Chapter 4: Using and Deploying Web Applications

Update from Marty: For examples from this chapter that have been updated to the latest specification, please see http://courses.coreservlets.com/Course-Materials/msajsp.html.

Topics in This Chapter

- Registering Web applications with the server
- Organizing Web applications
- Deploying applications in WAR files
- Recording Web application dependencies on shared libraries
- Dealing with relative URLs
- Sharing data among Web applications

Chapter 4

Web applications (or "Web apps") let you bundle a set of servlets, JSP pages, tag libraries, HTML documents, images, style sheets, and other Web content into a single collection that can be used on any server compatible with servlet version 2.2 or later (JSP 1.1 or later). When designed carefully, Web apps can be moved from server to server or placed at different locations on the same server, all without making any changes to any of the servlets, JSP pages, or HTML files in the application.

This capability lets you move complex applications around with a minimum of effort, streamlining application reuse. In addition, since each Web app has its own directory structure, sessions, `ServletContext`, and class loader, using a Web app simplifies even the initial development because it reduces the amount of coordination needed among various parts of your overall system.

4.1 Registering Web Applications

With servlets 2.2 and later (JSP 1.1 and later), Web applications are portable. Regardless of the server, you store files in the same directory structure and access them with URLs in identical formats. For example, Figure 4–1 summarizes the directory structure and URLs that would be used for a simple Web application called `webapp1`. This section will illustrate how to install and execute this simple Web application on different platforms.

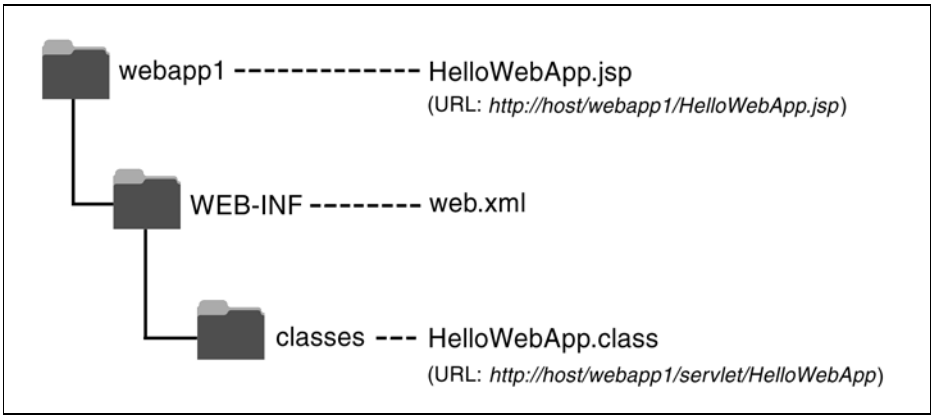


Figure 4–1 Structure of the `webapp1` Web application.

Although Web applications themselves are completely portable, the registration process is server specific. For example, to move the `webapp1` application from server to server, you don't have to modify anything *inside* any of the directories shown in Figure 4–1. However, the location *in which* the top-level directory (*webapp1* in this case) is placed will vary from server to server. Similarly, you use a server-specific process to tell the system that URLs that begin with *http://host/webapp1/* should apply to the Web application. In general, you will need to read your server's documentation to get details on the registration process. I'll present a few brief examples here, then give explicit details for Tomcat, JRun, and ServletExec in the following subsections.

My usual strategy is to build Web applications in my personal development environment and periodically copy them to various deployment directories for testing on different servers. I never place my development directory directly within a server's deployment directory—doing so makes it hard to deploy on multiple servers, hard to develop while a Web application is executing, and hard to organize the files. I recommend you avoid this approach as well; instead, use a separate development directory and deploy by means of one of the strategies outlined in Section 1.8 (Establish a Simplified Deployment Method). The simplest approach is to keep a shortcut (Windows) or symbolic link (Unix/Linux) to the deployment directories of various servers and simply copy the entire development directory whenever you want to deploy. For example, on Windows you can use the right mouse button to drag the development folder onto the shortcut, release the button, and select Copy.

To illustrate the registration process, the iPlanet Server 6.0 provides you with two choices for creating Web applications. First, you can edit iPlanet's *web-apps.xml* file (not *web.xml*!) and insert a `web-app` element with attributes `dir` (the directory containing the Web app files) and `uri` (the URL prefix that designates the Web application). Second, you can create a Web Archive (WAR) file and then use the `wdeploy`

command-line program to deploy it. WAR files are simply JAR files that contain a Web application directory and use *.war* instead of *.jar* for file extensions. See Section 4.3 for a discussion of creating and using WAR files.

With the Resin server from Caucho, you use a `web-app` element within *web.xml* and supply `app-dir` (directory) and `id` (URL prefix) attributes. Resin even lets you use regular expressions in the `id`. So, for example, you can automatically give users their own Web apps that are accessed with URLs of the form *http://hostname/~username/*.

With the BEA WebLogic 6 Server, you have two choices. First, you can place a directory (see Section 4.2) containing a Web application into the *config/domain/applications* directory, and the server will automatically assign the Web application a URL prefix that matches the directory name. Second, you can create a WAR file (see Section 4.3) and use the Web Applications entry of the Administration Console to deploy it.

Registering a Web Application with Tomcat

With Tomcat 4, creating a Web application consists simply of creating the appropriate directory structure and restarting the server. For extra control over the process, you can modify *install_dir/conf/server.xml* (a Tomcat-specific file) to refer to the Web application. The following steps walk you through what is required to create a Web app that is accessed by means of URLs that start with *http://host/webapp1/*. These examples are taken from Tomcat 4.0, but the process for Tomcat 3 is very similar.

1. **Create a simple directory called *webapp1*.** Since this is your personal development directory, it can be located at any place you find convenient. Once you have a *webapp1* directory, place a simple JSP page called *HelloWebApp.jsp* (Listing 4.1) in it. Put a simple servlet called *HelloWebApp.class* (compiled from Listing 4.2) in the *WEB-INF/classes* subdirectory. Section 4.2 gives details on the directory structure of a Web application, but for now just note that the JSP pages, HTML documents, images, and other regular Web documents go in the top-level directory of the Web app, whereas servlets are placed in the *WEB-INF/classes* subdirectory.

   You can also use subdirectories relative to those locations, although recall that a servlet in a subdirectory must use a package name that matches the directory name.

   Finally, although Tomcat doesn't actually require it, it is a good idea to include a *web.xml* file in the *WEB-INF* directory. The *web.xml* file, called *the deployment descriptor*, is completely portable across servers. We'll see some uses for this deployment descriptor later in this chapter, and Chapter 5 (Controlling Web Application Behavior with

web.xml) will discuss it in detail. For now, however, just copy the existing *web.xml* file from *install_dir/webapps/ROOT/WEB-INF* or use the version that is online under Chapter 4 of the source code archive at *http://www.moreservlets.com*. In fact, for purposes of testing Web application deployment, you might want to start by simply downloading the entire *webapp1* directory from *http://www.moreservlets.com*.

2. **Copy that directory to *install_dir/webapps*.** For example, suppose that you are running Tomcat version 4.0, and it is installed in *C:\jakarta-tomcat-4.0*. You would then copy the *webapp1* directory to the *webapps* directory, resulting in *C:\jakarta-tomcat-4.0\webapps\webapp1\HelloWebApp.jsp*, *C:\jakarta-tomcat-4.0\webapps\webapp1\WEB-INF\classes\HelloWebApp.class*, and *C:\jakarta-tomcat-4.0\webapps\webapp1\WEB-INF\web.xml*. You could also wrap the directory inside a WAR file (Section 4.3) and simply drop the WAR file into *C:\jakarta-tomcat-4.0\webapps*.

3. **Optional: add a `Context` entry to *install_dir/conf/server.xml*.** If you want your Web application to have a URL prefix that exactly matches the directory name and you are satisfied with the default Tomcat settings for Web applications, you can omit this step. But, if you want a bit more control over the Web app registration process, you can supply a `Context` element in *install_dir/conf/server.xml*. If you do edit *server.xml*, be sure to make a backup copy first; a small syntax error in *server.xml* can completely prevent Tomcat from running.

   The `Context` element has several possible attributes that are documented at *http://jakarta.apache.org/tomcat/tomcat-4.0-doc/config/context.html*. For instance, you can decide whether to use cookies or URL rewriting for session tracking, you can enable or disable servlet reloading (i.e., monitoring of classes for changes and reloading servlets whose class file changes on disk), and you can set debugging levels. However, for basic Web apps, you just need to deal with the two required attributes: `path` (the URL prefix) and `docBase` (the base installation directory of the Web application, relative to *install_dir/webapps*). This entry should look like the following snippet. See Listing 4.3 for more detail.

```
<Context path="/webapp1" docBase="webapp1" />
```

   Note that you should not use */examples* as the URL prefix; Tomcat already uses that prefix for a sample Web application.

Core Warning

Do not use /examples *as the URL prefix of a Web application in Tomcat.*

4. **Restart the server.** I keep a shortcut to *install_dir/bin/startup.bat* (*install_dir/bin/startup.sh* on Unix) and *install_dir/bin/shutdown.bat* (*install_dir/bin/shutdown.sh* on Unix) in my development directory. I recommend you do the same. Thus, restarting the server involves simply double-clicking the shutdown link and then double-clicking the startup link.
5. **Access the JSP page and the servlet.** The URL *http://hostname/webapp1/HelloWebApp.jsp* invokes the JSP page (Figure 4–2), and *http://hostname/webapp1/servlet/HelloWebApp* invokes the servlet (Figure 4–3). During development, you probably use *localhost* for the host name. These URLs assume that you have modified the Tomcat configuration file (*install_dir/conf/server.xml*) to use port 80 as recommended in Chapter 1 (Server Setup and Configuration). If you haven't made this change, use *http://hostname**:8080**/webapp1/HelloWebApp.jsp* and *http://hostname**:8080**/webapp1/servlet/HelloWebApp*.

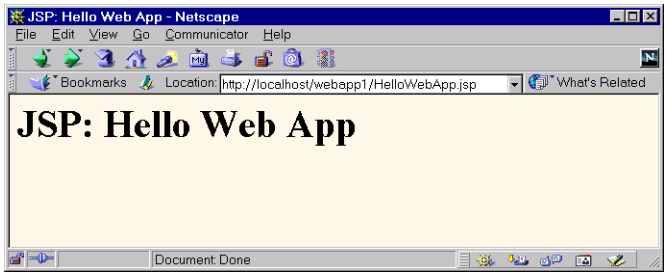


Figure 4–2 Invoking a JSP page that is in a Web application.

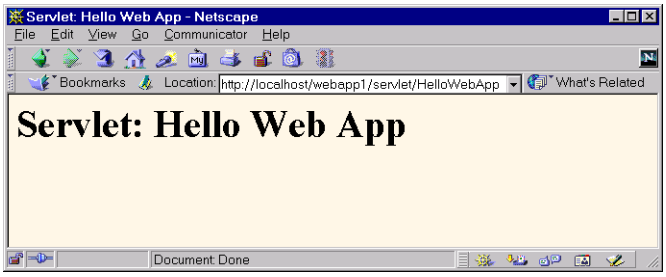


Figure 4–3 Invoking a servlet that is in a Web application.

Listing 4.1 *HelloWebApp.jsp*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>JSP: Hello Web App</TITLE></HEAD>
<BODY BGCOLOR="#FDF5E6">
<H1>JSP: Hello Web App</H1>
</BODY>
</HTML>
```

Listing 4.2 *HelloWebApp.java*

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWebApp extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
    String title = "Servlet: Hello Web App";
    out.println(docType +
                "<HTML>\n" +
                "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1>" + title + "</H1>\n" +
                "</BODY></HTML>");
  }
}
```

Listing 4.3 Partial *server.xml* for Tomcat 4

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Server>
  <!-- ... -->

  <!-- Having the URL prefix (path) match the actual directory
       (docBase) is a convenience, not a requirement. -->
  <Context path="/webapp1" docBase="webapp1" />
</Server>
```

Registering a Web Application with JRun

Registering a Web app with JRun 3.1 involves nine simple steps. The process is nearly identical to other versions of JRun.

1. **Create the directory.** Use the directory structure illustrated in Figure 4–1: a *webapp1* directory containing *HelloWebApp.jsp*, *WEB-INF/classes/HelloWebApp.class*, and *WEB-INF/web.xml*.
2. **Copy the entire *webapp1* directory to *install_dir/servers/default*.** The *install_dir/servers/default* directory is the standard location for Web applications in JRun. Again, I recommend that you simplify the process of copying the directory by using one of the methods described in Section 1.8 (Establish a Simplified Deployment Method). The easiest approach is to make a shortcut or symbolic link from your development directory to *install_dir/servers/default* and then simply copy the *webapp1* directory onto the shortcut whenever you redeploy. You can also deploy using WAR files (Section 4.3).
3. **Start the JRun Management Console.** You can invoke the Console either by selecting JRun Management Console from the JRun menu (on Microsoft Windows, this is available by means of Start, Programs, JRun) or by opening *http://hostname:8000/*. Either way, the JRun Admin Server has to be running first.
4. **Click on JRun Default Server.** This entry is in the left-hand pane, as shown in Figure 4–4.
5. **Click on Web Applications.** This item is in the bottom of the list that is created when you select the default server from the previous step. Again, see Figure 4–4.
6. **Click on Create an Application.** This entry is in the right-hand pane that is created when you select Web Applications from the previous step. If you deploy using WAR files (see Section 4.3) instead of an unpacked directory, choose Deploy an Application instead.

7. **Specify the directory name and URL prefix.** To tell the system that the files are in the directory *webapp1*, specify `webapp1` for the Application Name entry. To designate a URL prefix of */webapp1*, put `/webapp1` in the Application URL textfield. Note that you do not have to modify the Application Root Dir entry; that is done automatically when you enter the directory name. Press the Create button when done. See Figure 4–5.

8. **Restart the server.** From the JRun Management Console, click on JRun Default Server and then press the Restart Server button. Assuming JRun is not running as a Windows NT or Windows 2000 service, you can also double-click the JRun Default Server icon from the taskbar and then press Restart. See Figure 4–6.

9. **Access the JSP page and the servlet.** The URL *http://hostname/webapp1/HelloWebApp.jsp* invokes the JSP page (Figure 4–2), and *http://hostname/webapp1/servlet/HelloWebApp* invokes the servlet (Figure 4–3). During development, you probably use *localhost* for the host name. These are exactly the same URLs and results as with Tomcat and ServletExec. This approach assumes that you have modified JRun to use port 80 as recommended in Chapter 1 (Server Setup and Configuration). If you haven't made this change, use *http://hostname**:8100**/webapp1/HelloWebApp.jsp* and *http://hostname**:8100**/webapp1/servlet/HelloWebApp*.

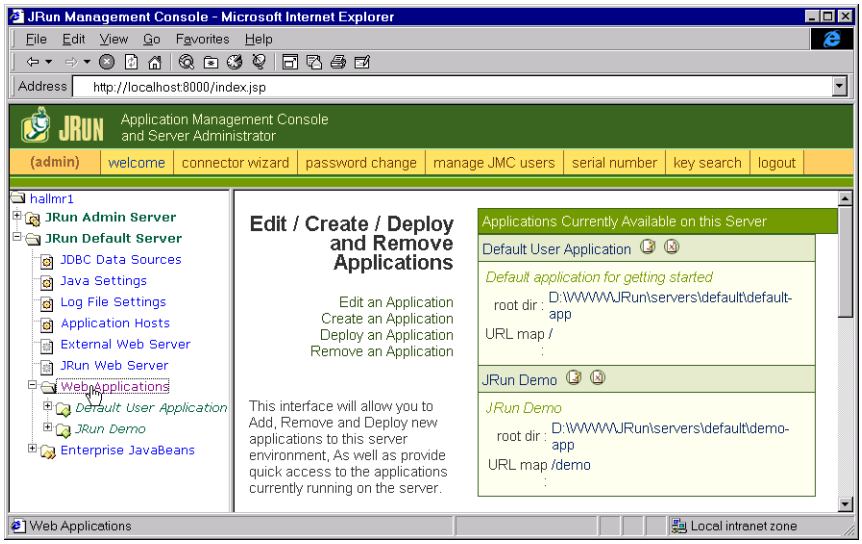


Figure 4–4 JRun Web application setup screen.

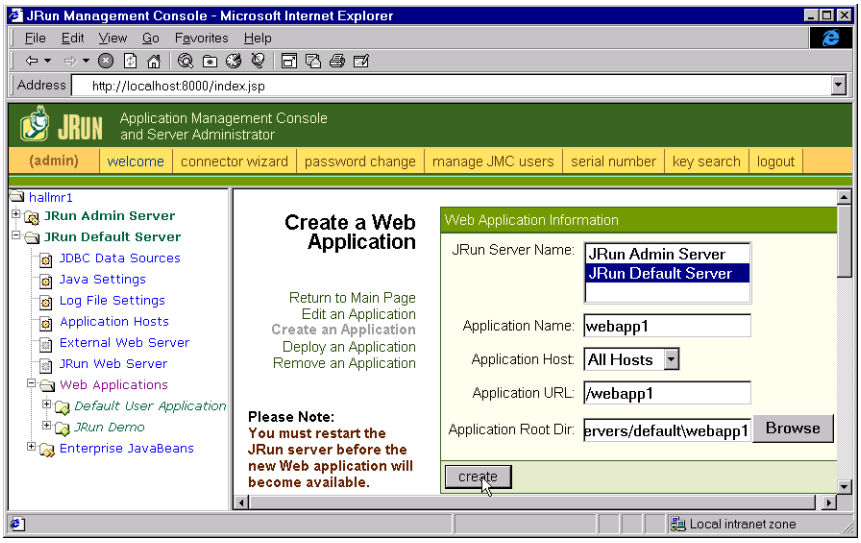
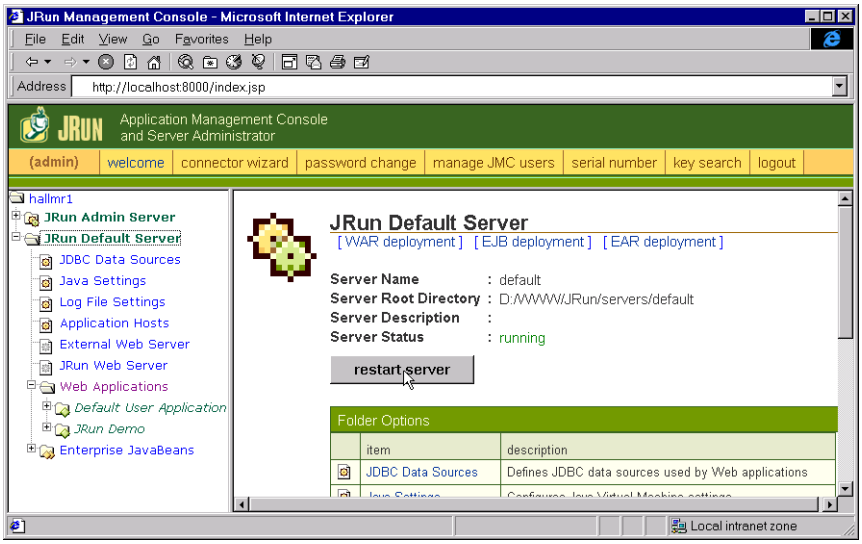


Figure 4–5 JRun Web application creation screen. You only need to fill in the Application Name and Application Root Dir entries.



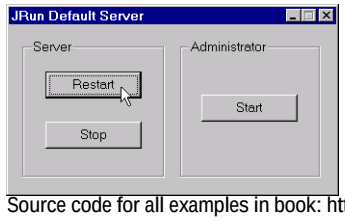


Figure 4–6 You must restart JRun for a newly created Web app to take effect.

Registering a Web Application with ServletExec

The process of registering Web applications is particularly simple with ServletExec 4. To make a Web app with a prefix *webapp1*, just create a directory called *webapp1* with the structure described in the previous two subsections. Drop this directory into *install_dir/webapps/default*, restart the server, and access resources in the Web app with URLs that begin with *http://hostname/webapp1/*. You can also drop WAR files (Section 4.3) in the same directory; the name of the WAR file (minus the *.war* extension) automatically is used as the URL prefix.

For more control over the process or to add a Web application when the server is already running, perform the following steps. Note that, using this approach, you do *not* need to restart the server after registering the Web app.

1. **Create a simple directory called *webapp1*.** Use the structure summarized in Figure 4–1: place a simple JSP page called *HelloWebApp.jsp* (Listing 4.1) in the top-level directory and put a simple servlet called *AppTest.class* (compiled from Listing 4.2) in the *WEB-INF/classes* subdirectory. Section 4.2 gives details on the directory structure of a Web app, but for now just note that the JSP pages, HTML documents, images, and other regular Web documents go in the top-level directory of the Web app, whereas servlets are placed in the *WEB-INF/classes* subdirectory. You can also use subdirectories relative to those locations, although recall that a servlet in a subdirectory must use a package name that matches the directory name. Later in this chapter (and throughout Chapter 5), we'll see uses for the *web.xml* file that goes in the *WEB-INF* directory. For now, however, you can omit this file and let ServletExec create one automatically, or you can copy a simple example from *http://www.moreservlets.com*. In fact, you can simply download the entire *webapp1* directory from the Web site.
2. **Optional: copy that directory to *install_dir/webapps/default*.** ServletExec allows you to store your Web application directory at any place on the system, so it is possible to simply tell ServletExec where the existing *webapp1* directory is located. However, I find it convenient to keep separate development and deployment copies of my Web applications. That way, I can develop continually but only deploy periodically. Since *install_dir/webapps/default* is the standard location for ServletExec Web applications, that's a good location for your deployment directories.
3. **Go to the ServletExec Web app management interface.** Access the ServletExec administration interface by means of the URL *http://hostname* and select Manage under the Web Applications heading. During development, you probably use *localhost* for the

host name. See Figure 4–7. This assumes that you have modified ServletExec to use port 80 as recommended in Chapter 1 (Server Setup and Configuration). If you haven't made this change, use *http://hostname**:8080***.

4. **Enter the Web app name, URL prefix, and directory location.** From the previous user interface, select Add Web Application (see Figure 4–7). This results in an interface (Figure 4–8) with text fields for the Web application configuration information. It is traditional, but not required, to use the same name (e.g., *webapp1*) for the Web app name, the URL prefix, and the main directory that contains the Web application.

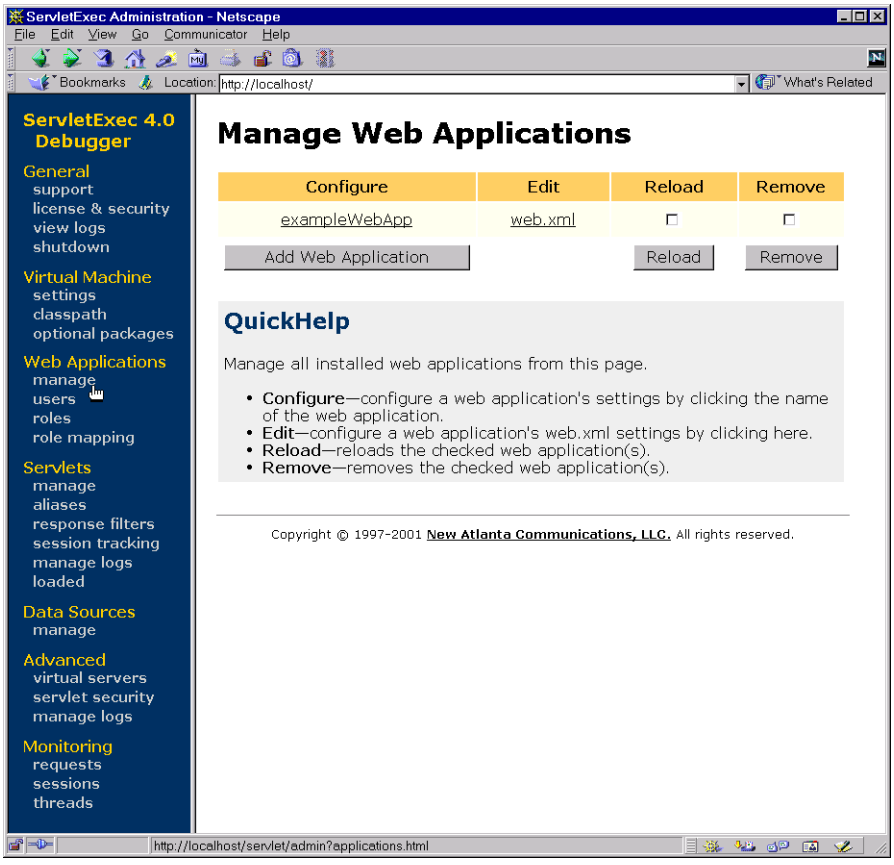


Figure 4–7 ServletExec interface for managing Web applications.

5. **Add the Web application.** After entering the information from Item 4, select Add Web Application. See Figure 4–8.
6. **Access the JSP page and the servlet.** The URL *http://hostname/webapp1/HelloWebApp.jsp* invokes the JSP page (Figure 4–2), and *http://hostname/webapp1/servlet/HelloWebApp* invokes the servlet (Figure 4–3). During development, you probably use *localhost* for the host name. These are exactly the same URLs and results as with Tomcat and JRun. This assumes that you have modified ServletExec to use port 80 as recommended in Chapter 1 (Server Setup and Configuration). If you haven't made this change, use *http://hostname**:8080**/webapp1/HelloWebApp.jsp* and *http://hostname**:8080**/webapp1/servlet/HelloWebApp*.

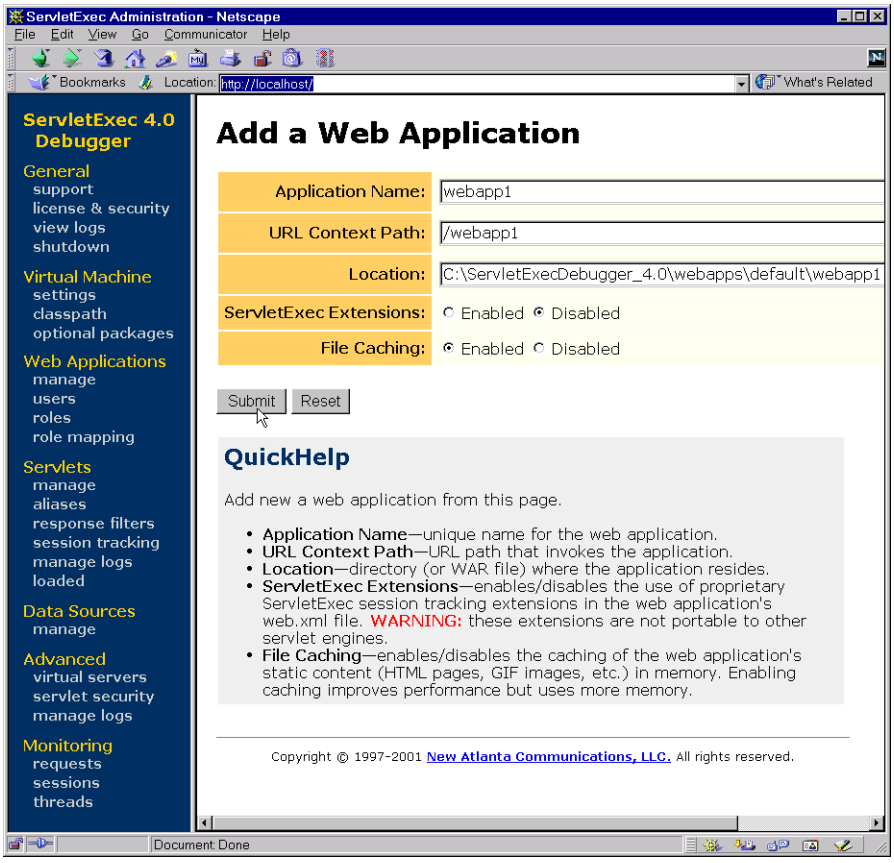


Figure 4–8 ServletExec interface for adding new Web applications.

4.2 Structure of a Web Application

The process of *registering* a Web application is not standardized; it frequently involves server-specific configuration files or user interfaces. However, the Web application *itself* has a completely standardized format and is totally portable across all Web or application servers that support version 2.2 or later of the servlet specification. The top-level directory of a Web application is simply a directory with a name of your choosing. Within that directory, certain types of content go in designated locations. This section provides details on the type of content that is placed in various locations; it also gives a sample Web application layout.

Locations for Various File Types

Quick summary: JSP pages and other normal Web documents go in the top-level directory, unbundled Java classes go in the *WEB-INF/classes* directory, JAR files go in *WEB-INF/lib*, and the *web.xml* file goes in *WEB-INF*. Figure 4–9 shows a representative example. For details or a more explicit sample hierarchy, check out the following subsections.

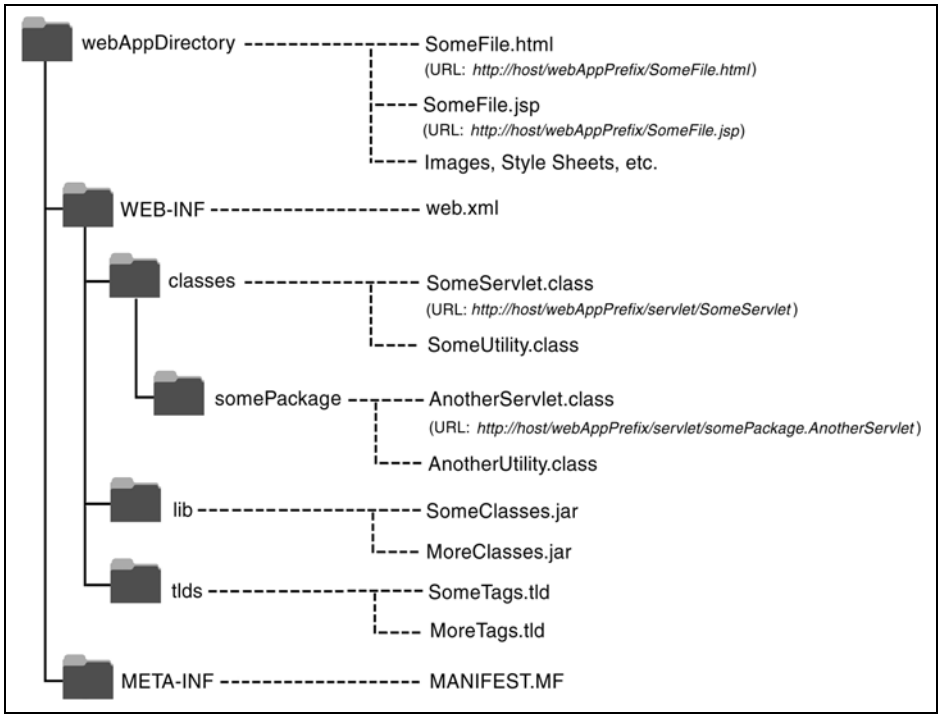


Figure 4–9 A representative Web application.

JSP Pages

JSP pages should be placed in the top-level Web application directory or in a subdirectory with any name other than *WEB-INF* or *META-INF*. Servers are prohibited from serving files from *WEB-INF* or *META-INF* to the user. When you register a Web application (see Section 4.1), you tell the server the URL prefix that designates the Web app and define where the Web app directory is located. It is common, but by no means mandatory, to use the name of the main Web application directory as the URL prefix. Once you register a prefix, JSP pages are then accessed with URLs of the form *http://hostname/webAppPrefix/filename.jsp* (if the pages are in the top-level directory of the Web application) or *http://hostname/webAppPrefix/subdirectory/filename.jsp* (if the pages are in a subdirectory).

It depends on the server whether a default file such as *index.jsp* can be accessed with a URL that specifies only a directory (e.g., *http://hostname/webAppPrefix/*) without the developer first making an entry in the Web app's *WEB-INF/web.xml* file. If you want *index.jsp* to be the default filename, I strongly recommend that you make an explicit `welcome-file-list` entry in your Web app's *web.xml* file. For example, the following *web.xml* entry specifies that if a URL gives a directory name but no filename, the server should try *index.jsp* first and *index.html* second. If neither is found, the result is server specific (e.g., a directory listing).

```
<welcome-file-list>
  <welcome-file>index.jsp</welcome-file>
  <welcome-file>index.html</welcome-file>
</welcome-file-list>
```

For details, see Section 5.7 (Specifying Welcome Pages).

HTML Documents, Images, and Other Regular Web Content

As far as the servlet and JSP engine is concerned, HTML files, GIF and JPEG images, style sheets, and other Web documents follow exactly the same rules as do JSP pages. They are placed in exactly the same locations and accessed with URLs of exactly the same form. In deployment scenarios, however, a servlet or JSP engine such as JRun, ServletExec, Tomcat, or Resin is often plugged into a regular Web server like Microsoft IIS, Apache, or older versions of the Netscape Web server. In such a case, the regular Web server usually serves regular Web pages more quickly than does the servlet and JSP engine. So, if your static Web documents are accessed extremely frequently, you are faced with a portability vs. performance trade-off. Putting the static documents in the Web application hierarchy lets you move the servlets, the JSP pages, *and* the static documents from server to server with a minimum of

changes. Putting the static resources in the regular Web server's hierarchy increases performance but requires server-specific changes when you move the Web app to another server. Fortunately, this issue is important only for the highest-traffic pages.

It depends on the server whether a default file such as *index.html* can be accessed with a URL that specifies only a directory (e.g., *http://hostname/webAppPrefix/*) without the developer first making an entry in the Web app's *WEB-INF/web.xml* file. If you want *index.html* to be the default filename, I recommend that you make an explicit `welcome-file-list` entry in *web.xml*. For details, see Section 5.7 (Specifying Welcome Pages).

Servlets, Beans, and Helper Classes (Unbundled)

Servlets and other *.class* files are placed either in *WEB-INF/classes* or in a subdirectory of *WEB-INF/classes* that matches their package name. During development, don't forget that your `CLASSPATH` should include the *classes* directory. The *server* already knows about this location, but your *development* environment does not. In order to compile servlets that are in packages, the compiler needs to know the location of the top-level directory of your package hierarchy. See Section 1.6 (Set Up Your Development Environment) for details.

The default way to access servlets is with URLs of the form *http://hostname/webAppPrefix/servlet/ServletName* or *http://hostname/webAppPrefix/servlet/packageName.ServletName*. To designate a different URL, you use the `servlet-mapping` element in the *web.xml* deployment descriptor file that is located within the *WEB-INF* directory of the Web application. See Section 5.3 (Assigning Names and Custom URLs) for details.

Servlets, Beans, and Helper Classes (Bundled in JAR Files)

If the servlets or other *.class* files are bundled inside JAR files, then the JAR files should be placed in *WEB-INF/lib*. If the classes are in packages, then within the JAR file they should be in a directory that matches their package name.

Deployment Descriptor

The deployment descriptor file, *web.xml*, should be placed in the *WEB-INF* subdirectory of the main Web application directory. For details on using *web.xml*, see Chapter 5 (Controlling Web Application Behavior with web.xml). Note that a few servers (e.g., Tomcat) have a global *web.xml* file that applies to all Web applications. That file is entirely server specific; the only standard *web.xml* file is the per-application one that is placed within the *WEB-INF* directory of the Web app.

Tag Library Descriptor Files

TLD files can be placed almost anywhere within the Web application. However, I recommend that you put them in a *tlds* directory within *WEB-INF*. Grouping them in a common directory (e.g., *tlds*) simplifies their management. Placing that directory within *WEB-INF* prevents end users from retrieving them. JSP pages, however, can access TLD files that are in *WEB-INF*. They just use a `taglib` element as follows

```
<%@ taglib uri="/WEB-INF/tlds/myTaglibFile.tld" ...%>
```

Since it is the server, not the client, that accesses the TLD file in this case, the prohibition that content inside of *WEB-INF* is not Web accessible does not apply.

WAR Manifest File

When you create a WAR file (see Section 4.3), a *MANIFEST.MF* file is placed in the *META-INF* subdirectory. Normally, the `jar` utility automatically creates *MANIFEST.MF* and places it in the *META-INF* directory, and you ignore it if you unpack the WAR file. Occasionally, however, you modify *MANIFEST.MF* explicitly (see Section 4.4), so it is useful to know where it is stored.

Sample Hierarchy

Suppose you have a Web application that is in a directory named *widgetStore* and is registered (see Section 4.1) with the URL prefix */widgetStore*. Following is one possible structure for the Web app.

widgetStore/orders.jsp
widgetStore/specials.html
These files would be accessed with the URLs *http://hostname/widgetStore/orders.jsp* and *http://hostname/widgetStore/specials.html*, respectively.

widgetStore/info/company-profile.jsp
widgetStore/info/contacts.html
These files would be accessed with the URLs *http://hostname/widgetStore/info/company-profile.jsp* and *http://hostname/widgetStore/info/contacts.html*, respectively.

widgetStore/founder.jpg
Since the *orders.jsp* and *specials.html* files are in the same directory as this file, they would use a simple relative URL to refer to the image, as below.

```
<IMG SRC="founder.jpg" ...>
```

Since *company-profile.jsp* and *contacts.html* are in a lower-level directory, they would use a relative URL that contains "..", as below.

```
<IMG SRC="../founder.jpg" ...>
```

But what if you want to support the flexibility of moving a JSP page to a different directory without changing the URL that refers to the image? Or what if a servlet wants to refer to this image? This is slightly more complicated; see Section 4.5 (Handling Relative URLs in Web Applications) for a discussion of the problem and its solutions.

widgetStore/images/button1.gif

Since the *orders.jsp* and *specials.html* files are in the parent directory of this file, they would refer to the image by using a relative URL that contains the directory name, as below.

```
<IMG SRC="images/button1.gif" ...>
```

Since *company-profile.jsp* and *contacts.html* are in a sibling directory, they would use a relative URL that contains ".." and the directory name, as below.

```
<IMG SRC="../images/founder.gif" ...>
```

Again, if you want to be able to move the JSP page without changing the image URL or if you want to refer to the image from a servlet, things are a bit complicated. See Section 4.5 (Handling Relative URLs in Web Applications) for a discussion of the problem and its solutions.

widgetStore/WEB-INF/tlds/widget-taglib.tld

This tag library descriptor file would be referenced from a JSP page by use of a `taglib` element as follows.

```
<%@ taglib uri="/WEB-INF/tlds/widget-taglib.tld" ...%>
```

Note that the JSP page that uses this TLD file can be located anywhere within the *widgetStore* Web app directory. Note, too, that there is no potential problem regarding relative URLs as there is with images (as mentioned in the previous two subsections). Also note that it is legal (recommended, in fact) to place the *tlds* directory within the *WEB-INF* directory, even though the *WEB-INF* directory is not accessible to Web clients. It is legal because the server, not the client, retrieves the TLD file.

widgetStore/WEB-INF/web.xml

This is the deployment descriptor. It is not accessible by Web clients; it is used only by the server itself. See Chapter 5 (Controlling Web Application Behavior with web.xml) for details on its use.

widgetStore/WEB-INF/classes/CheckoutServlet.class
This packageless servlet would be accessed either with the URL *http://hostname/widgetStore/servlet/CheckoutServlet* or with a custom URL that starts with *http://hostname/widgetStore/*. The *web.xml* file would define the custom URL; see Section 5.3 (Assigning Names and Custom URLs) for details.

widgetStore/WEB-INF/classes/cart/ShowCart.class
This servlet from the `cart` package would be accessed either with the URL *http://hostname/widgetStore/servlet/cart.ShowCart* or with a custom URL that starts with *http://hostname/widgetStore/*. Again, the *web.xml* file would define the custom URL. Remember that dots, not slashes, separate package names from class names in URLs that refer to servlets. So be sure to use *http://hostname/widgetStore/servlet/cart.ShowCart*, not *http://hostname/widgetStore/servlet/cart/ShowCart*.

widgetStore/WEB-INF/lib/Utils.jar
The *Utils.jar* file could contain utility classes used by the servlets and by various JSP pages. If the classes are in packages, they should be in subdirectories within the JAR file, and the servlets or JSP pages that use them must utilize `import` statements.

4.3 Deploying Web Applications in WAR Files

WAR (Web ARchive) files provide a convenient way of bundling Web apps in a single file. Having a single large file instead of many small files makes it easier to transfer the Web application from server to server.

A WAR file is really just a JAR file with a *.war* extension, and you use the normal `jar` command to create it. For example, to bundle the entire `widgetStore` Web app into a WAR file named *widgetStore.war*, you would just change directory to the *widgetStore* directory and execute the following command.

```
jar cvf widgetStore.war *
```

For simple WAR files, that's it! However, in version 2.3 of the servlet API, you can create WAR files that designate that they need shared but nonstandard libraries installed on the server. This topic is covered in Section 4.4.

Of course, you can use other `jar` options (e.g., to digitally sign classes) with WAR files just as you can with regular JAR files. For details, see *http://java.sun.com/j2se/*

1.3/docs/tooldocs/win32/jar.html (Windows) or *http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/jar.html* (Unix/Linux).

Finally, remember that you have to follow slightly different procedures to register Web apps that are contained in WAR files than you do to deploy unbundled Web applications. For details, see Section 4.1 (Registering Web Applications).

4.4 Recording Dependencies on Server Libraries

With servlet and JSP engines (or "containers") supporting the servlet 2.2 and JSP 1.1 API, there is no portable way to designate that a Web app depends on some shared library that is not part of the servlet or JSP API itself. You have to either copy the library's JAR file into the *WEB-INF/lib* directory of each and every Web application, or you have to make server-specific changes that lack mechanisms for verification.

With servlet version 2.3 (JSP version 1.2), you can use the *META-INF/MANIFEST.MF* file to express dependencies on shared libraries. Compliant containers are required to detect when these dependencies are unfulfilled and provide a warning. Note that although support for these dependencies is a new capability in servlet version 2.3, the actual method of expressing the dependencies is the standard one for the Java 2 Platform Standard Edition, as described at *http://java.sun.com/j2se/1.3/docs/guide/extensions/versioning.html*. Furthermore, although the method for *expressing* dependencies is now standardized, the way to actually *implement* shared libraries is nonstandard. For example, with Tomcat 4, individual class files placed in *install_dir/classes* and JAR files placed in *install_dir/lib* are made available to all Web applications. Other servers might use an entirely different approach or might completely disallow code sharing across Web applications.

Creating a Manifest File

The *MANIFEST.MF* file is created automatically by the `jar` utility. Unless you are using shared libraries, you normally ignore this file altogether. Even if you do want to customize the manifest file so that you can use shared libraries more portably, you rarely edit it directly. Instead, you typically create a text file with a subset of manifest file entries and then use the `m` option to tell `jar` to add the contents of the text file to the autogenerated manifest file. For example, suppose that you have a file called *myAppdependencies.txt* that is in the top-level directory of your Web application. You can create a WAR file called *myApp.war* by changing directory to the top-level Web application directory and then issuing the following `jar` command.

```
jar cvmf myAppDependencies.txt myApp.war *
```

There are two small problems with this approach, however. First, *myAppDependencies.txt* is actually part of the WAR file. This inclusion is unnecessary and may confuse the deployer who sees an unneeded file mixed in with *index.jsp* and other top-level Web files. Second, the WAR file itself (*myApp.war*) is created in the top-level directory of the Web application. It is slightly more convenient to place the WAR file one directory level up from there, so that configuration and archive files are kept distinct from the Web application contents. Both of these minor problems can be solved by use of the `C` option to `jar`, which instructs `jar` to change directories before adding files to the archive. You place *myAppDependencies.txt* one level *up* from the main Web application directory (*myApp* in the following example), change directory to the location containing *myAppDependencies.txt*, and issue the following jar command:

```
jar cvmf myAppDependencies.txt myApp.war -C myApp *
```

This way, the dependency file and the WAR file itself are located in the directory that contains the top-level Web application directory (*myApp*), not in *myApp* itself. Note that it is not legal to go to the parent directory of *myApp* and then use a directory name when specifying the files, as below.

```
jar cvmf myAppDependencies.txt myApp.war myApp/* // Wrong!
```

The reason this fails is that the name *myApp* incorrectly becomes part of all of the WAR entries except for *MANIFEST.MF*.

Contents of the Manifest File

OK, ok. So you know how to use `jar` to create manifest files. But what do you put *in* the manifest file that lets you specify dependencies on shared libraries? There are four main entries that you need, each of which consists of a single plain-text line of the form "`Entry: value`".

- **Extension-List.** This entry designates one or more names of your choosing, separated by spaces. The names will be used in the rest of the manifest file to identify the library of interest. For example, suppose that several of your Web applications use JavaMail to send email. Rather than repeating the JavaMail JAR file in each and every Web application, you might use a server-specific mechanism to make it available to all Web applications. Then, your dependency file would contain a line like the following.

  ```
  Extension-List: javaMail
  ```

- ***name*-Extension-Name.** This entry designates the standard name for the library of interest. You cannot choose an arbitrary name here; you must supply the name exactly as it is given in the `Extension-Name` entry of the JAR file that contains the actual library of interest. For most standard extensions, the package name is used as the extension name. Note that the leading name must match whatever you specified for `Extension-List`. So, if the standard JavaMail JAR file uses an `Extension-Name` of `javax.mail`, the first two entries in your dependency file would look like the following.

```
Extension-List: javaMail
javaMail-Extension-Name: javax.mail
```

- ***name*-Specification-Version.** This entry gives the minimum required specification version of the library of interest. When the server finds an installed library, it compares the listed `Specification-Version` to the minimum specified here. For example, if your Web apps require JavaMail version 1.2 or later, your dependency file might start as follows.

```
Extension-List: javaMail
javaMail-Extension-Name: javax.mail
javaMail-Specification-Version: 1.2
```

- ***name*-Implementation-URL.** This entry lets you specify the location where the server can find the JAR file. Note, however, that it is unclear how many servers (if any) will automatically download the JAR file when necessary, rather than simply generating an error message. Combining all four entries results in a dependency file like the following.

```
Extension-List: javaMail
javaMail-Extension-Name: javax.mail
javaMail-Specification-Version: 1.2
javaMail-Implementation-URL: http://somehost.com/javaMail.jar
```

Although these are the four most important entries, there are a number of other possible entries. For details, see *http://java.sun.com/j2se/1.3/docs/guide/extensions/versioning.html*. For information on JavaMail, see *http://java.sun.com/products/javamail/*.

4.5 Handling Relative URLs in Web Applications

Suppose you have an image that you want displayed in a JSP page. If the image is used only by that particular JSP page, you can place the image and the JSP page in the same directory; the JSP page can then use a simple relative URL to name the image, as below:

```
<IMG SRC="MyImage.gif" WIDTH="..." HEIGHT="..." ALT="...">
```

If there are lots of different images, it is usually convenient to group them in a subdirectory. But each URL remains simple:

```
<IMG SRC="images/MyImage.gif" ...>
```

So far, so good. But what if the same image is used by JSP or HTML pages that are scattered throughout your application? Copying the image lots of places would be wasteful and would make updating the image difficult. And even that wouldn't solve all your problems. For example, what if you have a servlet that uses an image? After all, you can't just use

```
out.println("<IMG SRC=\"MyImage.gif\" ...>"); // Fails!
```

because the browser would treat the image location as relative to the servlet's URL. But the default URL of a servlet is *http://host/webAppPrefix/servlet/ServletName*. Thus, the browser would resolve the relative URL *MyImage.gif* to *http://host/webAppPrefix/servlet/MyImage.gif*. That, of course, will fail since *servlet* is not really the name of a directory; it is just an artifact of the default URL mapping. You have precisely the same problem when using the MVC architecture (see Section 3.8) where a `RequestDispatcher` forwards the request from a servlet to a JSP page. The browser only knows about the URL of the original servlet and thus treats image URLs as relative to that location.

If you aren't using Web applications, you can solve all these problems the same way: by using a URL that is relative to the server's root directory, not relative to the location of the file that uses the image. For example, you could put the images in a directory called *images* that is in the root directory. Then, a JSP page could use

```
<IMG SRC="/images/MyImage.gif" ...>
```

and a servlet could do

```
out.println("<IMG SRC=\"/images/MyImage.gif\" ...>");
```

Unfortunately, however, this trick fails when you use Web applications. If a JSP page uses an image URL of

```
<IMG SRC="/images/MyImage.gif" ...>
```

the browser will request the image from the main server root, not from the base location of the Web application.

Exactly the same problem occurs with style sheets, applet class files, and even simple hypertext links that use URLs that begin with slashes. Note, however, that this problem does not occur in situations where the server resolves the URLs, only in cases where the browser does so. So, for example, it is perfectly safe for a JSP page that is in a Web app to do

```
<%@ taglib uri="/tlds/SomeFile.tld" ... %>
```

The server will correctly treat that URL as referring to the *tlds* directory within the Web application. Similarly, there is no problem using URLs that begin with / in locations passed to the `getRequestDispatcher` method of `ServletContext`; the server resolves them with respect to the Web application's root directory, not the overall server root.

Core Approach

URLs that are returned to the browser need to be handled specially. URLs that are handled by the server need not be.

There are three possible solutions to this dilemma. The first is the most commonly used but the least flexible. I recommend option (2) or (3).

1. **Use the Web application name in the URL.** For example, you could create a subdirectory called *images* within your Web application, and a JSP page could refer to an image with the following URL.
   `<IMG SRC="/webAppPrefix/images/MyImage.gif" ...>`
   This would work both for regular JSP pages and JSP pages that are invoked by means of a `RequestDispatcher`. Servlets could use the same basic strategy. However, this approach has one serious drawback: if you change the URL prefix of the Web application, you have to change a large number of JSP pages and servlets. This restriction is unacceptable in many situations; you want to be able to change the Web application's URL prefix without changing any of the files *within* the Web app.

2. **Assign URLs that are at the top level of the Web application.** For example, suppose you had a servlet named `WithdrawServlet` that was in the `banking` package of a Web application named `financial`. The default URL to invoke that servlet would be *http://host/financial/servlet/banking.WithdrawServlet*. Thus, the servlet would suffer from the problems just discussed when using images, style sheets, and so forth. But, there is no requirement that you use the default URL. In fact, many people feel that you should avoid default URLs in deployment scenarios. Instead, you can use the *web.xml* file to assign a URL that does not contain the *servlet* "subdirectory" (see Section 5.3, "Assigning Names and Custom URLs"). For example, Listing 4.4 shows a *web.xml* file that could be used to change the URL from *http://host/financial/servlet/banking.WithdrawServlet* to *http://host/financial/Withdraw*. Now, since the URL does not contain a "subdirectory" named *servlet*, the servlet can use simple relative URLs that contain only the filename or the subdirectory and the file, but without using a /. For instance, if the Web application contained an image called *Cash.jpg*, you could place it in the Web app's *images* directory and the servlet could use

```
out.println("<IMG SRC=\"images/Cash.jpg\" ...>");
```

Listing 4.4 *web.xml* that assigns top-level URL

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
  <servlet>
    <servlet-name>WithdrawServlet</servlet-name>
    <servlet-class>banking.WithdrawServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>WithdrawServlet</servlet-name>
    <url-pattern>Withdraw</url-pattern>
  </servlet-mapping>
</web-app>
```

3. **Use `getContextPath`.** The most general solution is to explicitly add the Web application name to the front of each URL that begins with /. However, instead of hardcoding the name, you can use the `getContextPath` method of `HttpServletRequest` to determine

the name at execution time. For example, a JSP page could do the following.

```
<% String prefix = request.getContextPath();
   String url = prefix + "/images/MyImage.jpg"; %>
<IMG SRC="<%= url %>" ...>
```

If you have a number of URLs of this nature, you can make use of the `BASE` element to standardize the location to which relative URLs are resolved. For example:

```
<HEAD>
<BASE HREF="<%= request.getContextPath() %>">
<TITLE>...</TITLE>
</HEAD>
```

The use of `getContextPath` is so generally applicable that it is worth capturing some of this functionality in a reusable utility. Listing 4.5 presents one such utility that not only modifies regular URLs, but also handles URLs that are to be used for session tracking that is based on URL rewriting.

Listing 4.5 *AppUtils.java*

```
package moreservlets;

import javax.servlet.http.*;

/** A small set of utilities to simplify the use of URLs in
 *  Web applications.
*/

public class AppUtils {

  /** For use in URLs referenced by JSP pages or servlets, where
   *  you want to avoid hardcoding the Web app name. Replace
   *         <PRE><XMP>
   *  <IMG SRC="/images/foo.gif" ...>
   *  with the following two lines:
   *  <% String imageURL = webAppURL("/images/foo.gif",
   *                                 request); %>
   *  <IMG SRC="<%= imageURL %>"...>
   *
   *         </XMP></PRE>
   */
```

Listing 4.5 *AppUtils.java (continued)*

```
  public static String webAppURL(String origURL,
                                 HttpServletRequest request) {
    return(request.getContextPath() + origURL);
  }

  /** For use when you want to support session tracking with
   *  URL encoding and you are putting a URL
   *  beginning with a slash into a page from a Web app.
   */

  public static String encodeURL(String origURL,
                                 HttpServletRequest request,
                                 HttpServletResponse response) {
    return(response.encodeURL(webAppURL(origURL, request)));
  }

  /** For use when you want to support session tracking with
   *  URL encoding and you are using sendRedirect to send a URL
   *  beginning with a slash to the client.
   */

  public static String encodeRedirectURL
                                (String origURL,
                                 HttpServletRequest request,
                                 HttpServletResponse response) {
    return(response.encodeRedirectURL
                          (webAppURL(origURL, request)));
  }
}
```

4.6 Sharing Data Among Web Applications

One of the major purposes of Web applications is to keep data separate. Each Web application maintains its own table of sessions and its own servlet context. Each Web application also uses its own class loader; this behavior eliminates problems with name conflicts but means that static methods and fields can't be used to share data among applications. However, it *is* still possible to share data with cookies or by using `ServletContext` objects that are associated with specific URLs. These two approaches are summarized below.

- **Cookies.** Cookies are maintained by the browser, not by the server. Consequently, cookies can be shared across multiple Web applications as long as they are set to apply to any path on the server. By default, the browser sends cookies only to URLs that have the same prefix as the one from which it first received the cookies. For example, if the server sends a cookie from the page associated with *http://host/path1/SomeFile.jsp*, the browser sends the cookie back to *http://host/path1/SomeOtherFile.jsp* and *http://host/path1/path2/Anything*, but not to *http://host/path3/Anything*. Since Web applications always have unique URL prefixes, this behavior means that default-style cookies will never be shared between two different Web applications.

  However, as described in Section 2.9 (Cookies), you can use the `setPath` method of the `Cookie` class to change this behavior. Supplying a value of `"/"`, as shown below, instructs the browser to send the cookie to *all* URLs at the host from which the original cookie was received.

```
Cookie c = new Cookie("name", "value");
c.setMaxAge(...);
c.setPath("/");
response.addCookie(c);
```

- **ServletContext objects associated with a specific URL.** In a servlet, you obtain the Web application's servlet context by calling the `getServletContext` method of the servlet itself (inherited from `GenericServlet`). In a JSP page, you use the predefined `application` variable. Either way, you get the servlet context associated with the servlet or JSP page that is making the request. However, you can also call the `getContext` method of `ServletContext` to obtain a servlet context—not necessarily your own—associated with a particular URL. This approach is illustrated below.

```
ServletContext myContext = getServletContext();
String url = "/someWebAppPrefix";
ServletContext otherContext = myContext.getContext(url);
Object someData = otherContext.getAttribute("someKey");
```

Neither of these two data-sharing approaches is perfect.

The drawback to cookies is that only limited data can be stored in them. Each cookie value is a string, and the length of each value is limited to 4 kilobytes. So, robust data sharing requires a database: you use the cookie value as a key into the database and store the real data in the database.

One drawback to sharing servlet contexts is that you have to know the URL prefix that the other Web application is using. You normally want the freedom to change a Web application's prefix without changing any associated code. Use of the `getContext` method restricts this flexibility. A second drawback is that, for security reasons, servers are permitted to prohibit access to the `ServletContext` of certain Web applications. Calls to `getContext` return `null` in such a case. For example, in Tomcat you can use a value of `false` for the `crossContext` attribute of the `Context` or `DefaultContext` element (specified in *install_dir/conf/server.xml*) to indicate that a Web application should run in a security-conscious environment and prohibit access to its `ServletContext`.

These two data-sharing approaches are illustrated by the `SetSharedInfo` and `ShowSharedInfo` servlets shown in Listings 4.6 and 4.7. The `SetSharedInfo` servlet creates custom entries in the session object and the servlet context. It also sets two cookies: one with the default path, indicating that the cookie should apply only to URLs with the same URL prefix as the original request, and one with a path of `"/"`, indicating that the cookie should apply to all URLs on the host. Finally, the `SetSharedInfo` servlet redirects the client to the `ShowSharedInfo` servlet, which displays the names of all session attributes, all attributes in the current servlet context, all attributes in the servlet context that applies to URLs with the prefix */shareTest1*, and all cookies.

Listing 4.6 *SetSharedInfo.java*

```
package moreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/** Puts some data into the session, the servlet context, and
 *  two cookies. Then redirects the user to the servlet
 *  that displays info on sessions, the servlet context,
 *  and cookies.
 */

public class SetSharedInfo extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    HttpSession session = request.getSession(true);
    session.setAttribute("sessionTest", "Session Entry One");
    ServletContext context = getServletContext();
    context.setAttribute("servletContextTest",
                         "Servlet Context Entry One");
```

Listing 4.6 *SetSharedInfo.java (continued)*

```
    Cookie c1 = new Cookie("cookieTest1", "Cookie One");
    c1.setMaxAge(3600);     // One hour
    response.addCookie(c1); // Default path
    Cookie c2 = new Cookie("cookieTest2", "Cookie Two");
    c2.setMaxAge(3600);     // One hour
    c2.setPath("/");        // Explicit path: all URLs
    response.addCookie(c2);
    String url = request.getContextPath() +
                 "/servlet/moreservlets.ShowSharedInfo";
    // In case session tracking is based on URL rewriting.
    url = response.encodeRedirectURL(url);
    response.sendRedirect(url);
  }
}
```

Listing 4.7 *ShowSharedInfo.java*

```
package moreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

/** Summarizes information on sessions, the servlet
 *  context and cookies. Illustrates that sessions
 *  and the servlet context are separate for each Web app
 *  but that cookies are shared as long as their path is
 *  set appropriately.
 */

public class ShowSharedInfo extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Shared Info";
    out.println(ServletUtilities.headWithTitle(title) +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=\"CENTER\">" + title + "</H1>\n" +
                "<UL>\n" +
                "  <LI>Session:");
```

Listing 4.7 *ShowSharedInfo.java (continued)*

```
    HttpSession session = request.getSession(true);
    Enumeration attributes = session.getAttributeNames();
    out.println(getAttributeList(attributes));
    out.println("  <LI>Current Servlet Context:");
    ServletContext application = getServletContext();
    attributes = application.getAttributeNames();
    out.println(getAttributeList(attributes));
    out.println("  <LI>Servlet Context of /shareTest1:");
    application = application.getContext("/shareTest1");
    attributes = application.getAttributeNames();
    out.println(getAttributeList(attributes));
    out.println("  <LI>Cookies:<UL>");
    Cookie[] cookies = request.getCookies();
    if ((cookies == null) || (cookies.length == 0)) {
      out.println("    <LI>No cookies found.");
    } else {
      Cookie cookie;
      for(int i=0; i<cookies.length; i++) {
        cookie = cookies[i];
        out.println("    <LI>" + cookie.getName());
      }
    }
    out.println("    </UL>\n" +
                "</UL>\n" +
                "</BODY></HTML>");
  }

  private String getAttributeList(Enumeration attributes) {
    StringBuffer list = new StringBuffer("  <UL>\n");
    if (!attributes.hasMoreElements()) {
      list.append("    <LI>No attributes found.");
    } else {
      while(attributes.hasMoreElements()) {
        list.append("    <LI>");
        list.append(attributes.nextElement());
        list.append("\n");
      }
    }
    list.append("  </UL>");
    return(list.toString());
  }
}
```

Figure 4–10 shows the result after the user visits the `SetSharedInfo` and `ShowSharedInfo` servlets from within the Web application that is assigned */shareTest1* as a URL prefix. The `ShowSharedInfo` servlet sees:

- The custom session attribute.
- The custom (explicitly created by the `SetSharedInfo` servlet) and standard (automatically created by the server) attributes that are contained in the default servlet context.
- The custom and standard attributes that are contained in the servlet context that is found by means of `getContext("/shareTest1")`, which in this case is the same as the default servlet context.
- The two explicitly created cookies and the system-created cookie used behind the scenes by the session tracking API.

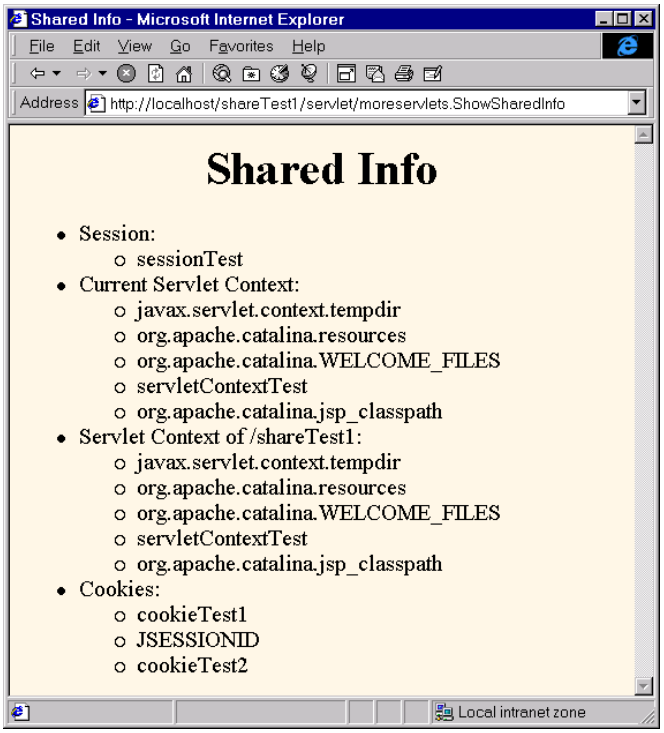


Figure 4–10 Result of visiting the `SetSharedInfo` and `ShowSharedInfo` servlets from within the same Web application.

Figure 4–11 shows the result when the user later visits an identical copy of the `ShowSharedInfo` servlet that is installed in a Web application that has */shareTest2* as the URL prefix. The servlet sees:

- The standard attributes that are contained in the default servlet context.
- The custom and standard attributes that are contained in the servlet context that is found by means of `getContext("/shareTest1")`, which in this case is different from the default servlet context.
- Two cookies: the explicitly created one that has its path set to `"/"` and the system-created one used behind the scenes for session tracking (which also uses a custom path of `"/"`).

The servlet does *not* see:

- Any attributes in its session object.
- Any custom attributes contained in the default servlet context.
- The explicitly created cookie that uses the default path.

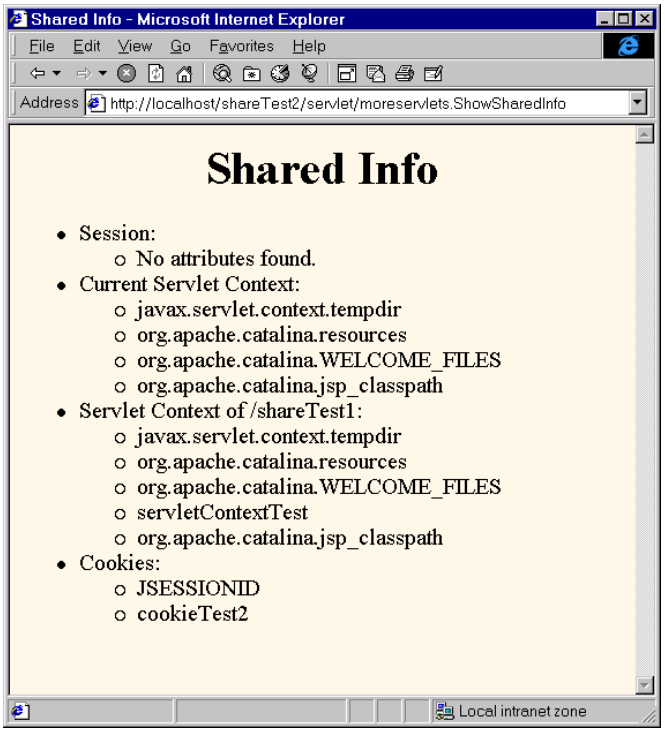


Figure 4–11 Result of visiting the `SetSharedInfo` servlet in one Web application and the `ShowSharedInfo` servlet in a different Web application.