Introduction to Spring Boot

Brian Clozel – bclozel@pivotal.io |

Table of Contents

Introduction

Spring Framework (https://projects.spring.io/spring-framework) is a popular, open-source, Java-based application framework - it's well known for its flexibility and its rich ecosystem. Many other projects are in the Spring Platform, for big data, batch workloads, storing data, securing applications, and more!

Spring Boot (https://projects.spring.io/spring-framework) takes an opinionated view of the Spring platform and third-party libraries. With Boot, it's easy to create production-grade Spring based applications for all types of workloads Most Spring Boot applications need very little Spring configuration. Spring Boot is a "convention over configuration" type of framework, with no code generation.

> During the whole lab, you'll find the Spring Boot reference documentation (http://docs.spring.io/spring-boot/docs/2.0.6.RELEASE/reference/htmlsingle/) / javadoc API (http://docs.spring.io/spring-boot/docs/2.0.6.RELEASE/api/) and the Spring Framework reference documentation (http://docs.spring.io/spring-framework/docs/5.0.10.RELEASE/spring-framework-reference/) / javadoc API (http://docs.spring.io/spring-framework/docs/5.0.10.RELEASE/javadoc-api/) quite useful.

During this lab, we'll first get to know the basic concepts behind Spring Framework and then create a Spring Boot application and experience the development lifecycle with it.

Getting started - Fast!

The best way to start a project with Spring is Spring Initializr (http://start.spring.io). This website will help you to create a minimal skeleton for your project.

For this lab, we'll select the following options:

- We'll generate a **Maven** project with Spring Boot **2.0.6.RELEASE**
- Our group Id `fr.insalyon.tc` and artifact Id `spring-boot-intro`

- In the dependencies search box, select `Web` and `Devtools`

Click on "Generate", then unzip the file somewhere.

In the `spring-boot-intro` folder, your `pom.xml` file should look like this:

pom.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>fr.insalyon.tc</groupId>
    <artifactId>spring-boot-intro</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>spring-boot-intro</name>
    <description>Demo project for Spring Boot</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.0.6.RELEASE</version> 1
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
    </properties>

    <dependencies> 2
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId> 3
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>


</project>
```

1 Getting the latest stable version of Spring Boot

2 Getting the Web, Devtools Dependencies we selected on start.spring.io and the starter for testing

3 No need to specify versions for all dependencies as Spring Boot manages many versions

If you've got an IDE installed, you can import it as a Java application and run the `SpringBootIntroApplication` class directly from the IDE. Using an IDE will assist you with imports, debugging, and much more.

Alternatively, you can open a new console and run the following command:

```bash
$ ./mvnw spring-boot:run  1

[...]
INFO 8181 --- [  restartedMain] o.s.b.d.a.OptionalLiveReloadServer       : LiveReload server is running on port 35729
INFO 8181 --- [  restartedMain] o.s.j.e.a.AnnotationMBeanExporter        : Registering beans for JMX exposure on startup
INFO 8181 --- [  restartedMain] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)  2
INFO 8181 --- [  restartedMain] f.i.t.s.SpringBootIntroApplication       : Started SpringBootIntroApplication in 2.182
seconds (JVM running for 2.518)
```

1 this will compile and run the application

2 an embedded server is started and listens on port 8080

Using `curl` or a browser, we can send an HTTP request to `http://localhost:8080`. You'll get an error page in HTML format with your browser and a JSON error message with `curl`:

```bash
$ curl http://localhost:8080/  1
{"timestamp":1519230961036,"status":404,"error":"Not Found","message":"No message available","path":"/"}%  2
```

1 a server is running on port 8080 (port chosen by default, but you can configure that)

2 since we haven't done anything in our app, Spring Boot answers with a default HTTP 404 error message

Anatomy of a Spring Boot application

Now you can browse the source code of your application.

```bash
spring-boot-intro
|- pom.xml  1
|- src/
   |- main/
   |  |- java/
   |  |  |- fr/insalyon/tc/springbootintro/
   |  |     |- SpringBootIntroApplication.java  2
   |  |- resources/
   |     |- static/  3
   |     |- templates/  4
   |     |- application.properties  5
   |- test/
      |- java/
         |- fr/insalyon/tc/springbootintro/
            |- SpringBootIntroApplicationTests.java  6
```

1 Our Maven build file

2 Main Application class

3 Static resources (e.g. CSS, JS)

4 Template files (for rendering HTML views)

5 Spring Boot application properties

6 An example test class

Our main Application class `SpringBootIntroApplication` looks like this:

src/main/java/fr/insalyon/tc/springbootintro/SpringBootIntroApplication.java

```java
@SpringBootApplication
public class SpringBootIntroApplication {

    public static void main(String[] args) { 1
        SpringApplication.run(SpringBootIntroApplication.class, args); 2
    }
}
```

1. You can start this application by just running the "main" method…
2. … which in turn runs our application using its main configuration class

Dependency Injection

This section explains the concept of Dependency Injection - you'll start writing code in the Using Dependency Injection section.

When writing an application, as developers, we break the problem we're trying to solve into smaller ones and do our best keep in line with the architecture and design principles we've chosen for our application: flexible, decoupled, testable, easy to understand, etc.

To do that we can break our application into components that collaborate: components are depending on each other. But this adds some cost: we now have to manually create those and link them together. Let's take an example of a Music Streaming application:

Bootstrapping our application

```java
// Setting up our components can be quite challenging
// and we have to maintain this code...

// Users database
UserRepository userRepository = new UserRepository();
// To manage subscriptions, we need the users' information
SubscriptionService subscriptionService = new SubscriptionService(userRepository);
// Songs database
SongsRepository songRepository = new SongRepository();
// We need to index songs to provide a search service
SearchService searchService = new SearchService(songRepository);

// What if you need to support playlists created by users?
// You'd need to create new components, and manually change
// the whole setup to add dependencies between them.
```

Dependency injection solves that problem, and more.

With Spring, you don't have to write that code, you just need to **express** those dependencies with Java annotations. Here's how we could write that code:

Creating Spring components

JAVA

```java
// SongRepository.java
@Component 1
public class SongRepository {

  //...
}


// SearchService.java
@Service 2
public class SearchService {

  private final SongRepository songRepository;

  @Autowired 3
  public SearchService(SongRepository songRepository) {
    this.songRepository = songRepository;
  }

  public void indexSongs() {
    // use this.songRepository to find existing songs and index them
  }
}
```

1 We declare our application classes as components, by annotating them… `@Component`

2 There are other, specialized annotations to declare Spring components, like `@Service`

3 By using `@Autowired` on a constructor, we're asking Spring to inject here dependencies

Once you've done that in your application, you need to configure Spring properly and start your application. Then Spring will do the following:

1. Look for components by scanning your application classpath
2. Register all those components in an **application context** as **bean definitions**; at that point it's building a dependency graph for your application
3. Spring will instantiate and set dependencies on your components, we can now call them **beans**. By default, Spring will create **singleton beans**, i.e. a single instance for each definition.

With this, we don't need to maintain the manual bootstrapping of our application, and more.

But what happens if you can't add a `@Component` annotation on a class because you didn't write it (let's say it comes from a library)?

In that case, you can write `@Bean` methods and declare dependencies on them. It's your reponsibility to instantiate classes in these cases:

Writing @Bean methods

JAVA

```java
// AppConfiguration.java
@Configuration 1
public class AppConfiguration {

  @Bean 2
  public PlaylistManager playlistManager(SongRepository songRepo, UserRepository userRepo) { 3
    // here, we can create a PlaylistManager instance using the provided dependencies
    return playlistManager;
  }

}
```

1 We need to declare `@Bean` methods in a Configuration class

2 A bean method creates a bean instance and is annotated with `@Bean`

3 You can declare dependencies as method arguments and Spring will call this method with the required params

In summary, you can create components by annotating your classes with `@Component` (or some other specialized annotation) and express dependencies between components with `@Autowired`. If you can't annotate a class yourself, you can achieve the same thing by creating them with `@Bean` methods and expressing dependencies with method parameters.

Using Dependency Injection

We'll now create a greeting application and apply those concepts.

As a first step, we can create an `ApplicationRunner` bean, which will be executed by Spring Boot when the application starts up.

Each code snippet shows where files should be located; make sure to follow the guidelines here.

src/main/java/fr/insalyon/tc/springbootintro/GreetingRunner.java

JAVA
```java
package fr.insalyon.tc.springbootintro;

import org.springframework.boot.ApplicationArguments;
import org.springframework.boot.ApplicationRunner;
import org.springframework.stereotype.Component;

@Component  1
public class GreetingRunner implements ApplicationRunner {  2

    @Override
    public void run(ApplicationArguments args) throws Exception {
        String name = "Spring";
        String greeting = String.format("Hello, %s!", name);
        System.out.println(greeting);
    }
}
```

1 We declare this class as a Spring component

2 This implements `ApplicationRunner`, so Spring Boot will detect it and run it during the application startup phase

Run this application from your IDE (run the `main` method of the `SpringBootIntroApplication` class) or run `./mvnw spring-boot:run`:

BASH
```
INFO 7198 --- [  restartedMain] o.s.b.d.a.OptionalLiveReloadServer       : LiveReload server is running on port 35729
INFO 7198 --- [  restartedMain] o.s.j.e.a.AnnotationMBeanExporter        : Registering beans for JMX exposure on startup
INFO 7198 --- [  restartedMain] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
Hello, Spring!
INFO 7198 --- [  restartedMain] f.i.t.s.SpringBootIntroApplication       : Started SpringBootIntroApplication in 1.603
seconds (JVM running for 2.139)
```

For the purpose of the demo, we'd like to separate things more into reusable components (let's say we'd like to produce greeting messages for the console, but also in web pages, when sending emails, etc).

Now, let's create an interface for our application in a new `hello` package - we want to use this interface in our codebase and not depend on a specific implementation:

src/main/java/fr/insalyon/tc/springbootintro/hello/GreetingService.java

JAVA
```java
package fr.insalyon.tc.springbootintro.hello;

public interface GreetingService {

  String greet(String name);
}
```

Your first job is to create an implementation for that, that returns "Hello, Spring!" so we can print that message in the console as the application starts. For that, do the following:

Create a `SimpleGreetingService` implementation of that interface, and mark is as a component with an annotation (using `@Component` or `@Service`). The implementation of the `greet` method should use the `String.format` piece of code we've been using.

You can verify that your implementation is working properly by running the following test with the `./mvnw test` command.

> Watch the location of this file, tests are located under `src/test/java` !

src/test/java/fr/insalyon/tc/springbootintro/hello/SimpleGreetingServiceTests.java

JAVA
```java
package fr.insalyon.tc.springbootintro.hello;

import org.junit.Test;

import static org.assertj.core.api.Assertions.assertThat;

public class SimpleGreetingServiceTests {

    @Test
    public void testGreeting() {
        SimpleGreetingService greetingService = new SimpleGreetingService(); 1
        String message = greetingService.greet("Spring");
        assertThat(message).isEqualTo("Hello, Spring!");
    }
}
```

1 We're testing our service implementation without Spring being involved, annotations are declarative and won't interfere here.

Now, in the `GreetingRunner` class, replace the code inside the `run` method so that your application uses our new component.

Here is the current state of your `GreetingRunner` class - read the notes for clues on what should be done here:

src/main/java/fr/insalyon/tc/springbootintro/GreetingRunner.java

JAVA

```java
package fr.insalyon.tc.springbootintro;

import org.springframework.boot.ApplicationArguments;
import org.springframework.boot.ApplicationRunner;
import org.springframework.stereotype.Component;

@Component
public class GreetingRunner implements ApplicationRunner {

  public GreetingRunner() {  1

  }

  @Override
  public void run(ApplicationArguments args) throws Exception {
    String name = "Spring";
    String greeting = String.format("Hello, %s!", name);  2
    System.out.println(greeting);
  }
}
```

1 You can ask Spring to inject a `GreetingService` here as a method parameter, look at our previous `SearchService` example.

2 Delete this line and call our `GreetingService` instead. Remember, you're not allowed to do `new SimpleGreetingService()` here!

You now should start your application and see that you still get the same result in the console.

DI Edge cases

Now, we're going to test a few cases to understand how a Spring Application reacts to some situations. For each case, try the suggested modifications, restart your application and see what happens. Of course, after each case, **revert those changes**, to get "back to normal".

Missing Component

What happens if you comment the `@Component` / `@Service` annotation on your `SimpleGreetingService` ?

Multiple candidates

Now, try adding `AnotherGreetingService` (which says "Bonjour" instead of "Hello"), marked as a component as well. Try again this time after adding a `@Primary` annotation on `SimpleGreetingService` .

Dependency cycle

Finally, try the following - what happens and why?

src/main/java/fr/insalyon/tc/springbootintro/hello/SimpleGreetingService.java

JAVA

```java
package fr.insalyon.tc.springbootintro.hello;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class SimpleGreetingService implements GreetingService {

  private final CycleService cycleService;

  @Autowired
  public SimpleGreetingService(CycleService cycleService) {
    this.cycleService = cycleService;
  }

  @Override
    public String greet(String name) {
    return String.format("Hello, %s!", name);
    }

}
```

src/main/java/fr/insalyon/tc/springbootintro/hello/CycleService.java

JAVA

```java
package fr.insalyon.tc.springbootintro.hello;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class CycleService {

  private final GreetingService greetingService;

  @Autowired
  public CycleService(GreetingService greetingService) {
    this.greetingService = greetingService;
  }
}
```

`@Primary` is not the only way to resolve multiple candidates, you can also use `@Qualifier`; check its javadoc to see how you could use it.

Creating a Web application with Spring

Spring provides a framework for building web applications, called Spring MVC. In this part, we'll use our previous `GreetingService` to create a web service.

Simple Controller

First, let's create a Spring MVC Controller that responds with greetings. Spring MVC is the Web Framework built in Spring; it helps you write web applications and takes care of a lot of boilerplate code, so you just have to focus on your application features.

Controllers are the link between the web (http clients, like browsers) and your application; Controllers should be lightweight and call other components in your application to perform actual work.

Now create the following:

src/main/java/fr/insalyon/tc/springbootintro/GreetingController.java

JAVA

```java
package fr.insalyon.tc.springbootintro;

import fr.insalyon.tc.springbootintro.hello.GreetingService;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller  1
public class GreetingController {

  private final GreetingService greetingService;

  @Autowired
  public GreetingController(GreetingService greetingService) {  2
    this.greetingService = greetingService;
  }

  @GetMapping("/greet")  3
  @ResponseBody  4
  public String greet(@RequestParam String name) {  5
    return greetingService.greet(name);
  }
}
```

1 A Controller is a Spring component that's used to handle HTTP requests; Spring will map incoming HTTP requests to a particular Controller method, execute it, and use the returned value to compute the HTTP response

2 We're getting injected with the existing `GreetingService`

3 All GET requests with path `"/greet"` will be mapped to this method

4 the value returned by the method will be written directly as the HTTP response body

5 With various annotations, you can inject `@PathVariable` (parts of the request path), `@RequestParam`, `@RequestHeader`, `@RequestBody` and more…

Restart your application. You should see that the Controller method is now mapped by Spring:

BASH

```bash
Mapped "{[/greet],methods=[GET]}" onto public java.lang.String
fr.insalyon.tc.springbootintro.GreetingController.greet(java.lang.String)
```

with an IDE, your development experience can get much better, especially with the Spring Boot Developer tools (http://docs.spring.io/spring-boot/docs/2.0.6.RELEASE/reference/html/using-boot-devtools.html).

You can now test this new Controller:

BASH

```bash
$ curl http://localhost:8080/greet?name=Alicia
Hello, Alicia!%

$ curl http://localhost:8080/greet?name=Bob
Hello, Bob!%
```

First Integration test

We've just run our application and tested the "/greet" endpoint manually with curl; we could do that automatically as part of our test suite.

One way to test this Controller is to launch an integration test that:

- launches the whole Spring application
- deploys it into the embedded HTTP server
- uses an HTTP client to send a request to that server and check the response

For that, we can create a new `GreetingControllerTests` class.

src/test/java/fr/insalyon/tc/springbootintro/GreetingControllerTests.java

```java
package fr.insalyon.tc.springbootintro;

import org.junit.Test;
import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.web.client.TestRestTemplate;
import org.springframework.test.context.junit4.SpringRunner;

import static org.assertj.core.api.Assertions.assertThat;

@RunWith(SpringRunner.class) 1
@SpringBootTest(webEnvironment= SpringBootTest.WebEnvironment.RANDOM_PORT) 2
public class GreetingControllerTests {

    @Autowired 3
    private TestRestTemplate restTemplate;

    @Test
    public void exampleTest() {
        String body = this.restTemplate.getForObject("/greet?name=Alicia", String.class); 4
        assertThat(body).isEqualTo("Hello, Alicia!"); 5
    }

}
```

1 We're using here a specific JUnit runner to plug the Spring Test Framework

2 This is a web integration test, starting a new server on any available port

3 Spring Boot is creating a test HTTP client bean for us (that knows about host and port), so let's inject it here

4 We're using that HTTP client to send a request to our `/greet` endpoint

5 We can then check thath the HTTP response body matches what we expect

Run that test with the `./mvnw test` command.

HTTP "content negotiation", explained

Content negotiation is a part of the HTTP specification, and it allows clients and servers to tell the other in which format they're sending data, and negotiate which format should be used depending on what the other is willing to accept.

With that, a single document living at a particular URL can be served in multiple formats. You can see that in action by opening the developer tools in your browser and looking at the Network tab; you'll see the request and response HTTP headers for all downloaded resources.

> One could compare that to file extensions in your operating system. A `note.txt` file should contain plain text. Opening that file on your laptop is likely to open a text editor which is configured to open ".txt" files.

In the next sample curl commands, we'll use a service that supports content negotiation for the `/image` endpoint, returning the appropriate format when possible.

BASH

```
curl http://httpbin.org/image -H "Accept: image/jpeg" -v -o /dev/null
> GET /image HTTP/1.1
> Host: httpbin.org
> Accept: image/jpeg  1
>
< HTTP/1.1 200 OK
< Content-Type: image/jpeg  2
< Content-Length: 35588
```

1 Asking for "image/jpeg"

2 Getting it as a response

BASH

```
curl http://httpbin.org/image -H "Accept: image/webp" -v -o /dev/null
> Host: httpbin.org
> Accept: image/webp  1
>
< HTTP/1.1 200 OK
< Server: nginx
< Content-Type: image/webp  2
< Content-Length: 10568
```

1 Asking for the "image/webp" format this time

2 Again, getting it

BASH

```
curl http://httpbin.org/image -H "Accept: application/vnd.insa.tc" -v
> GET /image HTTP/1.1
> Host: httpbin.org
> Accept: application/vnd.insa.tc  1
>
< HTTP/1.1 406 NOT ACCEPTABLE  2
< Content-Type: application/json  3
< Content-Length: 142

{"message": "Client did not request a supported media type.", "accept": ["image/webp", "image/svg+xml", "image/jpeg",
"image/png", "image/*"]}%
```

1 Asking for a type not supported by the server

2 Server says it does not support that

3 and answers with a JSON error message

```bash
curl http://httpbin.org/post -H "Content-Type: application/json" -v -d "{\"name\":\"Spring Framework\"}"
> POST /post HTTP/1.1
> Host: httpbin.org
> Accept: */*  1
> Content-Type: application/json  2
> Content-Length: 27
>
< HTTP/1.1 200 OK
< Content-Type: application/json  3
< Content-Length: 384
<
{
  "args": {},
  "data": "{\"name\":\"Spring Framework\"}",
  "files": {},
  "form": {},
  "headers": {
    "Accept": "*/*",
    "Content-Length": "27",
    "Content-Type": "application/json",
    "Host": "httpbin.org",
    "User-Agent": "curl/7.51.0"
  },
  "json": {
    "name": "Spring Framework"
  },
  "origin": "8.8.8.8",
  "url": "http://httpbin.org/post"
}
```

1 Accepting anything in return

2 Sending a JSON body in the request

3 Getting a JSON response from the server

Now, a JSON API endpoint

Let’s now create another Controller, but this time for an API that speaks JSON.

First, we need an `Book` class:

src/main/java/fr/insalyon/tc/springbootintro/Book.java

JAVA

```java
package fr.insalyon.tc.springbootintro;

@SuppressWarnings("serial")
public class Book {

    private Long id;

    private String isbn;

    private String author;

    private String title;

    public Book() {
    }

    public Book(String isbn, String author, String title) {
        this.isbn = isbn;
        this.author = author;
        this.title = title;
  }

    public String getIsbn() {
        return isbn;
    }

    public void setIsbn(String isbn) {
        this.isbn = isbn;
    }

    public String getAuthor() {
        return author;
    }

    public void setAuthor(String author) {
        this.author = author;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    @Override
    public String toString() {
        return "Book{" +
                "isbn='" + isbn + '\'' +
                ", author='" + author + '\'' +
                ", title='" + title + '\'' +
                '}';
    }
}
```

Then create the following Controller:

src/main/java/fr/insalyon/tc/springbootintro/BookController.java

JAVA
```java
package fr.insalyon.tc.springbootintro;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController  1
@RequestMapping("/books")  2
public class BookController {

  @GetMapping("/{isbn}")  3
    public Book findBook(@PathVariable String isbn) {  4
        Book algoDesign = new Book();
        algoDesign.setIsbn(isbn);
        algoDesign.setAuthor("Steven Skiena");
        algoDesign.setTitle("The Algorithm Design Manual");
        return algoDesign;  5
    }

}
```

1 `@RestController` is an annotation that contains both `@Controller` and `@ResponseBody`

2 Mapping all the methods in this Controller under a `"/books"` path prefix

3 Path mappings can include path variables, meaning for the request of path `"/books/42"` Spring will map a `isbn` variable to `"42"`

4 Those path variables can be then injected as method arguments, implicitely with the name of the method argument or explicitely with `@PathVariable("isbn")`

5 Returning a Java object works here — Spring will handle content negotiation and serialization for you

We can now rebuild our application and test this new endpoint:

BASH
```bash
$ curl http://localhost:8080/books/123456 -i
HTTP/1.1 200
Content-Type: application/json;charset=UTF-8

{"isbn":"123456","author":"Steven Skiena","title":"The Algorithm Design Manual"}%
```

Again, we should see that new endpoint declared during application startup:

BASH
```bash
Mapped "{[/books/{isbn}],methods=[GET]}" onto public fr.insalyon.tc.springbootintro.Book
fr.insalyon.tc.springbootintro.BookController.findBook(java.lang.String)
```

Have some spare time?

HTTP Caching

As you may know, HTTP clients can cache HTTP resources so they don't have to redownload them. There's much more to that and web applications can leverage that in APIs as well.

You can read up a bit more about this in this HTTP caching (https://developer.mozilla.org/en-US/docs/Web/HTTP/Caching) article.

First, let's update our `Book` class to add a `version` field. This value should change if we ever modify the information about a given book.

src/main/java/fr/insalyon/tc/springbootintro/Book.java

JAVA

```java
package fr.insalyon.tc.springbootintro;

@SuppressWarnings("serial")
public class Book {

    private Long id;

    private String isbn;

    private String author;

    private String title;

    private String version;

    public Book() {
    }

    public Book(String isbn, String author, String title, String version) {
        this.isbn = isbn;
        this.author = author;
        this.title = title;
    }

    public String getIsbn() {
        return isbn;
    }

    public void setIsbn(String isbn) {
        this.isbn = isbn;
    }

    public String getAuthor() {
        return author;
    }

    public void setAuthor(String author) {
        this.author = author;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getVersion() {
        return version;
    }

    public void setVersion(String version) {
        this.version = version;
    }

    @Override
    public String toString() {
        return "Book{" +
                "isbn='" + isbn + '\'' +
                ", author='" + author + '\'' +
                ", title='" + title + '\'' +
                ", version='" + version + '\'' +
                '}';
    }
}
```

Instead of just returning the `Book` instance, we can return a `ResponseEntity`, which can hold information about the HTTP response we'd like to return.

src/main/java/fr/insalyon/tc/springbootintro/BookController.java

JAVA

```java
package fr.insalyon.tc.springbootintro;

import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/books")
public class BookController {

  @GetMapping("/{isbn}")
  public ResponseEntity<Book> findBook(@PathVariable String isbn) { 1
    Book algoDesign = new Book();
    algoDesign.setIsbn(isbn);
    algoDesign.setAuthor("Steven Skiena");
    algoDesign.setTitle("The Algorithm Design Manual");
    algoDesign.setVersion("v1");
    return ResponseEntity
        .ok() 2
        .eTag(algoDesign.getVersion()) 3
        .body(algoDesign); 4
  }

}
```

1 Returning a `ResponseEntity` with a `Book` as its body

2 Response should be HTTP 200 OK

3 with an "ETag" response header holding the version of the book

4 and finally the book instance as its body

With those changes, we should see a new header in the HTTP response.

BASH

```bash
$ curl http://localhost:8080/books/123456 -H"Accept: application/json" -i
HTTP/1.1 200
ETag: "v1"
Content-Type: application/json;charset=UTF-8

{"isbn":"123456","author":"Steven Skiena","title":"The Algorithm Design Manual","version":"v1"}%
```

This means if an HTTP client sends the appropriate HTTP headers, we can have a different behavior and leverage HTTP caching features.

BASH

```bash
$ curl http://localhost:8080/books/123456 -H"Accept: application/json" -H"If-None-Match: \"v1\"" -i
HTTP/1.1 304
ETag: "v1"
```

Spring Getting Started Guides

If you've got some time left, you can check out the Spring.io guides (https://spring.io/guides) and try one that looks interesting to you.

Resources

You'll find a complete application for this lab in the source repository (https://github.com/bclozel/lectures/tree/master/spring-boot-lab/spring-boot-intro).

Last updated 2018-10-17 10:25:01 CEST