Step-by-step: Kotlin, JAX-RS (Jersey), and Docker

Posted on Feb 17, 2018

Hello Again and welcome to another blog post in my ongoing series on step-by-step tutorials. This one will actually build atop the last Hello Kotlin one and make a JAX-RS web service. This particular implementation of JAX-RS will be Jersey. Jersey is actually the reference implementation from Oracle, but there are a few out there that I've honestly haven't given enough time.

The code repository of everything is located on github for you to follow along.

Step 0: Prerequisites

For this one, I am going to start off with my Kotlin Hello World repository. This already has a Gradle wrapper setup, along with some basic things. I've changed `mainClassName` to `Server.kt` and `archivesBaseName` to match this projects name.

Step 1: Web Server

First step will be to get a working web server up and running. This project will be using the grizzly web server, which I recommend, as the embedded web server. It has a decent enough history of performance and from my usage it does a decent job.

So let's take a look at how a quick server can be setup so we can verify we have the HTTP pipeline correct.

src/Server.kt

```
1   import org.glassfish.jersey.grizzly2.httpserver.GrizzlyHttpServerFac
2   import javax.ws.rs.core.UriBuilder
3
4   fun main(args: Array<String>) {
5       val url = UriBuilder.fromUri("http://0.0.0.0/")
6               .port(8080)
7               .build()
8
9       val httpServer = GrizzlyHttpServerFactory.createHttpServer(
10              url,
11              true
```

```
12        )
13
14        if (System.getenv()["SHUTDOWN_TYPE"].equals("INPUT")) {
15            println("Press any key to shutdown")
16            readLine()
17            println("Shutting down from input")
18            httpServer.shutdownNow()
19        } else {
20            Runtime.getRuntime().addShutdownHook(Thread {
21                println("Shutting down from shutdown hook")
22                httpServer.shutdownNow()
23            })
24
25            println("Press Ctrl+C to shutdown")
26            Thread.currentThread().join()
27        }
28    }
```

Line 6: This creates a URI for the server to bind to. This particular one makes it so `http://localhost:8080` will work. Probably overkill but that is what the Grizzly Server factory wants.

Line 9: This creates the Grizzly Server and the 2nd parameter (`true`) makes the server start immediately.

Lines 14 - 27: Now, let me explain this because this is sort of magic. When running under Gradle (`./gradlew run`), Gradle captures all of the input and passes it along to the application. So for example, on the command line you usually stop applications by hitting `Ctrl+C`. This however doesn't work when you run it under Gradle because Gradle captures the signal and stops the daemon that is running the application. Sometimes, the Gradle daemon doesn't stop properly, or clean everything up and produces a mess the next go around.

So my hack to fix all of this was to have a feature flag in an environment variable. When running under Gradle (hold on one sec and I'll show you how) we will just wait for input. When running under everything else (i.e. docker, `java -jar`) we will wait for Ctrl+C or a kill signal from somewhere else.

Now for the toolchain bits, we need to add the dependencies and setup the run bits to work with out shutdown hooks.

build.gradle (only parts)

```
1    ...
2    dependencies {
3        compile 'org.jetbrains.kotlin:kotlin-stdlib-jre8'
4        compile "org.glassfish.jersey.containers:jersey-container-grizzl
5    }
6    ...
```

```
7    // Properties for `./gradlew run`
8    run {
9        standardInput = System.in
10       environment = ["SHUTDOWN_TYPE" : "INPUT"]
11   }
```

Line 4: We add the grizzly http server and Jersey container.

Line 8 - 11: This is how we make `./gradlew run` work. The `standardInput` we set to the normal `System.in` (it defaults to an empty set which breaks things). Then we setup the environment variables with the `environment` configuration.

We need to add a way for the port to be exposed by Docker to the outside world. Since we are binding on port 8080 we need to have that exposed as well. So in the `Dockerfile` we will add one little line before the CMD statement.

Dockerfile (only parts)

```
14   EXPOSE 8080
15
16   CMD ["java","-jar","run.jar"]
```

So verify the two ways to run the application and view `http://localhost:8080`.

First run `./gradlew run` and if you hit any key, it will shutdown.

Second run the docker command `docker build -t myserver . && docker run -p 8080:8080 myserver`. This you can run and then hit *Ctrl+C* to shut it down.

Quick aside on the docker command: there are two parts to it, first you build the image you want to run, then you run it. The `-t` you pass in the build command is a tag that makes it easier to look up later. In the run part the `-p` is the ports we will bind to the host. For this example we are taking the containers's 8080 port mapping it to the hosts 8080 port.

Step 2: Add Simple JAX-RS endpoint that prints "Hello World"

For this let's create the needed pipeline to get the needed resources. There is a little bit of yak shaving needed for this setup to get all of the right pieces in the place but I think I've boiled it down to the simple parts.

First let's see what the resource (these are what the groups of end points are called) look like for our first end point.

src/jaxrs/resources/HelloWorld.kt

```
1    package jaxrs.resources
```

```
2
3   import javax.ws.rs.GET
4   import javax.ws.rs.Path
5
6   @Path("helloWorld")
7   class HelloWorldResource{
8       @GET
9       fun helloWorld() = "Hello World"
10  }
```

There really isn't much to talk about here. The `@Path` is where the end point will be (so in this case `/helloWorld`). This will be retrieved from a HTTP GET.

Alright, the next step is to get the system setup to register these resources automatically. So let's setup the config needed for that.

```
1   package jaxrs
2
3   import org.glassfish.jersey.server.ResourceConfig
4
5   class Application : ResourceConfig() {
6       init {
7           packages("jaxrs.resources")
8       }
9   }
```

Again, this is fairly straight forward. The ResourceConfig is where all of base application is setup. In this example. we scan the `jaxrs.resources` namespace for any resources or other things that can register with JAX-RS. This is a specific to Jersey but it makes it easier to setup things.

Alright, now that we have all of the configuration setup for JAX-RS and a resource, lets make the server recognize it. So to the `Server.kt` file added this.

Server.kt (parts)

```
22    val httpServer = GrizzlyHttpServerFactory.createHttpServer(
23                url,
24                Application(),
25                true
26        )
```

Line 12: This is all you need to add. A new instance of the `Application` class and the Grizzly server does all the rest for you.

Now, one last thing, we need to add a dependency for Jersey. This one will make all of the magical dependency injection happen behind the scenes. The Jersey team have made this configurable dependency, so if you want a different type of injection system you can.

The specific one implementation we will use is HK2 as opposed to Guice or others. This was just the default that I've used and have had no problems with it so I recommend it. So here is the line that needs to be added.

build.gradle (parts)

```
22    dependencies {
23        compile 'org.jetbrains.kotlin:kotlin-stdlib-jre8'
24        compile "org.glassfish.jersey.containers:jersey-container-grizzl
25        compile "org.glassfish.jersey.inject:jersey-hk2:2.26"
26    }
```

Go ahead and run to see the output on `http://localhost:8080/helloWorld`.

Step 3: Add JSON handling

Now that we have a simple web server up, let's get JSON handling. This isn't configured out of the box but it is straight forward. So let's get it setup and working and have an end point setup to demonstrate that.

Let's setup our resource to have the parts needed to show JSON output.

src/jaxrs/resources/HelloWorld.kt

```
10    @GET
11    fun helloWorld() = "Hello World"
12
13    data class HelloJson(val prop1: Int, val prop2: String)
14
15    @GET
16    @Path("json")
17    @Produces(MediaType.APPLICATION_JSON)
18    fun helloJson() = HelloJson(1, "test")
```

Line 13: This is why I love Kotlin, this is a data class. A beautifully pure and simple structure to pass data around. We have two properties that will be exposed to the JSON interface. One an integer, the next a string.

Lines 15-18: This is how you setup an endpoint that will produce a JSON object (well, force it too, but that's for another day, or go read the Jersey documentation). We simply just return an instance of the `HelloJson` data class.

Now, we need to tweak our dependencies and add the JSON handling. While we do this, we will also be violating the Rule of 3, and we are going to refactor out the common things that may change. Let's see the result and we will talk about what exactly that means.

build.gradle (parts)

```
22    def jerseyVersion = "2.26"
23
24    dependencies {
25        compile 'org.jetbrains.kotlin:kotlin-stdlib-jre8'
26        compile "org.glassfish.jersey.containers:jersey-container-grizzl
27        compile "org.glassfish.jersey.inject:jersey-hk2:${jerseyVersion}
28        compile "org.glassfish.jersey.media:jersey-media-json-jackson:${
29    }
```

Line 22: We added this line because the parts that could change between all of these jersey dependencies is the Jersey version. So let's do the work now to make it so we can make the version change in one place. A small application of the Rule of 3, but hey, it works.

Line 28: This is adding Jackson to do our JSON processing. This is the magic dependency that we will need to make this all work.

At this point you can now run it and hit `http://localhost:8080/helloWorld/json` and you should see JSON output.

And with that, you should now have the basics to be dangerous with. You can setup a HTTP REST server and go to town with what ever you need. But hey, let me throw in one more little bonus step that may help you out.

Bonus Step 4: Add verbose logging

Just to make it easier to see requests coming in and out of your service on the command line. Which can help with proving you are either insane or sane depending on the circumstances, let's add a logger that will output each request.

Thankfully this fairly straight forward. We just need to tweak the `Application.kt` setup. So here is what you need.

src/jaxrs/Application.kt

```
12    register(LoggingFeature(Logger.getLogger(LoggingFeature.DEFAULT_LOGG
13            Level.INFO,
14            LoggingFeature.Verbosity.PAYLOAD_ANY,
15            Integer.MAX_VALUE)
16    )
```

These are the lines you need to add to the `init` block. Fairly straight forward (make sure to add all of the right `import` statements!). This should now display all of the requests and responses on the command line so you can verify things are going in and out as expected.

Erik Lindblom

A software developer trying to solve the problems with code.

Contact me