FRAMEWORKS

Web microservices development in Java that will Spark joy

The Spark framework might be the platform you need for building web applications that run in the JVM.

by Maarten Mulders

June 25, 2021

Spark is a compact framework for building web applications that run on the JVM. It comes with an embedded web server, Jetty, so you can get started in minutes.

After adding a dependency on `com.sparkjava:spark-core`, all you need to do is write the application skeleton and you're off and running.

```
import static spark.Spark.*;

public class JavaMagazineApplication {
  public static void main(String... args) {
    get("/hello", (req, res) -> "Hello World'
  }
}
```

You can see a couple of interesting things in this small snippet.

- Spark leverages functional interfaces, so it's easy to use lambda expressions for handling a request.
- Spark doesn't require annotations on a method to map it on a request path. Instead, it lets you create this mapping in a programmatic way using a clean domain-specific language (DSL).
- There's no boilerplate code required to bootstrap an application: It's all done for you.

Before diving in, I need to clear off a little bit of dust.

As microservices have become a ubiquitous architectural pattern, there's been a renewed interest in the size of deployed applications and their startup time. In recent years, Helidon, Micronaut, Quarkus, and Spring Boot have entered this space. But the concept of microframeworks is older than those new kids on the block. Let me introduce you to Spark.

If you search the web for that name, chances are you'll find information related to Apache Spark, the analytics engine, initially released in May 2014 by Matei Zaharia. That's something else entirely. Put it out of your mind.

Spark (the one in this article) was founded by Per Wendel, and cofounded by Love Löfdahl, and it dates back to 2013. It's even older than Apache Spark.

To distinguish the two Sparks, the microframework this article covers is often dubbed *Spark Java*, and that's even the URL: sparkjava.com. The official Stack Overflow tag for the microframework is even spark-java. That name doesn't *completely* cover the matter either, since you can equally well use Kotlin to write applications. But thinking about the framework as Spark Java will prevent you from reading about data science stuff and wondering how you got there.

Spark declares many *functional interfaces*, which is why Spark requires Java 8 at a minimum. All code samples in this article can be found on GitHub. All these samples require Java 11 to run; I wrote them that way to be more concise. You could code all the same functionality for Java 8.

Spark is mature, the API is stable, and it isn't upgraded every other week. That occurs about twice a year. Also, the project hasn't gotten a lot of attention lately, which may give the impression that it's been abandoned. (At the time of this writing, the current version is Spark 2.9.3, released October 2020. The previous one, 2.9.2, was released July 2020.)

Main Spark concepts

So, let's dive in and see what happens under the hood. Spark is designed around the concept of *routes*. A route is nothing more than a function that takes an HTTP request and response and returns content in the form of an `Object` that will be sent back to the client. You already saw that the `Route` interface declares one method, making it perfectly suitable for writing lambda functions. Although this allows you to write routes in one line of code, you can also write routes in separate classes, making it easier to test them. Using a method reference to such a route, you can still have an efficient listing of all routes in your application.

```
import static spark.Spark.get;
import static spark.Spark.post;
```

```
public class JavaMagazineApplication {
  public static void main(String... args) {
    var controller = new GreetingController(
    get("/hello", controller::greet);
    post("/name", controller::updateName);
  }
}
```

The `greet` method in the controller would return the traditional "Hello, World" message. The `updateName` method would let the user replace "World" for another name by issuing an HTTP request, as follows, and then retrieve the greeting again:

```
curl --data 'name=Java Magazine' http://local
curl localhost:4567/hello/simple
```

Using this concise DSL, Spark lets you register routes for all common HTTP verbs, such as `get`, `post`, `put`, and `delete`.

Any useful application would—at some point—deal with user input. That's why a `Route` has access to the `Request`. It's an abstraction over the HTTP request that was sent by the client. Of course, it provides access to cookies, headers, query parameters, and other forms of user input. Since Spark builds upon Jetty, the `Request` class abstracts away the low-level details of the `HttpServletRequest` class from the Servlet API. This also means Spark lets you work with request attributes and session management. I'll share more on that later.

A route must return a value that is sent back to the client. But how is that `Object` sent over the wire? It's the responsibility of a *response transformer* to translate such an `Object` into a string of characters for the response. A response transformer can, for example, serialize an object to a JSON structure using the Google Gson serialization/deserialization library.

```
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;
import spark.ResponseTransformer;

public class JsonTransformer implements Respo

  private final Gson gson = new GsonBuilder(

  @Override
  public String render(Object model) {
    return gson.toJson( model );
  }
}
```

A response transformer can transform a response, but what if you want to work with the request before it reaches the route?

That’s where *filters* come into play. Spark invokes your filters either before or after it runs the request through the route.

Until now, you have seen only static routing, where the request URI must match exactly the given path (such as `/hello`). Fortunately, Spark lets you define routes in a more flexible way. Let’s revisit the greeting application and make sure the user can supply a custom name by passing it as a path variable, as follows:

```
import static spark.Spark.get;
import static spark.Spark.post;

public class JavaMagazineApplication {
  public static void main(final String... arg
    var controller = new GreetingController(
    get("/hello/simple/:name", controller::gr
  }
}
```

This lets you customize the greeting.

```
curl localhost:4567/hello/simple/Java
```

But how can you implement a `Route` when you want to access path parameters or other client inputs? Spark provides you with access.

```
public Object greet(Request request, Response
  return "Hello, " + request.params("name");
}
```

Path parameters are available through the `params` method on the `Request` class. Similarly, query parameters (such as `?name="Java"`) are available through the `queryMap()` method on the same class.

What if you wanted to consume JSON data? Of course, it’s also possible to do that. And just as with producing JSON, it’s not something that Spark can do out of the box, which gives you the most flexibility for how you want to do it.

The way I typically do it is twofold. First, I have a factory method that produces a filter. And second, I configure Spark to run that before invoking a particular route.

The factory method looks like the following:

```
public static Filter forType(final Class<?> t
  return (request, response) -> {
    var body = gson.fromJson(request.body(),
    request.attribute(REQUEST_DATA, body);
  };
}
```

To register that method with Spark, I use the following snippet:

```
    before("/hello/complex", "application/json",
        JsonParsingFilter.forType(GreetingInput.
```

Of course, the `REQUEST_DATA` that I use in the `Filter` factory is a constant that can be used to retrieve the object inside a `Route`.

```
    var body = (GreetingInput) request.attribute
```

This may be a little more work than throwing an annotation on top of a method, but it gives you very fine-grained control over exactly how and when the request body is processed.

It's also possible to write a little bit more code than this and have the filter inspect annotations based on the route method. In that case, there would be only one instance of that filter at runtime, rather than one instance per route. On the other hand, the filter would become much more complex by inspecting those annotations.

Packaging an application for deployment

Running an application from a cozy IDE is all well and good, but that's not how to run applications in production. So how would you package a Spark application? You can choose to package it for deployment in an existing servlet container or a standalone app.

Deployment in an existing container. Recall that Spark includes Jetty. If you want to deploy your application in an existing servlet container, it wouldn't make sense to package Jetty with it. In fact, a bundled Jetty wouldn't even work, because its servlet container will not invoke the `main()` function written earlier.

Instead, you should rewrite the application class to implement the `SparkApplication` interface, whose `init()` method is the perfect place to declare filters, routes, and response transformers. That interface's `destroy` method is suitable for cleaning up any resources.

In addition, you must change the build to output a WAR file rather than a JAR file, and you need to exclude Jetty from the application package. Assuming you’re using Maven, here’s how you do that.

```
<packaging>war</packaging>

<dependencies>
  <dependency>
    <groupId>com.sparkjava</groupId>
    <artifactId>spark-core</artifactId>
    <version>2.9.3</version>
    <exclusions>
      <!-- remove Jetty from the packaged app
      <exclusion>
        <groupId>org.eclipse.jetty</groupId>
        <artifactId>jetty-server</artifactId
      </exclusion>
      <exclusion>
        <groupId>org.eclipse.jetty</groupId>
        <artifactId>jetty-webapp</artifactId
      </exclusion>
      <exclusion>
        <groupId>org.eclipse.jetty.websocket<
        <artifactId>websocket-server</artifa
      </exclusion>
      <exclusion>
        <groupId>org.eclipse.jetty.websocket<
        <artifactId>websocket-servlet</artif
      </exclusion>
    </exclusions>
   </dependency>
</dependencies>
```

Deployment in a standalone JAR. There are two options here: a fat JAR or a slim JAR.

The fat JAR approach bundles all classes, resources, and third-party dependencies into a single JAR file. This is the approach that Spring Boot takes, for example. It can result in big fat JAR files, especially as your dependencies grow.

But in the container era, it makes sense to consider the slim JAR approach. In this approach, you build a traditional JAR file with only the classes and resources of your own application. For packaging, copy all third-party dependencies into a folder close to the JAR file, such as `lib/`, and add a line in the JAR’s manifest file that tells the JVM to add all dependencies to the classpath and resolve them from that `lib/` folder, as follows:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId
  <artifactId>maven-jar-plugin</artifactId>
  <version>3.0.2</version>
  <configuration>
    <archive>
      <manifest>
        <addClasspath>true</addClasspath>
        <classpathPrefix>lib/</classpathPref
        <mainClass>it.mulders.spark.JavaMaga
```

```
          </manifest>
        </archive>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId
      <artifactId>maven-dependency-plugin</artifa
      <version>3.1.2</version>
      <configuration>
        <overWriteReleases>false</overWriteReleas
        <includeScope>runtime</includeScope>
        <outputDirectory>${project.build.directo
      </configuration>
      <executions>
        <execution>
          <phase>package</phase>
          <goals>
            <goal>copy-dependencies</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
```

Why is this relevant in the container era? Consider the following snippet for a Dockerfile:

```
    # Add Maven dependencies (not shaded into the
    ADD target/lib /opt/my-application/lib
    # Add the service itself
    ADD target/my-application.jar /opt/my-applica
```

This approach leverages the layered file system that Docker uses. The third-party components likely don't change as often as the application. By putting those files in a subfolder and writing them at once into one layer of the Docker file system, you let the container engine reuse that layer every time you build the application. The final layer of the Docker container, with the application in it, may change many times, but it's small: maybe a couple of KB. That means the container can be built quickly and deployed quickly, which is definitely a win.

Starting fast and staying small

You can expect the size of a packaged Spark application to be small. And indeed, a packaged Spark application "weighs" just around 4 MB. But this also benefits the startup time and memory usage of applications. **Table 1** compares Spark with some other frameworks.

Table 1. Comparison of Spark and other frameworks

	Packaged size (MB)	Startup time (milliseconds)	Memory usage (MB)
Spark Java with Gson	4	4	66
Micronaut*	13	805	112
Spring Boot**	22	2314	301

Table 1. Comparison of Spark and other frameworks

* The Micronaut app was generated using micronaut.io/launch/ with Micronaut 2.3.1.

** The Spring Boot app was generated using start.spring.io with Spring Boot 2.4.2.

I measured those numbers on a 2018 MacBook Pro. For the startup time, I took the output from the logging each application generates. I measured memory usage using the resident set output from the `ps` command-line tool. I averaged all numbers over five measurements.

I haven't used GraalVM to create native executables for those frameworks that support it. Even though there is a lot in this naive test setup that you could debate, you can see that the minimalistic approach Spark takes pays off. Having an extremely short startup time and low memory usage makes Spark an interesting choice for environments that demand low resource usage or high throughput.

I have employed Spark in some performance-critical environments for that reason, and I measured response times on a REST endpoint that hardly exceeded the time to perform necessary database queries; the overhead of the web framework was negligible.

Spark and REST

So far, I've covered applications with an HTTP interface, assuming that it would be JSON that you send and receive over the wire. But Spark can do more. You can add a template engine of your choice and start rendering full web pages straight from Spark. There are many choices, including (but not limited to) FreeMarker, Handlebars, Thymeleaf, and Velocity. Those engines are separate dependencies that you would need to add to your application.

As an example, let's look at embedding Thymeleaf. After adding the appropriate dependencies, you can update your route definitions as follows:

```
import static spark.Spark.get;
import static spark.Spark.post;

public class JavaMagazineApplication {
  public static void main(final String... arg
    var controller = new GreetingController(
    var thymeleaf = new ThymeleafTemplateEngi
    get("/hello/html/:name", controller::gree
  }
}
```

This requires you to update the controller a little bit as well. Rather than returning text to display, you must return a `ModelAndView` object.

```
import spark.ModelAndView;
import spark.Request;
import spark.Response;

import java.util.Map;

public class GreetingController {
  public Object greet(Request request, Respo
    var model = Map.of("name", request.params
    return new ModelAndView(model, "views/we
  }
}
```

You also need to write a view using Thymeleaf.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">

<head>
  <title>Simple Spark web application</title
</head>

<body>

<div>
  <h1 th:text="'Welcome to Spark, ' + ${name
    Welcome to Spark, unknown
  </h1>
</div>

</body>

</html>
```

It's beyond the scope of this article to explain all the bits and pieces of Thymeleaf. For now, note that the text inside the `h1` tag will be rendered by the test application using the `name` key from the model.

Conclusion

As mentioned above, the code samples are available on my GitHub and require Java 11 to run.

Spark isn't a complete toolbox, and it doesn't pretend to be. Rather, it should be one of the tools inside your toolbox, and you as a developer get to find the other tools to complete your toolbox.

I've used Spark in a few projects that had specific requirements for response times. In those situations, its size and the fact that Spark does not heavily use reflection make it an interesting choice.

Spark code is very readable and lends itself well for extensions that the Spark framework doesn't cover.

On the other hand, if your main concern is to deliver features quickly, having to wire tools and components together may, in fact, slow you down. In such scenarios, having a toolbox that is prefilled with solid tools may enable you to deliver features faster.

Dig deeper

- Building microservices with Micronaut
- Helidon: A simple cloud native framework
- How to test Java microservices with Pact

Maarten Mulders

Maarten Mulders (@mthmulders) is an Oracle Groundbreaker Ambassador and a passionate architect, senior developer, and trainer at Info Support. He is focused on "building the right thing" and "building the thing right." He prefers lean and elegant solutions, and he loves to share new ideas and knowledge. Outside of work, Mulders appreciates crafting, eating good food, photography, and music, in no particular order.