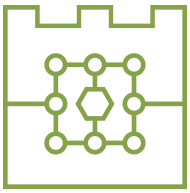


Programowanie w języku Java

Laboratorium nr 5

Politechnika Krakowska
Wydział Informatyki i Telekomunikacji
Katedra Informatyki

2024

Spring Boot

Dokumentacja:
https://spring.io/projects/spring-framework
Czym jest Spring Boot?

- Spring Boot to framework, który upraszcza budowę aplikacji Java, szczególnie tych zorientowanych na REST API i bazę danych.
- Automatycznie konfiguruje serwer (np. Tomcat), obsługę bazy danych i inne komponenty.

Kluczowe pojęcia:

- **@SpringBootApplication:** Główna adnotacja w Spring Boot. Łączy w sobie `@Configuration`, `@EnableAutoConfiguration` i `@ComponentScan`.
- **Tomcat:** Wbudowany serwer aplikacyjny, na którym działa Spring Boot.

1 REST API

Czym jest REST API?

- REST (Representational State Transfer) to styl architektury, który umożliwia komunikację między klientem (np. Postman) a serwerem.
- API REST udostępnia dane w formacie JSON poprzez HTTP.

Typowe operacje w REST API:

- **GET:** Pobranie danych.
- **POST:** Dodanie nowych danych.
- **PUT/PATCH:** Aktualizacja danych.
- **DELETE:** Usunięcie danych.

Adnotacje w Spring REST:

- **@RestController:** Klasa obsługująca żądania REST API.
- **@RequestMapping:** Definiuje bazowy URL dla endpointów w kontrolerze.
- **@GetMapping, @PostMapping:** Definiują operacje dla metod GET, POST itd.

JPA i Hibernate

Czym jest JPA?

- Java Persistence API (JPA) to standard mapowania obiektowo-relacyjnego (ORM), który pozwala na konwersję obiektów Javy na tabele w bazie danych i odwrotnie.

Kluczowe pojęcia JPA:

- **@Entity:** Klasa oznaczona tą adnotacją jest mapowana na tabelę w bazie danych.
- **@Id:** Definiuje klucz główny tabeli.
- **@GeneratedValue:** Automatyczne generowanie wartości klucza głównego.
- **@Embedded:** Umożliwia osadzenie obiektu w innej encji.

Warstwowa architektura aplikacji

Co to jest warstwowa architektura?

- Architektura aplikacji podzielona na warstwy odpowiedzialne za różne aspekty:
  1. **Kontroler:** Odpowiada za przyjmowanie i obsługę żądań HTTP.
  2. **Serwis:** Zawiera logikę biznesową aplikacji.
  3. **Repozytorium:** Obsługuje komunikację z bazą danych.

Kluczowe adnotacje w Spring Boot

Adnotacja	**Opis**
@SpringBootApplication	Oznacza główną klasę uruchamiającą aplikację Spring Boot.
@Entity	Klasa jest mapowana na tabelę w bazie danych.
@Id	Oznacza pole jako klucz główny tabeli.
@GeneratedValue	Automatyczne generowanie wartości dla klucza głównego.
@RestController	Klasa obsługująca żądania REST API.
@GetMapping	Obsługuje żądania HTTP GET.
@PostMapping	Obsługuje żądania HTTP POST.
@RequestBody	Mapuje dane JSON z żądania HTTP na obiekt Javy.

Cel laboratorium

Celem laboratorium jest migracja istniejącego projektu Maven do frameworka Spring Boot. Projekt musi:

- implementować warstwową architekturę,
- obsługiwać bazę danych PostgreSQL za pomocą JPA,
- tworzyć REST API umożliwiające komunikację z aplikacją.

Krok 1: Konfiguracja projektu Maven

1. Otwórz plik `pom.xml`.
2. Zmień konfigurację rodzica na Spring Boot:

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.1.4</version>
    <relativePath/>
</parent>
```

3. Dodaj zależności:

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
    </dependency>
</dependencies>
```

4. Dodaj wtyczkę Maven Spring Boot:

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

5. Przeładuj projekt Maven:

```
mvn clean install
```

Krok 2: Utwórz klasę startową Spring Boot

Dodaj klasę `MainApplication` w katalogu `src/main/java/org/example`:

```
package org.example;

import org.springframework.boot.SpringApplication;
import
    ↪ org.springframework.boot.autoconfigure.SpringBootApplication;
```

```java

@SpringBootApplication
public class MainApplication {
    public static void main(String[] args) {
        SpringApplication.run(MainApplication.class, args);
    }
}
```

Krok 3: Dostosuj klasy modelu do JPA

Przykład klasy `Rectangle`:

```java
@Entity
@Table(name = "rectangles")
public class Rectangle extends Shape {
    private double width;
    private double height;

    public Rectangle() {}

    public Rectangle(double width, double height, Color
        ↪ color) {
        super(color);
        this.width = width;
        this.height = height;
    }

    @Override
    public double getArea() {
        return width * height;
    }

    @Override
    public double getPerimeter() {
        return 2 * (width + height);
    }
}
```

Krok 4: Konfiguracja bazy danych

Otwórz plik `application.properties` i dodaj konfigurację:

```
spring.datasource.url=jdbc:postgresql://localhost:5432/your_database
spring.datasource.username=your_username
spring.datasource.password=your_password
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
```

Krok 5: Tworzenie repozytorium

Utwórz interfejs `ShapeRepository`:

```
@Repository
public interface ShapeRepository extends
    ↪ JpaRepository<Shape, Long> {
}
```

Krok 6: Tworzenie serwisu

Utwórz klasę `ShapeService`:

```
@Service
public class ShapeService {
    private final ShapeRepository shapeRepository;

    public ShapeService(ShapeRepository shapeRepository) {
        this.shapeRepository = shapeRepository;
    }

    public Shape saveShape(Shape shape) {
        return shapeRepository.save(shape);
    }

    public List<Shape> getAllShapes() {
        return shapeRepository.findAll();
    }
}
```

Krok 7: Tworzenie kontrolera REST

Utwórz klasę `ShapeController`:

```
@RestController
@RequestMapping("/api/shapes")
public class ShapeController {
    private final ShapeService shapeService;

    public ShapeController(ShapeService shapeService) {
        this.shapeService = shapeService;
    }

    @GetMapping
    public List<Shape> getAllShapes() {
        return shapeService.getAllShapes();
    }

    @PostMapping
    public Shape createShape(@RequestBody Shape shape) {
        return shapeService.saveShape(shape);
    }
}
```

Krok 8: Testowanie API

1. Otwórz Postman i utwórz nowe żądanie:
   - Metoda: **POST**
   - URL: `http://localhost:8080/api/shapes`
   - Body (JSON):

```
{
    "color": {
        "red": 255,
        "green": 0,
        "blue": 0,
        "alpha": 255
    },
    "width": 10.0,
    "height": 5.0
}
```

2. Kliknij **Send**, aby wysłać żądanie.
3. Sprawdź odpowiedź w formacie JSON:

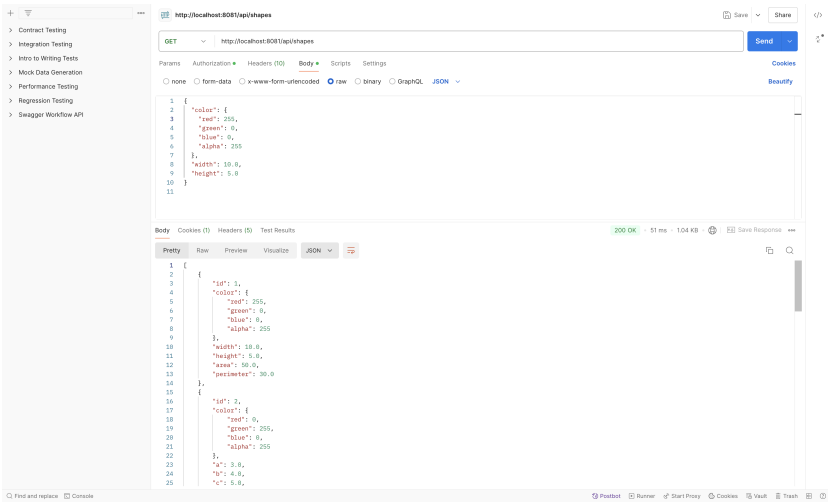
```
{
    "id": 1,
    "color": {
        "red": 255,
        "green": 0,
        "blue": 0,
        "alpha": 255
    },
    "width": 10.0,
    "height": 5.0
}
```

4. Możesz również wykonać żądanie **GET**:
   - Metoda: **GET**
   - URL: `http://localhost:8080/api/shapes`

   Wynik będzie wyglądał następująco:

```
[
    {
        "id": 1,
        "color": {
            "red": 255,
            "green": 0,
            "blue": 0,
            "alpha": 255
        },
        "width": 10.0,
        "height": 5.0
    }
]
```

Screen z aplikacji Postman:



Rysunek 2: Postman