Spring, 2009   CIS 455 / 555

Internet and Web Systems

Assignment 1: Web and Application Servers

Milestone 1 due February 3, 2009
Milestone 2 due February 17, 2009

Background

We are all familiar with how one accesses a Web server via a browser. The big question is what is going on under the covers of the Web server: how does it serve data, what is necessary in order to provide the notion of sessions, how is it extended, and so on.

This assignment focuses on developing an *application server*, i.e., a Web (HTTP) server that runs Java servlets, in two stages. In the first stage, you will implement a simple HTTP server for static content (i.e., files like images, style sheets, and HTML pages).

In the second stage, you will expand this work to emulate a full-fledged application server that runs servlets. Java servlets are a popular method for writing *dynamic* Web applications. They provide a cleaner and much more powerful interface to the Web server and Web browser than previous methods, such as CGI scripts.

If you have taken CIS 330 or 550, you should already be familiar with servlet programming; if you have not, it should not be too difficult to catch up. A Java servlet is simply a Java class that extends the class `HttpServlet`. It typically overrides the *doGet* and *doPost* methods from that class to generate a web page in response to a request from a Web browser. An XML file, `web.xml`, lets the servlet developer specify a mapping from URLs to class names; this is how the server knows which class to invoke in response to an HTTP request. Further details about servlets, including a tutorial and API reference, as well as sample servlets and a corresponding `web.xml` file, are available on the course web site on the Assignments page. We have also given you code to parse `web.xml`.

You can also find a more comprehensive description of servlets at:
`http://www.novocode.com/doc/servlet-essentials/`
and the servlet API 2.4 JavaDoc at:
`http://tomcat.apache.org/tomcat-5.5-doc/servletapi/index.html`

Developing and running your code

For development and testing, you should ssh to the machines `mod1.seas.upenn.edu` through `mod24.seas.upenn.edu`[1], from which you should have access to your normal (eniac) home directory. The `mod` machines are dedicated to distributed systems courses, and they all have ports 8080+ unblocked. One should be able to open a Web server at one of those ports, and to access the Web server from any browser via:
`http://mod1.seas.upenn.edu:8080`
(or whatever your hostname may be, and what port your Web server uses). Note that, because these machines are shared, you may need to change from port 8080 to 8081, etc. simply because someone else is using the other port!

[1]Note that not all machines may be functional on any given day. If you can't get to one of the machines, simply use another.

While you can use any Java IDE you like (or none at all), we recommend that you develop using Eclipse. In Eclipse, you will need to create a new project, copy in the supplied source files, and add the provided file `servlet-api.jar` as an "External JAR" under Project properties → Java build path → Libraries.

We will expect you to turn in your project by giving us access to a subdirectory in your home directory on the `mod` servers (note that this is actually the same as your eniac directory). This code should run directly from that directory, so we don't need to do any special setup to make it execute. You will need to coordinate with Mengmeng to ensure that she has sufficient access permissions to view your homework files.

Milestone 1: due February 3

For the first milestone, your task is relatively simple. You will develop a Web server that can be invoked from the command-line, taking the following parameters, in this order:

1. Port to listen for connections on. Port 80, the default HTTP port, is generally blocked by the SEAS firewall (at least from outside Penn). You should probably use the conventional secondary port, 8080, or perhaps 8081, etc.
2. Root directory of the static web pages. For example, if this is set to the directory `/website`, a request for `/mysite/index.html` will return the file `/website/mysite/index.html`.

Your program will accept incoming GET requests from a Web browser, and it will make use of a *thread pool* (as discussed in class) to invoke a worker thread to process each request. The worker thread will parse the HTTP request, determine which file was requested (relative to the root directory specified above) and return the file. If a directory was requested, the request should return a listing of the files in the directory.

If a GET request is made that is not a valid UNIX path specification, you should return the appropriate HTTP error. If no file is found, you should return the appropriate HTTP error. See the *HTTP Made Really Easy* paper for more details.

MAJOR SECURITY CONCERN: you should make sure that users are not allowed to request *absolute* paths or paths outside the root directory. We will validate, e.g., that we cannot get hold of /etc/passwd!

HTTP protocol

Your application server must be fully HTTP 1.1 compliant, as described in the document *HTTP Made Really Easy* given out in class; this means that it must be able to support HTTP 1.0 clients as well as 1.1 clients. It must also support persistent connections, which is mentioned as being optional for HTTP 1.1 servers.

Implementation techniques

For efficiency, your application server must be implemented using a thread pool, as discussed in class. Specifically, there should be one thread that runs the interactive menu, one to listen for incoming TCP requests and enqueue, and some number of threads that process the requests from the queue and return the responses. We will examine your code to make sure it is free of race conditions and the potential for deadlock, so code carefully! We expect you to write your own thread pool code, not use one from the Java system library or an external library.

Milestone 2: due February 17

The second milestone will build upon the Web server from Milestone 1, with support for POST and for invoking servlet code. To ease implementation, your application server will need to support only one web application at a time. Therefore, you can simply add the class files for the web application to the classpath when you invoke you application server from the command line, and pass the location of the `web.xml` file as an argument. Furthermore, you need not implement all of the methods in the various servlet classes; details as to what is required may be found below.

Invocation of the application server

You should add a third command-line argument: the location of the `web.xml` file for your web application.

You may accept additional optional arguments after the initial three (such as number of worker threads, for example), but the application should run with reasonable defaults if they are omitted. In addition, the application server must present (using the console) some sort of interactive menu. This must at least provide a way to shutdown the application server (after calling each servlet's `destroy` method, of course!), view the error log, and see the status of each thread in the pool. It may provide other (e.g., extra-credit) features as you see fit.

Implementation techniques

Dynamic loading of classes in Java — which you will need to do since a servlet can have any arbitrary name, as specified in `web.xml` — can be a bit tricky. Start by calling the method `Class.forName`, with the string name of the class as an argument, to get a `Class` object representing the class you want to instantiate (i.e. a specific servlet). Since your servlets do not define a constructor, you can then call the method `newInstance` on that Class object, and typecast it to an instance of your servlet. Now you can call methods on this instance.

Required application server features

Your application server must provide functional implementations of all of the non-deprecated methods in the interfaces `HttpServletRequest`, `HttpServletResponse`, `ServletConfig`, `ServletContext`, and `HttpSession` of the Servlet interface version 2.4 (see the URL on the first page of this assignment), with the following exceptions:

- `HttpServlet.getUserPrincipal`
- `HttpServlet.isUserInRole`
- `HttpServletRequest.getRequestDispatcher`
- `HttpServletRequest.getInputStream`
- `HttpServletResponse.getOutputStream`
- `ServletContext.getNamedDispatcher`
- `ServletContext.getRequestDispatcher`

This means that your application server will need to support cookies, sessions (using cookies — you don't need to provide a fall-back like path encoding if the client doesn't support cookies), servlet contexts, initialization parameters (from the web.xml file); in other words, all of the infrastructure needed to write real servlets. It also means that you won't need to do HTTP-based authentication, or implement the `ServletInputStream` and `ServletOutputStream` classes.

We suggest you start by determining *what* you need to implement:

1. Print the JavaDocs for `HttpServletRequest`, `HttpServletResponse`, `ServletConfig`, `ServletContext`, and `HttpSession`, from the URL given previously.

2. Create a skeleton class for each of the above, with methods that temporarily return null for each call. Be sure that your `HttpServletRequest` class inherits from the provided `javax.servlet.HttpServletRequest` (in the jar file), and so forth.

3. Print the sample `web.xml` from the `Servlets/web/WEB-INF` directory in the `Servlets.tgz` file. There is very useful information in the comments, which will help you determine where certain methods get their data.

You can find a simple parser for the `web.xml` file from the TestHarness code provided to you (see the last page of this handout). For the `ServletConfig` and `ServletContext`, note the following:

- There is a single ServletContext per "Web application," and a single ServletConfig per "servlet page." (For the base version of Milestone 2, you will only need to run one application at a time.) Assuming a single application will likely simplify some of what you need to implement in ServletContext (e.g., getServletNames).

- Most of the important ServletConfig info — servlet name, init parameter names, and init parameter list — come directly from `web.xml`. Note that the init parameters for ServletConfig come from `init-param` elements within the `servlet` element.

- The ServletContext init parameters come from the `context-param` elements within `web.xml`.

- The ServletContext attributes are essentially a hash map from name to value, and can be used, e.g., to communicate between multiple instances of the same servlet. By default, these can only be created programmatically by servlets themselves, unlike the initialization parameters, which are set in `web.xml`.

- The real path of a file can be getting the canonical path of the path relative to the Web root. It is straightforward to return a stream to such a resource, as well. The URL to a relative path can similarly be generated relative to the Servlet's URL.

- The ServletContext name is set to the display name specified in `web.xml`.

- You can simply return null for all deprecated methods.

Resources

We have provided you with a JAR file containing version 2.4 of the servlet API. You have also been given the source code for a simple application server that accepts requests from the command line, calls a servlet, and prints results back out. It will give you a starting point, though many of the methods are just stubs, which you will need to implement.

We have also provided a suite of simple test servlets and an associated `web.xml` file and directory of static content; it should put your application server through its paces. We will, however, test your application server with additional servlets.

Extra credit

Multiple applications and dynamic loading (+25%)

The project described above loads one web application and installs it at the root context. Extend it to dynamically load and unload other applications at different contexts. Add options to the main menu of the server to list installed applications, install new applications, and remove an installed applications. You'll need to take special care to ensure that static variables do not get shared between applications (i.e. the same class in two different applications can have different values for the same static variable). Each application should have its own servlet context as well. (Since each application may have its own classpath, be sure to add the capability to dynamically modify the classpath, too.)

Performance testing (+10%)

The supplied servlet `BusyServlet` performs a computationally intestive task that should take a number of seconds to perform on a modern computer. Experimentally determine the effect of changing the thread pool size on performance of the application server when many requests for `BusyServlet` come in at the same time. Comment on any trends you see, and try to explain them. Suggest the ideal thread pool size and describe how you chose it. Include performance measures like tables, graphs, etc.

TestHarness: A Primitive App Server

TestHarness gives you a simple command-line interface to your servlets. It reads your `web.xml` file to find out about servlets. Thus, in order to test a servlet you need to add the appropriate entry in `web.xml` first (as you would do in order to deploy it). You can then specify a series of requests to servlets on the command line, which all get executed within a servlet session.

Suppose you have a servlet which you've given the name 'demo' in your `web.xml` file.

To run this servlet:

1. Put the TestHarness classes and the servlet code in the same eclipse project.

2. Make sure the file servlet-api.jar has been added to the project as an 'external jar file.'

3. Create a new run profile (Run → Run...), choose TestHarness as the main class, and give the command line arguments `path/to/web.xml GET demo` to have the testHarness program run the demo servlet's doGet method. The servlet output is printed to the screen as unprocessed HTML. You can set the profile's root directory if it makes writing the path to the `web.xml` easier; it defaults to the root of the Eclipse project.

More interestingly, if you had a servlet called login, you could also run it with the arguments:

```
path/to/web.xml POST login?login=aaa&passwd=bbb
```

This will call the doPost method with the parameters login and passwd passed as if the servlet was invoked through tomcat.

Finally, TestHarness also supports sequences of servlets while maintaining session information that is passed between them. Suppose you had a servlet called listFeeds, which a used can run only after logging in. You can simulate this with the harness by doing:

```
path/to/web.xml POST login?login=aaa&passwd=bbb GET listFeeds
```

In general, since your servlets would normally expect to be passed the session object when executed, in order to test them with this harness you should simulate the steps that would be followed to get from the login page to that servlet. If for example after login you go to formA and enter some values and click a button to submit formA to servletA, and then you enter some more values in formB and click a button and go to servletB, to test servletB (assuming you use post everywhere) you would do:

```
path/to/web.xml POST login?login=aaa&passwd=bbb POST servletA?...attributes-values
of formA...  POST servletB?...attributes-values of formB...
```