Programming APIs WITH THE Spark Web Framework

AUTHORS

Kristopher Sandoval
Travis Spencer
Per Wendel

NORDIC APIS
nordicapis.com

Programming APIs with the Spark Web Framework

Nordic APIs

Contents

Preface

Often when developers think about Java web development they immediately think of huge frameworks, cumbersome configuration and lots of boilerplate code. As with many other areas of Java development - Cargo-cult conventions, guidelines and habits tend to hold back the language's evolution and hinders it from reaching its full potential. Spark aims to address these problems directly, which is one of the major reasons behind its rapid growth and popularity in the Java world.

Spark Framework utilizes the new language features of Java 8 and gives a much needed injection of energy to the many Java web developers exhausted by the "old ways" of doing things. Spark has no XML, no annotations, minimal configuration and a convention breaking, sleek yet expressive syntax. Spark's goal is to rejuvenate and streamline Java web development, and in the long run, help the language itself evolve.

With a clear philosophy Spark is designed not only to make you more productive, but also to make your code better under the influence of Spark's sleek and declarative syntax.

Hundreds of thousands of developers have already started to adopt to the Spark mindset. The question is: Are you ready to be ignited?

-Per Wendel

Software Architect & Founder of Spark Java

1. Introduction

Perhaps the singular most important choice an API developer can make is one of programming language and architecture. Choosing how an API will communicate with consumers, how security will be implemented, and how the API will function with other services is largely constrained by the language and methodology by which it is developed.

Some languages help solve certain problems while others inhibit solutions. Even after a language is chosen and tens-of-thousands of lines of code have been written, there is the possibility of reducing complexity by using new languages on the same **runtime**.

Collecting the research and walkthroughs we've written and published on the Nordic APIs blog channel, this mini ebook introduces some new technologies that our authors are using internally and would like to share with the web development community. We sponsored the Stockholm Java Meetup this year to discuss using Kotlin, Clojure, and Groovy to build APIs on the JV, and have included those insights as well.

Throughout the course of this guide, we introduce the benefits of using the Spark web framework. We'll demonstrate how it works with Java, and compare functionality with other languages like Kotlin and Scala. Read on to learn how to use Spark to design for speed, productitivy, purpose, and cloud readiness.

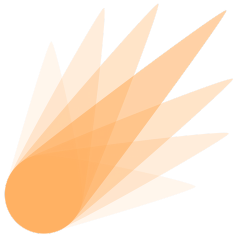
2. Using Spark to Create APIs in Java

In this chapter, we'll discuss a **Java** framework called **Spark**, its basic use, history, and compare it with other languages and toolkits. We'll highlight what makes Java Spark an incredibly functional and useful toolkit for crafting APIs, and provide some examples. In the following chapters, we will also show you how you can use this framework in Scala and Kotlin, two other languages that run on the Java Virtual Machine (JVM).

2.1 Introducing Spark

Spark is a Free and Open Source Software (FOSS) application framework written in Java. Not to be confused with Apache Spark, this toolkit is designed to make it easy and fast to create APIs. It is a lightweight library that you link into your application to start serving up data. It allows you to define routes and dispatch them to functions that will respond when those paths are

requested. This is done in code using a straight-forward interface, without the need for XML configuration files or annotations like some other web frameworks require.



It was first created in 2011 by Per Wendel, and version 2.0 was released in 2014 to take advantage of new language features in Java 8. Since inception, it has aimed to facilitate **rapid development**. According to a 2015 survey, 50% of Spark users utilized the toolkit to develop scalable REST APIs.

A simple example looks like the following prototypical snippet:

```
1  import static spark.Spark.*;
2
3  public class HelloWorld {
4      public static void main(String[] args) {
5          get("/hello", (request, response) -> "Hello World");
6      }
7  }
```

When this main method is executed, Spark will fire up a Web server, and you can immediately hit it like this:

```
1  curl "http://localhost:4567/hello"
```

When you do, the lambda function given to the statically imported `spark.Spark.get` method will fire. The output of the lambda is what the client will be served up (i.e., the string `Hello World`). The routing can get a bit more complicated, but that is the basic idea: you specify a path that gets dispatched to a certain function based on the URI, HTTP action, and accept headers.

2.2 More Complicated Example

Below is an example from Spark's GitHub repository that is slightly more complicated and shows off some of the tookit's other features:

```
1  import static spark.Spark.*;
2
3  public class SimpleExample {
4
5      public static void main(String[] args) {
6
7          get("/hello", (request, response) -> "Hello World!");
8
9          post("/hello", (request, response) -> "Hello World: " + request.body());
10
```

```
        get("/private", (request, response) -> {
            response.status(401);
            return "Go Away!!!";
        });

        get("/users/:name", (request, response) -> "Selected user: " + request.params(":na\
me"));

        get("/news/:section", (request, response) -> {
            response.type("text/xml");
            return "<?xml version=\"1.0\" encoding=\"UTF-8\"?><news>" + request.params("se\
ction") + "</news>";
        });

        get("/protected", (request, response) -> {
            halt(403, "I don't think so!!!");
            return null;
        });

        get("/redirect", (request, response) -> {
            response.redirect("/news/world");
            return null;
        });
    }
}
```

Let's break down the functionality piece by piece to see how else Spark can help with dispatching in your API. First, we have the static response for the `hello` endpoint (i.e. `http://localhost:4567/hello`) and the slightly more dynamic POST handler:

```
get("/hello", (request, response) -> "Hello World!");

post("/hello", (request, response) -> "Hello World: " + request.body());
```

Later, we'll compare this snippet to one written in Go, but for now notice that these two method calls cause Spark to route messages to the `hello` URL when made as an HTTP GET or POST request. The latter isn't much more complicated than the previous one — it just appends the request's contents to the string `Hello World:` and flushes that to the response.

Next, we have our API's more guarded routes:

```
get("/private", (request, response) -> {
    response.status(401);
    return "Go Away!!!";
});

// ...

get("/protected", (request, response) -> {
    halt(403, "I don't think so!!!");
    return null;
});
```

In these cases, a visit to the `/private` or `/protected` endpoints will respond with a failure status of 401 or 403, respectively. In the body of this response, we'll print either the value returned from the lambdas, `Go Away!!!` or nothing. The only difference between these two is that the former sets the status code explicitly using the `Response` object while the latter uses Spark's `halt` function. This is done to allow access to the private sub-URL when access is permitted; `halt`, on the other hand, immediately stops the request within the filer or route used, terminating the request.

Then, we have this interesting route:

```
get("/users/:name", (request, response) -> "Selected user: " + request.params(":name"));
```

In their most basic form, these two functions allow a user to login and visit a page which serves news items. The `get` request for `/users/:name` allows a client to provide a username (e.g., by requesting `http://localhost:4567/users/bob`) to login as a specified user, who may have certain settings and preferences saved on the server. It will also serve text reading `Selected user: bob`. The username, `bob`, is fetched from the URL using Spark's `param` method on the `Request` object, allowing you to build dynamic URLs and get their values at run-time.

Then, we have a route that changes the response type:

```
get("/news/:section", (request, response) -> {
    response.type("text/xml");
    return "<?xml version=\"1.0\" encoding=\"UTF-8\"?><news>" + request.params("section") \
+ "</news>";
});
```

This snippet also uses the `params` method. It also shows how the response type can be changed using the `Response` object's `type` method. When the user calls an HTTP GET on the `/news` endpoint for some section (such as `http://localhost:4567/news/US`), the API will return an XML document. To ensure that the user agent handles it properly by setting the `Content-Type` HTTP header with Spark's `type` method.

The last route shows how we can use Spark's `Response` object to create a 302 redirect from within the lambda that gets dispatched when the `/redirect` endpoint is hit:

```
get("/redirect", (request, response) -> {
    response.redirect("/news/world");
    return null;
});
```

2.3 Templatized Responses

Spark can also render views using a half a dozen templating engines, including Velocity, Freemarker, and Mustache. With a templated response, you wire up the route a little different than in the examples above. Here is a basic sample using Velocity:

```
public static void main(String args[])
{
    get("/hello", (request, response) -> {
        Map<String, Object> model = new HashMap<>();
        Map<String, String> data = new HashMap<>();

        data.put("message", "Hello Velocity World");
        data.put("att2", "Another attribute just to make sure it really works");

        model.put("data", data);
        model.put("title", "Example 07");

        return new ModelAndView(model, "hello.vm");
    }, new VelocityTemplateEngine());
}
```

In this sample, we route the path `/hello` to a lambda function (as before). This time though, we also pass in a new third parameter to the `get` method: a `VelocityTemplateEngine` object. This will produce the final output from a `ModelAndView` object that our lambda returns. This model contains a `Map` of data that will be used in the template, and the name of the template to use. This causes the model data and the following template to be renders:

```
<html>
<head>
    <title>$title</title>
</head>
<body>
    <h1>Example 07</h1>
    <h2>Velocity Example</h2>

    <p>Variables given from controller in the model:</p>

    <dl>
```

```
        #foreach ($e in $data.entrySet())
            <dt>$e.key</dt>
                <dd>$e.value</dd>
            #end
    </dl>
</body>
</html>
```

2.4 Spark Compared

Let's look at another, admittedly contrived, example and compare it to the syntax of Google Go, which we recently described in another walk through. Let's pull out those two routes from the last example where we varied the response based on the HTTP method used. With this, we have not only the GET method routed, but also POST which appends the request's body to the response:

```
import static spark.Spark.*;

public class HelloWorld2 {
    public static void main(String[] args) {
        get("/hello", (request, response) -> "Hello World!");

        post("/hello", (request, response) -> "Hello World: " + request.body());
    }
}
```

This is an extremely simple service, but complexity isn't the point, so bear with us. This time, when a user connects to the endpoint on localhost using an HTTP GET or POST (i.e. `http://localhost:4567/hello`), Spark will respond; otherwise, they'll receive an error status code of 404, not found . With Spark, the code is very concise — a quality that cannot be touted enough. Not only will this make it easier to debug, it will make it easier to maintain and evolve. To underscore this, let's compare this to a similar example in Golang using mux:

```
import (
    "io"
    "io/ioutil"
    "net/http"
    "github.com/gorilla/mux"
)

func main() {
    r := mux.NewRouter()

    r.HandleFunc("/hello", func (w http.ResponseWriter, r *http.Request) {
```

```
12          io.WriteString(w, "Hello, World!")
13      }).Methods("GET")
14
15      r.HandleFunc("/hello", func (w http.ResponseWriter, r *http.Request) {
16          body, _ := ioutil.ReadAll(r.Body)
17          io.WriteString(w, "Hello world: ")
18          w.Write(body)
19      }).Methods("POST")
20
21      http.ListenAndServe(":4567", r)
22  }
```

Looking at the two, which are identical in functionality, the Go version is certainly more code. They are both simple, but the real difference is in the **readability**. The latter example in Golang uses a syntax that might be cryptic and foreign to some programmers. On the other hand, the Spark example simply organizes a "request, response" relationship using a Domain Specific Language designed for this exact purpose. For this basic example, we don't really need to use mux. Go has built-in functionality that would be much less code if we also dropped the use of Spark from the Java example. When pitting the two languages against each other, sans toolkits, the Go example is far less complex. The point though is that Spark gives you a fluid way of setting up routes that is very approachable — even for a novice. In an upcoming post, we'll also show you how you scale the use of this toolkit as your API's codebase grows.

2.5 Benefits of Spark

Apparent from even these relatively simple examples, we've already discovered that Spark has some great benefits when building APIs, including:

- **Speed**: Spark is a thin wrapper around Java EE's Servlet API, which ranks very high on industry benchmarks where not only Java is tested but many programming languages and toolkits.
- **Productivity**: Spark aims to make you more productive, giving you a simple DSL for routing your API's endpoints to handlers.
- **Purpose Built**: Spark does not come with any bells or whistles. It is designed to do one thing and one thing very well — routing.
- **Cloud Ready**: Spark is a great, lightweight alternative to other, heavier frameworks, making it perfect for applications throughout the entire cloud stack.

Utilizing Java's familiar syntax and raw power is definitely an effective choice when launching your API. When you combine this with the benefits of Spark, you get a powerful pair that will make you **more productive**. Continue reading to see how to use this JVM programming language with Spark. We'll use that language to extend these basic examples to include **Controllers** and **Dependency Injection (DI)**.

3. Introducing Kotlin to the JVM

In this section, we'll explain why the **JVM** provides a strong basis on which to run your APIs, and how to simplify their construction using a framework called **Spark** and a new programming language called **Kotlin**. In this first part, we will introduce you to the JVM and how it can execute code written in many programming languages, including Kotlin. We'll discuss a number of Kotlin's features and show them in the context of APIs.

In the next chapter, we'll explore the Spark framework in detail, and how to complement this toolkit with additional components like Inversion of Control (IoC). During these chapters, we'll develop a boilerplate that you can use to **create robust APIs with Kotlin and Spark**. The code can be found on Github, so refer to it there as you read through the sample code below.

3.1 Introducing the JVM

The **Java Virtual Machine (JVM)** is a runtime that offers a range of language choices. Adoption of Clojure, Groovy, Scala, Jython, and other JVM-based languages are widespread. In addition to the wide range of language choices, another uncommon aspect of the JVM is its openness. You can get an implementation of the JVM not just from Oracle but also from an expansive list of other providers. Many distributions of Linux come preloaded with OpenJDK, which is the reference implementation of Java, allowing API developers to host their service with very little cost and hassle.

One of the core tenants of Java is that each new version is **backward compatible** with previous ones. This has caused the Java programming language to evolve more slowly than others like C#, where major changes in the language come with massive upgrade efforts.

This has improved a lot over the years, and Java 8 has removed a lot of cruft in the language with the introduction of **lambdas**. Regardless of the pace of Java's evolution, this commitment to backward compatibility has resulted in a **very stable base**. The ongoing compatibility of the JVM offers a number of very compelling reasons to build APIs on this platform:

- **Return on Investment (ROI)**: Code that continues to run can continue to produce a return on the investment made to write it.
- **Knowledge remains relevant**: Since old code continues to run, all the knowledge gained to write that code continues to be useful. With new versions of the JVM, you can keep writing old-style code. You don't have to retrain yourself when the runtime's vendor upgrades the platform.
- **Ecosystem**: These factors have caused a very large ecosystem to flourish around the JVM. This system includes massive corporations and some of the biggest open source communities that will work hard to ensure its longevity.

Probably one of the biggest drawbacks of selecting Java to code up your API is that it is ***so*** verbose. Who would use it if they didn't have to?! ROI is often enough of a reason for companies to select it even for greenfield work, putting programmers through the pains of its verbosity even with the advent of alternatives like Node.js and Golang.

But what if you had the most sugary syntax you could ever dream of without having to change anything else? What if you could retain your build system, packaging, dependencies, existing code and performance, *and* get the language you've been longing for?

3.2 Enter Kotlin

Kotlin is a new programming language from Jetbrains, the makers of some of the best programming tools in the business, including Intellij IDEA, an IDE for Java and other languages. As a leader in the Java community, Jetbrains understands the pains of Java and the many benefits of the JVM. Not wanting to throw the baby out with the bathwater, Jetbrains designed Kotlin to run on the JVM to get the best out of that platform.

They didn't constrain it to this runtime, however — developers can also compile their Kotlin code into **JavaScript**, allowing it to be used in environments like **Node.js**. To work with untyped and typed environments like these, Kotlin is statically typed by default, but allows developers to define dynamic types as well. This balance provides great power and many opportunities.

3.3 Why Kotlin?

Mike Hearn does a spectacular job explaining why you should consider using Kotlin. In his article, he lists the following reasons:

- Kotlin compiles to JVM bytecode and JavaScript
- Kotlin comes from industry, not academia
- Kotlin is open source and costs nothing to adopt
- Kotlin programs can use existing Java or JavaScript frameworks
- The learning curve is very low
- Kotlin doesn't require any particular style of programming (OO or functional)
- Kotlin can be used in Android development in addition to others where Java and JavaScript work
- There is already excellent IDE support (Intellij and Eclipse)
- Kotlin is highly suitable for enterprise Java shops
- It has the strong commercial support of an established software company

Read through Mike's article for more on the rationale for using Kotlin. Hopefully though, this list is enough to convince you to seriously consider Kotlin for your next API. To see how easy this can be, we'll explain how to use Kotlin with a micro-services framework called Spark. We'll delve deep into Spark in the next part of this series, but now we'll explain some of the great Kotlin language features with some sample code.

3.4 Examples of Kotlin's Syntax

To see how Kotlin can be used to create killer APIs, we'll walk through a sample that demonstrates many of the language's features in the context of APIs. You can find the entire code on Github, and it's all open source.

To start, here's the API's entry point:

```
fun main(args: Array<String>) = api(composer = ContainerComposer())
{
        route(
                path("/login", to = LoginController::class, renderWith = "login.vm"),
                path("/authorize", to = AuthorizeController::class, renderWith = "authorize.vm"),
                path("/token", to = TokenController::class))
}
```

Note that what we have is very readable even with zero knowledge of Kotlin. **This snippet starts an API that exposes various paths that are routed to controllers, some of which are rendered by a template and others that are not.** Easy. Kotlin lends itself to creating fluent APIs like this one, making for very readable code and approachable frameworks.

3.5 Functions and Constructors

To create our Domain Specific Language (DSL) for hosting APIs, we're using several of Kotlin's syntactic novelties and language features. Firstly:

- **Constructors**: In Kotlin, you do not use the `new` keyword to instantiate objects; instead, you use the class name together with parenthesis, as if you were invoking the class as a function (a la Python). In the above snippet, a new `ContainerComposer` object is being created by invoking its default constructor.
- **Functions**: Functions, which begin with the `fun` keyword, can be defined in a class or outside of one (like we did above). This syntax means that we can write Object Oriented (OO) code or not. This will give you more options and potentially remove lots of boilerplate classes.
- **Single expression functions**: The fluent API allows us to wire up all our routes in a single expression. When a function consists of a single expression like this, we can drop the curly braces and specify the body of our function after an equals symbol.
- **Omitting the return type**: When a function does not return a value (i.e., when it's "void"), the return type is `Unit`. In Kotlin, you do not have to specify this; it's implied.

If we weren't to use these last two features, the slightly more verbose version of our main function would be this:

```
fun main(args: Array<String>) : Unit {
    // ...
}
```

The difference is only a few characters, but the result is less noisy.

3.6 Named Arguments and Passing Function Literals

The next part of this main method is the call to the `api` function. This function uses a few other interesting features:

- **Named arguments**: When calling the `path` function, we're specifying certain arguments by name (e.g., `to` and `renderWith`). Doing so makes the code more fluid.
- **Passing function literals outside the argument list**: When a function, like our `api` function, expects an argument that is itself a function, you can pass it after closing the argument list. This cleans up the code a lot. To see what we mean by this, observe the `api` function definition:

```
fun api(composer: Composable, routes: () -> List<Application.RouteData<Controllable>>) {
    // ...
}
```

This function takes two arguments:

1. The composer (more on that in the next part of this series)
2. The routes as a function that takes no arguments and produces a list

The second argument is the interesting one. It is a lambda expression. In Kotlin, **lambdas** are written as `() -> T` where the parentheses contain the possibly-empty list of arguments and their types, an arrow symbol (i.e., `->` or "produces"), and the return type. Our `api` function uses this syntax to define it's last argument. In Kotlin, when the last argument of a function is also a function, we can pass the lambda expression *outside* of the call. Without this capability, calling the `api` method would look like this:

```
api(ContainerComposer(), {
        // ...
})
```

See the difference? If you're a jQuery programmer, you're probably numb to it. Have a look at this contrived example that shows the difference more clearly:

```
a({
    b({
            c({

            })
    })
})
```

See it now? When a method takes a function literal as an argument, you end of with a grotesque interchange of parentheses and braces. Instead, Kotlin allows us to pass the body of the function literal after the method call. JQuery programmers unite! This is your way out of the flip-flopping trap of delimiters you've found yourself in! Instead, Kotlin allows you write this sort of oscillation like this:

```
a()
{
    b()
    {
            c()
    }
}
```

So clear. So easy to understand. This convention is what we're using in the call to the `api` function. Our routes are defined in a lambda that returns a list of data objects (which we'll explain shortly). The result is half a dozen lines of code that start a Web server which hosts our API, sets up three routes, maps them to controllers, and associates templates to certain endpoints. That's the powerful triad we mentioned early — **language, runtime, and JVM-based frameworks** — that allow us to quickly start serving up APIs!

3.7 Generics

As you saw in the definition of the of the `api` function, Kotlin has support for **generics** (i.e., function templates). In Kotlin, generics aren't confusing like they are in Java. The thing that makes Java's generics so tricky is wildcard parameterized types. These were needed to ensure backward compatibility with older Java code. Kotlin doesn't have wildcards, making it much easier to use generics in this new language. In fact, generics are so easy for Java and C# developers that there's very little to say.

One aspect of Kotlin's generics that may be unclear at first is the use of the `out` annotation on some template parameters. This is seen in the definition of the `RouteData` class included in the sample code:

```
data class RouteData<out T : Controllable>(
    val path: String,
    val controllerClass: Class<out T>,
    val template: String? = null)
```

This class is parameterized along with the constructor argument; `controllerClass`. Both have this extra modifier on `T` though. What does this mean? It means that the output type `T` will not be passed in as input; instead, the `RouteData` class will only produce objects of type `T` as output. This is what is referred to as declaration-site variance. Because the `RouteData` class does not consume any values of type `T` but only produces them as return values, the compiler knows that it is safe for any class that extends `Controllable` to be used as a parameter value, since they can always safely be upcast at runtime to the `Controllable` interface.

If `T` were not an output-only type, a potentially unsafe downcast would be required at run-time. By providing this extra bit of info to the compiler, we can avoid wildcards (i.e., something like `RouteData<? extends Controllable>` in Java) and end up with safer, more easy-to-use generics.

3.8 Data Classes

The `data` annotation is another interesting feature of Kotlin that we're using in the `RouteData` class. This modifier tells the compiler that objects of this type will only hold data. This allows it to synthesize a bunch of boilerplate code for us. Specifically, the compiler will implement the following methods for us:

- `equals`
- `hashCode`
- `toString`
- `copy`; and
- Getters and setters for each constructor parameter that is marked with `val` (i.e., read-only variables).

After we define a data class we can also use it to declare and set the values of multiple variables simultaneously like this:

```
val (path, controllerClass, template) = routeData
```

Given a data object of type `RouteData`, we can sort of pull it apart into constituent parts, where each property of the class is assigned to respective variables. This can help you write self-documenting code, and can also be used to return more than one value from a function.

Multiple Return Values from Functions

One attractive language feature of Golang is its syntax that allows developers to define multiple return values for a function. In Go, you can write very descriptive code like this:

```
func (file *File) Write(b []byte) (n int, err error) {
    // ...
}
```

This says that the function `Write` returns two values: an integer called `n` and `err`, an `error` object. This unique language construct obviates the need to pass references to objects that the function will modify when an additional return value is also needed. This is what one has to do in Java, which you can see in a Java-based version of our `Controllable` type:

```
interface Controllable {
        default boolean get(Request request, Response response,
            final Map<String, Object> model) { return true; }

        default boolean post(Request request, Response response,
            final Map<String, Object> model) { return true; }

        default boolean put(Request request, Response response,
            final Map<String, Object> model) { return true; }

        // ...
}
```

We want to return two things from `get`, `post`, `put`, etc. — a Boolean flag that will be used to abort routing if `false`, and the model that contains the template values (if templating is used by the view). Here the Java language is actually working against us, forcing us to write unclear code. Google Go would help us write this more clearly, and so does Kotlin.

In Kotlin, our `Controllable` type is defined like this:

```
abstract class Controllable {
        public open fun get(request: Request, response: Response):
            ControllerResult = ControllerResult()

        public open fun post(request: Request, response: Response):
            ControllerResult = ControllerResult()

        public open fun put(request: Request, response: Response):
            ControllerResult = ControllerResult()

        // ...
}
```

Kotlin allows us to say that our controller's methods return a `ControllerResult` type. Like `RouteData` described above, `ControllerResult` is a data class:

```
data class ControllerResult(
            val continueProcessing: Boolean = true,
            val model: Map<String, Any> = emptyMap())
```

With this one line, we can easily define a result type that we can use in the `Router` to control the flow of processing requests and provide views with model data.

Big deal, you may say. Create a class in Java and do the same. Sure. With Java though, our `ControllerResult` class becomes 50 annoying lines of code after we implement `equals`, `toString`, `hashCode`, a copy constructor, getters, and setters. As we described above, data classes include all these in **one line of code**. Take that in like a cool refreshing drink after a long day's work in the hot summer sun!

Using data classes, our controllers can override methods in the `Controllable` class like this:

```
public override fun get(request: Request, response: Response): ControllerResult =
    ControllerResult(model = mapOf(
            "user" to request.session(false).attribute("username"),
            "data" to mapOf(
                    "e1" to "e1 value",
                    "e2" to "e2 value",
                    "e3" to "e3 value")))
```

Using the `to` keyword and a read-only map produced by the standard Kotlin library's `mapOf` function, we end up with a very succinct yet readable implementation of this method. Refer to the Kotlin docs for more info about creating maps with the `to` keyword, the `mapOf` function, and overriding methods defined in a base class. For now though, let's see how we can use the `ControllerResult` objects in a safe and intuitive manner using one of the coolest Kotlin language features.

3.9 Smart Casts

As we'll discuss more in the next part of this blog series, the methods of our Controllers are invoked dynamically depending on the HTTP method used by the client. This is done in the `Router` class like this:

```
val httpMethod = request.requestMethod().toLowerCase()
val method = controllerClass.getMethod(httpMethod, Request::class.java,
    Response::class.java)
val result = method.invoke(controller, request, response)

if (result is ControllerResult && result.continueProcessing)
{
   controller.after(request, response)

   model = result.model
```

In the first three lines, we are using reflection to invoke a method on the controller that has the same name as the HTTP method specified by our API consumer. `Invoke` is defined in the Java runtime to return an `Object` which Kotlin represents as `Any`. This is helpful because two of our `Controllable`'s methods, `before` and `after`, do not return `ControllerResult` objects while the bulk of them do. Using Kotlin's smart casting language feature, we can write **very clear and safe code** to handle this discrepancy.

In the snippet above, we check at run time to see if the `Any` object is of type `ControllerResult`. This is done using Kotlin's is operator in the if statement. If it is, we also check to see if the data object's `continueProcessing` property returns `true`. We do this *without* casting it. On the right side of the logical `&&` operator, Kotlin treats the `result` object not as type `Any` but as type `ControllerResult`. Without having to write code to perform the cast, we can access the object's `ContinueProcessing` property. Without smart casting and without properties, we'd have to write this conditional statement verbosely in Java like this:

```
if (result instanceof ControllerResult) {
        ControllerResult controllerResult = (ControllerResult)result;

        if (controllerResult.getContinueProcessing()) {
           // ...
        }
}
```

Even with C#'s `as` operator, we end up with code that isn't as clear as Kotlin's. In C#, we'd have to write this:

```
ControllerResult controllerResult = result as ControllerResult

if (controllerResult != null && controllerResult.continueProcessing) {
    // ...
}
```

Kotlin lets us avoid all this confusing, verbose code. This will ensure that our intent is more clear, and help us find and fix bugs in our APIs more quickly.

3.10 Using Kotlin

Because of its interoperability with other tools in the Java ecosystem, it is very easy to start using Kotlin. You can leverage all of the same tools and knowledge that you are using today in your Java development. Building with Ant or Maven? Those work with Kotlin. Writing code in Intellij IDEA, Eclipse, or Sublime? Keep using them!

To get started using your favorite tools with Kotlin, here's some helpful links:

- Intellij IDEA
- Kotlin Sublime Text 2 Package
- Eclipse
- Maven usage guide
- Using Gradle with Kotlin
- Ant integration

3.11 Converting Existing Code

When you start using Kotlin, it can help to convert an existing Java project to this new language. Then, you can see how familiar code looks using Kotlin. To convert existing code, you have to use Intellij, Eclipse, or (if you only have a snippet) the Kotlin on-line console. In the end, you will probably rewrite a lot of the converted code to use more idioms and Kotlinic styles, but converting existing code is a great starting point.

3.12 Compiling Kotlin

Whether you convert your code or write it from scratch, you are going to need to compile it. The IDE plug-ins make this easy while you're working. In order to use this language in larger projects, however, you are going to need to automate those builds. This will require integration with Maven, Ant, or Gradle. Jetbrains and the Kotlin community provides these out of the box, and usage is as you would expect. For an existing Java-based project using one of these, switching to Kotlin can be done by replacing just a few lines in your build script. For instance, the sample project accompanying this series was converted to compile Kotlin code instead of Java with just a few dozen lines.

3.13 Debugging

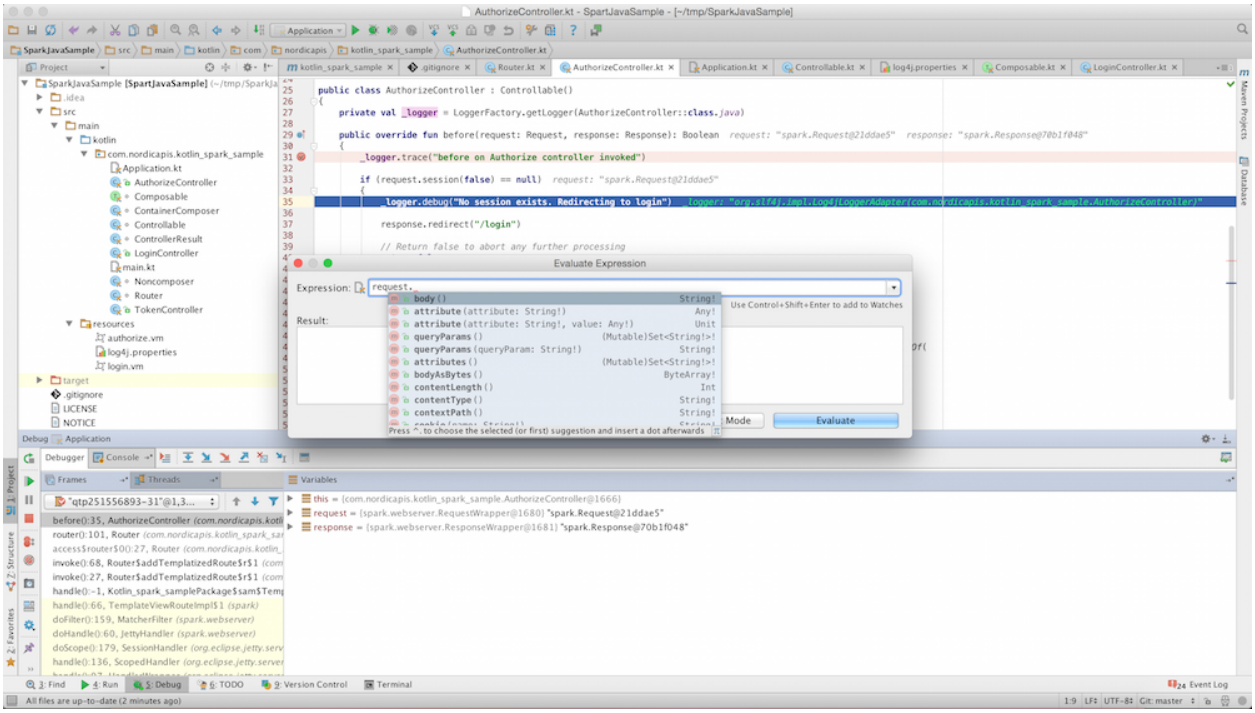
Compiling code isn't necessarily working code. To make your software functional, you can use one of the many JVM-based logging frameworks to print out debug statements. You can see this in the sample project that uses SLF4J and Log4j to print statements about API requests and responses. Here's an indicative sample from the `AuthorizationController`:

```
public class AuthorizeController : Controllable()
{
        private val _logger = LoggerFactory.getLogger(AuthorizeController::class.java)

        public override fun before(request: Request, response: Response): Boolean
        {
            _logger.trace("before on Authorize controller invoked")

            if (request.session(false) == null)
            {
                _logger.debug("No session exists. Redirecting to login")

                response.redirect("/login")

                // Return false to abort any further processing
                return false
            }

            _logger.debug("Session exists")

            return true
        }

        // ...
}
```

A few things to note about that snippet before we move on:

- It's normal SLF4J stuff, so there's no learning curve for Java devs.
- The SLF4J `LoggerFactory` requires a static class to initialize it. This is a Java class, so we use Kotlin's `::class` syntax to get a `KClass` object and then call its `java` extension property. Voila! Java interop :)

Basic instrumentation will only take you so far; sometimes you need to roll up your sleeves and dive into the running code though. You can do this *easily* in Intellij and Eclipse using those tools' debuggers just as you would with your Java code. In these environments, you have all the same tools you have come to rely on when debugging other JVM-based code. For instance, the **call stack**, list of **running threads**, **expression evaluation**, and of course **breakpoints**, are all available to you! Here's a screenshot of the above code running in Intellij with the debugger attached:



Finding bugs will be *easier* with that. This is a *very* compelling example of why Kotlin is a better choice for creating large-scale APIs than some other languages (e.g., Go). (Golang can be debugged using GDB, but that tool is not nearly as user-friendly as Intellij's and Eclipse's debuggers.)

3.14 Conclusion

Unlike other programming environments, the JVM gives you both language and vendor choices that can help you create incredible API. On the JVM you can choose between **dynamic** languages like Groovy, **functional** ones like Scala and Clojure, and now another rival — Kotlin. We talked through a number of reasons to use this language, and showed some of the features this new JVM upstart delivers.

All this is great, you may be saying, but what's the catch? There are some drawbacks, but they are minor. The strongest argument against Kotlin is its immaturity — it isn't at v1 yet. This is a problem that will inevitably be fixed with a bit more time. Even now at milestone 12, the compiler gives you warnings about the use of deprecated language features that will not be included in the initial release. This makes it easy to pivot and prepare.

In the next part of this series, we'll delve deeper into Spark and explain it more using the sample Kotlin code we started developing in this post.

4. Building APIs on the JVM using Kotlin and Spark

If you are building APIs or microservices on the **Java Virtual Machine** (JVM), you owe it to yourself to check out the micro-framework Spark. This tiny toolkit is designed after a similar Ruby framework called Sinatra, providing a library that makes it easy to create APIs and web sites. Much of this simplicity comes from its use of new language features introduced in Java 8, like lambdas, which give programmers an elegant way to define their APIs.

In this second chapter on using Kotlin, a new programming language from Jetbrains, we will show you how you can make the programming model even sweeter using this new JVM language. We'll use it to build some additional components you will need to **create truly great APIs using Spark**. Building off the previous Spark intro, the new components we'll create in this part of our series will give you a useful starting point to leverage Spark in your APIs, while demonstrating the potential and power of Kotlin.

4.1 Recapped Intro to Spark

In our introduction to Spark, we explain that Spark is a toolkit that you link into your API to **define and dispatch routes** to functions that handle requests made to your API's endpoints (i.e., a multiplexer or router). It is designed to make the definition of these routes quick and easy. Because Spark is written in Java and that is its target language, it provides a simple means to do this using Java 8's lambdas. It also does not rely on annotations or XML configuration files like some comparable frameworks do, making it easier to get going.

A typical Hello World example (which we also included in our Spark intro) is this:

```
1  import static spark.Spark.*;
2
3  public class HelloWorld {
4          public static void main(String[] args) {
5                  get("/hello", (request, response) -> "Hello World");
6          }
7  }
```

> Note that this snippet is Java and not Kotlin; we'll get to our Kotlin version in a bit.

When executed, Spark will start a Web server that will serve up our API. Compile, run this, and surf to `http://localhost:4567/hello`. When hit, Spark will call the lambda mapped with the `spark.Spark.get` method. This will result in a dial tone.

4.2 Building a Robust API with Spark

From our intro (which includes a number of links to additional docs and samples), you can see that Spark is a useful framework on its own. Purpose built to perform routing, it can start a Web server and can be plugged with certain templating engines. These are the only bells and whistles you'll get with it, though. When you begin using it to build a production-caliber API, you will need more.

Before going live with your API, you will probably write a lot of code. To make this evolvable, testable, and supportable, you will also need:

- **Controllers**: Spark gives you a way to model your data and views to present it, but there's no concept of controllers. You will need these if you're going to follow the Model View Controller (MVC) pattern (which you should).
- **Dependency Injection (DI)**: To make your API more modular and robust, you will need a way to invert the resolution of dependencies, allowing them to be easily swapped in tests. Spark doesn't provide out-of-the-box integration with any particular Dependency Injection (DI) framework.

- **Localization**: Spark makes it easy to define views using any number of templating engines, but resolving message IDs is beyond what the framework provides. You will need this if you are targeting a global market with your API.
- **Server Independence**: Spark starts Jetty by default. If you want to use another server or different version of Jetty, you will have to do some additional work. This is trivial if you are distributing your app as a WAR, but you will have to write some code if this isn't the case.

If you take Spark, its support for various template languages, and add these things, you have a very complete toolset for building APIs. If you wrap all this up in a fluent API using Kotlin and utilize this language to build your controllers, you will be able to rapidly develop your service.

We won't show you how to extend Spark with all of the above, but we will delve into the first two. Send a pull request with your ideas on how to support other features.

> It should be noted that there are other frameworks for building APIs that already include these (e.g., Spring Boot). With these, however, the third-party libraries used (if any), and how these features are provided, is already set. If you would like to use a different DI framework, for example, it may not be possible. If it is, you may incur bloat as the provided DI framework is not needed in your case. With Spark, these decisions are yours to make (for better or worse).

4.3 Templating in Kotlin

We explain Spark's templating support in our intro. We talk about how the toolkit comes with support for a half-dozen template engines that you can use to render responses. In this post, we want to show you how this can be done with the Kotlin-based sample we've been building. We use Spark's same object model, but wrap in our own fluent API that makes the syntax *a lot* cleaner. With this sugary version, rendering a response with a template is nearly the same as without. You can see this in the service's entry point where we fluently define all the routes:

```
1  route(
2          path("/login", to = LoginController::class, renderWith = "login.vm"),
3          path("/authorize", to = AuthorizeController::class, renderWith = "authorize.vm"),
4          path("/token", to = TokenController::class))
```

Here we are passing the name of the template in the `renderWith` named argument. You can read more about the `path` function in the first part of this series, but the important part to note here is that, in contrast to our simple Java-based templating sample, the data model is not mixed up in the definition of the routes — that is left to the controllers. At this point, we are only defining which template should be used with which route.

> You can also define this type of syntactic sugar in Java. The Kotlin sample was originally written in that language and converted using the Jetbrains-provided

> tools. Before it was Kotlinized, the Spark API was wrapped up in more convenient API that fit our usage model better. You can see the old Java version in the GitHub history, but suffice it to say that the Kotlin version is a lot cleaner.

4.4 Adding DI Capabilities to a Spark-based API

To implement DI in your API, you can use various frameworks, including Guice, Pico and the Kotlin-native, Injekt. Whichever you decide on, you will need to integrate them with Spark. In this subsection, we will walk you through this using **Pico**.

> It is beyond the scope of this article to introduce DI. If you are unaware of how this pattern works, refer to Jacob Jenkov's introductory article on DI.

The Pico integration is handled in our `Application` class which inherits from the `SparkApplication` class. It uses another of our classes called `Router`; the two are what ties Spark and Pico together. The important parts of the `Application` class are shown in the following listing:

```
1  public class Application(
2              var composer: Composable = Noncomposer(),
3              var appContainer: MutablePicoContainer = DefaultPicoContainer(),
4              var routes: () -> List<Application.RouteData<Controllable>>) : SparkApplication
5  {
6          private var router = Router()
7
8          init
9          {
10             composer.composeApplication(appContainer)
11         }
12
13         fun host()
14         {
15             // Explained below...
16         }
17
18         // ...
19 }
```

The `Application` class' constructor takes three arguments:

1. An instance of type `Composable` which defaults to an object that won't do any composition of dependencies (e.g., in simple cases where DI isn't used by the API)
2. A `MutuablePicoContainer` that will house the API's dependencies
3. The lambda function that will produce the routes (as described in part 1 of this series).

To see more clearly how this class wires up Pico and Spark, we need to look at how we compose dependencies. Then we will talk about the `Router` in detail.

Composing Dependencies

With DI, an object does not instantiate its dependencies directly. Instead, object creation is inverted and dependent objects are *provided* — not created. In order to do this, a DI container must be populated with objects and object resolvers. In our sample API boilerplate, this is done through subtypes of the `Composable` interface. A composer registers the API's dependencies, relating an interface to a concrete implementation that should be used by all objects that depend on that interface. Objects are resolved at various levels or scopes, resulting in a hierarchy of object resolution. We can also create factories and add these to the containers; these will produce objects that others depends on, allowing us to do complicated object creation outside of the composers.

As the `Application` object comes to life and the `init` method is called, the first thing it will do is compose the application's dependencies. It does this using the given `Composable`. This interface looks like this:

```
1  interface Composable {
2
3          fun composeApplication(appContainer: MutablePicoContainer) { }
4
5          fun composeRequest(container: MutablePicoContainer) { }
6  }
```

As you can see, composers do two things:

1. Compose the API's application-wide dependencies
2. Compose dependencies that should have request-level scope (i.e. objects that only exist for the lifetime of an HTTP request/response)

The former method, `composeApplication`, is what is called by the `Application` class' `init` method. This method is called once, as the API server is started. The later method, `composeRequest`, is called per request by the `Router` class (described below).

In the composer, you can register dependencies in any way Pico supports. It offers a number of very useful mechanisms that make it a good tool to consider using in your API implementation. While we will not dive into Pico in this post, we will show you a simple implementation of a `Composable` subclass that is included in the sample project:

```
class ContainerComposer : Composable
{
        public override fun composeApplication(appContainer: MutablePicoContainer)
        {
            appContainer.addComponent(javaClass<AuthorizeController>())
            appContainer.addComponent(javaClass<TokenController>())
            appContainer.addComponent(javaClass<LoginController>())
        }

        public override fun composeRequest(container: MutablePicoContainer) { }
}
```

This particular composer is pretty dumb — yours will probably be much more complicated. The important things here are that:

- The controllers are in the application container
- All of their dependencies will be resolved from the application or request container when instances of them are fetched in the `Router`.

This will make it easy to create controllers because their dependencies will be given to them when they spring to life. (This will be made even easier using some reflection that automatically sets up routes, which we'll explain below.)

Resolving Dependencies as Requests are Routed

The `Router` class works with the `Application` class to glue all these frameworks together. As you add support for localization and more advanced things that are not covered in this post, you will find that your version becomes quite intricate. For the sake of this post we'll keep our sample relatively simple.

`Router` inherits from `SparkBase`, so that it can gain access to its protected `addRoute` method. This low-level Spark API is what is called by the higher-level static methods, `get`, `post`, etc., which were discussed in our Spark intro and shown in the Hello World listing above. We don't use those — instead we use our own fluent API that ends up invoking this lower-level Spark interface. Our `Router` exposes one public method, `routeTo` which you can see here:

```
class Router constructor() : SparkBase()
{
        public fun <T : Controllable> routeTo(
            path: String, container: PicoContainer, controllerClass: Class<T>,
            composer: Composable, template: String? = null)
        {
            // ...
        }
}
```

The `routeTo` method is called in the `Application` class for each route that is setup with our DSL. You can see this in the `host` method of `Application` (which was elided from the above listing of that class):

```
fun host()
{
        var routes = routes.invoke() // Invoke the lambda that produces all the routes

        for (routeData in routes)
        {
            val (path, controllerClass, template) = routeData

            router.routeTo(path, appContainer, controllerClass, composer, template)
        }
}
```

> Refer to the previous installment of this series for information about the `routeData` data class and how its being used to simultaneously assign multiple values.

When `routeTo` is called like this, it does two important things:

1. Reflectively looks for the methods defined in the given controller to determine what HTTP methods should be routed; and
2. Tests to see if a template has been assigned to the route, and, if so, calls different overloads of the `SparkBase` class to register the route appropriately.

You can see this in the source code more clearly, but the important part is shown below. Note how two private methods — `addRoute` and `addTemplatizedRoute` — are called for each method on the controller which are found using reflection.

```
if (template == null || template.isBlank()) {
        addRoute(methodName, path, container, controllerClass, composer)
}
else {
        addTemplatizedRoute(methodName, template, path, container, controllerClass, composer)
}
```

> Note here that we used a smartcast to convert the nullable `template` variable to a string after first checking if it is null. This is one of Kotlin's coolest features.

Regardless of whether or not a template should be used, both of these private methods create a Spark `RouteImpl` instance. To do this, a closure must be instantiated and passed to Spark's `addRoute` method. In the case of a templatized route, the closure and `RouteImpl` are created like this:

```
val r = fun (request: Request, response: Response): ModelAndView
{
        var model = router(request, response, container, controllerClass, composer)

        return ModelAndView(model, template)
}

SparkBase.addRoute(httpMethod, TemplateViewRouteImpl.create(path, r, VelocityTemplateEngin\
e()))
```

> We do this similarly in the method that handles the non-templatized case (which can be seen in the source repository).

The part to note there is that the `Router`'s private `route` method is called in the closure, `r`. This function pointer, and thus `route`, gets called with every request. This is how we can integrate DI at the request level.

The `route` method starts by creating a new request-level container that has the application-wide container as its parent. This will cause dependencies to be resolved firstly from the request-level child container. Only if they aren't found there will the parent be investigated. (This is that hierarchical dependency resolution we alluded to above.) Then, we call the composer's `composeRequest` method, passing in this new container. Once composition is done, we fetch the controller from the container, and invoke it.

You can see this in the following snippet:

```
private fun <T: Controllable> router(request: Request, response: Response, appContainer: P\
icoContainer,
                                            controllerClass: Class<T>, composer: Composable) : Ma\
p<String, Any>
{
   val requestContainer = DefaultPicoContainer(appContainer)
   var model : Map<String, Any> = emptyMap()

   composer.composeRequest(requestContainer)

   try
   {
           val controller = requestContainer.getComponent(controllerClass)

           // ...
   }
   catch (e: Exception)
   {
           halt(500, "Server Error")
   }

   return model
}
```

We will return to this method a bit later when we discuss the controllers, but this gives you a good overview of how to integrate a DI framework like Pico with Spark. For more details, review the source or leave a comment below.

4.5 Implementing API Logic in Controllers

As you build out your API, you are very likely to have dozens, hundreds, or even thousands of **endpoints**. Each of these will have a different **logic** — validating inputs, calling back-end services, looking up info in a data store — the list goes on. This processing has to be done in an orderly manner or else your code base will become **unmaintainable**. To avoid this, your API's logic should be encapsulated in one controller per endpoint.

With Spark, you get a routing system that dispatches to functions. You can use lambdas as in the samples, but this becomes untenable as the size of your API grows. Once you're past the prototype phase, you'll realize this is not enough. There are many ways to add this capability on top of Spark, and this is what makes it such a great framework. As with DI, you are free to choose the way that works best for you (for better or worse). In this post, we will offer you one suggestion that will satisfy these goals:

- It should be fast and easy to create controllers.
- Programmers should not need to focus on how routing is done as they build controllers.

- All of a controller's dependencies should be injected using constructor injection.
- A controller should not be cluttered with a bunch of noisy annotations.

With these goals in mind, we start with the `Controllable` type (which we touched on in the last post). Every controller within our API will inherit from this class.

```
1  abstract class Controllable
2  {
3          public open fun before(request: Request, response: Response): Boolean = true
4          public open fun get(request: Request, response: Response): ControllerResult = ControllerR\
5  esult()
6          public open fun post(request: Request, response: Response): ControllerResult = Controller\
7  Result()
8          public open fun put(request: Request, response: Response): ControllerResult = ControllerR\
9  esult()
10         public open fun delete(request: Request, response: Response): ControllerResult = Controll\
11 erResult()
12         public open fun patch(request: Request, response: Response): ControllerResult = Controlle\
13 rResult()
14         public open fun head(request: Request, response: Response): ControllerResult = Controller\
15 Result()
16         public open fun trace(request: Request, response: Response): ControllerResult = Controlle\
17 rResult()
18         public open fun connect(request: Request, response: Response): ControllerResult = Control\
19 lerResult()
20         public open fun options(request: Request): ControllerResult = ControllerResult()
21         public open fun after(request: Request, response: Response) { }
22 }
```

Extending this type does not require the subclass to override *any* methods. In practice, this wouldn't happen, as that would mean that no routes would be set up and the server wouldn't respond to requests. The point though is that controllers only need to implement the actions they require — no others, making it simple and fast to implement one.

In the sample code, we have defined three controllers that simulate the logic of the OAuth code flow:

1. `AuthorizeController`
2. `LoginController`
3. `TokenController`

If you are not familiar with how this message exchange works, we will *briefly* explain:

To achieve our goal of keeping it simple to create these controllers, and to not burden the programmer with routes and annotations, we use reflection to discover which of the `Controllable` class' methods have been overridden. This is done in the `Router` just before it calls Spark's `addRoute` method (described above):

```
public fun <T : Controllable> routeTo(path: String, container: PicoContainer, controllerCl\
ass: Class<T>,
                                                composer: Composable, template: String? = null)
{
        for (classMethod in controllerClass.getDeclaredMethods())
        {
            val methodName = classMethod.getName()

            for (interfaceMethod in javaClass<Controllable>().getMethods())
            {
                if (methodName == interfaceMethod.getName() && // method names match?
                        classMethod.getReturnType() == interfaceMethod.getReturnType() && // meth\
od return the same type?
                        Arrays.deepEquals(classMethod.getParameterTypes(), interfaceMethod.getPar\
ameterTypes())) // Params match?
                {
                    // Call templatized or non-templatized version of Spark's addRoute method (sh\
own above) to
                    // get route wired up

                    break
                }
            }
        }
}
```

This is taking the controller class we passed to the `path` method (in the API's entry point), and checking each of its methods to see if the name, return type, and parameter types match any of those defined in the `Controllable` base class. If so, a route is setup, causing that method to be called when the path is requested with the right action.

> To use Java reflection from Kotlin like this, you need to ensure that you have `kotlin-reflect.jar` in your classpath (in addition to `kotlin-runtime.jar`). If you are , add this dependency to your POM like this:
>
> ‘‘‘‘xml
>
> ‘‘‘‘

To make this more concrete, let's look at the `AuthorizeController` which is the first one called in our simplified OAuth flow:

```
public class AuthorizeController : Controllable()
{
        public override fun before(request: Request, response: Response): Boolean
        {
            if (request.session(false) == null)
            {
                // No session exists. Redirect to login
                response.redirect("/login")

                // Return false to abort any further processing
                return false
            }

            return true
        }

        // ...
}
```

The important part here is the `before` method, which is not routed. Spark has these kind of pre and post processing filters, but we don't use those because we want to abort the call to the routed method if `before` returns false. So, we have our own before/after filters that the `Router` class uses to implement this algorithm in the `router` method. This is done just after we create and compose the request container (described above):

```
if (controller.before(request, response))
{
   // Fire the controller's method depending on the HTTP method of the request
   val httpMethod = request.requestMethod().toLowerCase()
   val method = controllerClass.getMethod(httpMethod, javaClass<Request>(), javaClass<Resp\
onse>())
   val result = method.invoke(controller, request, response)

   if (result is ControllerResult && result.continueProcessing)
   {
           controller.after(request, response)

           model = result.model
   }
}
```

This if condition will be false for the `AuthorizationController` when the user isn't logged in. So, the GET made by the client will never be dispatched to the controller's `get` method. Instead, the redirect in the `before` filter will cause the user to be sent to the login endpoint.

The `LoginController` handles the GET made by the client that follows the redirect. They are presented with the view that was associated with that endpoint. This allows the user to enter their credentials and post them back to the same controller. To process this, the `LoginController` also overrides `Controllable`'s `post` method like this:

```
public override fun post(request: Request, response: Response): ControllerResult
{
        var session = request.session() // Create session

        // Save the username in the session, so that it can be used in the authorize endpoint (e.\
g., for consent)
        session.attribute("username", request.queryParams("username"))

        // Redirect back to the authorize endpoint now that "login" has been performed
        response.redirect("/authorize")

        return ControllerResult(continueProcessing = false)
}
```

Here, we create a session for the user using Spark's `Session` class (which is a thin wrapper around `javax.servlet.http.HttpSession`), saving the username for later processing. Then, we redirect the user back to the `AuthorizeController`. (We also abort further processing in this method, causing the `after` method of the controller to not be called.) When the user follows this redirect, the `before` filter of the `AuthorizeController` will return true, allowing the access token to be issued this time by ensuring that the overriden `get` method is called.

We admit that this OAuth example is contrived, but it shows how you can add controllers to Spark and how these can have their dependencies injected using DI. With the niceties of the Kotlin syntax, we can even make it easy to wire up all these components. Fork the sample, and see if you can prototype the logic of your API. If it doesn't work, *please* let us know in a comment!

4.6 Conclusion and Next Steps

This series has been a long couple of posts with the Spark intro sandwiched in between. If you have read this far though, you now have three powerful tools in your API toolbelt:

- **The Java Virtual Machine**: an open platform that prioritizes backward compatibility, ensuring the longevity of your code.
- **A flora of frameworks**: Spark, Pico, and the others demonstrated in this series are only the tip of the iceberg of what's available in the Java ecosystem.
- **Kotlin**: an open source language being developed by Jetbrains, a forerunner in the Java community.

This triad will make you more productive and help you deliver higher quality APIs in a shorter amount of time. Using the boilerplate developed during this series, you now have a starting point to get going even faster. Twobo Technologies plans to adopt Kotlin now for non-shipping code, and to include it in product code as soon as version 1.0 of the language is released. We encourage you to use the sample and your new knowledge to formulate a similar roadmap.

5. Using Spark to Create APIs in Scala

We've discussed the strengths of the **Java Language** within the **Spark** framework, highlighting the ways Java Spark increases simplicity, encourages good design, and allows for ease of development. In this chapter we continue our coverage on Spark, a micro framework great for defining and dispatching routes to functions that handle requests made to your web API's endpoints.How we examine the counterpoint to Java Spark, **Scala Spark**. We'll discuss the origin, methodologies, and applications of Scala, as well as some use-cases where Scala Spark is **highly effective**.

5.1 Reintroducing Spark

In the first piece of this series, Using Spark to Create APIs in Java, we discussed Spark as a toolkit to primarily define and dispatch routes to functions that handle requests made to the

API endpoint. Spark was designed specifically to make these route definitions quick and easy, utilizing the lambdas built into Java 8.

When we first introduced Spark, we used this typical Hello Word example:

```
import static spark.Spark.*;

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

When this snippet is run, Spark will spin up a web server that will serve our API. The user can then navigate to http://localhost:4567/hello, which will call the lamba mapped with spark.Spark.get method. This will return Hello World.

5.2 Scala — It's Origin and Purpose

The basic architecture and design of Scala was released in 2001 at the École Polytechnique Fédérale de Lausanne (EPFL) research university by German computer scientist and professor of programming methods Martin Odersky. Designed primarily as a **Java bytecode compliant language** meant to function without the shortcomings of Java proper, Scala was named as a portmanteau of the words "scalable" and "language".

This name highlights the goal of Scala proper — a language that is extensible, powerful, and designed to grow as the demands of its users and developers grow. Because the language is derived from the Java bytecode, it is functionally object-oriented in nature.

Benefits of Scala

There are many benefits inherent in Scala that makes it a wonderful choice for a wide range of applications.

- **Functional *and* Useful**: We've previously discussed the difference between functionality and usabiltiiy, and Scala meets both of these very different requirements with finesse. Scala is designed to support migration, simple syntax, immutability, and cross-support with many other languages and extensions.
- **Scalable by Design**: Scala is short for "Scalable" — and it shows. Because Scala is, by design, concise in syntax and utilizes lower resources than other languages, it is adept at both small, portable applications and large, complex systems.
- **Object Oriented**: Scala is entirely object-oriented by its very nature. Every value is an object, and every operation a method-call; all augmented by complex classes and traits allowing for advanced architecture and designs. In contrast to other languages, generics in Scala are well supported, increasing usefulness and extensibility.
- **Precise**: Scala is precise due to its strict constraints. This means that, more often than not, issues will be caught during compilation rather than after full deployment.

5.3 Why Scala? Why Not Java?

For many developers, the question of whether or not to use Scala or Java is one of convenience. "I know Java, and Scala is so much like Java...so why should I switch?" It's a legitimate question, and one that has a simple answer — Scala does some things extremely well that Java does not, and in **less space**.

Scala is, to quote Systems Engineer at Motorola Brian Tarbox, "transformational rather than procedural." Whereas Java explains to the system how to do something, Scala offers simple steps to perform that same function without verbosity. It's arguably cleaner, and thus simpler, while accomplishing the same thing.

Take this simple comparison. Let's create a list of strings, using standard usage language. (There are many ways to shorten Java code, but many are not supported or are not in standard usage, and thus will not be discussed here.)

The **Java** version:

```
List<String> list = new ArrayList<String>();
list.add("1");
list.add("2");
list.add("3");
list.add("4");
list.add("5");
list.add("6");
```

Compare this to the **Scala** version:

```
val list = List("1", "2", "3", "4", "5", "6")
```

While some may not view this as a large enough reduction in space, keep in mind that as code expands to larger and larger lengths, the importance of **compactness** certainly adds up. As another example, let's create a snippet that draws from a pre-defined User class, and returns all the products that have been ordered by that user.

The **Java** version:

```
public List<Product> getProducts() {
    List<Product> products = new ArrayList<Product>();
    for (Order order : orders) {
        products.addAll(order.getProducts());
    }
    return products;
}
```

The **Scala** version:

```
def products = orders.flatMap(o => o.products)
```

Now that's a huge difference — from seven lines to one.

5.4 Different, More Efficient Methods

Reduction in complexity is certainly valuable, but there's something more fundamental going on between Java and Scala. As a further example to demonstrate the reduction in complexity, the previously-quoted Tarbox created the following log processing method in both Java and Scala:

The **Java** version:

```
 import java.io.BufferedReader;
  import java.io.DataInputStream;
  import java.io.FileInputStream;
  import java.io.InputStreamReader;
  import java.util.HashMap;
  import java.util.Iterator;
  import java.util.Map;
  import java.util.Scanner;
  import java.util.TreeMap;
import java.util.Vector;
 void getTimeDiffGroupedByCat() {
        FileInputStream fstream = new FileInputStream("textfile.txt");
        DataInputStream in = new DataInputStream(fstream);
       BufferedReader br = new BufferedReader(new InputStreamReader(in));
        String strLine;
        Long thisTime;
        HashMap<String, Long> lastCatTime = new HashMap<String, Long>();
        TreeMap<String, Vector<Long>> catTimeDiffs
                    = new TreeMap<String, Vector<Long>>();
        while ((strLine = br.readLine()) != null)   {
                Scanner scanner = new Scanner(strLine);
                thisTime = scanner.nextLong();
                 String category = scanner.next();
                 Long oldCatTime = lastCatTime.put(category, thisTime);
                 if(oldCatTime != null) {
                       if(catTimeDiffs.get(category) == null) {
                         catTimeDiffs.put(category, new Vector<Long>());
                       }
                       catTimeDiffs.get(category).add(thisTime - oldCatTime);
                 }
          }
          for(Map.Entry<String, Vector<Long>> thisEntry:
```

```
33                      catTimeDiffs.entrySet()) {
34                  System.out.println("Category:" + thisEntry.getKey());
35                  Iterator it = thisEntry.getValue().iterator();
36                  while(it.hasNext()) {
37                          System.out.println(it.next());
38                          if(it.hasNext())
39                                  System.out.print(", ");
40                  }
41          }
42  }
```

This code creates a simple HashMap that holds the timestamps keyed, organizing it by category. These values are then compared to previous timestamps from the HashMap, returning the difference as an appended category vector of time differences in the generated TreeMap. Lines 28-36 print these results in a series of comma-separated lists.

The **Scala** version:

```
1       import Source.fromFile
2       def getTimeDiffGroupedByCat = {
3         val lines = fromFile("file.txt").getLines
4          val tuppleList = for(oneLine <- lines) yield {val z = oneLine.split
5                    (' '); (z(0).toInt, z(1)) }
6          for(groupedList <- tuppleList.toList.groupBy(oneTuple =>
7                    oneTuple._2)) {
8             val diffList = for(logPairs <- groupedList._2.sliding(2)) yield
9                       (logPairs(1)._1 - logPairs(0)._1)
10            println(groupedList._1 + ":" + diffList.mkString(","))
11      }
12  }
```

Forgoing the fact that the Scala version is far less verbose — certainly a reason to adopt Scala alone — there is something far more important going on here. Because Scala is immutable, the handling is dealt with differently in the Scala version, transforming data in lists rather than in the mutable Java HashMaps.

These lines are generated using the getLines function, which is then iterated over the list of lines, executing the code following the yield valuation. This result is then added to the tuppleList variable, which is then grouped using groupBy, which after several additional manipulations following the oneTuple._2 and logPairs(0)._1 definitions, are printed.

Simply put, the Scala version handles the same function with less code, in a more succinct way, and with less complexity, while allowing for immutable manipulation and typing.

5.5 Scala Performance and Integration

Spark is brief — it was designed to be compact, and to carry out performance with relatively

small codebases. Though Java does this to a point, Scala holds this spirit in its very essence. Scala is by all accounts a language that results in smaller codebases, more efficient processing, and easier troubleshooting and design.

For all its benefits, **Scala** can be more complex than the intended functionality. For this reason, some have shied away from it. Complexity arising from the nature of the code's implicit functionality interactions can often make implementation of encryption, transformation, authentication, etc. more complex than it would be in Java.

That being said, the end result of this complexity is counter-intuitive simplicity in the actual codebase. Once the functionality is figured out, and the function map conceived, this complexity is represented by relatively small code samples, and thus efficient processing.

5.6 Conclusion

Scala and Java are two wonderful languages. Like every other language, however, there are serious strengths and weaknesses to consider when utilizing them for development. While Java is the best choice for a developer who is firmly entrenched in the Java mindset or has heavy programming experience in the language, Scala is a wonderful alternative that certainly reduces complexity, while allowing for more complex manipulation.

This is, of course, only part of the equation — integration of security systems and protocols, methodologies of microservice development, and the fundamental API architecture is just as important as the language it is being developed in. With a full understanding of the language chosen, and a complete visualization of the requirements inherent in your API, successful development becomes an attainable goal.

Resources

The following resources can help novice and experienced programmers alike find out more about Spark, Scala, Kotlin, and other techs mentioned throughout this eBook. Read up on these, and try out Spark today. Feel free to let us know how it turns out.

Spark

- Getting Started with Spark
- Named parameters and wildcard patterns in routes (example)
- Before/after filters (example)
- Halting the request (example)
- Request, response, session, and cookie objects
- Exception mapping
- Filtering by accept headers
- Redirecting the response (example)
- Serving up static content (example)
- Adding Spark to your POM

Scala

- Getting Started in Scala
- Complete Scala Language Specification

Kotlin

- Kotlin language reference (start here)
- Kotlin programming language cheat sheet
- Designing DSLs in Kotlin
- RE: Why Kotlin is my next programming language - SubReddit discussion
- Sample Web API project using Kotlin and Spark

About Nordic APIs

Nordic APIs is the number one place for information and resources on API technology and business. We produce eBooks, events, blog posts and much more.

API Themed Events

Nordic APIs holds meetups, conferences, and seminars throughout Scandinavia and Europe. Subscribe to our newsletter or check our event calendar. Also follow our YouTube channel for more videos.

Follow the Nordic APIs Blog

Much of our eBook content originates from the Nordic APIs blog, where we publish in-depth API-centric thought pieces and walkthroughs twice a week. Sign up to our newsletter to receive blog post updates via our Weekly Digest, or visit our blog for the latest posts!

More eBooks by Nordic APIs:

Visit our eBook page to download any of the following eBooks for free:

- **Securing the API Stronghold**: A comprehensive dive into the core tenants of modern web API security.
- **The API Lifecycle**: A holistic approach to maintaining your API throughout it's entire lifecycle, from conception to deprecation.
- **Developing The API Mindset** : Details the distinction between Public, Private, and Partner API business strategies with use cases from Nordic APIs events.
- **Nordic APIs Winter Collection**: Our best 11 posts published in the 2014 - 2015 winter season.
- **Nordic APIs Summer Collection 2014**: A handful of Nordic APIs blog posts offering best practice tips.

Endnotes

Nordic APIs is an independent blog and this publication has not been authorized, sponsored, or otherwise approved by any company mentioned in it. All trademarks, servicemarks, registered trademarks, and registered servicemarks are the property of their respective owners.

Nordic APIs AB Box 133 447 24 Vargarda, Sweden

Facebook | Twitter | Linkedin | Google+ | YouTube

Blog | Home | Newsletter | Contact