Fundusze Europejskie
Wiedza Edukacja Rozwój

Politechnika Wrocławska

Unia Europejska
Europejski Fundusz Społeczny

„ZPR PWr – Zintegrowany Program Rozwoju Politechniki Wrocławskiej”

Politechnika Wrocławska

Wydział Informatyki i Zarządzania

Zaawansowane Technologie Webowe

Laboratorium

Temat:	REST Api na przykładzie Spring Boot
Opracował:	mgr inż. Piotr Jóźwiak
Data:	lipiec 2020
Liczba godzin:	2 godziny

Table of Contents

Wstęp

Na dzisiejszym laboratorium omówimy REST API na przykładzie jednej z najpopularniejszych technologii wykorzystywanych do oprogramowania backandu – **Spring Boot**.

Spring Boot jest Open Sourcowym frameworkiem dostępnym w języku Java stworzonym do pisania mikro serwisów. Został on opracowany na bazie frameworka Spring wprowadzając szereg udogodnień dla programistów. Najważniejsze z nich to wprowadzenie zasady „*Convention over Configuration*", która minimalizuje wymogi związane z konfiguracją oprogramowania wprowadzając szereg dobrze dobranych wartości domyślnych oraz przemyślanych konwencji implementacji kodu. Drugą istotną zaletą tego frameworka jest uproszczenie zarządzania zależnościami, która jest całkiem uciążliwa w czystym Springu. Wszystkie wprowadzone zmiany do SB mają na celu przyspieszenie wytwarzania oprogramowania względem jego bazowego rozwiązania Spring.

REST Api

Zacznijmy jednak od rozszyfrowania skrótu REST API. Samo słowo REST pochodzi od słów: „*Representational State Transfer*". Czym zatem jest REST? Jest ono zdefiniowanym stylem architektury oprogramowania wprowadzającym zestaw reguł opisujących definiowanie zasobów oraz sposób dostępu do nich. Został on zaproponowany przez Roya Feldinga w 2000 roku. Samo słowo API pochodzi od słów „Application Programming Interface" – co oznacza zestaw reguł definiujących interfejs komunikacji z daną aplikacją.

Aby móc posługiwać się określeniem RESTful application to musi ona spełnić szereg wymogów:

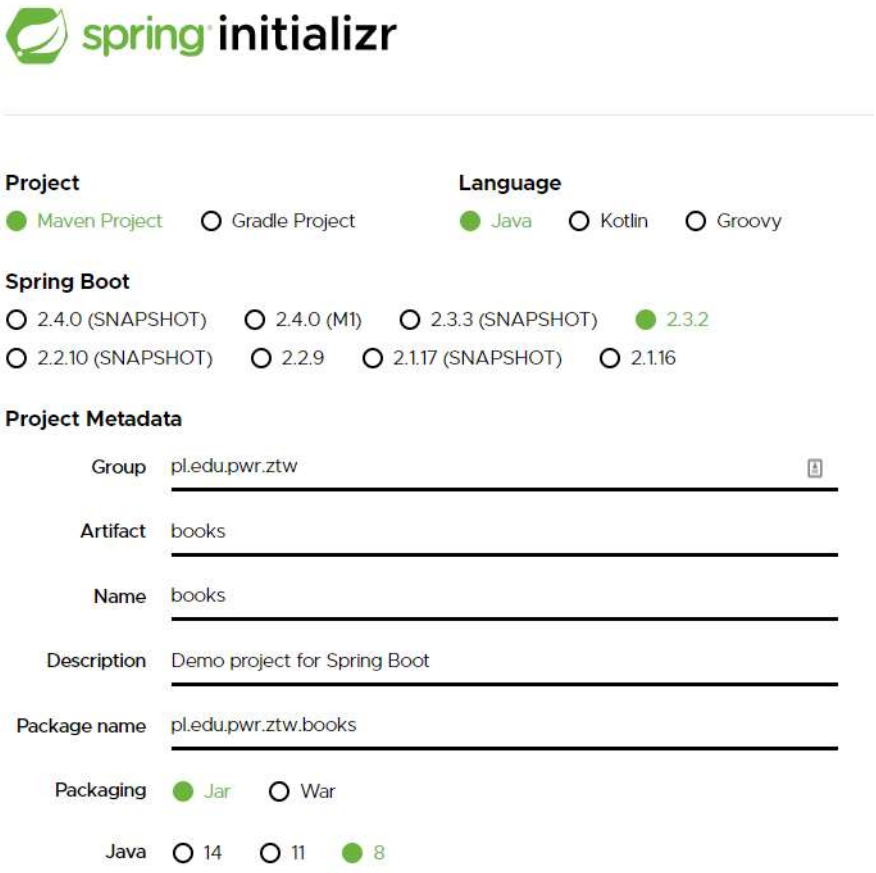
1. Separacja forntendu od backendu. Interfejs użytkownika nie ma prawa ingerować w to, co oraz jak dzieje się po stronie serwera. Ta zasada działa także w drugą stronę.
2. Bezstanowość – oznacza to, że żądanie od klienta musi posiadać komplet informacji, ponieważ serwer nie posiada wiedzy o poprzednich żądaniach danego klienta.
3. Cacheability – zwracane odpowiedzi powinny jasno definiować czy mogą zostać cachowane czy nie.
4. Warstwowość – klient nie wie czy łączy się bezpośrednio z endpointem czy poprzez jakiekolwiek proxy lub balancer. Innymi słowy rozwiązanie musi dawać możliwość wprowadzania warstw pośrednich pomiędzy klienta i serwer, które nie zakłócą komunikacji.

Przejdźmy zatem do praktyki i przeanalizujmy przykład pokazujące sposób pracy ze środowiskiem Spring Boot.

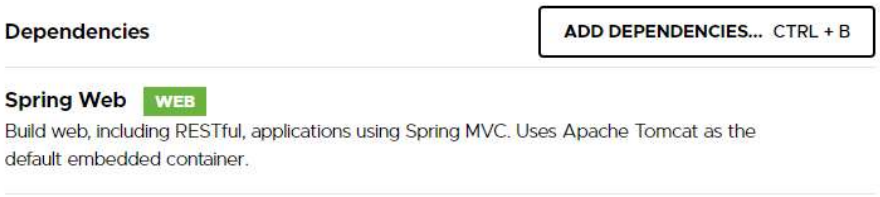
Spring Boot – jak zacząć

Najprostszym sposobem rozpoczęcia pracy jest skorzystanie z ***Spring Initializr***, który w kilku prostych krokach przygotuje nam niezbędną konfigurację oraz hierarchię folderów. Aby skorzystać z tego rozwiązania należy przejść do strony: https://start.spring.io/. Ukaże się formularz, w którym trzeba odpowiedzieć na klika pytań, m. in. o typ projektu (Maven czy Gradle), język (Java, Kotlin, Groovy), wersję Spring Boot oraz ustawienia naszego pakietu w części Metadata.

W naszym przykładzie skupimy się na opracowaniu prostej aplikacji umożliwiającej dostęp do katalogu książek. Dlatego dla powyższego przykładu generujemy kod z poniższymi ustawieniami:



W sekcji *Dependencies* dodajemy zależność do Spring Web:



Szablon pobieramy poprzez przycisk **Generate** w postaci pliku zip. Po ściągnięciu rozpakowujemy archiwum i otwieramy projekt. Do pracy z projektem można skorzystać ze środowiska IDE IntelliJ lub Eclipse lub innym wspierającym pracę ze środowiskiem Java.

Teraz jest dobry moment na skompilowanie pustego szablonu projektu oraz uruchomienie go i zaobserwowanie co się stanie. W trakcie uruchamiania na output wysyłane są komunikaty. Po skończonym uruchomieniu wyglądają one tak jak na poniższym obrazku:

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: 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\OneI
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
```

Warto tutaj od razu zauważyć, że w trakcie kompilacji został obsadzony serwer WWW **Tomcat** i w momencie uruchamiania naszej aplikacji jest on wykorzystywany do obsługi żądań. Sami nie musimy instalować serwera WWW (choć możemy to zrobić ręcznie). Zatem zobaczmy co pojawi się w przeglądarce jak wejdziemy na adres tego serwera. *Tomcat* został uruchomiony na adresie http://localhost:8080. Po wejściu na tę stronę otrzymujemy poniższy wynik:

Jest to strona informująca o błędzie, że pod danym adresem nie ma zdefiniowanego mapingu, o czym opowiem za chwilę. Innymi słowy pod tym adresem nie ma żadnego serwisu/strony WWW. Rozwiązaniem tego problemu zajmiemy się w następnym kroku.

Naprawmy wyświetlony błąd. Prześledzenie poniższego procesu pozwoli nam zrozumieć podstawy działania kontrolera. Zatem utwórzmy klasę, która wygląda tak:

```java
package pl.edu.pwr.ztw.books;

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

@RestController
public class Hello {

    @RequestMapping("/")
    public String run() {
        return "Hello ! I'm, running";
    }
}
```

Zanim omówimy działanie powyższego kodu proponuję zobaczenie efektu jego działania:

Jak widać z powyższego obrazka, teraz jako strona główna wyświetla się nam napis. Pora na omówienie działania kodu, który wygenerował powyższy rezultat;

Utworzyliśmy klasę o nazwie **`Hello`**. Pierwszą istotną rzeczą na jaką należy zwrócić uwagę jest adnotacja tej klasy **`@RestController`**. Adnotacja ta informuje framework, że klasa ta ma być zarządzanym Beanem. Powoduje to, że każde żądanie HTTP będzie przetwarzane przez klasy z w/w adnotacją. Zatem nasza klasa jest kontrolerem żądań HTTP.

W ciele tej klasy utworzyliśmy jedną metodę o nazwie **`run()`**. Działanie jej jest banalne i sprowadza się do zwrócenia napisu, który pojawił się w przeglądarce. Ciekawszym elementem tej części kodu jest kolejna adnotacja **`@RequestMapping`**. Informuje ona framework, że anotowana metoda zajmuje się obsługą żądania HTTP GET. Jako parametr tej anotacji wskazuje się część adresu URL na jaki dana metoda ma zostać uruchomiona. W naszym przypadku jest to „/”, co oznacza root naszej aplikacji.

Jak widać z powyższego przykładu tak niewielka ilość kodu wygenerowała bardzo dużą funkcjonalność. To wszystko zawdzięczamy zasadzie *Convention over Configuration*, która sama za nas wykonała żmudną cześć implementacji wewnątrz frameworka Spring Boot. Przyjrzyjmy się głębiej możliwościom tego narzędzia.

Serwis zwracający listę książek

Aby lepiej poznać możliwości Spring Boot przeanalizujemy przykład, którego zadaniem będzie zwracanie listy książek zapisanych w aplikacji. Na początek musimy zdefiniować jak będzie wyglądała nasza klasa opisująca książkę. W tym przypadku przyjmie ona postać:

```
package pl.edu.pwr.ztw.books;

public class Book {
    private int id;
    private String title;
    private String author;
    int pages;

    public Book(int id, String title, String author, int pages) {
        this.id = id;
        this.title = title;
        this.author = author;
        this.pages = pages;
```

```
    }

    public int getId() { return id; }
    public String getTitle() { return title; }
    public void setTitle(String title) { this.title = title; }
    public String getAuthor() { return author; }
    public void setAuthor(String author) { this.author = author; }
    public int getPages() { return pages; }
    public void setPages(int pages) { this.pages = pages; }
}
```

Jak widać nie ma tutaj nic nadzwyczajnego. Jedynie klasa z trzema polami, konstruktorem i zestawem seterów i geterów.

Przejdźmy do definicji serwisu odpowiedzialnego za dostęp do naszej kolekcji książek. W tym celu utworzymy w pierwszej kolejności interfejs **`IBooksService`**:

```
package pl.edu.pwr.ztw.books;

import java.util.Collection;

public interface IBooksService {
    public abstract Collection<Book> getBooks();
}
```

Tutaj także nie odnajdujemy nic skomplikowanego. Oczywiście można by było pominąć tworzenie tego interfejsu, ale dobra praktyka programistyczna narzuca wprowadzenie warstwy interfejsów, aby móc dowolnie zmieniać elementy systemu bez potrzeby jego rekompilacji.

Sama implementacja serwisu przyjmie postać:

```
package pl.edu.pwr.ztw.books;

import org.springframework.stereotype.Service;

import java.util.ArrayList;
import java.util.Collection;
import java.util.List;

@Service
public class BooksService implements IBooksService {
    private static List<Book> booksRepo = new ArrayList<>();

    static {
        booksRepo.add(new Book(1,"Potop", "Henryk Sienkiewicz", 936));
```

```
        booksRepo.add(new Book(2,"Wesele", "Stanisław Reymont", 150));
        booksRepo.add(new Book(3,"Dziady", "Adam Mickiewicz", 292));
    }

    @Override
    public Collection<Book> getBooks() {
        return booksRepo;
    }
}
```

Jest to klasa implementująca nasz serwis **IBooksService**. Klasa ta przypomina bardziej zaślepkę (*mockup*) niż prawdziwy serwis z obsługą repozytorium ksiażek. Ale to właśnie na tym przykładzie najlepiej widać jakie udogodnienia niesie wprowadzenie warstwy interfejsów w implementacji systemu. Wyobraźmy sobie teraz taką sytuację, w której pracujemy w zespole odpowiedzialnym za przygotowanie aplikacji REST dającej dostęp do zasobów aplikacji. Jednakże oprogramowanie repozytoriów, które zapewne będą zapisane gdzieś w bazie danych zajmuje się inny zespół. Ten zespół pracuje równolegle do naszego. Wprowadzenie warstwy interfejsów umożliwia prowadzenie prac nad aplikacją bez oczekiwania na kod dostępu do rzeczywistego repozytorium. Zamiast tego dostarczyliśmy własną klasę **BooksService**, która posiada niewielkie *repo* w postaci listy książek inicjowanych w statycznym kostruktorze.

Wspomniany serwis implementuje jedyną metodę **getBooks()**, które zadaniem jest zwrócenie listy wszystkich książek zapisanych w bazie danych.

To co zasługuje na specjalną uwagę jest adnotacja w klasie jaką jest **@Service**. Adnotacja ta informuje framework, że dana klasa ma być zarządzana przez mechanizm wstrzykiwania zależności. Jest to funkcjonalność Springa, która niweluje potrzebę wykorzystywania tradycyjnego sposobu na utworzenie obiektu poprzez słowo **new**. Instancja tak adnotowanej klasy zostanie utworzona przez Springa a cykl jej życia także będzie poza naszym zainteresowaniem. Jak wszystko do tej pory, będzie się to działo automatycznie poza nami. Tutaj należy jedynie zapamiętać, że adnotacja **@Service** jest dedykowana klasom, których zadaniem jest dostarczanie usług.

Ogólnie rzecz ujmując framework Spring dostarcza szereg adnotacji: **@Component**, **@Service** and **@Controller**. Adnotacja **@Component** jest najbardziej generycznym typem dla dowolnego *beana* zarządzanego przez *Springa*. **@Repository**, **@Service**, **@Controller** (**@RestController**) są wyspecjalizowanymi adnotacjami wywodzącymi się z **@Component**. Tym samym dla klas, których zadaniem jest przechowywanie i agregowanie danych najlepszą adnotacją jest **@Repository**. Dla klas dostarczających usługi **@Service**, natomiast dla klas warstwy prezentacji lub/i API aplikacji najlepiej pasują adnotacje **@Controller** oraz **@RestController**. Więcej o różnego rodzaju adnotacjach można przeczytać tutaj: http://zetcode.com/springboot/annotations/.

Pora na utworzenie kontrolera dla naszego serwisu. W tym celu utworzymy klasę według poniższego listingu:

```
package pl.edu.pwr.ztw.books;
```

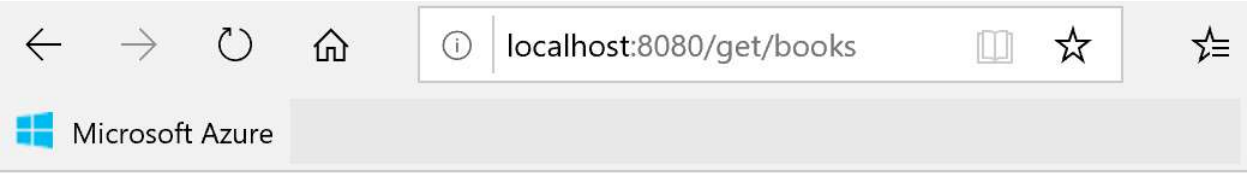
```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class BooksControler {
    @Autowired
    IBooksService booksService;

    @RequestMapping(value = "/get/books", method = RequestMethod.GET)
    public ResponseEntity<Object> getBooks(){
        return new ResponseEntity<>(booksService.getBooks(), HttpStatus.OK);
    }
}
```

W powyższym przykładzie adnotacja **@RestController** została już omówiona wcześniej. Podobnie ma się rzecz z adnotacją **@RequestMapping**. Jak widać metoda **getBooks()** zwraca obiekt **ResponseEntity**, który jako pierwszy parametr przyjmuje kolekcję książek dostarczonych poprzez interfejs **IBooksService**, bo tym właśnie jest pole **booksService**. Pozostaje pytanie, jak w takim razie to pole zostało zainicjowane instancją klasy **BooksService**? W przykładzie nie ma nigdzie przypisania tego obiektu do danego pola. Za ten efekt odpowiada wcześniej omówiona już adnotacja **@Service** zdefiniowana na klasie BooksService oraz adnotacja **@Autowired** na polu **booksService** w klasie **BooksControler**. **@Autowired** oznacza tyle, że framework Springa jest proszony o dostarczenie klasy implementującej żądany interfejs/typ. Ponieważ my adnotowaliśmy klasę **BooksService** jako **@Service**, to też framework miał już tą instancję pod swoim zarządem. Zatem dostarczył ją nam do naszego obiektu. Jest to cześć wstrzykiwania zależności Springa.

Zatem skompilujmy kod i włączmy go, po czym przejdźmy do przeglądarki pod adres: http://localhost:8080/get/books. Powinniśmy otrzymać poniższy rezultat:



[{"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}]

Jest to odpowiedź naszego serwisu w postaci JSON. Powstaje pytanie jak to się stało, że nasza kolekcja książek została zwrócona w formacie JSON? Za to wszystko także odpowiedzialny jest framework Spring Boot, który domyślnie wykonuje serializację do tego formatu. Nic specjalnie nie trzeba było robić. Za wszystko odpowiedzialny jest framework. Czyż nie jest to wygodne?

Parametryzowanie żądania

A co w sytuacji, w której chcielibyśmy jedynie w odpowiedzi otrzymać pojedynczą książkę, a nie całą kolekcję. Książka powinna być identyfikowana na podstawie parametru. Prześledźmy zatem rozwiązanie powyższego problemu.

Na początek dopisujemy do definicji interfejsu **`IBooksService`** odpowiednią metodę:

```
public abstract Book getBook(int id);
```

Następnie dopisujemy jej implementację w klasie **`BooksService`** w postaci:

```
    @Override
    public Book getBook(int id) {
        return booksRepo.stream()
                .filter(b -> b.getId() == id)
                .findAny()
                .orElse(null);
    }
```

W ten oto sposób mamy już gotową funkcjonalność odpowiedzialną za filtrowanie elementów kolekcji. Pora podpiąć to do kontrolera. W tym celu w klasie **`BooksControler`** dopisujemy poniższy kod:

```
    @RequestMapping(value = "/get/book/{id}", method = RequestMethod.GET)
    public ResponseEntity<Object> getBook(@PathVariable("id") int id){
        return new ResponseEntity<>(booksService.getBook(id), HttpStatus.OK);
    }
```

Po uruchomieniu aplikacji z powyższymi zmianami po przejściu do przeglądarki pod adres http://localhost:8080/get/book/2 otrzymujemy poniższy rezultat:

Zatem całość działa tak jak sobie zaplanowaliśmy. Omówmy teraz dlaczego działa. Zwróćmy uwagę, że w klasie kontrolera funkcja **`getBook()`** została adnotowana **`@RequestMapping`**, w którym ścieżka posiada placeholder **`{id}`**. Oznacza to, że wartość, która się w tym miejscu pojawi będzie dostępna jako parametr dla aplikacji. Aby połączyć ten parametr z parametrem metody **`getBook()`** należy przed definicją parametru dopisać adnotację **`@PathVariable`**, w której wskazujemy nazwę parametru z URL żądania.

Kilka słów o typach żądań HTTP

Do tej pory omówiliśmy sobie przykłady żądań typu HTTP GET. Idealnie nadają się one wszelkiego rodzaju zapytań, które mają na celu zwrócić nad dane z serwera aplikacyjnego. Co jednak, gdyby chcieć napisać API, które umożliwi dodanie nowej książki? Ogólnie rzecz ujmując, jak podejść do napisania pełnego zestawu funkcjonalności CRUD (Create, Read, Update, Delete). W tym celu należy skorzystać z innego rodzaju żądań HTTP. Odpowiednie użycie żądania HTTP do realizowanej funkcjonalności przedstawia poniższa tabela:

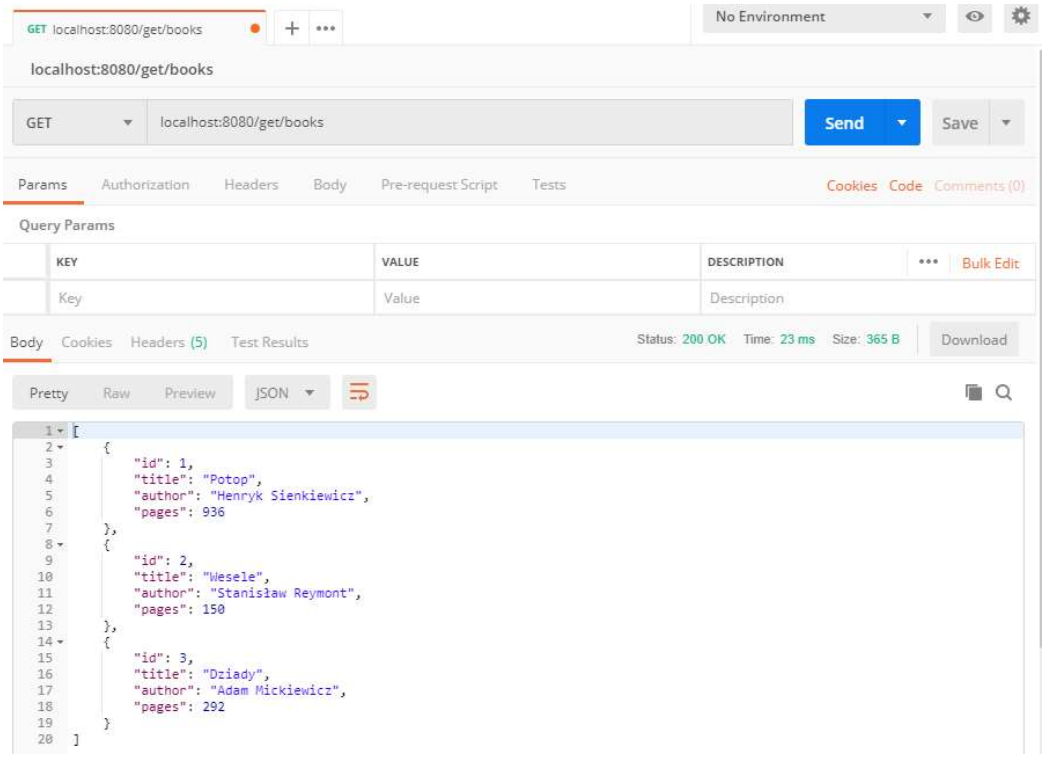
Działanie	HTTP
Create	PUT/POST
Read	GET
Update	POST/PUT/PATCH
Delete	DELETE

Ze względu na ograniczone możliwości czasowe nie jest możliwe omówienie wszystkich przykładów w danym materiale. Dlatego po szczegóły odsyłam do chociażby poniższego materiału: https://www.tutorialspoint.com/spring_boot/spring_boot_service_components.htm

Testowanie interfejsu API

Poświęćmy jeszcze kilka słów zagadnieniu testowania interfejsu API. O ile żądania typu GET, które w swej naturze mają za zadanie zwrócić wartość z serwera bardzo łatwo można symulować z przeglądarki, to już żądanie typu POST czy PUT nie jest trywialne do osiągnięcia samą przeglądarką. Jak sobie z tym poradzić? Jak zbudować takie zapytanie, aby przetestować funkcjonalność? Oczywiście można posłużyć się narzędziem z linii komend takim jak **`curl`**. Jednak trudno tutaj mówić o prostocie pracy z interfejsem konsolowym.

Lepszym rozwiązaniem jest np. aplikacja **Postman**. Pozwala ona w graficzny sposób ogarnąć skomplikowane żądania. Oprócz wskazania typu żądania mamy pełną kontrolę nad parametrami, nagłówkami, ciałem żądania i innymi zagadnieniami związanymi chociażby z autoryzacją. Poniżej przedstawiam wynik zapytania o listę książek dla naszego przykładowego serwisu:



Praca z powyższym narzędziem jest dużo bardziej wydajna i łatwiejsza. Aplikację można pobrać tutaj: https://www.postman.com/

Swagger 2 – sposób na dokumentację dużych serwisów

Innym sposobem testowania i dokumentowania interfejsu API jest biblioteka **Swagger**. Jest ona dostępna dla wielu platform i frameworków – nie tylko Spring Boot. Postarajmy się jednak prześledzić podstawowe jej działanie na naszym przykładzie.

Podłączmy zatem tą bibliotekę. W tym celu przechodzimy do repozytorium **mavena** (https://mvnrepository.com/), w którym wyszukujemy biblioteki Springfox Swagger2. W naszym przykładzie skorzystamy z repozytorium maven w postaci:

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

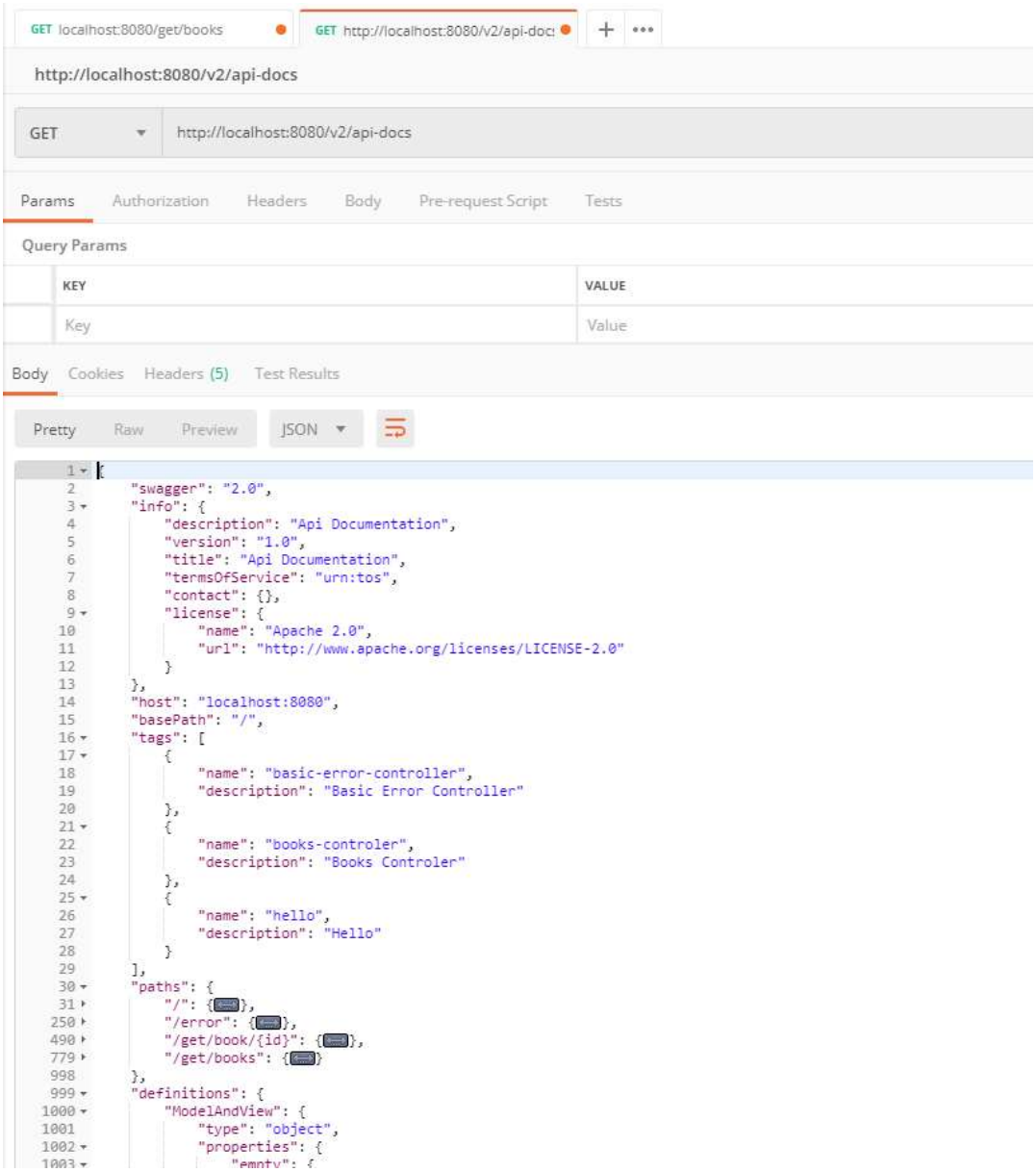
Wspomnianą zależność dodajemy do pliku **`pom.xml`** pomiędzy tagi **`<dependencies>`**.

Do uruchomienia Swagerra jest potrzebna jeszcze tylko jedna zmiana. Musimy odpowiednio dodać adnotację na klasie uruchomieniowej naszej aplikacji. W naszym przypadku jest to klasa **`BooksApplication`**, która powinna wyglądać następująco:

```
@SpringBootApplication
@EnableSwagger2
public class BooksApplication {
    // dalej ciało klasy
```

Anotacja **@EnableSwagger2** włącza funkcjonalność dokumentowania API. Jest to celowy wymóg, aby mieć pewność, że poprzez nieświadome dodanie zależności do Swaggera nie wystawiać API na świat. Mogłoby to naruszać bezpieczeństwo aplikacji, które nie powinny się chwalić, jak działają.

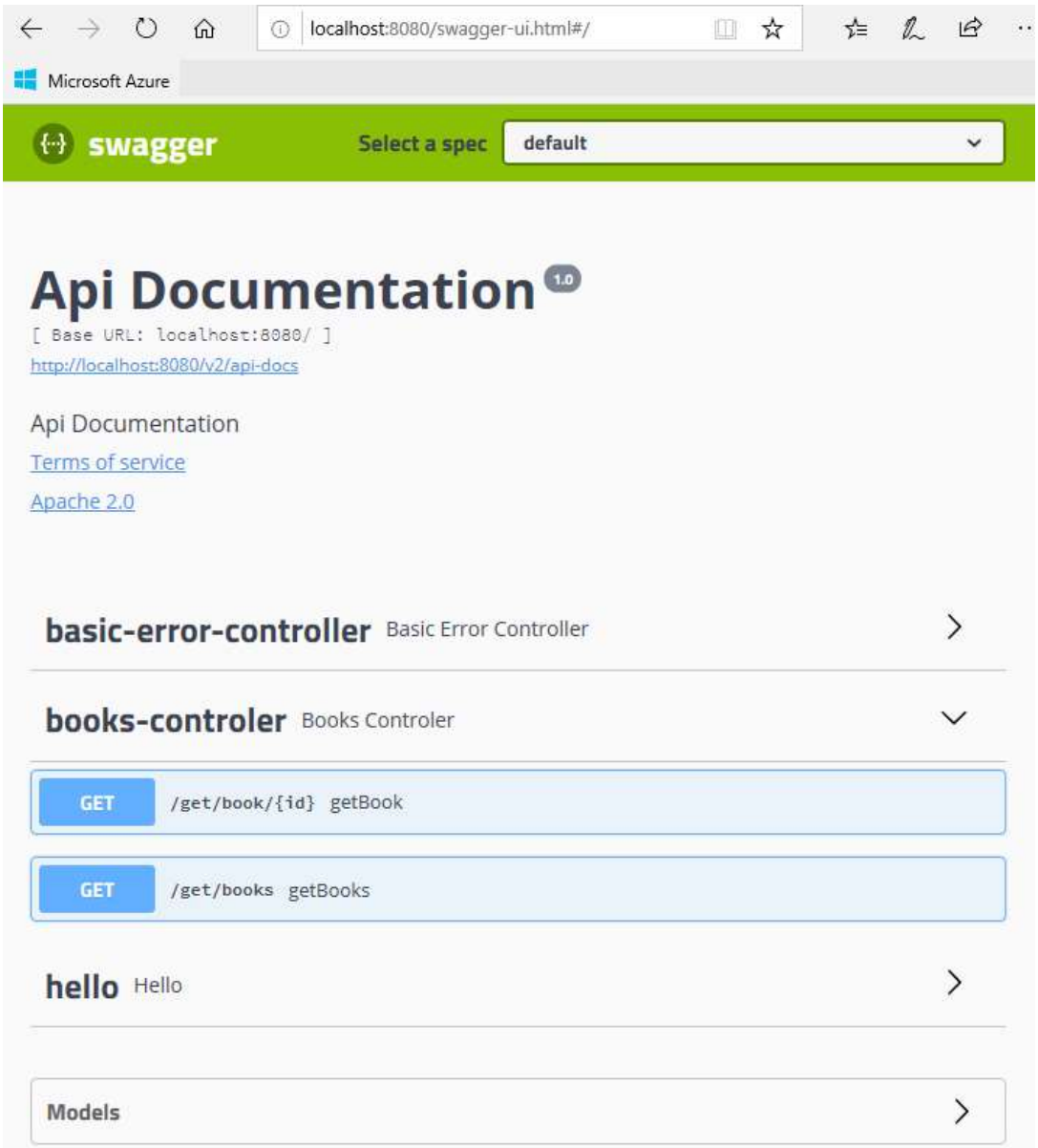
Zatem uruchomimy nasz przykład po zmianach i korzystając z Postmana odpytajmy adres http://localhost:8080/v2/api-docs. Jak widać na poniższym obrazku dany endpoint zwraca szczegółowe informacje w formacie JSON o naszym API:



Wszystko fajnie, tylko można by rzec, że takiego JSONa dalej trudno się czyta. Zatem pójdźmy o krok dalej i dodajmy przyjazny interfejs graficzny do naszego przykładu. Swagger dostarcza także bibliotekę **Swagger UI**. Aby z niej skorzystać dodajmy zależność do pliku **`pom.xml`**:

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

Po uruchomieniu aplikacji i przejściu pod adres http://localhost:8080/swagger-ui.html ukaże się nam dokumentacja w postaci strony internetowej. Poniżej widok interfejsu **`BooksControler`**:



Warto zwrócić uwagę, że po rozwinięciu definicji pojedynczego endpointu możemy od razu przetestować jego działanie wprost z przeglądarki.

Jak widać z powyższej prezentacji biblioteka Swagger jest bardzo ciekawym rozwiązaniem do pracy nad API aplikacyjnym. Omówiony przykład dotyka jedynie bardzo powierzchownie możliwości tego narzędzia. Więcej o pracy z tą biblioteką można przeczytać tutaj: https://swagger.io/