Java Servlets

I have presented a Java servlet example before to give you a sense of what a servlet looks like. From the example, you can get to know a servlet has methods like `doGet()`, `doPost()`, etc. to deal with different kinds of HTTP requests. But HTTP is not the only thing that servlets can do about. Actually they may communicate with other networking components based on various protocols. This note discusses the properties of generic servlets.

1 Servlet Life Cycle

To better understand the behavior of servlets, let's take a look at the life cycle of servlets.

A servlet is basically a small Java program that runs within a Web server. It can receive requests from clients and return responses. The whole life cycle of a servlet breaks up into 3 phases:

- Initialization: A servlet is first loaded and *initialized* usually when it is requested by the corresponding clients. Some websites allow the users to load and initialize servlets when the server is started up so that the first request will get responded more quickly.
- Service: After initialization, the servlets serve clients on request, implementing the application logic of the web application they belong to.
- Destruction: When all pending requests are processed and the servlets have been idle for a specific amount of time, they may be destroyed by the server and release all the resources they occupy.

More specifically, the behavior of a servlet is described in `javax.servlet.Servlet` interface, in which the following methods are defined:

- `public void init(ServletConfig config) throws ServletException`

  This method is called once when the servlet is loaded into the servlet engine, before the servlet is asked to process its first request.

  The `init` method has a `ServletConfig` parameter. The servlet can read its initialization arguments through the `ServletConfig` object. How the initialization arguments are set is servlet engine dependent but they are usually defined in a configuration file.

A typical example of an initialization argument is a database identifier. A servlet can read this argument from the `ServletConfig` at initialization and then use it later to open a connection to the database during processing of a request:

```
private String databaseURL;

public void init(ServletConfig config) throws ServletException {
  super.init(config);
  databaseURL = config.getInitParameter("database");
}
```

- `public void service(ServletRequest request, ServletResponse response) throws ServletException, IOException`

  This method is called to process a request. It can be called zero, one or many times until the servlet is unloaded.

  Once a servlet is loaded, it remains in the server's memory as a single object instance. Thereafter, the server invokes the servlet to handle a request using a simple, lightweight method invocation. Unlike with CGI, there's no process to spawn or interpreter to invoke, so the servlet can begin handling the request almost immediately. Multiple, concurrent requests are handled by separate threads, so servlets are highly scalable.

  Servlets are naturally enduring objects. Because a servlet stays in the server's memory as a single object instance, it automatically maintains its state and can hold on to external resources, such as database connections, that may otherwise take several seconds to establish. The following servlet presents information about how many times it has been accessed:

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

public class SimpleCounter extends HttpServlet {

  int count = 0;

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                             throws ServletException, IOException {
    res.setContentType("text/plain");
    PrintWriter out = res.getWriter();
    count++;
    out.println("Since loading, this servlet has been accessed " +
                count + " times.");
  }
}
```

The variable `count` is shared by all the threads each corresponding to a single request. So this provides a way for the threads to communicate with each other.

Unfortunately the concurrency that multiple threads (one per request) can execute this method in parallel also brings problems. Imagine that one thread increments the count and just afterward, before the first thread prints the count, the second thread also increments the count. Each thread will print the same count value, after effectively increasing its value by 2. The order of execution goes something like this:

```
count++            // Thread 1
count++            // Thread 2
out.println        // Thread 1
out.println        // Thread 2
```

Now, in this case, the inconsistency is not a real problem, but many other servlets have more serious opportunities for errors. To prevent these types of problems and the inconsistencies that come with them, we can add one or more synchronized blocks to the code. Anything inside a synchronized block or a synchronized method is guaranteed not to be executed concurrently by another thread. Before any thread begins to execute synchronized code, it must obtain a monitor (lock) on a specified object instance. If another thread already has that monitor – because it is already executing the same synchronized block or some other block with the same monitor – the first thread must wait. For example, we may wrap the addition and print operations in a synchronized blocks as follows:

```
PrintWriter out = res.getWriter();
synchronized(this) {
  count++;
  out.println("Since loading, this servlet has been accessed " +
              count + " times.");
}
```

The `javax.servlet.SingleThreadModel` interface provides another approach to avoid race condition.

If a servlet implements this interface, you are guaranteed that no two threads will execute concurrently in the servlet's `service` method. The servlet container can make this guarantee by synchronizing access to a single instance of the servlet, or by maintaining a pool of servlet instances and dispatching each new request to a free servlet.

If a servlet implements this interface, the servlet will be thread safe. However, this interface does not prevent synchronization problems that result from servlets accessing shared resources such as static class variables or classes outside the scope of the servlet.

- `public void destroy()`

  This method is called once just before the servlet is unloaded and taken out of service. The following gives an servlet example with both `init()` and `destroy()` methods:

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

public class InitDestroyCounter extends HttpServlet {

  int count;

  public void init() throws ServletException {
    // Try to load the initial count from our saved persistent state
    FileReader fileReader = null;
    BufferedReader bufferedReader = null;
    try {
      fileReader = new FileReader("InitDestroyCounter.initial");
      bufferedReader = new BufferedReader(fileReader);
      String initial = bufferedReader.readLine();
      count = Integer.parseInt(initial);
      return;
    }
    catch (FileNotFoundException ignored) { }  // no saved state
    catch (IOException ignored) { }            // problem during read
    catch (NumberFormatException ignored) { }  // corrupt saved state
    finally {
      // Make sure to close the file
      try {
        if (bufferedReader != null) {
          bufferedReader.close();
        }
      } catch (IOException ignored) { }
    }

    // No luck with the saved state, check for an init parameter
    String initial = getInitParameter("initial");
    try {
      count = Integer.parseInt(initial);
      return;
    } catch (NumberFormatException ignored) {
    }  // null or non-integer value

    // Default to an initial count of "0"
    count = 0;
  }

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                             throws ServletException, IOException {
    //...
  }

  public void destroy() {
    super.destroy();  // entirely optional
    saveState();
  }

  public void saveState() {
    // Try to save the accumulated count
    FileWriter fileWriter = null;
    PrintWriter printWriter = null;
    try {
      fileWriter = new FileWriter("InitDestroyCounter.initial");
      printWriter = new PrintWriter(fileWriter);
      printWriter.println(count);
      return;
    } catch (IOException e) {  // problem during write
      // Log the exception.
    }
    finally {
      // Make sure to close the file
      if (printWriter != null) {
        printWriter.close();
      }
    }
  }
}
```

Each time this servlet is unloaded, it saves its state in a file named `InitDestroyCounter.initial`. In the absence of a supplied path, the file is saved in the server process's current directory, usually the startup directory.

2 Servlet Context

All servlets belong to one servlet context. The `javax.servlet.ServletContext` interface in the Java Servlet API is responsible for the state of its servlets and knows about resources and attributes available to the servlets in the context. Here we will only look at how `ServletContext` attributes can be used to share information among a group of servlets.

There are three `ServletContext` methods dealing with context attributes: `getAttribute`, `setAttribute` and `removeAttribute`. In addition the servlet engine may provide ways to configure a servlet context with initial attribute values. This serves as a welcome addition to the servlet initialization arguments for configuration information used by a group of servlets, for instance a database identifier , a style sheet URL for an application, the name of a mail server, etc.

A servlet gets a reference to its `ServletContext` object through the `ServletConfig` object. The `HttpServlet` class actually provides a convenience method (through its superclass `GenericServlet`) named `getServletContext` to make it really easy:

```
...
ServletContext context = getServletContext();
String styleSheet = request.getParameter("stylesheet");
if (styleSheet != null) {
  // Specify a new style sheet for the application
  context.setAttribute("stylesheet", styleSheet);
}
...
```

The code above could be part of an application configuration servlet, processing the request from an HTML FORM where a new style sheet can be specified for the application. All servlets in the application that generate HTML can then use the style sheet attribute like this:

```
...
ServletContext context = getServletContext();
String styleSheet = context.getAttribute("stylesheet");
out.println("<HTML><HEAD>");
out.println("<LINK HREF=" + styleSheet + " TYPE=text/css REL=STYLESHEET>");
...
```

3 Handling Cookies

Cookies are small bits of textual information that a Web server sends to a browser and that the browser returns unchanged when visiting the same Web site or domain later. By having the server read information it sent the client previously, the site can provide visitors with a number of conveniences:

- *Identifying a user during an e-commerce session.* Many on-line stores use a ”shopping cart” metaphor in which the user selects an item, adds it to his shopping cart, then continues shopping. Since the HTTP connection is closed after each page is sent, when the user selects a new item for his cart, how does the store know that he is the same user that put the previous item in his cart? Cookies are a good way of accomplishing this. In fact, this is so useful that servlets have an API specifically for this, and servlet authors don’t need to manipulate cookies directly to make use of it. This is discussed later in section 4.

- *Avoiding username and password.* Many large sites require you to register in order to use their services, but it is inconvenient to remember the username and password. Cookies are a good alternative for low-security sites. When a user registers, a cookie is sent with a unique user ID. When the client reconnects at a later date, the user ID is returned, the server looks it up, determines it belongs to a registered user, and doesn’t require an explicit username and password.

- *Customizing a site.* Many portal sites let you customize the look of the main page. They use cookies to remember what you wanted, so that you get that result initially next time.

3.1 The Servlet Cookie API

To send cookies to the client, a servlet would create one or more cookies with the appropriate names and values via `new Cookie(name, value)`, set any desired optional attributes via `cookie.setXxx`, and add the cookies to the response headers via `response.addCookie(cookie)`. To read incoming cookies, call `request.getCookies()`, which returns an array of `Cookie` objects. In most cases, you loop down this array until you find the one whose name (`getName`) matches the name you have in mind, then call `getValue` on that `Cookie` to see the value associated with that name.

3.1.1 Creating Cookies

A `Cookie` is created by calling the `Cookie` constructor, which takes two strings: the cookie name and the cookie value. Neither the name nor the value should contain whitespace or any of:

```
[ ] ( ) = , " / ? @ : ;
```

3.1.2 Reading and Specifying Cookie Attributes

Before adding the cookie to the outgoing headers, you can look up or set attributes of the cookie. Here's a summary:

- `getComment/setComment`

  Gets/sets a comment associated with this cookie.

- `getDomain/setDomain`

  Gets/sets the domain to which cookie applies. Normally, cookies are returned only to the exact host name that sent them. You can use this method to instruct the browser to return them to other hosts within the same domain.

- `getMaxAge/setMaxAge`

  Gets/sets how much time (in seconds) should elapse before the cookie expires. If you don't set this, the cookie will last only for the current session (i.e. until the user quits the browser), and will not be stored on disk. See the LongLivedCookie class below, which defines a subclass of Cookie with a maximum age automatically set one year in the future.

  `getName/setName`

  Gets/sets the name of the cookie. The name and the value are the two pieces you virtually always care about. Since the `getCookies` method of `HttpServletRequest` returns an array of `Cookie` objects, it is common to loop down this array until you have a particular name, then check the value with `getValue`. See the `getCookieValue` method shown below.

- `getPath/setPath`

  Gets/sets the path to which this cookie applies. If you don't specify a path, the cookie is returned for all URLs in the same directory as the current page as well as all subdirectories. This method can be used to specify something more general. For example, `someCookie.setPath("/")` specifies that all pages on the server should receive the cookie. Note that the path specified must include the current directory.

- `getSecure/setSecure`

  Gets/sets the boolean value indicating whether the cookie should only be sent over encrypted (i.e. SSL) connections.

- `getValue/setValue`

  Gets/sets the value associated with the cookie. Again, the name and the value are the two parts of a cookie that you almost always care about, although in a few cases a name is used as a `boolean` flag, and its value is ignored (i.e the existence of the name means true).

- `getVersion/setVersion`

  Gets/sets the cookie protocol version this cookie complies with. Version 0, the default, adheres to the original Netscape specification. Version 1, not yet widely supported, adheres to RFC 2109.

3.1.3 Placing Cookies in the Response Headers

The cookie is added to the `Set-Cookie` response header by means of the `addCookie` method of `HttpServletResponse`. Here's an example:

```
Cookie userCookie = new Cookie("user", "uid1234");
response.addCookie(userCookie);
```

3.1.4 Reading Cookies from the Client

To send cookies to the client, you created a `Cookie` then used `addCookie` to send a `Set-Cookie` HTTP response header. To read the cookies that come back from the client, you call `getCookies` on the `HttpServletRequest`. This returns an array of Cookie objects corresponding to the values that came in on the `Cookie` HTTP request header. Once you have this array, you typically loop down it, calling `getName` on each `Cookie` until you find one matching the name you have in mind. You then call `getValue` on the matching `Cookie`, doing some processing specific to the resultant value. This is such a common process that the following section presents a simple `getCookieValue` method that, given the array of cookies, a name, and a default value, returns the value of the cookie matching the name, or, if there is no such cookie, the designated default value.

3.2 Some Minor Cookie Utilities

Here are some simple but useful utilities for dealing with cookies.

3.2.1 Getting the Value of a Cookie with a Specified Name

The following example shows how to retrieve a cookie value given a cookie name by looping through the array of available `Cookie` objects, returning the value of any `Cookie` whose name matches the input. If there is no match, the designated default value is returned.

```
public static String getCookieValue(Cookie[] cookies,
                                    String cookieName,
                                    String defaultValue) {
  for(int i=0; i<cookies.length; i++) {
    Cookie cookie = cookies[i];
    if (cookieName.equals(cookie.getName()))
      return(cookie.getValue());
  }
  return(defaultValue);
}
```

3.2.2 `LongLivedCookie.java`

Here's a small class that you can use instead of `Cookie` if you want your cookie to automatically persist when the client quits the browser.

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

public class LongLivedCookie extends Cookie {
  public static final int SECONDS_PER_YEAR = 60*60*24*365;

  public LongLivedCookie(String name, String value) {
    super(name, value);
    setMaxAge(SECONDS_PER_YEAR);
  }
}
```

4 Session Tracking

An `HttpSession` class was introduced in the Servlet API. Instances of this class can hold information for one user session between requests. You start a new session by requesting an `HttpSession` object from the `HttpServletRequest` in your `doGet` or `doPost` method:

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

This method takes a boolean argument. `true` means a new session shall be started if none exist, while `false` only returns an existing session. The `HttpSession` object is unique for one user session. The Servlet API supports two ways to associate multiple requests with a session: cookies and URL rewriting. If cookies are used, a cookie with a unique session ID is sent to the client when the session is established. The client then includes the cookie in all subsequent requests so the servlet engine can figure out which session the request is associated with. URL rewriting is intended for clients that don't support cookies or when the user has disabled cookies. With URL rewriting the session ID is encoded in the URLs your servlet sends to the client. When the user clicks on an encoded URL, the session ID is sent to the server where it can be extracted and the request associated with the correct session as above. To use URL rewriting you must make sure all URLs that you send to the client are encoded with the `encodeURL` or `encodeRedirectURL` methods in `HttpServletResponse`.

An `HttpSession` can store any type of object. A typical example is a database connection allowing multiple requests to be part of the same database transaction, or information about purchased products in a shopping cart application so the user can add items to the cart while browsing through the site. To save an object in an `HttpSession` you use the `putValue` method:

```
...
Connection con = driver.getConnection(databaseURL, user, password);
session.putValue("myappl.connection", con);
...
```

In another servlet, or the same servlet processing another request, you can get the object with the `getValue` method:

```
...
HttpSession session = request.getSession(true);
Connection con = (Connection) session.getValue("myappl.connection");
if (con != null) {
  // Continue the database transaction
...
```

You can explicitly terminate (invalidate) a session with the invalidate method or let it be timed-out by the servlet engine. The session times out if no request associated with the session is received within a specified interval. Most servlet engines allow you to specify the length of the interval through a configuration option. In the Servlet API there's also a `setMaxInactiveInterval` so you can adjust the interval to meet the needs of each individual application.

5 GenericServlet

6 HttpServlet