

Chapter

4

SECTION I: JAVA SERVLETS AND JAVA SERVER PAGES

Introduction To Java Server Pages

Since modern enterprise applications are moving from two-tier towards three-tier and N-tier architectures, there arises a need of discovering different ways to deliver applications and data to users.

The major drawback with the traditional thick client that holds the entire application on the local computer is that it becomes difficult distributing and updating the application.

Web based clients provide an excellent alternative for building Intranet and Internet enterprise applications. The new **Java Server Pages [JSP]** technology 2.1, part of the Java EE 5, gives web and Java developers a simple yet powerful mechanism for creating such web applications.

The JSP technology:

- ❑ Is a language for developing JSP pages, which are text-based documents that describe how to process a request and construct a response
- ❑ Is an expression language for accessing server-side objects

48 Java Server Pages For Beginners

- ❑ Allows creating web content with both static and dynamic components
- ❑ Makes available all the dynamic capabilities of Java Servlet technology but provides a more natural approach to creating static content
- ❑ Provides mechanisms for defining extensions to the JSP language
- ❑ Provides developers with the ability to access remote data via mechanisms such as Enterprise Java Beans [EJB], Remote Method Invocation [RMI] and Java DataBase Connectivity [JDBC]
- ❑ Provides developers with the ability to encapsulate and separate program logic from the presentation i.e. the HTML elements, to help maximize code reuse and flexibility. This separation of logic and presentation is a major advantage over the web application architectures such as Java Servlets and CGI scripts

A JSP page is a text document that contains two types of text:

- ❑ **Static data**, which can be expressed in any text-based format such as HTML, SVG, WML and XML
- ❑ **JSP elements**, which construct dynamic content

The recommended file extension for the source file of a JSP page is **.jsp**. The page can be composed of a top file that includes other files that contain either a complete JSP page or a fragment of a JSP page. The recommended extension for the source file of a fragment of a JSP page is **.jspx**.

Why Use Java Server Pages

The benefits of using JSP:

Nobody Can Borrow The Code

When a web site that does something really cool, which attracts the attention of the users, developers look at the source code of the page and copy the JavaScript or other code into their own pages which then do the same cool stuff.

With JSP, this issue does not arise at all. The code written runs and remains on the Web server. All of its functionality is handled before the page is sent to a Browser.

Faster Loading Of Pages

When DHTML, JavaScript or any other client-side technology is used to customize page content, the page developer must send all of the content that a user might want plus the code that is necessary to hide and reveal those sections.

With JSP, decisions can be made about what the user wants to see at the Web server prior the pages being dispatched. Hence, only the content that the user is interested in will be dispatched to the user, with no extra code and extra content.

No Browser Compatibility Issues

Those who craft JavaScript or other scripts know that the code should always be checked thoroughly across several versions of several browsers to make sure it will work as expected. The most common browsers are Netscape, Mozilla Firefox, Internet Explorer and Opera. Making JavaScript work in all these Browsers either things are kept very simple or custom code is created for multiple versions of some of the browsers and delivered appropriately.

JSP has no such issues. The developer ends up sending standard HTML to a user Browser. This largely eliminates scripting issues and cross Browser compatibility.

JSP Support

JSP is supported by a number of Web servers. Built-in support for JSP is available in Java Web Server from Sun.

For Web servers that do not directly support JSP and Java Servlets, add-on support is provided through products such as Live Software's JRun [recently acquired by Allaire]. JRun works with a number of popular Web servers like Apache, Microsoft IIS and PWS, Netscape's FastTrack and Enterprise Web servers and others.

Compilation

Another important benefit is that the JSP is always compiled before the web server processes it. The older technologies such as CGI require the server to load an interpreter and the target script each time the page is requested.

JSP gets around this problem by compiling each JSP page into executable code the first time it is requested and invoking the resulting code directly on all subsequent requests. When coupled with a persistent JVM on a Java enabled web server, this allows the server to handle JSP pages much faster.

JSP Elements In HTML / XML Pages

A JSP page looks a lot like an HTML or XML page. It holds text marked up with an assortment of tags. While a regular JSP page is not a valid XML page, there is a variant JSP tag syntax that lets the developer use JSP tags within XML documents.

What is different about JSP compared to regular HTML is that the tags are not all processed by the browser instead the Web server processes the JSP tags allowing page content to change dynamically prior its delivery to a user Browser. This is very similar to other dynamic page generation systems such as Microsoft's Active Server Pages [ASP] or Allaire's ColdFusion. JSP tags allow Java code to be embedded directly in an HTML page, which is then executed first prior the page being delivered to a user Browser.

Disadvantages Of JSP

The disadvantages of using JSP:

Attractive Java Code

Putting Java code spec within a web page is really bad design, but JSP makes it tempting to do just that. Avoid this as far as possible. One excellent solution is to use a template engine.

Template engines improve on JSP by removing the ability to put raw code in the page. Thus template engines enforce good page design.

Java Code Required

To do relatively simple things in JSP can actually demand putting Java code in a page. Assume a page needs to determine the context root of the current web application, perhaps to create a link to the web applications, home page.

This is done using Java code in JSP:

```
<A HREF='<%=request.getContextPath() %>/index.html'>Home page</A>
```

Java code can be avoided by using the `<jsp:getProperty>` element but that makes the code spec more complex:

```
<A HREF='<jsp:getProperty name="request" property="contextPath"/>/index.html'>
  HomePage </A>
```

Simple Tasks Are Hard To Code

Even including page headers and footers is a bit difficult with JSP. Assume there is a header template and a footer template to be included on all pages. Each template also includes the current page's title.

In JSP the best way to do this is as follows:

```
<% String title = "The Page Title"; %>
<%@ include file="/header.jsp" %>
/* Some content here */
<%@ include file="/footer.jsp" %>
```

Page creators must remember to place a semi-colon in the first line. They must also make sure to declare **title** as a Java String.

Additionally, the **/header.jsp** and **/footer.jsp** must be made publicly accessible somewhere under the document root even though they are not full pages themselves.

Difficult Looping In JSP

Looping is difficult in JSP. Following is the JSP code that loops through a vector of JSP objects and prints out the name of each:

```
<%
Enumeration e = list.elements();
while (e.hasMoreElements())
{
    out.print("The next name is ");
    out.println(((JSP)e.nextElement()).getName());
    out.print("<BR>");
}
%>
```

In future there's the possibility of having custom tags for **loops**, as well as for **If** conditions.

Occupies A Lot Of Space

Java Server Pages consume extra hard drive and memory [RAM] space. For every 30K JSP file on the Web server there will be a corresponding much larger class file created. This essentially more than doubles the hard drives, space requirements, to store JSP pages. Considering how easily a JSP file can **include** a large data file for display, this is a real concern. Each JSP's, class file data, must be loaded into the Web server's memory. This means the Web server may eventually store the entire JSP document tree in its memory.

A few JVMs have the ability to remove such class file data from memory [i.e. RAM], however, the programmer generally has no control over the rules for reclaiming used memory. For large sites memory reclamation may not be aggressive enough, this will slow down Web Server performance. Using template engines there's no need to duplicate page data into a second file, so hard drive space is spared. Template engines also give the programmer complete control over how templates are cached in memory.

JSP v/s Servlets

Servlets provide the ability to build dynamic content for Web sites using Java and are supported by all major Web servers.

Why JSP If Java Already Has An API For Answering HTTP Requests

Since Java Server Pages get automatically translated into Java Servlets. Hence, there is no difference between what can be done by a JSP or a Servlet. Both JSP and Servlets are executed by a Java Virtual Machine [JVM]. This eliminates the need for a Web server to create a new process each time a web page request arrives. A really huge advantage over CGI scripts.

The distinct advantage of JSP over Servlets is that the JSP allows a logical division between what is displayed [i.e. the generated HTML] and the Web server side code spec that dictates what content fills the page. It is easy to modify the look and feel of what is delivered by a JSP without having to alter any Web server side, Java code spec.

While it's true that anything done with a JSP can also be done with using a Servlet, JSPs provide a nice clean separation of the application's presentation layer from its data manipulating layer. JSP's are simpler to craft than Servlets. Servlets and JSPs work well together.

JSP v/s ASP

JSP and ASP programming environments offer developers identical features. Both JSP and ASP use tags to allow embed their code within an HTML page such as session tracking, database connections and so on.

The following are the major differences between JSP and ASP:

Platform Independent

ASP are written in VBScript or JScript. Therefore, ASP are not platform independent. JSP are written in Java programming language. Hence, JSP are platform independent.

Components

ASP use ActiveX components. JSP use Java Beans technology as the component architecture.

Compilation

JSP are compiled the first time they are requested leading to faster page loading for subsequent requests, while ASP have to be translated each and every time. Thus, this gives the advantage of speed and scalability to JSP compared to ASP.

Extensible Tags

JSP have an advanced feature called extensible tags, which enables developers to create custom tags. In other words, extensible tag allows extending the JSP tag syntax. This is not possible in ASP.

Platform Compatibility

Although third-party vendor, ChiliSoft, offers an ASP implementation available for other platforms, ASP works only with Microsoft's **IIS Web server** or **Personal Web Server**. In other words, ASP requires a commitment to Microsoft's products.

JSP is available on any Web server and platform that support Java Servlets since the JSP engine itself is written entirely in Java. Therefore, in today's world, JSP pages are becoming a widely supported standard.

JSP Architecture

There are two basic ways of using JSP technology. They are:

- ❑ Page-Centric or Client/Server approach
- ❑ Dispatcher or N-tier approach

The Page-Centric Approach

Applications built using the client-server approach consist of one [or more] application programs running on client machine, connecting to a single common server-based application. With the arrival of Servlet technology such 2-tier applications can be developed using Java. This model allows JSPs or Servlets direct access to resources such as databases or legacy application code spec to service a client's request.

The JSP page is where the incoming request is intercepted, processed and an appropriate response sent back to the client. JSPs differ from Servlets in this scenario by empowering programmers to separate code spec that displays page content from code spec that manipulates application data. Code spec that manipulates application data is done on Web server side using Beans or EJBs.

Though this model makes application development well structured, it does not scale up well. When there are a large number of simultaneous client requests received at the Web server a significant amount of request processing must be performed. Each request must establish or share a potentially scarce/expensive connection to a common resource such as a database. Thus server response takes a hit.

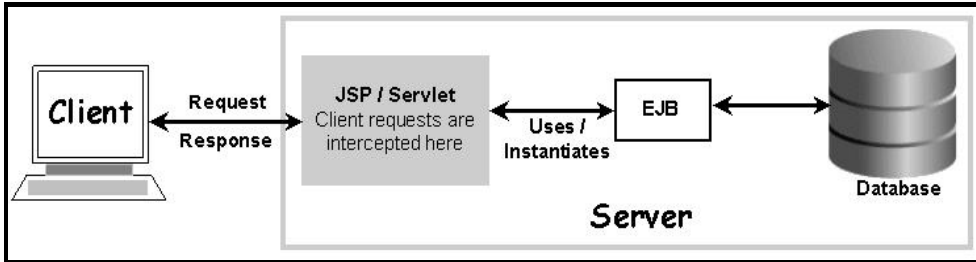


Diagram 4.1: Page-Centric Approach

Page View Architecture

This architecture involves direct requests being made to a Web server page with embedded Java code and markup tags. Java code spec determines which content must be delivered and the HTML markup elements determine the look and feel of the content. This is a straightforward approach with many benefits. All Java code spec may be embedded within HTML, so changes are confined to a very limited area, reducing complexity drastically.

The big trade-off here is in the level of sophistication. As the scale of the application grows, limitations begin to surface such as bloating of business logic code within the page, which is instead of moving the business logic to a Servlet or a worker bean. It is a fact that utilizing a Servlet and/or helper beans helps to separate the applications GUI from its business logic more cleanly and improves the potential for code reuse.

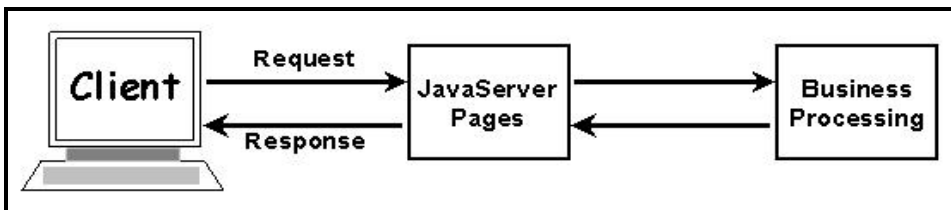


Diagram 4.2: Page View Architecture

Page View With Bean Architecture

This pattern is used when the Java and Markup code spec architecture becomes too cluttered with business-related code and data access code.

The Java code spec, which represents business logic and data storage implemented in the earlier model is moved from the JSP to the worker beans. This restructuring leaves a much cleaner JSP with limited Java code, which can be comfortably crafted by GUI designers, since it encapsulates mostly markup tags.

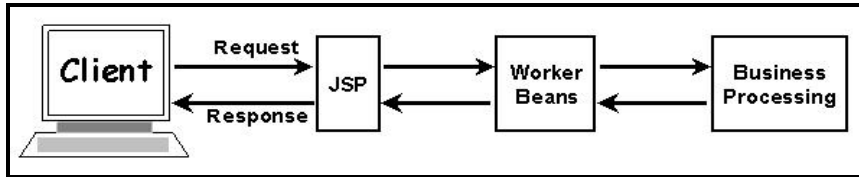


Diagram 4.3: Page View With Bean Architecture

The Dispatcher Approach

In this approach, a Servlet/JSP acts as a controller, which delegates requests to other JSP pages and Java Beans. Three different architectures are commonly used with this approach:

- ❑ Mediator-view
- ❑ Mediator-composite view
- ❑ Service to workers

The web server side architecture is packed into multiple tiers in an N-tier application. But in the dispatcher approach, the web server side application consists of multiple tiers. Here, the JSP interacts with the back end resources via another object or EJB component.

Enterprise Java Beans provide managed access to multiple resources, support transactions and access to underlying security mechanisms, thus helping eliminate the resource sharing and performance issues of the 2-tier approach.

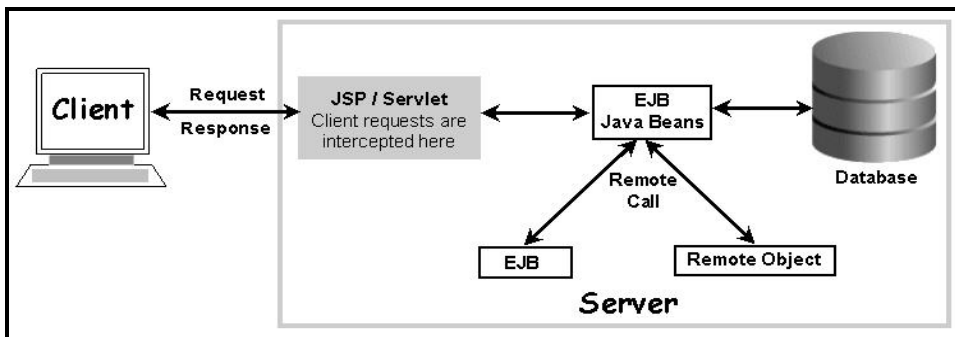


Diagram 4.4: The Dispatcher Approach

The first step in N-tiered application design is identifying the correct objects and their interaction. The second step is identifying the JSPs or Servlets that will act as the front end of the Web server application that interacts with the next tiers. One approach would be:

- ❑ **Front end** JSPs or Servlets that manage application flow and business logic evaluation. They act as an entry point that intercepts HTTP requests coming from users. This single entry point to an application, simplifies security management thus making application state relatively simple to maintain
- ❑ **Presentation** JSPs or Servlets generate HTML or XML. Their main purpose in the application being the presentation of dynamic content. They contain only presentation and presentation markup logic

Life Cycle Of A JSP Page

Since JSP architecture is based on Servlet architecture, a **Java enabled** Web server provides the mechanism that deals with both. Individual JSPs are actually text files stored on the Web server and accessed via their path.

For example: if a JSP page named main.jsp resides at the root of a web application named **sample**, it would be accessed by a request to http://localhost:8080/sample/main.jsp, assuming that the Web server is bound to port 8080 on localhost, when the JSP page is requested.

The Web server's JSP engine uses the content of the JSP file to generate the required Servlet's source code. The generated source is then compiled and run by the Web server's Servlet engine. Servlet code execution services a client's request.

Once JSP code spec has been converted to a Servlet and Servlet code spec has been compiled, the compiled version is saved and used to service additional client requests according to the standard Servlet life cycle.

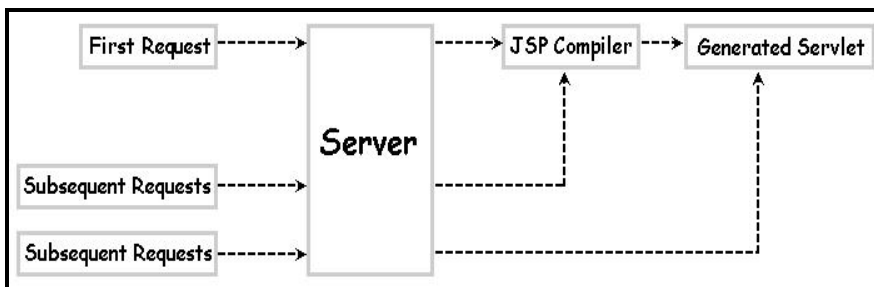


Diagram 4.5: Life cycle of a JSP page

Whenever the JSP file changes, the Web server automatically detects the change and rebuilds the corresponding Servlet. The JSP to Servlet compilation phase imposes a slight delay the first time a page is retrieved. Many web servers permit pre-compilation of JSPs to get around this problem.

There are 5 phases in the JSP lifecycle, these are as follows:

1. **Compilation Phase:** When a JSP page is requested by a client Browser, the JSP engine first checks whether the JSP page requested needs to be compiled.

- If the JSP page is not compiled

OR

If the JSP page has been modified since it was last compiled,

- The JSP engine recompiles the page.

The compilation process of a JSP page involves:

- Parsing the JSP page
- Translating the JSP page into a Servlet
- Compiling the Servlet thus generated

2. **Loading Phase:** After the compilation process completes, the JSP engine loads the generated class file into the JVM and creates an instance of the Servlet. This is done so that the initial request from the client Browser can continue being processed
3. **Initialization Phase:** Although the JSP is compiled into a Servlet, the `init()` method of the Servlet is not overridden. Instead the JSP specific initialization method `jspInit()` is overridden

The `jspInit()` method is automatically invoked by the JVM when the JSP page is initialized. This method can be implemented by a programmer. This provides appropriate initialization code spec for the JSP. This method is automatically generated during the JSP compilation process. The `jspInit()` method is called exactly once and is used for initializing a database connection and so on

Syntax:

```
public void jspInit()
```

4. **Execution Phase:** Once the JSP page is initialized `_jspService()` is invoked. `_jspService()` handles a request and returns a response to the client Browser

JSP Scriptlets and Expressions end up inside this method since they apply to a single request. JSP Declarations and Directives are not included inside this method since they apply to the entire page

Syntax:

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

5. **Destruction Phase:** When the execution phase completes, its time to clean up memory. `jspDestroy()` does all cleaning up such as releasing database connections, closing all open files and so on. JSP's `jspDestroy()` is equivalent to the Servlet's `destroy()`. Once this method is called, the JSP page can no longer serve any requests

Syntax:

```
public void jspDestroy()
```

How Does A JSP Function

JSP page code spec can be broken into 2 categories:

- ❑ **Elements** that are processed by the JSP engine on the Web server
- ❑ **Template data** or everything other than such elements, that the JSP engine **ignores**

Examining JSP execution will provide an insight into the elements contained in the JSP and what they actually do.

A JSP page is executed by a Web/Application server that either has a built in **JSP engine** or accesses a third party JSP engine, which it is configured to use. When a client asks for a JSP resource the Web server fields that request and delivers it to the JSP engine along with a RESPONSE object.

The JSP engine uses the JSP code spec referenced by the client to process the client's request, obtain whatever is required from associated resources [i.e. database table or a test file] mapped to the client's request, formats and delivers this throughput back to the Web Server for onward delivery to the client.

REMINDER



Keep in mind that the underlying layer of a JSP is similar to that of a Servlet.

JSP uses **HttpServletRequest** and **HttpServletResponse** for the HTTP protocol similar to that of a Servlet.

To see how a JSP works and understand its lifecycle take a look at a simple JSP.

Create a simple HTML form, **Example.html**, which allows a user to type in a number. **Example.html** submits the number captured to the Web Server when the user clicks on the form's **Submit** button.

The Web server executes **Example** JSP code spec and responds with an HTML page with **Hello, World!** repeated as many times as specified.

The HTML page also displays some information at the bottom about the functionality of the Web server side program that handled the information returned by the HTML page.

Code spec for Example.html:

```
<HTML>
  <HEAD>
    <TITLE>JSP Example</TITLE>
  </HEAD>
  <BODY>
    <P>How many times? </P>
    <FORM METHOD="GET" ACTION="/MyWebApplication/MyJSP/Example">
      <INPUT TYPE="TEXT" SIZE="2" Name="numtimes">
      <INPUT TYPE="SUBMIT" VALUE="Submit">
    </FORM>
  </BODY>
</HTML>
```

When viewed in a Browser the HTML page looks as shown in diagram 4.6

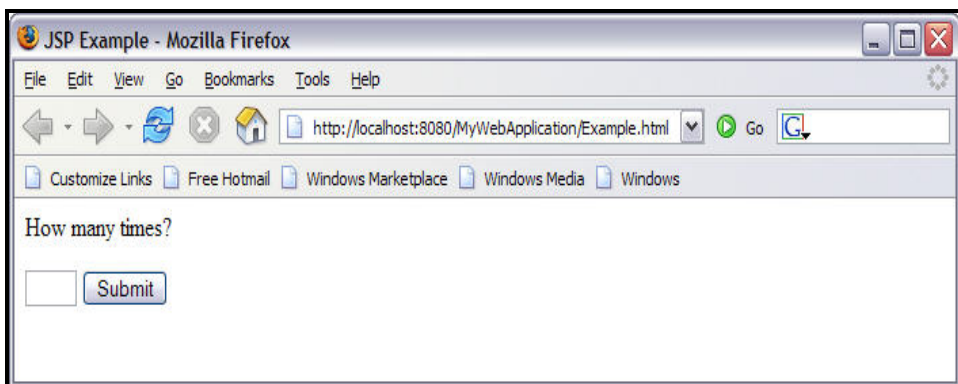


Diagram 4.6: The Example.html form

When the user clicks **Submit**, the browser dispatches the data captured by the form as a request to the JSP page [Example] resident on the Web server for further processing.

60 Java Server Pages For Beginners

The Web server accepts the data returned from the Browser and passes it as a parameter to the JSP code spec name **Example**, which creates the **response** HTML code, to be sent back to the Browser via the Web Server.

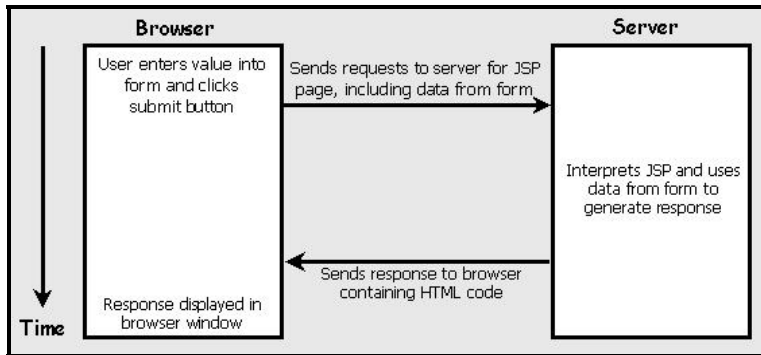


Diagram 4.7

The response from the Web server, displayed in the user Browser is as shown in diagram 4.8

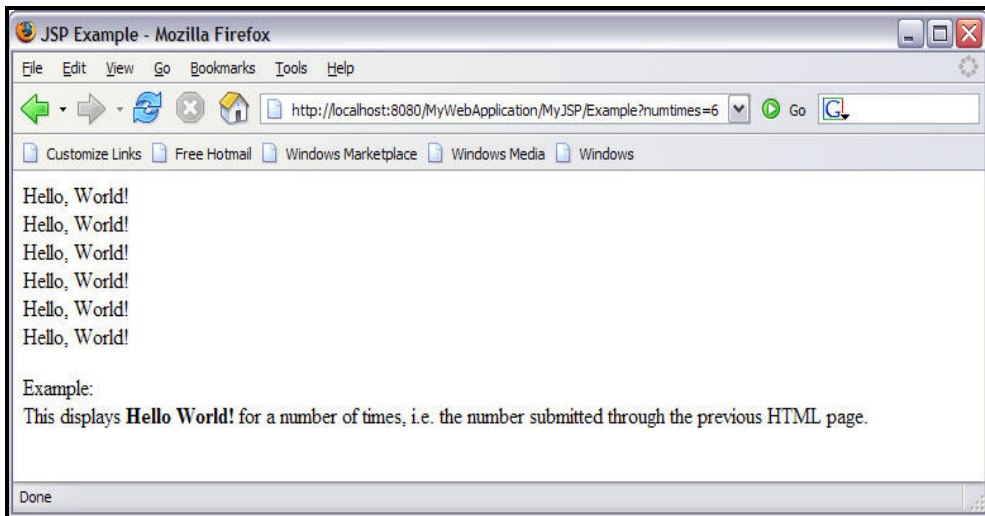


Diagram 4.8: The response from the Example.jsp file

Code spec for Example.jsp:

```
<%@ page language="java" %>
<HTML>
<HEAD>
  <TITLE>JSP Example</TITLE>
</HEAD>
```

```

<BODY>
  <P>
    <!-- <%
      int numTimes = Integer.parseInt(request.getParameter("numtimes"));
      for(int i = 0; i < numTimes; i++)
      {
        %> -->
        Hello, World! <BR>
      <!-- <%
        }
      %> -->
    </P>
    <%@ include file="Note.html" %>
  </BODY>
</HTML>

```

Explanation:

Most of `Example.jsp` is plain HTML, interspersed with special JSP tags:

- ❑ `<%@ page language="java" %>`: This is a JSP **directive** [denoted by `<%@`]. Hence the JSP engine recognizes that the code on the page is written in Java
- ❑ The next highlighted block shows a pair of scriptlets [enclosed by the `<% ... %>` tags] on either side of plain HTML code. The scriptlet encloses Java code, which accepts and processes the value entered by the user on the form received from the Web Server's **request** object and converts it into an integer value. [*For simplicity, this code does not handle errors*]

The code then executes a loop, which outputs the HTML code [Hello World! `
`] the number of times the loop executes.

Notice that, this HTML code is not included inside the `<% ... %>` tag. It is not to be processed itself, but passed straight to the Web Server for onward delivery to the client Browser.

- ❑ `<%@ include file="Note.html"%>`: This **directive** includes the contents of the file `Note.html` at this point in the HTML page being delivered to the user Browser, by the Web Server

REMINDER

The contents of the file `Note.html` is as follows:

```
<P>
```

```
  Example:<BR>
```

```
  This displays <B> Hello World! </B> for a number of times, i.e. the number
  submitted through the previous HTML page.
```

```
</P>
```

62 Java Server Pages For Beginners

That's how this simple JSP page executes. Assume that the value **6** has been entered in the textbox appearing in the HTML page named Example.html.

Then the HTML code that is sent back to the Browser is shown below. By comparing it with the contents of Example.jsp itself, it can be seen how the JSP tags have been interpreted and replaced in the response being delivered to the user Browser via the Web Server:

```
<HTML>
<HEAD>
  <TITLE>JSP Example</TITLE>
</HEAD>
<BODY>
  <P>
    Hello, World!<BR>
    Hello, World!<BR>
    Hello, World!<BR>
    Hello, World!<BR>
    Hello, World!<BR>
    Hello, World!<BR>
  </P>
  <P>
    Example: <BR>
    This displays <B> Hello World! </B> for a number of times, i.e. the number
    submitted through the previous HTML page.
  </P>
</BODY>
</HTML>
```

How Does JSP Execute

The following happens when a user Browser requests **Example.jsp**.

The browser sends its request to the Web server as:

```
http://localhost:8080/MyWebApplication/MyJSP/Example.jsp?numtimes=6
```

This specifies the value of **numtimes** [the number entered into the form by the user] as a GET parameter.

The Web server recognizes the Example.jsp file in the URL sent in by the Browser. The Web server recognizes Example.jsp as a Java Server Page by its extension and that the information delivered by the Browser encoded in the URL must be passed on to Example.jsp.

Example.jsp is then translated into a Java class by the JSP engine, which is a child to the Web server, by having its embedded Java code compiled into a Servlet. This translation and compilation phase occurs **only when** the JSP file is **first called** [or it is subsequently changed]. Hence, there is a slight delay the first time that **Example.jsp** is run.

REMINDER



For every subsequent request for that JSP page thereafter, there is no delay because the request is forwarded directly to the Servlet byte code already in memory.

When the Servlet is run by the JVM its **init()** method is called first. The code spec of **init()** performs the global initializations used by every request of the Servlet.

Then individual requests are sent to the Servlet's **service()** method where their responses are assembled.

service() spawns the `HttpServletRequest` and `HttpServletResponse` objects. User data returned by the Web page is retrieved by the Servlets `HttpServletRequest` object's `getParameter()`.

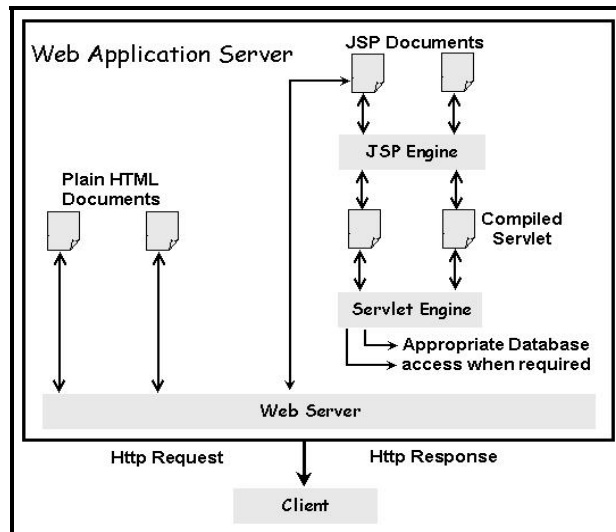


Diagram 4.9

Features Of JSP 2.1

Following are the new features that every web-application developer should be familiar with to get the most out of the JSP 2.1. The following are an overview of the new features introduced in version 2.1 of JSP technology:

The Unified Expression Language [EL]

The Expression Language [EL] included in JSP 2.0 technology offers page authors multiple advantages. Using simple expressions, page authors can conveniently access external data objects from their JSP pages.

Subsequently, JSP created a more powerful EL, which is called the **Unified EL** that supports the following features:

- ❑ Deferred expressions are expressions that will be evaluated at different stages of the JSP page life cycle
- ❑ Expressions that set the data of external objects as well as get data of external objects
- ❑ Method expressions, which invoke methods that perform event handling, validation and other functions
- ❑ A flexible mechanism that permits customization of variable and property resolution for the evaluation of an EL expression

REMINDER



The EL is useful beyond all of the web technology specifications. This is why the EL is independent of the technology hosting it. EL is currently defined through its own independent document within the JSP specification. Thus, the EL is not dependent on the JSP specifications and might therefore have its own JSR in the foreseeable future.

Better Alignment With JSF

The main focus of JSP 2.1 technology is to provide strong alignment with the next release of Java Server Faces [JSF] technology i.e. version 1.2

The misalignment between these two technologies started with the fact that JSF technology version 1.0 depended on JSP technology 1.2. JSP 1.2 software was already widely available at the time and the intention was to make the JSF 1.0 interface accessible to a broader audience. A consequence of this requirement was that JSF technology could not take advantage of the

EL introduced in JSP technology version 2.0. Neither could JSP 2.0 technology be modified to accommodate the needs of JSF technology.

JSF 1.2 did not support an EL. Hence JSF technology introduced an EL that was suited to its needs as a **User Interface [UI]** component framework. As a result, page authors using JSF technology tags with JSP code have encountered incompatibilities between the two code blocks.

The expert groups have worked together on the upcoming releases of JSP 2.1 and JSF 1.2 technologies in Java EE 5 to fix such integration issues and have made sure that the two technologies now work together in harmony. One result is that all web-tier technologies now share a Unified EL, which allows mixing code from all of these technologies freely and compatibly.

Injection Annotations

Prior to Java EE 5, accessing data sources, web services, environment entries and Enterprise Java Beans [EJB] bean references required the use of the **Java Naming and Directory Interface [JNDI]** API, along with the declaration of entries in a deployment descriptor.

Because of injection annotations, it is now possible to inject container-managed objects without having to write traditional boilerplate code and deal with deployment descriptors.

Example:

A tag handler could access a database using the following code

```
@Resource(name="jdbc/myDB")
javax.sql.DataSource mydata;
public getInventory()
{
    // Getting a connection and executing the query.
    Connection conn = mydata.getConnection();
    ...
}
```

Explanation:

The field mydata of the type javax.sql.DataSource is annotated using **@Resource** and is injected by the server prior to the tag handler being made available to the application. The data source JNDI mapping is inferred from the field name and type javax.sql.DataSource. Moreover, the mydata resource no longer needs to be defined in the deployment descriptor.

Blank Lines Removed

Often the output of a JSP page is cluttered by blank lines or white spaces. This is a universal complaint on user forums.

Example:

```
<%@page contentType="text/html"%>
<%@page trimDirectiveWhitespaces="false"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<HTML>
  <HEAD>
    <TITLE>Trimming</TITLE>
  </HEAD>
  <BODY>
    <c:forEach items="${books.items}" var="bookname" varStatus="stat">
      ${bookname} <BR>
    </c:forEach>
  </BODY>
</HTML>
```

Prior to JSP 2.1 technology, the above JSP code spec would produce the following output: [where ... indicates a blank line]

```
...
...
<HTML>
  <HEAD>
    <TITLE>Trimming</TITLE>
  </HEAD>
  <BODY>
    ...
    Ajax For Beginners<BR>
    ...
    PHP 5.1 For Beginners<BR>
    ...
    Visual Basic 2005 For Beginners<BR>
    ...
  </BODY>
</HTML>
```

Although these blank lines do not change the way a Browser displays the output page, they are part of the HTTP data stream from Web server to Browser and add to the download time taken for the HTML page. This also makes the HTML source code difficult to read. JSP 2.1 now offers the configuration parameter **trimDirectiveWhitespaces**, which resolves this issue. This can be specified as a JSP directive or as a property group configuration parameter that applies to a group of pages.

When the `trimDirectiveWhitespaces` is enabled, template text containing only blank lines or white spaces is removed from the HTML response output prior its delivery to the Browser.

Example:

The lines containing blank lines are removed and the output looks like:

```
<HTML>
  <HEAD>
    <TITLE>Trimming</TITLE>
  </HEAD>
  <BODY>
    Ajax For Beginners<BR>
    PHP 5.1 For Beginners<BR>
    Visual Basic 2005 For Beginners<BR>
  </BODY>
</HTML>
```

Backward Compatibility

All Java EE platform releases maintain backward compatibility with earlier versions. For JSP 2.1, EL's new support of the syntax `#{`, which refers to deferred expressions is the only backward compatibility issue. The `#{` character sequence does create backward compatibility problems in two situations:

- ❑ When the sequence is included in template text

OR

- ❑ In attribute values

In JSP 2.1 technology, a deferred expression does not make any sense if included in the context of template text, because evaluation of a deferred expression is always done immediately. Because no entity other than the JSP server processes template text, a deferred expression would always be **left unevaluated** if it were included in template text.

Therefore, any JSP page authors who include the `#{` character sequence in template text probably meant to use `${}` syntax. Acknowledging this fact, JSP specification authors have made it mandatory that the inclusion of `#{` in template text, triggers a translation error in JSP 2.1 technology.

With respect to tag attribute values, tag libraries based on JSP versions prior to 2.1 such as the 1.1 version of the JSF component tag library made extensive use of the `#{` syntax to specify deferred expressions. These tag libraries are responsible for parsing deferred expressions, hence they must execute in an environment in which the `#{` character sequence is processed as a string literal.

To determine whether or not to treat the `#{` character sequence as a string literal, the JSP server relies on the JSP version indicated by the Tag Library Descriptor [TLD] associated with the tag library. The JSP version specified in the TLD tells the JSP server which version of the JSP specification the tag library is written for.

JSP versions previous to 2.1 processed `#{` as a string literal. With version 2.1 or later, the character sequence `#{` represents a deferred expression, assuming that the attribute has been declared to support deferred expressions in the TLD. If the attribute does not support deferred expressions, then the presence of the `#{` character sequence results in a translation error.

Web application developers who developed applications that run properly in a JSP version **prior** to JSP 2.1 technology, may now have to deploy the same application and run it in a JSP 2.1 server. Then they must do the following to ensure that the JSP server handles all instances of the `#{` character sequence correctly:

- ❑ Escape each instance of the `#{` characters using a backslash, `\#{`
- ❑ Globally permit usage of the `#{` character sequence as a string literal by setting the JSP property groups element **deferred-syntax-allowed-as-literal** to **TRUE** or by setting the page/tag-file directive attribute **deferredSyntaxAllowedAsStringLiteral** to **TRUE**

Aside from the incompatibilities relating to the `#{` character sequence, all JSP 2.0 applications will run as is in JSP 2.1 servers.

However, if a web application that uses third-party tag libraries that are based on JSF 1.1 technology or earlier, some new features provided by JSP 2.1 and JSF 1.2 technologies will not be available.