJythonServlet</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>AddNumbers</servlet-name>
        <url-pattern>/AddNumbers</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>AddToPage</servlet-name>
        <url-pattern>/AddToPage</url-pattern>
    </servlet-mapping>
</web-app>
```

Note that when using the PyServlet implementation you should exclude those portions in the *web.xml* above that are used for the object factory implementation. The PyServlet mapping can be contained within the *web.xml* in both implementations without issue. That's it, now you can deploy the application to your Google App Engine development environment, and it should run without any issues. You can also choose to deploy to another web server to test for compatibility if you wish. You can deploy directly to the cloud by right-clicking the application and choosing the "Deploy to App Engine" option.

## Using Eclipse

If you wish to use the Eclipse IDE for development, you should definitely download the Google App Engine plug-in using the link provided earlier in the chapter. You should also use the PyDev plug-in which is available at http://pydev.sourceforge.net/. For the purposes of this section, we used Eclipse Galileo and started a new project named "JythonGAE" as a Google Web Application. When creating the project, make sure you check the box for using Google App Engine and uncheck the Google Web Toolkit option. You will find that Eclipse creates a directory structure for your application that is much the same as the project template that is included with the Google App Engine SDK.

If you follow through the code example from the previous section, you can create the same code and set up the *web.xml* and *appengine-web.xml* the same way. The key is to ensure that you create a *lib* directory within the *WEB-INF* and you place the files in the appropriate location. You will need to ensure that your Jython servlets are contained

in your CLASSPATH by either adding them to the source root for your project, or by going into the project properties and adding the *war* directory to your *Java Build Path*. When doing so, make sure you do *not* include the *WEB-INF* directory or you will receive errors.

When you are ready to deploy the application, you can choose to use the Google App Engine development environment or deploy to the cloud. You can run the application by right-clicking on the project and choosing *Run As* option and then choose the Google Web Application option. The first time you run the application you may need to set up the runtime. If you are ready to deploy to the cloud, you can right-click on the project and choose the *Google -> Deploy to App Engine* option. After entering your Google username and password then your application will be deployed.

## Deploy Modjy to GAE

We can easily deploy WSGI applications using Jython's modjy API as well. To do so, you need to add an archive of the Jython directory to your WEB-INF project directory. According to the modjy web site, you need to obtain the source for Jython, then zip the directory and place it into another directory along with a file that will act as a pointer to the zip archive. The modjy site names the directory and names the pointer file . This pointer file can be named anything as long as the suffix is . Inside the pointer file you need to explicitly name the zip archive that you had created for the directory contents. Let's assume you named it lib.zip, in this case we will put the text "lib.zip" without the quotes into the file. Now if we add the modjy demonstration application to the project then our directory structure should look as follows:

Listing 17-9.

```
modjy_app
    demo_app.py
    WEB-INF
        lib
            jython.jar
            appengine-api-1.0-sdk-1.2.2.jar
        python-lib
            lib.zip
            all.pth
```

Now if we run the application using Tomcat it should run as expected. Likewise, we can run it using the Google App Engine SDK web server and it should provide the expected results.

The Google App Engine is certainly an important deployment target for Jython. Google offers free hosting for smaller applications, and they also base account pricing on bandwidth. No doubt that it is a good way to put up a small site, and possibly build on it later. Most importantly, you can deploy Django, Pylons, and other applications via Jython to the App Engine by setting up your App Engine applications like the examples we had shown in this chapter.

## Java Store

Another deployment target that is hot off the presses at the time of this book is the Java Store or Java Warehouse. This is a new concept brought to market by Sun Microsystems in order to help Java software developers market their applications via a single shop that is available online via a web start application. Similar to other application venues, The Java Store is a storefront application where people can go to search for applications that have been submitted by developers. The Java Warehouse is the repository of applications that are contained within the Java Store. This looks to be a very promising target for Java and Jython developers alike. It be as easy as generating a JAR file that contains a Jython application and deploying it to the Java Store. Unfortunately, because the program is still in alpha mode at this time, we are unable to provide any specific details on distributing Jython applications via the Java Store. However, there are future plans to make alternative VM language applications easily deployable to the Java Warehouse. At this time, it is certainly possible to deploy a Jython application to the warehouse, but it can only deploy as a Java application. As of the time of this writing, only Java and JavaFX applications are directly deployable to the Java Warehouse. Please note that because this product is still in alpha mode, this book will not discuss such aspects of the program as memberships or fees that may be incurred for hosing your applications on the Java Store.

The requirements for publishing an application to the warehouse are as follows:

- Your application packed in a single jar file
- Descriptive text to document your application
- Graphic image files used for icons and to give the consumer an idea of your application's look.

In Chapter 13, we took a look at packaging and distributing Jython GUI applications in a JAR file. When a Jython application is packaged in a JAR file then it is certainly possible to use Java Web Start to host the application via the web. On the other hand, if one wishes to make a Jython GUI application available for purchase or for free, the Java Store would be another way of doing so. One likely way to deploy applications in a single JAR is to use the method discussed in Chapter 13, but there are other solutions as well. For instance, one could use the *One-Jar* product to create a single JAR file

containing all of the necessary Jython code as well as other JAR files essential to the application. In the following section, we will discuss deployment of a Jython application using One-JAR so that you can see some similarities and differences to using the Jython standalone JAR technique.

## Deploying a Single JAR

In order to deploy an application to the Java Warehouse, it must be packaged as a single JAR file. We've already discussed packaging Jython applications into a JAR file using the Jython standalone method in Chapter 13. In this section, you will learn how to make use of the One-JAR (http://one-jar.sourceforge.net/) product to distribute client-based Jython applications. In order to get started, you will need to grab a copy of One-JAR. There are a few options available on the download site, but for our purposes we will package an application using the source files for One-JAR. Once downloaded, the source for the project should look as follows.

Listing 17-10.

```
src
    com
        simontuffs
            onejar
                Boot.java
                Handler.java
                IProperties.java
                JarClassLoader.java
```

This source code for the One-Jar project must reside within the JAR file that we will build. Next, we need to create separate source directories for both our Jython source and our Java source. Likewise, we will create a separate source directory for the One-Jar source. Lastly, we'll create a *lib* directory into which we will place all of the required JAR files for the application. In order to run a Jython application, we'll need to package the Jython project source into a JAR file for our application. We will not need to use the entire *jython.jar*, but rather only a standalone version of it. The easiest way to obtain a standalone Jython JAR is to run the installer and choose the standalone option. After this is done, simply add the resulting jython.jar to the lib directory of application. In the end, the directory structure should resemble the following.

Listing 17-11.

```
one-jar-jython-example
    java
    lib
        jython.jar
```

```
    LICENSE.txt
    onejar
        src
            com

                simontuffs
                    onejar
                        Boot.java
                        Handler.java
                        IProperties.java
                        JarClassLoader.java
            one-jar-license.txt
    src
```

As you can see from the depiction of the file structure in this example, the *src* directory will contain our Jython source files. The LICENSE.txt included in this example was written by Ryan McGuire (http://www.enigmacurry.com). He has a detailed explanation of using One-Jar on his blog, and we've replicated some of his work in this example. . .including a version of the build.xml that we will put together in order to build the application. Let's take a look at the build file that we will use to build the application JAR. In this example we are using Apache Ant for the build system, but you could choose something different if you'd like.

Listing 17-12. build.xml

```
<project name="JythonSwingApp" default="dist" basedir=".">

  <!-- ################################################
       These two properties are the only ones you are
       likely to want to change for your own projects:     -->
  <property name="jar.name" value="JythonSwingApp.jar" />
  <property name="java-main-class" value="Main" />
  <!-- ################################################  -->

  <!-- Below here you don't need to change for simple projects -->
  <property name="src.dir" location="src"/>
  <property name="java.dir" location="java"/>
  <property name="onejar.dir" location="onejar"/>
  <property name="java-build.dir" location="java-build"/>
  <property name="build.dir" location="build"/>
  <property name="lib.dir" location="lib"/>

  <path id="classpath">
    <fileset dir="${lib.dir}" includes="**/*.jar"/>
  </path>

  <target name="clean">
    <delete dir="${java-build.dir}"/>
    <delete dir="${build.dir}"/>
    <delete file="${jar.name}"/>
```

```
  </target>

  <target name="dist" depends="clean">
    <!-- prepare the build directory -->
    <mkdir dir="${build.dir}/lib"/>
    <mkdir dir="${build.dir}/main"/>
    <!-- Build java code -->
    <mkdir dir="${java-build.dir}"/>
    <javac srcdir="${java.dir}" destdir="${java-build.dir}"
classpathref="classpath"/>
    <!-- Build main.jar -->
    <jar destfile="${build.dir}/main/main.jar" basedir="${java-build.dir}">
      <manifest>
            <attribute name="Main-Class" value="Main" />
      </manifest>
    </jar>
    <delete file="${java-build.dir}"/>
    <!-- Add the python source -->
    <copy todir="${build.dir}">
      <fileset dir="${src.dir}"/>
    </copy>
    <!-- Add the libs -->
    <copy todir="${build.dir}/lib">
      <fileset dir="${lib.dir}"/>
    </copy>
    <!-- Compile OneJar -->
    <javac srcdir="${onejar.dir}" destdir="${build.dir}"
classpathref="classpath"/>
    <!-- Copy the OneJar license file -->
    <copy file="${onejar.dir}/one-jar-license.txt" tofile="${build.dir}/one-
jar-license.txt" />
    <!-- Build the jar -->
    <jar destfile="${jar.name}" basedir="${build.dir}">
      <manifest>
            <attribute name="Main-Class" value="com.simontuffs.onejar.Boot"
/>
            <attribute name="Class-Path" value="lib/jython-full.jar" />
      </manifest>
    </jar>
    <!-- clean up -->
    <delete dir="${java-build.dir}" />
    <delete dir="${build.dir}" />
  </target>

</project>
```

Because this is a Jython application, we can use as much Java source as we'd like. In this example, we will only use one Java source file *Main.java* to "drive" our application. In this case, we'll use the *PythonInterpreter* inside of our *Main.java* to invoke our simple Jython Swing application. Now let's take a look at the*Main.java* source.

Listing 17-13. Main.java

```
import org.python.core.PyException;
import org.python.util.PythonInterpreter;

public class Main {
    public static void main(String[] args) throws PyException{
        PythonInterpreter intrp = new PythonInterpreter();
        intrp.exec("import JythonSimpleSwing as jy");
        intrp.exec("jy.JythonSimpleSwing().start()");
    }
}
```

Now that we've written the driver class, we'll place it into our *java* source directory. As stated previously, we'll place our Jython code into the *src* directory. In this example we are using the same simple Jython Swing application that we wrote for Chapter 13.

Listing 17-14. JythonSimpleSwing.py

```
import sys

import javax.swing as swing
import java.awt as awt

class JythonSimpleSwing(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 = ""
```

At this time, the application is ready to build using Ant. In order to run the build, simply traverse into the directory that contains *build.xml* and initiate the *ant*command. The resulting JAR can be run using the following syntax:

java -jar JythonSwingApp.jar

In some situations, such as deploying via web start, this JAR file will also need to be signed. There are many resources online that explain the signing of JAR files that topic will not be covered in this text. The JAR is now ready to be deployed and used on other machines. This method will be a good way to package an application for distribution via the Java Store.

## Mobile

Mobile applications are the way of the future. At this time, there are a couple of different options for developing mobile applications using Jython. One way to develop mobile applications using Jython is to make use of the JavaFX API from Jython. Since JavaFX is all Java behind the scenes, it would be fairly simple to make use of the JavaFX API using Jython code. However, this technique is not really a production-quality result in my opinion for a couple of reasons. First, the JavaFX scripting language makes GUI development quite easy. While it is possible (seehttp://wiki.python.org/jython/JythonMonthly/Articles/December2007/2 for more details), the translation of JavaFX API using Jython would not be as easy as making use of the JavaFX script language. The second reason this is not feasible at the time of this writing is that JavaFX is simply not available on all mobile devices at this time. It is really just becoming available to the mobile world at this time and will take some time to become acclimated.

Another way to develop mobile applications using Jython is to make use of the Android operating system which is available via Google. Android is actively being used on mobile devices today, and its use is continuing to grow. Although in early stages, there is a project known as *Jythonroid* that is an implementation of Jython for the Android

Dalvik Virtual Machine. Unfortunately, it was not under active development at the time of this writing, although some potential does exist for getting the project on track.

If you are interested in mobile development using Jython, please pay close attention to the two technologies discussed in this section. They are the primary deployment targets for Jython in the mobile world. As for the *Jythonroid* project, it is open source and available to developers. Interested parties may begin working on it again to make it functional and bring it up to date with the latest Android SDK.

## Summary

Deploying Jython applications is very much like Java application deployment. For those of you who are familiar with Java application servers, deploying a Jython application should be a piece of cake. On the contrary, for those of you who are not familiar with Java application deployment this topic may take a bit of getting used to. In the end, it is easy to deploy a Jython web or client application using just about any of the available Java application servers that are available today.

Deploying Jython web applications is universally easy to do using the WAR file format. As long as *jython.jar* is either in the application server classpath or packaged along with the web application, Jython servlets should function without issue. We also learned that it is possible to deploy a JAR file containing a Jython GUI application via Java web start technology. Using a JNLP deployment file is quite easy to do; the trick to deploying Jython via a JAR file is to set the file up correctly. Once completed, an HTML page can be used to reference the JNLP and initiate the download of the JAR to the client machine.

Lastly, this section discussed use of the Google App Engine for deploying Jython servlet applications. While the Google App Engine environment is still relatively new at the time of this writing, it is an exceptional deployment target for any Python or Java application developer. Using a few tricks with the object factory technique, it is possible to deploy Jython servlets and use them directly or via a JSP file on the App Engine. Stay tuned for more deployment targets to become available for Jython in the coming months and years. As cloud computing and mobile devices are becoming more popular, the number of deployment targets will continue to grow and Jython will be more useful with each one.

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