JavaServer Pages Fundamentals

John Zukowski

1.0

jGuru.com

(C)2000 jGuru.com

Contents

Chapter 1. JavaServer Pages Fundamentals 1
1.1. Introduction to JavaServer Pages technology 1
1.2. JavaServer Pages Fundamentals 1
1.2.1. Introduction 2
1.2.1.1. JSP Advantages 2
1.2.1.2. Comparing JSP with ASP 3
1.2.1.3. JSP or Servlets? 4
1.2.2. JSP Architecture 5
1.2.3. JSP Access Models 8
1.2.4. JSP Syntax Basics 10
1.2.4.1. Directives 10
1.2.4.2. Declarations 11
1.2.4.3. Expressions 11
1.2.4.4. Scriptlets 11
1.2.4.5. Comments 12
1.2.5. Object Scopes 12
1.2.6. JSP Implicit Objects 12
1.2.7. Synchronization Issues 13
1.2.8. Exception Handling 14
1.2.9. Session Management 15
1.2.10. Standard Actions 16
1.2.10.1. Using JavaBean Components 16
1.2.10.2. Forwarding Requests 18
1.2.10.3. Including Requests 21
1.3. Resources 22
1.3.1. Web Sites 22
1.3.2. Documentation and Specs 22
1.3.3. Articles 22

Chapter 2. JavaServer Pages Fundamentals : Exercises 24
2.1. Magercise: Installing and Configuring Tomcat 25
2.2. Magercise: Understanding JSP object scope 27
2.3. Magercise: Exception Handling in JSP 29

(C)2000 jGuru.com

2.4. Magercise: Form processing using JSP 31

Appendix A. 34

(C)2000 jGuru.com

Chapter 1. JavaServer Pages Fundamentals

1.1. Introduction to JavaServer Pages technology

Concepts

After completing this module you will understand:

- The advantages of JSP technology
- The JSP architecture
- The life-cycle of a JSP page
- JSP syntax and semantics
- The role of JavaBean components within JSP pages

Objectives

By the end of this module you will be able to:

- Manage session-related information from JSP
- Communicate between JSP pages
- Process forms with JSP

Prerequisites

A general familiarity with object-oriented programming concepts and the Java programming language. If you are not familiar with these capabilities, see the Java Tutorial (http://java.sun.com/docs/books/tutorial/) . The exercises require the ability to modify and build simple Java programs and HTML-like pages. It may also help to understand the fundamentals of Web computing and servlets. For help on servlet-specific issues, see the earlier Fundamentals of Java Servlets (http://developer.java.sun.com/developer/onlineTraining/Servlets/Fundamentals/index.html) course, though that is on the Servlets 2.1 API, instead of the newer 2.2 version.

(C)2000 jGuru.com

About the Author

John Zukowski is the content czar at jGuru.com (http://www.jguru.com) , contributing author of *Professional JSP* from Wrox, as well as the Focus on Java (http://java.about.com) guide at About, Inc.

1.2. JavaServer Pages Fundamentals

1.2.1. Introduction

While there may be numerous technologies for building Web applications that serve dynamic content, the one that has really caught the attention of the development community is JavaServer Pages TM (JSP). And not without ample reason either. JSP not only enjoys cross-platform and cross-Web-server support, but effectively melds the power of server-side Java technology with the WYSIWYG features of static HTML pages.

JSP pages typically comprise of static HTML/XML components, special JSP tags, and optionally, snippets of code written in the Java programming language called "scriptlets". Consequently, JSP pages can be created and maintained by conventional HTML/XML tools. It is important to note that the JSP specification is a standard extension defined on top of the Servlet API. Thus, it leverages all of your experience with servlets. But there are significant differences between the two technologies. Unlike servlets, which is a programmatic technology requiring significant developer expertise, JSP enjoys a much wider audience. It cannot only be used by developers, but also by page designers who can now play a more direct role in the development lifecycle. Another advantage of JSP is the inherent separation of presentation from content facilitated by the technology, due its reliance upon reusable component technologies like the JavaBeans TM component architecture and Enterprise JavaBeans TM technology. This course provides you with an in-depth introduction to this versatile technology, and will use the Tomcat (http://jakarta.apache.org/tomcat/) JSP 1.1 Reference Implementation from the Apache group for running the example programs.

1.2.1.1. JSP Advantages

Separation of static from dynamic content: With servlets, the logic for generation of the dynamic content is an intrinsic part of the servlet itself, and is closely tied to the static presentation templates responsible for the user interface. Thus, even minor changes made to the UI typically result in the recompilation of the servlet. This tight coupling of presentation and content typically results in brittle, inflexible applications. However, with JSP, the logic to generate the dynamic content is kept separate from the static presentation templates by encapsulating it within external JavaBeans components. These are then created and used by the JSP page using special tags and scriptlets. When a page designer makes any changes to the presentation template, the JSP page

(C)2000 jGuru.com

is automatically recompiled and reloaded into the web server by the JSP engine.

Write Once Run Anywhere: JSP technology brings the "Write Once, Run Anywhere" paradigm to interactive Web pages. JSP pages can be easily moved not only across platforms, but also across web servers, without any changes.

Dynamic content can be served in a variety of formats: There is nothing that mandates the static template data within a JSP page to be of a certain format. Consequently, JSP can service a diverse clientele ranging from conventional browsers using HTML/DHTML, to handheld wireless devices like mobile phones and PDAs using WML, to even other B2B applications using XML.

Recommended Web access layer for n-tier architecture: Sun's J2EE Blueprints (http://java.sun.com/j2ee/blueprints/) , which offers guidelines for developing large-scale applications using the enterprise Java APIs, categorically recommends JSP over servlets for serving dynamic content.

Completely leverages the Servlet API: If you are a servlet developer, there is very little that you have to "unlearn" in moving over to JSP. In fact, servlet developers are at a distinct advantage because JSP is nothing but a high-level abstraction of servlets. You can do almost anything that can be done with servlets using JSP – but more easily!

1.2.1.2. Comparing JSP with ASP

Although the features offered by JSP may seem similar to that offered by Microsoft's Active Server Pages (ASP), they are fundamentally different technologies, as shown by the following table:

(C)2000 jGuru.com

	JavaServer Pages	Active Server Pages
Web Server Support	Most popular web servers including Apache, Netscape, and Microsoft IIS can be easily enabled with JSP.	Native support only within Microsoft IIS or Personal Web Server. Support for select servers using third-party products.
Platform Support	Platform independent. Runs on all Java-enabled platforms.	Is fully supported under Windows. Deployment on other platforms is cumbersome due to reliance on the Win32-based component model.
Component Model	Relies on reusable, cross-platform components like JavaBeans, Enterprise JavaBeans, and custom tag libraries.	Uses the Win32-based COM component model.
Scripting	Can use the Java programming language or JavaScript.	Supports VBScript and JScript for scripting.
Security	Works with the Java security model.	Can work with the Windows NT security architecture.
Database Access	Uses JDBC for data access.	Uses Active Data Objects for data access.
Customizable Tags	JSP is extensible with custom tag libraries.	Cannot use custom tag libraries and is not extensible.

1.2.1.3. JSP or Servlets?

It is true that both servlets and JSP pages have numerous features in common, and can be used for serving up dynamic Web content. Naturally, this may cause some confusion amongst developers as to when to opt for one of the technologies over the other. Luckily, Sun's J2EE Blueprints (http://java.sun.com/j2ee/blueprints/) offers some guidelines towards this. According to the Blueprints, servlets should be used strictly as a web server extension technology. This could include the implementation of specialized controller components offering services like authentication, database validation, and so forth. It is interesting to note that what is commonly known as the "JSP engine" itself is a specialized servlet running under the control of the servlet engine. Since JSP only deals with textual data, you will have to continue to use servlets when communicating with Java applets and applications.

JSP should be the preferred technology for developing typical web applications that are reliant upon dynamic content. JSP should also be used in place of proprietary web server extensions like server-side includes since it offers excellent features for handling repetitive content.

Exercise

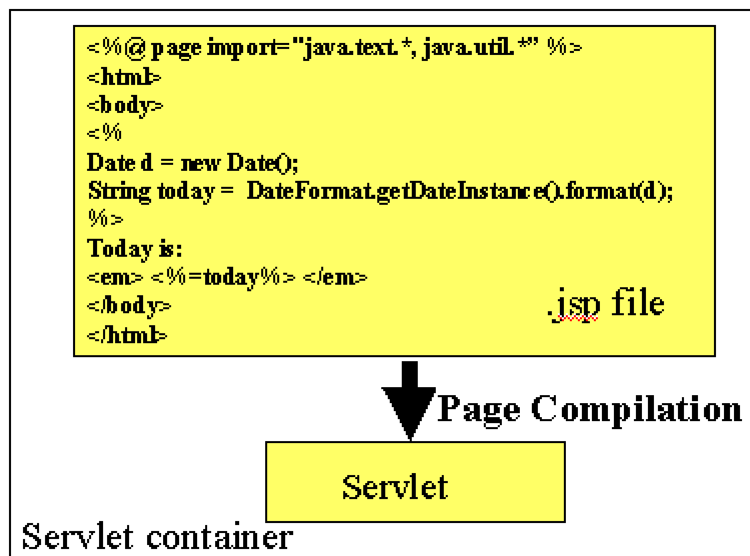
(C)2000 jGuru.com

1. Installing and Configuring Tomcat (page 25)

1.2.2. JSP Architecture

The purpose of JSP is to provide a declarative, presentation-centric method of developing servlets. As noted before, the JSP specification itself is defined as a standard extension on top the Servlet API. Consequently, it should not be too surprisingly that 'under the covers', servlets and JSP pages have a lot in common.

Typically, JSP pages are subject to a translation phase and a request processing phase. The translation phase is carried out only once, unless the JSP page changes, in which case it is repeated. Assuming there were no syntax errors within the page, the result is a JSP page implementation class file that implements the Servlet interface, as shown below.



The translation phase is typically carried out by the JSP engine itself, when it receives an incoming request for the JSP page for the first time. It should however be noted that the JSP 1.1 specification also allows for JSP pages to be precompiled into class files. Precompilation may be especially useful in removing the start-up lag that occurs when a JSP page delivered in source form receives the first request from a client. Many details of the translation phase, like the location where the source and class files are stored, and so on, are implementation dependent. The source for the class file generated by Tomcat for this example JSP page (shown in the above figure) is as follows:

```
package jsp;
```

(C)2000 jGuru.com

```
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.PrintWriter;
import java.io.IOException;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.util.Vector;
import org.apache.jasper.runtime.*;
import java.beans.*;
import org.apache.jasper.JasperException;
import java.text.*;
import java.util.*;

public class _0005cjsp_0005cjsptest_0002ejspjsptest_jsp_0
        extends HttpJspBase {

 static {
 }
 public _0005cjsp_0005cjsptest_0002ejspjsptest_jsp_0( ) {
 }

 private static boolean _jspx_inited = false;
 public final void _jspx_init() throws JasperException {
 }

 public void _jspService(HttpServletRequest request,
   HttpServletResponse response)
     throws IOException, ServletException {

     JspFactory _jspxFactory = null;
     PageContext pageContext = null;
     HttpSession session = null;
     ServletContext application = null;
     ServletConfig config = null;
     JspWriter out = null;
     Object page = this;
     String _value = null;
     try {
          if (_jspx_inited == false) {
             _jspx_init();
             _jspx_inited = true;
         }
         _jspxFactory = JspFactory.getDefaultFactory();
         response.setContentType("text/html");
         pageContext = _jspxFactory.getPageContext(this,
                        request,response, "", true, 8192, true);
```

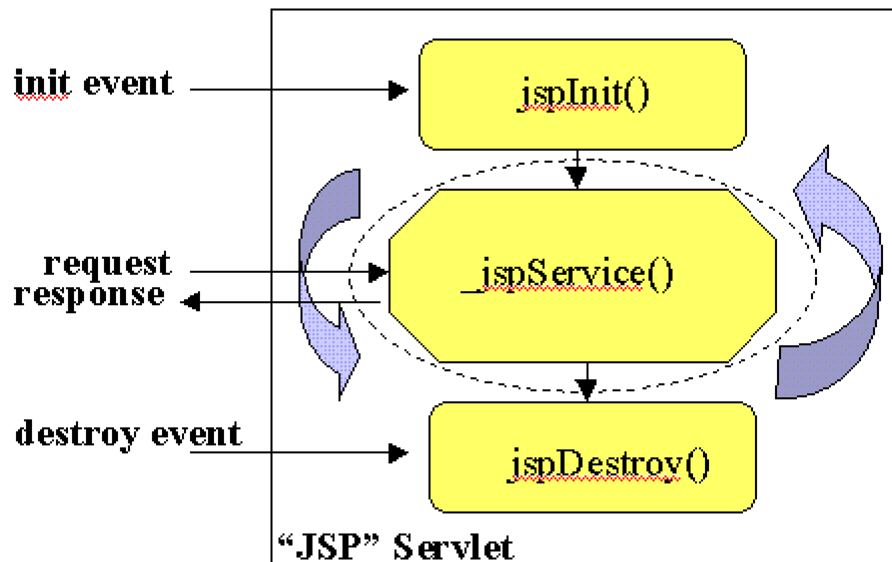
(C)2000 jGuru.com

```
                application = pageContext.getServletContext();
                config = pageContext.getServletConfig();
                session = pageContext.getSession();
                out = pageContext.getOut();
                // begin
                out.write("\r\n<html>\r\n<body>\r\n");
               // end
               // begin [file="E:\\jsp\\jsptest.jsp";from=(3,2);to=(5,0)]
               Date d = new Date();
               String today = DateFormat.getDateInstance().format(d);
               // end
               // begin
               out.write("\r\nToday is:  \r\n<em> ");
               // end
               // begin [file="E:\\jsp\\jsptest.jsp";from=(7,8);to=(7,13)]
               out.print(today);</b>
               // end
               // begin
               out.write(" </em>\r\n</body>\r\n</html>\r\n");
               // end
           } catch (Exception ex) {
              if (out.getBufferSize() != 0)
              out.clear();
              pageContext.handlePageException(ex);
           } finally {
              out.flush();
              _jspxFactory.releasePageContext(pageContext);
           }
       }
   }
```

The JSP page implementation class file extends *HttpJspBase*, which in turn implements the **Servlet** interface. Observe how the service method of this class, **_jspService()**, essentially inlines the contents of the JSP page. Although **_jspService()** cannot be overridden, the developer can describe initalization and destroy events by provinding implementations for the **jspInit()** and **jspDestroy()** methods within their JSP pages.

Once this class file is loaded within the servlet container, the **_jspService()** method is the one responsible for replying to a client's request. By default, the **_jspService()** method is dispatched on a separate thread by the servlet container in processing concurrent client requests, as shown below:

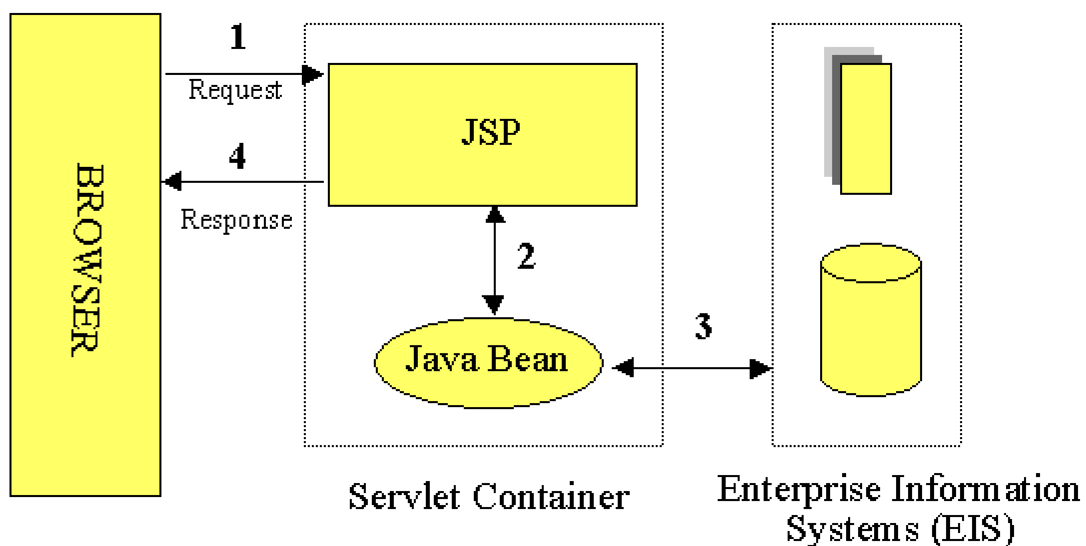
(C)2000 jGuru.com



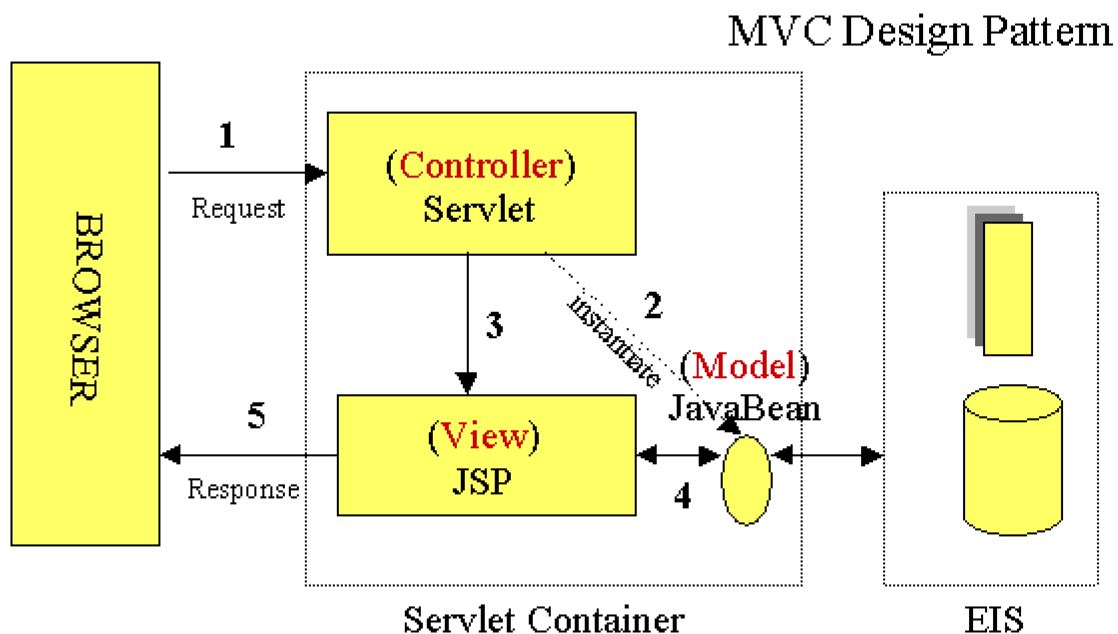
1.2.3. JSP Access Models

The early JSP specifications advocated two philosophical approaches, popularly known as Model 1 and Model 2 architectures, for applying JSP technology. These approaches differed essentially in the location at which the bulk of the request processing was performed, and offer a useful paradigm for building applications using JSP technology.

Consider the Model 1 architecture, shown below:



In the Model 1 architecture, the incoming request from a Web browser is sent directly to the JSP page, which is responsible for processing it and replying back to the client. There is still separation of presentation from content, because all data access is performed using beans. Although the Model 1 architecture should be perfectly suitable for simple applications, it may not be desirable for complex implementations. Indiscriminate usage of this architecture usually leads to a significant amount of scriptlets or Java code embedded within the JSP page, especially if there is a significant amount of request processing to be performed. While this may not seem to be much of a problem for Java developers, it is certainly an issue if your JSP pages are created and maintained by designers – which is usually the norm on large projects. Another downside of this architecture is that now, each of the JSP pages have to be individually responsible for managing application state and verifying authentication and security.



The Model 2 architecture, shown above, is a server-side implementation of the popular Model/View/Controller design pattern. Here, the processing is divided between presentation and front components. Presentation components are JSP pages that generate the HTML/XML response that determines the user interface when rendered by the browser. Front components (also known as controllers) do not handle any presentation issues, but rather, process all the HTTP requests. Here, they are responsible for creating any beans or objects used by the presentation components, as well as deciding, depending on the user's actions, which presentation component to forward the request to. Front components can be implemented as either a servlet or JSP page.

The advantage of this architecture is that there is no processing logic within the presentation component itself; it is simply responsible for retrieving any objects or beans that may have been previously created by the controller, and extracting the dynamic content within for insertion

(C)2000 jGuru.com

within its static templates. Consequently, this clean separation of presentation from content leads to a clear delineation of the roles and responsibilities of the developers and page designers on the programming team. Another benefit of this approach is that the front components present a single point of entry into the application, thus making the management of application state, security, and presentation uniform and easier to maintain.

1.2.4. JSP Syntax Basics

JSP syntax is fairly straightforward, and can be classified into directives, scripting elements, and standard actions.

1.2.4.1. Directives

JSP directives are messages for the JSP engine. They do not directly produce any visible output but instead tell the engine what to do with the rest of the JSP page. JSP directives are always enclosed within the < %"@" … % > tag. The two primary directives are **`page`** and **`include`**. (Note that JSP 1.1 also provides the **`taglib`** directive, which can be used for working with custom tag libraries, although this will not be discussing it here.)

Page directive

Typically, the **`page`** directive is found at the top of almost all your JSP pages. There can be any number of **`page`** directives within a JSP page, although the attribute/value pair must be unique. Unrecognized attributes or values result in a translation error. For example,

```
<%@ page import="java.util.*, com.foo.*" buffer="16k" %>
```

makes available the types declared within the included packages for scripting and sets the page buffering to 16K.

Include directive

The **`include`** directive lets you separate your content into more manageable elements, such as those for including a common page header or footer. The page included could be a static HTML page or more JSP content. For example, the directive:

```
<%@ include file="copyright.html" %>
```

can be used to include the contents of the indicated file at any location within the JSP page.

(C)2000 jGuru.com

1.2.4.2. Declarations

JSP declarations let you define page-level variables to save information or define supporting methods that the rest of a JSP page may need. While it is easy to get led away and have a lot of code within your JSP page, this move will eventually turn out to be a maintenance nightmare. For that reason, and to improve reusability, it is best that logic-intensive processing is encapsulated as JavaBean components.

Declarations are found within the < %! ... % > tag. Always end variable declarations with a semicolon, as any content must be valid Java statements:

```
<%! int i=0; %>
```

You can also declare methods. For example, you can override the initialization event in the JSP lifecycle by declaring:

```
<%! public void jspInit() {
        //some initialization code
    }
%>
```

1.2.4.3. Expressions

With expressions in JSP, the results of evaluating the expression are converted to a string and directly included within the output page. Typically expressions are used to display simple values of variables or return values by invoking a bean's getter methods. JSP expressions begin within < %= ... % > tags and do not include semicolons:

```
<%= fooVariable %>
 <%= fooBean.getName() %>
```

1.2.4.4. Scriptlets

JSP code fragments or scriptlets are embedded within < % ... % > tags. This Java code is then run when the request is serviced by the JSP page. You can have just about any valid Java code within a scriptlet, and is not limited to one line of source code. For example, the following displays the string "Hello" within H1, H2, H3, and H4 tags, combining the use of expressions and scriptlets:

```
<% for (int i=1; i<=4; i++) { %>
    <H<%=i%>>Hello</H<%=i%>>
<% } %>
```

1.2.4.5. Comments

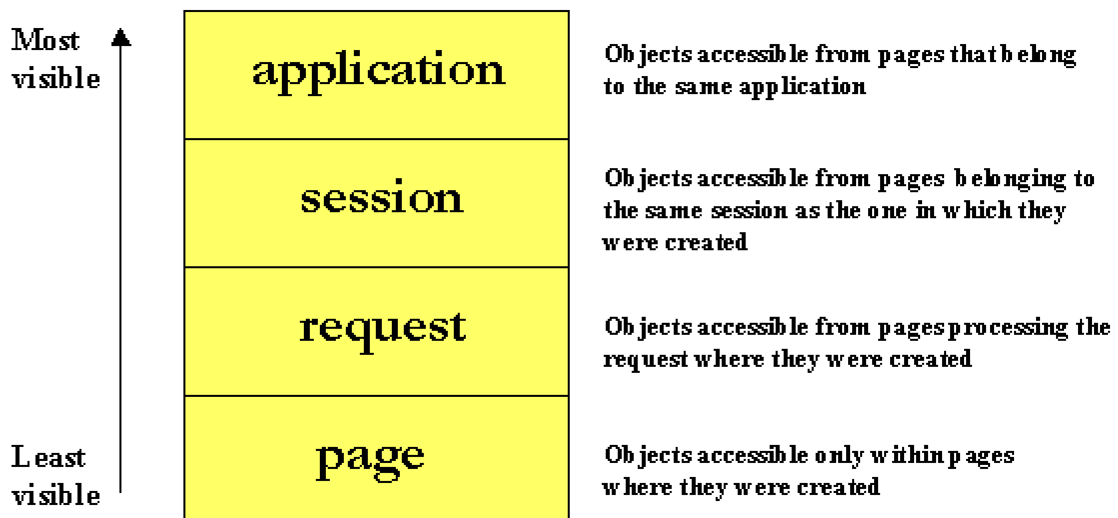
Although you can always include HTML comments in JSP pages, users can view these if they view the page's source. If you don't want users to be able to see your comments, you would embed them within the $<\%- \ldots -\%>$ tag:

```
<%- comment for server side only -%>
```

A most useful feature of JSP comments is that they can be used to selectively block out scriptlets or tags from compilation. Thus, they can play a significant role during the debugging and testing process.

1.2.5. Object Scopes

Before we take a look at JSP syntax and semantics, it is important to understand the scope or visibility of Java objects within JSP pages that are processing a request. Objects may be created implicitly using JSP directives, explicitly through actions, or, in rare cases, directly using scripting code. The instantiated objects can be associated with a scope attribute defining where there is a reference to the object and when that reference is removed. The following diagram indicates the various scopes that can be associated with a newly created object:



1.2.6. JSP Implicit Objects

As a convenience feature, the JSP container makes available implicit objects that can be used within scriptlets and expressions, without the page author first having to create them. These

(C)2000 jGuru.com

objects act as wrappers around underlying Java classes or interfaces typically defined within the Servlet API. There are nine implicit objects made available to the JSP author, and are shown below:

- request: represents the HttpServletRequest triggering the service invocation. Request scope.
- response: represents HttpServletResponse to the request. Not used often by page authors. Page scope.
- pageContext: encapsulates implementation-dependent features in PageContext. Page scope.
- application: represents the ServletContext obtained from servlet configuration object. Application scope.
- out: a JspWriter object that writes into the output stream. Page scope.
- config: represents the ServletConfig for the JSP. Page scope.
- page: synonym for the "this" operator, as a HttpJspPage. Not used often by page authors. Page scope.
- session: A HttpSession. Session scope. More on sessions shortly (page 15) .
- exception: the uncaught Throwable object that resulted in the error page being invoked. Page scope.

It should be noted that these implicit objects are only visible within the system generated **`_jspService()`** method. They are not visible within methods you define yourself in declarations.

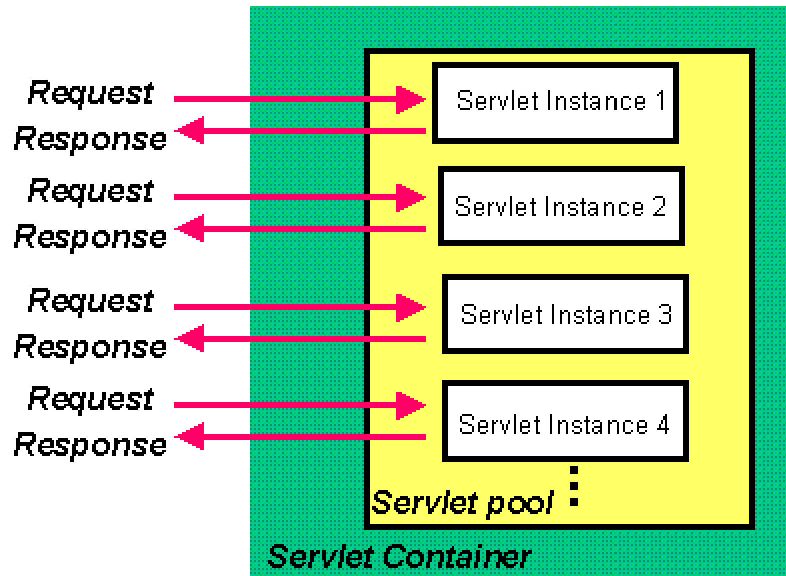
1.2.7. Synchronization Issues

By default, the service method of the JSP page implementation class that services the client request is multithreaded. Thus, it is the responsibility of the JSP page author to ensure that access to shared state is effectively synchronized. There are a couple of different ways to ensure that the service methods are thread-safe. The easy approach is to include the JSP page directive:

```
<%@ page isThreadSafe="true" %>
```

This causes the JSP page implementation class to implement the **`SingleThreadModel`** interface, resulting in the synchronization of the service method, and having multiple instances of the servlet to be loaded in memory. The concurrent client requests are then distributed evenly amongst these instances for processing in a round-robin fashion, as shown below:

(C)2000 jGuru.com



The downside of using this approach is that it is not scalable. If the wait queue grows due to a large number of concurrent requests overwhelming the processing ability of the servlet instances, then the client may suffer a significant delay in obtaining the response.

A better approach would be to explicitly synchronize access to shared objects (like those instances with application scope, for example) within the JSP page, using scriptlets:

```
<%
synchronized (application) {
   SharedObject foo = (SharedObject)
        application.getAttribute("sharedObject");
   foo.update(someValue);
   application.setAttribute("sharedObject",foo);
}
%>
```

1.2.8. Exception Handling

JSP provides a rather elegant mechanism for handling runtime exceptions. Although you can provide your own exception handling within JSP pages, it may not be possible to anticipate all situations. By making use of the **page** directive's **errorPage** attribute, it is possible to forward an uncaught exception to an error handling JSP page for processing. For example,

```
<%@ page isErrorPage="false" errorPage="errorHandler.jsp" %>
```

informs the JSP engine to forward any uncaught exception to the JSP page **errorHan-**

dler.jsp. It is then necessary for **errorHandler.jsp** to flag itself as a error processing page using the directive:

```
<%@ page isErrorPage="true" %>
```

This allows the **Throwable** object describing the exception to be accessed within a scriptlet through the implicit **exception** object.

Exercise

2. Exception Handling in JSP (page 29)

1.2.9. Session Management

By default, all JSP pages participate in an HTTP session. The HttpSession object can be accessed within scriptlets through the **session** implicit JSP object. Sessions are a good place for storing beans and objects that need to be shared across other JSP pages and servlets that may be accessed by the user. The session objects is identified by a session ID and stored in the browser as a cookie. If cookies are unsupported by the browser, then the session ID may be maintained by URL rewriting. Support for URL rewriting is not mandated by the JSP specification and is supported only within a few servers. Although you cannot place primitive data types into the session, you can store any valid Java object by identifying it by a unique key. For example:

```
<%
Foo foo = new Foo();
session.putValue("foo",foo);
%>
```

makes available the Foo instance within all JSP pages and servlets belonging to the same session. The instance may be retrieved within a different JSP page as:

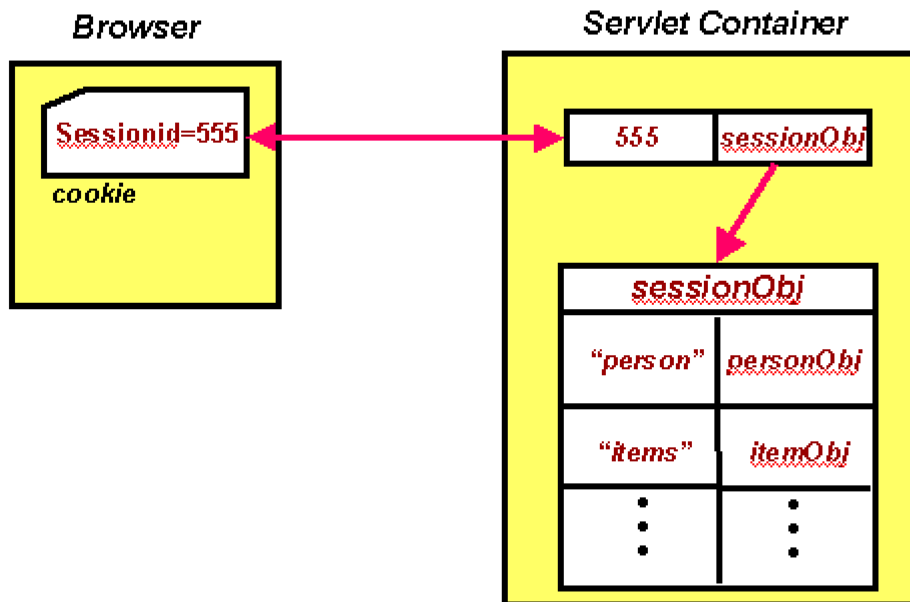
```
<%
Foo myFoo = (Foo) session.getValue("foo");
%>
```

The call to **session.getValue()** returns a reference to the generic **Object** type. Thus it is important to always cast the value returned to the appropriate data type before using it. It is not mandatory for JSP pages to participate in a session; they may choose to opt out by setting the appropriate attribute of the **page** directive:

```
<%@ page session="false" %>
```

(C)2000 jGuru.com

There is no limit on the number of objects you can store into the session. However, placing large objects into the session may degrade performance as they take up valuable heap space. By default, most servers set the lifetime of a session object to 30 minutes, although you can easily reset it on a per session basis by invoking `setMaxInvalidationInterval(int secs)` on the session object. The figure below highlights the general architecture of session management:



The JSP engine holds a live reference to objects placed into the session as long as the session is valid. If the session is invalidated or encounters a session timeout, then the objects within are flagged for garbage collection.

1.2.10. Standard Actions

Actions allow you to perform sophisticated tasks like instantiating objects and communicating with server-side resources like JSP pages and servlets without requiring Java coding. Although the same can be achieved using Java code within scriptlets, using action tags promotes reusability of your components and enhances the maintainability of your application.

1.2.10.1. Using JavaBean Components

The component model for JSP technology is based on JavaBeans component architecture. JavaBeans components are nothing but Java objects which follow a well-defined design/naming pattern: the bean encapsulates its properties by declaring them private and provides public accessor (getter/setter) methods for reading and modifying their values.

(C)2000 jGuru.com

Before you can access a bean within a JSP page, it is necessary to identify the bean and obtain a reference to it. The **< jsp:useBean >** tag tries to obtain a reference to an existing instance using the specified id and scope, as the bean may have been previously created and placed into the session or application scope from within a different JSP page. The bean is newly instantiated using the Java class name specified through the class attribute only if a reference was not obtained from the specified scope. Consider the tag:

```
<jsp:useBean id="user" class="com.jguru.Person"
 scope="session" />
```

In this example, the Person instance is created just once and placed into the session. If this **useBean** tag is later encountered within a different JSP page, a reference to the original instance that was created before is retrieved from the session.

The **< jsp:useBean >** tag can also optionally include a body, like

```
<jsp:useBean id="user" class="com.jguru.Person"
   scope="session">
<%
  user.setDate(DateFormat.getDateInstance().format(new Date()));
  //etc..
%>
</jsp:useBean>
```

Any scriptlet (or **< jsp:setProperty >** tags which are explained shortly) present within the body of a **< jsp:useBean >** tag are executed only when the bean is instantiated, and are used to initialize the bean's properties.

Once you have declared a JavaBean component, you have access to its properties to customize it. The value of a bean's property is accessed using the **< jsp:getProperty >** tag. With the **< jsp:getProperty >** tag, you specify the name of the bean to use (from the id field of **useBean**), as well as the name of the property whose value you are interested in. The actual value is then directly printed to the output:

```
<jsp:getProperty name="user" property="name" />
```

Changing the property of a JavaBean component requires you to use the **< jsp:setProperty >** tag. For this tag, you identify the bean and property to modify and provide the new value:

```
<jsp:setProperty name="user" property="name"
  value="jGuru" />
```

or

(C)2000 jGuru.com

```
<jsp:setProperty name="user" property="name"
  value="<%=expression %>" />
```

When developing beans for processing form data, you can follow a common design pattern by matching the names of the bean properties with the names of the form input elements. You would also need to define the corresponding getter/setter methods for each property within the bean. The advantage in this is that you can now direct the JSP engine to parse all the incoming values from the HTML form elements that are part of the request object, then assign them to their corresponding bean properties with a single statement, like this:

```
<jsp:setProperty name="user" property="*"/>
```

This runtime magic is possible through a process called introspection, which lets a class expose its properties on request. The introspection is managed by the JSP engine, and implemented through the Java reflection mechanism. This feature alone can be a lifesaver when processing complex forms containing a significant number of input elements.

If the names of your bean properties do not match those of the form's input elements, they can still be mapped explicitly to your property by naming the parameter as:

```
<jsp:setProperty name="user" property="address"
  param="parameterName" />
```

Exercises

3. Understanding JSP object scope (page 27)
4. Form processing using JSP (page 31)

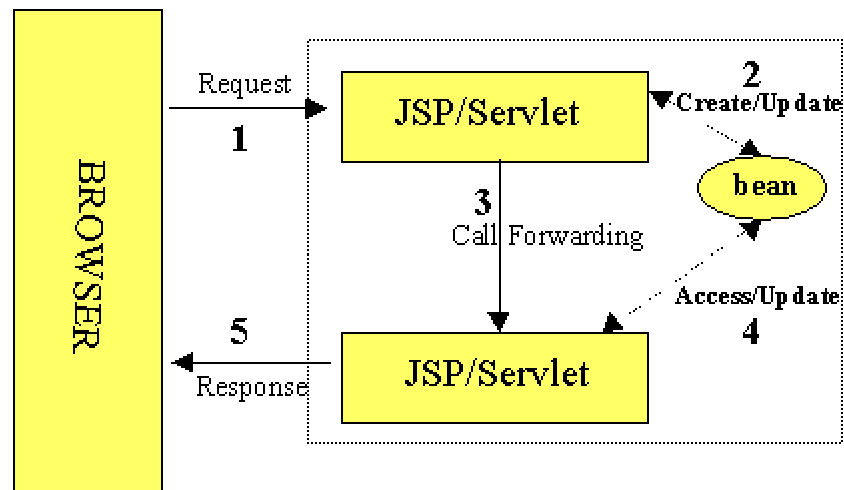
1.2.10.2. Forwarding Requests

With the **`< jsp:forward >`** tag, you can redirect the request to any JSP, servlet, or static HTML page within the same context as the invoking page. This effectively halts processing of the current page at the point where the redirection occurs, although all processing up to that point will still take place:

```
<jsp:forward page="somePage.jsp" />
```

The invoking page can also pass the target resource bean parameters by placing them into the request, as shown in the diagram:

(C)2000 jGuru.com

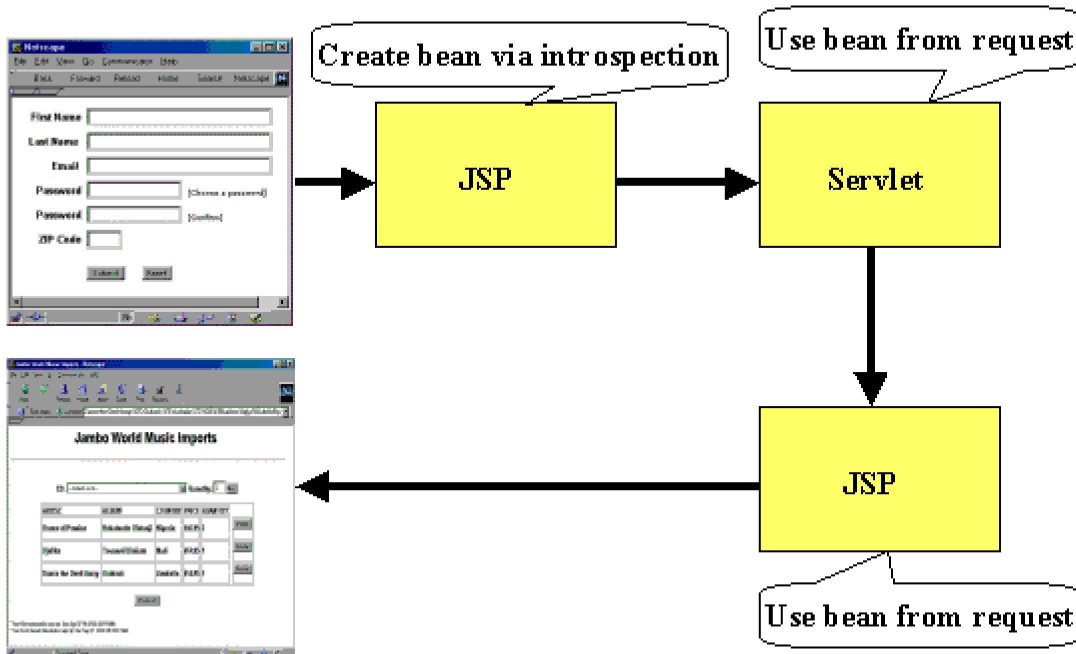


A < **jsp:forward** > tag may also have **jsp:param** subelements that can provide values for some elements in the request used in the forwarding:

```
<jsp:forward page="<%= somePage %>" >
<jsp:param name="name1" value="value1" />
<jsp:param name="name2" value="value2" />
</jsp:forward>
```

Request Chaining

Request chaining is a powerful feature and can be used to effectively meld JSP pages and servlets in processing HTML forms, as shown in the following figure:



Consider the following JSP page, say **Bean1.jsp**, which creates a named instance **fBean** of type **FormBean**, places it in the request, and forwards the call to the servlet **JSP2Servlet**. Observe the way the bean is instantiated – here we automatically call the bean's setter methods for properties which match the names of the posted form elements, while passing the corrosponding values to the methods.

```
<jsp:useBean id="fBean" class="govi.FormBean"
  scope="request"/>
<jsp:setProperty name="fBean" property="*" />
<jsp:forward page="/servlet/JSP2Servlet" />
```

The servlet **JSP2Servlet** now extracts the bean passed to it from the request, makes changes using the appropriate setters, and forwards the call to another JSP page **Bean2.jsp** using a request dispatcher. Note that this servlet, acting as a controller, can also place additional beans if necessary, within the request.

```
public void doPost (HttpServletRequest request,
                    HttpServletResponse response) {
  try {
    FormBean f = (FormBean) request.getAttribute ("fBean");
    f.setName("Mogambo");
    // do whatever else necessary
    getServletConfig().getServletContext().
      getRequestDispatcher("/jsp/Bean2.jsp").
        forward(request, response);
```

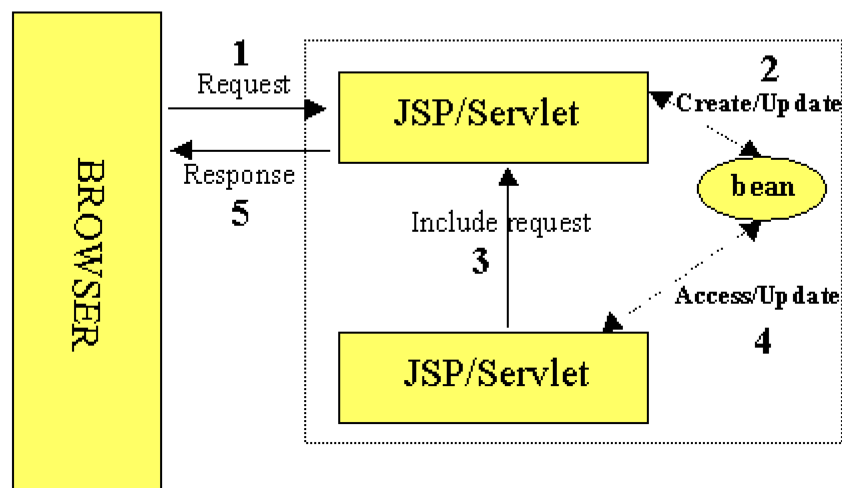
```
    } catch (Exception ex) {
      . . .
    }
  }
```

The JSP page **`Bean2.jsp`** can now extract the bean **`fBean`** (and whatever other beans that may have been passed by the controller servlet) from the request and extract its properties.

```
<html>
<body>
<jsp:useBean id="fBean" class="govi.FormBean"
   scope="request"/>
<jsp:getProperty name="fBean" property="name" />
</body>
</html>
```

1.2.10.3. Including Requests

The **`< jsp:include >`** tag can be used to redirect the request to any static or dynamic resource that is in the same context as the calling JSP page. The calling page can also pass the target resource bean parameters by placing them into the request, as shown in the diagram:



For example:

```
<jsp:include page="shoppingcart.jsp" flush="true"/>
```

(C)2000 jGuru.com

not only allows `shoppingcart.jsp` to access any beans placed within the request using a `< jsp:useBean >` tag, but the dynamic content produced by it is inserted into the calling page at the point where the `< jsp:include >` tag occurs. The included resource however cannot set any HTTP headers, which precludes it from doing things like setting cookies, or else an exception will be thrown.

1.3. Resources

1.3.1. Web Sites

The following sites have product information as well as whitepapers on JSP and Servlets:

- Sun Microsystems, JSP Home Page (http://java.sun.com/products/jsp/)
- JSP-INTEREST Mailing List Archive (http://archives.java.sun.com/archives/jsp-interest.html)
- jGuru's JSP FAQ (http://www.jguru.com/jguru/faq/faqpage.jsp?name=JSP)
- jGuru's Servlets FAQ (http://www.jguru.com/jguru/faq/faqpage.jsp?name=Servlets)

1.3.2. Documentation and Specs

The Java Technology (http://java.sun.com) site at Sun Microsystems includes a Products and APIs (http://java.sun.com/products/) page which lists enterprise-related products and APIs. Several of the ones relevant to JSP are listed here:

- JSP 1.1 Specification (http://java.sun.com/products/jsp/download.html)
- Sun Microsystems, Inc. Java 2 Enterprise Edition (J2EE) Home page (http://java.sun.com/j2ee/)
- The Tomcat Project (http://java.sun.com/products/jsp/tomcat/)
- JSP Technical Resources (http://java.sun.com/products/jsp/technical.html)
- Java Servlet API (http://java.sun.com/products/servlet/)
- JSP Whitepaper (http://java.sun.com/products/jsp/whitepaper.html)
- JSP Syntax Card (http://java.sun.com/products/jsp/syntax.html)

1.3.3. Articles

Some articles on JSP computing include:

- Advanced Form Processing using JSP (http://www.javaworld.com/javaworld/jw-

(C)2000 jGuru.com

03-2000/jw-0331-ssj-forms.html) by Govind Seshadri (JavaWorld, March 2000)

- JSP Architectures (http://www.brainopolis.com/jsp/book/jspBook_Architectures.html) by Lance Lavandowska, brainopolis.com

- Java serves the Web (http://builder.cnet.com/Programming/JSP/) by John Zukowski (Builder.com, February 2000)

- Internationalize JSP-based Websites (http://www.javaworld.com/javaworld/jw-03-2000/jw-03-ssj-jsp.html) by Govind Seshadri (JavaWorld, February 2000)

- The Problems with JSP (http://www.servlets.com/soapbox/problems-jsp.html) by Jason Hunter (Servlets.com, January, 2000)

- Understanding JSP Model 2 Architecture (http://www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj-jspmvc.html) by Govind Seshadri (JavaWorld, December 1999)

- JSP for the ASP Developer (http://www.ASPToday.com/articles/19991022.htm) by Cindy Nordahl (ASP Today, October 1999)

(C)2000 jGuru.com

Chapter 2. JavaServer Pages Fundamentals : Exercises

Welcome to the jGuru exercises for the *JavaServer Pages Fundamentals* course module.

These exercises demonstrate how to use Tomcat – the JSP 1.1 Reference Implementation, as well as how to design, implement, and deploy JSPs.

When you finish these exercises, you will know the basic steps for designing, compiling, and deploying JSP web components.

(C)2000 jGuru.com

2.1. Magercise: Installing and Configuring Tomcat

This exercise steps you through the process of downloading and installing Tomcat – the JSP 1.1 Reference Implemetation (RI), on your machine. Tomcat comprises of a simple HTTP server as well as a Web container that can run JSP pages and servlets. Tomcat supports the Servlet 2.2 and JSP 1.1 specifications. We will use this server for the subsequent exercises.

Educational goal(s):

- Install Tomcat
- Configure your machine properly for compiling and deploying JSPs

Prerequisites

None

Solution There is no solution to this exercise. When the tasks in this exercise have been completed, Tomcat will be installed, running, and available for the subsequent exercises.

Introduction

This exercise steps you through the process of downloading and installing Tomcat – the JSP 1.1 Reference Implementation (RI) on your machine. The exercises are specific to Tomcat - if you wish to use a different server for the remainder of these exercises, you should ensure that it is JSP 1.1 compliant and install it now.

Perform the following tasks:

1. Check your system requirements to make sure you have an adequate hardware and software platform for installing and running Tomcat.
2. Download the appropriate version of Tomcat 3.1 from the Apache

(C)2000 jGuru.com

website (http://jakarta.apache.org/downloads/binindex.html) .

3. Uncompress the file.
4. Set the environment variable **`JAVA_HOME`** to point to the root directory of your JDK hierarchy. Be sure the Java interpreter is in your **`PATH`** environment variable.
5. Change to the **`bin`** directory and start Tomcat using the command-line command **`startup`**.
6. Tomcat is now installed and running on port 8080 by default. Explore the Tomcat documentation within the documentation site (http://java.sun.com/products/jsp/tomcat/) to familiarize yourself more with Tomcat.

(C)2000 jGuru.com

2.2. Magercise: Understanding JSP object scope

In this exercise you will observe the behavior of a Counter bean when used within a JSP page with different scope attributes.

Educational goal(s):

- Understand the importance of the **`scope`** attribute when instantiating beans using the **`useBean`** tag.
- Examine the difference between **`session`** and **`application`** scope.

Prerequisites

Installing and Configuring Tomcat (page 25)

Skeleton Files

- Counter.jsp
- CounterBean.java

Solution The following files contain a complete implementation of the example demonstrating JSP variable scope:

- Solution/Counter.jsp
- Solution/com/jguru/CounterBean.java

Introduction

This exercise implements a simple JSP page (**`Counter.jsp`**), which instantiates two instances of a bean which maintains a counter (**`CounterBean.java`**), but with differing scope. One of the beans is attributed with session scope, and the other with application scope. Each time the JSP page in invoked, the count of each of the beans is incremented by one. You can

(C)2000 jGuru.com

observe the difference between session and application scope when you access the counter page from different browsers. You will notice that each browser maintains a distinct count for their session, but share the counter with application scope, since it is treated as a global variable.

Perform the following tasks:

1. Develop a simple counter bean, `CounterBean.java`.
2. Compile the counter bean.
3. Deploy the bean within Tomcat.
4. Develop a JSP page, `Counter.jsp`, which creates two instances of the counter bean, one with session scope, and the other with application scope.
5. Deploy the JSP file for the example within Tomcat.
6. Run the example.

(C)2000 jGuru.com

2.3. Magercise: Exception Handling in JSP

In this exercise you will learn how to redirect runtime exceptions occuring within a JSP page to an error handling page.

Educational goal(s):

- Learn how to handle runtime exceptions occuring within JSP pages by automatically forwarding them to an "error handler" page
- Understand how exceptions can be accessed from within a JSP "error handler" page

Prerequisites

Installing and Configuring Tomcat (page 25)

Skeleton Files

- errhandler.jsp
- errorpage.jsp

Solution The following files contain a complete implementation of the JSP error handling example:

- Solution/errhandler.jsp
- Solution/errorpage.jsp

Introduction

This exercise implements a JSP page (**`errhandler.jsp`**), which processes a POST operation and throws an exception in case of an "incorrect" answer. You will see how these exceptions can be automatically forwarded by the JSP engine to an "error handler". You also develop an error processing JSP page (**`errorpage.jsp`**), which receives the exception by means of the

`exception` implicit variable.

Perform the following tasks:

1. Design a JSP page called **`errhandler.jsp`** that can process a POST operation.
2. Indicate an error page, **`errorpage.jsp`**, using the **`page`** directive for the JSP page.
3. Process the posted form elements. Throw an exception if the value posted for the input element is not equal to an expected value, else print an acknowledgement back to the user.
4. Develop an error page, **`errorpage.jsp`**, which can access the runtime exception.
5. Deploy the JSP files for the example within Tomcat.
6. Run the error handling example.

(C)2000 jGuru.com

2.4. Magercise: Form processing using JSP

In this exercise you will learn how to process HTML forms using JSPs, and understand the introspective features provided by the JSP engine.

Educational goal(s):

- Understand the ease with which HTML forms can be processed using JSP pages.
- Understand the role played by beans in streamling form processing.

Prerequisites

Installing and Configuring Tomcat (page 25)

Skeleton Files

- FormBean.java
- form.jsp

Solution

The following Java source files represent a solution to this Magercise:

- Solution/com/jguru/FormBean.java
- Solution/form.jsp

Introduction

In this exercise, you will develop a simple JSP page (**`form.jsp`**), which is capable of processing an HTML form containing typical input elements like textboxes, radio buttons and checkboxes. You will also develop a bean (**`FormBean.java`**), whose property names mirror the input elements of

(C)2000 jGuru.com

the form. You will then examine the automatic instantiation of the bean on a form POST operation, using the introspective features provided by the JSP engine.

Perform the following tasks:

1. You are given the JSP page containing the form. Observe that the form posts to itself recursively. Instantiate the bean `FormBean` when you recognize that a POST operation has taken place. Allow the setter methods to be called on the bean using introspection.
2. Deploy the JSP page within Tomcat.
3. Develop the bean, `FormBean.java` with properties matching the form elements.
4. Compile the bean source `FormBean.java`.
5. Deploy the bean within Tomcat.
6. Run the example.

(C)2000 jGuru.com

(C)2000 jGuru.com

Appendix A

About jGuru Exercises

A jGuru exercise is a flexible exercise designed to provide help according to the needs of the student. For example, some students will simply complete the exercise given the information and the task list in the exercise body; some students may want a few hints while others may want a step-by-step guide to successfully complete a particular exercise. Students may use as much or as little help as they need per exercise. Moreover, since complete solutions are also provided, students can skip a few exercises and still be able to complete future exercises requiring the skipped ones.

The Anatomy of An Exercise

Each exercise has a list of any prerequisite exercises, a list of skeleton code for you to start with, links to necessary API pages, and a text description of the exercise goal. In addition, the following information is available via five buttons:

- **Expected behavior**: Launches an applet illustrating the desired behavior from your applet.
- **Table of contents**: Brings up the table of contents for the course notes and the list of exercises.
- **Help**: Gives you help or hints on the current exercise (an annotated solution).
- **Solution**: The `< applet >` tag and Java source resulting in the expected behavior.
- **API Documentation**: A link directly to the online API documentation.

Exercise Design Goals

There are three fundamental exercise types:

"Blank screen"
: The programmer is confronted with a "blank screen"; i.e., the programmer creates the entire desired functionality.

Extension
: The programmer extends the functionality of an existing, correctly-working program.

Repair

The programmer repairs undesirable behavior in an existing program.

Where possible, **the programmer shall be relieved from chores that are irrelevant or unrelated to the technique or concept under examination**.

Where reasonable, **a common thread shall run through the exercises** for each lab section.

Given the constraints of the technique or concept under examination, the **exercises shall be made as interesting or useful as possible** without presenting an overly-complex programming problem to the student.

Exercises shall execute via the web unless a particular concept related to non-web execution is required or the browser does not support the capabilities yet.

In addition, exercises that must access Java features or library elements causing web security violations are not executed on the web.

(C)2000 jGuru.com