Laboratorium 7. *Spring Boot* i *REST API*

Cel zajęć

Realizacja zadań z niniejszego laboratorium pozwoli studentom poznać zasady tworzenia warstwy serwerowej aplikacji typu *REST* w *Spring* [2], współpracującej z aplikacją klienta, zaimplementowaną w języku *JavaScript* z wykorzystaniem interfejsu *Fetch API* [8].

Zakres tematyczny

- Utworzenie *REST API* z funkcjonalnością *CRUD* w *Spring Boot* za pomocą narzędzia *Initializr*.
- Konfiguracja *REST API* do pracy z danymi w formacie *JSON*.
- Przetestowanie *REST API* za pomocą narzędzia *Postman*.
- Utworzenie aplikacji klienckiej, współpracującej z *REST API* za pomocąinterfejsu *Fetch API* w *JavaScript*.

Wprowadzenie

Tworzone do tej pory przykładowe aplikacje były aplikacjami typu *MPA* (ang. *Multi Page Application*), dla których widoki (*JSP*, *Thymeleaf*) generowane były **po stronie serwera** i wysyłane do przeglądarki internetowej w czasie rzeczywistym. Obsługę żądań w tego typu aplikacjach realizuje się w trybie synchronicznym.

Innym rozwiązaniem są aplikacje typu *SPA* (ang. *Single Page Application*), korzystające tylko z *REST API*, udostępnianego przez serwer. Taka aplikacja składa się zwykle tylko z jednego, głównego dokumentu *HTML*, który poprzezu *JavaScript* komunikuje się z serwerem WWW w architekturze *REST*, za pośrednictwem protokołu *HTTP*, wysyłając żądania w trybie asynchronicznym (w tle). Do implementacji *SPA* po stronie klienta można skorzystać z czystego języka *JavaScript* i jego interfejsu *Fetch API* (do przesyłania żądań w trybie asynczronicznym) lub z dedykowanych szkieletów programistycznych (np. *Angular*, *React*, *Vue*). Do implementacji *REST API* po stronie serwera w języku *Java* wykorzystuje się najczęściej *Spring REST API*. *REST API* jest elastycznym i szeroko rozpoznawanym interfejsem dla systemów stron trzecich, również dla innych klientów (np. aplikacji mobilnych).

Do realizacji zadań z niniejszego laboratorium potrzebny będzie klient do przetestowania utworzonego *REST API*. W przykładach wykorzystano narzędzie *Postman*, które można pobrać ze strony: https://www.postman.com/downloads/.

Zadanie 7.1. Inicjalizacja projektu *REST API*

Do utworzenia projektu zastosuj *Spring Initializr* (https://start.spring.io/). Dodaj zależności:

- *Rest Repositories*,
- *Spring Data JPA*,
- *MySQL Driver*,
- *Lombok* (opcjonalna, wygodna biblioteka wspierająca m.in. automatyczne generowanie konstruktorów i metod ***get*** i ***set***).

Otwórz projekt w swoim *IDE*. Do pliku ***application.properties*** dodaj konfigurację do połączenia z bazą danych za pomocą sterownika dla *MySQL* (pamiętaj aby przed uruchomieniem projektu uruchomić też serwer *MySQL*). Sprawdź, czy podana w przykładzie 7.1 konfiguracja jest odpowiednia dla bazy danych (w przykładzie wykorzystana jest baza danych ***test***). Zbuduj projekt z dodanymi zależnościami.

Przykład 7.1. Zawartość pliku application.properties

```
server.port=8080
#Konfiguracja BD i MySQL:
#struktura BD ma być zaktualizowana przy starcie aplikacji:
spring.jpa.hibernate.ddl-auto = update
spring.jpa.show-sql = true
spring.datasource.url =
jdbc:mysql://localhost:3306/test?serverTimezone=UTC&useUnicode=yes&chara
cterEncoding=UTF-8
spring.datasource.username = root
spring.datasource.password =
#Ustawienie strategii nazewnictwa (Naming strategy) dla Hibernate
spring.jpa.hibernate.naming-strategy =
org.hibernate.cfg.ImprovedNamingStrategy
```

Zadanie 7.2. Podstawowe elementy *REST API*

7.2.1. Adnotacje *@JsonView* w klasie encji

W projekcie utwórz pakiet ***entities*** z klasą encji ***Student***, która zawiera 4 pola prywatne: ***id***, ***name***, ***surname*** i ***average.*** Pamiętaj o dodaniu znanych już adnotacji ***@Entity, @Id*** i ***@GeneratedValue*** (z odpowiednimi parametrami) oraz wszystkie pola poza ***id,*** poprzedź adnotacją ***@JsonView***. Adnotacja ***@JsonView*** pozwala mapować łańcuch w formacie *JSON* z danymi studenta (przesyłanymi z formularza w parametrach żądania) na obiekt klasy ***Student*** po stronie aplikacji. Metody ***get*** i ***set*** dodaj za pomocą adnotacji ***@Getter*** i ***@Setter*** poprzedzających klasę encji, korzystając z **biblioteki *Lombok***.

7.2.2. Repozytorium i klasa z adnotacją *@Service*

W pakiecie ***entities*** utwórz interfejs ***StudentRepository*** rozszerzający ***CrudRepository***. Następnie przygotuj klasę (usługę, serwis) wspomagającą pracę z danymi. W tym celu utwórz nowy pakiet ***services***, a w nim klasę ***StudentService*** poprzedzoną adnotacją ***@Service***. Adnotacja ***@Service*** jest dedykowana klasom, które dostarczają usługi. Do klasy ***StudentService*** „wstrzyknij" prywatne pole ***studentRepository*** typu ***StudentRepository***, poprzedzając go adnotacją ***@Autowired***. Do klasy dodaj także metodę publiczną ***getStudentList()*** do pobrania z repozytorium listy obiektów ***Student*** (Przykład 7.2).

Przykład 7.2. Klasa StudentService

```
package bp.pai_rest.services;

import bp.pai_rest.entities.Student;
import bp.pai_rest.entities.StudentRepository;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class StudentService {
  @Autowired
  private StudentRepository studentRepository;

  public List<Student> getStudentList() {
       return (List<Student>) studentRepository.findAll();
  }
}
```

7.2.3. *REST* kontroler

Metody obsługujące żądania *HTTP* w przypadku tworzonego *REST API*, najczęściej otrzymują w żądaniu (*Request Body*) i wysyłają w odpowiedzi do klienta (*Response Body*) dane w formacie *JSON*. Klasa kontrolera poprzedzona adnotacją ***@RestController***, domyślnie obsługuje format danych *JSON*.

Dla przypomnienia, tablica z danymi o studentach w formacie *JSON* jest reprezentowana przez łańcuch postaci, np.:

```
[
 { "id":1, "name":"Olek", "surname":"Olewski", "average":4.5},
 { "id":2, "name":"Ola", "surname":"Olewska", "average":4.45}
]
```

Ostatnim elementem aplikacji jest zatem utworzenie klasy kontrolera, z metodami do obsługi żądań. Utwórz pakiet ***controllers*** z klasą ***StudentController*** z adnotacją ***@RestController*** (Przykład 7.3).

Przykład 7.3. Klasa StudentController

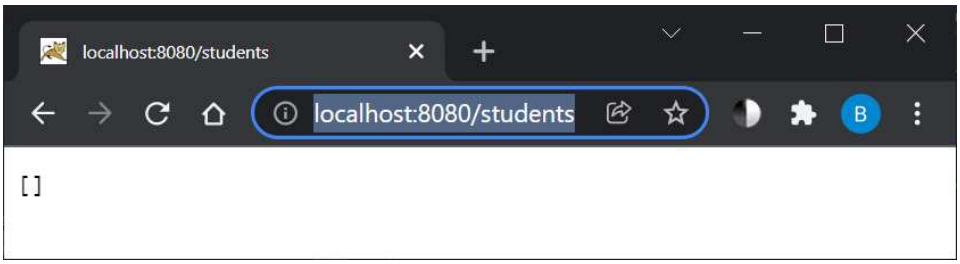
```
@RestController
public class StudentController {
    @Autowired
    private StudentService studentService;

    @GetMapping("/students")
    public List<Student> getAll() {
        return studentService.getStudentList();
    }
}
```

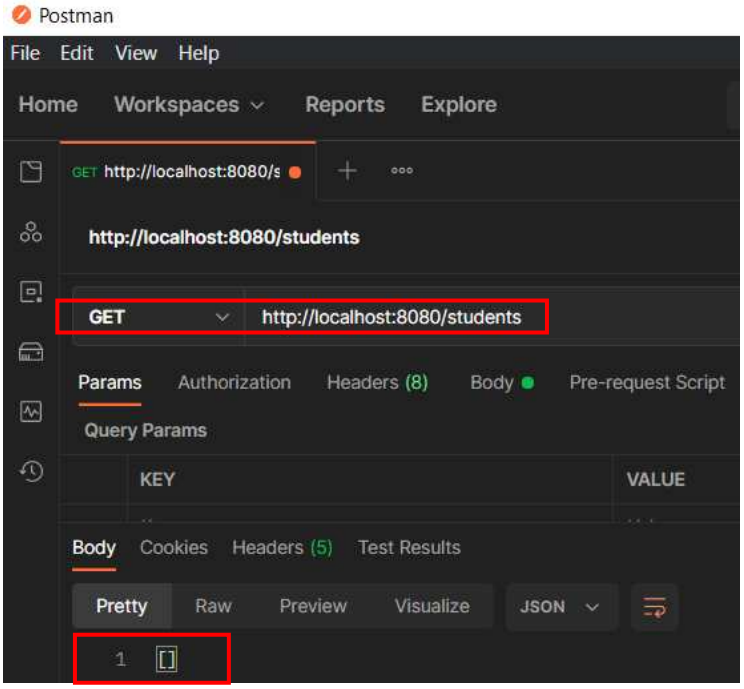
Na rysunku 7.1 pokazano strukturę plików projektu.

Rys. 7.1. Struktura projektu *pai_rest*

Uruchom aplikację a następnie w przeglądarce (Rys. 7.2) oraz w programie *Postman* (Rys. 7.3) wyślij żądanie postaci: ***http://localhost:8080/students***. W odpowiedzi przesłana zostanie pusta tablica, ponieważ nie utworzono jeszcze rekordów w bazie danych.



Rys. 7.2. Odpowiedź z *REST API* – wynik w przeglądarce



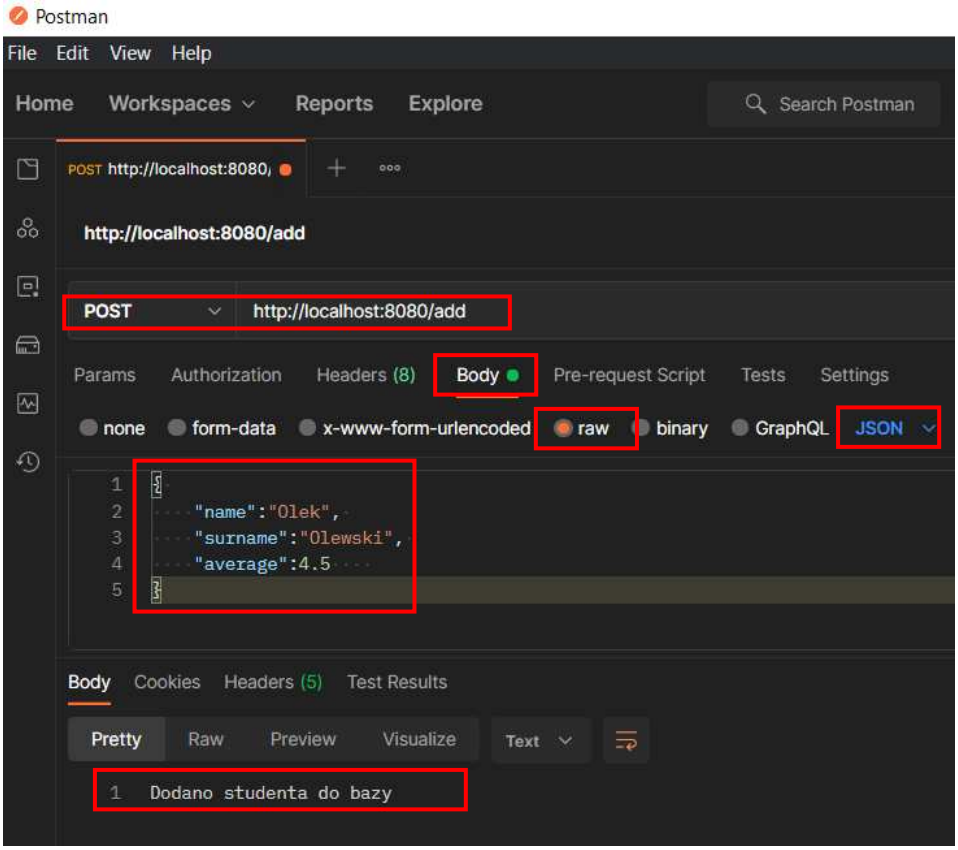
Rys. 7.3. Odpowiedź z *REST API* – fragment okna z wynikiem w narzędziu *Postman*

7.2.4. Utrwalenie danych z żądania

1. Korzystając z biblioteki *Lombok*, do klasy ***Student*** dodaj konstruktor bezparametrowy i konstruktor z wszystkimi parametrami za pomocą adnotacji ***@NoArgsConstructor*** i ***@AllArgsConstructor*** poprzedzających definicję klasy ***Student***.
2. W klasie ***StudentService*** utwórz nową metodę ***addStudent*** z parametrem klasy ***Student***, której zadaniem ma być dodanie obiektu do bazy danych.
3. W kontrolerze utwórz metodę, która obsłuży żądanie *POST,* związane z dodaniem nowego studenta do bazy. Metoda ta powinna posiadać adnotację ***@PostMapping***. Jako parametr metoda przyjmuje obiekt typu ***Student*** poprzedzony adnotacją ***@RequestBody***, dzięki czemu przesłane w żądaniu dane w formacie *JSON* zostaną zmapowane na obiekt ***Student*** (następuje automatyczne pobranie danych *JSON* z ciała żądania i przypisanie ich do odpowiednich pól obiektu ***Student***, o ile tylko nazwy kluczy z *JSON* pokrywają się z nazwami pól w klasie ***Student***).

Zbuduj i uruchom ponownie aplikację, a następnie korzystając z narzędzia *Postman* wyślij żądanie *HTTP* typu *POST* pod adres: ***http://localhost:8080/add***, ale tym razem w ciele żądania dodaj dane w formacie *JSON* postaci (Rys. 7.4):

```
{ "name":"Olek", "surname":"Olewski", "average":4.5}
```



Rys. 7.4. Żądanie i odpowiedź – metoda *POST* z dołączonymi danymi w formacie *JSON*

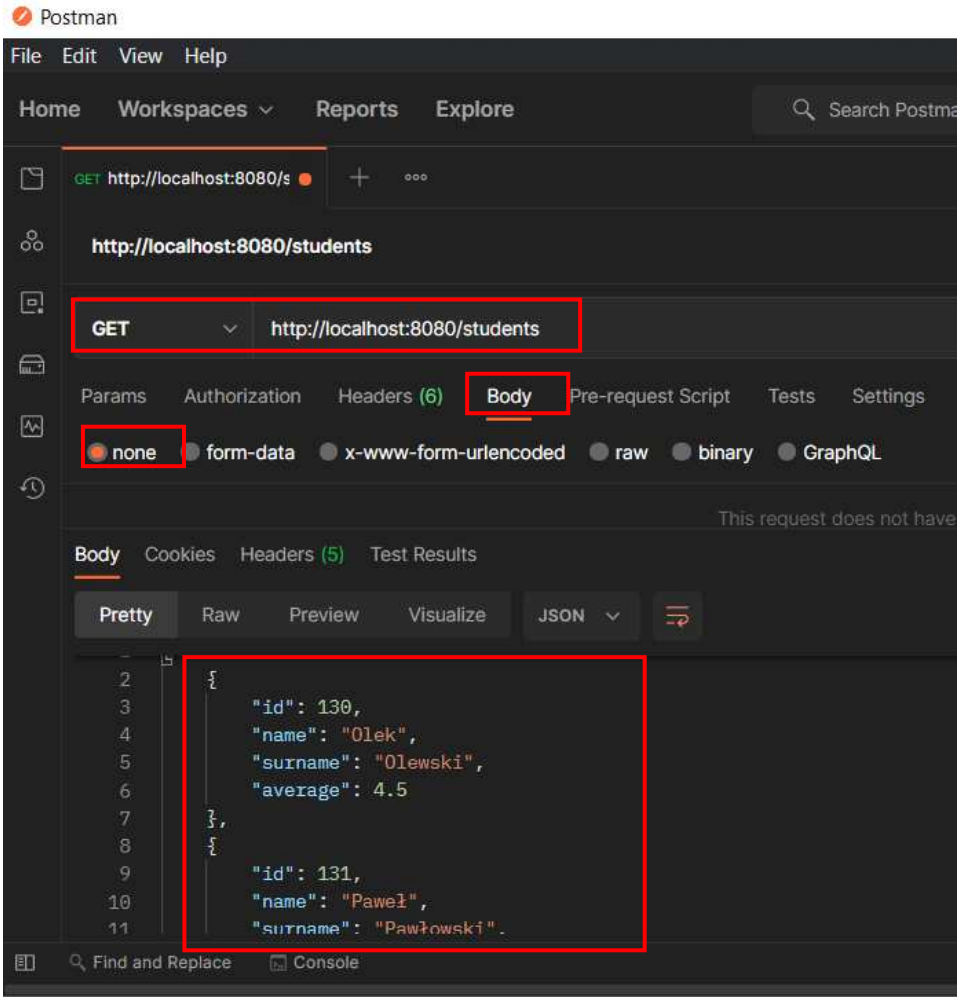
Wyślij kolejne zapytanie z danymi studenta (z polskimi literami):

```
{ "name":"Paweł", "surname":"Pawłowski", "average":4.99}
```

Czy dane studenta z polskimi literami w imieniu lub nazwisku zostały prawidłowo wprowadzone do bazy danych? Jeśli nie i pojawił się problem *com.mysql.cj.jdbc.exceptions.MysqlDataTruncation: Data truncation: Incorrect string value: '\xC5\x82' ...*, to zmień ręcznie w *MySQL* kodowanie w kolumnach ***name*** i ***surname*** na *UTF8-general-ci*.

Powtórz żądanie dodania studenta dla *Pawła Pawłowskiego*. Tym razem rekord powinien być poprawnie dodany do bazy.

Następnie ponownie wyślij zapytanie typu *GET* w celu pobrania wszystkich studentów z bazy (Rys. 7.5). Sprawdź rekordy w bazie danych w tabeli ***student***.



Rys. 7.5. Wynik po wykonaniu dodawania 2 studentów

Zadanie 7.3. Obsługa metod *DELETE* i *PUT*

Uzupełnij funkcjonalność aplikacji o dodatkowe możliwości usuwania i edycji studenta po wskazanym ***id*** (przy czym operacje te można realizować tylko, gdy istnieje student o takim ***id)***. Po dodaniu kolejnej funkcjonalności – testuj jej

działanie za pomocą narzędzia *Postman*, wysyłając odpowiednio żądanie za pomocą metod *DELETE* i *PUT*.

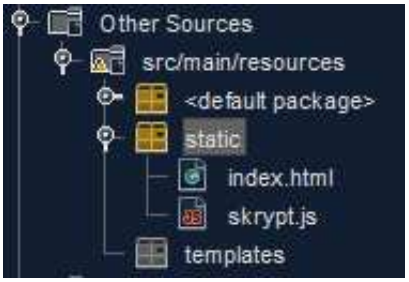
1. Utwórz metodę ***deleteStudent*** w klasie ***StudentService*** i w kontrolerze metodę do obsługi akcji usunięcia studenta z adnotacją ***@DeleteMapping***.
2. Utwórz metodę do modyfikacji (w klasie ***StudentService)*** wszystkich danych studenta poza ***id***. W kontrolerze dodaj metodę z adnotacją ***@PutMapping***, która pozwoli na aktualizację danych studenta.

UWAGA! Usuwanie lub edycję danych można zrealizować po sprawdzeniu, czy obiekt o zadanym ***id*** istnieje w bazie danych. Do sprawdzenia można wykorzystać metodę repozytorium: ***studentRepository.existsById(id)***, która zwraca wartość logiczną.

Zadanie 7.4. *JavaScript* z *Fetch API*

Korzystając z interfejsu *Fetch API* w *JavaScript*, do projektu dodaj plik ***index.html*** oraz odpowiednie funkcje *JavaScript,* które pozwolą na przesyłanie żądań do *REST API* z poziomu strony ***index.html.***

Plik ***index.html*** oraz skrypt z funkcjami *JavaScript* (np. ***skrypty.js***) umieść w folderze projektu: ***resources/static*** (Rys. 7.6).



Rys. 7.6. Zasoby aplikacji w *HTML* i *JavaScript*