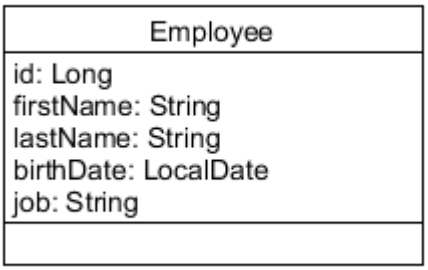
Systemy rozproszone i integracja usług - ćwiczenia nr 2: implementacja usługi REST

Celem ćwiczenia jest nabycie umiejętności implementacji usługi typu REST oferującej podstawowe operacje na przykładowym modelu danych.

Część 1: zadanie wprowadzające

W ramach tego ćwiczenia zaimplementujemy usługę REST w technologii Java Spring, umożliwiającą podstawowe operacje typu CRUD (Create, Retrieve, Update, Delete) na następującym modelu danych:

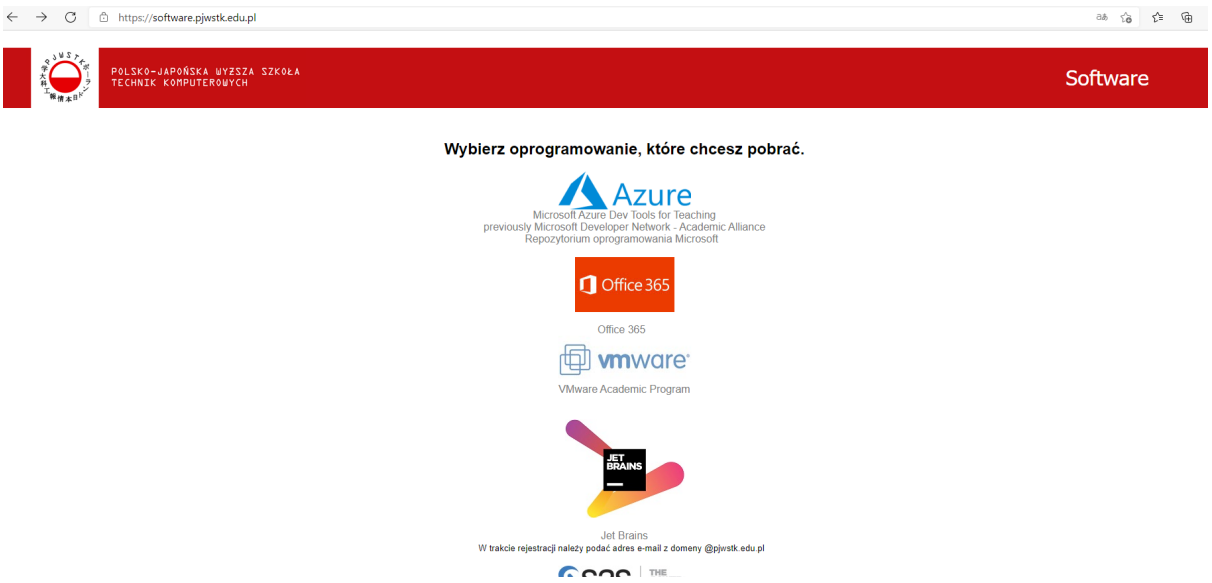


Dane będą zapisywane w prostej, relacyjnej bazie danych H2 uruchamianej wraz z aplikacją.

1. Instalacja oprogramowania

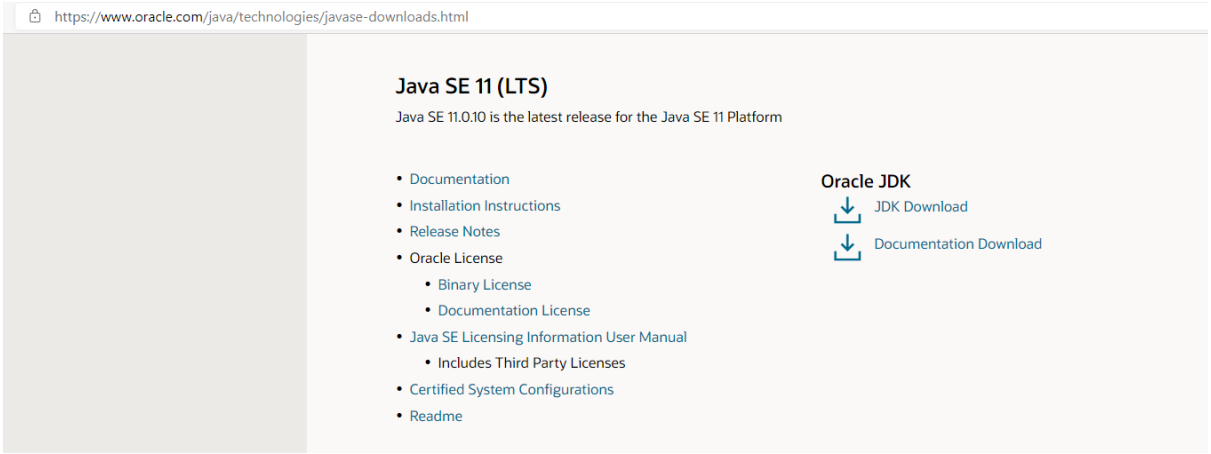
1.1. Środowisko developerskie

Zalecanym środowiskiem do tworzenia aplikacji jest oprogramowanie JetBrains IntelliJ IDEA Ultimate. Studenci PJATK mają możliwość uzyskania bezpłatnej licencji do celów edukacyjnych rejestrując się z adresu szkolnego. Więcej informacji na stronie: https://software.pjwstk.edu.pl/ i https://www.jetbrains.com/community/education/#students



1.2. Oprogramowanie Java JDK.

W celu kompilacji i uruchomienia stworzonego kodu należy zainstalować oprogramowanie Java Development Kit. Zalecaną wersją jest Oracle JDK w wersji 11 LTS (Long Time Support) https://www.oracle.com/java/technologies/javase-downloads.html



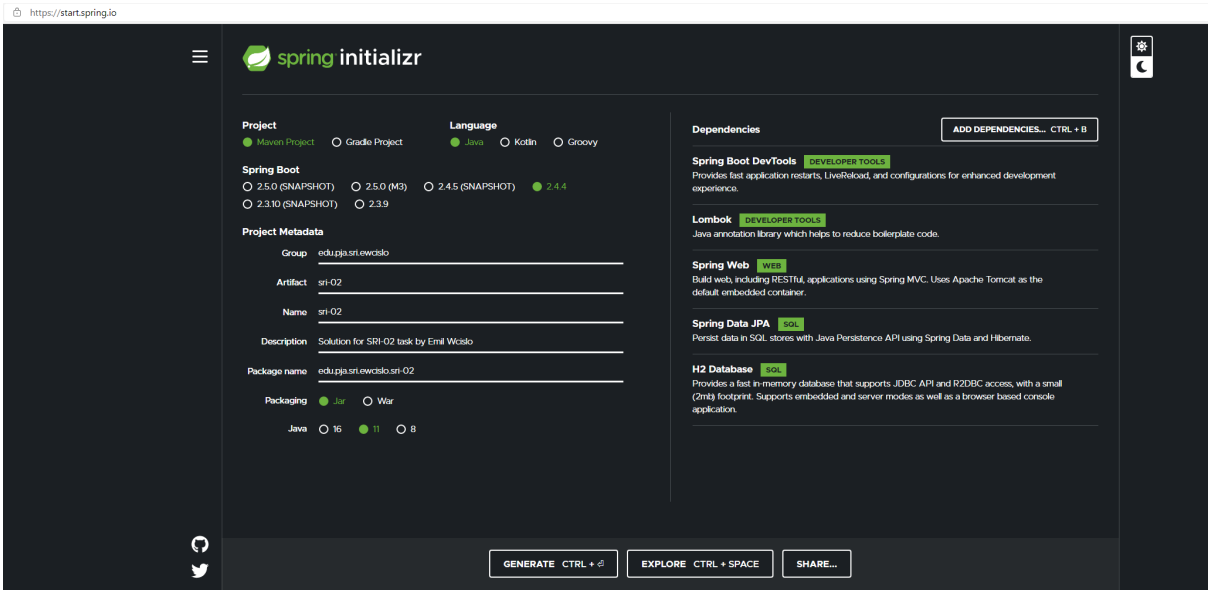
1.3. Oprogramowanie Postman

Podobnie jak w poprzednim ćwiczeniu będziemy wykorzystywać aplikację Postman do testowania gotowych endpointów typu REST. Aplikacja jest do pobrania na stronie: https://www.postman.com/downloads/

2. Tworzenie szkieletu aplikacji

Aplikacja będzie oparta o framework Spring Boot. Wygodnym sposobem tworzenia nowych aplikacji w tej technologii jest wykorzystanie generatora szkieletu projektu Spring Initializr (https://start.spring.io/). Tworząc nowy projekt można w łatwy sposób dodać najważniejsze biblioteki aplikacji.

Nowy projekt należy skonfigurować wg poniższego zrzutu ekranu:



Wybieramy stabilną wersję frameworku Spring Boot, w momencie tworzenia tej lekcji była to wersja 2.4.4
W polu ‘Group’ należy nazwać główny pakiet aplikacji, powinien on zawierać Twój numer indeksu.
Należy wybrać wersję Javy taką, jaką uprzednio pobrano i zainstalowano (w tym przypadku jest to Java 11)
Za pomocą przycisku ‘Add dependencies’ po prawej stronie konfiguratora należy dodać następujące biblioteki:

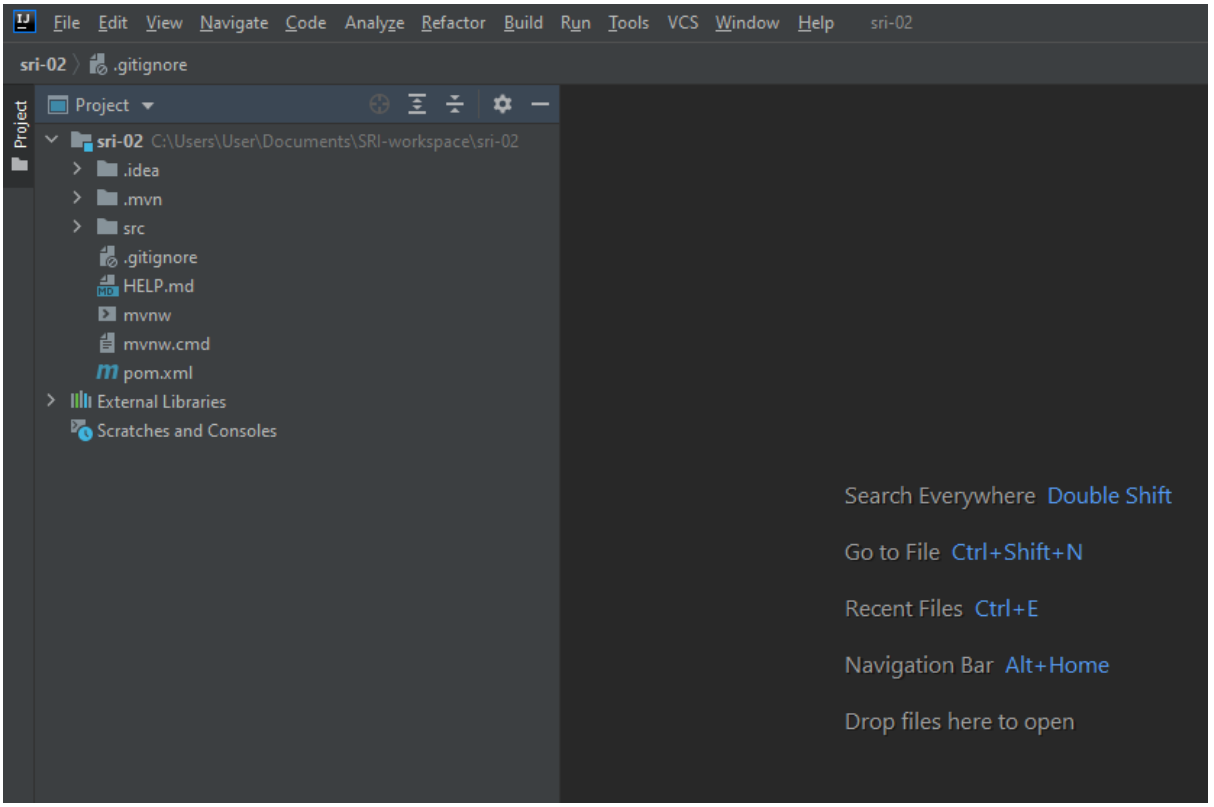
- Spring Dev Tools (opcjonalne, rekomendowane) - umożliwia szybki restart aplikacji podczas wprowadzania zmian w kodzie
- Lombok (opcjonalne, rekomendowane) - upraszcza kod aplikacji generując typowe wymagane fragmenty kodu podczas procesu budowania aplikacji, np. metody typu get i set, konstruktory.
- Spring Web (wymagane) - umożliwia min. tworzenie kontrolerów typu REST
- Spring Data JPA (opcjonalne, rekomendowane) - umożliwia mapowanie klas encji na relacyjną bazę danych. Zapewnia implementację repozytoriów udostępniających operacje na bazie danych
- H2 Database (opcjonalne, rekomendowane) - implementacja prostej relacyjnej bazy danych uruchamianej w ramach procesu aplikacji. Umożliwia uruchomienie konsoli administracyjnej bazy danych ‘h2-console’

Należy podkreślić, że w tym ćwiczeniu nie będziemy korzystać z biblioteki ‘Rest Repositories’, która umożliwia automatyczne generowanie funkcjonalności CRUD dla wybranego modelu danych i udostępnianie ich przez interfejs REST. Posiada ona istotne ograniczenia, które będą omówione na przykładzie kolejnych ćwiczeń.

Po określeniu konfiguracji projektu należy wybrać opcję ‘Generate’, po czym zostanie pobrany plik zip zawierający szkielet projektu. Należy zapisać go w folderze z projektami do przedmiotu (niekoniecznie bezpośrednio na pulpicie) i rozpakować.

3. Uruchomienie i konfiguracja środowiska developerskiego
   W kolejnym kroku należy uruchomić oprogramowanie IntelliJ IDEA i otworzyć folder z rozpakowanym projektem (z menu File -> Open). IDE powinno rozpoznać typ

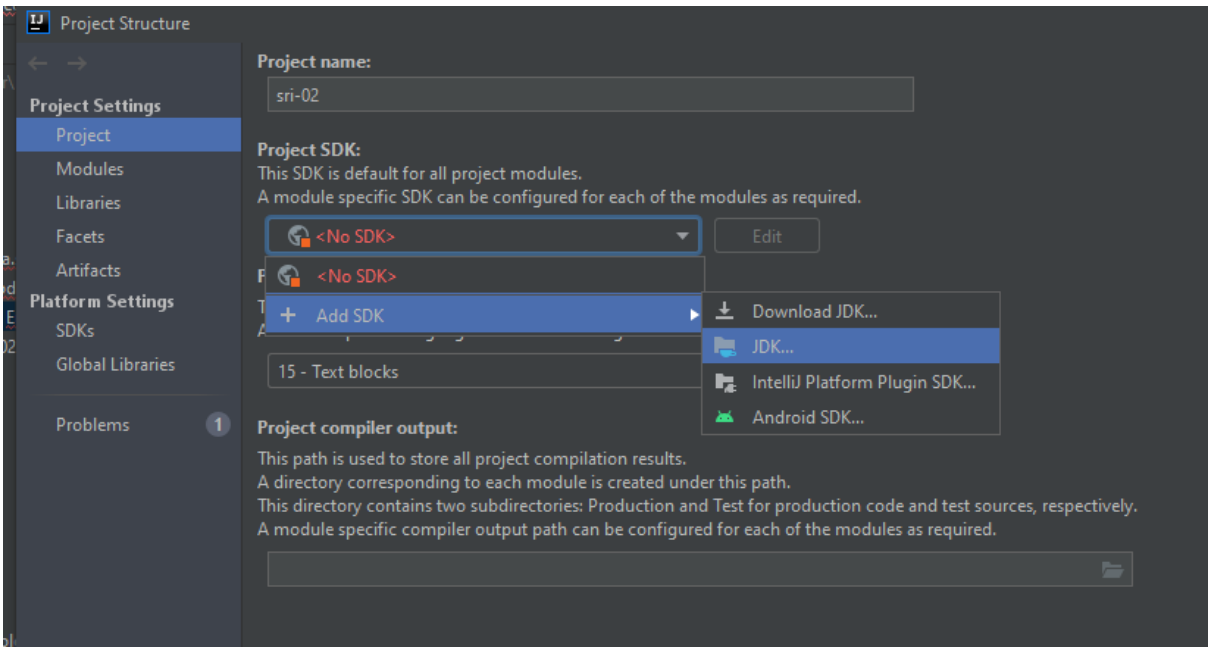
projektu i pobrać wymagane biblioteki opisane w pom.xml dostarczonym z szkieletem aplikacji.

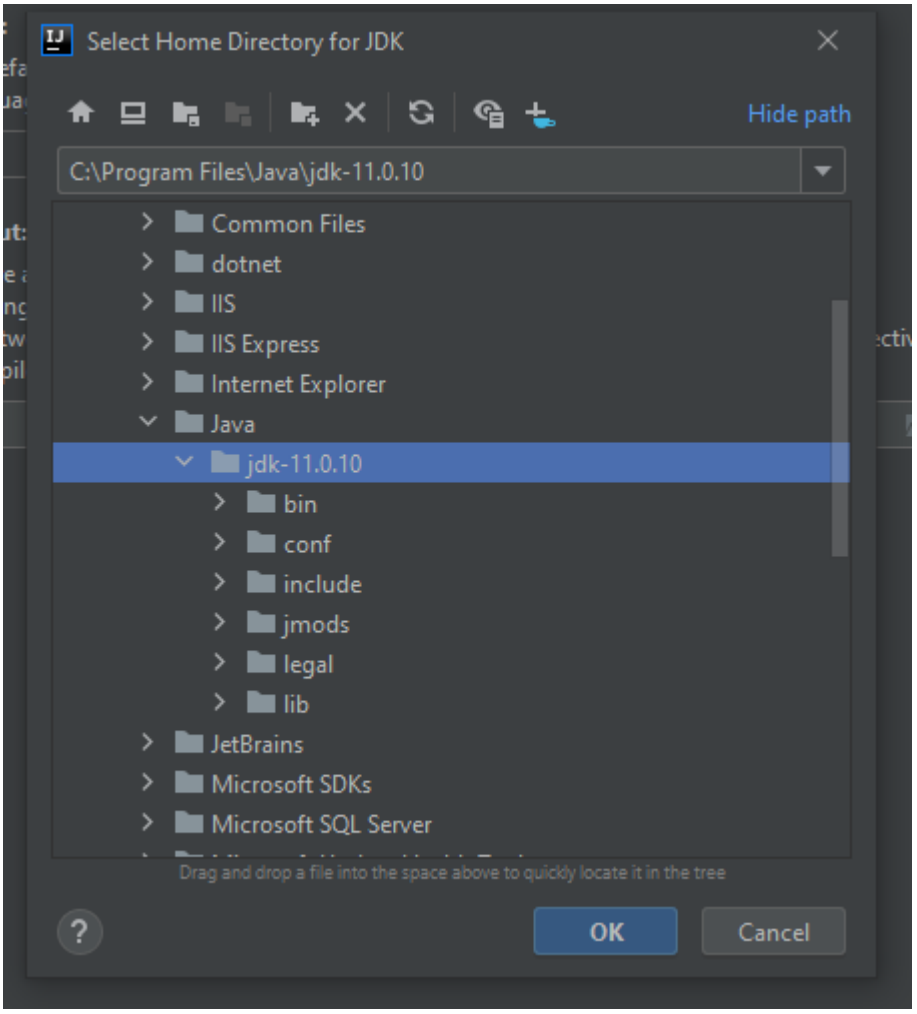


Należy upewnić się, że następujące ustawienia środowiska zostały włączone:

3.1. Kompilator Java

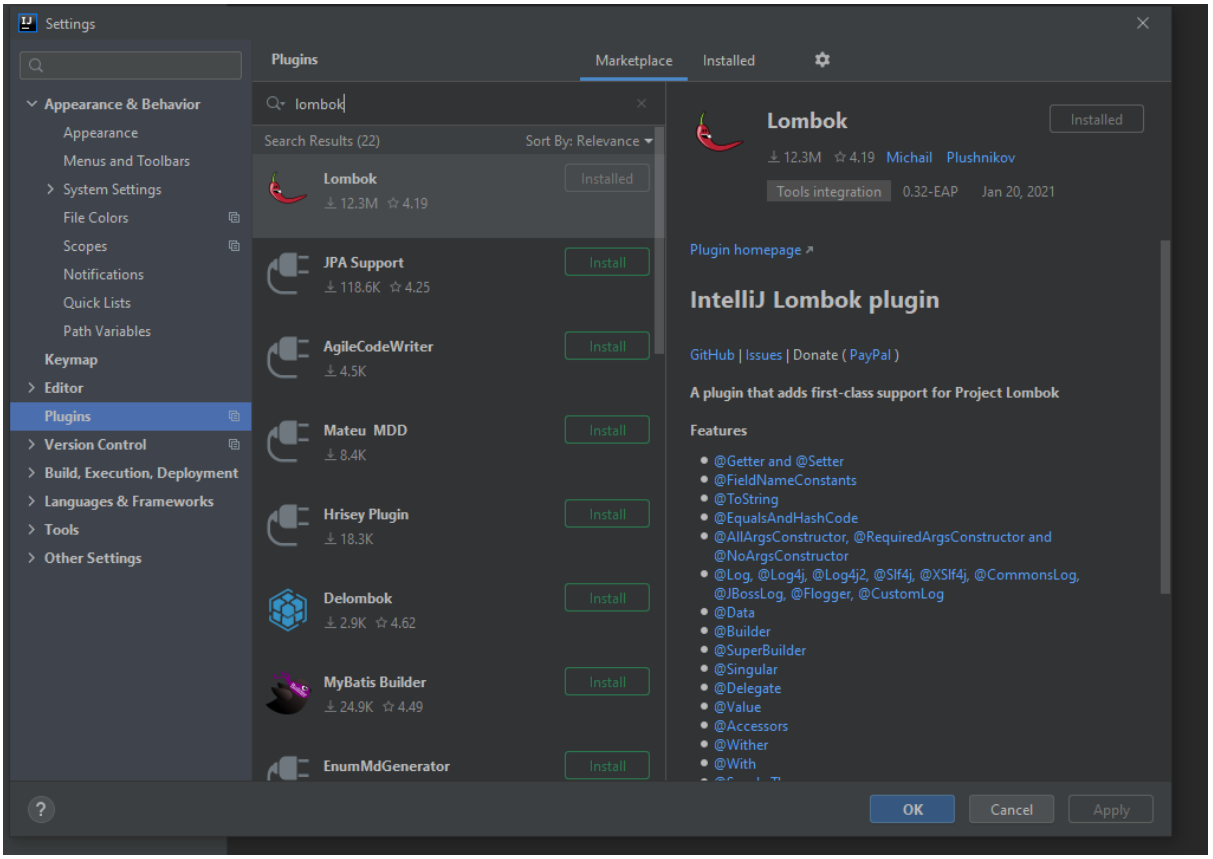
Po wejściu w ustawienia (File -> Project structure) w zakładce Project Settings -> Project należy poprawnie ustawić własność Project SDK na zainstalowane środowisko Java tak, jak na poniższym zrzucie ekranu:





3.2. Plugin Lombok

Po wejściu w File -> Settings i zakładkę Plugins należy upewnić się, że zainstalowano plugin Lombok (IDEA może zainstalować go automatycznie).



Jeśli plugin nie został zainstalowany, należy to uczynić.

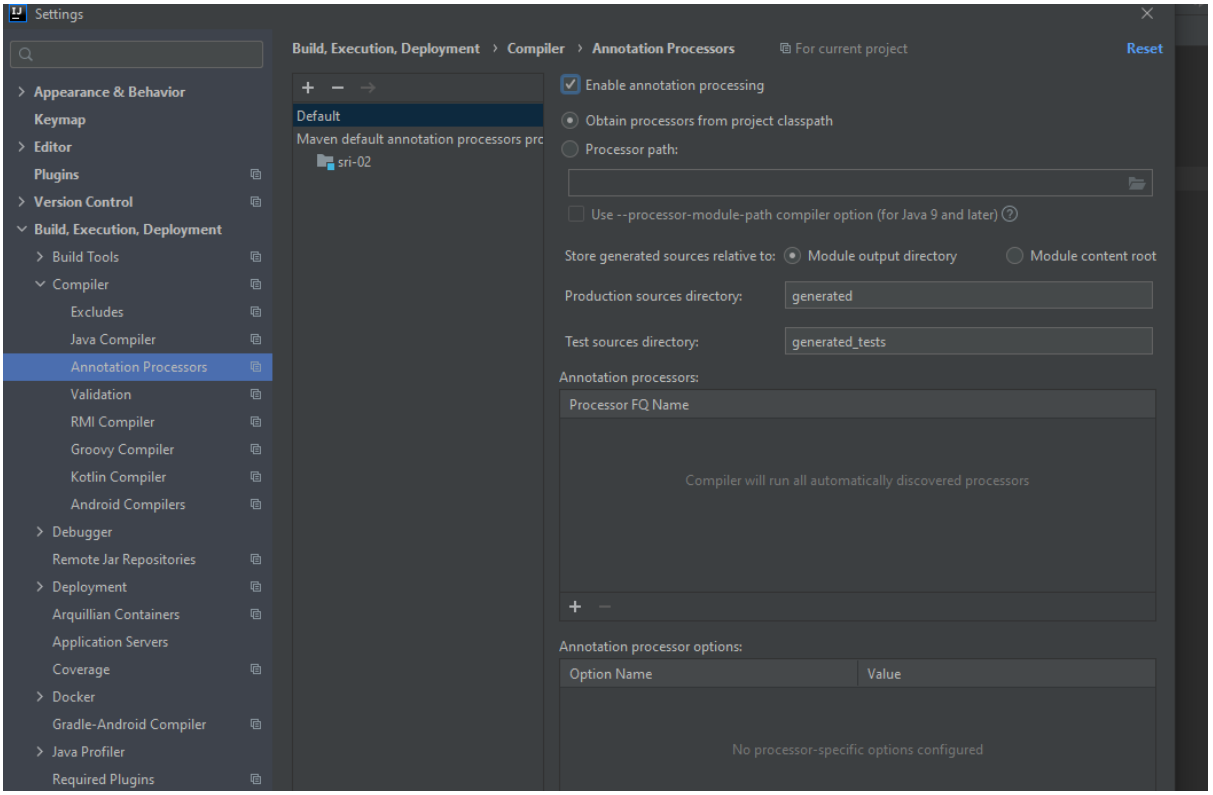
3.3. Procesor adnotacji

W celu umożliwienia poprawnego działaniu powyższego pluginu należy uruchomić procesor adnotacji automatycznie uruchamiany przed kompilacją kodu. Ustawienie to można znaleźć w: File -> Settings -> Build, Execution, Deployment -> Compiler -> Annotation Processors

3.4. Konfiguracja bazy danych i konsoli administracyjnej bazy danych

Należy zmodyfikować plik `application.properties` znajdujący się w projekcie w katalogu: `src/main/resources` dodając następujące ustawienia:

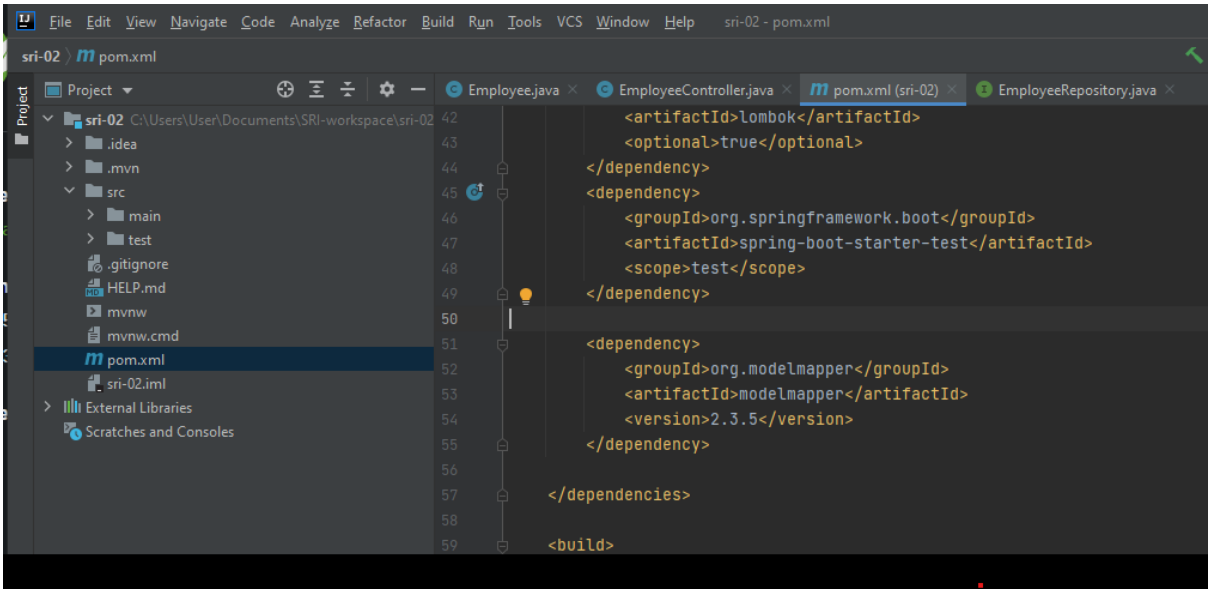
```
spring.datasource.url=jdbc:h2:mem:sri-hr
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.h2.console.enabled=true
```



3.5. Instalacja biblioteki `modelmapper`

W celu łatwego przenoszenia danych pomiędzy klasami o tożsamej, lub podobnej strukturze danych, użyjemy biblioteki `modelmapper`. Aby ją zainstalować, należy zmodyfikować plik pom.xml znajdujący się w głównym katalogu projektu. Przed zamykającym elementem `</dependencies>` należy umieścić następujący fragment:

```
<dependency>
      <groupId>org.modelmapper</groupId>
      <artifactId>modelmapper</artifactId>
      <version>2.3.5</version>
</dependency>
```



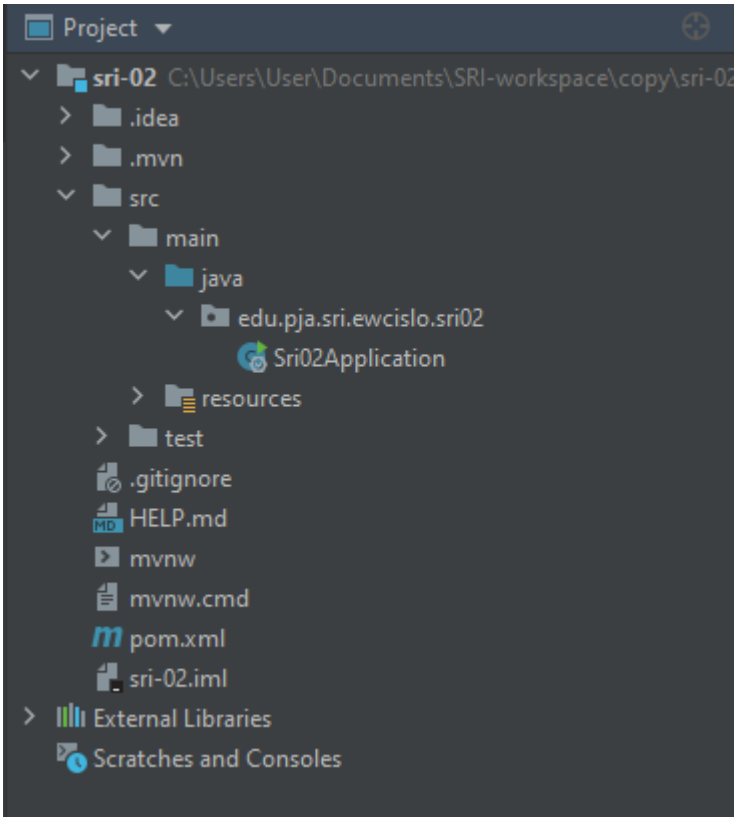
Po zapisaniu zaktualizowanego pliku pom.xml należy kliknąć prawym klawiszem myszy na głównej gałęzi projektu w oknie po lewej stronie i z menu kontekstowego wybrać Maven -> Reload project. Nowa biblioteka powinna zostać pobrana i zainstalowana automatycznie.

Następnie należy dopisać poniższą metodę do klasy konfigurującą aplikację, w tym przypadku będzie to klasa Sri02Application:

```
@Bean
public ModelMapper modelMapper() {
	return new ModelMapper();
}
```

Metoda ta umożliwia korzystanie z mappera jako obiektu zarządzanego przez Spring, co ułatwi podłączenie go do innych części aplikacji.

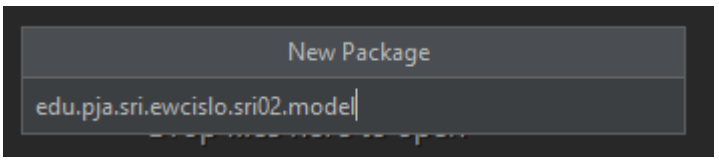
4. Implementacja modelu danych.
   Główny kod aplikacji znajduje się w folderze src/main/java, należy rozwinąć go w drzewie katalogów po lewej stronie ekranu.



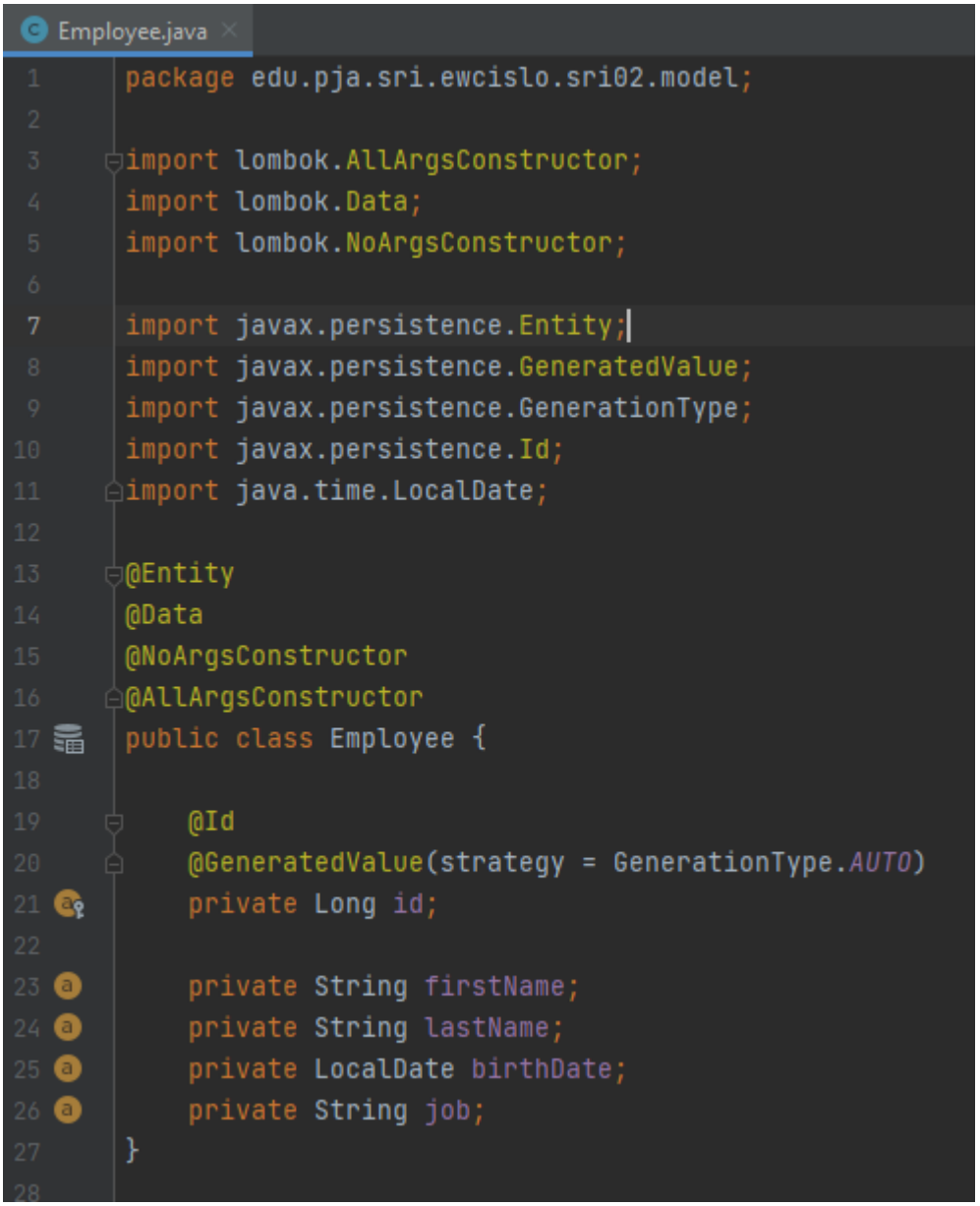
Należy zwrócić uwagę na nazwę głównego pakietu, w którym znajduje się inicjująca klasa aplikacji (w tym przypadku edu.pja.sri.ewcislo.sri02). Wszystkie stworzone klasy naszej aplikacji muszą znajdować się w tym pakiecie, lub pakietach podrzędnych, w przeciwnym wypadku mogą nie zostać rozpoznane (np. klasy encji) i aplikacja może nie działać poprawnie.

4.1. Implementacja klasy encji Employee

Pierwszą stworzonym fragmentem kodu będzie klasa Employee reprezentująca encję mapowaną do bazy danych. Klasy o różnym zastosowaniu grupujemy w oddzielnych pakietach, należy utworzyć pakiet model jako podrzędny wobec głównego pakietu aplikacji:



Następnie tworzymy klasę Employee, będzie ona wyglądać w następujący sposób:



Poniżej objaśnienie poszczególnych fragmentów:

- Adnotacja @Entity z biblioteki JPA (Java Persistence API) oznacza, że klasa reprezentuje encję mapowaną na bazę danych
- Adnotacja @Data z biblioteki Lombok pozwoli na wygenerowanie metod get i set dla wszystkich prywatnych atrybutów klasy, co nieco zmniejszy ilość wymaganego kodu i zwiększy jego przejrzystość
- Adnotacje @NoArgsConstructor i @AllArgsConstructor prowadzą do wygenerowania odpowiednio konstruktora bezargumentowego (który jest wymagany dla encji JPA) oraz konstruktor z wszystkimi atrybutami (który nieco ułatwi tworzenie nowych obiektów)

W ciele klasy zdefiniowane zostały jej atrybuty, docelowo mapowane na kolumny w tabeli bazy danych. Atrybut id zawiera dodatkowo adnotacje @Id wskazującą na klucz główny tabeli, oraz @GeneratedValue(strategy = GenerationType.AUTO) wskazujący na sposób przydzielania wartości tego klucza, w tym przypadku będą one generowane przez domyślny mechanizm bazy danych.

4.2. Implementacja klasy reprezentującej dane transferowe EmployeeDto

Zwykle nie jest dobrym pomysłem bezpośrednie zwracanie instancji encji poprzez API typu REST z kilku powodów:

- Często chcemy dostosować zbiór atrybutów zwracanych przez API, w tym pominąć niektóre dane wrażliwe (np. hasła)
- Jeżeli klasa zawiera powiązanie z inną klasą encji, która zawiera referencję zwrotną to wystąpią problemy przy mapowaniu danych do formatu JSON (kod mapujący wpadnie w nieskończoną rekursję)
- Zmiany w modelu danych encji nie zawsze powinny wpływać na dane udostępnione przez API
- Zwiększamy przejrzystość kodu unikając mnożenia zbędnych adnotacji w jednej klasie
- Możemy zastosować różne klasy DTO dla różnych formatów danych które obsługuje nasze API
- Ogólnie zwiększa to elastyczność, kosztem utrzymania dodatkowej klasy

Więcej o powodach użycia DTO można przeczytać w poniższym artykule:
https://thorben-janssen.com/dont-expose-entities-in-api/

Należy utworzyć klasę EmployeeDto w nowym podpakiecie dto o następującej treści:

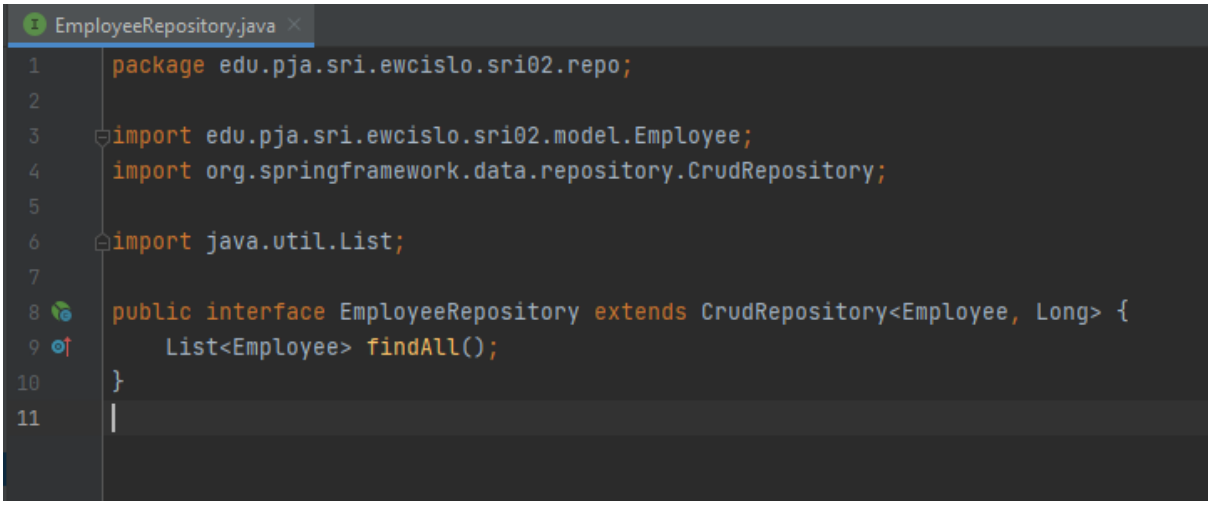
EmployeeDto.java

```java
package edu.pja.sri.ewcislo.sri02.dto;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.time.LocalDate;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class EmployeeDto {
    private Long id;
    private String firstName;
    private String lastName;
    private LocalDate birthDate;
    private String job;
}

```

W tym przypadku kod klasy EmployeeDto jest kopią Employee, z wyjątkiem usuniętych adnotacji JPA odnoszących się do mapowania obiektowo - relacyjnego.

5. Implementacja repozytoriów danych
   W celu zapewnienia podstawowych operacji bazodanowych na encji Employee należy dostarczyć klasę repozytorium oferującą takie funkcje. W tym przypadku stworzymy repozytorium typu Spring Data JPA.

W nowym podpakiecie repo należy stworzyć interfejs EmployeeRepository o następującej treści:



Interfejs ten dziedziczy po interfejsie CrudRepository z biblioteki Spring Data JPA. W definicji szablonu (oznaczonym w nawiasach trójkątnych) wskazujemy typ encji, która ma być obsługiwana przez repozytorium, oraz typ klucza głównego (taki sam, jak zdefiniowany w klasie encji).
Repozytoria Spring Data JPA oferują domyślnie podstawowe operacje typu CRUD: https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html?is-external=true

W tym przypadku nadpisaliśmy definicję metody findAll() aby zwracała listę obiektów zamiast domyślnego typu Iterable, co umożliwi bardziej intuicyjne korzystanie z wyniku tej metody.
Możliwe jest dopisanie kolejnych metod mających na celu wyszukiwanie po wybranej kombinacji atrybutów, sortowania itp. W tym celu należy odpowiednio zdefiniować sygnaturę metody (w szczególności jej nazwę) bez potrzeby jej implementowania. Więcej informacji można uzyskać w dokumentacji biblioteki Spring Data JPA: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods

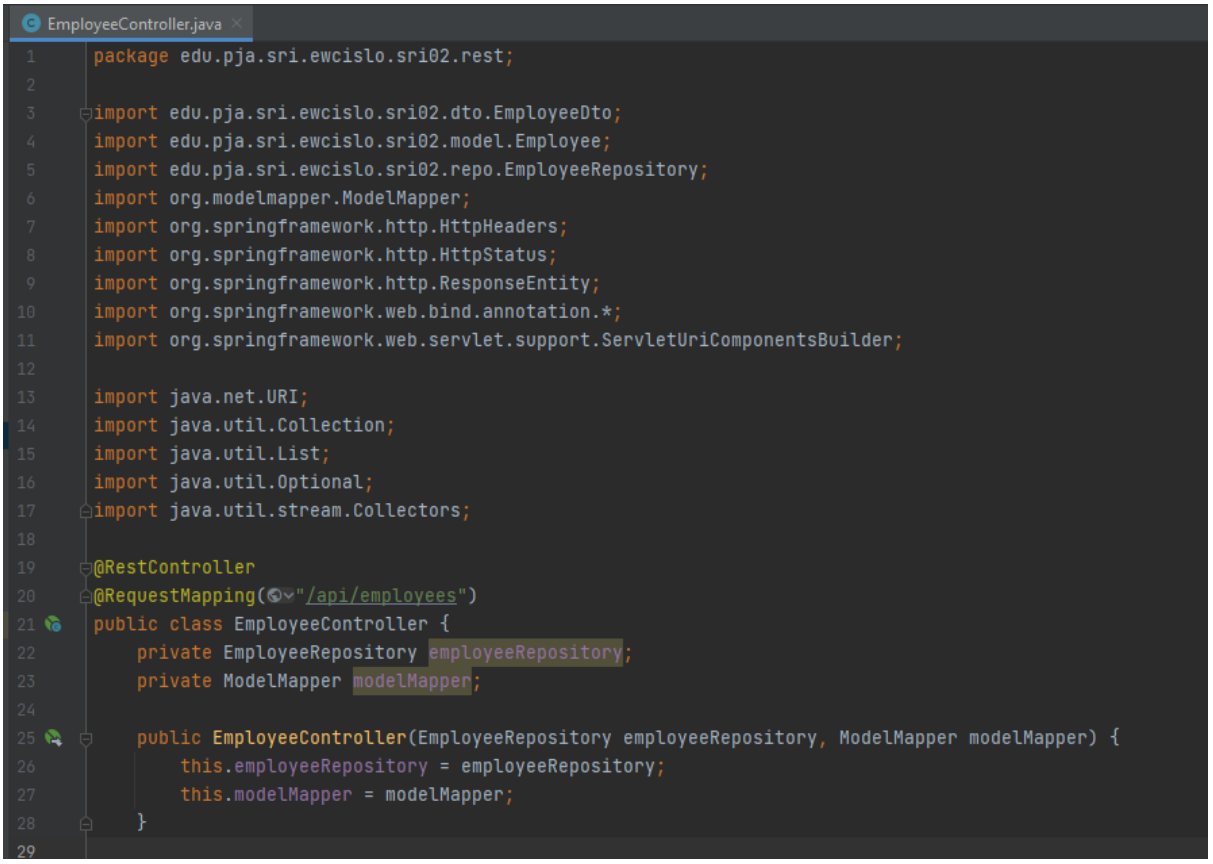
Implementacja tego interfejsu zostanie dostarczona przez Spring w trakcie uruchomienia, nie jest wymagane samodzielne implementowanie tego interfejsu.

6. Implementacja kontrolera REST

W następnym kroku stworzymy klasę EmployeeController umieszczoną w podpakiecie rest. Odpowiedzialnością tej klasy będzie udostępnianie funkcjonalności biznesowej aplikacji dla danej encji za pomocą REST API. Zaimplementujemy pięć metod służących do:

- Pobierania wszystkich obiektów Employee
- Pobierania obiektu Employee po jego id
- Zapisywania nowego obiektu Employee
- Aktualizacji wybranego obiektu Employee
- Usuwania wybranego obiektu Employee

Początkowy fragment kodu nowej klasy będzie wyglądać następująco:



Klasa zawiera adnotację Spring @RestController oznaczającą, że jej metody będą wywoływane przy odpowiednich żądaniach HTTP do naszej aplikacji. @RequestMapping wskazuje na adres URL pod jakim metody kontrolera będą dostępne (w poszczególnych metodach można określić dalsze fragmenty tego adresu).
Klasa ta jest komponentem Spring, co oznacza, że jej instancje będą tworzone przez kontener Springa, oraz będzie dostarczał on zależne atrybuty za pomocą mechanizmu IoC (Inversion on Control). Oznacza to, że inne niezbędne komponenty zarządzane przez Spring będą dostarczone automatycznie podczas tworzenia obiektu naszej klasy. Nie musimy zajmować się ich tworzeniem lub wyszukiwaniem. W tym przypadku będziemy potrzebować uprzednio zdefiniowanego repozytorium dla encji Employee (EmployeeRepository). W celu mapowania obiektów encji na DTO i vice-versa użyjemy obiektu klasy ModelMapper. Dostarczenie ('wstrzyknięcie') zależności będzie odbywać się przez odpowiednio napisany konstruktor, który przypisuje przekazane wartości komponentów do atrybutów naszej klasy. Konstruktor będzie automatycznie wywołany przez Spring, który zarządza zależnościami i będzie 'wiedział' jakie wartości ma przekazać. Więcej o IoC w Spring można przeczytać tutaj: https://docs.spring.io/spring-framework/docs/current/reference/html/core.html

6.1. Implementacja metod pomocniczych `convertToDto` i `convertToEntity`
Ponieważ będziemy operować zarówno na obiektach encji (operacje bazodanowe), jak i DTO (dane wejściowe i wyjściowe API), potrzebujemy metod konwertujących dane pomiędzy tymi typami. W tym celu zaimplementujemy następujące metody:

```
private EmployeeDto convertToDto(Employee e) {
    return modelMapper.map(e, EmployeeDto.class);
}
```

```
    private Employee convertToEntity(EmployeeDto dto) {
        return modelMapper.map(dto, Employee.class);
    }
```

Metody te korzystają z biblioteki ModelMapper, która automatycznie kopiuje dane operując na tożsamych strukturalnie klasach Employee i EmployeeDto

6.2. Implementacja metody getEmployees()
Metoda ta będzie pobierać listę obiektów encji z bazy danych, konwertować je na obiekty DTO, a następnie przekaże je jako rezultat, który zostanie skonwertowany na format JSON i zwrócony do klienta w odpowiedzi HTTP. Metoda ta będzie wyglądać następująco:

```
@GetMapping
public ResponseEntity<Collection<EmployeeDto>> getEmployees() {
    List<Employee> allEmployees = employeeRepository.findAll();
    List<EmployeeDto> result = allEmployees.stream()
            .map(this::convertToDto)
            .collect(Collectors.toList());
    return new ResponseEntity<>(result, HttpStatus.OK);
}
```

Adnotacja @GetMapping wskazuje na metodę HTTP GET, która będzie wywoływać naszą metodę. Ponieważ nie wskazaliśmy opcjonalnego adresu URL w parametrze tej adnotacji, będzie on taki sam jak główny adres URL klasy wskazany w adnotacji @RequestMapping. Kod metody pobiera listę encji z bazy danych przy pomocy obiektu repozytorium, następnie z wykorzystaniem strumieni i metody convertToDto konwertuje pobraną listę encji na listę obiektów typu EmployeeDto. Następnie metoda zwraca obiekt typu ResponseEntity, który oprócz zwracanych danych umożliwia określenie min. kodu odpowiedzi (w tym przypadku 200 OK). Opcjonalnie można również określić dodatkowe nagłówki odpowiedzi HTTP.

6.3. Implementacja metody getEmployeeById()
Metoda ta będzie zwracać obiekt encji na podstawie jego identyfikatora przekazanego w adresie URL, będzie ona wyglądać następująco:

```
    @GetMapping("/{empId}")
    public ResponseEntity<EmployeeDto>
getEmployeeById(@PathVariable Long empId) {
        Optional<Employee> emp =
employeeRepository.findById(empId);
        if(emp.isPresent()) {
            EmployeeDto employeeDto = convertToDto(emp.get());
            return new ResponseEntity<>(employeeDto,
HttpStatus.OK);
        } else {
```

```
            return new ResponseEntity<>(null,
HttpStatus.NOT_FOUND);
        }
    }
```

W tym przypadku adnotacja @GetMapping zawiera argument w postaci adresu URL, który stanowi dalszą część adresu określonego dla całej klasy w adnotacji @RequestMapping w nagłówku klasy. Użycie nawiasów klamrowych oznacza, że ta część adresu będzie wykorzystana jako parametr metody, opisanej w jej sygnaturze adnotacją @PathVariable. Kod metody obejmuje pobranie obiektu z repozytorium po wskazanym id, następnie w zależności czy dany obiekt istnieje w bazie danych zostanie zwrócona jego reprezentacja (po uprzedniej konwersji do DTO), lub w przeciwnym wypadku kod 404 NOT_FOUND.

6.4. Implementacja metody saveNewEmployee()
Metoda ta będzie przyjmować dane przesłane w ciele żądania typu POST, a następnie zapisywać je do bazy danych. Dodatkowo, będzie zwracać nagłówek HTTP “Location” wskazujący na link do nowego obiektu. Wygląda ona następująco:

```
    @PostMapping
    public ResponseEntity saveNewEmployee(@RequestBody EmployeeDto
emp) {
        Employee entity = convertToEntity(emp);
        employeeRepository.save(entity);

        HttpHeaders headers = new HttpHeaders();
        URI location = ServletUriComponentsBuilder
            .fromCurrentRequest()
            .path("/{id}")
            .buildAndExpand(entity.getId())
            .toUri();
        headers.add("Location", location.toString());

        return new ResponseEntity(headers, HttpStatus.CREATED);
    }
```

Dane przekazane w ciele żądania są mapowane na obiekt z użyciem adnotacji @RequestBody. Adres URL nowego obiektu jest tworzony na podstawie adresu bieżącego żądania z dopisanym id (nadanym przez bazę danych w momencie zapisu) nowego obiektu.

6.5. Implementacja metody updateEmployee()
Metoda ta będzie aktualizować istniejący obiekt w bazie (wskazany na podstawie jego id przekazanego w adresie URL). Metoda będzie dostępna za pomocą metody HTTP PUT. Czasami metody PUT używa się też do tworzenia nowych obiektów, jeżeli klient poda komplet danych obiektu, łącznie z jego identyfikatorem. W naszym przypadku id jest zarządzany przez bazę danych i przez to jest atrybutem tylko do odczytu, zatem metoda ta będzie służyć wyłącznie do aktualizacji istniejących

danych. W przypadku nieznalezienia rekordu w bazie zwracany jest kod 404 NOT_FOUND

```
    @PutMapping("/{empId}")
    public ResponseEntity updateEmployee(@PathVariable Long
empId, @RequestBody EmployeeDto employeeDto) {
        Optional<Employee> currentEmp =
employeeRepository.findById(empId);
        if(currentEmp.isPresent()) {
            employeeDto.setId(empId);
            Employee entity = convertToEntity(employeeDto);
            employeeRepository.save(entity);
            return new ResponseEntity(HttpStatus.NO_CONTENT);
        } else {
            return new ResponseEntity(HttpStatus.NOT_FOUND);
        }
    }
```

6.6. Implementacja metody deleteEmployee()
Metoda służy do usuwania obiektu po id:

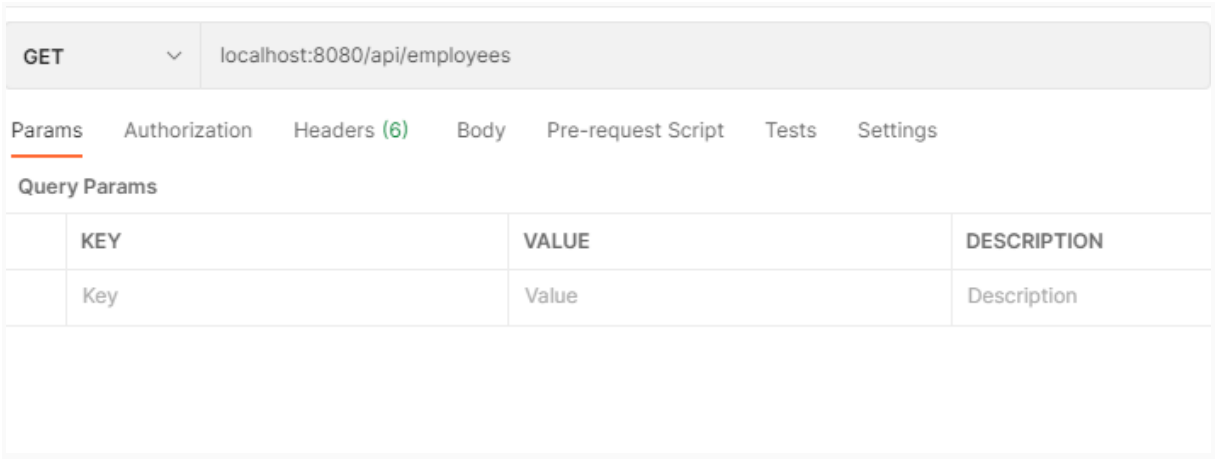
```
    @DeleteMapping("/{empId}")
    public ResponseEntity deleteEmployee(@PathVariable Long empId)
{
        employeeRepository.deleteById(empId);
        return new ResponseEntity(HttpStatus.NO_CONTENT);
    }
```

7. Uruchomienie i testowanie aplikacji
Aby uruchomić aplikację należy kliknąć prawym klawiszem myszy na klasie startowej (w tym przypadku na klasie edu.pja.sri.ewcislo.sri02.Sri02Application) i z menu kontekstowego wybrać opcję ‘Run’. Domyślnie aplikacja uruchomi się na porcie 8080, należy upewnić się że dany port nie jest zajęty, ewentualnie zmienić go w pliku konfiguracyjnym application.properties dopisując linię:

```
server.port=[nowy port serwera]
```

Należy oczywiście ustalić nową wartość w miejscu `[nowy port serwera]`

Testowanie należy przeprowadzić za pomocą aplikacji Postman tworząc odpowiednie żądania do każdej ze stworzonych metod, np:



Część 2: zadanie na ocenę

Na podstawie wiedzy zawartej w poprzednim ćwiczeniu należy zaimplementować własną usługę REST bazującą na własnym modelu danych, oferującą podstawowe operacje typu CRUD. Usługę należy przetestować za pomocą aplikacji Postman i dołączyć wyeksportowaną kolekcję z zapytaniami do rozwiązania. Rozwiązanie powinno składać się z kompletnego projektu aplikacji oraz kolekcji Postman z zapytaniami testującymi. Dopuszczalne jest użycie innej technologii implementacji niż Java Spring, w takim przypadku należy dołączyć dokładną instrukcję instalacji i uruchomienia programu.