S+JavaServer Pages Documentation

TABLE OF CONTENTS

Chapter 1: Introduction

The S-PLUS JavaServer Pages (S+JSP) module integrates S-PLUS into Web applications using S-PLUS Server for UNIX/Linux. In such an integrated application, Web pages use S-PLUS Server to perform customized analyses and produce reports and graphs. These results are displayed in the web page.

In this scenario the Web server acts as a client application to S-PLUS Server for UNIX/Linux running a number of S-PLUS sessions. Users submit requests to the Web server, which calls S-PLUS to run the required analyses and produce graphs, tables, and reports. The Web server packages these into Web pages that it returns to the users.

Figure 1-1 shows the overall architecture.

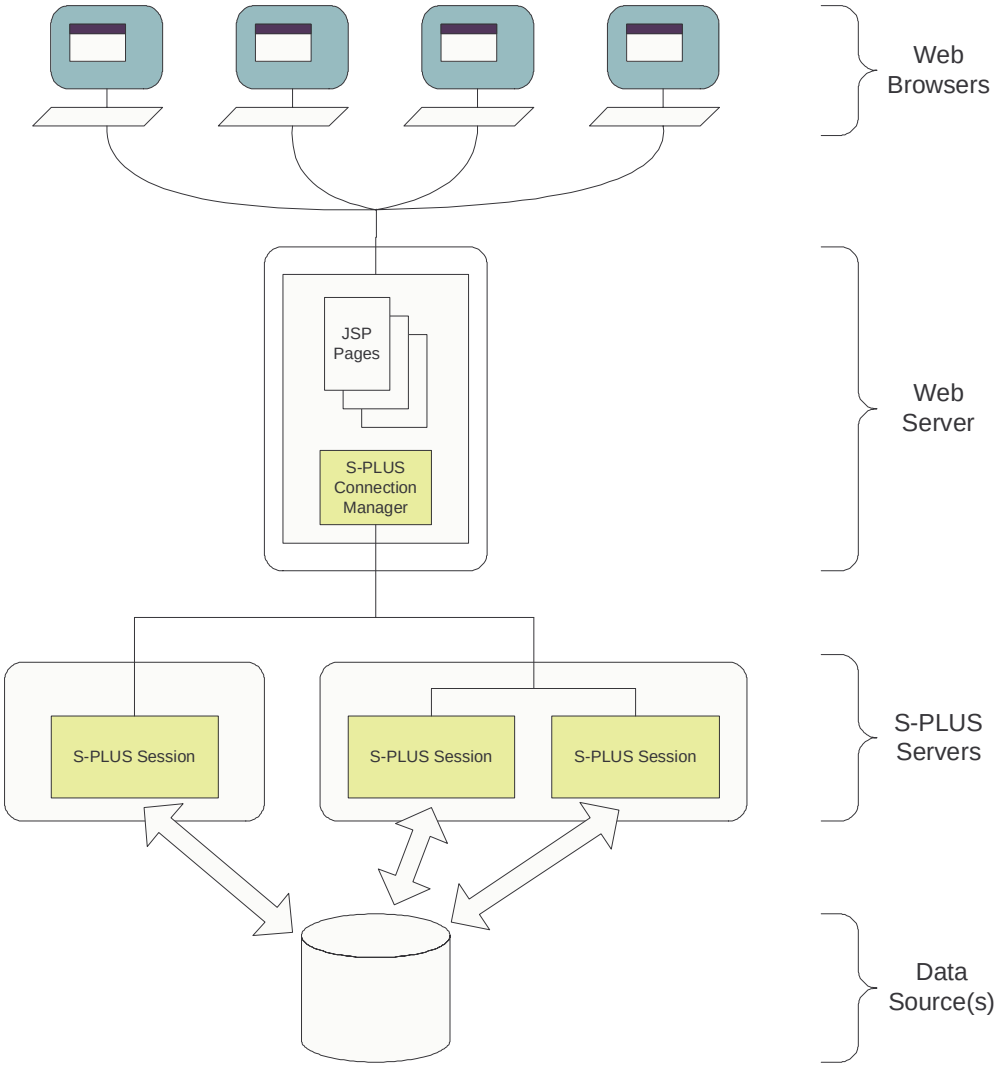

Figure 1-1. *Architecture of a Web application with S-PLUS Server.*

An application using this module uses JavaServer Pages (JSP) technology to process user requests. The JSP pages contain custom tags to access S-PLUS, in addition to the text and HTML to format the output. The JSP tags incorporate normal S-PLUS commands (typically function calls) that the tags run in an S-PLUS session. Each specific type of tag expects a

different type of output, which it includes in the Web page as text, as tables, or as graphs as appropriate.

The Web server runs an S-PLUS connection manager that pools a number of connections to running S-PLUS sessions and makes them available to incoming requests. When the number of requests exceeds the number of available sessions, the manager queues requests. An administrator can easily increase the number of S-PLUS connections to handle increased load, as long as hardware is available to run the sessions.

The S-PLUS connection manager, which manages multiple S-PLUS sessions, is thread-safe and fault tolerant. It can handle multiple requests received simultaneously by the Web server without interference. If one of the sessions becomes unavailable for some reason (for example, if the connection is lost), the connection manager is able to switch the job to another session within its pool. This process is automatic and invisible to the client's application.

Other Types of S-PLUS Server Applications

S-PLUS Server can also be used for other types of applications that do not require this module.

Some Web applications use graphs, tables and reports generated by S-PLUS, but they use previously generated versions of these results and do not need to call S-PLUS at request time. Such applications can use S-PLUS Server to prepare the components in batch mode every night or every week, but do not require this module.

Chapter 4 of the *S-PLUS Server Programmer's Guide* discusses two types of Web applications: those that use previously generated results and those that call S-PLUS at request time. The chapter discusses which approach to use for a particular application.

S-PLUS Server can also be used with an S-PLUS client GUI that comes with the product, and customers can write their own custom Java applications that access S-PLUS over a local network. Chapter 1 of the *S-PLUS Server Getting Started Guide* discusses these deployment scenarios.

Requirements

This module is intended to be used with S-PLUS Server 6.2 for UNIX/Linux. S-PLUS Server provides the advanced data analysis methods and graphics of S-PLUS 6.2 for UNIX/Linux in a distributed environment. It builds on the CONNECT/Java interface provided in S-PLUS and allows you to develop scalable applications that serve large numbers of users. This module is not for use with S-PLUS Server for Windows.

The module has been written to Version 2.3 of the Java Servlet specification and Version 1.2 of the JavaServer Pages specification, and it has been tested with Version 4.x and 5.x of Tomcat, the reference implementation for these APIs, as well as in several other Web containers. We anticipate that the module will run with little or no alteration in any Web server supporting these versions (or higher) of the APIs. Most of today's Web servers either support these APIs themselves or can be run with an add-on engine that supports them. A list of Web servers and add-on engines that support Java servlets and JSP pages is available at

```
http://java.sun.com/products/servlet/industry.html
```


The Java code portion of this module is designed to run with Version 1.4 (or higher) of the Java 2 Platform (J2SE). Your Web server might put more rigid requirements on the version of Java that you use.

Contents of this Module

This module consists of

- a main sample application,
- components and tools for authoring your own application
- documentation

The main sample application includes

- several example JSP pages illustrating how to use the custom tags to access S-PLUS
- pages that allow you to interactively experiment with your own tag code as well as access the underlying S-PLUS sessions
- pages for administering the underlying S-PLUS sessions
- an online tag reference system
- access to all documentation

We recommend that you install and try out this sample application as a way to begin learning about this module. Chapter 2 of this Guide provides installation instructions for the sample application, and Chapter 3 uses the sample application to provide a guided tour of the S-PLUS tag library.

The components and tools for authoring your own application include

- the S-PLUS connection manager code
- the S-PLUS tag library
- tools for accessing an S-PLUS session in a running Web application
- sample configuration files
- sample administrative Web pages

Chapter 4 of this Guide describes how to author your own Web application using the connection manager and the tag library. Chapter 5 tells how to write, edit, and deploy your own S-PLUS functions with your Web application. Chapter 6 discusses administrative issues such as the configuration files.

The documentation includes

- this Getting Started Guide
- a Tag library summary page
- a Tag library reference
- Javadoc for the Java code

Chapter 2: Setting Up the Sample Application

This Chapter describes how to set up and run the main S+JSP sample application. Later sections of this document — particularly *Chapter 6: Deployment and Administration* — provide more details.

One Web container that is convenient for setting up and running this sample application is Tomcat, which is available for free from Apache's Jakarta project (`http://jakarta.apache.org/`). Tomcat (Version 4.x or 5.x) is the official reference implementation for the Java servlet and JSP APIs that this module uses. Tomcat can run as a standalone Web server for demonstration and development purposes, or as a servlet and JSP container in a full-featured Web server (such as Apache).

These instructions explain how to set up the sample application with Tomcat running in standalone mode. If you use a different Web container, some of the details will be different; please refer to the documentation for the container you use.

The setup for the sample application consists of two parts:

1. Installing S-PLUS Server, and
2. Installing the Web container and the sample application code.

The S-PLUS Server and the Web container can be on the same computer or on different computers on a network. S-PLUS Server for UNIX/Linux will run on a Linux, Solaris or AIX system. Tomcat, a pure Java application, will run under any operating system that supports Java, including Windows and Unix. If you use a different Web server, check with the manufacturer to see what operating systems it will run under.

One handy configuration for testing and development is to run Tomcat on your desktop PC and to run S-PLUS Server on a backend server. Because Tomcat is simple to install on Windows and Unix platforms, and because it is easy to use, you might consider using it for testing this sample application and for developing and testing your own code, even if you plan to use a different Web container to deploy your code.

NOTE: Security and the Sample Application

The sample application has several pages intentionally designed to allow you to run your own JSP code and S-PLUS code. These pages are great for learning how the S+JSP module works and for testing out your code, but a malicious user might use these pages to access and damage your system. In addition, the sample application includes pages for administering S-PLUS connections, and a malicious user could use these to halt the sample application itself.

We recommend running the Web container for the sample application in a protected environment, for example on your desktop computer on a local area network accessible only to trusted users. If you make the sample application more widely available, for example on the World Wide Web, then you should restrict access to the insecure pages or remove them completely. Chapter 6 includes a section on security that summarizes the important issues, for the sample application and for your own application.

Installing S-PLUS Server for UNIX/Linux

1. Install S-PLUS Server according to the directions that come with that product.

2. Start the S-PLUS Server factory:

   ```
   SplusAS SERVER -factory [options]
   ```

   See the S-PLUS Server instructions if you need more details.

If you want the sample application to use more than one server to run its S-PLUS sessions, repeat the above instructions for each server on which you'll be running these sessions.

Installing the Web Container and Sample Application

1. On your Web server machine, install a Java Development Kit conformant to J2SDK 1.4 or later, if one is not installed there already. Java Development Kits for Solaris, Linux and Windows operating systems are available from `http://java.sun.com`.

   Create an environment variable named `JAVA_HOME`, whose value is the path to the new JDK directory.

   Add the bin subdirectory of the new JDK directory to your path.

2. Download Tomcat from `http://jakarta.apache.org/` and install it according to the instructions that come with it. NOTE: On Windows, make sure that the path to the directory in which Tomcat is installed does NOT have a space in the name, i.e. do not install to `C:\Program Files\Tomcat` since this path has a space in the name.

In the instructions that follow, `TOMCAT_HOME` refers to the top-level Tomcat directory.

3. Copy the file `sjsp.war` to `TOMCAT_HOME/webapps`.

4. Launch Tomcat. For Windows, Linux, or Unix, the appropriate batch file or script is in `TOMCAT_HOME/bin`. Tomcat will automatically unpack the file `sjsp.war` and set up the sample Web application.

5. (Optional) Edit the file

   ```
   TOMCAT_HOME/webapps/sjsp/WEB-INF/SplusConnections.xml
   ```

   to describe the S-PLUS session(s) you want to run. You can also launch these sessions using the administrative tools in the Web site. Sessions described in the `SplusConnections.xml` file automatically get launched each time you start the Web server, saving you the hassle of starting them by hand every time. The XML code for one session will look something like this:

   ```
   <connection name="Friendly Name">
         <host>computer</host>
         <username>account</username>
         <password>yourpassword</password>
   </connection>
   ```


Here *Friendly Name* is any name you want to use for the session, *computer* is the name or IP address of the computer running S-PLUS Server, *account* is the name of your account on the host computer, and *yourpassword* is your password. If you want the sample application to run multiple S-PLUS sessions, include a section of XML code like that above for each such session. The multiple sessions can be on the same computer or on different computers.

For more information on the `SplusConnections.xml` file, see Chapter 6.

Viewing the Application

To view the sample application, point your browser to

`http://host:8080/sjsp/`

where *host* is the name or IP address of the computer running the Web container. The "8080" in this URL is the port that Tomcat uses by default when it runs in standalone mode. If you reconfigure Tomcat or use another Web server, adjust the URL appropriately. Figure 2-1 shows the sample application.

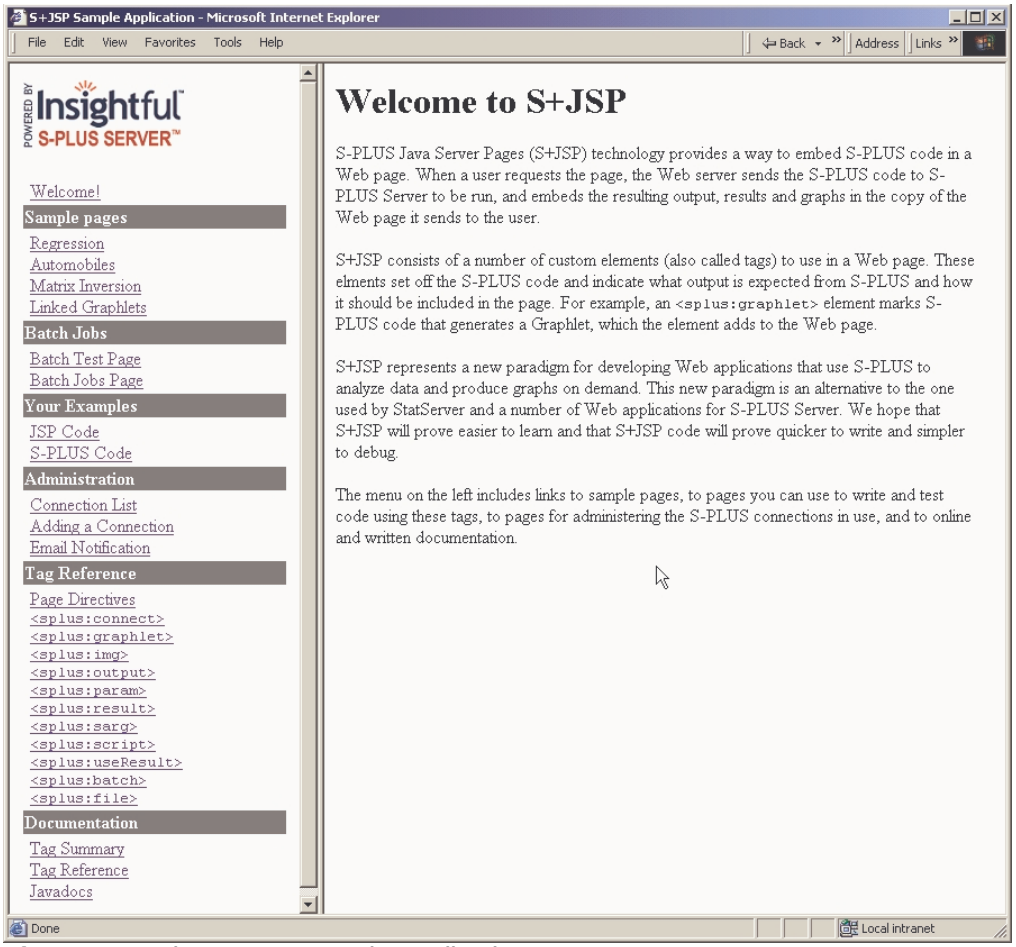

Figure 2-1. *The S+JSP Sample Application.*

Checking the S-PLUS Connections

For the sample application to work correctly, it must have working connections to one or more running S-PLUS sessions. To check, click the Connection List link in the Administration section of the frame on the left in the sample application.

If you edited the `SplusConnections.xml` file when you set up the Web container (see Step 5 of the Installing the Web Container and Sample Application section), and if everything is configured properly, the application will report active sessions as shown in Figure 2-2.

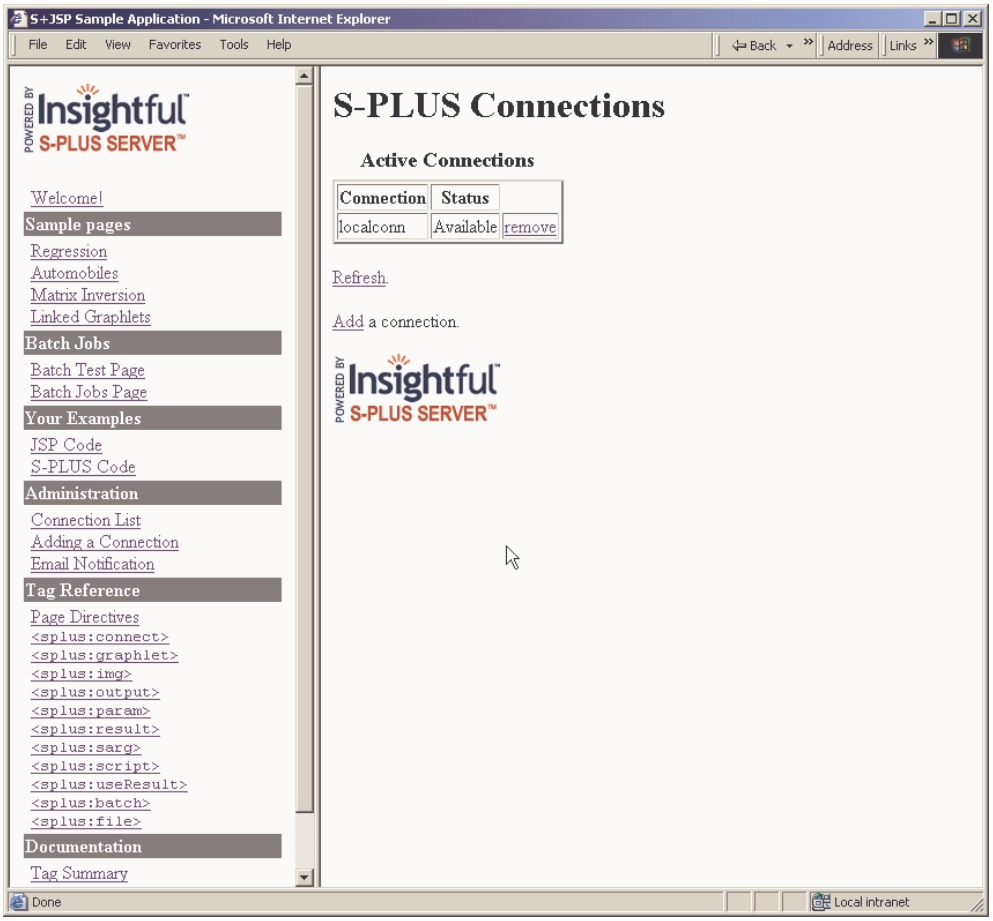

Figure 2-2. *A Connection List showing an active S-PLUS session.*

If you do not have any active connections, the application will show one or more inactive connections, or report that the connection pool is empty. In this case click on the Adding a Connection link to add a connection, using the page shown in Figure 2-3.

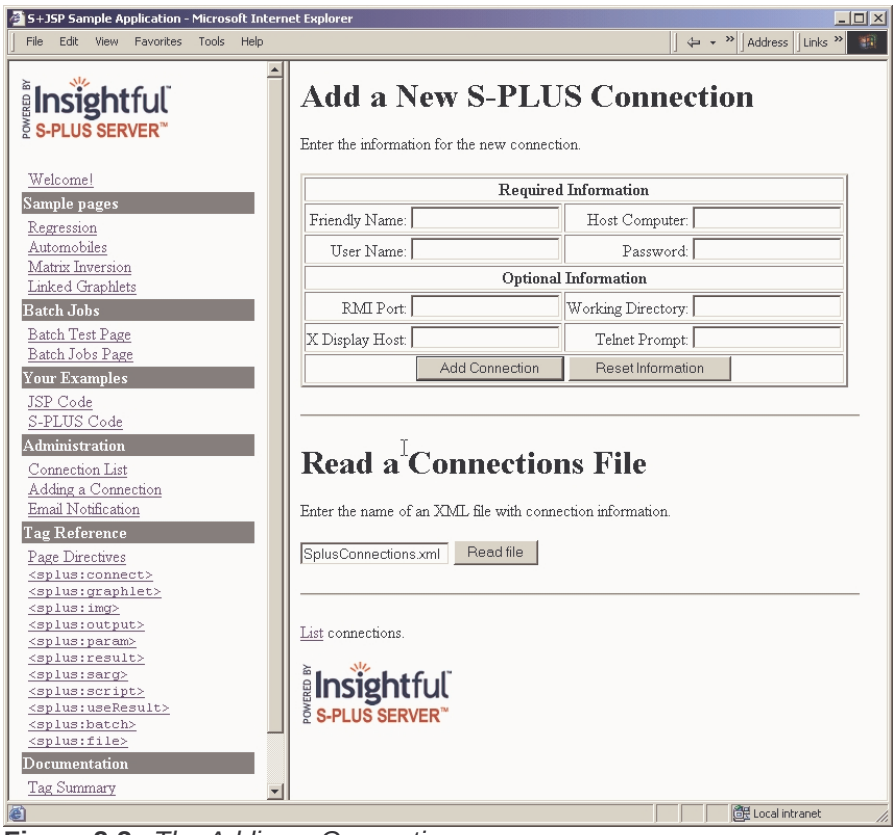

Figure 2-3. *The Adding a Connection page.*

To add a connection, fill out the form at the top of the page. Here `Friendly Name` is any name you want to use for the S-PLUS session, `Host Computer` is the name or IP address of the computer running S-PLUS Server, `User Name` is the name of your account on the host computer, and `Password` is your password. When you have filled out the required information, click the Add Connection button. The server will add the connection you specified and return with a list of the current connections, as shown in Figure 2-2.

If you want the sample application to run multiple S-PLUS sessions, add the additional connections in the same way.

The Adding a Connection page provides a handy way to add connections to a running application. However this page does not change the `SplusConnections.xml` file for the application, so any connections you add this way will be lost if you restart the Web container.

Chapter 6 gives more information about configuring the S-PLUS connections for your application.

Trying Out the Application

Once you've installed the sample application and established a connection to an S-PLUS session, go ahead and try out the various pages in the application.

The next chapter uses the sample application to give a tour of the custom tags for writing your own Web applications with this module.

Chapter 3: A Guided Tour of the Tag Library

In this Chapter we'll see how to write Web pages that include dynamically generated results and graphs from S-PLUS. Custom S-PLUS tags, similar to HTML tags, do nearly all of the work. The first section explains the tags and shows how to try out tags and other small samples of JSP code from within the sample application. The next two sections describe the commonly used tags in more detail.

Testing a Tag in the Sample Application

After installing the `sjsp` sample application according to the results in the previous Chapter and starting up your Web server, open the application by pointing your browser to this URL:

```
http://host:port/sjsp/
```

where *`host`* is the name of the computer running your Web server and *`port`* is the number of the port it is using. (By default, Tomcat uses port 8080. If your Web server is using port 80, you don't have to specify it in the URL.) This URL requests the page shown in Figure 3-1.

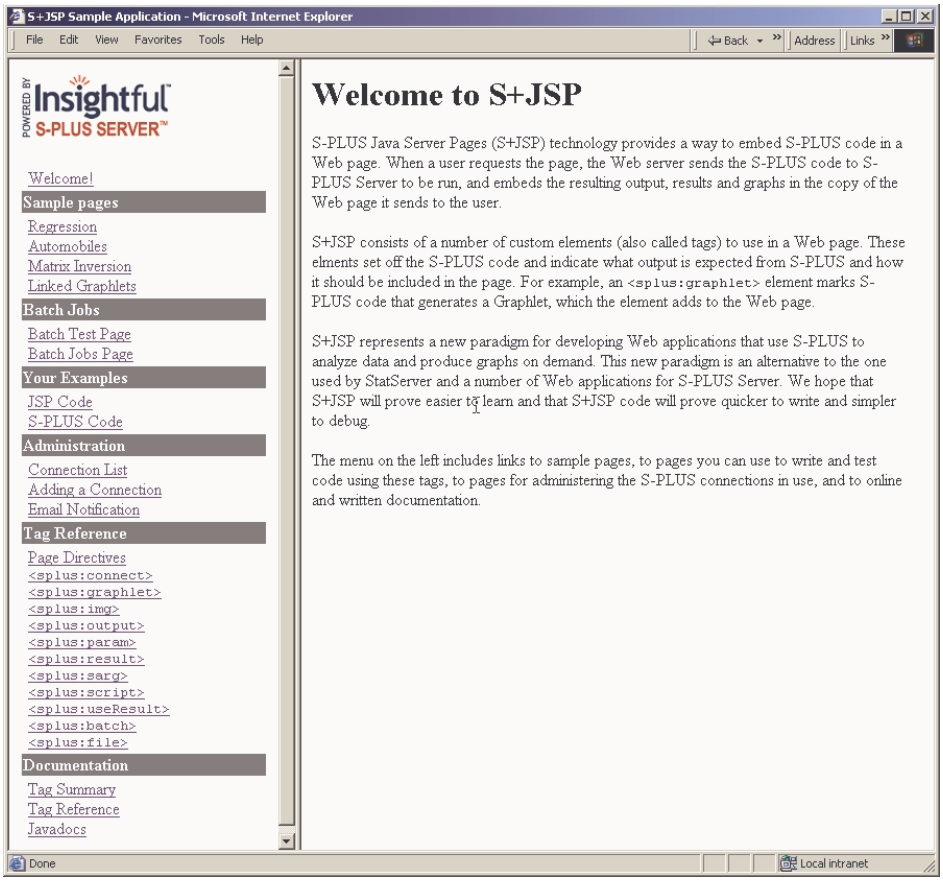

Figure 3-1: *The S+JSP Sample Application.*

In the Your Examples section of the menu on the left, click the JSP Code link to bring up the page shown in Figure 3-2 below.

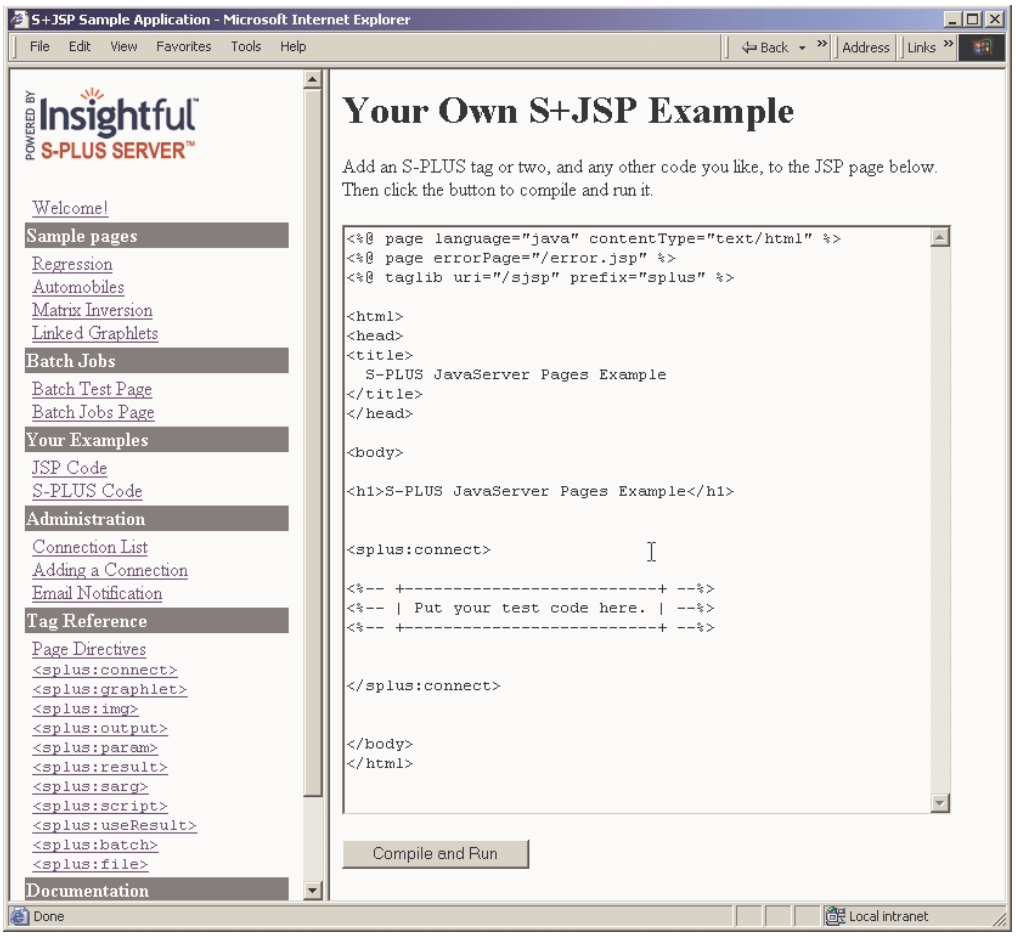

Figure 3-2: *A JSP page for testing small samples of code.*

The right frame contains a text window, which already contains code. This code is an example of a JavaServer Pages (JSP) Web page. JSP pages are the primary — and in many cases the only — components needed to construct a Web application with S-PLUS Server for UNIX/Linux.

Let's examine this JSP page in some detail. Most of the page consists of text and HTML that is sent to the user's browser in response to a request. The remaining elements are instructions that tell the Web container what to do when at request time. The top three lines on the page declare this page to be a JSP page, specify what page to run instead if this page contains an error, and declare that this page uses the S+JSP custom tag library to communicate with S-PLUS. The `<splus:connect>` tag tells the Web server to obtain a connection to an S-PLUS session, and the `</splus:connect>` tag tells the server to release the connection.

We'll now add an additional custom tag to make use of this connection. Between the two connect tags, add this line of code:

```
The value of pi is <splus:result expr="pi" />.
```

The entire page should now look something like page in Figure 3-3.

Figure 3-3. *Test code added to a sample JSP page.*

When a user requests this page, the `<splus:result expr="pi" />` tag tells the server to send the expression pi to S-PLUS, and to add the result to the HTML page before sending it to the user requesting it.

Now click the Compile and Run button. This button simulates what would happen if the page with your edits was part of a Web application, and if you requested the page. The system displays the results in a separate window, as shown below.

Figure 3-4. *Output from the sample JSP page.*

Note that the custom tag has been replaced with the value 3.141592653589793, calculated by S-PLUS at request time. If you view the page source you'll see that all of the HTML on the

original page was sent to your browser, but none of the JSP code was sent, except that the `<splus:result>` tag was replaced by the numerical value.

The JSP Code page in the sample application (shown in Figure 3-2) provides a convenient way to test small samples of JSP code. We'll continue to use this page to look at more custom S-PLUS tags as well as some coding suggestions. This page, however, should not be used for developing large applications. Later we'll discuss setting up a development environment for S+JSP applications.

The next two sections give a tour of the most commonly used custom tags. More information on these tags is in the Tag Reference section of the `sjsp` Sample Application. For example, you can click on the `<splus:img>` link to bring up the page shown below. The Documentation section of the Sample Application gives links to additional documentation, namely the Tag summary and Tag reference.

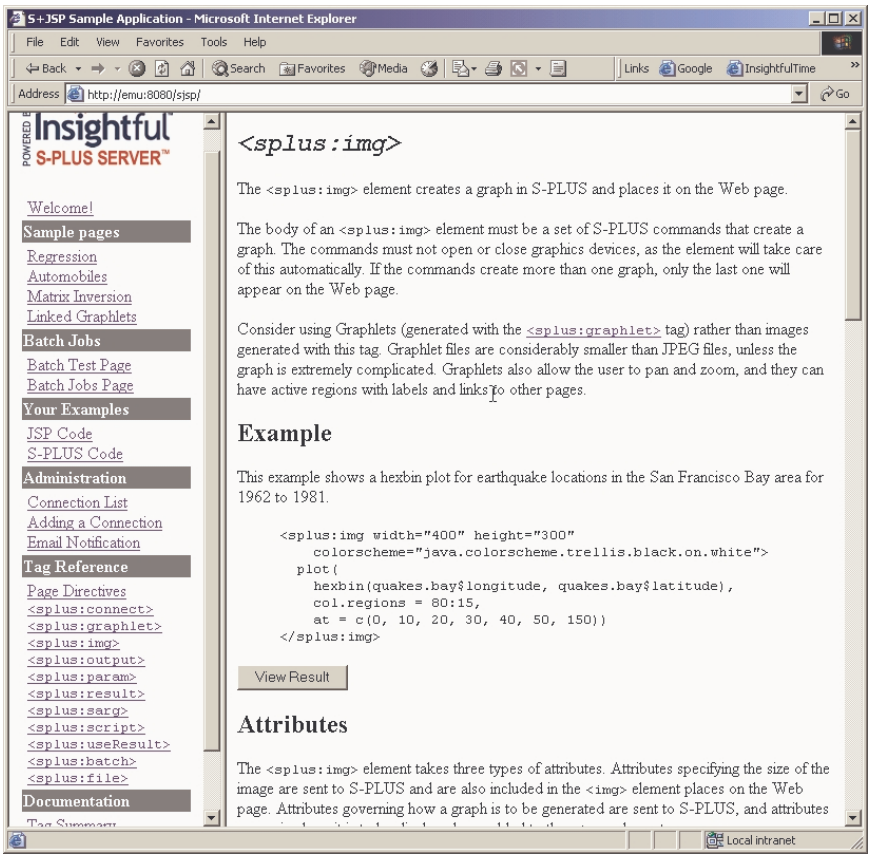

Figure 3-5. *Online documentation for the `<splus:img>` tag.*

S-PLUS Graphics

This section introduces custom tags for graphs. The next section describes tags for text and numerical output from S-PLUS.

Our first graphical example displays a JPEG with a three-dimensional graph of velocity versus position for a spiral galaxy. Click the JSP Code link in the left frame of the browser to get a fresh

copy of the JSP page, as shown in Figure 3-2. Inside the `<splus:connect>` element on that page, insert the code shown below.

```
<splus:img height="440" width="550">
  print(cloud(velocity ~ east.west * north.south,
    data = galaxy))
</splus:img>
```

After inserting this code, your page should look something like Figure 3-6.

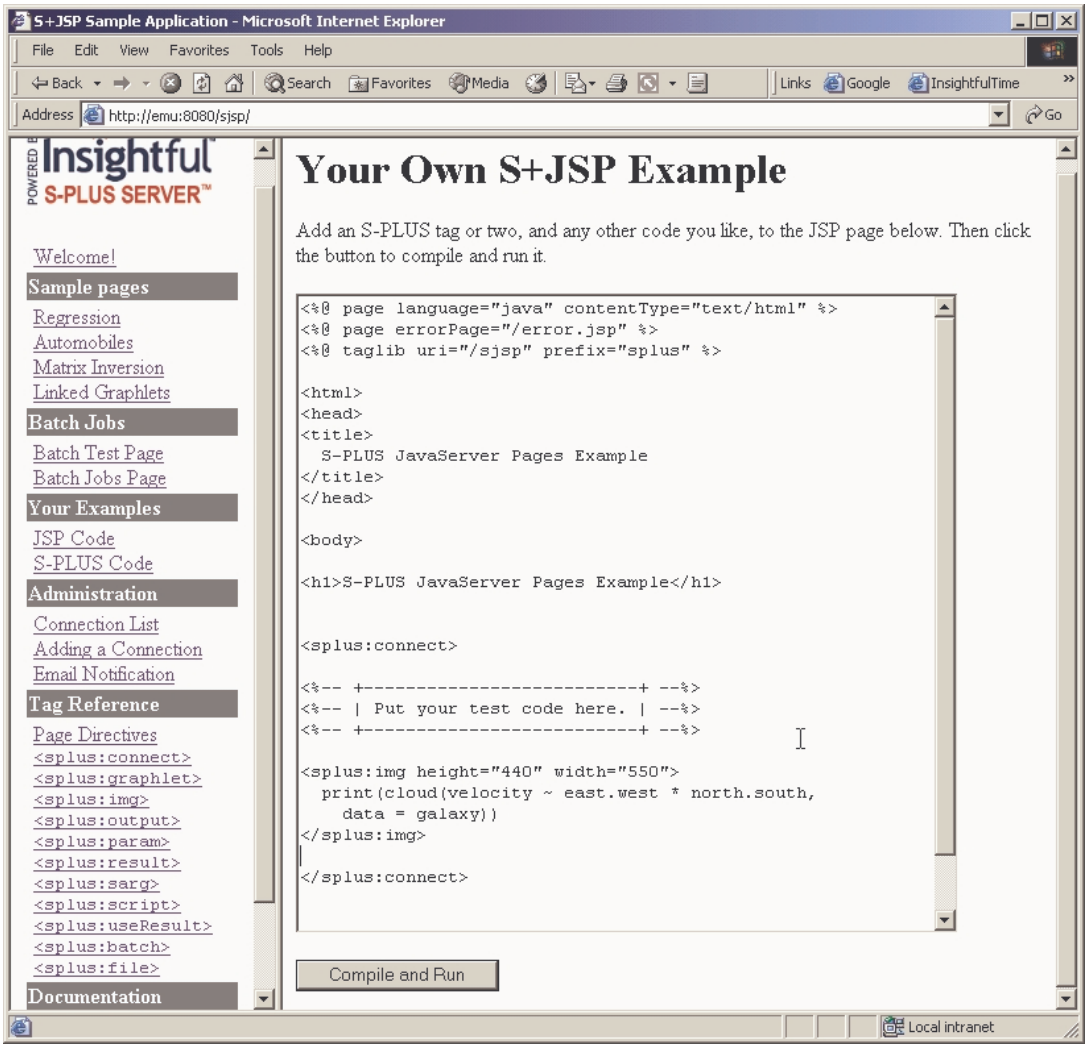

Figure 3-6. *JSP code for the JPEG example.*

The `<splus:img>` tag tells S-PLUS to generate a graph, and specifies the size. The code between the `<splus:img>` and `<splus:img>` tags is S-PLUS code that plots the graph. When you Compile and Run the page, the output looks something like the page in Figure 3-7.

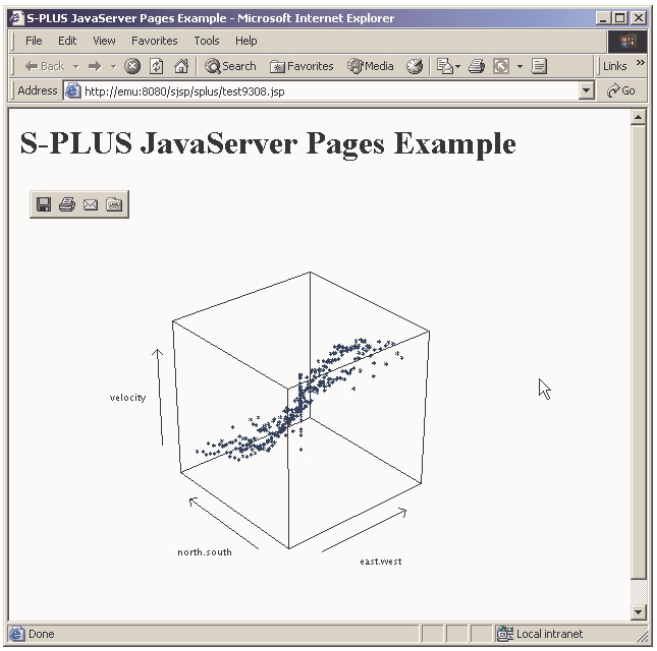

Figure 3-7. *Output from an* `<splus:img>` *tag.*

At request time, the code for the `<splus:img>` element calls S-PLUS to generate the requested graph, which it stores in a JPEG file on the Web server. The element then adds an HTML `<img>` tag to the Web page sent to the user. The `<img>` tag causes the Web page to load the JPEG file. For more information, view the HTML code sent to your browser for this page and examine the `<img>` element. The `<splus:img>` element takes care of all of these details for you, so you can concentrate on the overall design of your Web page.

Next, we'll generate this graph as a Graphlet. A Graphlet is an interactive graph generated by S-PLUS and viewed on a Web page. We can create a Graphlet rather than a JPEG by simply changing the `<splus:img>` element to an `<splus:graphlet>` element. In your JSP code, make the changes shown below in bold:

```
<splus:graphlet height="440" width="550">
  print(cloud(velocity ~ east.west * north.south,
    data = galaxy))
</splus:graphlet>
```

Figure 3-8 shows the graph as a Graphlet.

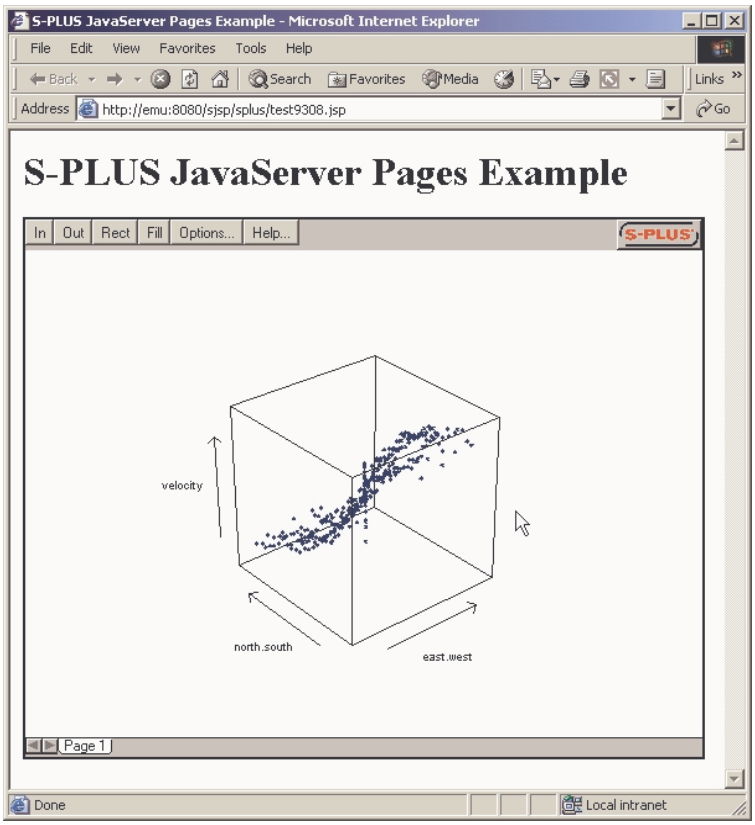

Figure 3-8. *Output from an <splus:graphlet> tag.*

The `<splus:graphlet>` element calls S-PLUS to generate the requested graph, which it stores in a Graphlet file on the Web server. The element then adds and HTML `<applet>` tag to the Web page sent to the user. The `<applet>` tag causes the Web page to load a Java applet, which in turn loads and displays the Graphlet file. For more information, view the HTML code sent to your browser for this page and examine the `<applet>` element. The `<splus:graphlet>` element takes care of all of these details for you, so you can concentrate on the overall design of your Web page.

Notice the Page 1 tab in the lower left of the Graphlet. Such page tabs are useful in multi-page Graphlets, but we don't need this tab for this single-page Graphlet. Let's edit the code to remove the tab. To do so, add the line of code shown below in boldface to the code in your JSP page.

```
<splus:graphlet height="440" width="550">
  <splus:param name="spjgraph.tabs" value="off" />
  print(cloud(velocity ~ east.west * north.south,
    data = galaxy))
</splus:graphlet>
```

Then compile and run the page, and notice that the tab is no longer present.

The `<splus:param>` element, which we used to remove the Page 1 tab, sets parameters for the way the applet displays a Graphlet. For more information, see the documentation for the `<splus:param>` element, or read the Graphlets chapter in the *S-PLUS Server Programmer's Guide*.

Output and Results from S-PLUS

Now we'll examine tags for obtaining text and numerical output from S-PLUS.

For these examples we'll use the fuel.frame data set built into S-PLUS. We'll perform a *t*-test to compare gas mileage for large and medium-size cars, and look at several ways to format the output.

Once again click the JSP Code link in the menu of the sample application to get a fresh JSP page, as shown in Figure 3-2. Add the following code inside the `<splus:connect>` element.

```
<splus:script>
  tAns <- t.test(
    fuel.frame$Mileage[fuel.frame$Type == "Large"],
    fuel.frame$Mileage[fuel.frame$Type == "Medium"],
    alternative = "less")
</splus:script>

<pre>
<splus:output>
  print(tAns)
</splus:output>
</pre>
```

The `<splus:script>` element executes commands in S-PLUS, but it adds nothing to the HTML page sent to the user. In this case we use an `<splus:script>` element to perform the *t*-test and to assign the results to a variable named `tAns`. In this and the next several examples we'll explore ways to output information contained in the S-PLUS object `tAns`. Make sure you keep this `<splus:script>` element on your page for the rest of the examples in this section.

The `<splus:output>` element displays the text output that a user would see running the same code in a command-line version of S-PLUS. In this example, printing `tAns` creates a text report of the *t*-test results. When you compile and run the page you will see the output shown in Figure 3-9.

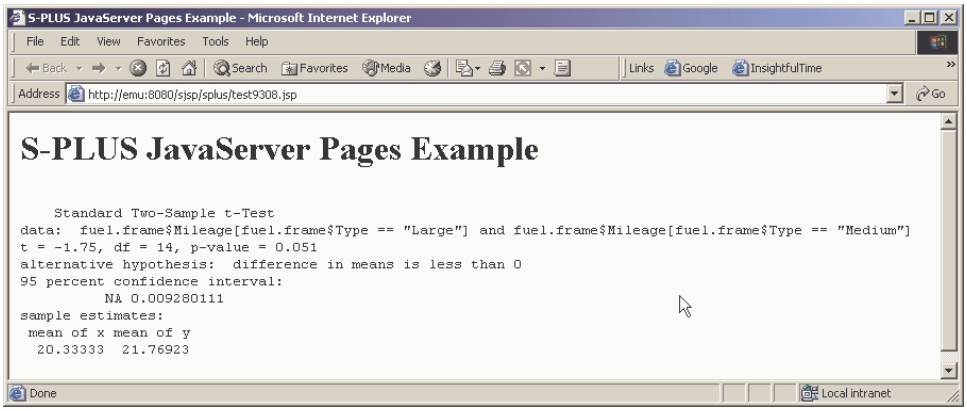

Figure 3-9. *Output from an `<splus:output>` tag.*

Notice that we put the `<splus:output>` element inside of an HTML `<pre>` element. The `<pre>` element preserves the text formatting; without it, the user's browser would reformat the

lines and paragraphs of the report, making it much harder to read. Try removing the `<pre>` and `</pre>` tags and look at the output you get.

The t-test report looks nice, but suppose we want to format our own report. The `<splus:result>` tag returns the number, text or vector returned by an S-PLUS expression. We'll start by using this tag to report the p-value to the user. On the JSP Code page, replace the code

```
<pre>
<splus:output>
  print(tAns)
</splus:output>
</pre>
```

with the code

```
The p-value is <splus:result expr="tAns$p.value" />.
```

Compile and run the page to get the result shown below.

> The p-value is 0.050996532236794956.

Next let's draw a conclusion from the *t*-test. If the p-value is less than 5% we'll conclude that the medium size cars get better gas mileage; otherwise we'll conclude that the mileage is the same. Add this code to the JSP page:

```
<h2>Conclusion</h2>

<splus:useResult expr="tAns$p.value < 0.05"
  id="bReject" className="boolean[]" />

With 95% confidence we conclude that
<% if (bReject[0]) { %>
  medium cars have better gas mileage than large cars.
<% } else { %>
  medium cars do not have better gas mileage
  than large cars.
<% } %>
```

Like the `<splus:result>` element, `<splus:useResult>` element obtains a result from S-PLUS, but rather than adding it directly to the output page it assigns it to a Java variable. In this example the Java variable is named `bReject`. Java code inside the `<%...%>` elements uses this variable. The Java code in this example is simple if-then code to determine which conclusion to print.

When you compile and run this code, you get the output shown in Figure 3-10.

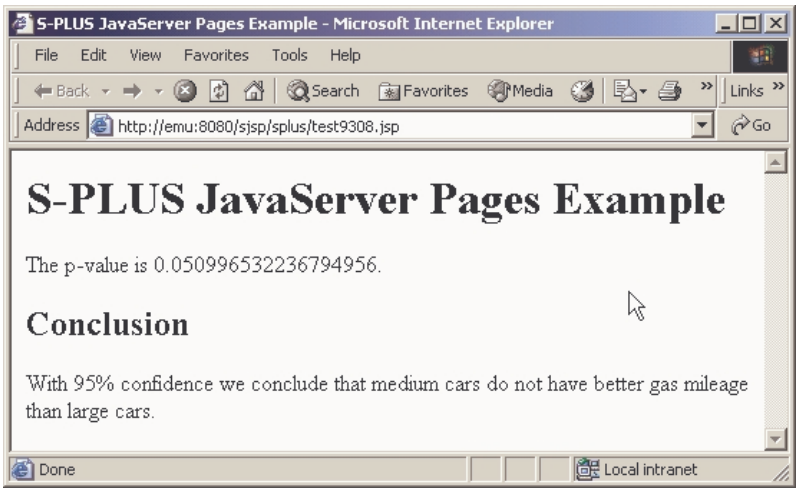

Figure 3-10. *Output from a page using the* `<splus:result>` *and* `<splus:useResult>` *tags.*

We end this tour with one more `<splus:result>` example. The `<splus:result>` tag can display tables created by the S-PLUS function `html.table`. For example, we can display all the data in the `tAns` object in an HTML table by using the code

```
<splus:result>
  html.table(unlist(tAns))
</splus:result>
```

The table is shown in Figure 3-11.

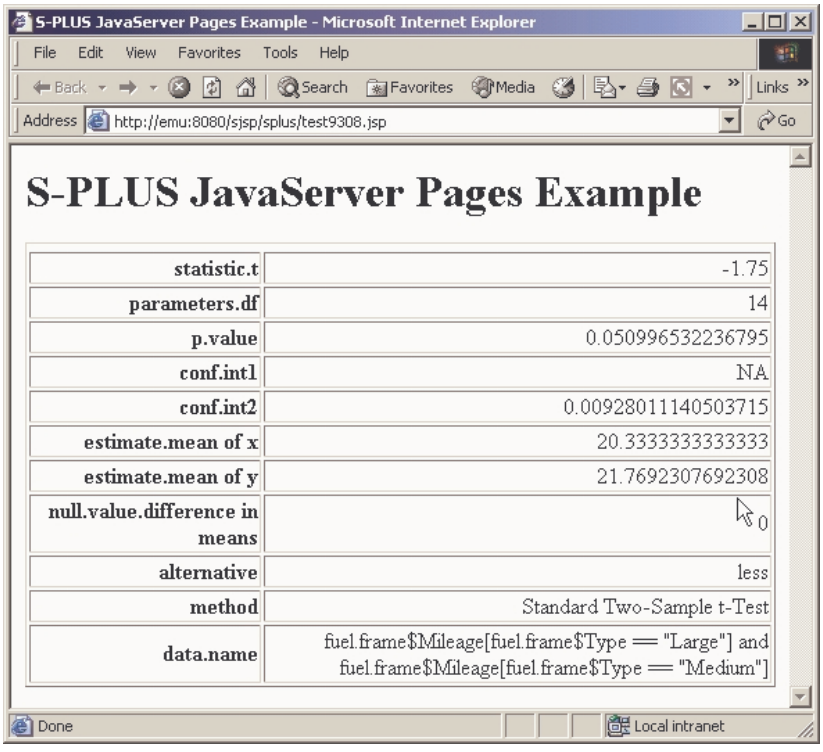

Figure 3-11. *Output from a page using the* `html.table` *function and the* `<splus:result>` *tag.*

Chapter 4: Authoring Your Own Applications

In the previous chapter we used the JSP Code page of the sample application to test a number of S-PLUS tags. This JSP Code page provides a handy way to test small samples of code, but should not be used as a development environment. This chapter discusses authoring Web applications that use the S+JSP module.

Modifying the Sample Application

You can quickly create small Web applications by modifying the sample application. Simply remove all the JSP and HTML pages that you don't want, and replace them with the new ones that you write.

If you add a JSP page to a running Web application, or edit a JSP page already in the application, the Web container will recompile the page the next time it is requested. This Web container feature is very handy during development. You can modify the sample application one page at a time and check your work as you go, all without restarting the Web container.

When your application is complete you can distribute and deploy it in a standard archive format known as a WAR file. A later section of this chapter describes WAR files.

Application Components

For a larger Web application, simply modifying the sample application is a rather haphazard way to proceed. You will probably prefer to construct a directory tree with your source code, back up the directory tree in a source control system, and use a build utility to assemble the application. This section discusses the components of an S+JSP Web application, the components you will need to include in your directory tree and to assemble with the build utility in order for the application to work. The next section makes some suggestions regarding this approach.

Figure 4-1 shows the main components of an S+JSP Web application. The directory tree shows the deployment structure for the application. You might use a somewhat different directory structure during development, and use a build tool to assemble the pieces in the structure shown in the figure for deployment.

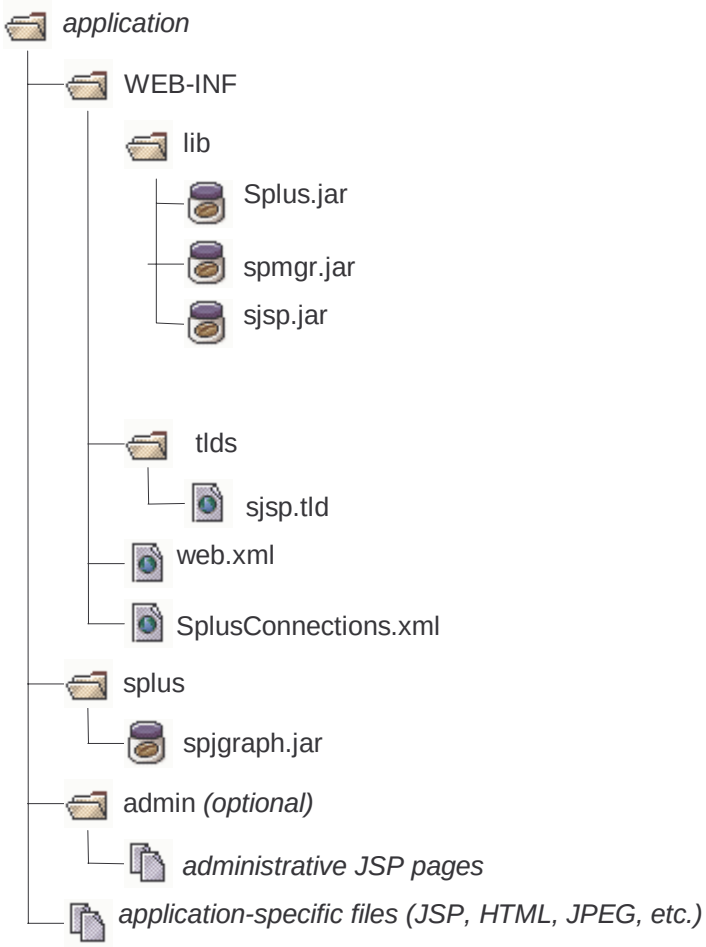

Figure 4-1. *Components of an S+JSP Web application, arranged for deployment.*

Java Code

The WEB-INF/lib directory contains the Java code for the S+JSP module, compressed into JAR files. You must include these files:

- `Splus.jar`: Code to access S-PLUS Server. All S-PLUS Server clients use the code in this jar file.
- `spmgr.jar`: Code to manage S-PLUS sessions.
- `sjsp.jar`: Tag library code.

Tag Library Descriptor File

In the `WEB-INF/tlds` directory include one file:

- `sjsp.tld`: The descriptor file for the S+JSP custom tags.

In some cases you can omit this file and use the descriptor information in the `sjsp.jar` file directly. See Chapter 6 for more information.

Configuration Files

In the `WEB-INF` directory include these two configuration files:

- `web.xml`: The standard configuration file for JSP/servlet Web applications.
- `SplusConnections.xml`: A file specifying the number and locations of S-PLUS sessions to run.

Chapter 6 discusses these two files in detail.

The splus directory

The `splus` directory serves as a storage area for graph files generated by S-PLUS to be included on Web pages. The S+JSP module periodically deletes old files in this directory.

If you use Graphlets in your application, you must create this directory in your application and include in it a copy of `spjgraph.jar`, the applet that displays Graphlets. If you do not use Graphlets, you do not need the `spjgraph.jar` file, and the module will construct this directory automatically if you do not create it.

You can use a different name for this directory if you specify it in the `web.xml` configuration file. Chapter 6 describes the details.

Administrative pages

Consider including a copy of the administrative pages from the sample application in your application. These pages provide a way to add and remove connections to S-PLUS sessions while your application is running.

If you include these administrative pages, however, and if your application is available beyond a trusted core audience, you will need to control access to these pages for security reasons. See the documentation for your Web container to learn how to control access to certain pages.

Application-specific pages

The rest of the application consists of the pages you write for your application. These can be in the top-level directory for your application, or they can be organized into a directory tree appropriate for the structure of the application.

Additional suggestions

Authoring JSP pages

You can write JSP pages in a text editor or in a JSP-aware Web page editor.

The file `blank.jsp`, included in the top-level directory of the sample application, provides a template for writing your own JSP pages. Simply copy the file, rename the copy, and add your own code.

WAR files

A WAR (Web ARchive) file is a zipped file containing all the pages and components of a Web application. Distributing your application is easy; you simply provide the WAR file. Most Web containers understand WAR files and can unpack and install them. Tomcat, for example, automatically installs any new WAR files it finds in its `webapps` directory at startup time.

If you create your application in the Web container, you can create a WAR file using the jar command. For example, if you have modified the `sjsp` sample application, and you want to archive it as a new application named `myapp`, use commands similar to the following:

```
cd TOMCAT_HOME/webapps/sjsp
jar cf ../myapp.war *
```

If you create a development directory with all the source files for your application, you can use a build utility to assemble everything into a WAR file. One such utility especially suited for Web applications is ant, which is available for free from Apache's Jakarta project (`http://jakarta.apache.org/ant/`).

Chapter 5: S-PLUS Code in Web Applications

All the sample pages in the main sample application for this module use only built-in S-PLUS functions. S-PLUS provides *thousands of* thoroughly tested functions providing expert statistical analysis and powerful graphics, and for many Web applications these will suffice. However S-PLUS is also a programming language, and many Web applications require custom S-PLUS functions. This chapter discusses how to write, troubleshoot and deploy such functions for a Web application.

Writing Custom S-PLUS Functions for a Web Application

Use a standalone S-PLUS session, not a Web application, to write your custom S-PLUS code and to test it as much as possible. You can use the S-PLUS Java GUI with S-PLUS Server, or a desktop version of S-PLUS if you have one.

We recommend keeping your code in source form in text files outside of S-PLUS. This is the easiest form for a human to read and the best form to store in a source control system. Text files containing S-PLUS source code typically have the extension .q in Unix and .ssc in Windows. The script window in the Windows S-PLUS product is especially convenient for editing source code files and reading them into S-PLUS for testing. With other versions of S-PLUS you can edit your code in a text editor and use the S-PLUS source function to read it in.

Here are some suggestions for writing custom S-PLUS functions for Web applications.

- Each function can return a value, generate a graphic, or output a report (using the cat command, for example), but it should do only one of these things. Otherwise some of the work of the function will be lost, as the tag that calls it will be able to handle only one type of result or side effect. If your function performs a long analysis that produces several results, graphs and/or reports, design the function to return a list of them (or an S-PLUS object that contains them). Then a JSP page can perform the analysis once, save the list as a variable in S-PLUS, and use several tags to get the results, graphs, and reports to format on the output page.

- Avoid opening, closing, and otherwise manipulating graphics devices in your code. The tags that display the graphs will take care of the devices for you. If you need explicit graphics device code to test your function in the standalone S-PLUS session, put that code in a short wrapper function that calls the main function you are writing.

- Where possible, avoid using the `java.identify` command in your custom functions. Instead place the `java.identify` command in the tag right after the call to your custom function. This way you avoid hard-coding page links within the S-PLUS code. Names of linked pages appear only within the tags, so all editing of the structure of the application can be done in the JSP pages.

  In some cases, separating a `java.identify` command from the rest of the code in a function is difficult or impossible to do. In such cases, consider passing in as function arguments the names of the pages to which you're providing links. This is another way to avoid hard coding the page names in the S-PLUS code.

Deploying S-PLUS Functions

The custom S-PLUS functions are best stored in a separate directory attached in position 2 or lower in the search path.

To install the S-PLUS code for the select Web application in a production system, follow these steps.

1. Create a directory for the custom S-PLUS code. Any directory accessible from the S-PLUS sessions is fine.

2. Copy the file functions (lets say they are in `myfuns.q`) `myfuns.q` to this directory. Then use the following commands to create an S-PLUS chapter and install the functions in `myfuns.q`.

   ```
   SplusAS CHAPTER
   SplusAS make install.funs
   ```

3. In the top-level working directory for each S-PLUS session you use, add this `.First` function:

   ```
   .First <- function()
   {
         attach(dir)
   }
   ```

   where *`dir`* is the directory you created in Step 1.

4. If you run S-PLUS sessions on multiple machines, repeat Steps 1 through 3 for each machine. Alternatively, create the directory in Step 1 on a drive that is mounted on all the machines running S-PLUS sessions.

Chapter 6: Deployment and Administration

This chapter describes how to configure a Web application that uses the S+JSP module. The module reads from two configuration files at startup time. One of these, named `web.xml`, is the standard configuration file for all JSP and servlet Web applications. The module checks the `web.xml` file for some configuration parameters, and uses default values for any parameters not specified in that file. The other file, `SplusConnections.xml`, is specific to S+JSP applications and specifies the S-PLUS connections that the module must initiate and administer. For running batch jobs, there is a separate connections file called `SplusBatch.xml` that specifies the connections to use when running a batch process. Having separate connections files allows one to specify separate users and different machines to run longer processes.

Connections Configuration File: SplusConnections.xml and SplusBatch.xml

The connections configuration file, typically named `SplusConnections.xml`, lists the S-PLUS connections that the system will attempt to initiate at startup time. Listing 6-1 shows a sample of such a file.

```
<?xml version="1.0" encoding="ISO8859_1" standalone="yes"?>
<splus-connections>

      <connection name="bubble 1">
            <host>bubble</host>
            <username>splusweb</username>
            <password>d0ntTell</password>
      </connection>

<splus-connections>
```

Listing 6-1. *A sample `SplusConnections.xml` file.*

The sample in Listing 6-1 describes a single connection specified by the `<connection>` element. The friendly name for this connection is “bubble 1”. The S+JSP module uses the friendly name to identify the session, for example in log entries. The `<connection>` element has three required child elements:

`<host>`	Name or IP address of the host computer on which to run the S-PLUS session. The host machine, which must be running S-PLUS Server, can be the same machine as the one running the Web container, or it can be another machine accessible over a network.
`<username>`	Name of the account under which to run the S-PLUS session. During development this will often be the Web author’s account. For production consider creating a new account representing the Web application that will be running the S-PLUS sessions.
`<password>`	Password for the account under which the S-PLUS session will run.

The `<connection>` element also accepts four optional child elements:

`<rmiport>`	RMI port for S-PLUS Server on the host machine. This value defaults to 1099.

<cwd>	Directory to use as the S-PLUS working directory. If this element is missing, S-PLUS uses its default working directory, typically ~/MySwork.
<display>	X-server display string. S-PLUS needs access to an X-server to produce JPEG and PNG graphics files. Typical values for this element are :0.0 and :1.0, to access an X-server or X11 virtual frame buffer on the host computer. X has a program called Xvfb for X-virtual frame buffer that can be run on the server and be accessed to create static graphics.
<prompt>	Prompt string for telnet login.

The connections file can contain multiple <connection> elements, one for each S-PLUS session that the application should initiate at startup time. The S-PLUS sessions can all be on the same machine, or they can be on multiple machines on a network. In general you should not run more than one or perhaps two S-PLUS sessions per CPU; more sessions will only decrease efficiency as they compete for CPU time.

For security reasons, the connections file must be in the Web application's WEB-INF directory. By default, the module will look for a file named SplusConnections.xml. If you want to use a different name for this file you can specify the name in the web.xml file for the application; see the next section for details.

While a Web application is running an administrator can add and remove S-PLUS sessions, for example by using administrative JSP pages. Changes made at run time, however, are not automatically reflected in the connections file. If you restart the Web container, the application will revert to the original configuration of S-PLUS sessions, unless you edit the connections file to change this configuration.

JSP Configuration File: web.xml

The web.xml file is the standard configuration file for JSP (and Java servlet) Web applications. This section describes elements of the web.xml file for configuring an S+JSP application.

A sample web.xml file is included with the S+JSP module. Having a copy of this sample available to look at might be helpful as you read this section. You can also edit it for use with your application, or cut and paste sections of it into your configuration file.

Tag library

The code below tells the Web container where to find the file sjsp.tld, the descriptor file for the S+JSP tag library.

```
<taglib>
  <taglib-uri>/sjsp</taglib-uri>
  <taglib-location>/WEB-INF/tlds/sjsp.tld</taglib-location>
</taglib>
```

You should add this code to the web.xml file for your applications. With this declaration in the web.xml file, the Web container can read the page directive


```
<%@ taglib uri="/sjsp" prefix="splus" %>
```

at the top of a JSP page and find the tag library code to run the S-PLUS tags on the page.

The name of the `tlds` directory is conventional but not required. You can put the file `sjsp.tld` elsewhere (within the `WEB-INF` subtree) and adjust the code in your `web.xml` file to point to it.

According to the JSP Specification, a Web container should be able to use tag library descriptor information that is inside a jar file, in this case the file `sjsp.jar`. However some of the popular Web containers we tested do not appear to support this feature at the time of this writing. If your container supports this feature, you can omit the file `WEB-INF/tlds/sjsp.tld` from your application and use the code shown below in your `web.xml` file.

```
<taglib>
  <taglib-uri>/sjsp</taglib-uri>
  <taglib-location>/WEB-INF/lib/sjsp.jar</taglib-location>
</taglib>
```

Initialization and shutdown servlet

A servlet is a particular type of Java class that runs in a Web container. The S+JSP initialization and shutdown servlet, included in this module, is primarily useful for cleanly shutting everything down when the Web container shuts down. It can also initialize the application when the Web container starts up, by attempting to establish the specified S-PLUS sessions and by starting the maintenance and cleanup threads. The initialization is optional, however. If the servlet is not configured for initialization, the first request that accesses S-PLUS automatically initializes everything. In the sample applications in this module the servlets are configured not to perform any initialization, so that you can edit the `SplusConnection.xml` files before they get run.

For the servlet to run you need to declare it in the `web.xml` file, and specify that it be loaded at startup time. The following code does the trick.

```
<servlet>
  <servlet-name>
    SaspInitServlet
  </servlet-name>
  <servlet-class>
    com.insightful.webapp.taglib.SaspInitServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```

The code above is recommended but not required. Initialization is optional anyway, and most applications do not need to be shut down cleanly.

You can enable or disable the servlet initialization feature with an application-level context parameter named `splus-start-managers`, as shown in the sample below.

```
<context-param>
  <param-name>splus-start-managers</param-name>
  <param-value>true</param-value>
</context-param>
```


A parameter value of `true` or `yes` enables initialization. If the parameter is missing or has any other value, the servlet performs no initialization.

Configuring S-PLUS Persistence in web.xml

The S+JSP module can automatically manage persistent information within S-PLUS. In fact, the module offers several levels of support for persistence. An administrator can set the appropriate level of support by specifying a parameter in the `web.xml` file. Before considering the module's solution, however, let's look at the problem it solves.

Persistence in S-PLUS

S-PLUS can store information, for example as a result of an assignment statement like

```
x <- 7
```

Many Web applications need to store information in S-PLUS this way. An application might save the results from one analysis for use in subsequent analyses. Or it might allow a user to upload data and then perform a sequence of analyses on it. The information needs to be stored on a per-user basis; each person uploading data, for example, should see the all the data he uploaded, and no data anyone else uploaded.

Many other Web applications do not need persistence. An application might provide reports and graphs based on data pulled from a database, and never need to store intermediate results. The sample application described in Chapters 2 and 3 does not need persistence, for example. Such applications should not incur the extra overhead of providing persistence when they don't need it.

In a Web application, users share a pool of S-PLUS sessions. Without careful management of persistent information, unwanted results can occur. For example, one user might assign a value to *x* (by calling a page with a tag on it that makes the assignment). Later, when the user wants to use that value, she might get a connection to a different S-PLUS session, one with a different value for *x* or even no value at all. Even if she gets the same S-PLUS session, someone else might have used it and changed the value of *x*.

Regardless of how the module is configured, persistence is always guaranteed for the time that a JSP page holds an S-PLUS connection. For example, the following code will always work.

```
<splus:connect>

<splus:script>
  x <- 7
</splus:script>

The value of x is <splus:result expr="x" />.

</splus:connect>
```

In this case the `<splus:script>` element that sets the value of *x* and the `<splus:result>` element that uses the value both make calls to the same S-PLUS session, and no other requests can access that session in between the calls. If the `<splus:result>` element is on another JSP page, however, then special persistence management is necessary to make the code work.

The S+JSP module manages persistence by managing S-PLUS's working directories. It can automatically attach a user's working directory before providing an S-PLUS session to a JSP page requested by that user, and it automatically detaches any such directory when the JSP page returns the connection to the pool.

Levels of persistence

The S+JSP module currently offers two levels of persistence support: no support, and session-level support. In the next section we'll see below how to set the level of support for an application.

A support level of no support is appropriate for applications that do not need persistence in S-PLUS, and it incurs no overhead attaching and detaching directories.

With session-level support the module automatically manages working directories on a session basis. A "session" is the largest group of requests that the Web container can recognize as coming from a single user. If a user switches to a different computer, he begins a new session. Depending on how the Web container is configured, it will typically time out inactive sessions; the default time out period is typically 30 minutes.

For session-level persistence support, the S+JSP module creates a new working directory for a user the first time she makes a request in a given session. The module automatically attaches the directory every time it provides an S-PLUS session to a JSP page that the user has called, and detaches the directory when the JSP page returns the session to the pool. When the session ends, the module deletes the working directory.

Some Web applications will need an additional type of support for S-PLUS persistence, namely persistence at the user level. This level of support would ensure that a user gets the same working directory every session. It will require writing a new CWDManager class that integrates with the application's login code. The S+JSP module currently does not provide this level of persistence.

The BatchCWDManager will manage the working directories for batch processes if a working directory is specified in the batch tag. The BatchCWDManager differs from the session persistence in that the directory used by the batch process is kept around and not removed when the session is over. This allows a user to return to the batch analysis and continue with another analysis or view results that may not present well in an e-mail. The application developer can determine how the user returns to the batch S-PLUS directory.

S-PLUS can store data in places other than the working directory. For example, S-PLUS can assign data to other directories in the search path. The S+JSP module does not manage persistence in any such cases. In almost all cases Web authors — not end users — write the S-PLUS code for a Web application, and they should take care to store data only in the top-level working directory.

Setting the persistence level

You can specify the persistence level for an application by defining a context parameter named `splus-CWD-policy` in the `web.xml` file, as shown below.

```
<context-param>
  <param-name>splus-CWD-policy</param-name>
  <param-value>null</param-value>
```



```
</context-param>
```

A value of `null` for this parameter causes the S+JSP module to perform no persistence management. A value of `session` causes the module to provide session-level support.

In addition to the values `null` and `session`, the `splus-CWD-policy` parameter can be set to the name of a Java class derived from `com.insightful.webapp.spcon.CWDManager`. This feature allows for the addition of custom-written CWD managers in the future.

For session-level support, another context parameter named `splus-CWD-parent-dir` specifies where the session-specific directories are created. The example code below sets this directory to `/tmp`.

```
<context-param>
  <param-name>splus-CWD-parent-dir</param-name>
  <param-value>/tmp</param-value>
</context-param>
```

The value `splus-CWD-parent-dir` can be specified as an absolute path, or as a relative path from the original working directory that the S-PLUS sessions use. If you don't provide a value, the code will use the original working directory specified in `SplusConnections.xml` itself as the parent directory. Using the original working directory or a path relative based on that directory works only if all S-PLUS sessions share the same original working directory. The S-PLUS session is run as the user specified in `SplusConnections.xml` so one needs to make sure that the user listed in `SplusConnections.xml` has complete access to the directory listed in `splus-CWD-parent-dir`.

If an application runs S-PLUS sessions on multiple computers and implements session-level persistence, the value of the `splus-CWD-parent-dir` must be a directory on a drive that is cross mounted to all the computers running the S-PLUS sessions.

Other connections parameters in web.xml

The S-PLUS connection management code can read several additional configuration parameters from the `web.xml` file. These parameters are described below. All of these parameters are optional; you don't have to include them in the `web.xml` file. The default values for these parameters work fine for most applications. You might want to change these parameters as you fine tune your application.

`splus-connections-file`

A context parameter named `splus-connections-file` specifies the name of the S-PLUS connections configuration file. The code fragment below specifies the filename to be `SplusConnections.xml`, which in any case is the default value.

```
<context-param>
  <param-name>splus-connections-file</param-name>
  <param-value>SplusConnections.xml</param-value>
</context-param>
```

`splus-inspect-interval-minutes`

The S-PLUS connection management code runs an inspector thread that identifies hung connections and attempts to fix broken connections. The `splus-inspect-interval-minutes` parameter specifies how frequently the inspector will run. The default is every 5 minutes. If you set this parameter to 0, the inspector will never run. The sample code below causes the inspector to run every hour.

```
<context-param>
  <param-name>splus-inspect-interval-minutes</param-name>
  <param-value>60</param-value>
</context-param>
```

`splus-hang-time-minutes`

The inspector identifies hung S-PLUS sessions based on how long each session has been processing its current request. If a session has been running too long, the inspector deems it to be hung and it restarts the session. The `splus-hang-time-minutes` parameter specifies how long a session can process a request before the inspector deems it to be hung. The default value is 5 minutes. If you set the value to 0, the inspector will never deem any sessions to be hung. Make sure you set this parameter to a value longer than your longest analysis can run. The sample below sets the hang time to 10 minutes.

```
<context-param>
  <param-name>splus-hang-time-minutes</param-name>
  <param-value>10</param-value>
</context-param>
```

`splus-max-hit-count`

S-PLUS sessions running under S-PLUS Server leak substantial amounts of memory, 30 kb per graph in one test. The connection manager restarts the sessions from time to time to reclaim the leaked memory. The `splus-max-hit-count` parameter specifies the number of times an S-PLUS session can be used before it is restarted. The default value is 300 times. If this value is set to 0, the session will never be restarted on the basis of the hit count. The sample code below causes S-PLUS sessions to be restarted after every 1000 uses.

```
<context-param>
  <param-name>splus-max-hit-count</param-name>
  <param-value>1000</param-value>
</context-param>
```

`splus-reconnect-interval-minutes, splus-reconnect-start-time`

The S-PLUS connection manager can also restart all the S-PLUS connections periodically, for example every night. The `splus-reconnect-interval-minutes` parameter specifies how often the manager reconnects the sessions. The default value is once per day (1440 minutes). If the value is 0, the manager will never reconnect the sessions for this reason.

The `splus-reconnect-start-time` parameter specifies the first time the manager will reconnect all the S-PLUS sessions. The default value is the current time plus the `splus-reconnect-interval-minutes` value.

The sample code below tells the manager to reconnect all S-PLUS sessions every other day at 1:38 AM.

```
<context-param>
  <param-name>splus-reconnect-interval-minutes</param-name>
  <param-value>2880</param-value>
</context-param>
<context-param>
  <param-name>splus-reconnect-start-time</param-name>
  <param-value>1:38:00 AM</param-value>
</context-param>
```

Graphs parameters in web.xml

The S+JSP module temporarily stores graph files (like Graphlets and JPEGs) on the Web server so that the users' browsers can find and download them. The module periodically deletes these files. Several optional parameters in the `web.xml` file specify where the graph files are stored and how often they are deleted.

`graphs-directory`

The graphs-directory parameter specifies where the graphs are stored. Its value is a path relative to the root directory for the Web application. The default value is `/splus/`. The sample code fragment below changes the value to `/temp/`.

```
<context-param>
  <param-name>graphs-directory</param-name>
  <param-value>/temp/</param-value>
</context-param>
```

`graphs-delete-interval-minutes`

The `graphs-delete-interval-minutes` parameter specifies how often the module deletes old graphs files. The default value is once per day. A value of 0 specifies that the module never deletes the files. For a high-volume Web site you might want to delete the files more often. The sample code below specifies that the files be deleted every 5 mintues.

```
<context-param>
  <param-name>graphs-delete-interval-minutes</param-name>
  <param-value>5</param-value>
</context-param>
```

`graphs-delete-start-time`

The `graphs-delete-start-time` parameter specifies the first time the module should delete the graphs. This parameter is primarily useful if the module deletes the files only once per day and you want it to do so at night. The default value is the current time plus the value of the graphs-delete-interval-minutes parameter. The sample XML element below sets the first deletion time to 1:37 AM.

```
<context-param>
  <param-name>graphs-delete-start-time</param-name>
```



```
    <param-value>1:37:00 AM</param-value>
  </context-param>
```

`graphs-delete-age-minutes`

The `graphs-delete-age-minutes` parameter specifies how old a graph file must be before it is deleted. The default is six hours. For a high-volume Web site you might want to delete much younger files. The sample code below specifies that files at least 3 minutes old will be deleted.

```
  <context-param>
    <param-name>graphs-delete-age-minutes</param-name>
    <param-value>3</param-value>
  </context-param>
```

Security

This section summarizes the security considerations for a deployed Web application using S-PLUS Server. It identifies pages and servlets that pose risks if made available to malicious users. You can restrict access to these pages and servlets, and indeed to any or all pages in your application, by using access control provided by the Web container or by implementing your own access control. Information about your Web container or a good book on JavaServer Pages will explain the techniques.

You can, of course, simply leave out of your application all pages that pose risks, or deploy your application in a local environment where all possible users are trusted.

- In any Web application, pages that let users run their own JSP code are a serious security risk. In the sample application, the pages `jspCode.html` and `jspResult.jsp` provide this ability. If you make this application available to untrusted users, either remove these pages or restrict access to them.

- Pages that allow users to write and run their own S-PLUS code are a security risk. Examples of such pages are the pages `splusCode.html` and `splusResult.jsp` in the sample application. Any such pages in your application should be restricted to trusted users.

- Administrative pages for listing, adding and removing S-PLUS connections can be used to shut down your application. Including them in your application can simplify administrative tasks, but you should allow only administrators to access these pages.