Chapter 7

Events and the Web

7.1 Introduction

In this chapter we discuss event based programming and the World Wide Web. In recent years the web has become ubiquitous. There are millions of millions web pages available to computer users around the world. Even new verbs, e.g. "to google", have entered the lexicon, as a result of the web. But how does it relate to event based programming? There are several ways:

- **GUI Structures on Web pages**
  Scripting languages like `Javascript` make is easy to include GUI components such as buttons and textboxes on web pages. In recent years high end IDEs like NetBeans and Visual Studio make it possible to create these web pages in a drag and drop fashion, without even needing to know scripting.
- **Web Applications**
  A *web application* is a computer program whose primary interface is made up of web pages. The program is running on a server, and the user interacts with it through one or more web pages. You don't have to look far to find web applications. Most major email systems come with a web client. Similarly, as college professors, the authors regularly look up student records on-line. If you visit `http://yahoo.games.com`, you will find dozens of games with web interfaces. Universally, all of these systems are event based in their interactions with the server. They follow the *Request-Response* interaction model discussed in Chapter 1.
- **Web Services**
  Web applications are great. We can take our model a step further, however. Why limit ourselves to just interacting with them via web pages? A *Web Service* is a service oriented program running on a web server, but the client may be a web browser, a standalone program, or another web server. Web services let companies produce and sell services via the web. The classic example of a web service is *PayPal*. *PayPal* securely manages the financial portion of sales, so users no longer have to enter their credit card numbers on-line to make purchases. Both the consumer and the seller interact with PayPal to complete the transaction.

The World Wide Web is obviously a loosely coupled distributed system and both web applications and web services are distributed in nature. To a certain extent, then, this chapter is just another chapter about distributed programming, just using a different distributed infrastructure. There are good reasons to use the web infrastructure to develop distributed systems. The primary reason is that web is fairly mature and the protocols and languages used to communicate between browsers and servers are very well understood. This makes it easier for developers to create and debug their applications.

7.1.1 Historical Perspective

Although the basic concepts of hypertext and networking extend back farther, the World Wide Web originated in 1990 when two scientists working at CERN in Switzerland, Sir Tim Berners-Lee and Robert Cailliau, proposed building a network storing hypertext pages that could be viewed by browsers. The Web really took off when the Mosaic Web browser was developed in 1993 by Marc Andreesen and others at the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign.

Early web pages were *static*. A web designer created an `html` document and deployed it on a web server. Users browsed to the page and read the document's contents. The page didn't change unless the web designer deployed an updated version. It soon became clear that the static model was inadequate because content on the web changed. By 1995 Netscape had introduced *server side pushing* of new content to browsers[1]. This allowed the server to update browser pages at its discretion. JavaScript was introduced in 1996. JavaScript is embedded in web pages and allows web pages to modify themselves by executing code as needed.

Today, the Web is very dynamic. Web servers respond to users not only with static html pages, but also with content created on the fly, primarily in response to information sent with requests from the user. *Common Gateway Interface (CGI)* scripts, *Servlets*, *Java Server Pages (JSPs)*, *PHP*, *Ruby on Rails*, and a variety of other technologies have emerged that let programs execute on web servers and produce dynamic content.

7.1.2 Multi-tiered Architectures

Many real world web applications use a multi-tiered architecture, where web pages serve as the user interface, the web server contains application code, a.k.a. business logic, and there is a database back-end, which stores persistent information for later reference.

In this model the browser serves as a *thin client*, with limited responsibilities. It provides a user interface and validates data before sending it to the server. Local data validation is important, because the web server may be remote and there can be noticeable time lag when interacting with it. Local validation saves time and server-side CPU cycles.

The server executes the application code. It runs programs that process orders, update inventory, or generate student record reports.

The database back-end serves as persistent storage. Orders for good, inventory, or student records all have very long lifetimes. The database stores this information for future reference.

This chapter does not contain multi-tiered architecture examples. While it is an interesting topic, they are not at the heart of understanding events and the web.

7.2 Web Fundamentals

Before we can intelligently develop web applications, we need to understand the fundamentals of how the web works.

[1]Note the use of event based terminology.

7.2.1 HyperText Transfer Protocol (HTTP)

For two computing systems to communicate they need an agreed upon set of standards for their messages. For example, in the previous chapter we briefly discussed IIOP, the Internet Inter-Orb Protocol, which CORBA uses. In fact, the Internet is based on a layered set of protocols, from the physical layer, made up of integrated circuits and wires, at the low-end, to the application layer at the high-end; all of which must work together to facilitate intercomputer communication.

The web uses *HyperText Transfer Protocol (HTTP).* It is an application level protocol. It was designed to facilitate communication of *hypertext* documents [2]. Its most common use is to transfer web pages between a server and web browser. HTTP is based on request-response interactions, where a web browser requests a page and a web server responds with the page. For example, HTTP defines the standard for *Uniform Resource Locators (URLs)* which are entered after the `http:` in a web browser.

HTTP also defines the services that may be requested from the server. There are eight. Three of are interest to us: *Head* and *Get*.

- **Get**
  `Get` requests a web page from the server. Ideally, `Get` should not change the state of the web server. That is, multiple calls to `get` the same web page should always return an identical page.

- **Head**
  `Head` functions much like `get`, but does not request the entire page. A web page has a number of parts, including a heading and a body. The heading contains summary information about the page, but not the page content. `Head` only requests the heading information for the page. `Head` is typically used to test a link before an entire document is requested.

- **Post**
  `Post` submits data to the web server. Functionally, *Post* is similar to *Get*, in that a request is sent from a browser and a response is returned by the server. Because *Post* is meant for data submission, it is acceptable to have identical *Post* requests vary their response based on a number of factors, such as whether the information being sent was submitted previously.

7.2.2 HyperText Markup Language (HTML)

HyperText Markup Language (HTML) is the language used to describe web pages[3]. Markup languages surround data or information with tags in angular brackets < and > that describe either how the information should be presented or the information's content. Different markup languages include different sets of tags. HTML is strictly a presentation language. The tags state how things should be displayed. Below is a short HTML document that displays in Internet Explorer as shown in Figure 7.1.

[2] *Hypertext* documents contains links to other documents, that the reader can access, typically by clicking on the link with the mouse.

[3] Use of HTML is not limited to web pages. For example, it is also frequently used to format messages in email systems.

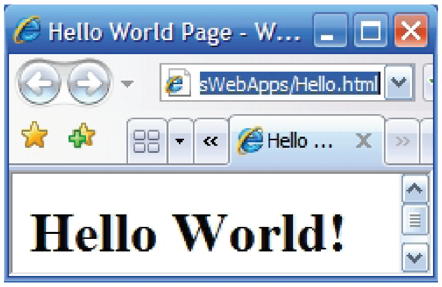


Figure 7.1: A simple web page

```
 1 <!--
 2 Hello.html
 3 A simple web page to illustrate html tags.
 4 Written by: Stuart Hansen
 5 Date: March 23, 2009
 6 -->
 7 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
 8 <html>
 9   <head>
10     <title>Hello World Page</title>
11     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
12   </head>
13   <body>
14     <h1>Hello World!</h1>
15   </body>
16 </html>
```

Lines	**Commentary**
1–6	Comments within HTML documents appear between `<!--` and `-->`.
7	This is the document type declaration. Including this declaration allows the document to be validated, guaranteeing that it is well-formed. The two primary families of document types currently in use for web pages are `HTML 4.01 XXX` and `XHTML 1.X XXX`. For our purposes the differences among the various versions may be ignored.
8–16	The part of the HTML document that will be displayed appears between the `<html>` and `</html>` tags. There are two required elements within the html document, <head> </head> and <body> </body>
9–12	The head element contains the title, meta-information, and the locations of scripts.
11	Meta information is information about the page. In this example, it tells us that the document is a text document encoded in UTF-8.
13–15	The body of the document contains the information that should be displayed in the web page. In this example, it contains one line, ”Hello World!” that will be displayed as a top-level heading, <h1>.

There are many HTML tags and for our purposes it is foolish to try to memorize all of them. Instead, there are any of a number of software tools that will help produce nice looking web pages. As programmers[4] the authors recommend using NetBeans. NetBeans is bundled with a web server for displaying pages (see the next section), and drag and drop tools for inserting tags into pages.

7.2.3 Web and Application Servers

You probably already have a good notion of what a web server is. It is a program running on a computer that returns a web page when a request is made for it. Older web servers were designed just to return static web pages and not much more. As it became evident that dynamic web content was becoming increasingly important, web servers were extended to include the ability to run other programs to create the content. For example, an ability to execute Perl scripts was a standard add-on to early servers.

In recent years, as web applications have become more and more important, the industry has shifted from web servers to *application servers*. An *application server* is designed to expose the public interface of the application, efficiently creating dynamic content, while still being willing to respond with static pages when required. There are numerous application servers available from different vendors, including: Tomcat, GlassFish, JBoss, WebLogic, WebObjects and WebSphere.

Like static web pages, web applications need to be deployed to the server where they will run. Often this amounts to simply creating a subdirectory and copying files into it. Unfortunately, the details differ significantly from application server to application server. Fortunately, many servers now come with deployment tools to aid the developer.

7.3 Java Servlets

As in previous chapters, we will illustrate these concepts using Java.

Servlets are Java programs that run on an application server. The *servlet container* is the portion of the application server that executes the Java byte code. The servlet container is responsible for managing the servlets. For example, it maps URLs to servlets. It starts the Java virtual machine when a request is made for a servlet. It checks the access permissions of the requester to ascertain whether they have permission to run the servlet.

There are two files needed to work with servlets. On the browser side, we need a web page that requests that a servlet be run.

On the server side, we need the servlet. Servlets are pure Java code. They require a specialized Java library: `javax.servlet.*`. Our servlet class will extend `HTTPServlet`, overriding methods to accomplish our tasks.

7.3.1 Calculator Example

The ideas behind web applications and their dynamic content become much clearer when we look at an example. The examples in this chapter were developed using NetBeans 6.5. This IDE comes with graphical tools for building web pages and several application servers that can be used for development. It also comes packaged with the libraries (`javax.servlet.*`) that are needed to develop servlets.

Below is an abbreviated web page that contains a calculator. The web page is shown in Figure 7.2. The result of carrying out a calculator operation is shown in Figure 7.3.

[4]This is a programming text, after all.

```
10 <html>
11     <head>
12         <title>Calculator</title>
13         <meta http−equiv=”Content−Type” content=”text/html; charset=UTF−8”>
14     </head>
15     <body>
16         <form name=”Calculate” action=”Calculate”>
17             <input type=”text” name=”FirstNumber” value=”0” />
18             <input type=”text” name=”SecondNumber” value=”0” />
19             <select name=”Operation”>
20                 <option>+</option>
21                 <option>−</option>
22                 <option>∗</option>
23                 <option>/</option>
24             </select>
25             <input type=”submit” value=”OK” name=”SubmitButton” />
26         </form>
27     </body>
28 </html>
```

Lines	**Commentary**
1–9	The first 9 lines of the file contain heading comments and the data type declaration and are omitted to save space.
16–26	The body of the page contains a form, which is where the user enters information for processing. The `action=''Calculate''` attribute says that the `Calculate` servlet should be called when the form is submitted.
17–18	There are two text fields on the form, where the user enters the two numbers to be processed.
19–24	The form contains a drop down list, from where the user selects the operation to be performed.
25	The `Submit` button is clicked to send the request to the server.

There are more elements that can be included in a form, including radio buttons, multi-line input fields, and checkboxes. A complete discussion of html forms is beyond the scope of this text.

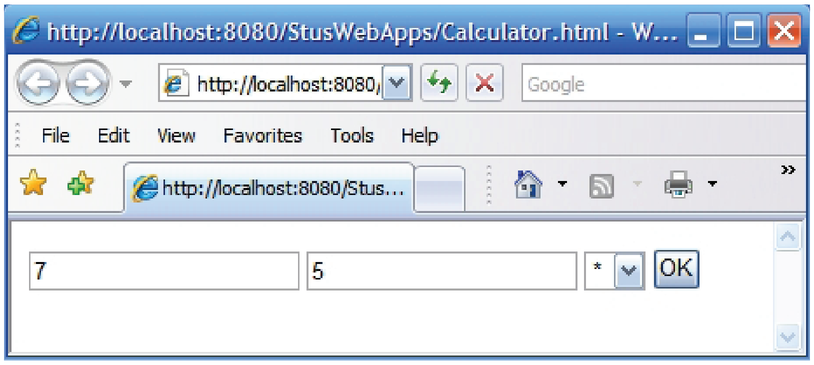
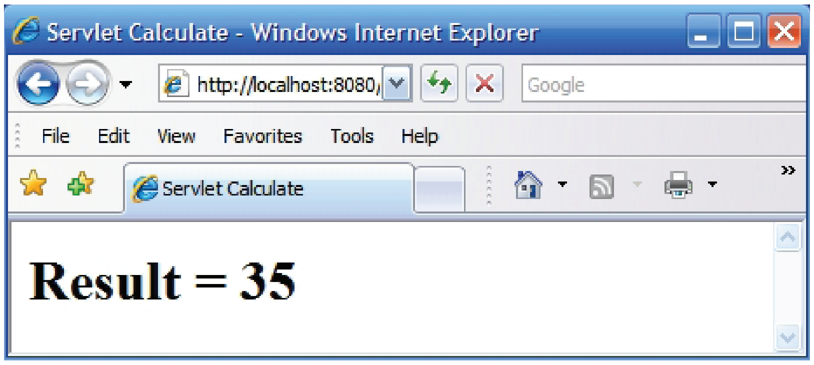

http://localhost:8080/StusWebApps/Calculator.html - W...
http://localhost:8080/
Google
File Edit View Favorites Tools Help
http://localhost:8080/Stus...
7
5
*
OK

Figure 7.2: The interface for the calculator.


Servlet Calculate - Windows Internet Explorer
http://localhost:8080/
Google
File Edit View Favorites Tools Help
Servlet Calculate

Result = 35

Figure 7.3: The result of carrying out a calculator operation.

The Calculate Servlet

```java
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Enumeration;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * Calculate.java shows how to process form data in a servlet.
 * This file responds to requests generated by Calculator.html
 *
 * @author Stu Hansen
 * @version March 29, 2009
 */
public class Calculate extends HttpServlet {
    /**
     * Processes requests for both GET and POST methods.
     * @param request servlet request
     * @param response servlet response
     * @throws ServletException if a servlet-specific error occurs
     * @throws IOException if an I/O error occurs
     */
    protected void processRequest(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");

        // Get the three parameters passed in from the web page
        int i = Integer.parseInt(request.getParameter("FirstNumber"));
        int j = Integer.parseInt(request.getParameter("SecondNumber"));

        // Carry out the requested operation
        String op = request.getParameter("Operation");
        int result = 0;
        if (op.equals("+"))
            result = i+j;
        else if (op.equals("-"))
            result = i-j;
        else if (op.equals("*"))
            result = i*j;
        else if (op.equals("/"))
            result = i/j;
43          // Respond back to the browser with the result
44          PrintWriter out = response.getWriter();
45          try {
46              out.println("<html>");
47              out.println("<head>");
48              out.println("    <title>Servlet Calculate</title>");
49              out.println("</head>");
50              out.println("<body>");
51              out.println("    <h1>Result = " + result + "</h1>");
52              out.println("</body>");
53              out.println("</html>");
54          } finally {
55              out.close();
56          }
57      }
```

Lines	**Commentary**
1–7	The servlet packages do not come with Sun's JDK, but may be downloaded separately. They do come bundled with NetBeans.
16	Servlet classes extend HttpServlet. This is an abstract class and programmer must override at least one method, generally `doGet()` and/or `doPost()`..
24 & 25	Both `doGet()` and `doPost()` call `processRequest()`. The `request` parameter contains all the data included with the request arriving from the web browser. The `response` parameter is used to dynamically generate a page and send it to the browser.
29, 30 & 33	We make three calls to `request.getParameter()` to obtain the two numbers and the operation we are to carry out. Note that all parameters are `String`s, so the numbers need to be massaged into their correct integer representations.
35–42	We use an `if..else if` structure to determine the operation and carry it out.
45–57	We dynamically generate a web page to be returned to the browser.

The remainder of the servlet is autogenerated by NetBeans. Both `doGet()` and `doPost()` call `processRequest()`, which is defined above.

```
    /**
     * Handles the HTTP <code>GET</code> method.
     * @param request servlet request
     * @param response servlet response
     * @throws ServletException if a servlet-specific error occurs
     * @throws IOException if an I/O error occurs
     */
    @Override
    protected void doGet(HttpServletRequest request,
                                    HttpServletResponse response)
    throws ServletException, IOException {
        processRequest(request, response);
    }

    /**
     * Handles the HTTP <code>POST</code> method.
     * @param request servlet request
     * @param response servlet response
     * @throws ServletException if a servlet-specific error occurs
     * @throws IOException if an I/O error occurs
     */
    @Override
    protected void doPost(HttpServletRequest request,
                                    HttpServletResponse response)
    throws ServletException, IOException {
        processRequest(request, response);
    }

    /**
     * Returns a short description of the servlet.
     * @return a String containing servlet description
     */
    @Override
    public String getServletInfo() {
        return "Short description";
    }
}
```

Deploying the Calculator Servlet

Deploying a web application is primarily the process of copying the application files to the appropriate subdirectory within the server's servlet container and then registering the servlet with the container. While this seems straightforward, the details get confusing[5]. NetBeans and other IDEs will autogenerate a `web.xml` script that `ant` will use to do the deployment. This is, by far, your best option for deploying web applications.

Running the Calculator Servlet

A servlet is invoked by having the browser send a request for the servlet to the server. In our case, the request will include parameters. Clicking the "Submit" button will generate the request, but

[5]Each servlet has a servlet name, a class name, and a URL pattern. Giving the same servket three different names strikes the authors as particularly confusing. We recommend that whenever possible use the same name for all three.

the user can also type the request into the browser's navigation bar. For the calculator, running on the local machine, on port 8080, the user would enter:

```
http://localhost:8080/StusWebApps/Calculate?FirstNumber=5&SecondNumber=7&Operation=*&SubmitButton=OK
```

7.4 Adding State to Web Applications

Recall that an important feature of event based systems is that they are state based. Unfortunately, HTTP was designed as a stateless protocol. That is, web servers are not required to retain any information about a client between the client's requests. This worked fine when the web consisted of static web pages, but has created havoc in a universe where we expect web servers to maintain shopping carts or other user specific data. For example, each time a user adds something to a shopping cart, the server should maintain that information, at least for a while. This is a change in the data state of the server.

It is also frequently necessary to maintain control state. Consider, for example, a two person game where the people alternate turns. A web interface for this game requires that the one player be able make a move and that the other player be blocked from moving. That is, there is control state needed.

The problem of maintaining state in web applications has been solved in a number of ways, including *cookies*, *sessions*, *hidden variables* and *databases*.

7.4.1 Cookies

A *cookie* is a small file sent by the server and stored on the client. The contents of the file will be returned to the server with subsequent requests. This method avoids problems with session timeouts, but creates several new issues. A server may store private information, e.g. credit card numbers, in a cookie, and others using the computer may see its contents. Cookies may track all the pages within a site that the user visits, building a profile of the user. This gives some people serious ethical concerns. Most browsers now allow the user to turn off cookies, preventing them from being stored.

7.4.2 Sessions

To understand *sessions*, we first need to understand a bit about a servlet's lifecycle. There are two practical considerations that guide this discussion. First, web and application servers handle many requests with limited resources. Thus, efficient management of resources is still very important on web servers. Second, when a user browses a website, they often visit the same or related pages repeatedly.

With limited memory resources, it makes sense for the servlet container to wait for a request for a servlet before loading it into memory. Once in memory, it makes sense to keep the servlet loaded for some time, as the user may well return to the servlet again. The period of time the user interacts with the servlet is known as a *session*. The servlet may remain in memory longer than that, as there may be multiple users interacting with the same servlet.

In modern web applications, there are many types of information that developers may want to keep for the duration of a session. For example, a shopping cart contains items the user is interested in purchasing, but hasn't yet paid for. We don't want these to be forgotten. Similarly, many websites now require logins. A session should keep track of whether the user has already logged in.

In most cases, it also makes sense to have sessions time out. That is, if a user makes no more requests of a servlet during a given time interval, the session should be dropped. The default timeout

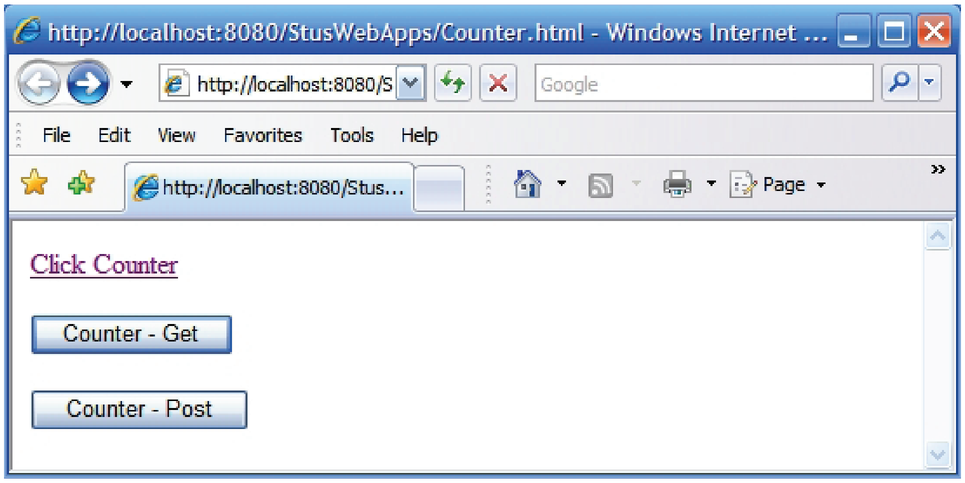
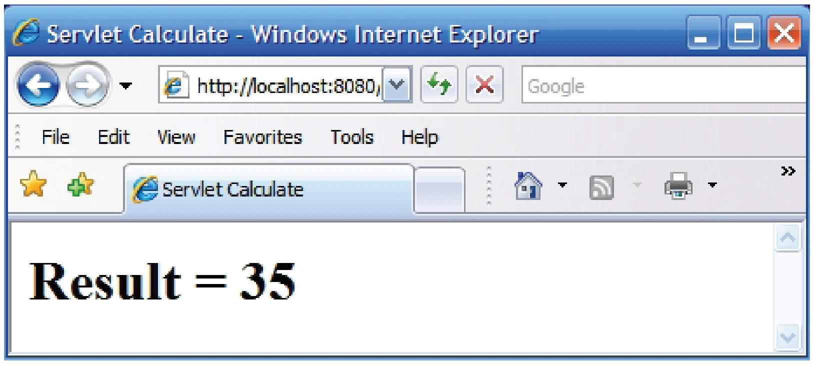

http://localhost:8080/StusWebApps/Counter.html - Windows Internet ...
http://localhost:8080/S
Google
File Edit View Favorites Tools Help
http://localhost:8080/Stus...
Page
Click Counter
Counter - Get
Counter - Post

Figure 7.4: The interface for the counter application.


Servlet Calculate - Windows Internet Explorer
http://localhost:8080/
Google
File Edit View Favorites Tools Help
Servlet Calculate

Result = 35

Figure 7.5: The page that results after the servlet has been visited 35 times.

period is typically 30 minutes. In Java, the `HTTPSession` class contains `setMaxInactiveInterval()` which allows the programmer to change that time.

Java's servlet library contains an `HttpSession` class. An `HttpSession` object is automatically created for a user's interactions with the servlet. The session object operates much like a hashtable, where the key is a string and the value is an arbitrary Java object. When we want to store state using a session, we obtain a reference to the session object and then get and set attributes within it.

The Counter Example

The following listing is for `Counter.html`, a web page with three ways to contact the same servlet: a link, a button that performs a get request, and a button that performs a post request.

```
 8 <html>
 9   <head>
10     <title></title>
11     <meta http−equiv=”Content−Type” content=”text/html; charset=UTF−8”>
12   </head>
13   <body>
14     <a href=”Counter”>Click Counter</a>
15     <form name=”CounterGetForm” action=”Counter”>
16         <input type=”submit” value=”Counter − Get” name=”GetCounter” />
17     </form>
18     <form name=”CounterPostForm” action=”Counter” method=”POST”>
19         <input type=”submit” value=”Counter − Post” name=”PostCounter” />
20     </form>
21   </body>
22 </html>
```

Lines	**Commentary**
1–7	The first seven lines contain heading comments and data type declarations. They are omitted for brevity.
14	This link requests the servlet.
15	This button requests the servlet using ”get”.
18	This button requests the servlet using ”post”.

The excerpted listing below uses a session object to count how many times the servlet has been visited during this session. The entire code listing for this servlet is available on the text’s website as `Counter.java`.

```java
    protected void processRequest(HttpServletRequest request,
            HttpServletResponse response) throws ServletException, IOException {
        PrintWriter out = response.getWriter();

        HttpSession session = request.getSession(true);

        int count = 0;

        // Create the new session counter
        if (!session.isNew()) {
            count = ((Integer) session.getAttribute("count")).intValue();
        }

        // Increment the counter for this visit
        count++;
        session.setAttribute("count", count);

        response.setContentType("text/html;charset=UTF-8");

        try {
            out.println("<html>");
            out.println("<head>");
            out.println("    <title>Servlet Counter</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("    <h1>Counter value for this session is: "
                                                            + count + "</h1>");
            out.println("</body>");
            out.println("</html>");
        } finally {
            out.close();
        }
    }
```

Lines	Commentary
1–20	These lines contain imports, heading comments and the class declaration. They are virtually identical to the code in the previous example and are omitted for brevity. We only show the `processRequest()` method, which contains all the code of interest.
25	We obtain a reference to the `session` for this servlet. The `true` parameter directs the call to create a new session object i f one has not already been created. You may think of a session as a hashtable. It stores any object that the programmer wants to keep around between visits to the servlet.
30–32	In our case, we want to store the value of a counter that tells us how many times this client has visited this servlet during this session. Note that a client request arriving from another IP address will have its own session and counter.
36	Sessions use `String`s and keys and hold `Object`s. We use autoboxing to cast our int into an Integer. This relates back to line 31, where we we had to explicitly cast the `Object` to be an `Integer`, and then used auto unboxing to convert it to an `int`.
40–53	Respond with a dynamically created web page.
46	Note the use of `count`.

The remainder of the servlet code is autogenerated and is completely identical to that in the previous example, so is omitted.

7.4.3 Other Ways to Maintain State

Hidden Parameters

We have already seen how data can be collected from forms and sent to the server. *Hidden parameters* behave the same way as interface parameters, but are not visible on the web page. Web pages and servers can exchange information via hidden parameters.

Databases

Many web applications use database back-ends to store persistent information. For example, when you place an order for a cashmere sweater on the web, a database record will be created storing this information for further processing. Databases are very important in many web applications, but their role is to maintain information with very long lifespans (*persistent information*), much longer than session based information. The record of your sweater order will be around while the order is filled, shipped, arrived, and is possibly returned. This is much longer than a session.

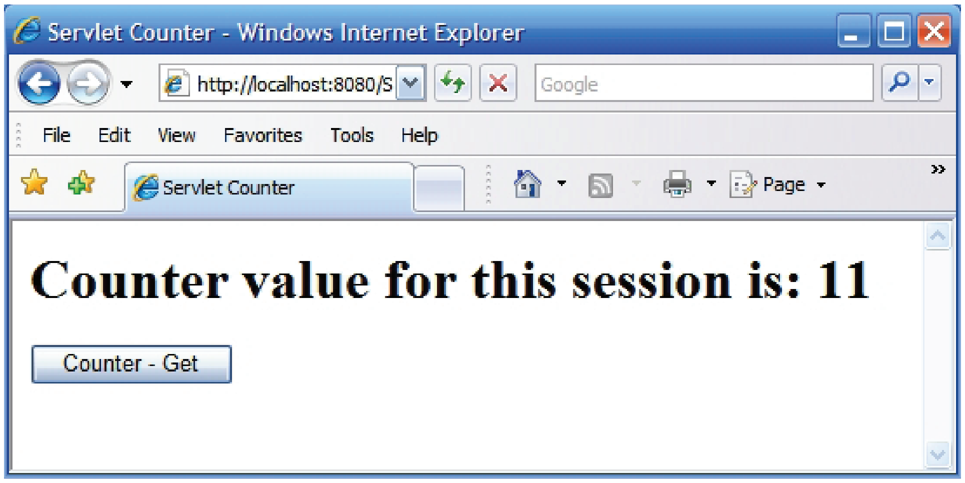
7.4.4 Combining Request and Response Pages

The counter example in the previous section contains a rather large maintenance problem. The servlet is paired with a web page that requests it. A change to either will require a change to the other. Since the web page and servlet may be stored in different directories or on different machines, maintenance becomes an issue. If you consider doing a larger website, with numerous requesting pages and servlets, you can see how nightmarish the situation could become.

One solution to this problem is to make all the web content dynamic. Our requesting page was originally static. It was HTML code that was stored in a file. We requested that page, then used that page to invoke the servlet which generated the counter response page. Instead, we can have a servlet generate both the requesting page and the response page. In our example, we can have the page be self contained. That is, we will put the counter and the controls to increment the counter on the same page. There is no reason why a servlet couldn't generate multiple very different pages, however, based on session or parameter values.

```
    protected void processRequest(HttpServletRequest request,
        HttpServletResponse response)  throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            // Get the session variable
            HttpSession session = request.getSession(true);

            int count2 = 0;  // an integer counter

            // Update the session's attribute, if it exists
            if (session.getAttribute("count2") != null) {
                count2 = ((Integer) session.getAttribute("count2")).intValue();
            }

            // Increment the counter for this visit
            count2++;

            // Set the session's attribute
            session.setAttribute("count2", count2);

            // Return a new web page to the browser
            response.setContentType("text/html;charset=UTF-8");
            out.println("<html>");
            out.println("<head>");
            out.println("   <title>Servlet Counter</title>");
            out.println("</head>");
            out.println("<body>");

            out.println("<h1>Counter value for this session is: "
                                                    + count2 + "</h1>");
            out.println("<form name=\"CounterGetForm\" action=\"Counter2\">");
            out.println("   <input type=\"submit\" value=\"Counter - Get\"
                                                 name=\"GetCounter\" />");
            out.println("</form>");

            out.println("</body>");
            out.println("</html>");
        } finally {
            out.close();
        }
    }
```

The remainder of the servlet code is autogenerated and is completely identical to that of the previous example. It is omitted for brevity.


Servlet Counter - Windows Internet Explorer
http://localhost:8080/S
Google
File Edit View Favorites Tools Help
Servlet Counter
Page

Counter value for this session is: 11

Figure 7.6: The interface for the self-contained counter servlet.

Lines	Commentary
1–24	These lines contain imports, heading comments and the class declaration. They are virtually identical to the code in the previous example and are omitted for brevity. We only show the `processRequest()` method, which contains all the code of interest. The entire code listing is available on the text's website.
27–44	These lines are virtually identical to those in the previous example. We get the `session` variable, see if there is a `counter2` attribute, and if not, initialize it.
46–61	Our `out.println`s dynamically create a web page.
54	Note the use of `counter2` on this line.
56–59	We include a form on the response. By doing so, we alleviate the need for a separate web page for input. Instead, the requesting page is just a different version of the response page.

7.5 Java Server Pages (JSPs)

Java Server Pages (JSPs) are an extension to servlets that make combining dynamic and static content a bit more natural. A JSP is identified by having the name suffix `.jsp`. A JSP looks much like a static web page, but has Java code embedded within it, between <% and %>. A servlet is autogenerated on the application server that wraps the JSP's content into something very much like the previous example.

The web page for this example is identical to the web page in the previous section, so its figure is omitted for brevity.

```
<%--
    Document   : JSPCounter
    This file contains a jsp page that keeps track of a counter
    Created on : Mar 29, 2009, 10:23:20 AM
    Author     : Stu Hansen
--%>

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Counter</title>
    </head>
    <body>
        <%
        // We include the code that is used to keep track of our counter
        // Note that jsp understand a number of default variables like:
        // request and session

        int jspCount = 0;  // Our counter

        // Check if this is the first time during this session that we
        // have been to the jsp.  If not, get the counter.
        if (session.getAttribute("jspCount") != null) {
            jspCount = ((Integer) session.getAttribute("jspCount")).intValue();
        }

        // Update the counter and store it back into the session
        jspCount++;
        session.setAttribute("jspCount", jspCount);

        // Respond to the browser with the counter
        out.println("<h1>Counter value for this session is: "
                                                  + jspCount +  "</h1>");
        %>

        <!-- In place of the out.println above, we could use the following -->
        <!--<h1> Counter value for this session is :
                  <% out.print(session.getAttribute("jspCount")); %> </h1> -->

        <form name="CounterGetForm" action="JSPCounter.jsp">
            <input type="submit" value="Counter - Get" name="JSPCounter" />
        </form>

    </body>
</html>
```

Lines	Commentary
1–10	These are heading comments and document type declarations. Note the difference in comment tags for JSPs rather than static web pages.
12–17	These are standard html tags.
17 & 37	The <% and %> tags surround a block of Java code that is embedded within the JSP.
18–36	This is standard Java code that looks very much like the code in `processRequest()` in the previous example. It is executed every time the JSP is loaded.
27, 33 & 36	JSPs automatically give the programmer access to certain standard variables, including `session` and `out`. There is no need to declare or instantiate them.
43–45	This again is standard html code that places the button on the page.
48–49	Terminate the open tags.

There is much more to JSPs than discussed here. There are custom tag libraries that make handling content easier. Students interested in delving further into JSPs are encouraged to pick up any of a number of excellent texts on the topic.

7.6 Web Services

The web application model presented in the previous chapter has been very successful, with many organizations deploying web applications as part of their web sites. How event based are they, however?

In some ways, web applications are quite event based:

- The web based front-end usually contains a form with text boxes and buttons, and code is associated with these components.
- The web application literature often refers to the web front-end as a view, the servlet as a controller, and the database as the model, again mirroring the terminology of the event based literature.

In other ways, web applications aren't all that event based:

- The web based front-end, the servlets running on the server, and the database back-end, although distributed, are all tightly coupled into one application. The association of the web pages, the servlets and the database is done statically, when the system is created, not dynamically, as we typically associate with events.
- The exchanges between the browser and server follow a request/response interaction, much like a method call. The browser requests a page from the server and sends along some parameters. The server returns the page. The interaction is asynchronous in the sense that the browser does not know how long it will take to receive a response from the server, but it is blocking in the sense that the browser waits for the response[6].

[6]Most browsers have a separate user thread that is not blocked, however, allowing the user to open another page or carry out other activities.

Web services utilize the same web infrastructure as web applications, but are built on a different model. A *web service* offers services to client programs. The *client* may be a web browser, a different web server, or a standalone program. The client requests a service from the hosting application server. The service response may be a web page, a different document, or nothing at all.

There are numerous examples of web services. Here are a few popular ones:

- *PayPal*, `https://developer.paypal.com/`, securely manages the financial portion of sales, so users no longer have to enter their credit card numbers to make on-line purchases. Both the consumer and the seller interact with PayPal to complete the transaction.

- *Google documents*, `http://docs.google.com`, lets users create, edit and store word processing documents, spreadsheets and presentations.

- *Blogger.com*, `http://www.blogger.com/developers/api/1_docs/`, lets programmers add blogging capabilities to their applications.

- *Flickr photo sharing*, `http://www.flickr.com/services/api/`, lets programmers add photo sharing capabilities to their applications.

Some authors argue that web services are the future of desktop computing. Applications and documents will be stored on servers. It won't matter where you are, or what computer you work at. As long as you have any computer and the Internet available, you will be able to access all your work.

Like many things in computing, web services continue to evolve at a rapid rate. The next few sections discuss the basic model used by web services. However, just about all aspects of this model are currently in a state of flux, with new ideas and technologies being introduced regularly. We will briefly address these issues at the end of the chapter.

7.6.1 Stock Quote Example

A good way to understand how web services work is to walk through an example.

Pretty Good Investment Advisers wants to provide a way for clients to look up stock quotes on-line. They develop a web service where the user enters a stock symbol. In real time, the service looks up the price of the stock on their corporate office's mainframe computer, and returns the result to the user. They advertise their new service at: `http://www.xmethods.net`. One of their best clients is a CS professor at a local college. She downloads the documents describing the stock quote service, then writes a client program to work with the service to look up stock prices. She runs her program and enters `IBM` as the stock symbol, The web service looks up IBM and returns the stock quote in a document very similar the one shown below. Her program parses this document and displays the result.

7.6.2 The eXtensible Markup Language (XML)

Web services rely on *eXtensible Markup Language (XML)* for several purposes. It is used to describe the services. It is used to request services, and it is used for the responses. Like HTML, XML uses text based tags to organize the content of documents. For example, below is a brief XML document describing a stock quote.

```
 1   <Stock>
 2      <Symbol>            IBM     </Symbol>
 3      <Last>            82.71     </Last>
 4      <Date>       12/15/2008     </Date>
 5      <Time>           1:00pm     </Time>
 6      <Change>          +0.51     </Change>
 7      <Open>            82.51     </Open>
 8      <High>            82.86     </High>
 9      <Low>             80.00     </Low>
10      <Volume>        3007015     </Volume>
11   </Stock>
```

Lines	**Commentary**
1 & 11	Each stock quote is between <Stock> and </Stock>. In our example we only have one quote, but it would be easy to extend to a list of stocks.
2–10	The stock quote consists of a variety of attributes, each within its own pair of tags. For example, the low selling price of IBM on December 15, 2008 was 80.00.

XML Languages, Schemas and SOAP

We need a standard way to represent stock quotations, otherwise the client programs will have a terrible time working with the service. The pattern for the standard can be seen in the IBM quote above. Each quote is surrounded by <Stock> and </Stock> and has a list of properties between. *An XML language* is a well defined set of XML tags that provides information on a particular topic, e.g. stock quotes, that follows a particular scheme. When we say we want a standard way to represent stock quotes, what we are saying in XML terms is that we want to define an XML Language.

An *XML Schema* describes an XML language. Thus, there is an XML StockQuote schema that describes all the tags that can occur in any stock quote document. Anyone implementing a stock quote service, or using a stock quote service can read the schema and what tags go into a stock quote, allowing them to create and parse stock quote documents[7].

The *Simple Object Access Protocol (SOAP)* is an XML language designed for communicating the structure and values of an object between systems. The request and response messages used by web services use SOAP to represent their content.

7.6.3 Finding a Web Service and its API

To be useful, a web service needs to advertise itself. This includes its locations (its URL) and its public interface (API). It does so by registering a *Web Services Description Language (WSDL)* document in a *Universal Description Discovery and Integration (UDDI)* registry. The basic model is shown in Figure 7.7. Note that the UDDI registry contains the URL for the WSDL document, not the actual document.

The *WSDL* document describes a service, including:

- the URL for the service,

[7]An XML schema is actually another XML document. Thinking a little recursively, there is also an XML schema that describes schemas.

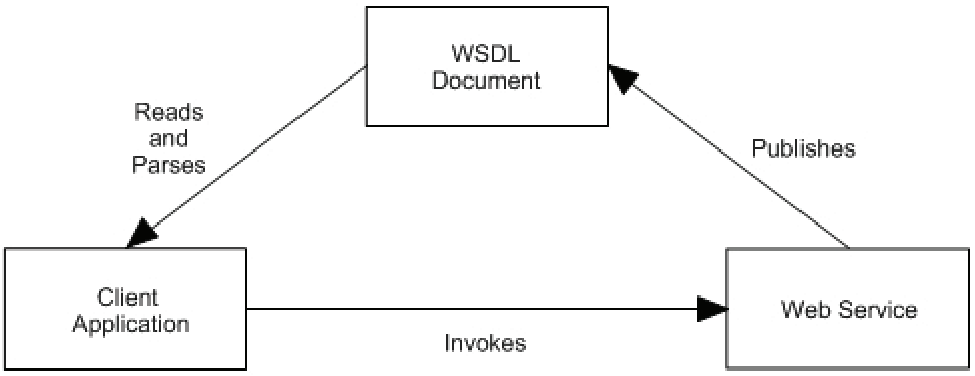


Figure 7.7: The basic pieces needed for a web service to be used.

- the public methods the service provides,
- arguments and types for the methods,
- the type of value returned from each method.

The UDDI registry entry contains:

- a reference (URL) to the WSDL document,
- information about the publisher of the service,
- a categorization of the service, and
- technical information about the service.

WSDL and UDDI are XML languages. Together the WSDL and UDDI documents for a service provide a web service client the information it needs to successfully access the service.

One popular UDDI registry is `http://www.xmethods.net`. When you browse this site, you will find thousands of web services. If you burrow down through a service's links, you will find complete descriptions of the service, including the WSDL document text.

When a programmer develops a web service client, they start by locating the WSDL documents for any services they will use. Most modern IDEs, including NetBeans and Visual Studio .Net, provide built-in methods to parse the WSDL documents and autogenerate code to make calls to the services methods. The programmer only has to write the client code that uses the service.

7.6.4 Summary of XML Uses in Web Services

We have thrown a lot of jargon and acronyms into this section, so here is a summary:

Term	Description
XML	eXtensible Markup Language: a generalization of html. Tags surround the content of a document. They are used to add meaning and formatting information. XML is used in numerous ways in web services.
WSDL	Web Service Description Language: the XML language for describing a web service. The wsdl document is used by a programmer to develop a web service's client.
UDDI	Universal Description Discovery and Integration: an XML language used to advertise a web service. UDDI registries contain UDDI documents for many different web services.
SOAP	Simple Object Access Protocol: an XML language used to represent objects. It is used for requests and responses in web services.
Schema	An XML document that describes an XML language. Thus, there is a schema for WSDL, UDDI, and SOAP. SOAP is also specialized for requests and responses in a particular web service. Each of these also has a schema.

7.7 Developing a Web Service

It is possible to code a web service or a client from scratch. The only reason to do so, however, is to thoroughly learn the underlying XML languages and protocols. Almost all popular languages have IDEs associated with them that will autogenerate code to interact with clients. This abstracts away the low-level details, allowing the programmer to concentrate on developing the service instead.

In this section we will develop a Java based web service using the NetBeans IDE. As briefly discussed in the previous chapter, NetBeans is a free IDE available from `http://www.netbeans.org`. It is packaged with several application servers making developing and deploying web services much simpler.

7.7.1 Quadratic Equations Revisited

Our example is that of a quadratic equation solving service. Recall from high school that quadratic equations have the form

$$ax^2 + bx + c = 0$$

`a`, `b`, and `c` are known as the coefficients. By changing `a`, `b`, and `c` we get different equations.

A solution to a quadratic equation is a value of `x` that makes the equality true. For example,

$$1x^2 + -3x + 2 = 0$$

has two solutions: $x = 1$ and $x = 2$ A quadratic equation may have 0, 1 or 2 solutions, depending on its coefficients.

You probably also recall that this type of equation may be solved using the quadratic formula, which is

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Our service will contain a single method, named `solve()`, which takes three parameters (`a`, `b`, and `c`). It will use the quadratic formula to find and return the solutions to the equation.

7.7.2 Initial Steps

The initial steps to creating the web service in NetBeans are:

1. Create a new Project.
    (a) In the dialogs choose `Java Web` and `Web Application`. Note that while we are not creating a web application as described in the previous chapter, we are using the same infrastructure, e.g. application servers and servlet classes.
    (b) Name the project anything appropriate.
    (c) Choose any installed application server. The authors recommend choosing `Glass Fish V2`, as it allows the programmer to interactively test the web service.
    (d) Click `Finish`.
2. Right click on the project icon and choose `New Web Service`. Name it `QuadraticService`. Name its package `QuadService`. This will generate the class stub shown below.

```
 1 package QuadService;
 2 
 3 import javax.jws.WebService;
 4 
 5 /**
 6  *
 7  * @author Stu Hansen
 8  */
 9 @WebService()
10 public class QuadraticService {
11 
12 }
```

Lines	Commentary
3	Note the auto-generated `import` statement, that brings in the `WebService` interface.
9	The `@WebService()` annotation makes our class into a web service.

7.7.3 Web Service Annotations

Beginning with Java 1.5, the language has included annotations. Annotations are meta-data, describing some aspect of the code. An annotation begins with the `@` symbol and includes information and directives for tools processing the Java file.

Some annotations are used directly by the Java compiler. For example, `@Override` before a method name tells the compiler that this method is meant to override a method in a super class. If the signature of the method is incorrect, and it doesn't override a super class method, the compiler is required to generate an error. `@Override` is not required, nor does it affect the byte code generated. Instead, it is meant as a means of helping the programmer manage their code, guaranteeing that the signature of the method is correct.

Other annotations are used by different tools besides the compiler. For example, Sun's SDK includes `wsgen` and `wsimport`. These are command line tools to generate web service artifacts including classes and script files used to develop, deploy and invoke web services. Because we are working within NetBeans, we will not call these tools directly. NetBeans will do it for us. The three annotations we will use that are recognized by these tools are: `@WebService`, `@WebMethod`, and `@WebParam`.

7.7.4 Completing the Web Service

We complete our web service by writing and annotating the `solve()` method. NetBeans has a GUI tool to help programmers define web methods. This is available by clicking the `Design` tab at the top of the editor pane. The annotations in our example are simple enough that they can easily be entered manually, too.

```java
package QuadService;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;


/**
 * A web service with exactly one method.
 * The method solves quadratic equations.
 *
 * @author Stu Hansen
 * @version April 2009
 */
@WebService(name = "Quadratic", serviceName = "QuadraticService")
public class QuadraticService {
    @WebMethod(operationName = "solve")
    public double [] solve (@WebParam(name = "a") double a,
                            @WebParam(name = "b") double b,
                            @WebParam(name = "c") double c) {
        double [] results = new double [2];

        // Because Java's doubles use Infinity and Nan, we don't need to
        // worry about exceptions.
        double discriminant = b*b - 4*a*c;
        results[0] = (-b + Math.sqrt(discriminant))/(2*a);
        results[1] = (-b - Math.sqrt(discriminant))/(2*a);

        return results;
    }
}
```

Lines	Commentary
3–5	Our import statements need to be expanded to include all three types of annotations.
15	We expand the `@WebService()` annotation including `name` and `serviceName` parameters. These will be used later when developing the client in order to identify the service.
17	The `@WebMethod` annotation declares `solve()` to be a method that will be exposed in the web service's API. The `operationName` parameter just tells us that it will still be called `solve`.
18–30	The `solve()` method solves the quadratic equation using the quadratic formula. Because there may be up to two solutions, we return an array of doubles. Java's `double` uses `Infinity` and Not a Number (`Nan`) for divide by zero and taking the squareroot of a negative. Thus, no exception handling code is needed.
18–20	Each parameter is annotated with `@WebParam`. The name parameter for each tells us that each parameter will retain its own name (a, b, and c) in the web service.

7.7.5 Testing and Deploying the Web Service

As with any distributed computation, things become much more complex when we use multiple computers with a network joining them. As such, common sense tells us that we should test our web service locally, as much as possible, before deploying it. The `QuadraticService` class in our example, is just another Java class. The annotations do not affect our ability to test the code locally. We can (and should) write `JUnit` tests or a test `main()` to exercise the code removing any bugs discovered.

Deploying the web service is a matter of installing our web service class, along with autogenerated code onto an application server. NetBeans makes this easy. It has autogenerated an `Ant` deployment descriptor, `web.xml`. Right click on the project icon and choose `Deploy`. It may take several minutes to start the application server and deploy the web service, but it will all happen without further programmer intervention.

If you are using the `Glass Fish v2` application server, after the service is deployed, it can be tested interactively. In `NetBean`'s `Projects` window, expand the Web Services folder by clicking on the `+` sign. Right click on `QuadraticService` and choose `Test Web Service`. NetBeans will open a browser and display a page containing a form that lets you interactively enter `a`, `b`, and `c`. When you click the `solve` button, the browser sends a request to the web service, which returns the solution to the quadratic[8].

7.7.6 WSDL and Schema Revisited

How could NetBeans generate a web page to test the service? It must know where the service is located and its public API. This is exactly the information stored in the wsdl and schema documents for the service. These are quite long and complex, and we have chosen not to include them here. The key ideas to remember is that they allow the dynamic testing of the service, and shortly will be used to develop the web client.

[8]Way cool!

Request and Response Documents

The web service is incorporated into a servlet on the application server. Unlike the servlets in the previous chapter, however, the requests and responses to this servlet are XML documents. The form of these documents is specified in the web service schema.

Below is a sample request document for our web service:

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
 3     <S:Header/>
 4     <S:Body>
 5         <ns2:solve xmlns:ns2="http://QuadService/">
 6             <a>1.0</a>
 7             <b>0.0</b>
 8             <c>-4.0</c>
 9         </ns2:solve>
10     </S:Body>
11 </S:Envelope>
```

Lines	Commentary
2	SOAP documents are surrounded by a SOAP envelope. Note that this tag tells us where the schema for SOAP documents is located.
4–10	This is the body of the request.
5 & 9	We are making a request to the `solve()` method.
6–8	The call to `solve()` takes three parameters, a, b, and c.

Here is the response document:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
3     <S:Body>
4         <ns2:solveResponse xmlns:ns2="http://QuadService/">
5             <return>2.0</return>
6             <return>-2.0</return>
7         </ns2:solveResponse>
8     </S:Body>
9 </S:Envelope>
```

Lines	Commentary
4–7	This is the body of the response to our request.
5–6	Two values are returned. 2.0 and -2.0 are the solutions to the quadratic equation sent in the request.

7.8 Developing a Web Service Client

One of the strengths of the web service model is that clients can be web browsers, other web servers, or standalone applications. In this section we develop a standalone client that consumes the quadratic web service.

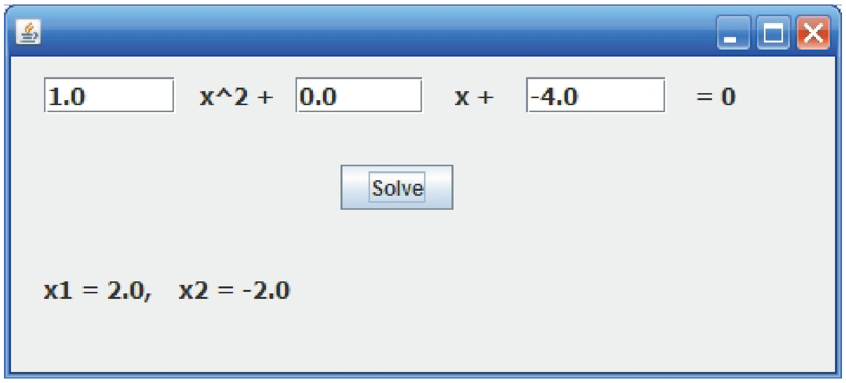


Figure 7.8: The GUI for the Quadratic Web Service Client.

7.8.1 Initial Steps

Figure 7.8 shows the GUI for the quadratic equation web service client. The user enters numbers for the coefficients of the equation and then clicks the Solve button. A message is sent to the web service, which responds with the solutions.

The initial steps to begin developing the client are:

1. Create a new NetBeans project. For this example, standard `Java Application` project is fine, although almost any type of project will work.
2. Import the web service wsdl. Make certain the web service has been deployed, then:
    - (a) Right click on the project icon and choose `New -> Web Service Client`.
    - (b) Since the web service is a NetBeans project developed on the same machine as the client, we leave the Project button checked.
    - (c) Choose Browse. Navigate to `QuadraticWebService -> QuadraticService`.
    - (d) Click OK, followed by Finish.
3. NetBeans will autogenerate proxy objects that will take care of all of the communication with the service. A new folder should appear in the project labeled: `Web Service References`. A number of Java files are generated. You don't need to do anything with these files, but you can see them by choosing the `Files` tab, then navigating to `build -> generated -> wsimport -> client -> quadservice`. The files include: `Quadratic.java`, `QuadraticService.java`, `Solve.java` and `SolveResponse.java`, as well as a couple of other support classes.

7.8.2 Completing the Client

Most of the code for the application is GUI code. Here we show only the snippets relevant to the web service exchange. The entire project is available for download from the text's website.

```
2 import quadservice.Quadratic;
3 import quadservice.QuadraticService;
```

Lines	Commentary
2–3	We import the two classes that act as the proxy for the service.

```
116     // All the work is done in the button's handler.
117     // We go to the service, solve the equation and display the results
118     private void jButton1ActionPerformed(java.awt.event.ActionEvent evt)
119     {
120         // We need a reference to the proxy (the Port) in order to call
121         // methods in the web service
122         QuadraticService quadService = new QuadraticService();
123         Quadratic quad = quadService.getQuadraticPort();
124
125         // Parse the coefficients
126         double a = Double.parseDouble(aField.getText());
127         double b = Double.parseDouble(bField.getText());
128         double c = Double.parseDouble(cField.getText());
129
130         // Find and display the solution.
131         // Note that the array of doubles in the service code has become a
132         // list in the proxy.
133         List<Double> solutions = quad.solve(a, b, c);
134         solutionsLabel.setText(”x1 = ” + solutions.get(0) +
135                                ”,    x2 = ” + solutions.get(1));
136     }
```

All of the interaction with the web service takes place in the button's handler.

Lines	Commentary
122 & 123	We obtain a reference to the web service by instantiating a `QuadraticService` object and then getting its port object. Note that these two names come from the `@WebService` annotation inserted in the web service class.
126–128	Convert the quadratic coefficients from Strings to doubles.
133	Call the `solve()` method on the web service via its proxy. Note that the web service class declared the return type to be double [ ], but here we use a List<Double>. This change of type was accomplished automatically, and was done because SOAP knows how to work with lists of objects, but not arrays of primitives.

7.9 Making Web Services More Event Based

Web services, as we have seen them so far, lack several of the properties that we have discussed as event based in earlier chapters. Since web services are servlets, they lack state, unless we use sessions or other special techniques. They also follow an asynchronous request-response interaction pattern.

While asynchronous request-response is event based, it can be argued that either a message passing system where no response is expected, or a publish-subscribe system is still more event based. For example, if no response is expected, then messages can easily be multicast, without waiting for any return messages.

7.9.1 Developing a Web Service with State

In this section, we show how to manage state in web services using session objects. Our example is a continuation of the quadratic equation problem. Rather than have the client provide the coefficients every time the service is called, however, we store the equations in session.

We only show code snippets for our web service in order to save space. As always, the complete code for the example is available on the text's website.

```
 3 import javax.annotation.Resource;
 4 import javax.jws.WebMethod;
 5 import javax.jws.WebParam;
 6 import javax.jws.WebService;
 7 import javax.servlet.http.HttpServletRequest;
 8 import javax.servlet.http.HttpSession;
 9 import javax.xml.ws.WebServiceContext;
10 import javax.xml.ws.handler.MessageContext;
```

Lines	**Commentary**
3–10	The imports are expanded to include several additional classes.
3	Resources are another type of Java annotation used to direct Java to autogenerate specific code. We will see it used below on line 23.
9 & 10	Two classes designed particularly to help us obtain and manage the session.

```
@WebService(name="QuadraticWithState", serviceName="QuadraticWithStateService")
public class QuadraticServiceWithState {

    // Some variables to help create and manage a session
    private @Resource WebServiceContext webServiceContext;
    private MessageContext messageContext;
    HttpSession session;

    /**
     * Web service operation to create an equation
     */
    @WebMethod(operationName = "create")
    public int create(
            @WebParam(name = "name") String name,
            @WebParam(name = "a") double a,
            @WebParam(name = "b") double b,
            @WebParam(name = "c")  double c) {

        // We only create a session if there isn't one already
        if (session == null) {
           messageContext = webServiceContext.getMessageContext();
           session = ((HttpServletRequest) messageContext.get(
                       MessageContext.SERVLET_REQUEST)).getSession();
        }

        Quadratic temp = new Quadratic(a, b, c);
        session.setAttribute (name, temp);
        //TODO write your implementation code here:
        return 0;
    }

    /**
     * Web service operation to solve the equation
     */
    @WebMethod(operationName = "solve")
    public double [] solve(
            @WebParam(name = "name") final String name) {
        Quadratic temp = (Quadratic) session.getAttribute(name);
        return temp.solve();
    }
```

Lines	Commentary
23–25	Three instance variables to help us create and manage the session.
23	Note the use of `@Resource`. This directs Java to instantiate and make available the WebServiceContext object. Our code never instantiates it, but can use it whenever needed. It will appear in our code on line 39.
31–35	Our first method is now named `create()`. It takes four parameters.
32	name is used as the key when storing the equation in the session.
38–42	We check whether we have already created a session object. If not, we create a new one.
39	Here, we use `webServiceContext` to help us create a session.
44	`Quadratic` is a private inner class used to represent quadratic equations. They are the objects we store in the session.
45	Store the quadratic equation in the session.
53–58	`solve()` takes the name of the equation, looks it up in the session and solves the equation.
56	We get the equation from the session.

7.9.2 Client for a Web Service with State

Developing a client for this web service follows much the same pattern as in the earlier example. Here, we give a simple text based client.

```java
import java.util.List;
import quadratic.QuadraticWithState;
import quadratic.QuadraticWithStateService;

/**
 * A simple client that manipulates quadratic equations stored in a
 * web service.
 * @author Stu Hansen
 * @version April 2009
 */
public class QuadraticClientWithState {
    public static void main (String [] args)
    {
        // We use two objects to get to the service's methods.
        QuadraticWithStateService qService = new QuadraticWithStateService();
        QuadraticWithState quad = qService.getQuadraticWithStatePort();

        // Create two quadratic equations
        quad.create("Easy", 1.0, 0.0, -4.0);
        quad.create("Tough", 1.0, 2.0, -6.0);

        // Work with an easy equation
        List<Double> solutions = quad.solve("Easy");
        System.out.println("The solutions to: " + quad.toString("Easy")
                                                                + " are: ");
        System.out.println("x1 = " + solutions.get(0));
        System.out.println("x2 = " + solutions.get(1));

        // Work with a harder equation
        solutions = quad.solve("Tough");
        System.out.println("The solutions to: " + quad.toString("Tough")
                                                                + " are: ");
        System.out.println("x1 = " + solutions.get(0));
        System.out.println("x2 = " + solutions.get(1));

        // Just to show that the "Easy" equation is still there
        System.out.print("Discriminant of " + quad.toString("Easy") + " is: ");
        System.out.println(quad.getDiscriminant("Easy"));
    }
}
```

Lines	Commentary
16 & 17	The web service reference and its port/proxy object.
20 & 21	Create two quadratic equations and store them in the web service.
23–37	Exercise various methods available in this web service. Note that `solve()` now takes the name of the equation, rather than the coefficients.

Here is the output from running the client program.

```
The solutions to: 1.0x^2 + 0.0x + -4.0 = 0 are:
x1 = 2.0
x2 = -2.0
The solutions to: 1.0x^2 + 2.0x + -6.0 = 0 are:
x1 = 1.6457513110645907
x2 = -3.6457513110645907
Discriminant of 1.0x^2 + 0.0x + -4.0 = 0 is: 16.0
```

@Oneway

Java web services also allow for `@Oneway` annotations. This annotation may be used with methods that have a `void` return type[9]. `@Oneway` is designed to improve efficiency, so that clients will not block waiting for a response from the server.

7.10 The Changing Landscape of Web Services

As stated earlier, the technologies and models used to develop web services continues to change. This section briefly presents some of the changes.

7.10.1 Web Services Inspection Language(WSIL)

UDDI registries never really caught on. In recent years, several of the larger ones have even shut down. One basic problem with them is that the client developer needs to refer to the UDDI document (located in a registry) and the WSDL document (located on a web server). The *Web Services Inspection Language (WSIL)* addresses this problem by combining the information found in the UDDI and WSDL documents into one document. WSIL documents can be stored anywhere. They don't need a special registry. They are generally found on the same machine as the web service. They can be shared publicly (if a service is trying to develop a clientele) or privately (if a service is only for already established service partners).

7.10.2 SOAP versus JSON

Because they are text, SOAP documents are readable by both humans and computers. To enhance human readability, we want each tag pair to be descriptive. For example, if we are describing a person, we might very well have the tag pair, <firstName> and </firstName>. This makes for verbose documents, however, as "firstName" is spelled out completely twice. If a network connection is slow, or there is a lot of network traffic, a long SOAP document will be slow to transmit.

[9]While Java's web services do not include OUT or INOUT parameters, other languages support this feature, and there are ways to make Java recognize them. `@Oneway` is also disallowed if these are used.

JavaScript Object Notation (JSON) is an alternative to SOAP. It is a light weight data-interchange format. JSON describes objects using name value pairs. The stock quote document above might be represented in JSON as:

```
{
    "Stock": {
        "Symbol": "IBM",
        "Last":   82.71,
        "Date":   "12/15/2008",
        "Time":   "1:00pm",
        "Change": 0.51,
        "Open":   82.51,
        "High":   82.86,
        "Low":    80.00,
        "Volume": 3007015,
    }
}
```

The code is shorter and possibly more readable.

JSON based web services have been becoming more popular in recent years.

7.10.3 RESTful Web Services

Roy Fielding, one of the principal authors of the HTTP specifications, developed the notion of *Representational State Transfer (REST)* as an architectural style for hypermedia systems, including the World Wide Web. REST is not a standard, but rather a set of principals for designing web services. The key idea is to map resources to unique URIs. For example, each method in a web service has its own URI. In the Java world, this is equivalent to saying that each method gets its own servlet. By contrast, in a SOAP based system, the method name is part of the SOAP document. The web service parses the document and then calls the method. There is a single servlet that does the parsing and dispatching for all the methods.

There are several advantages claimed by RESTful web services:

- does not require a separate discovery mechanism, as all operations are URIs,
- may be more efficient on the server side, as results can be cached,
- depends less on vendor supplied software, as there is not need for additional messaging layers, and
- provides better long term ability to evolve and change than other approaches.

7.10.4 Evolving Language Tools

A final problem to discuss is the changing set of tools available for developing web services and clients. Using Java as an example, the early package for web services was JAX-RPC. This was replaced with JAX-WS (WS stands for web services), which is not backwardly compatible with JAX-RPC. Recently, with the growing popularity of RESTful web services, Java has come out with JAX-RS. The situation is no better in other languages. Computer science students need to accept that there will be ongoing demands to upgrade their knowledge and skills throughout their careers. Web services is just one example of where this problem is currently a dominant theme.

7.11 Conclusion

Web applications are closely related to the event based paradigm. The web based front end generally contains GUI elements similar to those found in standalone GUI apps. The communication with the web server is asynchronous, because the browser does not know how long a request/response round trip will take.

In other ways, web applications are quite far from the event based paradigm. The request/response interaction with the server is blocking on the browser's side. That is, after the browser requests a new page, the browser blocks other user interactions until the page is loaded. The association of handlers (servlet code) with particular requests is done statically. The best we can do is to use request parameters to change the code that is executed via a switch statement, just as we did in the Calculator example.

Web services are more event based. Since there may be many types of clients for a web service, it is strictly up to the client developer to decide whether the client blocks of not. For some messages sent to the server, no response is even required.

Web applications and services are still new. The technologies keep changing at an alarming fast rate. The concepts of event based programming will continue to apply, however, regardless of the technology that is currently hot.

Bibliography

[Angel, 2008] Angel, E. (2008). *Interactive Computer Graphics: A Top-Down Approach Using OpenGL, 5/E*. Prentice-Hall.

[Beck and Andres, 1999] Beck, K. and Andres, C. (1999). *Extreme Programming Explained: Embrace Change, 2nd ed.* Addison-Wesley Professional.

[Beer and Heindl, 2007] Beer, A. and Heindl, M. (2007). Issues in testing dependable event-based systems at a systems integration company. *Proceedings of the The Second International Conference on Availability, Reliability and Security*, pages 1093–1100.

[Deitel and Deitel, 2007] Deitel, H. M. and Deitel, P. J. (2007). *Java: How to Program, 7/E*. Prentice-Hall.

[Deitel et al., 2002] Deitel, H. M., Deitel, P. J., and Santry, S. E. (2002). *Advanced Java 2 Platform: How to Program*. Prentice-Hall.

[Elliott et al., 2002] Elliott, J., Eckstein, R., Loy, M., and Wood, D. (2002). *Java Swing, Second Edition*. O'Reilly.

[Englander, 1997] Englander, R. (1997). *Developing Java Beans*. O'Reilly.

[Food and Drug Adminstration, 2006] Food and Drug Adminstration (2006). Fda press release (august 28, 2006): United states marshals seize defective infusion pumps made by alaris products–pumps can deliver excess medication and harm patients. Online. Internet. Available WWW:http://www.pritzkerlaw.com/alaris-infusion-pump-signature-edition/index.htm#pressrelease.

[Gamma et al., 2000] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (2000). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[Hyde, 2003] Hyde, R. (2003). *The Art of Assembly Language*. No Starch Press.

[Johnson, 2000] Johnson, J. (2000). *GUI Bloopers: Don'ts and Do's for Software Developers and Web Designers*. Morgan Kaufmann.

[Lamport, 1978] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21.

[Muhl et al., 1998] Muhl, G., Fiege, L., and Pietzuch, P. R. (1998). *Distributed Event-Based Systems*. Springer.

[Norris McWhirter, 1985] Norris McWhirter, e. (1985). *The Guiness Book of World Records, 23rd US edition*. Sterling Publishing Co., Inc.

[Shreiner et al., 2007] Shreiner, D., Woo, M., Neider, J., and Davis, T. (2007). *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 2.1, 6/E*. Prentice Hall.

[Stallings, 2008] Stallings, W. (2008). *Operating Systems: Internals and Design Principles, 6/E*. Prentice Hall.

[Tanenbaum, 2007] Tanenbaum, A. S. (2007). *Modern Operating Systems, 3/E*. Prentice Hall.

[Trewin and Pain, 1998] Trewin, S. and Pain, H. (1998). A model of keyboard configuration requirements. In *Assets '98: Proceedings of the third international ACM conference on Assistive technologies*, pages 173–181, New York, NY, USA. ACM.

[Walrath et al., 2004] Walrath, K., Campione, M., Huml, A., and Zakhour, S. (2004). *The JFC Swing Tutorial: A Guide to Constructing GUIs (2nd Edition)*. Addison-Wesley.