"ZPR PWr - Integrated Development ProgramWrocław University of Technology"

Wrocław University of Science and Technology

Advanced Web Technologies

Lab

Topic: REST Api on the example of Spring Boot

Prepared by: mgr Piotr Jozwiak

Date: July 2020

Number of hours: 2 hours

Table of Contents

Entry

In today's lab, we will discuss the REST API on the example of one of the most popular technologies used for backend software - Spring Boot.

Spring Boot is an Open Source framework available in Java, designed to write microservices. It was developed on the basis of the Spring framework, introducing a number of facilities for programmers. The most important of them is the introduction of the "Convention over Configuration" principle, which minimizes the requirements related to software configuration by introducing a number of well-chosen default values and well-thought-out code implementation conventions. The second important advantage of this framework is the simplification of dependency management, which is quite cumbersome in pure Spring. All changes introduced to SB are aimed at accelerating software development in relation to its base Spring solution.

REST Api

But let's start with deciphering the REST API shortcut. The word REST itself comes from the words:
"Representational State Transfer". So what is REST? It is a defined style of software architecture that introduces a set of rules describing defining resources and how to access them. It was proposed by Roy Felding in 2000. The word API itself comes from words
"Application Programming Interface" - which means a set of rules defining the communication interface with a given application.

To be able to use the term RESTful application, it must meet a number of requirements:

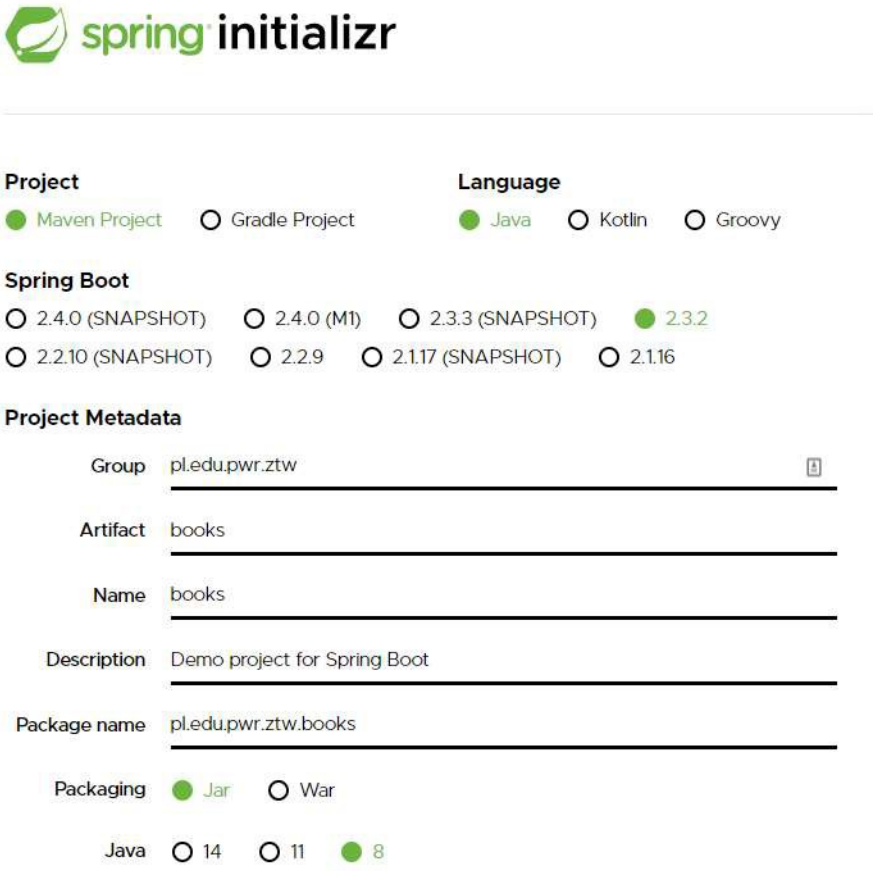
1. Separation of the forntend from the backend. The user interface has no right to interfere with what and how is happening on the server side. This principle also works the other way around.
2. Statelessness – this means that the request from the client must have complete information, because the server does not know about previous requests of the given client.
3. Cacheability – returned responses should clearly define whether they can be cached or not.
4. Layering – the client does not know whether it connects directly to the endpoint or through any proxy or balancer. In other words, the solution must allow for the introduction of intermediate layers between the client and the server, which will not interfere with communication.

Let's move on to practice and analyze an example showing how to work with the Spring Boot environment.

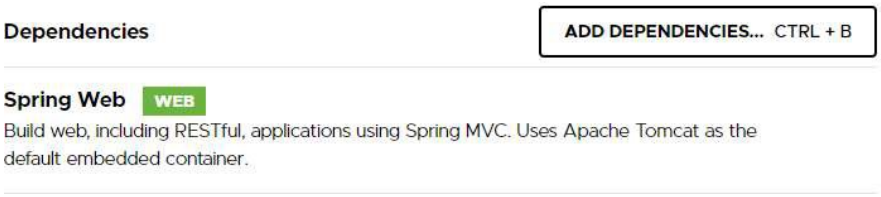
Spring Boot - how to start

The easiest way to start working is to use Spring Initializr, which in a few simple steps will prepare the necessary configuration and folder hierarchy. To use this solution, please go to:https://start.spring.io/. A form will appear in which you must answer several questions, including about the project type (Maven or Gradle), language (Java, Kotlin, Groovy), Spring Boot version and the settings of our package in the Metadata section.

In our example, we will focus on developing a simple application that allows access to a book catalog. Therefore, for the above example, we generate the code with the following settings:



In the Dependencies section, we add a dependency to Spring Web:



Download the template via the Generate button in the form of a zip file. After downloading, unpack the archive and open the project. To work with the project, you can use the IntelliJ or Eclipse IDE environment or any other supporting work with the Java environment.

Now is a good time to compile a blank project template and run it and see what happens. Messages are sent to output during startup. After the startup is finished, they look like in the picture below:

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.3.2.RELEASE)

2020-07-28 21:34:25.425  INFO 35348 --- [           main] pl.edu.pwr.ztw.books.BooksApplication    : Starting BooksApplication on MIC20107 with PID 35348 (C:\Prv\OneD
2020-07-28 21:34:25.432  INFO 35348 --- [           main] pl.edu.pwr.ztw.books.BooksApplication    : No active profile set, falling back to default profiles: default
2020-07-28 21:34:28.665  INFO 35348 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
2020-07-28 21:34:28.683  INFO 35348 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2020-07-28 21:34:28.684  INFO 35348 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.37]
2020-07-28 21:34:28.827  INFO 35348 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2020-07-28 21:34:28.827  INFO 35348 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 3301 ms
2020-07-28 21:34:29.143  INFO 35348 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor  : Initializing ExecutorService 'applicationTaskExecutor'
2020-07-28 21:34:29.383  INFO 35348 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
2020-07-28 21:34:29.400  INFO 35348 --- [           main] pl.edu.pwr.ztw.books.BooksApplication    : Started BooksApplication in 4.794 seconds (JVM running for 6.08)
2020-07-28 21:36:33.162  INFO 35348 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet 'dispatcherServlet'
2020-07-28 21:36:33.162  INFO 35348 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
2020-07-28 21:36:33.168  INFO 35348 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Completed initialization in 5 ms
```

It's worth noting right away that the Tomcat web server was populated during compilation and when our application is launched, it is used to handle requests. We do not have to install a web server ourselves (although we can do it manually). So let's see what will appear in the browser when we enter the address of this server. Tomcat was started on the addresshttp://localhost:8080. After entering this page, we get the following result:

This is an error page that there is no mapping defined at the given address, which I will talk about in a moment. In other words, there is no website/page at this addressWWW.We will tackle this problem in the next step.

Let's fix the displayed error. Following the process below will allow us to understand the basics of how the controller works. So let's create a class that looks like this:

```
packagepl.edu.pwr.ztw.books;

importorg.springframework.web.bind.annotation.RequestMapping;
importorg.springframework.web.bind.annotation.RestController;

@RestControllerpubli
c classhello{

    @RequestMapping("/")publi
    cStringrun() {
        return"Hello! I'm running";
    }
}
```

Before we discuss the operation of the above code, I suggest you see the effect of its operation:

As you can see from the picture above, now we see the inscription as the home page. It's time to discuss the code that generated the above result;

We have created a class called Hello. The first important thing to note is @RestController annotation of this class. This annotation tells the framework that this class is to be a managed bean. This causes that every HTTP request will be processed by classes with the above annotation. So our class is an HTTP request controller.

In the body of this class, we created one method called run(). Its operation is trivial and boils down to returning the string that appeared in the browser. The more interesting element of this part of the code is another @RequestMapping annotation. It tells the framework that the annotated method handles the HTTP GET request. As a parameter of this annotation, a part of the URL to which the method is to be launched is indicated. In our case, it is "/", which means root of our application.

As you can see from the example above, such a small amount of code generated a lot of functionality. We owe it all to the Convention over Configuration principle, which did the tedious part of the implementation inside the Spring Boot framework for us. Let's take a deeper look at the capabilities of this tool.

A service that returns a list of books

In order to get to know the possibilities of Spring Boot better, we will analyze an example whose task will be to return a list of books saved in the application. First, we need to define what our book class will look like. In this case, it will take the form:

```
packagepl.edu.pwr.ztw.books;

public
    classbook{privateintid
    ;privateStringtitles;p
    rivateStringauthor;int
    pages;

    publicbook(intid,Stringtitles,Stringauthor,intpages)
        {this.id= id;
        this.titles=
        titles;this.author=
        author;
```

```
    }

    publicintgetId() {returnid; }
    publicStringgetTitle() {returntitles; }
    publicvoidsetTitle(Stringtitles) {this.titles= titles;
    }publicStringgetAuthor() {returnauthor; }
    publicvoidsetAuthor(Stringauthor) {this.author= author;
    }publicintgetPages() {returnpages; }
    publicvoidsetPages(intpages) {this.pages= pages; }
}
```

As you can see, there is nothing unusual here. Just a class with three fields, a constructor, and a set of setters and getters.

Let's move on to the definition of the website responsible for access to our book collection. To do this, we will first create the IBooksService interface:

```
packagepl.edu.pwr.ztw.books;

importjava.util.Collection;publi

c interfaceiBooksService{
    public abstractscollection<book>getBooks();
}
```

Nothing too complicated here either. Of course, you could omit the creation of this interface, but good programming practice requires the introduction of an interface layer to be able to freely change system elements without the need to recompile it.

The service implementation itself will take the form:

```
packagepl.edu.pwr.ztw.books;

importorg.springframework.stereotype.Service;i

mportjava.util.ArrayList;
importjava.util.Collection;i
mportjava.util.List;

@service
public classBooksServiceimplementsiBooksService{private
    staticLetter<book>booksRepo=newArrayList<>();

    static{
        booksRepo.add(newbook(1,"Flood","Henryk Sienkiewicz",936));
```

```
        booksRepo.add(newbook(2,"Wedding reception","Stanislaw
        Reymont",150));booksRepo.add(newbook(3,"Forefathers","Adam
        Mickiewicz",292));
    }

    @Override
    publiccollection<book>getBooks()
        {returnbooksRepo;
    }
```

This is a class that implements our IBooksService. This class is more like a mockup than a real book repository service. But it is on this example that it is best to see what improvements are brought by the introduction of the interface layer in the system implementation. Now let's imagine a situation in which we work in a team responsible for preparing a REST application that gives access to application resources. However, the repository software, which will probably be stored somewhere in the database, is handled by a different team. This team works in parallel to ours. The introduction of the interface layer allows you to work on the application without waiting for the access code to the real repository. Instead, we provided our own BookService class,

The mentioned service implements the only method getBooks(), which task is to return a list of all books stored in the database.

What deserves special attention is the annotation in the @Service class. This annotation tells the framework that the class is to be managed by dependency injection. This is a Spring feature that eliminates the need to use the traditional way to create an object with the word new. An instance of such annotated class will be created by Spring and its life cycle will also be beyond our interest. Like everything so far, this will happen automatically outside of us. Here, it is only necessary to remember that the @Service annotation is dedicated to classes whose task is to provide services.

In general, the Spring framework provides a series of annotations: @Component, @Service and @Controller. The @Component annotation is the most generic type for any Spring managed bean. @Repository, @Service, @Controller (@RestController) are specialized annotations derived from @Component. Thus, for classes whose task is to store and aggregate data, the best annotation is @Repository. For classes that provide @Service services, while for classes of the presentation layer and/or application API, @Controller and @RestController annotations are best suited. You can read more about the different types of annotations here:http://zetcode.com/springboot/annotations/.

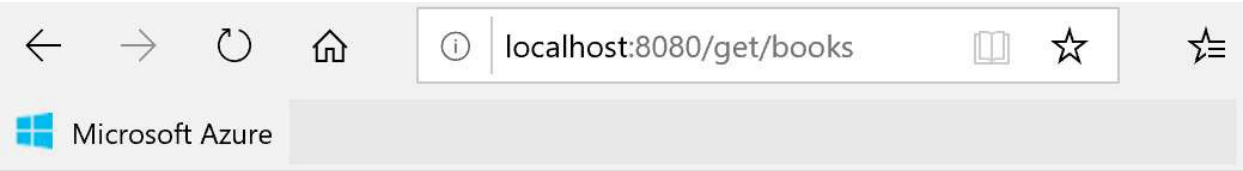
It's time to create a controller for our website. For this purpose, we will create a class according to the following listing:

```
packagepl.edu.pwr.ztw.books;
```

```
importorg.springframework.beans.factory.annotation.Autowired;
importorg.springframework.http.HttpStatus;
importorg.springframework.http.ResponseEntity;
importorg.springframework.web.bind.annotation.RequestMapping;
importorg.springframework.web.bind.annotation.RestController;

@RestController
public
    classBooksController{@Aut
    owired
    iBooksServicebooksService;

    @RequestMapping(value ="/get/books", method

    =RequestMethod.GET)publicResponseEntity<object>getBooks(){
        return newResponseEntity<>(booksService.getBooks(),HttpStatus.APPROX);
    }
```

In the aboveIn this example, the @RestController annotation has already been discussed earlier. The same applies to the @RequestMapping annotation. As you can see, the getBooks() method returns a ResponseEntity object, which takes a collection of books delivered via the IBooksService interface as the first parameter, because that's what the booksService field is all about. The question remains, then how was this field initialized with an instance of the BooksService class? In the example, there is no assignment of this object to a given field anywhere. This effect is due to the already discussed @Service annotation defined on the BooksService class and the @Autowired annotation on the booksService field in the BooksControler class. @Autowired just means that the Spring framework is asked to provide a class that implements the desired interface/type. Since we annotated the BooksService class as @Service, then the framework already had this instance under its management. So he delivered it to us at our facility. This is part of Spring's dependency injection.

So let's compile the code and run it, then go to the browser at:http://localhost:8080/get/books. We should get the following result:



[{"id":1,"title":"Potop","author":"Henryk Sienkiewicz","pages":936},
{"id":2,"title":"Wesele","author":"StanisÅ,aw Reymont","pages":150},
{"id":3,"title":"Dziady","author":"Adam Mickiewicz","pages":292}]

This is the response of our service in the form of JSON. The question arises how it happened that our book collection was returned in JSON format? The Spring Boot framework is also responsible for all this, which serializes to this format by default. There was nothing special to do. The framework is responsible for everything. Isn't it convenient?

Parameterizing the request

What if we only want a single book in response, not the entire collection? The book should be identified by a parameter. So let's look at the solution to the above problem.

Onat the beginning, we add the appropriate method to the definition of the IBooksService interface:

```
public abstractsbookgetBook(intid);
```

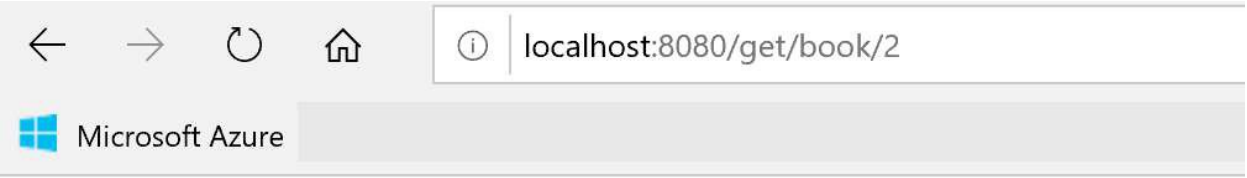
Thenwe add its implementation in the BooksService class in the form:

```
    @Override
    publicbookgetBook(intid)
        {returnbooksRepo.stream()
                .filters(b->b.getId() == id)
                .findAny()
                .orElse(null);
    }
```

In this way, we already have the functionality responsible for filtering the elements of the collection. It's time to hook it up to the controller. To do this, add the following code to the BooksControler class:

```
    @RequestMapping(value ="/get/book/{id}", method
    =RequestMethod.GET)publicResponseEntity<object>getBook(@PathVariable(
    "id")intid){
        return newResponseEntity<>(booksService.getBook(id),HttpStatus.APPROX);
```

After launching the application with the above changes after going to the browser at the addresshttp://localhost:8080/get/book/2 we get the following result:



So everything works as planned. Now let's discuss why it works. Notice that in the controller class the getBook() function has been annotated with @RequestMapping where the path has placeholder {id}. This means that the value that appears here will be available as a parameter for the application. In order to connect this parameter with the parameter of the getBook() method, before the parameter definition, add the @PathVariable annotation, in which we indicate the parameter name from the URL of the request.

A few words about HTTP request types

So far we have discussed examples of HTTP GET requests. They are ideal for all kinds of queries that are designed to return over data from the application server. But what if you wanted to write an API that would allow you to add a new book? In general, how to approach writing a full set of CRUD (Create, Read, Update, Delete) functionality. To do this, use a different type of HTTP request. The appropriate use of the HTTP request for the implemented functionality is presented in the table below:

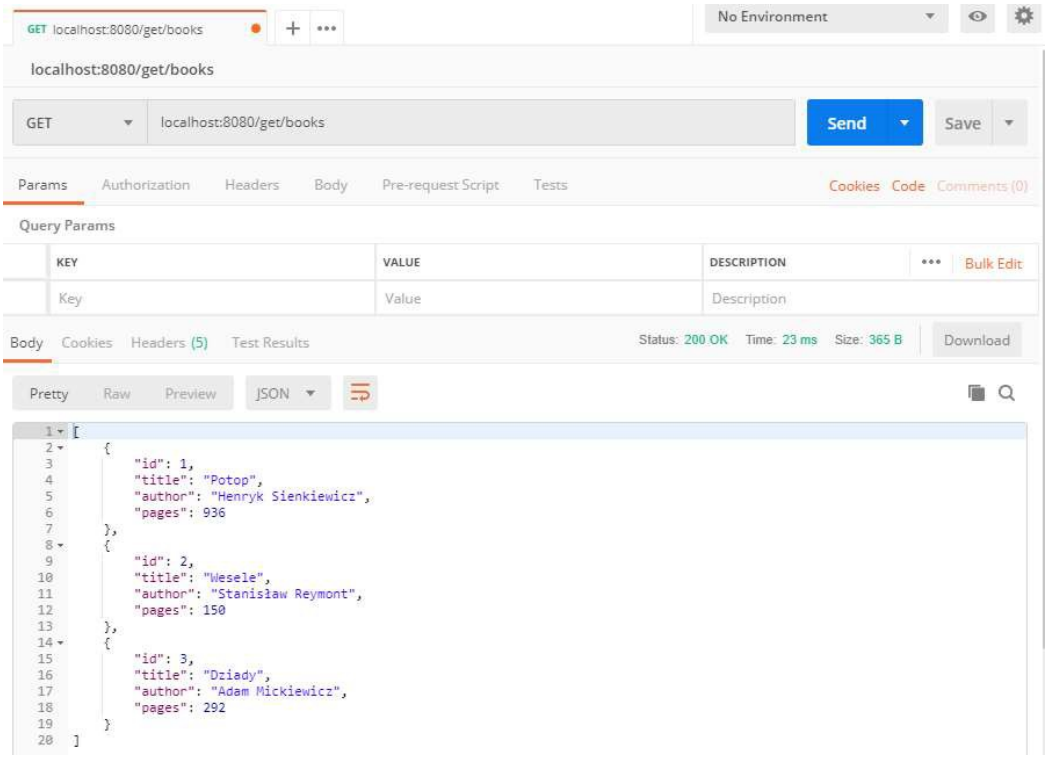
ActionHTTP	
Create	PUT/POST
Read	GET
update	POST/PUT/PATCH
Delete	DELETE

Due to limited time available, it is not possible to discuss all the examples in a given material. Therefore, for details, I refer you to the material below:https://www.tutorialspoint.com/spring_boot/spring_boot_service_components.htm

API testing

Let's spend a few more words on testing the API. While GET requests, which in their nature are intended to return a value from the server, can be easily simulated from a browser, a POST or PUT request is not trivial to achieve with the browser itself. How to deal with this? How to build such a query to test the functionality? Of course you can use a command line tool like curl. However, it is difficult to talk about the simplicity of working with the console interface here.

A better solution is, for example, the Postman application. It allows you to graphically handle complex requests. In addition to indicating the request type, we have full control over the parameters, headers, request body and other issues related to, for example, authorization. Below is the result of a query for a list of books for our sample website:



Working with the above tool is much more efficient and easier. The application can be downloaded here:https://www.postman.com/

Swagger 2 – a way to document large websites

Another way to test and document your API is with the Swagger library. It is available for many platforms and frameworks - not just Spring Boot. However, let's try to trace its basic operation on our example.

So let's connect this library. To do this, go to the maven repository (https://mvnrepository.com/), in which we search for Springfox Swagger2 libraries. In our example, we will use the maven repository in the form of:

```
<!-- https://mvnrepository.com/artifact/io.springfox/springfox-swagger2 -->
<dependencies>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.9.2</version>
</dependencies>
```

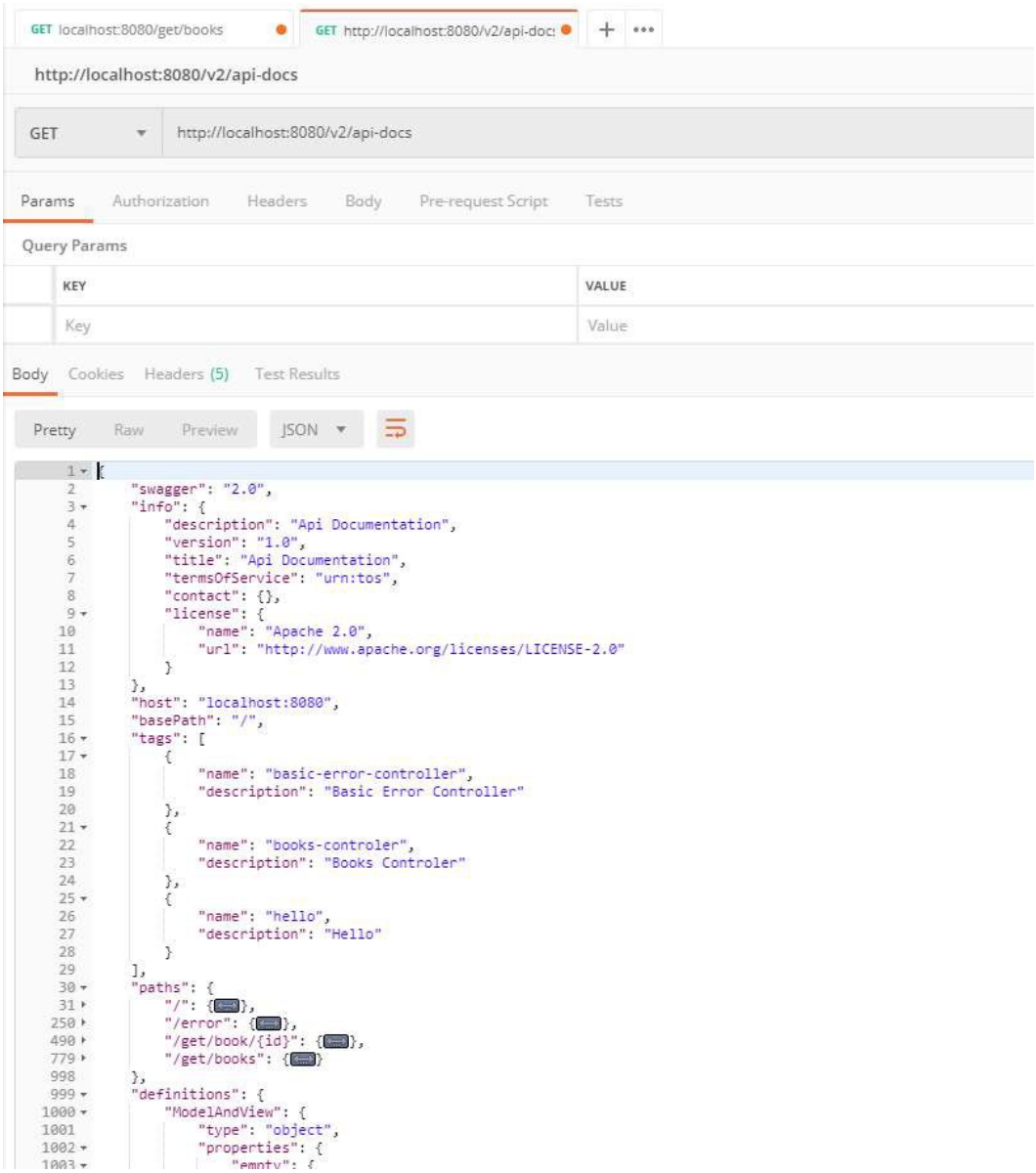
mentioneddependency is added to the pom.xml file between the <dependencies> tags.

Only one more change is needed to run Swagerr. We need to properly annotate our app's runtime class. In our case, this is the BooksApplication class, which should look like this:

```
@SpringBootApplication@Enable
Swagger2
public classBooksApplication{
    // further class body
```

@EnableSwagger2 annotation enablesAPI documentation functionality. This is a deliberate requirement to make sure you don't expose the API to the world by unknowingly adding dependencies to Swagger. This could compromise the security of apps that shouldn't brag about how they work.

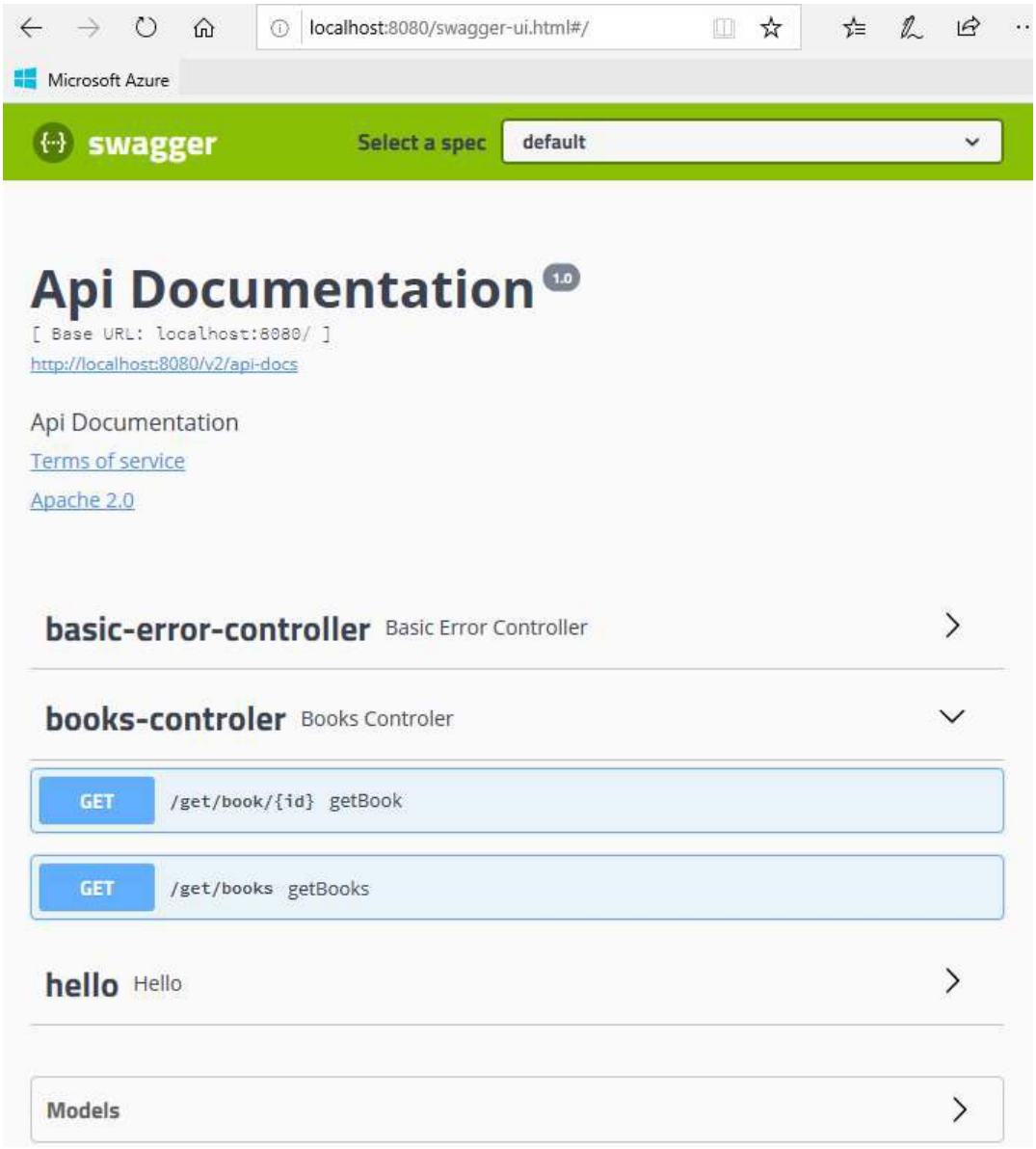
So we'll run our example after the changes and use Postman to query the addresshttp://localhost:8080/v2/api-docs. As you can see in the picture below, a given endpoint returns detailed information in JSON format about our API:



Everything is fine, but you could say that such JSON is still difficult to read. So let's go a step further and add a friendly GUI to our example. Swagger also provides the Swagger UI library. To use it, let's add a dependency to the pom.xml file:

```
<dependencies>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.9.2</version>
</dependencies>
```

After starting the application and going to the addresshttp://localhost:8080/swagger-ui.html documentation will be available on the website. Below is a view of the BooksControler interface:



It is worth noting that after expanding the definition of a single endpoint, we can immediately test its operation directly from the browser.

As you can see from the above presentation, the Swagger library is a very interesting solution for working on the application API. The discussed example touches only very superficially the capabilities of this tool. You can read more about working with this library here:https://swagger.io/