CIS 455/555: Internet and Web Systems

Spring 2013

Assignment 1: Web and Application Server

Milestone 1 due February 6, 2013, at 10:00pm
Milestone 2 due February 20, 2013, at 10:00pm

1. Implement a simple HTTP server for static content(images, style sheets, and HTML pages)
2. Expand this -> emulate a full-fledged application server that runs servlets.

1 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 links to tutorials and an 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`.

2 Developing and running your code

We strongly recommend that you do the following before you start writing code:

1. Carefully read the entire assignment (both milestones) from front to back and make a list of the features you need to implement.
2. Think about how the key features will work. For instance, before you start with MS2, go through the steps the server will need to perform to handle a request. If you still have questions, have a look at some of the extra material on the assignments page, or ask one of us during office hours.
3. Spend at least some time thinking about the *design* of your solution. What classes will you need? How many threads will there be? What will their interfaces look like? Which data structures need synchronization? And so on.
4. Regularly check your changes into the subversion repository. This will give you many useful features, including a recent backup and the ability to roll back any changes that have mysteriously broken your code.

We recommend that you continue using the VM image we have provided for HW0. This image should already contain all the tools you will need for HW1. You can check out the framework code for HW1 using the same process as for HW0 (there should now be an additonal "HW1" folder).

Of course, you are free to use any other Java IDE you like, or no IDE at all, and you do not have to use any of the tools we provide. However, to ensure efficient grading, your submission **must** meet the requirements specified in 3.4 and 4.7 below - in particular, it must build and run correctly in the original VM image and have an `ant` build script.

2.1 Testing your server

To test your server, you have several options:

- You can use the Web Console in Firefox to inspect the HTTP headers. Open the "Tools" menu, choose "Web Developers", and click on "Web Console". This should pop up a new window, which will list all the HTTP requests processed by Firefox (click on a request for extra details).
- If you want to check whether you are using the correct headers, you may find the site web-sniffer.net useful.
- You may also want to consider using the `curl` command-line utility to do some automated testing of your server. `curl` makes it easy to test HTTP/1.1 compliance by sending HTTP requests that are purposefully invalid - e.g., sending an HTTP/1.1 request without a `Host` header. '`man curl`' lists a great many flags.
- To stress-test your server, you can use Apachebench (the `ab` command, which is already pre-installed in the VM). Apachebench can be configured to make many requests concurrently, which will help you find concurrency problems, deadlocks, etc.

3 Milestone 1: Multithreaded HTTP Server (due February 6)

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 is the default HTTP port, but it is often blocked by firewalls, so your server should be able to run on any other port (e.g., 8080)

2. *Root directory of the static web pages*. For example, if this is set to the directory `/var/www`, a request for `/mydir/index.html` will return the file `/var/www/mydir/index.html`. (do *not* hard-code any part of the path in your code - your server needs to work on a different machine, which may have completely different directories!)

Note that the second milestone will add a third argument (see below). If your server is invoked without any command-line arguments, it must output your full name and SEAS login name.

Your program will accept incoming `GET` and `HEAD` requests from a Web browser (such as the Firefox browser in the VM image), 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. Your server should return the correct MIME types for some basic file formats, based on the extension (.jpg, .gif, .png, .txt, .html); keep in mind that image files must be sent in binary form -- not with println or equivalent -- otherwise the browser will not be able to read them.

If a `GET` or `HEAD` request is made that is not a valid UNIX path specification, if no file is found, or if the file is not accessible, 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 can't get hold of `/etc/passwd`!

3.1 HTTP protocol version and features

Your application server must be HTTP 1.1 compliant, as described in the document HTTP Made Really Easy used in class. This means that it must be able to support HTTP 1.0 clients as well as 1.1 clients. **Persistent connections** are suggested but not required for HTTP 1.1 servers. If you do not wish to support persistent connections, be sure to include "`Connection: close`" in the header of the response. **Chunked encoding** (sometimes called **chunking**) is also not required. Support for persistent connections and chunking is extra credit, described near the end of this assignment.

3.2 Special URLs

Your application server should implement two special URLs. If someone issues a `GET /shutdown`, your server should shut down immediately; however, any threads that are still busy handling requests must be aborted properly (do not just call `System.exit`!). If someone issues a `GET /control`, your server should return a 'control panel' web page, which must contain at least a) your full name and SEAS login, b) a list of all the threads in the thread pool, c) the status of each thread ('waiting' or the URL it is currently handling), and d) a button that shuts down the server, i.e., is linked to the special `/shutdown` URL. It must be possible to open the special URLs in a normal web browser.

3.3 Implementation techniques

For efficiency, your application server must be implemented using a thread pool that you implement, as discussed in class. Specifically, there should be one thread that listens for incoming TCP requests and enqueues them, 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. This includes the queue, which you should implement by yourself, using condition variables to block and wake up threads. You may not use the `BlockingQueue` that comes with Java, or any similar classes.

3.4 Requirements

Your solution must meet the following requirements (please read carefully!):

1. Your main class must be called `HttpServer`, and it must be located in a package called `edu.upenn.cis455.webserver`.
2. Your submission must contain a) the entire source code, as well as any supplementary files needed to build your solution, b) an `ant` build script called `build.xml` (a template is included with the code in svn), and c) a `README` file. The `README` file must contain 1) your full name and SEAS login name, 2) a description of features implemented, 3) any extra credit claimed, and 4) any special instructions for building or running.
3. When your submission is unpacked in the original VM image and the `ant` build script is run, your solution must compile correctly. Please test this before submitting!

4. Your server must accept the two command-line arguments specified above, and it must output your full name and SEAS login name when invoked without command-line arguments.
5. Your solution must be submitted via `turnin` (!) before 10:00pm on Feb 6, 2013 (MS1) and 10:00pm on Feb 20, 2013 (MS2), respectively. The only exception is if you have obtainend an extension online, using the link on the course web page. The project name for `turnin` should be `hw1ms1` for the first milestone, and `hw1ms2` for the second milestone.
6. Your code must contain a reasonable amount of useful documentation.

You may not use any third-party code other than the standard Java libraries (exceptions noted in the assignment) and any code we provide.

4 Milestone 2: Servlet Engine (due February 20)

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.

4.1 The Servlet

A servlet is typically stored in a special "war file" (extension `.war`) which is essentially a jar file with a special layout. The configuration information for a servlet is specified in a file called `web.xml`, which is typically in the `WEB-INF` directory. The servlet's actual classes are typically in `WEB-INF/classes`. The `web.xml` file contains information about the servlet class to be invoked, its name for the app server, and various parameters to be passed to the servlet. See below for an example:

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee" version="2.4">
  <servlet>
    <servlet-name>HelloWorld</servlet-name>
    <servlet-class>edu.upenn.HelloWorld</servlet-class>
    <init-param>
      <param-name>greeting</param-name>
      <param-value>Bonjour!</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>HelloWorld</servlet-name>
    <url-pattern>/Hello</url-pattern>
  </servlet-mapping>
  <context-param>
    <param-name>server</param-name>
    <param-value>my455server</param-value>
  </context-param>
</web-app>
```

The `servlet` and `servlet-class` elements are used to establish an internal name for your servlet, and which class it binds to. The servlet-mapping associates the servlet with a particular sub-URL (`http://my-server/HelloWorld` or something similar). There are two kinds of URL patterns we want to handle:

1. Exact pattern (must start with a /). This is the most common way of specifying a servlet.
2. Path mapping (starts with a / and ends with a *, meaning it should match on the prefix up to the *). This is used in a certain Web service scheme called "REST" (which we discuss later in the term).

There are two ways that parameters can be specified from "outside" the servlet, e.g., to describe setup information such as usernames and passwords, servlet environment info, etc. These are through `init-param` elements, which appear *within* `servlet` elements and establish name-value pairs for the servlet *configuration*, and the `context-param` elements, which establish name-value pairs for the servlet *context*. We will discuss how these are accessed programmatically in a moment.

4.2 Basic Servlet Operation

All servlets implement the `javax.servlet.http.HttpServlet` interface, which extends the Servlet interface. You will need to build the "wrapping" that invokes the `HttpServlet` instance, calling the appropriate functions and passing in the appropriate objects.

Servlet initialization, config, and context. When the servlet is first activated (by starting it in the app server), this calls the `init()` method, which is passed a `ServletConfig` object. This may request certain resources, open persistent connections, etc. The `ServletConfig` details information about the servlet setup, including its ServletContext. Both of these can be used to get parameters from `web.xml`.

`ServletConfig` represents the information a servlet knows about "itself". Calling `getInitParameter()` on the `ServletConfig` returns the servlet `init-param` parameters. The method `getParameterNames()` returns the full set of these parameters. Finally, one can get the servlet's name (from `web.xml`) through this interface. `ServletContext` represents what the servlet sees about its related Web application. Calling `getInitParameter()` on the `ServletContext` returns the servlet `context-param` parameters. The method `getParameterNames()` returns the full set of these parameters. Through the context, the servlet can also access resources that are within the `.war` file, and determine the real path for a given "virtual" path (i.e., a path relative to the servlet). Perhaps more important, the `ServletContext` provides a way of passing objects ("attributes" that are name-object pairs) among different parts of a Web application. You may ignore the context's logging capabilities.

Service request and response. When a request is made of the servlet by an HTTP client, the app server calls the `service()` method with a `javax.servlet.ServletRequest` parameter containing request info, and a `javax.servlet.ServletResponse` parameter for return info. For an HTTP servlet (the only kind we are implementing), `service()` typically calls a handler for the type of HTTP request. The only ones we care about are `doGet()` for `GET` requests, and `doPost()` for `POST` requests. (There are other kinds of calls, but these are seldom supported in practice.) Both `doGet()` and `doPost()` are given parameter objects implementing `javax.servlet.HttpServletRequest` and `javax.servlet.HttpServletResponse` (which are subclassed from the original `ServletRequest` and `ServletResponse`). `HttpServletRequest`, naturally, contains information about the HTTP request, including headers, parameters, etc. You can get header information from `getHeader()` and its related methods, and get form parameters through `getParameter()`. `HttpSession` is used to store state across servlet invocations. The `getAttribute()` and related methods support storing name-value pairs. The session should time-out after the designated amount of time (specified as a default or in `setMaxInactiveInterval()`). `HttpServletResponse` contains an object that is used to return information to the Web browser or HTTP client. The `getWriter()` or `getOutputStream()` methods provide a means of directly sending data that goes

onto the socket to the client. Also important are `addHeader()`, which adds a name-value pair for the response header, and its sibling methods for adding header information. Note that there are a variety of important fields you can set this way, e.g., server, content-length, refresh rate, content-type, etc. Note that you should ensure that an HTTP response code (e.g., "200 OK") is sent to the client before any output from the writer or output stream are returned. If the servlet throws an exception before sending output, you should return an error code such as "500 Internal Server Error". You should return a "302 Redirect" if the servlet calls `HttpServletResponse`'s `sendRedirect()` method.

Servlet shutdown. When the servlet is deactivated, this calls the servlet's `destroy()` method, which should release resources allocated by `init()`.

4.3 Invocation of the application server

You should add a third command-line argument: the location of the `web.xml` file for your web application. In your submission, this file should be located in the `conf` subdirectory. 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.

4.4 Special URLs

You should now augment the special URLs you implemented for Milestone 1. The `/shutdown` URL should properly shut down all the servlets, by invoking their `destroy` methods, and the `/control` URL should now provide a way to view the error log. It may provide other (e.g., extra-credit) features as you see fit.

4.5 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.

4.6 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:

- `ServletContext.log`
- `ServletContext.getMimeType` (return `null`)
- `ServletContext.getNamedDispatcher`
- `ServletContext.getResource`
- `ServletContext.getResourceAsStream`
- `ServletContext.getResourcePaths`
- `HttpServletRequest.getPathTranslated`
- `HttpServletRequest.getUserPrincipal`
- `HttpServletRequest.isUserInRole`
- `HttpServletRequest.getRequestDispatcher`
- `HttpServletRequest.getInputStream`

- `HttpServletResponse.getOutputStream`
- `HttpServletRequest.getLocales`
- `ServletContext.getNamedDispatcher`
- `ServletContext.getRequestDispatcher`

You can return `null` for the output of all of the above methods, as well as all deprecated methods.
We will also make the following simplifications and clarifications of the spec:

- `HttpRequest.getAuthType` should always return BASIC AUTH ("BASIC")
- `HttpRequest.getPathInfo` should always return the remainder of the URL request after the portion matched by the url-pattern in web-xml. It starts with a "/".
- `HttpRequest.getQueryString` should return the HTTP `GET` query string, i.e., the portion after the "?" when a `GET` form is posted.
- `HttpRequest.getCharacterEncoding` should return "ISO-8859-1" by default, and the results of `setCharacterEncoding` if it was previously called.
- `HttpRequest.getScheme` should return "http".
- `HttpResponse.getCharacterEncoding` should return "ISO-8859-1".
- `HttpResponse.getContentType` should return "`text/html`" by default, and the results of `setContentType` if it was previously called.
- `HttpServletRequest.getLocale` should return `null` by default, or the results of `setLocale` if it was previously called.

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 `extra/Servlets/web/WEB-INF` directory. 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 (see 5.1 and thecode in `extra/TestHarness`). 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`.

- The `ServletContext` init parameters come from the context-param elements in `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 `ServletContext` name is set to the display name specified 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.

4.7 Requirements

Your solution must meet the same requirements as MS1 (see 3.4 above), with two exceptions. First, your solution must now support three command-line arguments (the third is the location of `web.xml`), and second, you need to submit at least one test case for each of the major classes you implemented. Be sure to note the relative path of your HTML form page in the `README`, so that we may test it with a browser.

5 Resources

The framework code in svn includes a JAR file containing version 2.4 of the servlet API, as well as (in the `extra` directory) 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.

5.1 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 'demo' in your `web.xml` file. To run:

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...

6 Extra credit

6.1 Persistent HTTP connections and chunked encoding (+10%, due with MS1)

We do not require persistence or chunking in your basic HTTP 1.1 server. However, each of these will count for part of up to 10% extra credit.

6.2 Event-driven server (+15%, due with MS1)

This extra-credit item is only available for MS1. To claim it, you must submit two working versions of your web server: One with a thread pool, and a second, event-driven one. The event handlers must be non-blocking, i.e., all I/O must be asynchronous (Java NIO). We recommend that you first implement the basic server and then refactor it to be event-driven.

6.3 Performance testing (+10%, due with MS2)

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.

6.4 Multiple applications and dynamic loading (+25%, due with MS2)

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 any 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.)

6.5 Servlet test suite (+up to 10%, due on February 15)

Write a number of servlets that test various (nontrivial!) aspects of the servlet container, similar to the example servlets that were distributed with the framework code. When your servlets are loaded and accessed with a browser, they should display a web page that describes which features are being tested, as well as UI elements (links, buttons, ...) that start the test. The result should clearly describe whether the test succceeded, and if not, what went wrong. Submit your servlets by Feburary 15th as a self-contained archive (with an `ant` build script) using `hw1test` as the `turnin` project name. The number of points will be based on the number and complexity of the tests. All tests must pass with Tomcat or Jetty!