CHAPTER 1

Introducing JavaServer Pages and Tomcat

What makes the Web really useful is its interactivity. By interacting with some remote server, you can find the information you need, do your banking, or buy online. And every time you type something into a web form, an application "out there" interprets your request and prepares a web page to respond. JavaServer Pages (JSP) is a technology that helps you create such dynamically generated pages.

Sun Microsystems introduced the Java servlet application programming interface (API) in June 1997 with the purpose of providing an efficient and easily portable mechanism to develop web pages with dynamic content. In a nutshell, the servlet package defines Java classes to represent requests sent to the server by the remote web browsers and responses traveling in the opposite direction. A servlet is nothing other than a Java object residing on a server that receives requests via the Internet, accesses resources (such as databases), implements the logic to prepare the responses, and sends the responses back to the network.

The Apache Software Foundation (ASF) developed the Apache Tomcat application server to provide an environment in which servlets can execute. Tomcat is also capable of converting JSP documents into servlets.

In this chapter, we'll introduce you to Java servlets and JSP, and we'll show you how they execute together within Tomcat to generate dynamic web pages. We'll barely scratch the surface of both JSP and Tomcat, and we won't even mention JSF. We'll show you how to develop applications with basic tools, rather than in an environment that takes care of most menial tasks and provides sophisticated checking and debugging capabilities. This will give you a better understanding of what modern tools can do for you.

We know that you're eager to jump into the thick of things. Therefore, after briefly describing how JSP-based web applications are structured, we'll show you at once a nontrivial example, without explaining everything beforehand.

We recommend that you first install the software packages as described in Appendix A. You'll then be able to execute the examples and get a feel for them, rather than just go through the code in print.

What Is JSP?

As we said, JSP is a technology that lets you add dynamic content to web pages. Without JSP, you always have to update the appearance or the content of plain static HTML pages by hand. Even if all you want to do is change a date or a picture, you must edit the HTML file and type in your modifications. Nobody is going to do it for you, whereas with JSP, you can make the content depend on many factors, including the time of the day, the information provided by the user, her history of interaction with your web site, and even her browser type. This capability is essential to provide online services in which each customer is treated differently depending on her preferences and requirements. A crucial aspect of providing meaningful online services is for the system to be able to *remember* data associated with the service and its users. That's why databases play an essential role in dynamic web pages. But let's take it one step at a time.

HISTORY

Sun Microsystems introduced JSP in 1999. Developers quickly realized that additional tags would be useful, and the JSP Standard Tag Library (JSTL) was born. JSTL is a collection of custom tag libraries that encapsulate the functionality of many JSP standard applications, thereby eliminating repetitions and making the applications more compact. Together with JSTL also came the JSP Expression Language (EL).

In 2003, with the introduction of JSP 2.0, EL was incorporated into the JSP specification, making it available for custom components and template text, not just for JSTL, as was the case in the previous versions. Additionally, JSP 2.0 made it possible to create custom tag files, thereby perfecting the extensibility of the language.

In parallel to the evolution of JSP, several frameworks to develop web applications became available. In 2004, one of them, JavaServer Faces (JSF), focused on building user interfaces (UIs) and used JSP by default as the underlying scripting language. It provided an API, JSP custom tag libraries, and an expression language.

The Java Community Process (JCP), formed in 1998, released in May 2006 the Java Specification Request (JSR) 245 titled *JavaServer Pages 2.1*, which effectively aligns JSP and JSF technologies. In particular, JSP 2.1 includes a Unified EL (UEL) that merges together the two versions of EL defined in JSP 2.0 and JSF 1.2 (itself specified as JSR 252). Sun Microsystems includes JSP 2.1 in its Java Platform, Enterprise Edition 5 (Java EE 5), finalized in May 2006 as JSR 244. The classes included in EE 5 rely on the general classes that form the Java Platform, Standard Edition 5 (Java SE 5), which is available as Java Runtime Environment (JRE) and Java Development Kit (JDK).

Meanwhile, the servlet technology has evolved, and Sun Microsystems released Servlet 2.5 in September 2005. The JCP formally specified Servlet 2.5 as an updated version of JSR 152 in May 2006.

In summary, Java EE 5 includes JSP 2.1, which in turn specifies a UEL consistent with JSF 1.2, while Java SE 5 provides the foundation classes, and Servlet 2.5 includes a library to handle HTTP requests.

Viewing a Web Page

To understand JSP, you first need to have a clear idea of what happens when you ask your browser to view a web page, either by typing a URL in the address field of your browser or by clicking on a hyperlink. Figure 1-1 shows you how it works.

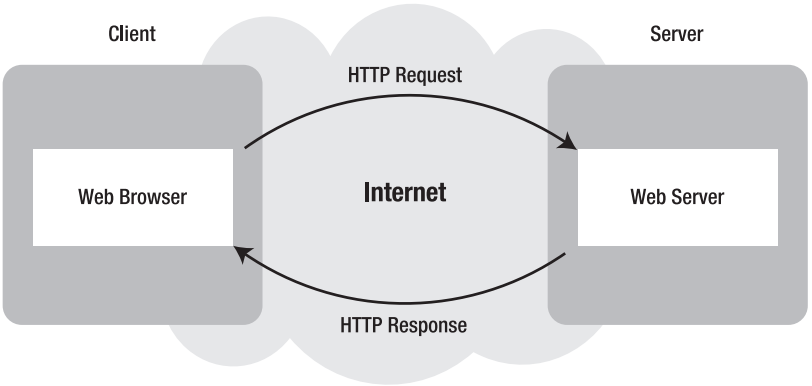

Figure 1-1. *Viewing a plain HTML page*

The following steps show what happens when you request your browser to view a web page:

1. When you type an address such as `http://www.website.com/path/whatever.html` into the address field, your browser first resolves `www.website.com` (i.e., the name of the web server) into the corresponding Internet Protocol (IP) address, usually by asking the domain name server provided by your Internet Service Provider (ISP). Then your browser sends an HTTP request to the newly found IP address to receive the content of the file identified by `/path/whatever.html`.
2. In reply, the web server sends an HTTP response containing a plain-text HTML page. Images and other nontextual components, such as applets and sounds, only appear in the page as references.
3. Your browser receives the response, interprets the HTML code contained in the page, requests the nontextual components from the server, and displays the lot.

Viewing a JSP Page

With JSP, the web page doesn't actually exist on the server. As you can see in Figure 1-2, the server creates it fresh when responding to each request.

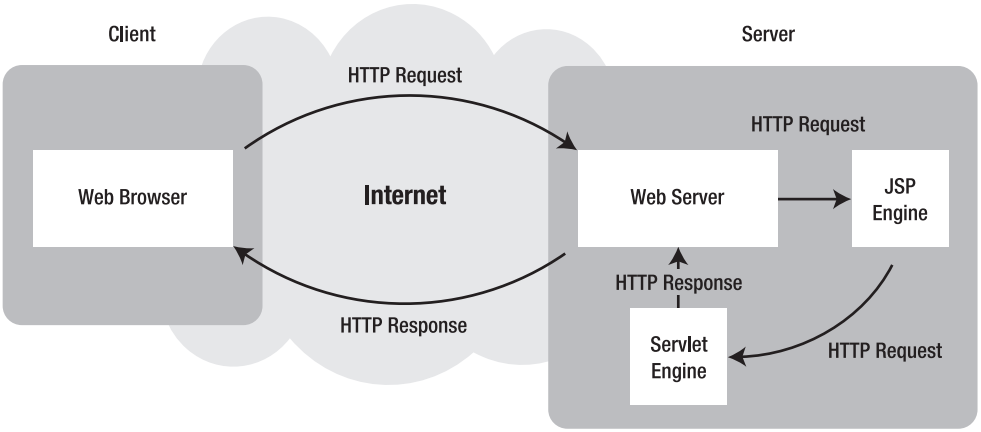

Figure 1-2. *Viewing a JSP page*

The following steps explain how the web server creates the web page:

1. As with a normal page, your browser sends an HTTP request to the web server. This doesn't change with JSP, although the URL probably ends in `.jsp` instead of `.html`.

2. The web server is not a normal server, but rather a Java server, with the extensions necessary to identify and handle Java servlets. The web server recognizes that the HTTP request is for a JSP page and forwards it to a JSP engine.

3. The JSP engine loads the JSP page from disk and converts it into a Java servlet. From this point on, this servlet is indistinguishable from any other servlet developed directly in Java rather than JSP, although the automatically generated Java code of a JSP servlet is difficult to read, and you should never modify it by hand.

4. The JSP engine compiles the servlet into an executable class and forwards the original request to a servlet engine. Note that the JSP engine only converts the JSP page to Java and recompiles the servlet if it finds that the JSP page has changed since the last request. This makes the process more efficient than with other scripting languages (such as PHP) and therefore faster.

5. A part of the web server called the *servlet engine* loads the `Servlet` class and executes it. During execution, the servlet produces an output in HTML format, which the servlet engine passes to the web server inside an HTTP response.

6. The web server forwards the HTTP response to your browser.

7. Your web browser handles the dynamically generated HTML page inside the HTTP response exactly as if it were a static page. In fact, static and dynamic web pages are in the same format.

You might ask, "Why do you say that with JSP, the page is created fresh for each request, if it is only recompiled when it has been updated?"

What reaches your browser is the *output* generated by the servlet (by the converted and compiled JSP page), not the JSP page itself. The same servlet produces different outputs depending on the parameters of the HTTP request and other factors. For example, suppose you're browsing the products offered by an online shop. When you click on the image of a product, your browser generates an HTTP request with the product code as a parameter. As a result, the servlet generates an HTML page with the description of that product. The server doesn't need to recompile the servlet for each product code.

The servlet queries a database containing the details of all the products, obtains the description of the product you're interested in, and formats an HTML page with that data. This is what dynamic HTML is all about!

Plain HTML is not capable of interrogating a database, but Java is, and JSP gives you the means of including snippets of Java inside an HTML page.

Hello World!

A small example of JSP will give you a more practical idea of how JSP works. Let's start once more from HTML. Listing 1-1 shows you a plain HTML page to display "Hello World!" in your browser's window.

Listing 1-1. *hello.html*

```
<html>
<head><title>Hello World static HTML</title></head>
<body>
Hello World!
</body>
</html>
```

Create this folder to store `hello.html`:

```
C:\Program Files\Apache Software Foundation\Tomcat 6.0\webapps\hello\
```

Type this URL to see the web page in the browser:

```
http://localhost:8080/hello/hello.html
```

Normally, to ask your browser to check that the syntax of the page conforms to the XHTML standard of the World Wide Web Consortium (W3C), you would have to start the page with the following three lines:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

You'd also have to replace this line:

```
<html>
```

with this line:

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
```

However, for this simple example, we prefer to keep the code to what's essential. Figure 1-3 shows you how this page will appear in your browser.

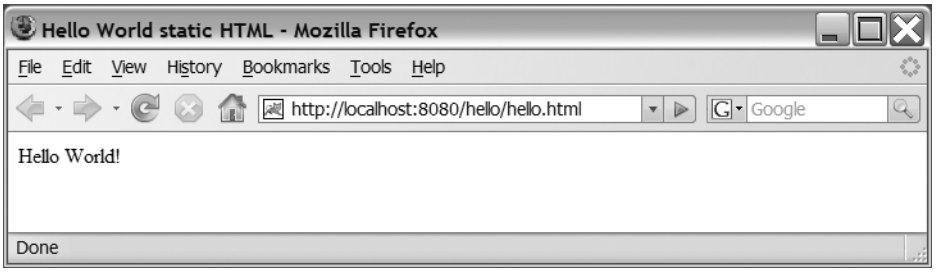

Figure 1-3. *"Hello World!" in plain HTML*

If you view the page source through your browser, not surprisingly you'll see exactly what's shown in Listing 1-1. To obtain exactly the same result with a JSP page, you only need to insert a JSP directive before the first line, as shown in Listing 1-2, and change the file extension from `html` to `jsp`.

Listing 1-2. "*Hello World!" in a Boring JSP Page*

```
<%@page language="java" contentType="text/html"%>
<html>
<head><title>Hello World not-so-dynamic HTML</title></head>
<body>
Hello World!
</body>
</html>
```

■ **Note** Microsoft Internet Explorer 7 only interprets JSP pages that include the `page contentType` directive.

Obviously, there isn't much point in using JSP for such a simple page. It only pays to use JSP if you use it to include dynamic content. Check out Listing 1-3 for something more juicy.

Listing 1-3. *hello.jsp*

```
<%@page language="java" contentType="text/html"%>
<html>
<head><title>Hello World dynamic HTML</title></head>
<body>
Hello World!
<%
  out.println("<br/>Your IP address is " + request.getRemoteAddr());

  String userAgent = request.getHeader("user-agent");
  String browser = "unknown";

  out.print("<br/>and your browser is ");
  if (userAgent != null) {
    if (userAgent.indexOf("MSIE") > -1) {
      browser = "MS Internet Explorer";
      }
    else if (userAgent.indexOf("Firefox") > -1) {
      browser = "Mozilla Firefox";
      }
    }
  out.println(browser);
  %>
</body>
</html>
```

As with `hello.html`, you can view `hello.jsp` by placing it in the `webapps\hello\` folder.

The code within the <% ... %> pair is a scriptlet written in Java. When Tomcat's JSP engine interprets this module, it creates a Java servlet containing 92 lines of code, among which you can find those shown in Listing 1-4 (with some indentation and empty lines removed).

Listing 1-4. *Java Code from the "Hello World!" JSP Page*

```
out.write("\r\n");
out.write("<html>\r\n");
out.write("<head><title>Hello World dynamic HTML</title></head>\r\n");
out.write("<body>\r\n");
out.write("Hello World!\r\n");
out.write('\r');
out.write('\n');
out.println("<br/>Your IP address is " + request.getRemoteAddr());
String userAgent = request.getHeader("user-agent");
String browser = "unknown";
out.print("<br/>and your browser is ");
```

```
if (userAgent != null) {
  if (userAgent.indexOf("MSIE") > -1) {
    browser = "MS Internet Explorer";
    }
  else if (userAgent.indexOf("Firefox") > -1) {
    browser = "Mozilla Firefox";
    }
  }
out.println(browser);
out.write("\r\n");
out.write("</body>\r\n");
out.write("</html>\r\n");
```

As we said before, this servlet executes every time a browser sends a request to the server. However, before the code shown in Listing 1-4 executes, the variable `out` is bound to the content of the response. As a result, everything written to `out` ends up in the HTML page that you'll see in your browser. The scriptlet, shown in bold, is copied to the servlet. Everything else is written to the output. This should clarify how the mix of HTML and Java is achieved in a JSP page.

As the variable `out` is defined in each servlet, you can use it within any JSP module to insert something into the response. Another such "global" JSP variable is `request` (of type `HttpServletRequest`). The request contains the IP address from which the request was originated—that is, of the remote computer with the browser (remember that this code runs on the server). To extract the address from the request, you only need to execute its method `getRemoteAddr()`. The request also contains information about the browser. When some browsers send a request, they provide somewhat misleading information, and the format is complex. However, the code in Listing 1-4 shows you how to determine whether you're using Microsoft Internet Explorer or Mozilla Firefox. Figure 1-4 shows the generated page as it appears in a browser.

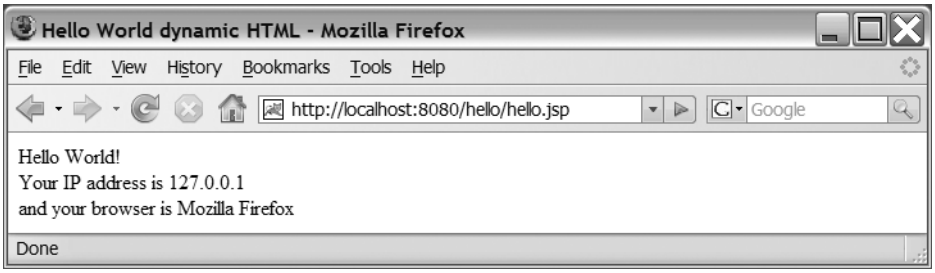

Figure 1-4. *"Hello World!" in JSP*

Notice that IP address `127.0.0.1` is consistent with the host `localhost`. And just in case you want to see that the HTML is indeed dynamic, check out Figure 1-5. Incidentally, the method you used in `hello.jsp` to identify the Internet Explorer from the user agent is the official one provided by Microsoft.

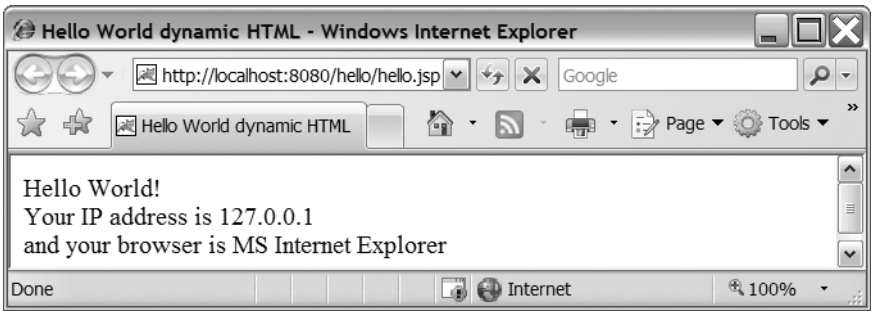

Figure 1-5. *"Hello World!" in JSP with Internet Explorer*

JSP Application Architectures

The insertion of Java code into HTML modules opens up the possibility of building dynamic web pages, but to say that it is possible doesn't mean that you can do it efficiently and effectively. If you start developing complex applications by means of scriptlets enclosed in <% ... %> pairs, you'll rapidly reach the point where the code will become difficult to maintain. The key problem with mixing Java and HTML, as in "Hello World!," is that the application logic and the way the information is presented in the browser are mixed. In general, the business application designers and the web page designers are different people with complementary and only partly overlapping skills. While application designers are experts in complex algorithms and databases, web designers focus on page composition and graphics. The architecture of your JSP-based applications should reflect this distinction. The last thing you want to do is blur the roles within the development team and end up with everybody doing what somebody else is better qualified to do.

The Model 1 Architecture

The first solution to this problem that developers found was to define the JSP Model 1 architecture, in which the application logic is implemented in Java classes (i.e., Java beans), which you can then use within JSP (see Figure 1-6).

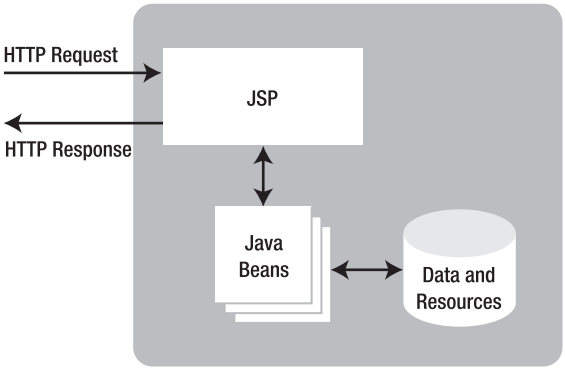

Figure 1-6. *The JSP Model 1 architecture*

Model 1 is acceptable for applications containing up to a few thousand lines of code, and especially for programmers, but the JSP pages still have to handle the HTTP requests, and this can cause headaches for the page designers.

The Model 2 Architecture

A better solution, also suitable for larger applications, is to separate application logic and page presentation. This solution comes in the form of the JSP Model 2 architecture, also known as the model-view-controller (MVC) design pattern (see Figure 1-7).

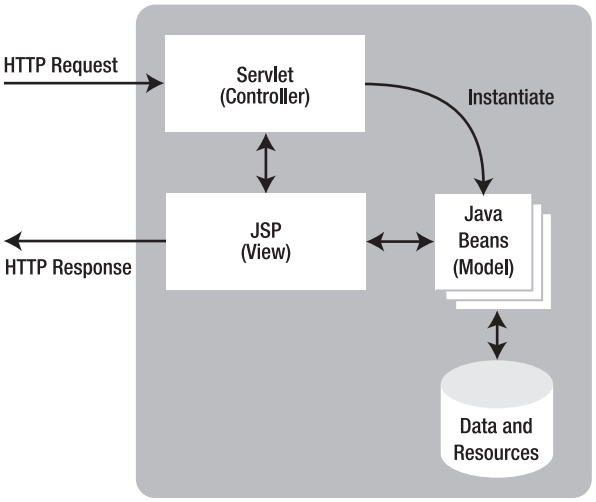

Figure 1-7. *JSP Model 2 architecture*

With this model, a servlet processes the request, handles the application logic, and instantiates the Java beans. JSP obtains data from the beans and can format the response without having to know anything about what's going on behind the scenes. To illustrate this model, we will describe a sample application called Ebookshop, a small application to sell books online. Ebookshop is not really functional, because the list of books is hard-coded in the application rather than stored in a database. Also, nothing happens once you confirm the order. However, this example serves the purpose of showing you how Model 2 lets you separate business logic and presentation.

Figure 1-8 shows the Ebookshop's home page, which you see when you type `http://localhost:8080/ebookshop` in your browser's address field.

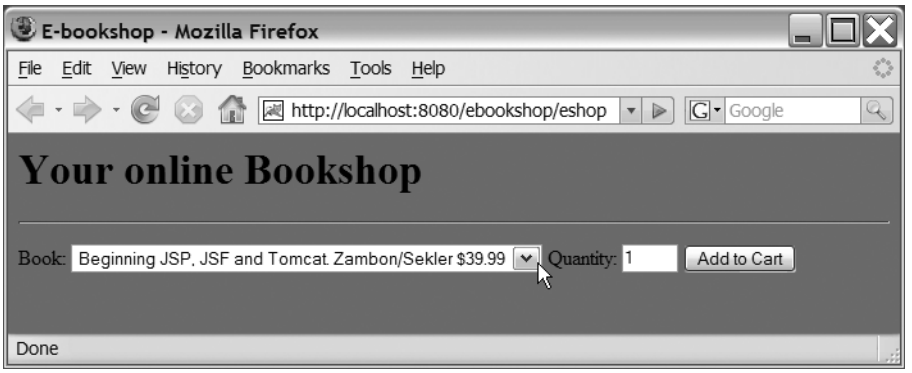

Figure 1-8. *The Ebookshop home page*

You can select a book by clicking on the drop-down list, as shown in the picture, type in the number of copies you need, and then click the `Add to Cart` button. Every time you do so, the content of your shopping cart appears at the bottom of the window, as shown in Figure 1-9.

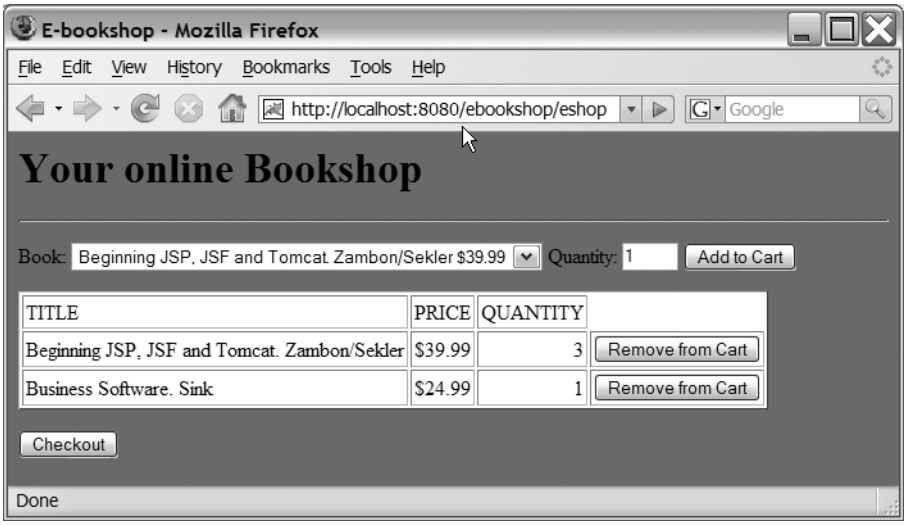

Figure 1-9. *The Ebookshop home page displaying the shopping cart*

You can remove an item from the shopping cart or go to the checkout. If you add additional copies of a book to the cart, the quantity in the cart will increase accordingly.

If you click on the `Checkout` button, you'll see the page shown in Figure 1-10.

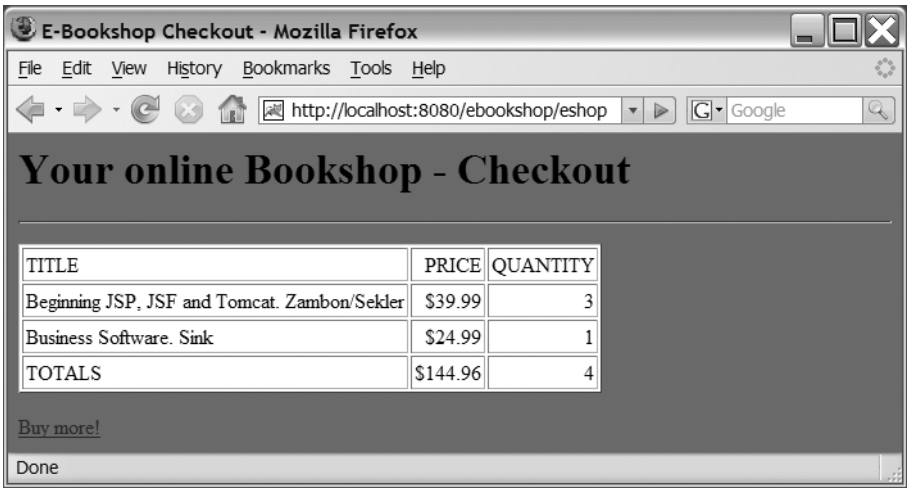

Figure 1-10. *The Ebookshop checkout page*

If you click on the `Buy more!` link, you'll go back to the home page with an empty shopping cart, ready for more shopping.

The Ebookshop Home Page

Listing 1-5 shows the home page `http://localhost:8080/ebookshop/index.jsp`. For your convenience, we've highlighted the JSP directives and scriptlets in bold.

Listing 1-5. *The Ebookshop Home Page index.jsp*

```
<%@page language="java" contentType="text/html"%>
<%@page session="true" import="java.util.*, ebookshop.Book"%>
<html>
<head>
  <title>E-bookshop</title>
  <style type="text/css">
    body {background-color:gray; font-size=10pt;}
    H1 {font-size:20pt;}
    table {background-color:white;}
    </style>
  </head>
<body>
  <H1>Your online Bookshop</H1>
  <hr/><p/>
<%  // Scriptlet 1: check whether the book list is ready
  Vector booklist = (Vector)session.getValue("ebookshop.list");
  if (booklist == null) {
    response.sendRedirect("/ebookshop/eshop");
    }
```

```
 else {
  %>
    <form name="addForm" action="eshop" method="POST">
      <input type="hidden" name="do_this" value="add">
      Book:
      <select name=book>
<%  // Scriptlet 2: copy the book list to the selection control
        for (int i = 0; i < booklist.size(); i++) {
          out.println("<option>" + (String)booklist.elementAt(i) + "</option>");
          }
  %>
       </select>
      Quantity: <input type="text" name="qty" size="3" value="1">
      <input type="submit" value="Add to Cart">
      </form>
    <p/>
<%  // Scriptlet 3: check whether the shopping cart is empty
    Vector shoplist = (Vector)session.getValue("ebookshop.cart");
    if (shoplist != null  &&  shoplist.size() > 0) {
  %>
      <table border="1" cellpadding="2">
      <tr>
      <td>TITLE</td>
      <td>PRICE</td>
      <td>QUANTITY</td>
      <td></td>
      </tr>
<%  // Scriptlet 4: display the books in the shopping cart
      for (int i = 0; i < shoplist.size(); i++) {
        Book aBook = (Book)shoplist.elementAt(i);
  %>
        <tr>
          <form name="removeForm" action="eshop" method="POST">
            <input type="hidden" name="position" value="<%=i%>">
            <input type="hidden" name="do_this" value="remove">
            <td><%=aBook.getTitle()%></td>
            <td align="right">$<%=aBook.getPrice()%></td>
            <td align="right"><%=aBook.getQuantity()%></td>
            <td><input type="submit" value="Remove from Cart"></td>
            </form>
          </tr>
<%
        } // for (int i..
  %>
      </table>
```

```
      <p/>
      <form name="checkoutForm" action="eshop" method="POST">
        <input type="hidden" name="do_this" value="checkout">
        <input type="submit" value="Checkout">
        </form>
<%
      } // if (shoplist..
    } // if (booklist..else..
  %>
  </body>
</html>
```

First, `index.jsp` (shown in Scriptlet 1) checks whether the list of books to be sold is available and, if it isn't, it passes the control to the servlet, which then must initialize the book list. In reality, the book list would be very long and kept in a database. Note that JSP doesn't *need to know* where the list is kept. This is the first hint at the fact that application logic and presentation are separate. You'll see later how the servlet fills in the book list and returns control to `index.jsp`. For now, let's proceed with the analysis of the home page.

If Scriptlet 1 discovers that the book list exists, it copies it into the `select` control one by one (as shown in Scriptlet 2). Notice how JSP simply creates each `option` by writing to the `out` stream. When the buyer clicks on the `Add to Cart` button after selecting a title and setting the number of copies, the home page posts a request to the servlet with the address `eshop` and with the hidden parameter `do_this` set to `add`. Once more, the servlet takes care of updating or creating the shopping cart by instantiating the class `Book` for each new book added to the cart. This is application logic, not presentation of information.

Scriptlet 3 checks whether a shopping cart exists. `index.jsp`, being completely data-driven, doesn't remember what has happened before, so it runs every time from the beginning. Therefore, it checks for the presence of a shopping cart even when the buyer sees the book list for the very first time.

Scriptlet 4 displays the items in the shopping cart, each one with its own form. If the buyer decides to delete an entry, `index.jsp` sends a request to the servlet with the hidden parameter `do_this` set to `remove`.

The sole purpose of the last two scriptlets is to close the curly brackets of `if`s and `for`s. However, notice that the form to ask the servlet to do the checkout is only displayed to the buyer when the shopping cart isn't empty. If the buyer clicks on the `Checkout` button, `index.jsp` will send a request to the servlet with the hidden parameter `do_this` set to `checkout`.

Finally, notice that some elements enclosed within `<%=` and `%>` are mixed inside the normal HTML. They are `<%=i%>`, `<%=aBook.getTitle()%>`, `<%=aBook.getPrice()%>`, and `<%=aBook.getQuantity()%>`. These elements let you embed values resulting from JSP expressions in HTML without having to execute scriptlets. The first expression, `<%=i%>`, is the position of the book within the shopping cart. The other three are the execution of methods of an object of type `Book`, which the servlet instantiated for each new book added to the cart.

You've probably noticed that the address shown in the browser is `http://localhost:8080/ebookshop/eshop`. This is actually the address of the Java servlet that controls the application.

The Ebookshop Servlet

Listing 1-6 shows the source code of the servlet.

Listing 1-6. *ShoppingServlet.java*

```
package ebookshop;
import java.util.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import ebookshop.Book;

public class ShoppingServlet extends HttpServlet {

  public void init(ServletConfig conf) throws ServletException  {
    super.init(conf);
    }

  public void doGet (HttpServletRequest req, HttpServletResponse res)
      throws ServletException, IOException {
    doPost(req, res);
    }

  public void doPost (HttpServletRequest req, HttpServletResponse res)
      throws ServletException, IOException {
    HttpSession session = req.getSession(true);
    Vector<Book> shoplist =
      (Vector<Book>)session.getAttribute("ebookshop.cart");
    String do_this = req.getParameter("do_this");
    if (do_this == null) {
      Vector<String> blist = new Vector<String>();
      blist.addElement("Beginning JSP, JSF and Tomcat. Zambon/Sekler $39.99");
      blist.addElement("Beginning JBoss Seam. Nusairat $39.99");
      blist.addElement("Founders at Work. Livingston $25.99");
      blist.addElement("Business Software. Sink $24.99");
      blist.addElement("Foundations of Security. Daswani/Kern/Kesavan $39.99");
      session.setAttribute("ebookshop.list", blist);
      ServletContext    sc = getServletContext();
      RequestDispatcher rd = sc.getRequestDispatcher("/");
      rd.forward(req, res);
      }
```

```
else {
  if (do_this.equals("checkout"))  {
    float dollars = 0;
    int   books = 0;
    for (int i = 0; i < shoplist.size(); i++) {
      Book  aBook = (Book)shoplist.elementAt(i);
      float price = aBook.getPrice();
      int   qty = aBook.getQuantity();
      dollars += price * qty;
      books += qty;
      }
    req.setAttribute("dollars", new Float(dollars).toString());
    req.setAttribute("books", new Integer(books).toString());
    ServletContext    sc = getServletContext();
    RequestDispatcher rd = sc.getRequestDispatcher("/Checkout.jsp");
    rd.forward(req, res);
    } // if (..checkout..
  else {
    if (do_this.equals("remove")) {
      String pos = req.getParameter("position");
      shoplist.removeElementAt((new Integer(pos)).intValue());
      }
    else if (do_this.equals("add")) {
      boolean found = false;
      Book aBook = getBook(req);
      if (shoplist == null) {  // the shopping cart is empty
        shoplist = new Vector<Book>();
        shoplist.addElement(aBook);
        }
      else {  // update the #copies if the book is already there
        for (int i = 0; i < shoplist.size() && !found; i++) {
          Book b = (Book)shoplist.elementAt(i);
          if (b.getTitle().equals(aBook.getTitle())) {
            b.setQuantity(b.getQuantity() + aBook.getQuantity());
            shoplist.setElementAt(b, i);
            found = true;
            }
          } // for (i..
        if (!found) {  // if it is a new book => Add it to the shoplist
          shoplist.addElement(aBook);
          }
        } // if (shoplist == null) .. else ..
      } // if (..add..
```

```
        session.setAttribute("ebookshop.cart", shoplist);
        ServletContext sc = getServletContext();
        RequestDispatcher rd = sc.getRequestDispatcher("/");
        rd.forward(req, res);
        } // if (..checkout..else
      } // if (do_this..
    } // doPost

  private Book getBook(HttpServletRequest req) {
    String myBook = req.getParameter("book");
    int    n = myBook.indexOf('$');
    String title = myBook.substring(0, n);
    String price = myBook.substring(n+1);
    String qty = req.getParameter("qty");
    return new Book(title, Float.parseFloat(price), Integer.parseInt(qty));
    } // getBook
  }
```

As you can see, the `init()` method only executes the standard servlet initialization, and the `doGet()` method simply executes `doPost()`, where all the work is done. If you were to remove the `doGet()` method, you would effectively forbid the direct call of the servlet. That is, if you typed `http://localhost:8080/ebookshop/eshop` in your browser, you would receive an error message that says the requested resource isn't available. As it is, you can type the URL with or without trailing `eshop`.

When you analyze `index.jsp`, you can see that it can pass control to the servlet on four occasions, as listed here from the point of view of the servlet:

1. **If no book list exists**: This happens when the buyer types `http://localhost:8080/ebookshop/`. The servlet executes without any parameter, initializes the book list, and passes control straight back to `index.jsp`.

2. **When the buyer clicks on `Add to Cart`**: The servlet executes with `do_this` set to `add` and a parameter containing the book description. Normally, this would be done more elegantly with a reference to the book rather than the whole description, but we want to keep things as simple as possible. The servlet creates a cart if necessary and adds to it a new object of type `Book` or, if the same book is already in the cart, updates its quantity. After that, it passes the control back to `index.jsp`.

3. **When the buyer clicks on `Remove from Cart`**: The servlet executes with `do_this` set to `remove` and a parameter containing the position of the book within the cart. The servlet removes the book in the given position by deleting the object of type `Book` from the vector representing the cart. After that, it passes the control back to `index.jsp`.

4. **When the buyer clicks on `Checkout`**: The servlet executes with `do_this` set to `checkout`. The servlet calculates the total amount of money and the number of books ordered, adds them as attributes to the HTTP request, and passes the control to `Checkout.jsp`, which has the task of displaying the bill.

More on Ebookshop

By now, it should be clear to you how the servlet is in control of the application and how JSP is only used to present the data. To see the full picture, you only need to see `Book.java`, the Java bean used to represent a book, and `Checkout.jsp`, which displays the bill. Listing 1-7 shows the code for `Book.java`.

Listing 1-7. *Book.java*

```
package ebookshop;
public class Book {
  String title;
  float  price;
  int    quantity;
  public Book(String t, float p, int q) {
    title    = t;
    price    = p;
    quantity = q;
    }
  public String getTitle()         { return title; }
  public void   setTitle(String t) { title = t; }
  public float  getPrice()         { return price; }
  public void   setPrice(float p)  { price = p; }
  public int    getQuantity()      { return quantity; }
  public void   setQuantity(int q) { quantity = q; }
  }
```

In a more realistic case, the class `Book` would contain much more information, which the buyer could use to select the book. Also, the class attribute `title` is a misnomer, as it also includes the author names, but you get the idea. Listing 1-8 shows the code for `Checkout.jsp`.

Listing 1-8. *Checkout.jsp*

```
<%@page language="java" contentType="text/html"%>
<%@page session="true" import="java.util.*, ebookshop.Book" %>
<html>
<head>
  <title>E-Bookshop Checkout</title>
  <style type="text/css">
    body {background-color:gray; font-size=10pt;}
    H1 {font-size:20pt;}
    table {background-color:white;}
    </style>
  </head>
<body>
  <H1>Your online Bookshop - Checkout</H1>
  <hr/><p/>
  <table border="1" cellpadding="2">
```

```
    <tr>
      <td>TITLE</td>
      <td align="right">PRICE</td>
      <td align="right">QUANTITY</td>
      </tr>
<%
    Vector shoplist = (Vector)session.getValue("ebookshop.cart");
    for (int i = 0; i < shoplist.size(); i++) {
      Book anOrder = (Book)shoplist.elementAt(i);
 %>
      <tr>
        <td><%=anOrder.getTitle()%></td>
        <td align="right">$<%=anOrder.getPrice()%></td>
        <td align="right"><%=anOrder.getQuantity()%></td>
        </tr>
<%
      }
    session.invalidate();
  %>
    <tr>
      <td>TOTALS</td>
      <td align="right">$<%=(String)request.getAttribute("dollars")%></td>
      <td align="right"><%=(String)request.getAttribute("books")%></td>
      </tr>
    </table>
  <p/>
  <a href="/ebookshop/eshop">Buy more!</a>
  </body>
</html>
```

`Checkout.jsp` displays the shopping cart and the totals precalculated by the servlet, and it invalidates the session so that a new empty shopping cart will be created if the application is restarted from the same browser window.

Note that you could have included the checkout logic in `index.jsp` and made its execution dependent on the presence of the two totals. However, we wanted to show you a more structured application. It's also better design to keep different functions in different JSP modules. In fact, we could have also kept the shopping cart in a separate JSP file. In real life, we would have certainly done so. In addition, we would have saved the styles in a Cascading Style Sheets (CSS) file rather than repeating them in all JSP sources. Finally, there is close to no error checking and reporting. You could easily crash this application.

Before we move on, you'll certainly find it interesting to see the dynamic HTML page, which actually reaches the browser after adding one item to the shopping cart (see Listing 1-9, in which we've removed some empty lines).

Listing 1-9. *HTML Generated by index.jsp*

```
<html>
<head>
  <title>E-bookshop</title>
  <style type="text/css">
    body {background-color:gray; font-size=10pt;}
    H1 {font-size:20pt;}
    table {background-color:white;}
    </style>
  </head>
<body>
  <H1>Your online Bookshop</H1>
  <hr/><p/>
    <form name="addForm" action="eshop" method="POST">
      <input type="hidden" name="do_this" value="add">
      Book:
      <select name=book>
<option>Beginning JSP, JSF and Tomcat. Zambon/Sekler $39.99</option>
<option>Beginning JBoss Seam. Nusairat $39.99</option>
<option>Founders at Work. Livingston $25.99</option>
<option>Business Software. Sink $24.99</option>
<option>Foundations of Security. Daswani/Kern/Kesavan $39.99</option>
        </select>
      Quantity: <input type="text" name="qty" size="3" value="1">
      <input type="submit" value="Add to Cart">
      </form>
    <p/>
      <table border="1" cellpadding="2">
      <tr>
      <td>TITLE</td>
      <td>PRICE</td>
      <td>QUANTITY</td>
      <td></td>
      </tr>
        <tr>
          <form name="removeForm" action="eshop" method="POST">
            <input type="hidden" name="position" value="0">
            <input type="hidden" name="do_this" value="remove">
            <td>Beginning JSP, JSF and Tomcat. Zambon/Sekler </td>
            <td align="right">$39.99</td>
            <td align="right">3</td>
            <td><input type="submit" value="Remove from Cart"></td>
            </form>
          </tr>
      </table>
```

```
      <p/>
      <form name="checkoutForm" action="eshop" method="POST">
        <input type="hidden" name="do_this" value="checkout">
        <input type="submit" value="Checkout">
        </form>
  </body>
</html>
```

Neat, isn't it?

You now have in your hands the full code of a nontrivial Java/JSP application, but you still need to know how to make these four modules work together. For that, you need a web server that translates your JSP pages into Java, executes your servlet, and acts as an interface between your web application and the remote browsers. You guessed it: Tomcat.

What Role Does Tomcat Play in All This?

Tomcat is what makes the Ebookshop application accessible over the Internet. Its latest release (6.0) implements JSP 2.1 and EL 2.1. It obviously requires the Java SE 5 runtime environment, because older releases don't include the correct versions of JSP and EL. The beauty of Tomcat is that it also includes its own HTTP server, so you don't need anything else to handle client requests.

Rather than providing abstract and general explanations, we'll use the Ebookshop example to tell you what to do in practice. You have to organize the following files: `index.jsp`, `Checkout.jsp`, `ShoppingServlet.java`, and `Book.java`. First, create the folder structure shown in Figure 1-11 in C:\Program Files\Apache Software Foundation\Tomcat 6.0\webapps\, and place your four source files as indicated (see Appendix A for the installation procedure of Tomcat).

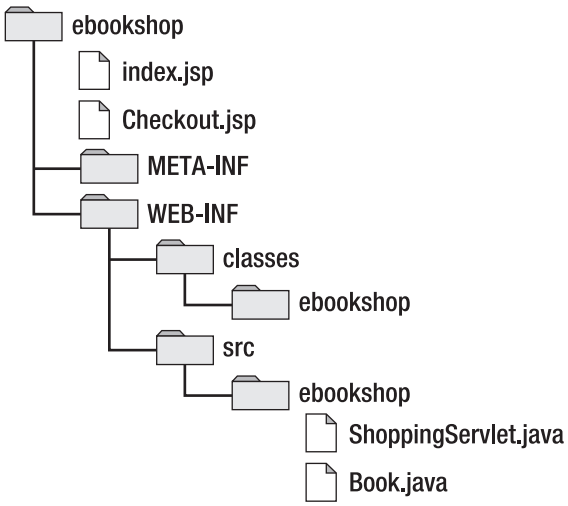

Figure 1-11. *The Ebookshop folder structure*

To get the application to work, you first need to compile the two Java modules. Assuming that you've already installed the appropriate JDK (again, see Appendix A for the instructions), it boils down to executing the Java compiler from the command line. You should write a small batch file rather than always open a DOS window, change the directory, and type the command. Listing 1-10 shows an example of a batch file you could use. Just place it inside the `WEB-INF` folder and double-click it.

Listing 1-10. *compile_it.bat*

```
@echo off
set aname=ebookshop
set /P fname=Please enter the java file name without extension:
set fil=%aname%\%fname%
echo *** compile_it.bat: compile src\%fil%.java
javac -verbose -deprecation -Xlint:unchecked -classpath classes src\%fil%.java
if %errorlevel% GTR 1 goto _PAUSE
echo *** compile_it.bat: move the class to the package directory
move /y src\%fil%.class classes\%fil%.class
:_PAUSE
pause
```

The batch file opens a DOS window automatically and asks you to type the name of a Java file (without the extension). It then compiles the file and moves the resulting class into the `classes\ebookshop\` subfolder. This line invokes the Java compiler with the switches that maximize both the information you get and the checks on your sources:

```
javac -verbose -deprecation -Xlint:unchecked -classpath classes src\%fil%.java
```

Notice the `classpath` switch, which tells the compiler to look for classes in the local directory in addition to the usual places where the Java libraries are kept. This is necessary, because `ShoppingServlet.java` imports the class `Book` and without the `classpath` switch, the compiler wouldn't know where to find it. This also means that you have to compile `Book.java` *before* `ShoppingServlet.java`.

When executing your application, Tomcat looks for classes in the `WEB-INF\classes\` folder immediately inside the root folder of your application (in this case `ebookshop`), which in turn is immediately inside `webapps`. The directory structure inside `WEB-INF\classes` must reflect what you write in the `package` statement at the beginning of the Java sources. In your Java sources, you have this statement:

```
package ebookshop;
```

If you had written this instead:

```
package myApplications.ebookshop;
```

you would have had to insert a `myApplications` folder below classes and above `ebookshop`. To avoid confusion, note that the package name has nothing to do with the name of the application. That is, you could have named the package (and, therefore, the folder below the classes)

qwertyuiop instead of ebookshop. In fact, you could have dispensed with the package statement altogether and placed your classes directly inside the classes folder. Finally, you could have also created a JAR file.

Tomcat automatically converts your JSP files into Java classes the first time they're needed, but before you're ready to go, you still need to write an additional file where you describe the structure of your application to Tomcat. This file *must* be named web.xml and placed in WEB-INF, as you can see in Listing 1-11.

Listing 1-11. *web.xml*

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=~CCC
"http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">
  <display-name>Electronic Bookshop</display-name>
  <description>
    e-bookshop example for
    Beginning JSP, JSF and Tomcat: from Novice to Professional
    </description>
  <servlet>
    <servlet-name>EBookshopServlet</servlet-name>
    <servlet-class>ebookshop.ShoppingServlet</servlet-class>
    </servlet>
  <servlet-mapping>
    <servlet-name>EBookshopServlet</servlet-name>
    <url-pattern>/eshop</url-pattern>
    </servlet-mapping>
</web-app>
```

The two crucial lines are those highlighted in bold. The first one tells Tomcat that the servlet is in classes\ebookshop\ShoppingServlet.class. The second one tells Tomcat that the requests will refer to the servlet as /eshop. As the root folder of this application (i.e., the folder immediately inside webapps) is ebookshop, Tomcat will then route to this servlet all the requests it will receive for the URL http://servername:8080/ebookshop/eshop.
The element <servlet-name> in both <servlet> and <servlet-mapping> is only needed to make the connection between the two. If you now open a browser and type http://localhost:8080/ebookshop/, you should see the application's home page.

You might be wondering about the purpose of the META-INF folder. Place inside that folder an empty file named MANIFEST.MF, zip the content of the whole application folder (webapps\ebookshop\), taking care to include entries for the relative folder paths, and rename the file ebookshop.war. By doing so, you've created a Web ARchive (WAR). To deploy your application on a different server, just place the WAR file inside the webapps folder. Tomcat unpacks it for you automatically *as soon as it realizes* that it's there. What could be easier than that?

Summary

In this chapter, we told you what happens on the server when you click a link in your browser to view a new page. We then introduced servlet and JSP technologies and explained what role they play in a web server.

We showed you a simple HTML page and how you can begin to add dynamic content to it with JSP. Next, we described how JSP-based applications are structured. Using the Ebookshop sample application, we showed a practical example of the MVC architecture. Finally, we introduced you to Tomcat and described how to install the Ebookshop application.

We covered a lot of ground in this chapter, showing you all the parts of a small working web application. Don't worry if you're finding it a bit difficult to absorb it all. Everything will become clear as we proceed through the next chapters in a more systematic fashion.