

# The Anatomy of a JavaServer Page

The Java 2 Platform, Enterprise Edition (J2EE) has two different but complementary technologies for producing dynamic web content in the presentation tier—namely **Java Servlets** and **JavaServer Pages (JSP)**.

Java Servlets was the first of these technologies to appear and it was initially described as extensions to a web server for producing dynamic web content. JSP on the other hand is a newer technology, but is equally capable of generating the same dynamic content. However, the way in which a servlet and a JSP page produce their content is fundamentally different; servlets embed content into logic whereas JSP pages embed logic into content.

JSP pages contain markup interlaced with special JSP elements that provide logic for controlling the dynamic content. Servlets are built using Java classes that contain statements to output markup code. Of these two different paradigms, JSP pages are preferred for presenting dynamic content in the presentation tier due to their greater readability, maintainability, and simplicity. Further increasing the simplicity and ease of use of JSP pages was one of the main objectives of the JSP 2.0 specification, which includes several new features to make it easier than ever to embrace JSP technology, especially for developers who aren't fluent in the Java syntax.

The inclusion of a new **expression language (EL)** enables JavaScript-style JSP code to be embedded within pages, which makes it much easier for web developers not familiar with the Java syntax to understand the JSP logic. A library of standard actions known as the **JavaServer Pages Standard Tag Library (JSTL)** is also included to provide a host of useful, reusable actions such as conditional statements, iteration, and XML integration to name a few. These actions are applicable in some shape or form to most JSP web applications, and their use will greatly improve the reliability and ease of development for JSP page authors. Custom actions (also known as custom tags) also benefit from changes in the JSP 2.0 specification and it's now possible to write a custom action entirely in JSP syntax instead of Java syntax! These new features will help make JSP pages easier to write and maintain and are discussed in detail in the following chapters:

- ❑ The JSP 2.0 expression language (EL) (see Chapter 3)
- ❑ The JavaServer Pages Standard Tag Library (JSTL) (see Chapter 4)
- ❑ JSP 2.0 Custom Tags (see Chapters 5, 6, and 7)

In this chapter you'll take a look at some of the fundamental concepts based around JSP technology, such as:

- ❑ The mechanics of a JSP page
- ❑ Typical JSP architectures
- ❑ Core JSP syntax
- ❑ Tag libraries

The aim of this chapter is for you to gain a grounding in the basics of JSP technology to help you to make full use of the rest of the chapters in this book that build on these basic principles.

## Before You Begin

To begin examining the basics of JSP technology it's essential that you have a cursory familiarity with the alternative and complimentary presentation tier web component, Java Servlets. The next chapter will discuss servlets in more detail.

### Java Servlets

As mentioned earlier, a servlet can most simply be described as custom web-server extensions, whose job are to process requests and dynamically construct appropriate responses. In practice such responses are usually returned in the form of HTML or XML and are the result of a user making an HTTP request via a web browser. Servlet technology has been an extremely popular choice for building dynamic web applications such as e-commerce sites, online banking, and news portals to name a few, for reasons of simplicity, extensibility, efficiency, and performance over alternative technologies such as CGI scripts.

Some of the most basic advantages of servlet technology are as follows:

- ❑ **Simplicity:** Servlets are easy to write, and all the complicated threading and request delegating is managed by the servlet container.
- ❑ **Extensibility:** The Servlet API is completely protocol independent.
- ❑ **Efficiency:** Unlike CGI scripts the execution of a servlet doesn't require a separate process to be spawned by the web server each time.
- ❑ **Performance:** Servlets are persistent and their life cycle extends beyond that of each HTTP request.

Servlets are simply Java classes that inherit from the `javax.servlet.Servlet` interface, which are compiled and deployed inside of a servlet container, which is a Java environment that manages the life cycle of the servlet and deals with the lower-level socket-based communication. The servlet container may be part of an existing Java-enabled web server itself or may be used as a stand-alone product that is integrated with a third-party web server. The servlet Reference Implementation container, Jakarta Tomcat for example, may

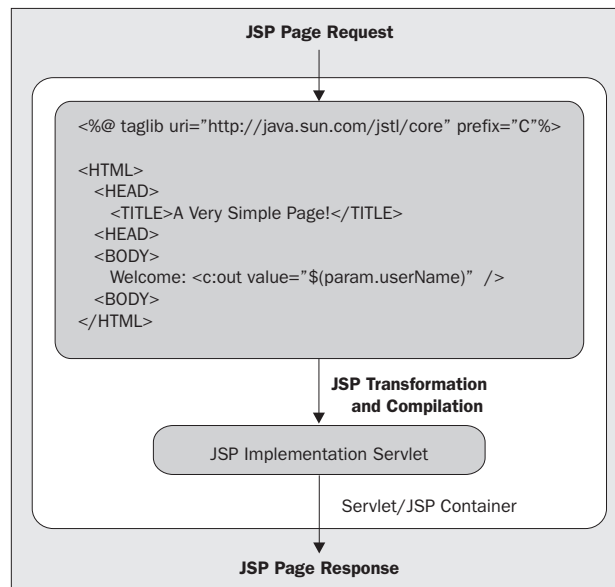
be used as a stand-alone web server or can be used as a separate servlet container inside a larger commercial web server such as the Apache web server.

Servlets are typically used for returning text-based content such as HTML, XML, WML, and so on, but are equally at home returning binary data such as images or serialized Java objects, which are often used by further servlets to generate some appropriate dynamic response.

## JSP Under the Hood

A JSP page is simply a regular text file that contains markup (usually HTML) suitable for display inside a browser. Within this markup are special JSP elements that you'll learn more about later. These are used to provide processing logic that enables dynamic content to be produced on a request-by-request basis.

In JSP terms, any markup that isn't a JSP element is known as template text, and this really can be any form of text-based content such as HTML, WML, XML, or even plain text! Of course the mixture of JSP elements and template text cannot simply be sent to the browser without any form of processing by the server. We mentioned earlier how JSP technology is an extension of servlet technology, and so you probably won't be surprised to learn that each JSP page is, in fact, converted into a servlet in order to provide this processing logic. This servlet is known as the **JSP implementation servlet**.

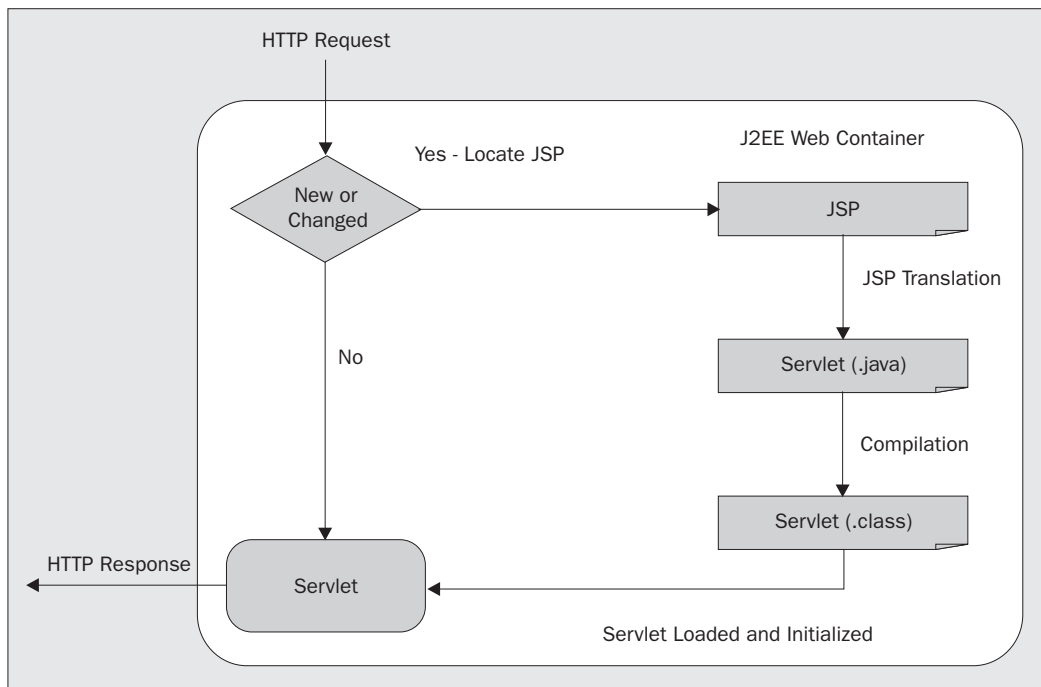


A request for a JSP page is handled initially by the web server, which then delegates the request to the JSP container. The JSP engine will translate the contents of the JSP into its implementation servlet, which it uses to service the request. Usually a JSP container will check to see if the contents of a JSP page have changed before deciding if it needs to retranslate the page in response to a request. This feature can make on-the-spot changes to JSP pages easy since the next request will automatically cause a retranslation and the most up-to-date content will be returned. Compare this with a purely servlet-based approach, which would need the servlet container to be shut down in order to have the necessary changes made, such as recompilation, testing, and finally, a restart!

Let's take a closer look at the process involved in taking a plain JSP text file and turning it into a dynamic web component, also known as the JSP life cycle.

## The JSP Life Cycle

As you've just seen, JSP pages don't directly return content to the client browser themselves. Instead, they rely on some initial server-side processing that converts the JSP page into the JSP page implementation class, which handles all requests made of the JSP:



As you can see from this diagram, the JSP servlet container decides whether or not the JSP has been translated before. If not, then the JSP container starts the translation phase to generate the JSP page implementation servlet, which is then compiled, loaded, and initialized and used to service the request. If the JSP container detects that a JSP has already been translated and hasn't subsequently changed then the request is simply serviced by the implementation servlet that exists already inside the container.

The life cycle of a JSP can be further split into approximately four phases to cater for the initial JSP translation right through to the servicing of requests and ultimately to the death of the servlet.

### **Translation**

The first stage in the life cycle of a JSP is known as the translation phase.

When a request is first made for a JSP (assuming it hasn't been precompiled), the JSP engine will examine the JSP file to check that it's correctly formed and the JSP syntax is correct. If the syntax check is successful then the JSP engine will translate the JSP into its page implementation class, which takes the form of a standard Java servlet. Once the page's implementation servlet has been created it will be compiled into a class file by the JSP engine and will be ready for use.

Each time a container receives a request, it first checks to see if the JSP file has changed since it was last translated. If it has, it's retranslated so that the response is always generated by the most up-to-date implementation of the JSP file.

### **Initialization**

Once the translation phase has been completed, the JSP engine will need to load the generated class file and create an instance of the servlet in order to continue processing of the initial request. Therefore, the JSP engine works very closely with the servlet container and the JSP page implementation servlet and will typically load a single instance of the servlet into memory.

As you may or may not be aware, the Java Servlet Specification provides two separate threading models that can be used. The models determine whether single or multiple instances of a servlet can exist. As with servlets, the default threading model is the multithreaded one that requires no additional work for the developer. To select the single-threaded model for your JSP, there's an attribute of the page directive called `isThreadSafe` that must be set to `false` to serialize all requests to the implementation servlet behind the JSP:

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

Of course such directives must be placed in the JSP so that the JSP engine can make the necessary changes to the page implementation servlet during the translation phase.

### **Servicing**

Once the JSP and servlet containers have completed all of their preliminary work, the initial request can be serviced. There are three important methods that are generated in the page implementation servlet by the JSP engine during the translation phase. These three methods bear a striking resemblance to the `init()`, `service()`, and `destroy()` methods from the `javax.servlet.Servlet` interface. They're the key methods involved in the life cycle of a servlet (as you'll see in Chapter 2).

Each method is explained in greater detail in the following list:

#### □ `jspInit()`

As the name suggests, this method is used for initializing the implementation servlet in an identical manner to the standard servlet `init()` method, which is used to initialize a servlet. The behavior of both methods can be regarded as identical and each is called exactly once. Although this method is automatically generated during the translation phase, it's possible to override this method in the JSP using a declaration. It can be used for initializing the JSP in order to open a database connection or initialize some session and context variables, for example.

```
<%! Connection conn = null; %>
<%!
    public void jspInit() {
        try {
            conn = getConnection(...);
        } catch (SQLException sqle){}
    }
%>
```

#### □ `_jspService()`

This method provides all of the functionality for handling a request and returning a response to the client. All of the scriptlets and expressions end up inside this method, in the order in which they were declared inside the JSP. Notice that JSP declarations and directives aren't included inside this method because they apply to the entire page, not just to a single request, and therefore exist outside the method. `_jspService()` may not be overridden in the JSP.

## **Destruction**

The final method worthy of explanation that is generated as a result of the translation phase is the `jspDestroy()` method. Like the `destroy()` method found in a normal servlet, this method is called by the servlet container when the page implementation servlet is about to be destroyed. This destruction could be for various reasons, such as the server being low on memory and wanting to free up some resources, but the most common reason would be when the servlet container is shutting down or being restarted.

Once this method has been called, the servlet can no longer serve any requests. Like the `destroy()` method, `jspDestroy()` is an excellent place to release or close resources such as database connections when the servlet is shutting down. To do this, simply provide an implementation of this method via a JSP method declaration. For example, to close the database connection you opened inside the `jspInit()` method, you would use the following:

```
<%!
    public void jspDestroy() {
        try {
            conn.close();
        } catch (SQLException sqle){}
        conn = null;
    }
%>
```

# JavaServer Pages Best Practices

One of the design goals of this book apart from the obvious introduction to the concepts and mechanics of JSP technology was to teach the best practices learned from experience right from the start. Of all the best practices that have been established around JSP, one of the most important suggests that there should be as little Java code embedded inside a JSP as possible. Experience has shown us that there are three key factors that benefit from this practice:

- ❑ Reusability
- ❑ Readability
- ❑ Maintainability

Let's look at each of these in turn and see how their use can benefit your JSP applications.

## Reusability

A common goal associated with using any programming language is that of reuse, whether it involves structuring code inside modules, classes, or some other language-specific construct. Reusing code leads to increased maintainability and productivity, and higher quality since changes to such common functionality only need to be made in a single place. Although the concept of building web-based applications is relatively new, this goal applies equally to building Java-based web applications with JSP.

Web-based applications are typically built up around the pages or screens from which the application is comprised. For example, in an online bookstore, you might build the welcome page first, followed by the page that shows a list of books and then a page that displays the information about a single book. With the ability to embed Java code inside JSP pages, there can be a tendency to simply reuse code on a source-code level by copying and pasting it between JSP pages. While this does achieve some reuse, it brings with it a dramatic decrease in the maintainability of such code as changes and bugs slowly creep in and around the system. Ideally you're looking for reusability at the class or component level.

Throughout this book you'll see many techniques for aiding reusability provided by the JSP specification such as JavaBeans components, custom tags, and tag libraries. A **tag library** (commonly known as a **taglib**) is simply a collection of one or more custom tags that are generally related in some way. For example, the JSP 2.0 specification for the first time includes a standard tag library known as the JSTL. The JSTL's core library contains tags that solve many of the common and recurring problems encountered when building JSP-based web applications. Once the tags are bundled up into a tag library, that tag library can be reused across the following:

- ❑ A single page
- ❑ The pages of a web application
- ❑ Different web applications

The ability to easily reuse custom tags across more than a single page illustrates the true potential of tag libraries to be used as reusable components when building web applications. This is something that you'll be seeing when you examine the best practices for designing and building custom tags in later chapters.

## Readability

Another important best practice is that of readability. Embedding too much Java code in the page can easily lead to pages that are unreadable as content (typically HTML) is mixed with JSP tags and Java code wrapped up as scriptlets. In addition to the confusion caused by the various syntaxes that each of these “languages” uses, one clear problem with embedding Java code inside JSP pages is that it’s hard to correctly indent your source code. Writing and indenting code is trivial when dealing with regular class files, but trying to correctly indent Java code that is mixed up with HTML and JSP is a different story.

Wrapping up reusable functionality as custom tags or JavaBean components removes this code from the page, therefore making it cleaner, shorter, and more readable. Also, choosing appropriate names for your custom tags can also make a page more readable by page designers—those people that are responsible for the look and feel of a page rather than the mechanics of how it works. This, as you’ll be seeing when we talk about some of the best practices associated with custom tags, is very important and often overlooked.

## Maintainability

Having a system that promotes reusability and readability is great, but what does that mean in the real world? The maintainability of an application is how well the system can be modified and fixed further on during its lifetime, which for a given application is typically hard to measure. However, in looking at any system, there are several signs that help us to identify whether or not that system will be easy or difficult to maintain. In reality, this is dictated by reuse and the need to ensure that the code is as readable as possible—the two goals that custom tags can help you achieve.

# JavaServer Pages Application Architecture

All of these factors mentioned in this chapter are improved by a good design or architecture, therefore it’s worth ensuring that sufficient time is taken early on in your project life cycle to select the architecture that best suits your environment and technologies. Although architecture and design can seem a little daunting at first (especially when you’re new to the technology and all you really want to do is cut code), with a little effort you’ll soon start to understand the benefits that are to be gained by using tried and tested patterns.

As you’re no doubt aware, the J2EE presentation tier consists of several different components, which may all be used to create a presentation layer for a web application, such as servlets, JSP pages, tag libraries, and JavaBeans to name but a few. All of these components have their relative strengths and weaknesses, particularly when used in different combinations. Good design is therefore not only concerned with selecting the correct component for a task but is also concerned with ensuring that the component is used in the correct manner.

In recent times there have been two popular web application architectures that have been repeatedly used for web application design, and there are strengths and weaknesses to consider with both. Let’s discuss the simpler of these two architectures first.

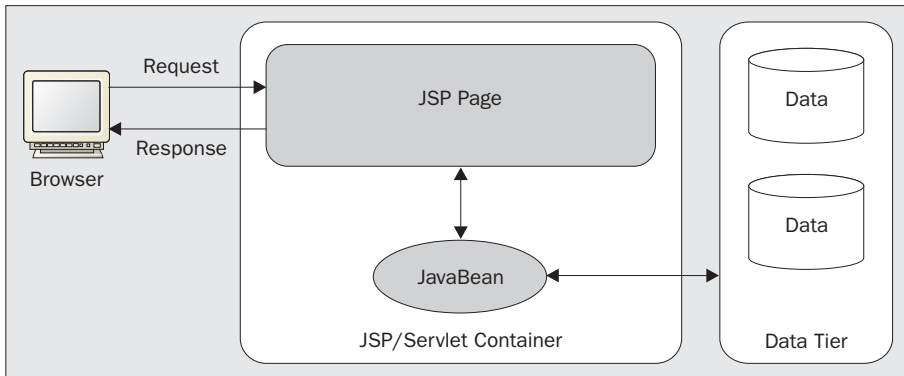
## Model 1 Architecture

The JSP model 1 architecture really is very simple and advocates a web application, thereby providing a number of JSP pages with which a user may interact.

The simplicity of the model 1 architecture is that each JSP page is entrusted to deal with its request entirely by itself, thereby generating a response and sending it back to the client, and for this reason it’s often known



as page-centric. Usually such JSP pages will make use of some form of **model** to represent the business logic of the application. Usually JavaBean components are used to provide the model so that business logic can be neatly encapsulated for reusability and at the same time keep the amount of processing logic in the page to a minimum.



As you can see from this diagram, a JSP page is entrusted with handling a client request and building an appropriate response all by itself, thereby possibly making use of JavaBeans to encapsulate business logic.

Although each JSP page has the potential to contain a lot of processing logic, as long as the application is relatively small with few pages, the model 1 architecture is a good choice because it's very quick and simple to put together. However, such a page-centric architecture can begin to introduce problems when used with larger more complex applications. Some of the more common problems are outlined here.

## Maintainability Problems

Because each JSP page is solely responsible for handling a client request, it will often have to directly interact with a business layer. This can result in the application structure being embodied within the pages themselves. This obviously makes the pages more complicated and more likely to contain lots of scriptlet code, which ultimately makes them far harder to maintain.

## Reusability Problems

When most of the processing logic is embedded into the JSP pages, it becomes much more difficult to reuse common functionality because it's usually implemented using scriptlets. Often this results in a lot of cutting and pasting of code that isn't only bad from a reusability perspective but is also likely to introduce errors and decrease productivity.

## Security Problems

As each JSP page is responsible for handling all of its processing, it's possible that any actions that require a user to be logged in or that access password-protected resources such as databases, could end up exposing sensitive information by embedding it in the page. It's therefore important to make sure that any such logic is encapsulated into JavaBean components or custom actions to prevent this possible security hole.

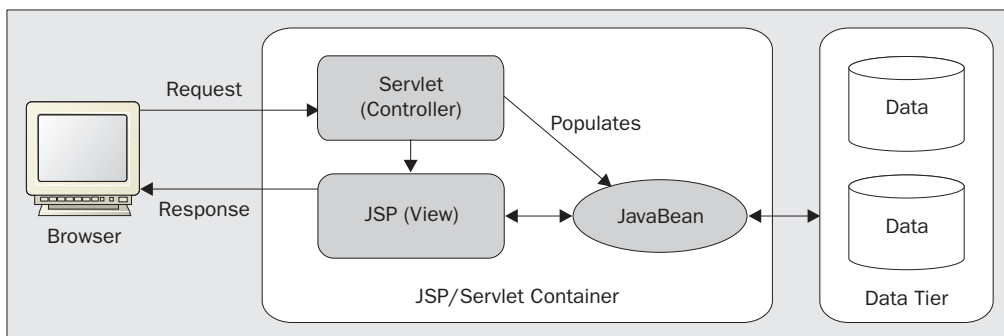
Of course it would make far more sense to provide such security controls via a single, centralized access point, and you shall see how the next architecture up for discussion does exactly this.

## Model 2 Architecture (Model-View-Controller)

As you might expect, the model 2 architecture builds on the model 1 architecture you've just seen, and it overcomes many of the problems identified earlier.

The model 2 architecture is actually a server-side implementation of the popular **Model-View-Controller (MVC)** design pattern. This pattern enforces the separation of the way application data is modeled (hence, the **model**) from the way it's presented (the **view**) and it also requires a separate component to handle the processing in between (the **controller**). Separating these responsibilities into components gives the developer a good opportunity to select the right type of component for each based on its suitability to the task.

As mentioned earlier, JSP pages are best used for presenting content and aren't particularly good for providing complex business processing for reasons of readability and maintainability. Servlets on the other hand are particularly good components for providing business processing, but aren't best suited to generating content. Most applications implementing the model 2 architecture therefore chose to utilize a controller servlet to handle all the request processing and delegate requests to separate JSP components to provide the presentation, thereby making the best use of both technologies. Remember that the model 1 architecture you saw earlier forced the controller and the view to coexist inside the same component, which accounts for a lot of its shortcomings.



As you can see from this diagram, all requests made of the web application are handled by a single controller servlet. Depending on the type of request received, the controller servlet is responsible for populating the model component with data that it has obtained, usually by interacting with the business layer. The controller then forwards the request to the JSP view component, which constructs a suitable response to the request based around the data stored in the model component.

There are, as always, varying versions of this architecture such as providing multiple controllers to distribute the request-handling functionality across multiple servlets. Although not recommended, it's also possible to provide JSP-based controllers and servlet-based view components; the choice is yours! Remember that the best designs select a component based on its suitability for its job. JSP pages make poor controllers since they're designed to render content, whereas servlets are best suited to request processing and computations instead of generating content.

Whatever components you select, you cannot fail to appreciate how much cleaner this architecture is than the model 1 architecture as each component has a definite, well-defined role.

Let's revisit some of the problems that the model 1 architecture faced and see how this new design helps solve some of them.

## Maintainability

Many of the maintainability problems associated with the model 1 architecture were a direct result of implementing a controller and view component as part of the same component: the JSP page. Because all of the business processing as well as the content generation was forced together, the result was messy pages that could be hard to maintain. By separating your application's logic from its presentation by using MVC components, it's far easier to develop cleaner code that focuses specifically on the job at hand, resulting ultimately in a more flexible and maintainable application.

## Security

By providing an initial single point of access for potential requests, a servlet-based controller component is an excellent place to provide some form of authentication. If the user making the request can pass the authentication mechanism (perhaps a username or password test) then the controller can continue with the request as normal or alternatively forward it to an appropriate page (perhaps a login page!) where the error can be dealt with.

Due to the fact that the controller component is responsible for handling each and every request, security checks only have to exist in a single place, and of course any changes to the security mechanism only have to be made once. By implementing your security constraints in a single place it's far easier to take advantages of the declarative security mechanisms denoted in the J2EE. Recall that the model 1 architecture required each page to provide similar security checks by itself, which provides a significant security hole if the developer forgets to provide it!

## Extensibility

One of the best points about the model 1 architecture is that all of the processing logic is so much more centralized. No longer does such code have to be placed in a scriptlet located deep within a JSP page where it's so much more difficult to access.

This centralization helps to provide a more component-based solution, utilizing JavaBean components and custom or standard actions where software components may be reused and extended, which greatly reduces the chance of making a change that causes a "ripple" effect to other dependent components.

# JSP Fundamentals—Hands On

Before you take an in-depth look at the individual components that comprise a JSP page you should have a basic idea how JSP applications are structured and deployed inside a JSP container.

## Basic Deployment

While all JSP and servlet containers have their own specific deployment processes, generally the basic techniques are still the same and involve copying your web application (JSP pages, servlets, and static HTML) in a predefined structure into a special deployment directory specified by the container.

The Java 2 Platform, Enterprise Edition (J2EE) specifies a special structure that all J2EE-compliant web applications must take so that J2EE containers know exactly where to find the resources that compose the web application. Most J2EE containers allow web applications to be deployed in one of the following two forms:

## ❑ Expanded Directory Format

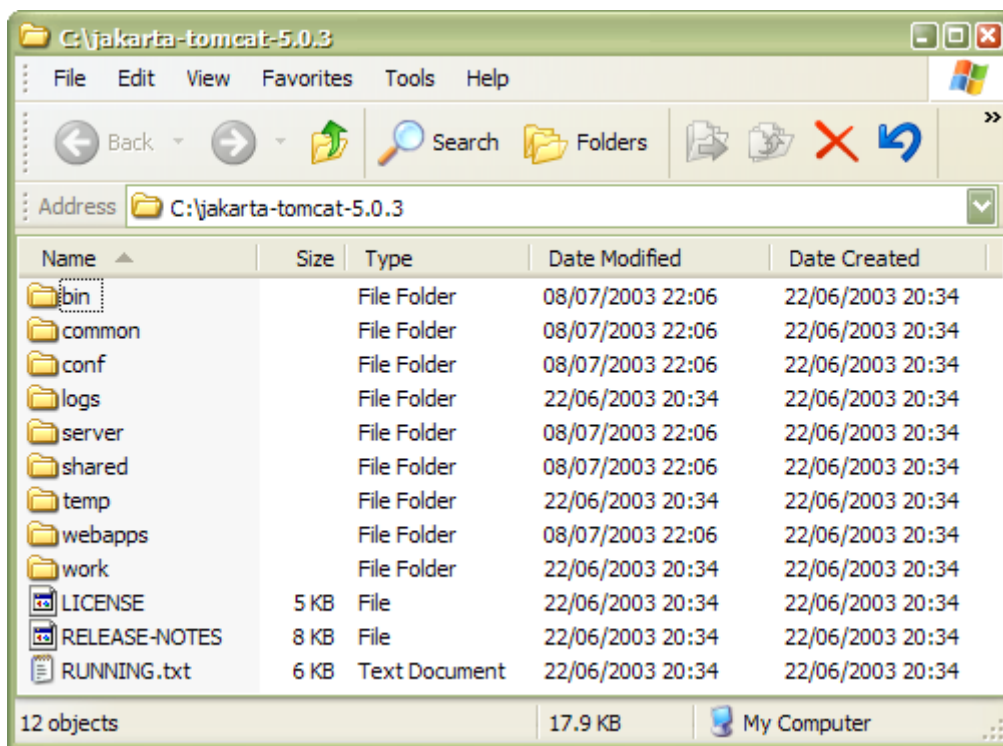
The web application in its predefined structure is simply copied into the J2EE container's deployment directory.

## ❑ Web ARchive File (WAR)

The web application in its predefined structure is archived into a compressed WAR before being copied to the J2EE containers' deployment directory.

The J2EE web application structure defines the exact locations inside your web applications to locate deployment descriptors, HTML and JSP pages, and compiled Java classes as well as third-party Java ARchive (JAR) files that are required by your web applications. We won't go into great detail explaining the intricacies of the web application structure, because this will be covered in far greater detail in later chapters. For now we shall explain the structure of a minimal but fully functional web application that you can use to test the simple JSP examples you shall learn about throughout this chapter.

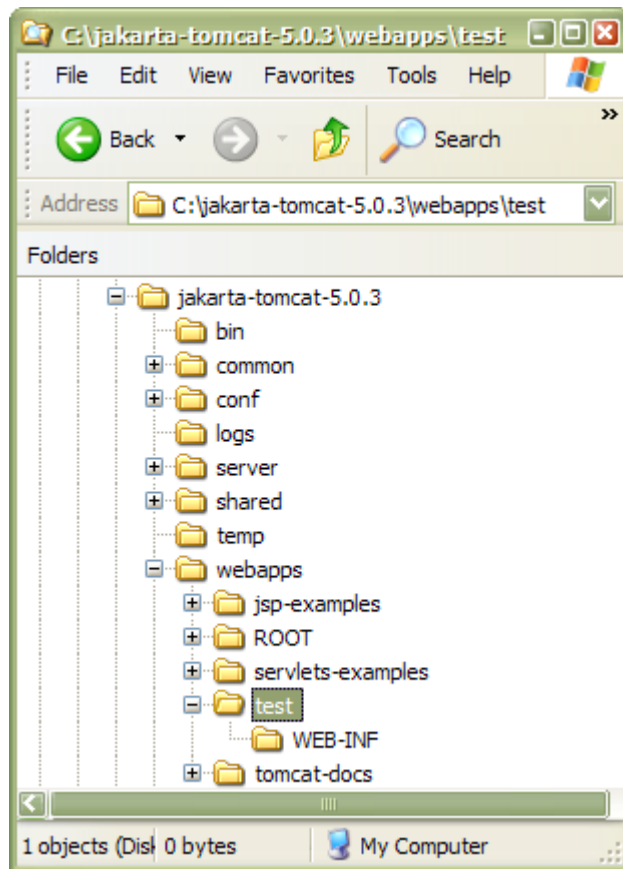
Because Tomcat 5 for the Apache Jakarta project is the reference implementation of the Servlet 2.4 and JSP 2.0 specifications and will therefore be featured heavily throughout this book, it makes sense to base the deployment introduction around the Tomcat 5 container; but feel free to use the container of your choice.



Notice that Tomcat has been installed beneath the C:\jakarta-tomcat-5.0.3 directory which shall be referred to as the %TOMCAT\_HOME% directory from now on.

As you can see, there are a fair number of subdirectories and each has a specific purpose. For example, the `bin` directory contains all of the necessary scripts required to start the container and the `conf` directory contains all of the XML-based configuration files used by Tomcat. Don't worry about understanding all the complex configuration files and directory structures because this exercise is designed to get a working web application for you to test the examples you'll see later in the chapter.

We mentioned earlier that most servlet containers have a special deployment directory whereby developers can place web applications as either WAR files or in exploded directory format. You may have guessed by now that in Tomcat it's the `%TOMCAT_HOME%\webapps` directory where you'll be creating the test application ready for deployment. Ignoring the subdirectories that already exist beneath the `webapps` directory (yes, they too are web applications in case you were wondering!), first create a directory to house the web application called `test` and then another directory inside this one called `WEB-INF`. Now you have to following structure:



This structure is the standard web application structure that exists inside all WAR files and is known as the exploded directory format. As you may be aware, all web applications must have a deployment descriptor in order to work, and your's is no different. Copy and paste this very minimal deployment descriptor into a file called `web.xml` and save it beneath the `WEB-INF` directory you just created:

```
<?xml version="1.0"?>

<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">

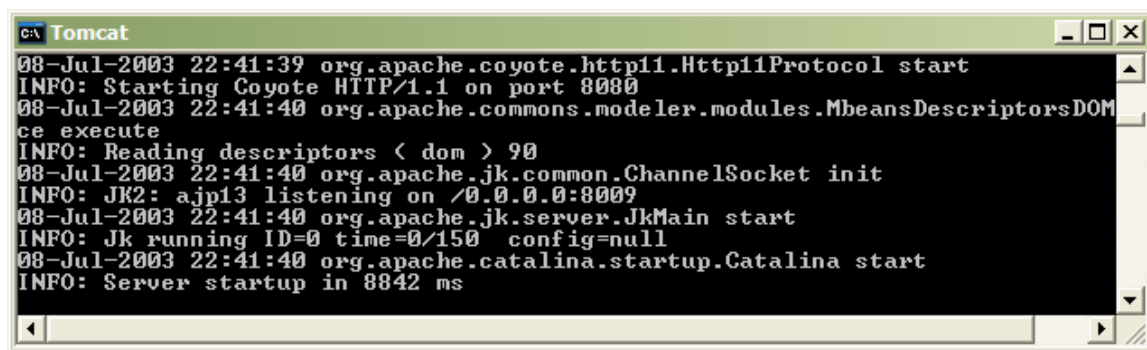
</web-app>
```

A point worth remembering is that the name of the `test` directory that you're using to house the web application is known as the **context** of the web application; you'll need to be aware of this context to request the resources of our test web application. You'll see this in action shortly.

Now that you've set up the required web application structure let's create a dynamic web component (that's a JSP to you and me!) that will return the current time of day along with a greeting. Copy the following JSP into a file called `%TOMCAT_HOME%\webapps\test\date.jsp`:

```
<html>
  <body>
    <h2>Greetings!</h2>
    <P>The current time is <%=new java.util.Date()%> precisely
  </body>
</html>
```

Again, don't worry about understanding the syntax of this JSP, you'll learn all about JSP syntax later. Start Tomcat by running the `%TOMCAT_HOME%\bin\startup.bat` (or `startup.sh`) script. When Tomcat is running you should see a prompt with messages like this:



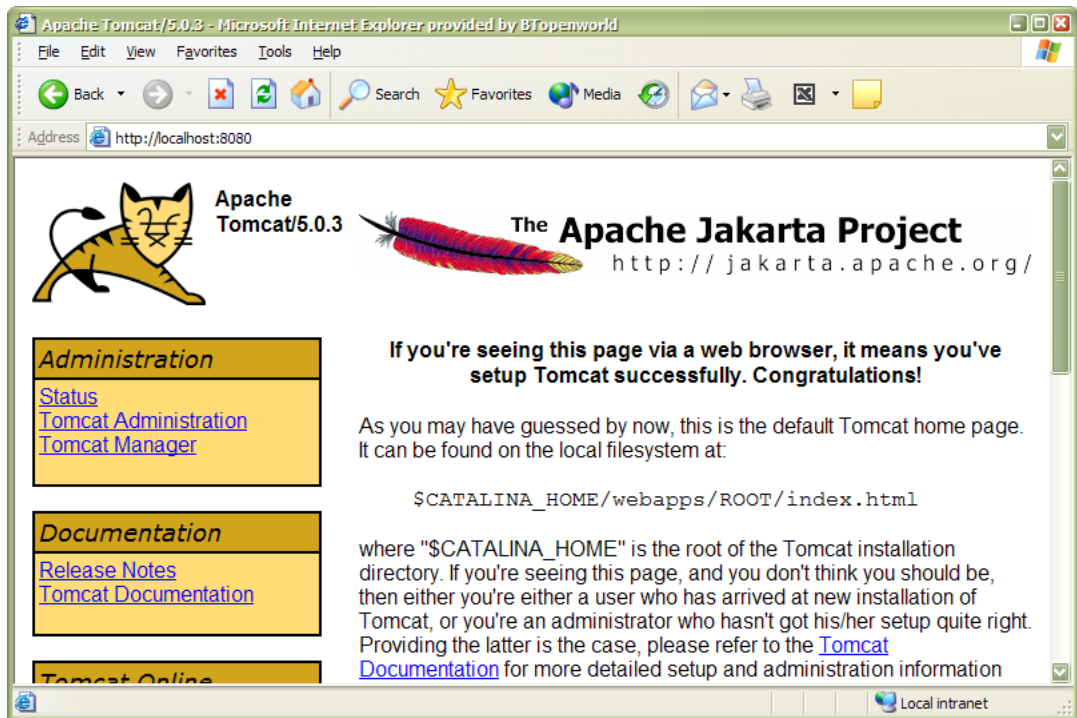
```
C:\ Tomcat
08-Jul-2003 22:41:39 org.apache.coyote.http11.Http11Protocol start
INFO: Starting Coyote HTTP/1.1 on port 8080
08-Jul-2003 22:41:40 org.apache.commons.modeler.modules.MbeansDescriptorsDOM
ce execute
INFO: Reading descriptors < dom > 90
08-Jul-2003 22:41:40 org.apache.jk.common.ChannelSocket init
INFO: JK2: ajp13 listening on /0.0.0.0:8009
08-Jul-2003 22:41:40 org.apache.jk.server.JkMain start
INFO: Jk running ID=0 time=0/150 config=null
08-Jul-2003 22:41:40 org.apache.catalina.startup.Catalina start
INFO: Server startup in 8842 ms
```

To test that Tomcat is running successfully, let's load the Tomcat welcome page by opening a web browser and typing the following link:

`http://localhost:8080`

Note that this assumes you installed Tomcat on its default port 8080, so you should change it as needed.

Now you should see the following Tomcat welcome page, which indicates that all is well:

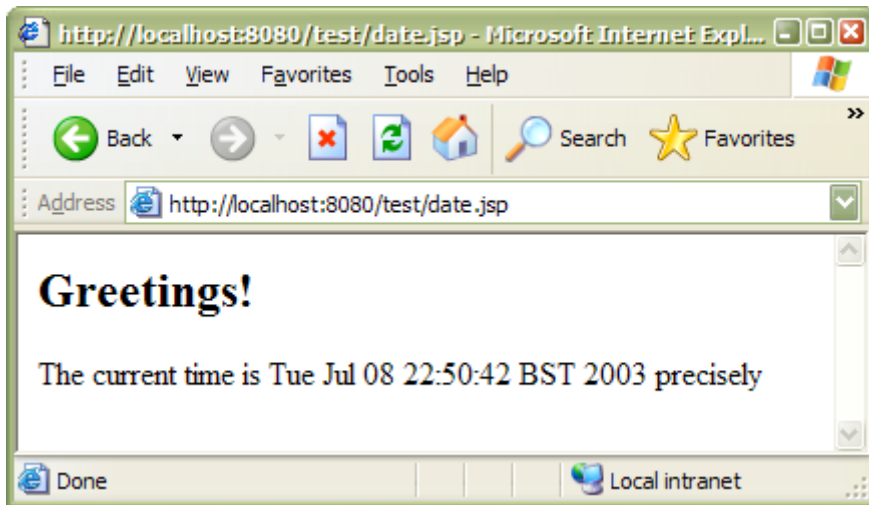


If for some reason you don't see this welcome screen, make sure that you have Tomcat running and are accessing the correct port. Consult the Tomcat documentation if problems persist.

Now you're at last ready to access the test web application. Do this by typing the following URL:

`http://localhost:8080/test/date.jsp`

Notice how you use the context of the web application (test) to inform the servlet container that it's your application whose `date.jsp` file you wish to access. You should now see some similar (not the same though—remember it's dynamic!) output to the following screen shot:



That's it, congratulations! Creating and deploying a JSP web-based application wasn't so hard after all, was it? For the remainder of this chapter you'll see lots of JSP code examples that you're encouraged to copy and paste into JSP files beneath the `webapps\test` directory (and change the link accordingly) to see the code in action. Note that you may need to stop and start Tomcat to see some changes in action.

## JavaServer Pages

As mentioned earlier, the sole purpose of JSP technology is to produce dynamic, web-based content. This capability is implemented by embedding programmatic logic among template data (usually markup such as HTML, XML, and so on) which together produces the dynamic content on a request-by-request basis. This programmatic logic may be classified into the following JSP elements:

- ☐ Scripting elements
- ☐ Directives
- ☐ Action elements

In a moment you'll look at each of the elements in turn so you can see how collectively they combine to produce the dynamic content required by today's web applications, but first let's look at the template text.



## Template Text

Any non-JSP code located inside a JSP page is known as **template text**. Template text can take any form as long as it's text based. The most common form of template text is markup such as HTML or XML. For example, if your web design team was to develop an HTML page that you were required to convert into a JSP page in order to add some form of dynamic processing, then all of the HTML markup would be referred to as template text:

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<%@ page import="com.apress.projsp20.jstl.CalendarBean"%>

<html>
  <head>
    <title>My HTML Example</title>
  </head>
  <body>
    <jsp:useBean id="cal" class="com.apress.projsp20.jstl.CalendarBean"/>
    <c:set var="hour" value="${cal.hour}" scope="request" />

    <c:choose>
      <c:when test="${hour > 0 && hour <=11}">
        Good Morning!
      </c:when>
      <c:when test="${hour >= 12 && hour <=17}">
        Good Afternoon!
      </c:when>
      <c:otherwise>
        Good Evening!
      </c:otherwise>
    </c:choose>
  </body>
</html>
```

This is a JSP page that dynamically produces a greeting depending on the time of the day. For now, don't worry about understanding the syntax of the various JSP elements but notice that the highlighted static HTML is referred to as template text. The reason for this term is simply that a JSP can be thought of as a “template” for producing some output. It's the JSP logic embedded inside this template text that is responsible for producing the output based upon this template.

During the translation phase, all of the template text found in the original JSP page is converted into Java statements inside the page implementation servlet that simply output the template text in the correct order as part of the response.

## Scripting Elements

Scripting elements are used within a JSP page to manipulate objects and perform computations that enable the generation of dynamic content. Scripting elements can be classified into the following individual elements:

- ❑ Comments
- ❑ Declarations

- ❑ Scriptlets
- ❑ Expressions
- ❑ Expression language expressions

We shall discuss each scripting element in turn.

## Comment

JSP comments are a good way of explaining any complicated logic that may have arisen for whatever reason—perhaps it could be used to flag a piece of scripting code to be simplified at a later date with a custom tag. Alternatively, comments provide non-Java-speaking HTML users or web designers some clues as to what a piece of “magic” JSP code does.

JSP comments may be declared inside a JSP as follows:

```
<%-- This is a JSP comment --%>
```

Comments in JSPs get stripped out during the translation phase and aren’t sent to the client as part of the response. HTML comments on the other hand, such as the one shown here, do get sent to a client’s browser and any client can view the comments by using the **View Source** options that most modern browsers provide:

```
<!-- This is an HTML comment -->
```

The fact that JSP comments are stripped and don’t form part of a client response is a good thing as it not only keeps the size of the response as small as possible thereby aiding performance, but also removes clues to a potential hacker with regards to the technology used to implement a web-based application the hacker is targeting.

There is of course no reason why JSP and HTML comments cannot work together:

```
<!-- HTML comment generated <%= new java.util.Date() %> -->
```

You’ll learn the meaning of this JSP expression shortly, but suffice to say the previous comment produces the following in the content returned to a client:

```
<!-- HTML comment generated Fri Jan 03 12:37:09 GMT 2003 -->
```

## Declarations

JSP pages allow both methods and variables to be declared in a similar manner to the way in which they’re declared inside normal Java classes. As with normal methods and variables, once declared inside a JSP page they’re available to subsequent scriptlets and expressions and so on for reference.

During the translation phase, any JSP declarations (methods or variables) found inside the page are actually created outside the normal `_jspService()` method in the JSP page implementation servlet and therefore are available to any scripting elements throughout the page.

JSP declarations must be placed between `<%!` and `%>` declaration delimiters. The general form of a declaration is shown here:

```
<%! declaration; [declaration;]+...%>
```

For example:

```
<%! Date now = new Date(); %>
```

```
<%!
    private int calculate(int a, int b) {
        ...
    }
%>
```

The previous examples demonstrate two simple JSP declarations. The first generates a `java.util.Date` instance that is available to the rest of the JSP (and therefore to the servlet), whereas the second actually declares a stand-alone method that again is available to the rest of the page.

**An important note to consider when declaring a page-level variable via a JSP declaration is to ensure that its access is thread-safe. Multiple threads can execute the same servlet (JSP implementation class) simultaneously, and any page-level variables are accessible by each thread.**

## Scriptlets

Quite simply, scriptlets are small blocks of source code contained within the `<%` and `%>` delimiters that can be used to provide programming-style language functionality around a page's content, thus making their output dynamic.

For example:

```
<%
    User user = (User)request.getAttribute("User");
    if (user != null) {
%>
    Welcome, you have successfully logged in!
<%
    }
%>
```

You can see from the previous example how a very simple piece of dynamic content can be created with a scriptlet. If an object attribute called `user` exists in the request then a welcome message is generated, otherwise one isn't! Admittedly, this piece of dynamic content isn't the most complex but hopefully you can see how scriptlets add logic between the markup of your JSP page to control the output.

The supported scripting languages available for use inside scriptlets are defined by the page directive's language attribute, which as of JSP 2.0 only supports the use of Java as a scripting language. Unlike declarations, all scriptlet code will be inserted into the `_jspService()` method of the generated servlet, which is used to handle the request and generate the response. When multiple scriptlets are included in any page they're included into the compiled servlet in the order in which they appear in the JSP. Unlike JSP declarations, any variables declared in a scriptlet aren't available to the rest of the JSP because they're treated as local variables inside the `_jspService()` method.

When JSP 1.0 first arrived, scriptlets were quickly adopted as the most popular way of adding dynamic features to JSP. Unfortunately, scriptlets became too popular and soon JSP page authors were embedding too much business logic among their markup. This caused several problems.

In multideveloper projects, it's quite common for a web designer with no Java or JSP skills to produce an HTML UI for an application that is then passed to Java developers to convert their work into JSP by adding dynamic content and hooking together business logic along the way. This caused numerous problems in the JSP 1.0 days, whereby delays and frustrations were created due to the dependencies formed between the UI and Java developer. Also problems arose due to the fact that the UI designer would struggle to maintain their pages, as they would need to understand the scriptlet code surrounding their markup properly in order to change it. On top of these difficulties, adding too many scriptlets to a JSP also makes it incredibly difficult to read and hence maintain. Anyone who has had to spend hours debugging a JSP only to find a closing brace is missing will testify how much more difficult it is to fix a page with too many scriptlets on it.

Thankfully, the early experiences of UI and JSP developers haven't been wasted and other methods are now considered better alternatives. For example, using standard JSP actions to manipulate JavaBeans, which contain business logic and encapsulating logic inside custom actions (also known as custom tags), are two alternatives that solve many of the problems mentioned earlier. Most noticeably, both solutions involve the use of XML-style tags that can be used in harmony with the tools of a UI designer.

JSP 2.0 introduces another two candidates that further facilitate scriptless JSP code. The first of these new features comes in the form of the JSTL, which provides a number of standard actions for many of the simple tasks required of a modern dynamic web application. Secondly, for the first time an **expression language (EL)** is available, which can be used to help reduce or even eradicate scriptlets.

As we mentioned at the start of this chapter, further information on JavaBeans, JSTL, and custom actions can be found later in this book where they're discussed in great detail. For now you just need to understand what scriptlets can do along with their limitations.

## Expressions

Expressions are similar to scriptlets, but as their name suggests they evaluate a regular Java expression and return a result. This result must be a `String` or be convertible to a `String`, otherwise an exception will be raised during the translation phase or at runtime. Expressions are evaluated by the JSP implementation servlet and are returned to the client as part of the response.

As with the other tag types, expressions must be placed between `<%=` and `%>` expression delimiters so that the JSP engine is aware of the developer's intent to return the value of the expression in the response. The general syntax is as follows:

```
<%= expression %>
```

Two very simple JSP expressions can be seen here. They could be part of any regular JSP that generates nonstatic HTML.

```
<h1>Welcome Back : <%= user.getName() %></h1>

<b>Today's date is <%= new java.util.Date() %></b>
```

Apart from producing dynamic content as part of the client response, JSP expressions can be used to pass request-time parameters and values to other JSP actions that may appear on the page. You'll look at an explanation of this later.

**Unlike declarations and scriptlets, JSP expressions don't require a closing semicolon (in fact they won't compile with one) as they evaluate the result of a single expression.**

## Expression Language Expressions

JSP 2.0, for the first time, introduces an EL based on both ECMAScript and XPath, which has been designed to be simple to use and more user-friendly than Java.

The new EL has built-in support for JavaBean access and manipulation, collections of objects, and automatic type conversion to name but a small part of its extensive feature list. If you're familiar with JavaScript you should have no problem understanding the syntax of the EL, which insists that all expressions must be enclosed within `${ }` and `}` delimiters.

EL expressions can be used in any attribute that accepts a runtime expression, usually a standard or custom action, or even in plain template text. The addition of the EL further facilitates the writing of scriptless JSP pages; that is, pages that don't contain any Java scriptlets, expressions, or declaration elements.

Although it's the subject of the next chapter, here are a couple of examples to give you a flavor of the new EL:

```
${anObject.aProperty}
```

```
<c:if test="${user.salary > 10000}" >
  ...
</c:if>
```

You can see from the first example mentioned here just how simple it is to access the property of any JavaBean object, with no Java knowledge required at all! The second example demonstrates one of the core actions from the JSTL that is used to provide conditional processing of JSP code. Here an EL expression is used to provide the Boolean test for the action.

## JSP Implicit Objects

All JSP scripting elements have access to a number of useful objects provided by the JSP container that are known as implicit objects. Each of these implicit objects are classes or interfaces as defined by either the Servlet or JSP specifications and are described in greater detail in this section.

### *request*

The most notable of the implicit objects is the `request` object, which is an instance of the `javax.servlet.http.HttpServletRequest` interface. The `request` object provides access to all of the available information about the user request such as request parameters and headers and may be used in exactly the same way as the `HttpServletRequest` parameter is used in the `service()` method of a normal servlet.

Let's consider an example. Imagine a simple JSP that expects a single-request parameter called `userName` and constructs a personalized response to the user.

```
<html>
<head><title>A Simple Example</title></head>
<body>
<h2>Hello<%=request.getParameter("userName")%>, Have a nice day!</h2>
</body>
</html>
```

This simple JSP extracts a request parameter called `userName` from the implicit request object and constructs an appropriate greeting. To send a request parameter to a JSP like the one outlined previously either use an HTML form or add the parameter to the query string of your request as follows:

`http://localhost:8080/test/Request.jsp?userName=Dan`

### *response*

In a similar manner to the `request` object seen earlier, there's also an accompanying implicit `response` object that represents the current response to be returned to the user. The `response` object is an instance of the `javax.servlet.http.HttpServletResponse` interface. Again, this object can be used in exactly the same way as the `HttpServletResponse` parameter received by the `service()` method of a normal servlet.

### *out*

The implicit `out` object represents an instance of the `javax.servlet.jsp.JspWriter` class that can be used to write character data to the response stream in a similar manner to that seen by the `java.io.PrintWriter` class. While the methods provided by the `JspWriter` such as `print()` and `println()` can be used to write text to the body of the response, it's normally sufficient to rely on plain template text and JSP action elements instead of explicitly writing to the `out` implicit object.

### *session*

The implicit `session` object provides a reference to an implementation of the client's individual `javax.servlet.http.HttpSession` object, which can be used to store and retrieve session data. While the `HttpSession` can be used explicitly, it should be noted that there are several action elements available that interact with the session that can be used instead.

## config

The `config` object simply provides the JSP developer with access to the `ServletConfig` object that is used by the web container to configure the JSP and its implementation servlet. The `ServletConfig` interface is most commonly used to provide access to any initialization parameters that have been configured for either the JSP or its implementation servlet via the deployment descriptor of the web application.

## application

The implicit `application` object provides a reference to the `javax.servlet.ServletContext` interface of the web application. The `ServletContext` is used by a web container to represent an entire web application and therefore any data that is stored inside it will be available to all resources included in the application.

Like the `session` object, several action elements exist that interact with the `ServletContext` so it may not be necessary to interact directly with the object itself.

## page

The implicit `page` object references an instance of the JSP's page implementation class and is declared of type `Object`. The `page` object is rarely used in scripting elements and simply serves as a link between the JSP and its implementing servlet.

## pageContext

The `pageContext` object is slightly different in its functionality from the rest of the available implicit objects. A `pageContext` instance provides the JSP developer with access to all of the available JSP scopes and to several useful page attributes such as the current request and response, the `ServletContext`, `HttpSession`, and `ServletConfig` to name but a few.

Perhaps the most useful piece of functionality provided by the `pageContext` variable is its ability to search for named attributes across multiple scopes. Therefore, if you were unsure as to which scope a particular attribute is located, the `pageContext` can be used to traverse all available scopes until the attribute is found.

The `pageContext` variable provides this cross-scope functionality due to the fact that it exists at a level of abstraction higher than the lower level JSP implementation classes. The JSP container will create a new unique instance of this class for each request received and assign it to the `pageContext` variable.

## exception

The implicit `exception` object is only available to those JSP pages that declare themselves as error pages using the following page directive (You'll learn more about directives shortly!).

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

The `application` object itself is an instance of a `java.lang.Throwable` and will represent a runtime error that occurred during the request process.

Any scripting elements inside a JSP page that reference the implicit `exception` object when that page hasn't been declared as an error page will cause a fatal error at translation time.

## JSP Directives

Directives are used for passing important information to the JSP engine. Although directives themselves generate no output, they provide a powerful mechanism for providing page-level information that is typically used during both the compilation and translation phases.

JSP page authors have the following three types of directives at their disposal:

- ❑ page directives
- ❑ include directives
- ❑ taglib directives

Each of these types of directives provides different information to the JSP engine or signifies some required behavior of the generated servlet. The information that is contained inside a directive is totally independent of any user request and is only of use to the JSP engine. All three directive types must be declared between `<%@` and `%>` directive delimiters and take the following form:

```
<%@ directive {attribute="value"}* %>
```

Generally speaking, directives should be placed at the top of the JSP page; however, the `include` directive, which you'll see later, is an exception to this rule. Let's examine each directive type in turn.

### **The page Directive**

The first directive type is the `page` directive, which is used to define any page-dependent properties that a JSP page may have, such as library dependencies, buffering, or error-handling requirements to name but a few.

The syntax of a `page` directive is as follows:

```
<%@ page page_directive_attr_list %>
```

where the `page_directive_attr_list` is used to define the name of any page attribute along with its value in the form `attributeName=attributeValue`.

Each `page` directive applies to the entire compilation unit (that is, the complete JSP plus any included JSP pages) and although multiple `page` directives may occur it should be noted that each attribute can only occur once in the `page` with the exception of the `import` attribute.

A table of the permitted attributes and their possible values as defined by the `page_directive_attr_list` is given here:



Attribute	Permitted Value	Description
language	"scriptingLanguage"	<p>The scripting language used in scriptlets, expressions, and declarations in the JSP.</p> <p>Currently, JSP 2.0 only supports a value of "Java" and any other value would cause a fatal translation error.</p>
extends	"className"	<p>The name of a fully qualified Java class that will form the superclass of the JSP page's implementation servlet.</p> <p>This attribute should not be used lightly as its use prohibits the JSP container from using its own specially optimized classes.</p>
import	"importList"	<p>Indicates the classes available for use within the scripting environment. Any import values must be fully qualified Java class names and result in a standard Java <code>import</code> statement in the page implementation servlet.</p> <p>Note that import attributes may be a fully qualified package name followed by a <code>"."</code> or a list of comma-separated classes.</p> <p>The default import list is as follows:</p> <pre>javax.servlet.jsp.* and javax.servlet.http.*</pre>
session	"true false"	<p>Indicates that the page requires an <code>HttpSession</code>. If a value of "true" is provided, an implicit scripting variable named <code>session</code>, which references the current <code>HttpSession</code> object, is available to the page.</p> <p>If a value of "false" is used, then any references to the implicit session variable will cause a fatal translation error.</p> <p>The default value is "true".</p>

Table continued on following page

Attribute	Permitted Value	Description
buffer	"none   sizekb"	<p>Specifies the buffering model for the initial <code>JspWriter</code> used to handle the response generated by the page.</p> <p>A value of "none" indicates no buffering is required and all output is written immediately. The size of the buffer can only be declared in kilobytes and the kb prefix is required. If a buffer size is specified, then all output is buffered with a buffer not smaller than the one specified.</p> <p>The default value isn't less than 8 KB.</p>
autoFlush	"true   false"	<p>Indicates whether the output buffer should be flushed automatically when full (specified by a value of "true") or whether an exception should be raised indicating buffer overflow (specified by a value of "false").</p> <p>Note that it's illegal to set <code>autoFlush=true</code> when <code>buffer=none</code>. The result is a translation-time error.</p> <p>The default value is "true".</p>
isThreadSafe	"true   false"	<p>Indicates the threading model to be used by the JSP and servlet container when dispatching requests to the page implementation servlet.</p> <p>A value of "true" ensures that the JSP container may choose to dispatch multiple requests to the page simultaneously. A value of "false" indicates that the JSP container must serialize multiple requests to the page one at a time.</p> <p>Note that if a value of "true" is passed then the JSP page author must ensure that access to any shared variables is synchronized to protect thread safety.</p> <p>The default value is "true".</p>
info	"info_text"	<p>Can be used to provide any arbitrary string, which is returned via a call to the page implementation servlet's <code>getServletInfo()</code> method.</p>
isErrorPage	"true   false"	<p>Indicates whether or not the current JSP is intended to be an error page, which may be referenced by the <code>errorPage</code> attribute of another JSP.</p> <p>If a value of "true" is specified an implicit scripting variable exception is made available that references the offending <code>Throwable</code> from another JSP page.</p> <p>The default value is "false".</p>

Attribute	Permitted Value	Description
errorPage	"error_url"	<p>Defines the URL to a resource that any throwable objects not caught by the page implementation are forwarded for error processing.</p> <p>When an error page for a web application is defined in its deployment descriptor (<code>web.xml</code>), the JSP's error page is tried ahead of the one defined by the deployment descriptor.</p>
contentType	"ctInfo"	<p>Defines the character encoding for the JSP page and its response as well as the MIME type of the response.</p> <p>The default value for the type is <code>text/html</code> and for the charset it's <code>ISO-8859-1</code>.</p>
pageEncoding	"peInfo"	<p>Defines the character encoding for the JSP page.</p> <p>The default value is <code>ISO-8859-1</code>.</p>
isELIgnored	"true false"	<p>Defines whether the EL expressions are evaluated for the JSP page.</p> <p>If enabled any EL expressions are simply ignored by the JSP container.</p> <p>The default value is "false".</p>

The following are some examples of the `page` directive in action:

```
<%@ page language="Java" %>
```

Here the scripting language to be used on the page is set to Java (the only permitted value):

```
<%@ page import="java.util.Date, java.text.*" %>
```

The `java.util.Date` class, along with all the classes from the `java.text` package, are available for use on the page.

```
<%@ page isThreadSafe="false" buffer="20kb" %>
```

Notice that it's possible to provide multiple attributes in the one `page` directive; here the JSP container is advised that multiple requests may not access the page simultaneously and also the page buffer should not be less than 20 KB.

## The *include* Directive

You've just seen how the `page` directive can be used to pass information to the JSP engine during the translation phase to control how the page implementation class is generated. The `include` directive also executes at translation time and enables the contents of a separate resource to be statically merged inside the original page, thus radically affecting the generated servlet.

After translation, the generated JSP servlet contains the content and logic as defined by the two separate resources, in the order that they were specified in the original JSP. This makes it seem as though they were from a single JSP file.

The following is the syntax for the `include` directive:

```
<%@ include file="relativeURL" %>
```

As you can see the directive accepts a single `file` attribute that is used to indicate the resource whose content is to be included in the declaring JSP. The `file` attribute is interpreted as a relative URL; if it starts with a slash it's interpreted as relative to the context of the web application (namely a context-relative path), otherwise it's interpreted as relative to the path of the JSP that contains the `include` directive (namely a page relative path). The included file may contain either static content, such as HTML or XML, or another JSP page.

For example:

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

As you can see from the previous example, a static HTML file located at the root of the web application context will be statically included into the JSP page that declares the directive. The included file in this case contains important legal copyright information that must be included on all pages of a web application. The `include` directive is an excellent mechanism for reusing a predefined component, such as the `copyright.html` file, to save the duplication of code on each page. This makes the JSP pages smaller, easier to read, and more maintainable, as changes only need to be applied in one place but are reflected throughout the application.

**Remember when using the `include` directive, the combined contents of the original JSP and all of its included resources are translated into the same implementation servlet. Therefore the original JSP page will share its scripting variables and declarations with those inside the included resources and any duplication of variables or methods names will result in a fatal JSP translation error, because the merged file won't be syntactically correct.**

The `include` directive is slightly different from the other JSP directives, which are typically declared only once at the top of each JSP page. A JSP page may contain any number of `include` directives at any position in the page to indicate the exact positions where the content from the included resource should be inserted. Therefore the `include` directive is well suited to the implement simple template mechanisms that are so commonly used in today's modern web applications. This enables all the commonly used resources of a web application (such as a header, footer, or navigation page) to be encapsulated as separate components (for example, JSP pages or static HTML pages) included throughout the web application.

Let's consider a real-world example of such a templating mechanism that utilizes the `include` directive to provide a consistent page layout for a web application.

Consider the following two JSP pages:

Header.jsp

```
<html>
  <head><title>A Very Simple Example</title></head>
  <body style="font-family:verdana,arial;font-size:10pt;">
    <table width="100%" height="100%">
      <tr bgcolor="#99CCCC">
        <td align="right" height="15%">Welcome to this example...</td>
      </tr>
      <tr>
        <td height="75%">
```

Footer.jsp

```
        </td>
      </tr>
      <tr bgcolor=" #99CC99">
        <td align="center" height="10%">Copyright ACompany.com 2003</td>
      </tr>
    </table>
  </body>
</html>
```

As you can see, `Header.jsp` declares the starting elements of an HTML table that is to be 100 percent of the size of the page and has two rows, whereas `Footer.jsp` simply declares the closing elements for the table. Used separately, either JSP will result in partial HTML code that will look very strange to a user but when they're combined using the `include` directive it's easy to create consistent pages as part of a web application.

Let's see just how simple this basic template mechanism is to use:

Content.jsp

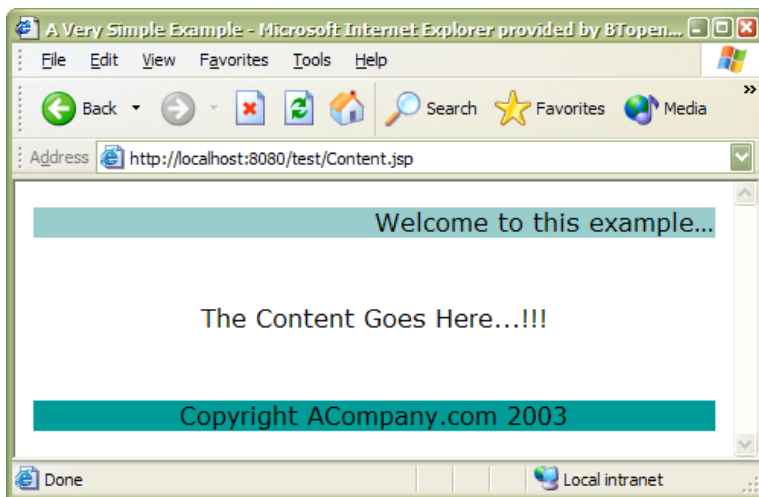
```
<%@ include file='./Header.jsp'%>
<p align="center">The Content Goes Here...!!!</p>
<%@ include file='./Footer.jsp'%>
```

As you can see, `Content.jsp` looks like a simple page with only three lines of code; the two `include` directives and the actual body of the page. Notice how the body of the page is actually between the two directives. This ensures that all of this page's body content is included inside the table declared in `Header.jsp`.

To run this example simply copy the three files `Content.jsp`, `Header.jsp`, and `Footer.jsp` beneath the test web application directory you created earlier and go to the following URL:

<http://localhost:8080/test/Content.jsp>

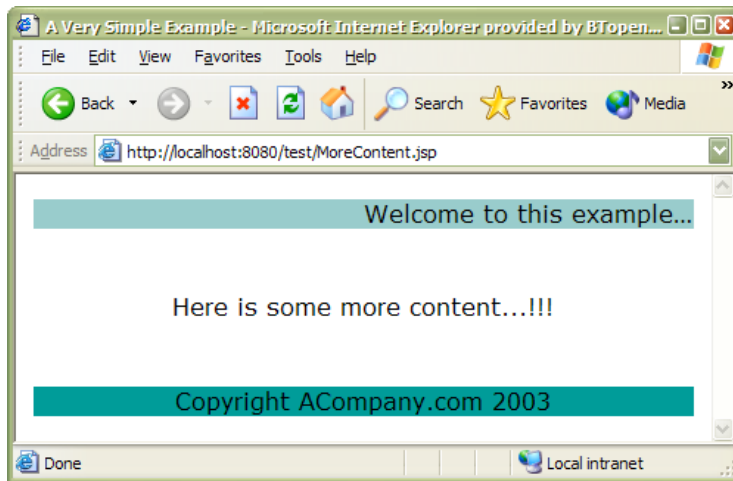
You should see something like the following screen shot:



To demonstrate just how effective this basic template mechanism is, let's construct another page (`MoreContent.jsp`) and see how easy it is to maintain the same look and feel:

```
<% include file="./Header.jsp"%>
<p align="center">Here is some more content...!!!</p>
<% include file="./Footer.jsp"%>
```

To open it, do exactly the same as you did for the `Content.jsp` page, only this time point to the URL <http://localhost:8080/test/MoreContent.jsp>:



This example should help demonstrate how useful the `include` directive is for constructing component-based web applications that share a consistent look and feel throughout, are extremely maintainable, and do not suffer from related problems caused by code duplication.

## The *taglib* Directives

A tag library quite simply contains a collection of actions (also known as tags) that can be grouped together to perform some form of logic. These actions are XML-based so their use is considerably easier for a non-Java-speaking UI designer. Of course another major benefit is that they can encapsulate large amounts of programmatic logic into a single line of code, which is a much better solution in terms of readability and maintainability and of course reuse, when compared to the ugly scriptlet-based approach you saw earlier.

Tag libraries come in two different flavors these days. The first is custom tag libraries, which have generally been put together by the development team or acquired from another team or project or even the Internet, and as of JSP 2.0, the JSTL, which contains a set of useful actions that are applicable to almost every web application in use today in some form.

To make use of a custom tag library, the web container needs to be made aware of specific information about the library itself. A special file called a **tag library descriptor** (TLD) is used for this purpose. The XML-based TLD file is used to provide general descriptive information about the custom tag library, such as a description of its usage and the JSP version that the tag supports. More importantly, the TLD file contains important information about each of the custom actions or tags that are included inside the tag library, such as which attributes are permitted by which tags, whether the tags accept body content, and so on.

Once the JSP container is made aware of the TLD for a particular custom tag library, the JSP developer can make use of any of the tags declared inside the library. Like custom Java classes (in fact, any class that doesn't reside in the core `java.lang` package), tag libraries must be imported into the page before they can be used. You've seen that Java classes are imported into a JSP page by using a JSP page directive, and in a similar fashion, tag libraries are imported using the `taglib` directive.

The syntax for the `taglib` directive is as follows:

```
<%@ taglib uri="/tagLibraryURI" prefix="tagPrefix" %>
```

The attributes are as follows:

Attribute	Description
<code>uri</code>	Can either be an absolute or a relative URI that identifies the TLD, and therefore, the tag library that is associated with the prefix.
<code>prefix</code>	<p>Indicates a uniquely identifiable string, which is used in the <code>&lt;prefix:tagname&gt;</code> declaration to identify the particular tag in use.</p> <p>Note that prefixes that start with any of the following aren't allowed because they're reserved by Sun: <code>jsp</code>, <code>jspx</code>, <code>java</code>, <code>javax</code>, <code>servlet</code>, <code>sun</code>, and <code>sunw</code>.</p> <p>All prefixes must follow the naming convention as specified in the XML namespaces specification.</p> <p>The current version of the JSP specification doesn't support empty prefixes.</p>

There are, in fact, three different ways that the `taglib` directive can be used to make a tag library available to JSP page authors, and you'll see each in turn.

### Option 1—Absolute URI

The first option for using the `taglib` directive involves passing an absolute value in the `uri` attribute that represents the location of the TLD file:

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

As you can see, the location of the TLD file is explicitly given by the `uri` attribute. In this example, the `WEB-INF/tlds/myTaglib.tld` file describes the tag library, and any references to any of the tags inside this library must be prefixed with `myPrefix` to distinguish the tag from any other tag library that may be available.

For example, if a tag named `displayImage` was included in the library, then its use may look something like this:

```
<myPrefix:displayImage file="logo.jpg" />
```

The `taglib` directive is typically used in this form during development, when the locations of resources, such as images and TLDs, haven't been finalized. Perhaps the application hasn't been packaged into a WAR file and still exists in exploded directory format, therefore an absolute value for the TLD file is usually the most convenient.



Leaving your `taglib` directives in this form after the development process has finished is perhaps not the most flexible option available. Every tag library used in the application would, therefore, have its TLD location hard-coded into the JSP source. If the location of a TLD had to change for some reason, then each JSP page would have to be altered, which could potentially be a long-winded, error-prone exercise.

### Option 2—Relative URI

The second form of the `taglib` directive involves using a relative URI to indicate the location of the TLD file. If a relative URI is to be used, then a relative mapping must be configured in the web application's deployment descriptor using the `<taglib>` element. Using the earlier example, you can see this in the following example:

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

If the deployment descriptor for a web application declared the relative URI as shown previously, then any JSP pages contained in the web application could import the tag library ready for use using the `/myTaglib` URI:

```
<%@ taglib uri="/myTaglib" prefix="myPrefix" %>
```

As you can see, the `taglib` directive no longer explicitly declares the location of the TLD file, but instead relies upon the existence of a relative URI mapping in the application's deployment descriptor. This form of the directive is the most popular due to the flexibility it provides. For example, by enabling the relative URI mapping to be set in the deployment descriptor, the location of the TLD files can effectively be set at deployment time, and any changes to the location can be made very simply in the one place.

### Option 3—Packaged JAR

The third and final use of the `taglib` directive involves providing an absolute path to an external JAR file. As the name suggests, a JAR file is simply a way of packaging compiled Java classes and resources into a compressed archive file that can be placed into your application's class path for use. Often when you use a third-party software component it will be distributed as a JAR file. JAR files are created using the `%JAVA_HOME%\bin\jar.exe` utility. Further information on its usage may be found by simply running the `jar` command without any parameters.

As mentioned, the `taglib` directive can accept an absolute value to a JAR file as the value of the `uri` attribute. This form requires that the JAR file should contain all of the tag handler classes as well as the TLD file, which must be located inside the `META-INF` directory of the JAR file.

This particular form of packaging is most commonly used when tag libraries are being used from external sources, perhaps when they're purchased from a third party or from another application. It provides a good way to encapsulate all the necessary aspects of a tag library into one distributable component.

Let's have a look at an example:

```
<%@ taglib uri="/WEB-INF/lib/myTaglib.jar" prefix="myPrefix" %>
```

As you can see, the entire tag library is stored in a single component inside the `WEB-INF/lib` directory, where it will be added to the web application's class path, from where it's available.

The one downside to packaging your tag libraries into an external JAR file is that the TLD file is difficult to access. Any changes to the attribute list or similar requires the JAR to be extracted first and then repackaged. Of course, the advantage of this method is that your tag libraries are self-contained and easy to distribute and reuse.

## Action Elements

We mentioned earlier how difficult it was to read and maintain JSP pages that are full of scriptlet code. Not only are such pages “ugly” but they're almost meaningless to a web developer unless they also happen to be a Java developer who may have written the scriptlet code in the first place.

A better alternative is to either make use of existing actions (tags) provided by a tag library that encapsulate pieces of functional logic so JSP pages are much cleaner and more readable, and because they're XML tag-based and are usable by a non-Java UI developer. In JSP 2.0 there are three different types of action elements, as you can see:

- ☐ Standard actions
- ☐ Custom actions
- ☐ JSTL actions

Let's take a brief look at each one in turn.

### Standard Actions

The JSP standard actions have been in existence since the first release of the JSP 1.0 specification and provide the JSP page author with a (relatively small) selection of useful actions. The majority of the provided functionality is based around the manipulation of JavaBean components as well as the dynamic inclusion of files at request time and URL forwarding.

Let's take a look at some of the more popular standard actions to get a “flavor” of their functionality.

#### The `<jsp:include>` Action

You saw earlier how the `include` directive provides a simple mechanism for including the contents of a separate web component to be included into the declaring JSP at translation time. The `<jsp:include>` action also provides a similar facility but with some subtle differences. The `<jsp:include>` action is actually executed at request time, thereby enabling the inclusion of both static and dynamic content and thus providing a more flexible approach.

Another major difference is that the `<jsp:include>` action doesn't include the actual content from the included resource in the same manner as the `include` directive. Instead, the `<jsp:include>` action will include any output generated by the included resource directly to the `JspWriter` assigned to the implicit `out` variable. This means that you can specify any different type of web resource, such as another JSP or servlet, as long as it produces content of the same type as the calling JSP page.

The syntax for using the standard `include` action is as follows:

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

Here you can see the action has two attributes. The `page` attribute is mandatory and contains a relative URL to the resource to be included, in the same way that the `file` attribute was interpreted by the `include` directive. The second optional attribute, `flush`, specifies whether or not the body of the response should be “flushed” or sent to the client browser before the page inclusion. The `flush` attribute is optional and defaults to `false` if not specified.

Due to the fact that the `flush` attribute can cause buffered content to be returned to a client before the included resource is executed, any included resources may not set any response headers.

We mentioned earlier that the `<jsp:include>` action allows static as well as dynamic content to be included, and we hinted that this ability offers flexibility that isn’t achievable using the `include` directive. One example of such flexibility is the fact that the `page` attribute can be specified via a request parameter, because the `<jsp:include>` action isn’t executed until the main page is requested:

```
<jsp:include page="${param.nextPage}" />
```

Here you can see how the value of the `page` attribute isn’t known until the main JSP containing the `<jsp:include>` is requested and the value is obtained by extracting a request parameter using the JSP 2.0 EL. In other words, it’s possible to create dynamic content that is so dynamic, its content isn’t known until request time!

### The `<jsp:useBean>` Action

Before any JavaBean component can be manipulated from a JSP page, it’s first necessary to obtain an instance of the JavaBean, either by retrieving a preexisting bean from one of the available JSP scopes or by creating a new instance. Either of these options could potentially take several lines of scripting code, especially if the JavaBean needs to be initialized before use, which as you’ve seen, can clutter the JSP.

The `<jsp:useBean>` action is specifically designed to simplify this process. This standard action associates an instance of a Java object (our JavaBean) that is defined with a given `scope` and `id` and makes it available as a scripting variable of the same `id`. The `<jsp:useBean>` action is highly flexible and its exact functionality is controlled by the attributes passed to the action. When used correctly, this action can greatly reduce the amount of scriptlet code that would otherwise be required.

The syntax for the `<jsp:useBean>` action is as follows:

```
<jsp:useBean id="name" scope="page|request|session|application" typeSpec/>
```

where:

```
typeSpec ::= class="className"           |
             class="className" type="typeName" |
             beanName="beanName" type="typeName" |
             type="typeName" beanName="beanName" |
                                     type="typeName"
```

Before you see an example of the action at work let's first consider a scriptlet-based alternative that will print out the current time of day (`dateScriptlet.jsp`):

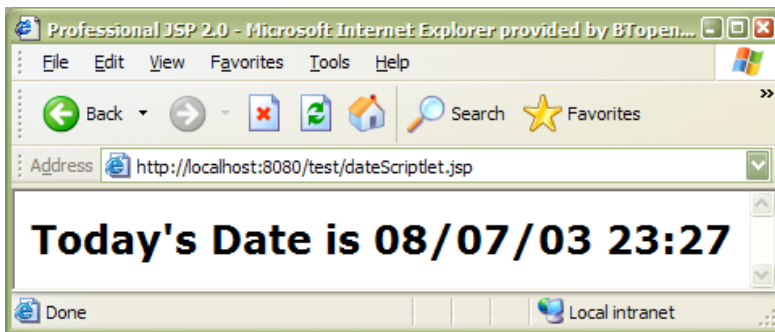
```
<%@ page import="java.util.Date, java.text.DateFormat"%>
<html>
  <head>
    <title>Professional JSP 2.0</title>
  </head>
  <body style="font-family:verdana;font-size:10pt;">
    <%
      DateFormat df = DateFormat.getInstance();
      Date today = new Date();
    %>

    <h2>Today's Date is <%= df.format(today) %></h2>
  </body>
</html>
```

As you can see, this JSP simply imports the `java.util.Date` and `java.text.DateFormat` classes for use and uses a scriptlet to initialize the `Date` and `DateFormat` objects. Some simple template text is used to construct an HTML page and finally a JSP expression is used to format the `Date` object.

Save the code printed earlier into a file called `dateScriptlet.jsp` beneath the test web-application folder in the normal manner. Open the following page in your browser, and the output should be similar to the following screen shot, displaying the correct date and time:

<http://localhost:8080/test/dateScriptlet.jsp>



Although the previous example is perfectly functional, you can see the problems a web designer with no Java skills would have understanding even these simple scriptlets. Another possible problem could be that the JSP page won't be compatible with the tools used by the web designer because she's used to XML-type languages. In a more complex example you can imagine how the problem gets worse and worse.

Let's now see how you can encapsulate the previous date-formatting functionality into a JavaBean component and make use of the `<jsp:useBean>` action to solve the problems mentioned earlier:

```

package com.apress.projsp20.ch01;

import java.util.Date;
import java.text.*;

public class DateFormatBean {
    private DateFormat dateFormat;
    private Date date;

    public DateFormatBean() {
        dateFormat = DateFormat.getInstance();
        date = new Date();
    }

    public String getDate() {
        return dateFormat.format(date);
    }

    public void setDate(Date date) {
        this.date = date;
    }

    public void setFormat(String format) {
        this.dateFormat = new SimpleDateFormat(format);
    }
}

```

As you can see, this simple JavaBean component (`DateFormatBean.java`) initializes itself with a default date and format on initialization as well as providing custom methods to set a different date format or time. When all the initialization is completed, the `getDate()` method simply returns the predefined date in the given date format.

Let's take a look at how simple it is to use the `<jsp:useBean>` to initialize an instance of the `DateFormatBean` as opposed to the scriptlet approach:

```

<html>
  <head>
    <title>Professional JSP 2.0 </title>
  </head>
  <body style="font-family:verdana;font-size:10pt;">
    <jsp:useBean id="date" class="com.apress.projsp20.ch01.DateFormatBean"/>

    <h2>Today's Date is <%= date.getDate()%></h2>
  </body>
</html>

```

As you can see in the code (`dateBean.jsp`), the `<jsp:useBean>` creates an instance of the JavaBean class and stores a reference to it in a scripting variable, which is called `date`. All this without a single scriptlet! A simple JSP expression is used to call the `getDate()` method to retrieve the formatted date. It should be no surprise to learn that the output is exactly the same as in the earlier example—only the implementation is different.

## The `<jsp:getProperty>` and `<jsp:setProperty>` Actions

You saw from the `<jsp:useBean>` action how simple it is to work with JavaBeans from inside the JSP pages. It should probably be of little surprise to learn that there are also standard actions designed to manipulate and retrieve the attributes of these JavaBeans, again without the need for scriptlets or expressions!

As the name suggests, the `<jsp:getProperty>` tag is used to retrieve or access the existing properties of a bean instance. Any bean properties that are retrieved using this tag are automatically converted to a `String` and are placed into the implicit `out` variable, as output.

The syntax for this tag is

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

As you can see, the `<jsp:getProperty>` tag has two attributes, `name` and `property`, both of which must be present. The `name` attribute is used to reference the name of the JavaBean instance on which the `property` attribute exists. This attribute will search all available JSP scopes until the named JavaBean is found. Should the tag fail to locate the requested JavaBean, then a suitable exception will be thrown at request time.

As you can see, this tag is relatively simple in the functionality that it provides, but to make use of it in a JSP, you must ensure that the JavaBean has already been made available to the JSP engine through a previously declared `<jsp:useBean>` tag or a similar. Without the inclusion of this extra tag, neither the `<jsp:getProperty>` nor the `<jsp:setProperty>` tag will function as expected.

While the last example you saw was a great improvement over the earlier scriptlet-based example, you still relied on the use of a JSP expression to access the `date` property from the `DateFormatBean`. You can use the `<jsp:getProperty>` action instead of the expression so that the entire JSP is XML-based.

Let's take a look at the changes to the earlier example that would be necessary (`dateBean_getProperty.jsp`):

```
<html>
  <head>
    <title>Professional JSP 2.0 </title>
  </head>
  <body style="font-family:verdana;font-size:10pt;">
    <jsp:useBean id="date" class="com.apress.projsp20.ch01.DateFormatBean"/>
    <h2>Today's Date is <jsp:getProperty name="date" property="date"/></h2>
  </body>
</html>
```

Again, this is a better solution. Yet you still haven't changed the content returned, just its implementation.

To complement the `<jsp:getProperty>` action, the `<jsp:setProperty>` tag can be used to set the value of an attribute inside a JavaBean. The `<jsp:setProperty>` action is somewhat more flexible and provides the ability to set properties based on request attributes, and so on. In its simplest form the action may be used as follows:

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

As you can see the `name` and `property` attributes are used exactly in the same way as with the `<jsp:getProperty>` action; the additional `value` attribute simply indicates the new value to set the `JavaBean` property to.

Although the `<jsp:setProperty>` action can be used anywhere within a JSP page, it's often used as a nested action inside the body content of the `<jsp:useBean>` action. The consequence of this is that the nested `<jsp:setProperty>` action will only be executed the first time the `<jsp:useBean>` instantiates a `JavaBean`. If an existing bean is located in any one of the JSP scopes then the nested action won't be called.

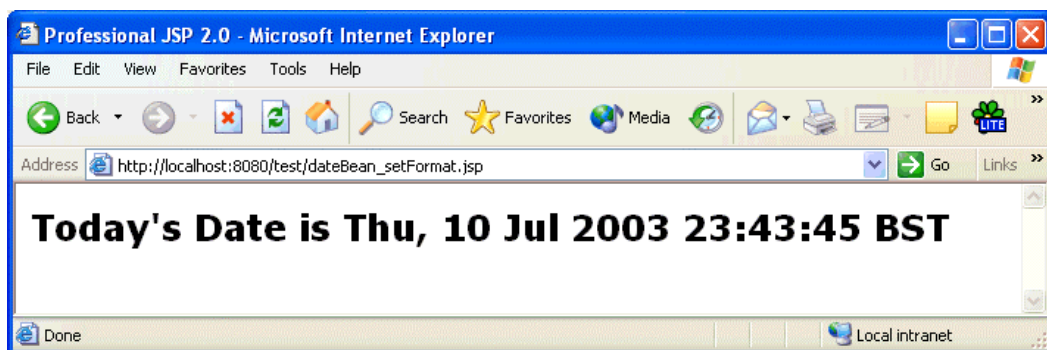
If you recall the listing for the `DateFormatBean` earlier there were two methods that additionally set the date and the date format properties to custom values. Let's make use of the `<jsp:setProperty>` action to set the date format to a different value from that in our previous example (shown in `dateBean_setProperty.jsp`).

```
<html>
  <head>
    <title>Professional JSP 2.0</title>
  </head>
  <body style="font-family:verdana;font-size:10pt;">
    <jsp:useBean id="date" class="com.apress.projsp20.ch01.DateFormatBean">
      <jsp:setProperty name="date" property="format"
        value="EEE, d MMM yyyy HH:mm:ss z"/>
    </jsp:useBean>
    <h2>Today's Date is <jsp:getProperty name="date" property="date"/></h2>
  </body>
</html>
```

If you want to use a compiled `JavaBean` in a JSP web application, the JSP engine, for example Tomcat, needs to know where to look for it. By default Tomcat (and any other servlet container) checks for classes in the `\WEB-INF\classes` directory under the web application directory, and any subdirectories of this. So, for our test web application Tomcat would look for `JavaBeans` in the `\webapps\test\WEB-INF\classes` directory, and all directories below this.

1. Create the directory structure `webapps\test\WEB-INF\classes\com\apress\projsp20\ch01`.
2. Then create a new file called `DateFormatBean.java` and enter the code from earlier in this section and compile it.
3. Next create the `dateBean.jsp`, `dateBean_getProperty.jsp`, and `dateBean_setProperty.jsp` pages beneath the test web application folder, and enter the code (also labeled earlier in this section).

Start Tomcat, open a browser, and run the `dateBean.jsp`, `dateBean_getProperty.jsp`, and `dateBean_setProperty.jsp` pages as you did earlier. Notice that this time, when you use the `dateBean_setProperty.jsp` page, you have changed the date format the JSP generates!



### The `<jsp:forward>` Action

Another very handy action available to JSP page authors is the `<jsp:forward>` action, which not surprisingly is used to forward the current request to another resource such as a static resource, a JSP page, or a servlet in the same context as the containing JSP, for processing.

The syntax for the action is as follows:

```
<jsp:forward page="relativeURL" />
```

Any buffered content that was written before the call to the `<jsp:forward>` action will be ignored. If any buffered content has already been flushed (sent to the client) then the call will result in an `IllegalStateException`.

Note that nested `<jsp:param>` actions may be used in the `<jsp:forward>` action in the same way as with the `<jsp:include>` action to pass additional request parameters to the new resource.

For example:

```
<jsp:forward page="/pages/login.jsp">
  <jsp:param name="userName" value="Dan" />
</jsp:forward>
```



## Custom Actions

Earlier you learned about the potential problems created by introducing too much (if any at all!) scriptlet code into a JSP page. Overuse of scriptlets complicates the lives of JSP developers as well as non-Java-speaking UI designers alike.

You've also seen how many of the problems associated with scriptlet code can be alleviated by encapsulating some of the ugly scriptlet code into JavaBean components and manipulating them using some of the standard actions. While this approach is far superior to the scriptlets approach, it's not the only available solution.

Custom actions are another mechanism for encapsulating functionality into reusable components for use inside JSP pages. Unlike JavaBean components, custom actions have full access to their environment (such as the request and session objects), which makes it far easier to provide functionality suitable for a web site. A good example of a custom action could be performing some calculations where the result is locale sensitive, such as being dependent on the language or number format. A JavaBean component has no idea about the environment in which it's run and therefore a developer would have to work a little harder to get the same functionality. That isn't to say that JavaBean components have no advantages. For one, they're by far the best mechanism for representing business objects and storing state, because they don't care about their environment!

Custom actions are packaged together (usually with several similar or complementary actions) into a tag library that must be registered with a JSP container via its TLD file, which advertises the services provided by the tag library. After a tag library is successfully installed inside a JSP container, the library must be imported using the `taglib` directive you saw earlier before any of the action it provides may be used.

The following example demonstrates the use of a custom action called `foo` from a tag library configured by a web applications deployment descriptor called `myTagLib`:

```
<%@ taglib uri="/myTagLib" prefix="myPrefix" %>

<myPrefix:foo>
    ...
</myPrefix:foo>
```

You can see from the previous example how a tag library must be imported before any of the actions it provides (in this case the `foo` action) may be used on the page. Notice how a prefix is used to provide a namespace for actions from one tag library to another.

You'll learn more about creating and using custom actions, including some of the new JSP 2.0 features for simplifying their creation in Chapters 5, 6, and 7.

## JavaServer Pages Standard Tag Library Actions

You saw previously how useful the standard actions included in the JSP specification are to page authors; well, the JSTL takes this idea a step further and signifies a new phase for JSP page authors.

The JSTL specification was first released in June 2002 with the sole purpose of making JSP pages easier to write. The JSTL provides four new tag libraries that may be used in a similar manner to the standard tags you saw earlier:

- ❑ Core
- ❑ Internationalization (I18n) and Formatting
- ❑ XML
- ❑ SQL

As the names suggest, each library contains a host of useful actions, which are suitable for many of the tasks that JSP page authors are continually having to code manually, such as conditional statements and looping, formatting based on locales, XML manipulation, and database access.

The JSTL also makes use of the new JSP 2.0 EL, which makes the actions even easier to use, especially for a developer unfamiliar with Java syntax. For a more in-depth look at the JSTL and how its tags can be controlled, take a look at Chapter 4.

## Summary

Hopefully this chapter has provided you with a general feel for where JSP technology fits within the J2EE and how it fits with regard to the other web components such as servlets, tag libraries, and JavaBeans, which exist in the J2EE web tier for providing dynamic web-based content.

You were also introduced to some of the most popular JSP architectures that are regularly used for designing the modern web application so that you can hopefully see the bigger picture next time you're confronted with design choices. Or maybe this chapter will help you to analyze an existing application.

Lastly, and most importantly, you were introduced to all of the core syntax-level attributes that are available to a JSP page author, including custom actions and the JSTL. Hopefully, this grounding will give you a head start when approaching some of the more complex chapters, which will build on the JSP basics that you've learned so far.