Chapter 17. Web-Application Development

Table of Contents

Objectives

At the end of this unit you will be able to:

- Understand some web applications;
- What servlets do;
- Write a web application using servlet packages and classes.

17.1 Introduction

Web applications are computer applications that the user does not run directly on their own computer, but rather accesses through a Web browser. The GUI is provided by the Web browser (usually as some form of HTML), and all of the application's processing is done either on the browser with JavaScript, on the Web server itself, or on both. This means that Web application development is, in essence, the development of network-based applications, including server-side (i.e., on the Web server) and client- side (i.e., on the Web browser) programming. This module has so far specifically dealt with client-side development, including HTML and JavaScript, but this chapter deals with server-side development.

There are many ways to programme a Web server, including such commonly used technology as PHP and ASP.net. The traditional method of doing so is to use the Common Gateway Interface (CGI), which is a language agnostic interface for adding programmability to Web servers. It allows CGI programmes to be written in any language, and these programmes are executed by the Web server whenever a resource which they supply is requested by a Web browser.

This chapter will introduce you to the equivalent Java solution: Java Servlets. This will allow you to make use of the knowledge that you've gained in the programming module, rather than

having to learn a new language.

Servlets are Java objects which are run on a Servlet aware Web server. Each Servlet object is mapped to particular URLs on the server. Instead of simply returning a Web document when a request for an URL is made, the server instead uses the Servlet mapped to the URL to handle the request. The Servlet is free to deal with the request however it feels it should – it can communicate with databases, read or write to the file system, and create arbitrary response data to be returned to the Web browser. Typically, however, this response data is HTML, which the browser than displays for the user.

While Servlets makes use of your Java knowledge, they also have another advantage: because they are designed to replace a portion of a Web server's functionality, they will also give you insight into how a Web server deals with a request for a Web page, more so than many other Web application frameworks. This is always useful knowledge to have, no matter what Web framework you will use in the future.

To begin, you will need access to a Web server able to run Servlets. You can use any such server that you have access to, but these notes will make use of Apache Tomcat, and all of the examples will reflect this. Instructions on how to install and use Tomcat were given in Chapter 2.

17.2 Examples of Web applications

The Web has grown to make use of a large number of Web applications. Typical examples that you may have come across include Web-based e-mail clients, online shopping sites, blogs, wiki-wiki, and so on. Visit the sites below to see what kind of functionality they provide:

- Amazon [http://www.amazon.com]
- Kalahari.net [http://www.kalahari.net]
- E-Bay [http://www.ebay.com]
- GMail [http://www.gmail.com]
- Wikipedia [http://www.wikipedia.org]
- Open Street Map [http://www.openstreetmap.org]
-

17.2.1 Blogs

A Web log (or blog) is often described as an online diary, but they have developed into websites that may contain commentary on events and media (such as reviews), original work (such as fiction, or comics), or contain various other media besides text, with video now becoming fairly common. As such, the main feature of blogging software is that it allows the blog's author to regularly and easily add content to it, and this content is typically displayed in chronological order (much like a diary), usually from most recent to least recent. Blogs also provide some functionality to the reader, such as the ability to search through the various entries, or to leave comments for the author and other readers. Blogs can also be written by more than one person, and the software will usually keep track of which author has written which entry.

TO DO

Visit a blog service provider such as WordPress, LiveJournal or BlogSpot, and read about the service. Sign up for an account if you are interested to see how such software works.

17.2.2 Wikis

A wiki is a content management system that allows people to view, add or modify the content that it contains. Unlike a blog, which always has a specific person or group of people acting as an author, a wiki usually lets anyone edit the content, and can be described as a collaborative content creation effort by all of its users. Here are some of the common features:

Prior registration to use the service is not typically necessary. This is meant to encourage everyone to

contribute, although it does mean that vandalism can more easily occur.

The editing process is often simple, and requires no review procedure prior to publication. In other words, changes are immediately made available to everyone. Again, this is to encourage high levels of contribution.

A wiki system may keep track of the changes made to the content, allowing the users to see the individual changes and when they were made. It also makes it easy to revert content back to an older version if necessary.

One of the main advantages of a wiki is that the knowledge and content that it contains is community-based. This might not mean that the content is more accurate, just more representative of its users' views. A good wiki page should contain a wide variety of views, and as such censorship is frowned upon in many wiki sites. Another advantage is that the system allows for information to be kept up to date with minimal maintenance by any one group of people.

TO DO

Visit a wiki site such as Wikipedia, a wiki-based encyclopaedia. Search for a topic and examine the page. Look closely at the discussion, edit and history tabs (on top of each page). Visit an entry on the subject that you might be an expert in. Can you see if there is anything missing? Find out how you could contribute. You could also visit the 'Help' ulink url read about different aspects of the site, such as the standards that people contributing to the site must adhere to.

17.2.3 Sakai

Sakai is an online educational course management system. It allows for managing course resources (notes, exercises, answers etc.), supplies forums, chat rooms, surveys, electronic hand-in facilities, schedulers and calendars, among various other features. The system allows both teachers and students to make use of it, offering them each features appropriate to the different roles that they play. At the University of Cape, our Sakai instance is called Vula, and all students have access to it.

TO DO

Visit the Vula site if you have not been regularly visiting. Do you think that such a facility has been useful to you? How could it be improved?

17.2.4 Digital Libraries

A digital library, like its real-world counterpart, is a collection of books and reference materials, albeit with the exception that the content is electronic and usually provided over a network. An example of a digital library is Project Gutenberg [http://www.gutenberg.org], a collection of freely available books no longer under copyright protection.

Digital libraries have a number of advantages of traditional libraries, including increased capacity, lower costs and easier accessibility. A digital library can contain large volumes of information that compared to the amount of physical space that it requires. It can also be cheaper to operate one as it requires less maintenance and staff costs, and being available over a network means that it can be used to by people all over the world if the library is made available over the Internet.

However, there are also some disadvantages. For example, there are concerns that copyright issues are hampering the effectiveness of digital libraries, while the sheer volume of information has caused the traditional techniques of finding information to become inefficient. Additionally, many people find it difficult to read long documents on a computer screen, and so they are often printed out, costing money and wasting paper.

TO DO

Visit a digital library such as Project Gutenberg. Find a book that you're interested in reading and see if you can download it. Read the library's conditions of use. What do you think of Project Gutenberg? Would you make use of it regularly?

17.3 What do Servlets do?

Servlets replace how a Web server responds to a request for some resource. In this way they are invisible to someone visiting a Web application using a browser: the browser makes standard requests for standard Web resources, the addresses of which usually look like ordinary Web pages. The server, on the other hand, will realise that a specific address is for a Servlet in a given Web application. It will then pass control to the Servlet to handle the whole request. Any data sent to the server with the request is past to the Servlet, and the Servlet may return any data it wishes (including HTML) to the Web browser.

Because Servlets replace a portion of a Web server's functionality, we need to know a bit about HTTP, the HyperText Transfer Protocol used by Web servers and Web browsers to communicate with each other.

17.3.1 Request Methods

When a Web browser requests a Web page, that request is made using HTTP. Now, HTTP defines a number of actions that can be taken on a Web page. These actions are called request methods. The two that are of interest to you in this chapter are GET and POST requests. Both requests are used when data (for instance, an HTML file) is being requested from a Web server. A GET request, in particular, is used only for requesting data, although it does have some ability to pass data to a Web server as well. POST requests also request data, but importantly, data is meant to be sent to the Web server as well, and this data is meant to be processed. A POST request may result in a Web application changing its state (as it processes the data). A GET request should never change the state of a Web application. Indeed, GET is defined by HTTP to be a Safe Method, which are request methods that never modify the state of the Web application.

When an HTML file is requested by a Web browser, this request is by default made as a GET request. The important exception to this is with HTML forms, which can request that the data be sent using a POST request (by setting the method attribute of the form tag to “POST”). The Web page set as the form's action will then be requested using a POST request.

When you are writing your own Servlets, you will have the chance to directly make use of, and implement, GET and POST requests. You will notice little difference between them apart from how they transfer data from the Web browser to the Web server: a GET request will always display the sent data in the URL, while a POST request will generally never do this. Apart from this, you must always remember to use the two request types only where they are intended. Specifically, it is incorrect to update the Web server's state using a GET request.

17.4 The structure of a Web application

Since the Servlet API 2.2 specification, every Servlet aware Web server is required to accept Web applications in a standard format. A Servlet Web application is defined as a collection of directories and files, all of which can possibly be compressed into a single file called a Web application archive (WAR). These WAR files are a useful way to distribute your Web application once you have written it, but we will not be covering them in these notes.

The Servlet specification defines in what directories various files making up the application should be placed. If our Web application is stored in a directory called APPLICATION, then the various files should be laid out as follows:

- HTML files are stored directly in the APPLICATION directory.
- The APPLICATION/Web-INF directory contains metadata about the Web application itself, and the various Servlets which make up the application. This metadata is stored in the APPLICATION/ Web-INF/web.xml file.
- APPLICATION/Web-INF/classes contains the class files implementing the various Servlets.
- APPLICATION/Web-INF/lib contains any JAR files the application uses.

All of the following examples use the layout described above.

17.5 Your first Web application

Now for a Web application that dynamically creates the simple Web page previously created using Java. As before, we begin by creating a subdirectory within webapps called “hello”.

Now we create the Java class which will handle the request for the Web page. In your favourite text editor, create a new file called HelloServlet.java and add the following text:

```
import java.io.*; import javax.servlet.*;
import javax.servlet.http.*;

public class HelloServlet extends HttpServlet
{
    public void doGet(HttpServletRequest request,
HttpServletResponse response)


    throws ServletException, IOException
    {
        PrintWriter out = response.getWriter();


out.println("<html>\n\t<head>\n\t\t<title>Hello!</title>\n\t</head>
”);
        out.println("\t<body>\n\t\tHello, world.\n\t</body>”);
    }
}
```

The javax.servlet packages are distributed with Tomcat in the servlet-api.jar file. You will find this in the “lib” subdirectory of your Tomcat distribution. Ensure that this file is in your classpath when compiling this code. If you are compiling from the command line, you can do this with the following line:

```
javac    -classpath    "<path    to    Tomcat>\lib\servlet-api.jar"
HelloServlet.java
```

Replace <path to Tomcat> with the Tomcat directory.

The first three lines import packages used in the code, and the contents of the two Servlet packages will be covered later. All of your Servlets will extend the HttpServlet class, which you will find in the javax.servlet package.

The class has one method, doGet(). This method is used to implement a GET request. This is appropriate, as when the Web browser requests the page produced by the Servlet it will be done using a GET request. doPost() is a similar method which takes the same arguments, but is called when a POST request is made. A Servlet may implement either or both methods, as it requires.

doGet() takes two arguments: an HttpServletRequest and HttpServletResponse. These encapsulate information concerning the request and the response the Servlet will make, in respect. This example is extremely simple, and only returns a message to the Web browser. It does this by getting the PrinterWriter instance provided by the response object. This is the exact same type of object as System.out, only it is used to return information to a Web browser.

Next, the class writes HTML output to the PrinterWriter instance. Notice that the output should produce exactly the same HTML as the previous example.

Compile the file. Inside the “hello” directory, create a directory called “Web-INF”, and inside this create a directory called “classes”. Copy the HelloServlet.class to this directory.

The Web application now needs to describe itself to Tomcat (or any other Servlet aware Web server it might run on) and also tell it what to do with the Servlet classes. This is done using the web.xml file.

Create web.xml inside the Web-INF directory, and fill it with the following text:

```
<?xml version="1.0"?>
<!DOCTYPE Web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
    2.3//EN" "http://java.sun.com/dtd/Web-app_2_3.dtd">

<Web-app>
  <display-name>The Hello World Application</display-name>
  <description>Prints hello world!</description>

  <servlet>
    <servlet-name>hello</servlet-name>
    <description>This is a simple Servlet</description>

    <servlet-class>HelloServlet</servlet-class>

  </servlet>

  <servlet-mapping>
    <servlet-name>hello</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>
</Web-app>
```

The first two lines declare that this file is an XML file, and gives the files document type. The document type should always be the same. The Web-app element is used to describe the application as a whole.

<display-name> and <description> describe and name the application to the Web server. Each Servlet provided by the Web application needs to be described in <servlet> elements. <servlet-name> must provide a name for the Servlet. This name does not have to be the same as the class name or the .class filename: it is the name that you will use throughout the rest of the web.xml when talking about the Servlet. <description> describes the Servlet in plain English. <servlet-class> names the actual Servlet class, in this case it is HelloServlet.

Each new Servlet must be described in its own <servlet> element, and each must have a unique name.

So far all that has been done is describe what Servlets are available to the Web application – now we need to tell it what to do with each Servlet. This is done with the <servlet-mapping> element. Each <servlet-mapping> element will map a particular Servlet (identified by its Servlet name) to a particular URL relative to the base Web application URL, as given by <url-pattern>. For instance, the above example of <url-pattern>/</url-pattern> says that any request made to the root address of the Web application should be handled by this Servlet. In this case, the URL would be http://localhost:8080/ hello. On the other hand, if we had used <url-pattern>/greetings</url-pattern>, then the address of the Servlet would be http://localhost:8080/hello/greetings.

Once you have saved the web.xml file to Web-INF, and copied the .class file to the classes directory, restart Tomcat and visit http://localhost:8080/hello. You should see exactly the same message as the previous simple Web page produced. You can view the produced HTML by asking for the page source from your browser.

What has happened is that the web.xml has made Tomcat aware that all requests made to the root address of the Web application should be handled by the HelloWorld Servlet class. When your Web browser requests the address, it does so using the GET request method. Tomcat forwards this GET request to an instance of HelloServlet, which then prints out the HTML code returned to the Web browser.

17.6 Handling user input

In this example we will create a Web application which asks the user for their name, and then greets them. The greeting occurs on a separate page. Note that it is possible to do this all with JavaScript

and the DOM in one page, on the browser. It is also possible to do this using only one page in a Web application, which we will do in the following chapter. For simplicity, however, we will use two Web pages.

As previously noted, GET and POST requests can both send data to the Web server, although both do so differently. Specifically, a GET request will encode the information in the actual URL, while a POST request hides the information in the HTTP header itself. Users will always able to see any information sent to the server using a GET request in the URL.

Because the two methods send data differently, the data must also be obtained from each request in a different way. For our convenience, the Servlet API hides all of this detail from us: all of the data is made available through the HttpServletRequest object, no matter which request method has been used, and no matter how the data was sent to the server.

We begin by creating a new Web application: create a directory in the Tomcat webapps subdirectory called “greetings”. Now we create a simple page that asks for the users name:

```
<html>
 <head>
   <title>Some Greetings</title>
 </head>
 <body>
   What is your name?
   <form action="response" method="GET">
     <input type="text" name="username">
     <input type="submit">
   </form>
 </body>
</html>
```

We save this as index.html. Of interest is the form. Notice that it sets its method to GET, which specifies how the information in the form is communicated to the server. It also is a hint that none of the Web application's internal state will change, otherwise POST would have been used. The form's action has also been set to “response”, which is the URL which the form's information will be sent to when the submit button is pressed. The form itself contains a text entry for the user to enter their name. The text entry has been called “username”. The Servlet will use this name when requesting the value in the text box.

You can view this page by opening the http://localhost:8080/greetings in your Web browser. “greetings” is just the directory name which you created in the webapps directory. Clicking the submit button should result in a page not found error.

We will now create the Servlet providing the response page:

```
import java.io.*; import javax.servlet.*;
import javax.servlet.http.*;

public class ResponseServlet extends HttpServlet
{
    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
 throws IOException
    {
 String name = request.getParameter("username");
 PrintWriter out = response.getWriter();

 out.println("<html><title>Hello! " + name +
 "</title</head>"); out.println("<body>\n<h1>Hello, " +
 name + ".</h1>"); out.println("It's good to meet you.");
    }
}
```

And here is a description of it, and the Web application, in the web.xml file.

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

<!DOCTYPE Web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
    2.3//EN" "http://java.sun.com/dtd/Web-app_2_3.dtd">

<Web-app>

  <display-name>A Greetings Application</display-name>
  <description>Asks the user's name and then greets
  them.</description>

  <servlet>
    <servlet-name>ResponseServlet</servlet-name>
    <description>Performs the actual
    greeting.</description>

    <servlet-class>ResponseServlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>ResponseServlet</servlet-name>
    <url-pattern>/response</url-pattern>
  </servlet-mapping>
</Web-app>
```

This Servlet uses the getParameter() method to request the value associated with the “user_name” entry in the form. getParameter() always returns a string, unless there is no entry in the form with the name passed as an argument to getParameter() – in this case it returns null.

Restart Tomcat. When you now open the Web application in your browser, you can enter your name and click the submit button. The application will now take you to a page greeting you by name.

This Servlet does currently have one obvious bug: it does not notice when no name has been entered, but the submit button has been pressed. Try this to see what happens. This problem can be fixed in two ways: first, JavaScript could be used in the browser to test if a name has actually been given. Secondly, the Servlet can itself test to see if a name has been given or not. We will implement the second method by modifying the Servlet's code to the following:

```
if (name.equals("")) {
    out.println("<html><title>Please enter your
    name</title</head>"); out.println("<body>\n<h1>Oh
    no!</h1>");
    out.println("You have not entered your name. Please
    press the back button o
 }
else {
    out.println("<html><title>Hello! " + name +
    "</title</head>"); out.println("<body>\n<h1>Hello, " +
    name + ".</h1>"); out.println("It's good to meet
    you.");
}
```

Recompile the class and copy it into the classes subdirectory. Restart Tomcat and retest the Web application's behaviour when not entering a name.

This has just been a simple example of a Web application. Later, we will build a Servlet which will store state not only on the server, but also on the browser.

17.7 Cookies and session tracking

This example shows you how to track users visiting your site.

HTTP has a specific problem: when a Web server receives a request for a particular page, let us say page A, followed by another request for a page, let us say page B, the server has no easy way to tell if the same user has made both requests, or if the two requests have come from two different users. This complicates the creation of a Web application: the Web application must, in some way, be able to track its users as they move from page to page, and the application must implement a method to do this itself. There are a number of ways to do this, the most simple of which is to use HTTP cookies.

A cookie is text which the Web application sends to the Web browser. The browser stores this text and sends it to the server on every HTTP request which it makes. A cookie can generally store up to four kilobytes of any text a Web application chooses to store in it. This text can be, for example, a unique ID to identify the user. Because the cookie is returned to the server with every HTTP request, the unique ID can be used to identify which user is accessing the page, and so can be used to track the user as they move across the different pages in a Web application. This is important for, say, for letting any changes the user made to the Web application persist while the user is using the site.

The following example shows how to implement a simple hit counter. A hit counter tracks the number of unique visits that a Web page receives. Our Web counter will not count multiple visits from the same person on the same day, so that no one person can arbitrarily increase the number of hits a site seems to be receiving.

The counter we will create will use a file called “counts.txt” which will store the number of hits the page has received. To begin, create a new directory in the webapps subdirectory called “counter”. We will only have one page in the Web application, which will be implemented with the following Servlet:

```
import java.io.*; import javax.servlet.*;
import javax.servlet.http.*;

class CounterException extends ServletException { public
    CounterException(String message)
    {
 super(message);
    }
}

public class Counter extends HttpServlet implements

    SingleThreadModel { private final String file_name =

    "counts.txt";
    private final String cookie_name = "visited"; private
    final String cookie_value = "returnSoon";

    private String getCookieValue(HttpServletRequest
      request, String name) throws CounterException
    {
 Cookie cookies[] = request.getCookies(); Cookie cookie;

 if (cookies == null)
     throw new CounterException("No cookies have been
 set");

 for (int i = 0; i < cookies.length; i++) { cookie =
     cookies[i];
     if (cookie.getName().equals(name)) return
  cookie.getValue();
 }
```

```
throw new CounterException("Unable to find a cookie named
      " + name);
   }

   private final String deleteCookieJavaScript(String
   name)
   {
return "var date = new Date();\n" + "document.cookie = '"
    + name + "=deleted;" + "expires=' +
    date.toGMTString() + ';;';\n" +


    "alert('cookie deleted.');";
   }


   public void doGet(HttpServletRequest request,
       HttpServletResponse response)
   {
BufferedReader file = null; int count = 0;

try {
    file = new BufferedReader(new FileReader(file_name));
    count = Integer.parseInt(file.readLine());
}
catch(IOException e) {
}
finally {
    try {
 if (file != null) file.close();
    }
    catch (IOException b) {}
}


String value = null; try {
    value = getCookieValue(request, cookie_name);
}
catch(CounterException e) {
    Cookie cookie = new Cookie(cookie_name,
    cookie_value); cookie.setMaxAge(60 * 60 * 24);
    response.addCookie(cookie);
    count += 1;
}


PrintWriter writer = null; try  {
    writer = new PrintWriter(new FileWriter(file_name));
    writer.println(count);
    writer.close(); writer = null;


    PrintWriter out = response.getWriter();
    out.println("<html>");
    out.println("\t<head><title>Hello, visitor number " +
  count + ".</title></head>");
    out.println("<body>");
    out.println("<p>Hello, visitor number " + count +
    ".</p>"); out.println("<form><input type='button'
    value='delete cookie' onclick=\""
```

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


            if (writer != null) writer.close();
        }

       }
    }
```

The web.xml file is as follows:

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

<!DOCTYPE Web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
    2.3//EN" "http://java.sun.com/dtd/Web-app_2_3.dtd">

<Web-app>
  <display-name>The counter application</display-name>
  <description>
    This application counts the number of unique visits it
    receives.
  </description>

  <servlet>
    <servlet-name>Counter</servlet-name>
    <description>
      A simple Servlet that uses cookies to track the
      number of unique visits it receives per day.
    </description>

    <servlet-class>Counter</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>Counter</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>
</Web-app>
```

As usual, make sure that the .class file and web.xml are placed in their appropriate directories. Let us look at the doGet() method. It begins like so:

```
        BufferedReader file = null; int count = 0;

        try {
            file = new BufferedReader(new FileReader(file_name));
            count = Integer.parseInt(file.readLine());
        }
        catch(IOException e) {
        }
        finally {
```

```
    try {
 if (file != null) file.close();
    }


catch (IOException b) {}
}
```

This code attempts to open the “counts.txt” file and reads in the single integer value that it contains. The integer “count” is initialized to the value zero. The code in the try block performs the actual file read. It needs to handle two exceptional situations: a.) the first time that the application is run, the file will not yet exist; b.) the Servlet may have a problem reading the file in general. Both problems will throw exceptions of type IOException, which we catch. We use a finally block to ensure that the file is always closed if it has been created. If an exception has occurred, we do nothing except let the doGet() method continue. This is perfectly fine: nearly all of the exceptions will have occurred because the “counts.txt” file does not exist. In this case, the count variable should be set to zero, which we have ensured it is when we declared it. More fine grained error control could be obtained by reporting an error for IOException (how to do this is explained later), and also catching the FileNotFoundException, in which case we do nothing and let the method continue, as above.

Now that we have read the hit count (if it so far exists), we need to decide if the counter should be incremented. This Web application uses cookies to determine if a user has previously visited the page. The following code examines if a cookie had previously been stored on the user's computer:

```
String value = null; try {
    value = getCookieValue(request, cookie_name);
}
catch(CounterException e) {
    Cookie cookie = new Cookie(cookie_name,
    cookie_value); cookie.setMaxAge(60 * 60 * 24);
    response.addCookie(cookie);
    count += 1;
}
```

getCookieValue() will throw an exception if the “visited” cookie has not been set. If this happens, the Web application creates a new cookie (which is, conveniently, an object of type Cookie) and tells the response object to store it on the users computer. The counter is also incremented. Notice that the counter is not incremented if the cookie exists. The Cookie constructor takes two arguments: the first is the cookie's name, and the second is an arbitrary text value which must be smaller than four kilobytes (a fairly substantial amount of text). By default, a cookie will only last for as long as the browser is open – the cookie is deleted when the user shuts their browser down. However, we would like the cookie to be stored on the user's computer for a whole day, so that our hit counter does not count multiple visits made on the same day. This is done using the Cookie object's setMaxAge() method. It accepts the number of seconds for which the cookie must be stored on the computer. We supply it with the number of seconds in a day (60 seconds in a minute, 60 minutes in an hour, 24 hours in a day).

The work of reading the cookie is done by the getCookieValue() method :

```
   private String getCookieValue(HttpServletRequest
     request, String name) throws CounterException
   {
Cookie cookies[] = request.getCookies(); Cookie cookie;

if (cookies == null)
    throw new CounterException("No cookies have been
set");

for (int i = 0; i < cookies.length; i++) { cookie =
```

```
    cookies[i];

    if (cookie.getName().equals(name)) return
 cookie.getValue();
}

throw new CounterException("Unable to find a cookie named
      " + name);
   }
```

A Web application can store multiple cookies (each with a different name) on a computer. You can gain access to the cookies using the request object's getCookies() method, which will return an array of cookies. getCookieValue() searches the array for the Cookie with the given name, and returns its value. The first time the application is run, getCookies() will return null, since no cookies have yet been set. In this case, or in the case where the requested cookie isn't found, we throw an exception of type CounterException.

Finally, we return some HTML and update the counts.txt file.

```
PrintWriter writer = null; try  {
    writer = new PrintWriter(new FileWriter(file_name));
    writer.println(count);

    PrintWriter out = response.getWriter();
    out.println("<html>");
    out.println("\t<head><title>Hello, visitor number " +
  count + ".</title></head>");
    out.println("<body>");
    out.println("<p>Hello, visitor number " + count +
    ".</p>"); out.println("<form><input type='button'
    value='delete cookie' onclick=\""
    out.println("</body>");
    out.println("</html>");
}
catch (IOException e) {
}
finally {
    if (writer != null) writer.close();
}
```

The first few lines opens “counts.txt” for reading, and writes the new counts value. Notice that this application can be streamlined by only writing to the file if the count variable had been incremented.

Next, we output the actual HTML. The page prints a greeting to the visitor, and tells them what the hit count is. Also, to test the application, it provides two buttons. One of which is a submit button which you can use to reload the page. The other shows how to delete cookies by setting their maximum age to zero. This can either be done in the Web application using setMaxAge(0), or it can be done as we do it here, using JavaScript. This code also catches IOExceptions, which can be thrown when writing to either the counts.txt file or the response object. The writer object is also closed in a finally block, to ensure that it is always closed.

The Web application should, when you load it (http://localhost:8080/count/), tell you the number of times the site has been visited. Pressing the reload button does not increment this number, except after you have pressed the 'delete cookie' button. You can completely reset the counter by removing the “counts.txt” file, which should be in Tomcat's bin directory.

Our above Servlet implements the SingleThreadModel interface. It does this because, typically, a Web application is used by multiple users at the same time, each of them trying to view this same page.

Tomcat spawns a thread to handle each user, and each thread will run the doGet() method. This makes

it possible that each thread may be trying to read and write from “counts.txt” at the same time. Clearly this could cause a problem (consider what would happen if one file reads counts.txt, updates the count variable, but before it can write the new value to the file another thread updates the file instead; now this update is going to be overwritten with an old value). The simplest solution, and the one used here, is to declare the class as implementing SingleThreadModel. This is an interface with no methods – all it does is to tell Tomcat to never use more than one thread at a time for this Servlet. This ensures that there is never a problem with multiple threads attempting to read and write to the file. This does have some performance implications for your Web application: it means that each user viewing the page must wait in turn for Tomcat to finish serving the page to the previous users. This is not an ideal solution when the number of users visiting the site increases. Because of this, and because Java already has its own techniques for handling synchronisation, SingleThreadModel has been deprecated, and no replacement has been offered. You should be using the concurrency mechanisms offered by the Java language to ensure that this type of problem does not occur – unfortunately a discussion of these methods is beyond the scope of these notes, but we do leave you with a simple example to show how this could be done:

```
   public void doGet(HttpServletRequest request,
       HttpServletResponse response)
   {
BufferedReader file = null; int count = 0;

synchronized (this) { try {
 file = new BufferedReader(new FileReader(file_name));
 count = Integer.parseInt(file.readLine());
    }
    catch(IOException e) {
    }
    finally { try {
     if (file != null) file.close();
 }
 catch (IOException b) {}
    }


    String value = null; try {
 value = getCookieValue(request, cookie_name);
    }
    catch(CounterException e) {
 Cookie cookie = new Cookie(cookie_name, cookie_value);
 cookie.setMaxAge(60 * 60 * 24);
 response.addCookie(cookie);
 count += 1;
    }


    PrintWriter writer = null; try    {
 writer = new PrintWriter(new FileWriter(file_name));
 writer.println(count);
 writer.close(); writer = null;

    }
    catch (IOException e) {
    }
    finally {
 if (writer != null) writer.close();
    }
}

try {
    PrintWriter out = response.getWriter();
```

```
      out.println("<html>");
      out.println("\t<head><title>Hello, visitor number " +
    count + ".</title></head>");
      out.println("<body>");
      out.println("<p>Hello, visitor number " + count +
      ".</p>"); out.println("<form><input type='button'
      value='delete cookie' onclick=\""
      out.println("</body>");
      out.println("</html>");
  }
  catch (IOException e) {
  }

     }
```

Notice that all of the instructions concerning the counts.txt file has been placed into a synchronized block.

17.8 Problems with cookies

Cookies are not the only way which can be used to track a user as they move between the different pages of a Web application. Cookies are convenient because they can be set up so that a user can completely leave the Web application, later return to it, and continue what they were doing without any loss of information. However, the user may have disabled cookies on their browser, causing any Web application using only cookies to track its users to no longer function correctly. Two other methods that can be used are “URL rewriting” (which keeps track of a user by modifying the query string at the end of an URL) and using hidden form fields to store user data.

However, to ease Web application development, the Servlets API provides a unified interface to using Cookies and URL rewriting. This is the Session Tracking API, made available to HTTP Servlets through the HttpSession interface. HttpSession supplies a consistent way of storing and retrieving data about the user, and should generally be used in place of cookies.

Session tracking is easy to use. The first step is to ask for the current session from the HttpServletRequest object, like so:

```
HttpSession session = request.getSession();
```

This will return the current user's session object. If the user is using the Web application for the first time and no session object currently exists, a new one will be created. If you would like to ensure that a new session object is not created by default, you can use the second form of the method:

```
HttpSession session = request.getSession(false);
```

If the session does not already exist, getSession(false) returns null. getSession(true) is exactly the same as getSession().

Once you have the session object, you can store arbitrary information about the user, like this:

```
session.setAttribute(“name”, value);
```

“name” can be an arbitrary string that gives a name for the value. The value variable itself can be any Java object, but in these notes we will only be using strings. The value can be read like so:

```
String value = (String) session.getAttribue(“name”);
```

Notice that you have to cast the object returned from getAttribute().

This is all there is to using session objects. Below is an updated version of the hit counter Servlet that uses the session tracking API rather than cookies. There are a number of ways that this could be done, including one in which no attribute needs to be set. We will, however, be setting an attribute.

```
import java.io.*; import javax.servlet.*;
import javax.servlet.http.*;

public class Counter extends HttpServlet {

    private final String file_name = "counts.txt"; private
    final String attribute_name = "visited"; private final
    String attribute_value = "returnSoon";

    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
    {
 BufferedReader file = null; int count = 0;

 synchronized (this) { try {
  file = new BufferedReader(new FileReader(file_name));
  count = Integer.parseInt(file.readLine());
     }
     catch(IOException e) {
     }
     finally { try {
      if (file != null) file.close();
  }
  catch (IOException b) {}
     }

     HttpSession session = request.getSession(); String
     value = null;

     value = (String)
     session.getAttribute(attribute_name);
     if (value == null) {
  session.setAttribute(attribute_name, attribute_value);
  session.setMaxInactiveInterval(60 * 60 * 24);
  count += 1;
     }
 Writer writer = null; try {
  writer = new PrintWriter(new FileWriter(file_name));
  writer.println(count);
  writer.close(); writer = null;
     }
     catch (IOException e) {
     }
     finally {
  if (writer != null) writer.close();
     }
 }

 try {
     PrintWriter out = response.getWriter();
     out.println("<html>");
     out.println("\t<head><title>Hello, visitor number " +
   count + ".</title></head>");
     out.println("<body>");
     out.println("<p>Hello, visitor number " + count +
     ".</p>"); out.println("</body>");
     out.println("</html>");
```

```
 }
 catch (IOException e) {
 }
    }
}

     PrintWriter writer = null; try    {
  writer = new PrintWriter(new FileWriter(file_name));
  writer.println(count);
  writer.close(); writer = null;
     }
     catch (IOException e) {
     }
     finally {
  if (writer != null) writer.close();
     }
 }

 try {
     PrintWriter out = response.getWriter();
     out.println("<html>");
     out.println("\t<head><title>Hello, visitor number " +
   count + ".</title></head>");
     out.println("<body>");
     out.println("<p>Hello, visitor number " + count +
     ".</p>"); out.println("</body>");

     out.println("</html>");
 }
 catch (IOException e) {
 }
    }
}
```

Notice how, in the trivial case, the session tracking API is easier to use than Cookies. There are some important differences between this version and the cookie version, however: you cannot easily invalidate session tracking by simply deleting a cookie, since that cookie may not even exist; setMaxInactiveInterval() sets the time, in seconds, from which the user last accessed the page to when the session should be invalidated(). In contrast, setMaxAge() is the time in seconds from the cookie's creation. To get identical behaviour you would have to use the session object's getCreationTime() (which returns time in seconds) and getLastAccessedTime() (which returns the time in seconds since the user last accessed the Web application) methods. If the difference between those two times is larger than a full 24 hours, then the session should be manually invalidated.

17.9 Response status

The Cookie-based counter example had various exception handlers which should have reported an error message and stopped the execution of the doGet() method. However, it is not clear how an error message might be reported. For instance, when writing output to the response object, it itself may throw an IOException which needs to be reported in some way. The way to do this over HTTP is to use the response status. This is a number which is returned the the Web browser from the server to indicate the status of the HTTP request.

Some commonly used response statuses are:

- 200 – OK
- 403 – Forbidden
- 404 – Not found
- 500 – Internal Server Error

When a page is successfully returned to the browser, HTTP also supplies the browser with the value 200 to indicate the the request was successful. If a request is made to a page which the user is not allowed to view, the response status 403 is given to the browser; 404 is returned when the requested page was not found. 500 is returned when there was an error processing a page. In all of these cases actual content to be displayed can also be returned. For instance, a special message indicating that the requested page was not found could be returned with the appropriate response.

Many of the response status codes are automatically sent by the Servlet based on context. For instance, if the doGet() or doPost() methods return successfully, and they have not explicitly requested a different status code, 200 will automatically be sent. For sending status codes in general, the response.setStatus(int) method can be used. The HttpServletResponse object also contains many public static final member variables that contain the appropriate values. For instance, the four codes presented above are contained in the SC_OK, SC_NOT_FOUND, SC_FORBIDDEN, SC_INTERNAL_SERVER_ERROR variables.

If you want to both return HTML and set a status other than 200, be sure to set the status before you sending any output.

Errors can now be checked for and reported in our Web applications. For example, in the above counter code we could have done the following:

```
    try  {
writer = new PrintWriter(new FileWriter(file_name));
```

```
writer.println(count); writer.close();
writer = null;
   }
   catch (IOException e) {
response.setStatus(HttpServletResponse.SC_INTERNAL_SERVE
R_ERROR); return;
   }
   finally {
if (writer != null) writer.close();
   }
```

Which would have stopped execution of doGet() and notified an error. This is still not as useful as it could be: the user is not given any indication of what failed, or why. The sendError() method allows for this:

```
   try  {
writer = new PrintWriter(new FileWriter(file_name));
writer.println(count);
writer.close(); writer = null;
   }
   catch (IOException e) { try {
response.sendError(HttpServletResponse.SC_INTERNAL_SERVE
R_ERROR,
                           }
catch (IOException e2)
{
response.setStatus(HttpServletResponse.SC_INTERNAL_SERVE
 R_ERROR); return;
}
return;
   }
   finally {
if (writer != null) writer.close();
   }
```

Notice that sendError can itself throw an IOException; if it does, you should fall back to only setting the status.

17.10 The Servlet packages and classes

There are two main packages which this chapter makes use of: javax.servlet and javax.servlet.http.

17.10.1 javax.servlet

The javax.servlet package provides all of the base classes and interfaces defining both what a Servlet is, and how it interacts with the Web server that is running it. Of special note are:

Interfaces

- Servlet – This defines all the methods that a Servlet must implement. It includes methods to initialize the destroy the Servlet, and a general method (service()) which handles all requests made to it.

Note that classes which implement the Servlet interface need not only handle HTTP requests: they can handle any other request made to the server over other protocols. Because of this, the Servlet interface does not have the doGet() or doPost() methods, but only the general service() method. This interface also defines the init() and destroy() methods, which can be overridden to perform initialise resources used by the Servlet, and the release them when the Servlet is destroyed.

- ServletRequest – An interface which defines the methods for all objects encapsulating information about requests made to the server.
- ServletResponse – An interface defining the methods for all objects which return a response from the server.
- ServletConfig – Defines an interface used to gain access to configuration parameters which are passed to the Servlet during initialisation.

Classes:

- GenericServlet – This is a generic class implementing Servlet. If you wish to write Servlet's for protocols other than HTTP, the easiest way of doing so is to extend GenericServlet rather than by directly implementing the Servlet interface.
- ServletException – An exception which can be thrown when the Servlet encounters a problem of some kind.
-

17.10.2 javax.servlet.http

The javax.servlet.http package provides classes specific to handling HTTP requests. It provides the HttpServlet class used in this chapter, which implements the appropriate interfaces from javax.servlet.

Interfaces:

- HttpServletRequest – An extension to the ServletRequest interface for features specific to HTTP.
- HttpServletResponse – An extension to the ServletResponse interface for features specific to HTTP.
- HttpSession – Provides access to the session tracking API.

Classes:

- HttpServlet – An abstract class providing functionality to implement HTTP requests. Note that the service() method defined in the Servlet interface will now call doGet() and doPost(), which can each be implemented to provide behaviour to the Servlet.
- Cookie – The cookie class, which provides an interface for storing small portions of data on the user's computer.
- HttpServletRequestWrapper and HttpServletResponseWrapper – Provides an implementation of the HttpServletResponse and HttpServletRequest interfaces.

17.11 Servlet Life-cycle

Now that we've covered some examples and seen the interfaces and classes which make up the Servlet API, we can discuss the life-cycle of a Servlet. The Servlet life-cycle consists of the steps through which Web server places a Servlet in order to satisfy a request for a resource implemented by a Servlet. This discussion is fairly general, and applies to all Servlets, not just those extending the HttpServlet class.

When a request for a resource implemented by a Servlet is made, the Web server does the following:

- If no instance of the particular Servlet class exists, the Web server will:
- Find the Servlet's .class file and load it.
- Instantiate the class.
- Call the classes init() method. init() is defined in the Servlet interface, so all Servlets must implement it. The method is guaranteed to be called before the Servlet handles any requests, and to only be called once. It is useful for initialising any resources the Servlet requires when handling requests, such as database connections, and so on.
- Call the Servlet's service() method. As previously mentioned, the service() method is called to handle all requests made to the Servlet. The HttpServlet class implements service() to call doGet() or doPost(), as appropriate.

When a Web server no longer requires a Servlet, it calls the Servlet's destroy() method. This is another method defined by the Servlet interface, and can be used to release any resources which the Servlet has acquired during its execution (for example, any database connections that it created when init() was called).

This Servlet life-cycle gives Servlet-based Web programmes a number of advantages over CGI programmes, which were briefly mentioned earlier in this chapter. These advantages are:

- CGI programmes are loaded in to memory whenever a new request is made for a resource implemented by the programme. When the request is completed, the programme is unloaded. Servlets always remain in memory, and so are typically faster to execute.
- Servlets share the same process as the Web server, while CGI programmes always run in their own processes. This point, along with point one above, means that Servlets remove the overhead of continuously creating new processes.
- Only a single instance of a given Servlet exists, and it answers all the requests made for a particular resource. This means that the Web server requires less memory to run, as the equivalent CGI programme will be loaded into memory multiple times in order to handle simultaneous requests.
- Just like JavaScript in a Web browser, Servlets can be run in a sandbox, giving the Servlet only restricted access to system resources, and so protecting the the machine on which the Web server is running from malicious Servlets.

We can now rewrite the above counter example to make use of the Servlet life-cycle The Counter class now looks as follows:

```
import java.io.*; import javax.servlet.*;
import javax.servlet.http.*;

public class Counter extends HttpServlet {

    private final String file_name = "counts.txt"; private
    final String attribute_name = "visited"; private final
    String attribute_value = "returnSoon";

    private int count = 0;

    public void init(ServletConfig config)
    {
 BufferedReader file = null; try {
     file = new BufferedReader(new FileReader(file_name));
     count = Integer.parseInt(file.readLine());
 }
```

```
catch(IOException e) {
}
```

```
                        finally {
    try {
 if (file != null) file.close();
    }
    catch (IOException b) {}
}
   }

   public void destroy()
   {
PrintWriter writer = null; try  {
    writer = new PrintWriter(new FileWriter(file_name));
    writer.println(count);
    writer.close(); writer = null;
}
catch (IOException e) {
}
finally {
    if (writer != null) writer.close();
}
   }

   public void doGet(HttpServletRequest request,
       HttpServletResponse response)
   {
HttpSession session = request.getSession(); String value
= null;

value = (String) session.getAttribute(attribute_name); if

(value == null) {
    session.setAttribute(attribute_name,
    attribute_value);
    session.setMaxInactiveInterval(60 * 60 * 24);

    synchronized (this) { count += 1;
    }
}

try {
    PrintWriter out = response.getWriter();
    out.println("<html>");
    out.println("\t<head><title>Hello, visitor number " +
  count + ".</title></head>");
    out.println("<body>");
    out.println("<p>Hello, visitor number " + count +
    ".</p>"); out.println("</body>");
    out.println("</html>");
}
catch (IOException e) {
}
   }
}
```

There are some important changes to notice: The previous versions of the Counter class read the counter.txt file when every request was made. This version of the class uses the init() method to read the file, and the destroy() method to overwrite the file. Recall that the init() and destroy() methods are called exactly once, which means that there is no need to synchronise any of the file access operations, as we have previously done. While all the code placed here in the init() method could be placed in the constructor, initialisation code placed in init() has certain advantages: first, it has access to the ServletConfig object, which can be used to provide configuration parameters to the Servlet. For instance, it could conceivably have contained the name of a file to store the counter information in. These configuration parameters can be set in the web.xml file. Second, ServletException objects cannot be thrown from within a constructor, but can be thrown from within init().

The doGet() method is called multiple times, whenever a service request is made. As with previous examples, doGet() may be called multiple times concurrently. The only area in the code accessing and overwriting shared data occurs where the count variable is incremented. This line is wrapped in a synchronisation statement to prevent two separate calls to doGet from interfering with each other during. It is possible that a second call to doGet could increment the count variable just before the variable is printed, but this will not cause the Servlet to misbehave.