1. Background

This assignment focuses on developing an application server, i.e., a Web (HTTP) server that runs Java-based services. You’ll be doing this 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). This web server will allow you to get a nice, limited-scale introduction to building a server system as it requires careful attention to concurrency issues and well-designed programming abstractions to support extensibility.

In the second stage, you will expand your webserver to handle Web service calls. Web service calls are the underpinnings of many aspects of the cloud and are used to invoke operations remotely. One method for creating services in Java is through servlets, with which you might be familiar if you’ve taken CIS 450/550. Increasingly, however, there has been a movement towards lighter-weight RESTful services that expose APIs over HTTP’s GET, POST, DELETE, PUT, and other operations. Such services are typically written by attaching handlers (functions called by the Web server) to various routes (URL paths and patterns). There are many frameworks that follow this paper, e.g., Node.js for JavaScript, Django for Python. In Java, perhaps the most popular framework for such operations is Spark Java (not to be confused with Apache Spark). We will implement a microservices framework that emulates the Spark API.

Eventually, your Homework 1 server and framework will be used in your Google-style final projects to (1) serve HTML pages, e.g., for your search engine; (2) support actions in response to HTML forms, e.g., to trigger the actual search; (3) support REST Web service calls, e.g., to build a distributed web crawler.

1.1. Milestone 0: A Sample Program in the Spark Framework

This homework assignment consists of two milestones, with the final result of Milestone 2 being a platform for users to create their own web services. The platform you will build emulates that of the Spark Java framework. Technically, you can build a one-off solution for Milestone 1, then throw much of it away in order to do Milestone 2. That’s not what we recommend -- instead we want you to understand where you are going with Milestone 2 before embarking on Milestone 1. If you implement Milestone 1 using the same raw framework as Milestone 2 (not necessarily implementing every bell and whistle initially), you’ll be much better positioned for the second milestone.
Thus, to begin we will explore the programming abstraction that you will be providing to users of your system. We will do that by experiencing, first-hand, what it’s like to use the Spark Java interface. Let’s look at a sample Spark user program, from their “Getting Started” documentation.

```java
import static spark.Spark.*;

public class HelloWorld {
    public static void main(String[] args) {
        get("/hello", (req, res) -> "Hello World");
    }
}
```

Upon a browser request (to localhost:45555/hello), the user wants to return “Hello World.” Super simple, right?

Before diving into how it works, it is worth noting two aspects of Java 8 that you may not have seen.

1. `import static spark.Spark.*` finds the spark.Spark class (in a JAR file) and imports all static functions in the Spark class to the global scope. Thus, `public static void Spark.get(...) is callable as get()`.
2. `(req, res) -> “Hello World” is a lambda function that may be familiar to you if you have used functional languages. It takes a pair of input parameters, req and res, and returns a string (“Hello World”). The types of req, res, and the function return value were defined by the get() function itself: namely, they are of type Route which takes a Request and a Response, returns a String, and optionally throws a HaltException if the request results in an error.

Recall that HTTP defines a mechanism for the Web server (or a client) to make a request, and that the server must do some computation and send back a response.

The call to `get` is registering a route handler for HTTP GET requests (to /hello). (Other kinds of HTTP requests, such as POST, DELETE, HEAD, etc. are also mapped to route handlers using the same basic style.) This simple route does not have any patterns or variables, but more complex routes can match on patterns in the URL, define parameters in the path, or parse an HTTP query string. The route handler is the trivial “Hello World” lambda function. In Spark, the handler writes results to a String return result, but also has the ability to modify aspects of the HTTP response by modifying the Response object. If something goes wrong, the handler can throw a HaltException that returns an HTTP error code. Later in the semester, we’ll see how REST calls are mapped to these same sorts of routes.

Your first task is to read the Spark Java documentation and try your hand at creating a simple calculator web service. We recommend that you continue using Eclipse and Bitbucket as with HW0. As with HW0, you should first fork the framework code for HW1 (from https://bitbucket.org/upenn-cis555/555-sparkdemo) to your own private repository; then clone and import it. Recall that there is a bit of work to set the Maven run options to clean, install, and execute the code.
The Spark web service you create should listen on a static port 45555 and respond to requests for 2 resources:

- **http://localhost:45555/add/<x>/<y>**  
  Return a document with the sum of x+y and nothing else
- **http://localhost:45555/mul?x=<x>&y=<y>**  
  Return a document with the product of x*y and nothing else

For example, entering **http://localhost:45555/mul?x=3&y=2** should return “6”. Please submit your CalcService.java to Gradescope:

- Go to the Terminal in Vagrant and type in:
  ```bash
cd /vagrant/555-sparkdemo
zip ../hw1-m0.zip -r pom.xml src/*
```

### 1.2. The Basic CIS 455/555 HTTP Server Framework

For the rest of the assignment, you will be implementing the framework that you just used. Our system architecture is heavily based on the Spark Java Framework, in order to ensure it is realistic. As such, your code may be tested against a variety of JUnit tests, and some aspects of it need to conform to our API.

As the saying goes, a picture is worth a thousand words. So we’ll start with an illustration of the standard “flow” among the classes we have given you, which we expect to occur as Web requests are made.

![Diagram of web server setup](image)

The **WebServer** contains your main function, any server configurations, and takes command line arguments. With the help of the **SparkController** (which emulates spark.Spark), it can create and configure a **WebService**, which does most of the work processing and handling requests. Internally, the
WebService contains an HttpListener that listens for incoming requests on a ServerSocket, a thread-safe HttpTaskQueue and a series of HttpWorkers that consume tasks from the queue. Each worker, as it receives an HttpTask, uses the HttpIoHandler to parse the data on the socket, create a Request, call some type of RequestHandler to handle the request, and creates a Response that is then sent via HttpIoHandler to the client.

For the first milestone, the handler will take a limited subset of possible information from the Request, such as the URL path and either return a file, show some control information, or shutdown the server. Subsequently, in Milestone 2 you’ll generalize the handler to invoke user-defined routes.

2. Developing and running your code

Now you have some understanding of how things are supposed to work. Before you start writing an implementation, we strongly recommend that you do the following:

1. Carefully read the entire assignment (both milestones) from front to back and make a list of the features you need to implement.
2. Read the debugging and testing tips below, for hints on how to approach testing and debugging.
3. Check out the project ProducerConsumerDemo from Bitbucket (https://bitbucket.org/upenn-cis555/555-producer-consumer.git; refer to HW0 instructions for cloning from Git and importing into Eclipse) and see if you can understand how it (1) uses synchronization and (2) uses logging. Where might these techniques be useful in your implementation?
4. Figure out what to prioritize. For Milestone 1 you are just serving static content: you don’t need to support advanced functions like forms and sessions, nor will you need user-provided routes, filters, etc. Those will come in Milestone 2. Most of the most important functions will be those described in “HTTP Made Really Easy.”
5. 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.
6. Spend at least some time thinking about the design of your solution. What classes, beyond the ones we have provided, will you need? How many threads will there be? What will their interfaces look like? Which data structures need synchronization? And so on.
7. Regularly check your changes into your git 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.

Do NOT look at or use any third-party code other than the standard Java libraries (exceptions noted in the assignment) and any code we provide. Yes, there are higher-level libraries that do some of the core functionality we want -- but the goal is to learn how these work!

2.1 Getting the Homework Source Code

Again, fork the framework code for HW1 (from https://bitbucket.org/upenn-cis555/555-hw1) to your own private repository; then clone and import it.
One difference in the maven “pom.xml” file is a set of arguments in the “exec-maven-plugin” section seen below. These configuration options specify the main class as well as two arguments that will be passed into the main method when using the goal “exec:java”. These arguments will be explained in Section 3.

```xml
<configuration>
  <mainClass>edu.upenn.cis.cis455.WebServer</mainClass>
  <arguments>
    <argument>45555</argument>
    <argument>/vagrant/555-hw1/www</argument>
  </arguments>
</configuration>
```

To ensure proper grading, your submission must meet the requirements specified in Sections 3.3 and 4 below - in particular, it must build and run correctly in Eclipse and have an mvn script that correctly builds and runs the code. During submission, the build server will attempt to “vet” your code and let you know if there are any issues, but we cannot promise that this pre-screening will be 100% effective.

### 2.1. Logging, testing, and mock objects

In this homework, we make use of three standard techniques and tools that are extremely valuable when trying to develop high-quality code (and save both sleep and sanity). Note that tests should help you flesh out, develop, and validate your code and you should be writing tests even as you write your code. (There are some who believe you write the test cases first and then write the code!) You should make use of:

1. **Logging infrastructure.** You’ve probably used “System.out.println” at various points to see what’s going on in your program. Logging tools, like Apache Log4J, are a generalization of this idea and allow much finer-grained control of what messages you see and where you see them.

2. **Modular interfaces and unit tests.** You have probably used JUnit tests before to validate functionality in your code. For something like a Web server, unit-testing sub-functions (e.g., the HTTP request parser, sleep and wake-up on the request queue, setting the response variables correctly, setting cookies) is extremely helpful in validating the overall functionality. Maven helps automate this process.

3. **Mock objects.** Sometimes you’ll need to simulate different parts of the code with stubs or fake (“mock”) objects. For instance, to test HTTP parsing from a socket, you might want to create a “fake” socket that lets you closely manage what is sent and received.

#### 2.1.1. Logging in Log4J

Let’s first talk about the logging system. Apache Log4J (and other implementations of logging infrastructure) allow your program to log progress, in several ways that are more powerful than “System.out.println”:

1. The log can go to a file instead of, or in addition to, the console. In real server applications, you would always write your data to a log file.

2. The logging infrastructure records what code wrote the log message, in addition to the message itself. That can be incredibly helpful in debugging!
3. The logging infrastructure lets you specify different *levels* of messages: low-level debug messages (DEBUG), informational messages (INFO), warnings (WARN), and full errors (ERROR). You can set the level of messages you want to see, e.g., to disable debug messages when you think the program is mostly running OK.

4. The logging infrastructure lets you turn on and off the *sources* of messages: you may only want to look at logging messages from a certain class or package.

We are using Apache Log4J v2 in this course. This version of Log4J will optionally look for a `log4j.yaml` describing where you want to send the log messages, what you want to capture, etc. You can find details on this via Google. For simplicity we are actually just setting the Log4J configuration in the main program, e.g., `edu.upenn.cis.cis455.WebServer` with a call to `Configurator.setLevel` that records DEBUG (and above) messages for everything in the `edu.upenn.cis.cis455` package. By default, the output will go to the console.

For each class that uses the logger, you need to (1) create a static logger object for the class, (2) write to the logger. If you look at `edu.upenn.cis.cis455.m1.handling.HttpParsing` (in the 555-sample-parser) you’ll see at the top how we create the logger, ultimately parameterizing it with the class. Be sure to pass in the correct class to the `getLogger()` function, as this is used to help you track and filter where log messages are originating. Finally, you can call `logger.info()`, `logger.debug()`, `logger.error()`, etc. in your code to write to the log.

2.1.2. Unit tests and mock objects

As mentioned previously, you should think carefully about the components of your server and how to expose the “right” interfaces. Hopefully the set of default classes and the figure early in this document will help a bit. If you do your design well (or iterate until you get it right), you’ll likely have some logical ways of testing the functionality in a self-contained way. **It’s time to develop a set of unit tests in JUnit.** If you put them anywhere under the `src/test/java` directory in your project, Maven will automatically run the tests when you build the program -- thus making sure you didn’t inadvertently introduce bugs.

We’ve given you one sample JUnit test, `edu.upenn.cis.cis455.m1.server.TestSendException`, which is designed to test that the `HttpIoHandler.sendException` function correctly sends a 404 HTTP error when called. You might want to do something similar to test that you can parse requests (and send errors when the parsing fails), to test that a correct response is sent, etc.

If you look at our sample code in the provided `TestHelper`, you’ll see that we create a “mock object” for the request Socket, using a library called Mockito, and making use of `ByteArray` streams to emulate the input and output to the socket. You can see the code here:

```java
Socket s = mock(Socket.class);
byte[] arr = socketContent.getBytes();
final ByteArrayInputStream bis = new ByteArrayInputStream(arr);
```
The `s` object partly emulates a `Socket`, but is in fact not a real socket. We create a “fake” input stream that is returned on a call to `s.getInputStream()` -- instead this returns the `ByteArrayInputStream` `bis`. Similarly, `s.getOutputStream()` returns the `ByteOutputStream` `output`. Similarly, we create a default `InetSocketAddress` for a call to `s.getRemoteSocketAddress()`. When we pass this object `s` to our code-to-be-tested, it can’t tell this isn’t a “real” socket so it will read/write as appropriate. When we test `sendException()` we expect it to not actually read from the input stream, but to write to the output stream. Now we can read the output stream from the byte array “underlying” the `ByteArrayOutputStream`, convert it to a string, and test what is in the string.

You should be able to generalize your own unit tests from this sample. Note that we’ve shown how to pass in an input to the mock socket here -- which isn’t useful for testing the `sendException()` function but might be useful in your own code.

### 2.2. “Integration testing” your server

If you did a good job with your unit tests, a lot of the basic functionality will work in the server. However, you ultimately need to “integration testing” as a whole server. Please take a look at the debugging/testing tips we have mentioned below, just as a reminder of how to chase down the inevitable bugs.

The expected “main program” for your Web server should be `WebServer`. To test your whole server, you have a variety of options. Note that, by default, utilities on your host OS will only be able to test your server if it is running at port 45555, but utilities on your guest OS can test on any port. Regardless, you should ensure that your server works correctly on any port by modifying the “pom.xml” file.

- You can use the **Developer Tools** in Chrome to inspect the HTTP headers. On your host machine, use Chrome to visit your webserver at `http://localhost:45555`. Open the main Chrome menu, choose "More Tools", and click on "Developer Tools". This should pop up a new window. Click on the **Network** tab, which will list all the HTTP requests processed by Chrome (click on a request for extra details).
- If you want to check whether you are using the correct headers, you may find the site `http://websniffer.cc/` useful.
- From the Terminal on either your host OS or VM, you can use the `telnet` command to directly interact with the server. Run:

```
    sudo apt-get update
    sudo apt-get install telnet
telnet localhost <port>
```

Then type in a GET request and hit Enter twice; you should see the server's response.
More advanced tools, as you have a “mature” server and want to see how well it does, can be run from the Terminal:

- 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:
  - Run `sudo apt-get update` and then `sudo apt-get install apache2-utils`.
  - Apachebench (ab) can be configured to make many requests concurrently, which will help you find concurrency problems, deadlocks, etc.

We suggest that you use multiple options for testing; if you only use Firefox, for instance, there is a risk that you hard-code assumptions about Firefox, so your solution won't work with curl or ab. You may also want to compare your server's behavior (NOT performance!) with that of a known-good server, e.g., the CIS web server. Please do test your solution carefully! Also, please do NOT run Apachebench across major Web sites, as you will likely make them angry for doing what looks like a DoS attack on them!

### 3. Milestone 1: Multithreaded HTTP Server

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 port in HTTP, but it is often blocked by firewalls, so your server should be able to run on any other port (e.g., 45555). 45555 is forwarded between your host and guest OSes. You can [modify the Vagrantfile](https://www.example.com) to change this.

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!) By default, we have included a subdirectory called www, but you should not assume this is the only possible Web “home” directory.

So, for instance, if you go into your `/vagrant/555-hw1` directory and run `mvn exec:java -Dexec.args ="45555 /home/vagrant/myweb"` this should run your server on port 45555. **By default, if no parameters are given, you should assume that your server uses port 45555, and that the root directory for static Web pages is “./www”**.

For this milestone, your program will accept incoming GET and HEAD requests from a Web browser, and it will make use of a thread pool (as discussed in class, and written yourself from data structures + synchronization primitives rather than using JDK functions such as the Executor Service) to invoke a worker thread to process each request. When an HTTP request is received by the worker, simplified Request and Response objects should be created (see .../m1.interfaces for the foundations), and a simple FileRequestHandler should determine which file was requested (relative to the root directory...
specified above) and return the file. **If a directory was requested, the request should (1) look for index.html within that directory and return that if it exists, and (2) if not, return a 404 error.**

Your server should return the correct MIME types for the most common basic file formats, based on the extension (.jpg, .gif, .png, .txt, .html). Keep in mind that image files must be sent in binary/byte stream 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 static file root directory. We will validate, e.g., that we can't get hold of /etc/passwd!

### 3.1. Expected level of HTTP protocol support

Your application server must be HTTP 1.1 compliant, and it must support all the features described in HTTP Made Really Easy. However:

- **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.

HTTP Made Really Easy is not a complete specification, so you will occasionally need to look at RFC 2616 (the 'real' HTTP specification; [http://www.ietf.org/rfc/rfc2616.txt](http://www.ietf.org/rfc/rfc2616.txt)) for protocol details. Note that a well-defined protocol and precisely matched protocol implementation is really important! The majority of web developers’ headaches come from small differences in interpretation of the spec. **Thus, it is important to think through every small detail including capitalization effects and what to do on invalid inputs.** If you have a protocol-related question, please make an effort to find the answer in the spec before you post the question to Piazza!

### 3.2. Special URLs

Your application server should implement two special URLs. If someone issues a **GET /shutdown**, your server should shut down immediately after any threads handling requests have finished. If someone issues a **GET /control**, your server should return a 'control panel' web page, which must contain at least a) a list of all the threads in the thread pool, b) the status of each thread ('waiting' or the URL it is currently handling), and c) 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 Core Requirements

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!

3.3.1. Standard API

You must also implement the standard interface, which we’ll be using to test. As in real systems, non-conforming code is incorrect code! Refer to the Figure in Section 1.2 for the overall “flow.”

- The factory and dispatcher-controller: SparkController. Just as developers statically import spark.Spark.* in the Spark Framework and use it to configure routes and server behavior, developers using your Web server framework will statically import edu.upenn.cis.cis455.SparkController. This class is used by the developer’s main program to control the Web services. It establishes the port, routes, etc., via functions such as get(), post(), etc.

- The WebService, which encapsulates an HttpListener listening on a specific port. It is the actual code that registers the route-handling and configuration calls such as get(), post(), etc., and it sets up the HttpListener and other code as appropriate.

- The objects to capture the HTTP Request and Response. We have a first version of these in _m1.interfaces, which have a subset of the whole API appropriate for Milestone 1. A second, more complete version appears in _m2.interfaces, e.g., handling cookies, form parameters, etc. You’ll need to develop your own implementations.

3.3.2. Provided Skeleton Code Whose API You Don’t Need to Maintain

We’ve also given you a bunch of starter code to help establish the main components of the project. These are all for your own internal use, as opposed to public APIs, so you may change the function interfaces. Note that some of our sample code does call functions here -- so you would need to change accordingly.

- HttpListener: The actual HTTP server with thread pool and thread management. This class should create and manage the thread poll, listen on sockets, and invoke handlers.
- HttpTask -- this is a “unit of work” for a Web service request handler, encompassing the data for a request.
- HttpWorker -- this is the actual Web service “worker” that takes an HTTP request, parses it (using HttpIoHandler), and sends responses (or exceptions).
- HttpTaskQueue -- this is intended to be the concurrency-protected queue between the server and the workers.
• **HttpIoHandler** -- this processes incoming requests on the socket, and sends outgoing responses. For now there is a (completely stubbed!) sendException() method.

You can see [https://bitbucket.org/upenn-cis555/555-sample-parser/](https://bitbucket.org/upenn-cis555/555-sample-parser/) for sample code for parsing HTTP requests – which you are free to use, or not! **Please note that this is just starter code and you may need to extend it.**

### 4. Milestone 2: Microservices Framework

The second milestone will build upon the Web server from Milestone 1, with support for building HTTP services, using the full routing framework model based on the Spark Framework interfaces. Basically, your first Milestone built a static HTML server, and your second Milestone builds out the full microservices framework.

- As a first step, you should probably checkpoint your Milestone 1 implementation in case you want to revisit it during regrades. Afterward, you should change the m1 imports in SparkController to use m2.

- A developer should be able to register **Route** handlers, **Filters**, and the like from the SparkController.
  - Recall that one registers a route handler by calling post(path, route) for a POST request, get(path, route) for a GET request, etc. The path may actually include a pattern with wildcards (see the description in the Spark Framework documentation). The handler takes a Route (which in turn has a handle method that takes a Request and a Response).
  - A filter is called before (or after) the route handler, and is typically there to prevent certain requests from being handled (by throwing a HaltException, which is caught before the call to the route handler) or to add parameters to the HTTP request (giving more detail for the route handler).

  Filters are commonly used to validate that a request is authorized (e.g., it belongs to an active session) before the route handler is called. See, for an example, the before() filter in this Spark Framework example.

- You are expected to fully implement objects that provide the functionality hinted at with the ...m2.interfaces “stub” classes (modeled after Spark Framework). Users should be able to create and query for **Cookies**, create **Sessions** that expire if not renewed, and process HTTP form and GET parameters.

- If a route handler throws a HaltException your server should catch the exception, and return an appropriate HTTP response with the error code and message.

- If a route handler calls redirect() in the response, your server should send an HTTP redirect message (a 301 or 302 code, as specified in the HTTP spec) to forward the browser to a new location.

- When creating a session, your session token must not be guessable: the client can ‘fake’ the session ID in a cookie.
For routes with wildcards, you may assume that a wildcard only matches a single ‘step’ between slashes. See extra credit for a more general solution.

Additionally:
- You should build test cases for the new functionality, and develop tests to validate parsing, responses, handlers, etc.
- Assume we may replace your version of WebServer with our own that instantiates a set of routes and filters.
- Logging messages from Log4j should be saved to a log file. You should be sure to use the INFO log level to track user requests for URLs.

To help you get started (and to form a reference specification), we have given you some additional, more fleshed-out classes, e.g., for requests, responses, and servers in the m2 “package tree.” You’ll note that most of these are subclassed from the m1 branch, but include functions for manipulating cookies, parameters, sessions, etc. Your Milestone 2 should completely implement the APIs here.

You’ll likely get a lot more insight into how the route handlers should work by looking at example applications for the Spark framework. Be careful to focus on the Spark Framework parts -- don’t let the Kotlin examples or the Spring code distract you from the key points!

4.2. Special URLs
You should now augment the /control URL from Milestone 1, to provide a way to view the Web server’s error log. It may provide other (e.g., extra-credit) features as you see fit.

5. Extra Credit

5.1. Directory display (+2%, due with MS1)
Some web servers provide a directory listing if you give them a URL to a path in which there is no index.html. This directory listing must include the files and directories within the specified directory, and they must be linkable. See as an example https://downloads.apache.org/accumulo/.

5.1. Persistent HTTP connections and chunked encoding (+10%, due with MS2)
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. To get full credit for chunking support, your server needs to be able to both send and receive chunked messages.

5.2. Performance testing (+5%, due with MS2)
Create a route handler with a computationally intensive task, which takes several 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 your route handler 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. Submit this with Milestone 2 as performance.pdf.
5.3. Multiple simultaneous sockets / servers (+15%, due with MS2)
The project described above loads one web application and installs it at the default port of 45555. Extend it to allow for multiple ports, optionally each with different routes (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).

5.4. General wildcards in routes (+5% EC)
For simplicity we are assuming routes can only have single step wildcards, i.e., a route wildcard has no slashes in it. More generally, the Spark Framework allows for multistep wildcards. Implement this more general approach and add support for the `splat()` function in the Request interface.

6. Additional Requirements and Submission
Before you submit, you should test your code extensively. The code must contain a reasonable amount of useful documentation and at least 5 new test cases for each of the milestones. You’ll submit each of the milestones separately, following the guidelines below.

Create a README file with information about any extra credit you did, how to run such extra credit functions, and any other comments you want to add.

6.1. Submitting Your Homework
Now you will need to create a Zip file.

- Go to the Terminal in Vagrant and type in:
  ```
  cd /vagrant/555-hw1
  zip ../*m1.zip -r pom.xml README src/*  
  # or -m2 for milestone 2
  ```

Submit your zip file to Gradescope just like you did with Homework 0. The autograder will provide some simple feedback after you submit but note that these are just very basic functionality. We will run many additional tests and you should too!