Spark Tutorial: Building Web Applications with Java

Simon Andrews

HackUMass VII

Table of Contents

Welcome!

HTTP crash course

Gradle crash course

“Hello, World!” revisited

Building a calculator

Improving the calculator

Templates

Recap and future steps

A taste of Spark: “Hello, World!”

```
import static spark.Spark.*;

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

Who is this workshop for?

- This workshop was *designed* for students and recent graduates of COMPSCI 121/186/187 at UMass who want to make something useful with their new Java skills.
- This workshop will *probably be useful* to anyone with any programming experience in any language.

Before we get started

Do you have a copy of the Java Development Kit (JDK) installed? Open a terminal and try to run “javac”.

- If you get a whole bunch of lines, you’re set.
- If you get something about javac not being found, you’ll need a copy of the JDK. Visit `https://adoptopenjdk.net/` and grab yourself a copy!

What’s the plan for today?

In one hour we will:

- Give a whirlwind introduction to HTTP and Gradle.
- Build a “hello world” website.
- Build a super bare-bones calculator application.
- Flesh out our calculator application a bit so it’s more like a project you would actually make.

Table of Contents

Welcome!

HTTP crash course

Gradle crash course

“Hello, World!” revisited

Building a calculator

Improving the calculator

Templates

Recap and future steps

What is HTTP?

- **H**yper**t**ext **T**ransfer **P**rotocol is a system for exchanging documents.
- Every HTTP request is an interaction between a **client** and a **server** using some **method** on a **resource**.
- This is what you use when you connect to websites using your web browser (*http://*www.example.org).
- (HTTPS is an extension of HTTP that encrypts your traffic.)

An “HTTP conversation”

Client	Server
	“Anyone out there?”
“GET /profile”	
	“200 OK: name=Jim, age=20”
“POST /update age=21”	
	“401 Forbidden”

Common HTTP request types

- GET: used to retrieve a resource from a server.
- POST: used to send data to a server to create/update a resource.
- Other less common request types: PUT, HEAD, DELETE, PATCH, OPTIONS

Common HTTP response codes

- 2XX: success messages
  - **200 OK:** Basically, everything went fine.
- 3XX: redirection messages
  - **301 Moved Permanently:** The requested resource is no longer at this address.
- 4XX: client error messages
  - **400 Bad Request:** Catch-all “you messed up” response.
  - **404 Not Found:** The requested resource does not exist.
- 5XX: server error messages
  - **500 Internal Server Error:** Catch-all “I messed up” response.

Comprehensive reference: `https://httpstatuses.com/`.

So what?

- The application we will write will act as a HTTP server.
- The framework we'll be using, Spark, is actually just a nice frontend to Jetty, an HTTP server.
- When we run our program, it will make available an HTTP server we can connect to.

Table of Contents

Welcome!

HTTP crash course

Gradle crash course

“Hello, World!” revisited

Building a calculator

Improving the calculator

Templates

Recap and future steps

But before we start with that. . .

- We're in the hands-on portion now!
- Go to *some URL* and download and extract the “stage 0” ZIP file.
- Next, fire up a terminal and `cd` over to the folder you just extracted.

What is Gradle?

- Gradle is a **build automation** tool popular with Java developers.
- Gradle's main jobs are to
  - manage your project's **dependencies**, and
  - control building your project's **artifacts**.
- Learn more: `https://gradle.org`

Why should you care?

- Gradle can be used by anyone in pretty much any workflow.
- Gradle helps you share your code and use code other people shared with you.

How to control Gradle: build.gradle

Here's what a minimal build.gradle looks like for a Java project.

```
plugins {
  id 'java'
}
```

It can be this small because Gradle makes a lot assumptions about your project's structure.

Organizing your sources for Gradle

- src/ – Source code
  - main/ – Application code
    - java/ – Java source code
    - resources/ – Other files (configuration, data, etc.)
  - test/ – Unit tests
- build.gradle – Instructions for building your project
- settings.gradle – Extra configuration stuff for Gradle

You can see some of these in the project from the stage 0 ZIP.

Adding dependencies to build.gradle

```
plugins {
  id 'java'
}

repositories {
  jcenter()
}

dependencies {
  implementation 'com.sparkjava:spark-core:2.9.1'
}
```

About JARs

A **JAR** file (**J**ava **AR**chive) is a container that you can put compiled Java code (+ other stuff) into one neat runnable package.

```
jar {
  manifest {
    attributes(
      'Main-Class':
        'org.simonandrews.sparktutorial.App'
    )
  }
  from {
    configurations.runtimeClasspath.collect {
        it.isDirectory() ? it : zipTree(it)
    }
  }
}
```

Actually using Gradle

- From build.gradle, Gradle defines **targets**, actions you can run.
- You can run these from the command line: ./gradlew *sometarget*.
- Try it:
  - Build: `./gradlew build`
  - Run: `java -jar build/libs/spark-tutorial.jar`
- Using Eclipse? Run `./gradlew eclipse` to turn your directory into an Eclipse project, then import it!

Table of Contents

Welcome!

HTTP crash course

Gradle crash course

“Hello, World!” revisited

Building a calculator

Improving the calculator

Templates

Recap and future steps

“Hello, World!” again

```java
package org.simonandrews.sparktutorial;

import static spark.Spark.*;

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

Replace the code in `src/main/org/simonandrews/sparktutorial/App.java` with this, then `./gradlew build`!

Let's run it

- `java -jar build/libs/spark-tutorial.jar`
- If all you get is some complaining about “SLF4J” you did it right!
- Point your web browser to `http://localhost:4567/hello`. You should see a “Hello, World!” message.
- Stop the program by doing “Ctrl-C” in your terminal.

What just happened?

1. When you ran the application with `java`, you started an HTTP server on your computer.
2. The web server listen for connections on `port` 4567.
3. When you connect to `http://localhost:4567/hello`, your web browser makes a GET request for the resource /hello on the server.
4. The server matches your request to a function that generates a string, then sends that string back to your browser.

Table of Contents

Welcome!

HTTP crash course

Gradle crash course

“Hello, World!” revisited

Building a calculator

Improving the calculator

Templates

Recap and future steps

Your mission

- "Hello, World!" is boring. Let's build something that can interact with the user.
- In this workshop, we'll be building a four function ($+$, $-$, $\times$, $\div$) calculator.
- Our interface will work like this: for every operation, `http://localhost:4567/n/op/m` will display the result of applying the operation *op* to the operands *n* and *m*.
- For example, `http://localhost:4567/4/plus/2` will display 6.

GET requests with parameters

- Programming `/0/plus/0`, `/0/plus/1`, `/0/plus/2`, etc. would take a pretty long time.
- Luckily, we can tell Spark to look for patterns in the requested resources and interpret them.
- We can make a route that looks like "$n/op/m$", then in the function for the route we can work with those values.

Introducing the calculator!

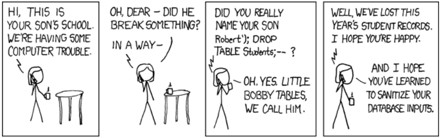
Update App.java:

```
package org.simonandrews.sparktutorial;

import static spark.Spark.*;

public class App {
  public static void main(String[] args) {
    get("/:n/:op/:m", (req, res) -> "TODO");
  }
}
```

Try compiling and running this program and accessing some URLs with your web browser. What works? What doesn't?

URL parameters with Spark

- ":n", ":op", and ":m" are allowed to be any string.
- Our next step will be to make sure that they're what we expect them to be.
- This is super important:
  - Makes sure our application behaves the way we expect it to.
  - For a "real world" application, this might be important for security as well.


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH, YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Comic by Randall Munroe of xkcd: https://www.xkcd.com/327/

Playing with parameters

We can get the value of a URL parameter with the request object's `param` method:

```
package org.simonandrews.sparktutorial;

import static spark.Spark.*;

public class App {
  public static void main(String[] args) {
    get("/:n/:op/:m", (req, res) -> req.params(":n"));
  }
}
```

Now every request will return whatever *n* was. For example, `http://localhost:4567/1/plus/3` will display 1.

Doing the computation

With that in our toolkit, let's build out our function a bit:

```
//omitted...
get("/:n/:op/:m", (req, res) -> {
  double n = Double.parseDouble(req.params(":n"));
  double m = Double.parseDouble(req.params(":m"));
  switch (req.params(":op")) {
    case "plus":  return String.valueOf(n + m);
    case "minus": return String.valueOf(n - m);
    case "times": return String.valueOf(n * m);
    case "div":   return String.valueOf(n / m);
    default:      return "oops!";
  }
});
//omitted...
```

Try it out!

- As usual, `./gradlew build` and `java -jar build/libs/spark-tutorial.jar`!
- Try your operations out. For example, what's displayed when you visit `http://localhost:4567/1/minus/0.5`?

Table of Contents

Welcome!

HTTP crash course

Gradle crash course

"Hello, World!" revisited

Building a calculator

Improving the calculator

Templates

Recap and future steps

Problems with the old system

- It's ugly.
- Terrible user experience.
- No one uses the Internet this way.

(It might be nice as an API though!)

What's the real solution?

- Wouldn't it be better if the user could just fill out a form, click submit, and get their result? There would be no need to fiddle with URLs by hand.
- Let's do exactly that!
- HTML has some support for forms built in with the `<form>` tags.

Remember POST requests?

- We will replace our “/:n/:op/:m” endpoint with one single endpoint “/”. This is your website’s **root**, which *basically* means it’s what we get when we visit `http://localhost:4567` without specifying a resource.
- / can be acted upon using two methods:
  - When the client GETs /, they will receive a form asking them for a calculation.
  - When the client POSTs to / with some numbers and an operation, the server will do the requested computation and the client will receive the result.

POSTing from a form

Here's what the HTML code for our form will look like:

```
<form action="/" method="POST">
  n: <input type="number" step="any" name="n" /> <br />
  op:
  <select name="op">
    <option value="plus">+</option>
    <option value="minus">-</option>
    <option value="times">*</option>
    <option value="div">/</option>
  </select>
  <br />
  m: <input type="number" step="any" name="m" /> <br />
  <input type="submit" value="Go!" />
</form>
```

What happens on the client side?

When the user clicks the “Go!” button in their web browser, the browser will make a POST request to / with string *n*, *op*, and *m* as parameters.

It is important to remember that *these strings could be anything*, even though we try to constrain the user’s input using HTML.

What happens on the server side?

We need to tell Spark how to handle these new request types. Let's try a simple example first. Replace App.java with:

```
package org.simonandrews.sparktutorial;

import static spark.Spark.*;

public class App {
  public static void main(String[] args) {
    get("/", (req, res) ->
        "You did a GET!" +
        "<form action=\"/\" method=\"POST\">" +
        "<input type=\"submit\" value=\"Now POST!\"/>" +
        "</form>");
    post("/", (req, res) -> "You did a POST!");
  }
}
```

What just happened?

- We just defined two different methods for the same resource.
- When you GET /, which is what your browser does by default, you see the message “You did a GET!” and a button.
- When you POST /, which is what the form code tells your browser to do, you see the message “You did a POST!”.

Now let's calculate!

```
// omitted...
String form = "<form action=\"/\" method=\"POST\">" +
              // omitted
              "</form>";
get("/", (req, res) -> form);
post("/", (req, res) -> {
  double n = Double.parseDouble(req.queryParams("n"));
  double m = Double.parseDouble(req.queryParams("m"));
  switch (req.queryParams("op")) {
    case "plus":  return String.valueOf(n + m);
    case "minus": return String.valueOf(n - m);
    case "times": return String.valueOf(n * m);
    case "div":   return String.valueOf(n / m);
    default:      return "oops!";
  }
});
// omitted...
```

Try it out!

- As usual, `./gradlew build` and `java -jar build/libs/spark-tutorial.jar`.
- Open `http://localhost:4567` in your browser.
- Fill out the form and click submit.
- Get your answer! Hooray, an interactive site!

Table of Contents

Welcome!

HTTP crash course

Gradle crash course

“Hello, World!” revisited

Building a calculator

Improving the calculator

Templates

Recap and future steps

The problem

- Baking HTML into Java code is terrible.
- You need to escape all your quotes.
- You need to keep track of all the strings you're concatenating.
- You editor doesn't understand you're writing HTML, so you won't get autocompletion or syntax highlighting.
- What should your indenting rules even be?
- What about CSS? What about JavaScript?
- What about long HTML documents?

The solution

Separate your front-end from your back-end!

- We will do this with **templates**, files that have spaces you can fill in dynamically.
- Basically, we can make a template “\$n \$op \$m = \$result” and then render it once we know the values for all those variables.
- We can do this in a separate file, and include that file in our JAR along with all our .classes. (Remember the resources directory from earlier?)

Introducing Velocity!

Apache Velocity™

- Velocity is a library for rendering templates. It has its own template language designed specifically to work nice with Java programs.
- We won't use them today, but you can use for-loops and if-statements in it too. Velocity is very flexible!

Adding the dependency

In build.gradle, change the `dependencies` section to

```
dependencies {
  implementation 'com.sparkjava:spark-core:2.9.1'
  implementation 'com.sparkjava:spark-template-velocity:2.7.1'
}
```

This adds a dependency on an adapter for Velocity that makes it work nicely with Spark.

Building our first template

- Files in the “src/main/resources” will be included in your JAR without any modification.
- Your Java code can then access those files.
- This is where we’ll be putting our templates.

Adding our templates

Put the following in src/main/resources/form.vtl:

```
<form action="/" method="POST">
  n: <input type="number" step="any" name="n" /> <br />
  op:
  <select name="op">
    <option value="plus">+</option>
    <option value="minus">-</option>
    <option value="times">*</option>
    <option value="div">/</option>
  </select>
  <br />
  m: <input type="number" step="any" name="m" /> <br />
  <input type="submit" value="Go!" />
</form>
```

Adding our templates

Put the following in src/main/resources/result.vtl:

```
<p>The result of $n $op $m is $result.</p>
```

Wiring up Spark

In App.java:

```
import java.util.HashMap;
import java.util.Map;
import spark.ModelAndView;
import spark.template.velocity.VelocityTemplateEngine;
// omitted...
get("/", (req, res) -> {
    Map<String, Object> emptyModel = new HashMap<>();
    return new ModelAndView(emptyModel, "form.vtl");
}, new VelocityTemplateEngine());
// TODO: post
// omitted...
```

The POST part

```
post("/", (req, res) -> {
  String op = req.queryParams("op");
  Double n = Double.parseDouble(req.queryParams("n"));
  Double m = Double.parseDouble(req.queryParams("m"));
  Double r;
  switch (op) {
    case "plus":  r = n + m; break;
    case "minus": r = n - m; break;
    case "times": r = n * m; break;
    case "div":   r = n / m; break;
    default:      throw new Exception("oops");
  }
  Map<String, Object> model = new HashMap<>();
  model.put("n", n); model.put("m", m);
  model.put("op", op); model.put("result", r);
  return new ModelAndView(model, "result.vtl");
}, new VelocityTemplateEngine());
```

Voilà!

It works (hopefully)! Go you!

Table of Contents

Welcome!

HTTP crash course

Gradle crash course

“Hello, World!” revisited

Building a calculator

Improving the calculator

Templates

Recap and future steps

What we did today

- Learned about HTTP and web servers.
- Learned about using Gradle and how Java packaging works.
- Built a calculator app in Java using Spark.
- Learned about templates and made super simple Velocity templates.

What we didn't do today

- Make a nice, user-friendly website.
- Store any data.
- Use best practices for large projects.
- Go in depth on MVC.

FIN

Thank you for coming!