

8. INSTALACJA BAZY POSTGRESQL

8.1. Należy ściągnąć najnowszą wersję bazy PostgreSQL (<https://www.enterprisedb.com/downloads/postgres-postgresql-downloads>). Podczas instalacji w systemie Windows zakładany jest użytkownik o nazwie *postgres*, który uruchamia PostgreSQL-a jako serwis. Zakładany jest również wewnętrzny użytkownik baz danych o tej samej nazwie, który ma prawa administratora dotyczące dostępu do baz. Jako hasło dla tego użytkownika można wpisać *postgres*. Dla ułatwienia pracy warto dodać do zmiennej środowiskowej *Path* ścieżkę do katalogu *bin* z narzędziami PostgreSQL-a (przykładowa ścieżka *C:\Program Files\PostgreSQL\9.5\bin*). Pamiętaj, że dodawana ścieżka musi być oddzielona średnikiem od już istniejących wpisów w zmiennej *Path*. W systemie Windows 10 podczas definiowania ścieżki do podkatalogu *bin* nie należy dodawać średnika (ze względu na sposób edycji zmiennej środowiskowej w tym systemie). Natomiast we wszystkich starszych systemach trzeba podczas wpisywania oddzielić ścieżkę średnikiem od już istniejących wpisów. Po każdej modyfikacji zmiennej środowiskowej zalecane jest ponownie uruchomienie komputera. Ścieżka wskazującą na podkatalog *bin* PostgreSQL-a została poprawnie dodana do zmiennej środowiskowej *Path*, jeżeli komenda *psql* jest rozpoznawana w windowsowym *Wierszu polecenia*.

8.2. Utwórz bazę danych o nazwie *projekty*. Możesz skorzystać z instalowanego wraz z bazą PostgreSQL programu *PgAdmin* lub wpisać w windowsowym *Wierszu polecenia* (użycie konsoli wymaga, aby w zmiennej środowiskowej *Path* była dodana ścieżka do katalogu z narzędziami PostgreSQL-a):

```
createdb --username=postgres projekty
```

(Zamiast *--username=postgres* można używać *-U postgres*, a także pomijać wpisywanie użytkownika i jego hasła, jeżeli zdefiniujesz zmienne systemowe *PGUSER* i *PGPASSWORD*. Pamiętaj, że w środowisku produkcyjnym korzystanie z takiego rozwiązania jest niedopuszczalne.)

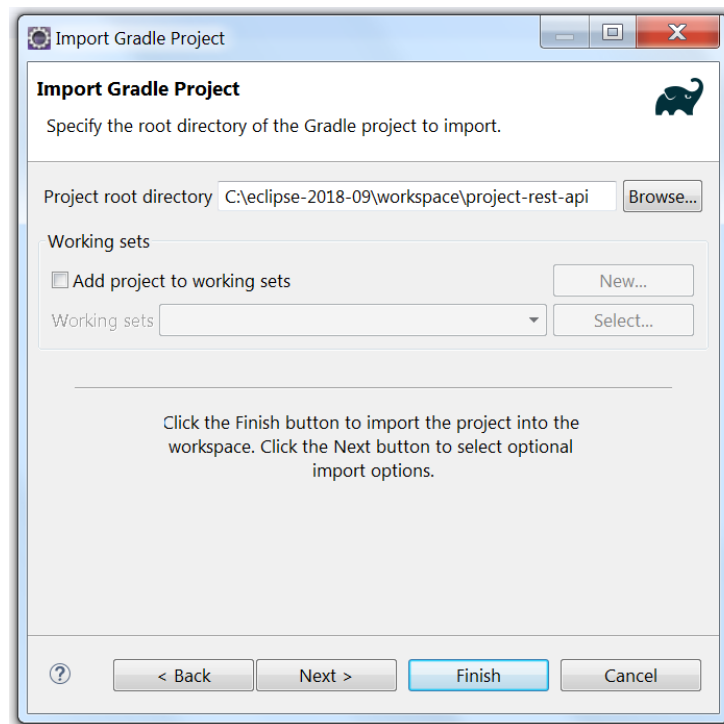
9. GENEROWANIE PROJEKTU

9.1. Otwórz stronę <https://start.spring.io> i wygeneruj szkielet projektu. Użyj przedstawionych poniżej nazw, ustawień i zależności.



Project <input type="checkbox"/> Maven Project <input checked="" type="checkbox"/> Gradle Project	Language <input checked="" type="checkbox"/> Java <input type="checkbox"/> Kotlin <input type="checkbox"/> Groovy	Dependencies ADD DEPENDENCIES... CTRL + B
Spring Boot <input type="checkbox"/> 2.3.0 M4 <input type="checkbox"/> 2.3.0 (SNAPSHOT) <input type="checkbox"/> 2.2.7 (SNAPSHOT) <input checked="" type="checkbox"/> 2.2.6 <input type="checkbox"/> 2.1.14 (SNAPSHOT) <input type="checkbox"/> 2.1.13		Spring Data JPA SQL Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.
Project Metadata Group <input type="text" value="com.project"/> Artifact <input type="text" value="project-rest-api"/> Name <input type="text" value="project-rest-api"/> Description <input type="text" value="Back-end of project management application"/> Package name <input type="text" value="com.project.rest"/> Packaging <input checked="" type="checkbox"/> Jar <input type="checkbox"/> War Java <input type="checkbox"/> 14 <input type="checkbox"/> 11 <input checked="" type="checkbox"/> 8		Spring Web WEB Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
		PostgreSQL Driver SQL A JDBC and R2DBC driver that allows Java programs to connect to a PostgreSQL database using standard, database independent Java code.
		Spring Boot DevTools DEVELOPER TOOLS Provides fast application restarts, LiveReload, and configurations for enhanced development experience.
		Spring Security SECURITY Highly customizable authentication and access-control framework for Spring applications.
<div> GENERATE CTRL + G EXPLORE CTRL + SPACE SHARE... </div>		

9.2. Archiwum wygenerowanego projektu rozpakuj bezpośrednio w tzw. przestrzeni projektów (*workspace*) - folderze z projektami środowiska Eclipse. Jeżeli zainstalowałeś Eclipse'a według opisu z niniejszej instrukcji to katalog z projektami będzie znajdował się w *C:\eclipse-2018-09\workspace* (możesz też sprawdzić lokalizację wybierając z menu *File -> Switch Workspace -> Other...*). Następnie w środowisku Eclipse wybierz z menu *File -> Import... -> Gradle / Existing Gradle Project*, wskaż główny katalog rozpakowanego projektu i naciśnij *Finish*.



9.3. Edytuj plik *project-rest-api/src/main/resources/application.properties* i dodaj poniższe parametry konfiguracyjne.

```
# Spring DataSource
spring.datasource.url=jdbc:postgresql://localhost:5432/projekty
spring.datasource.username=postgres
spring.datasource.password=postgres
spring.datasource.driver-class-name=org.postgresql.Driver

# Spring JPA
# The SQL dialect makes Hibernate generate better SQL for the chosen database
# (Postgres 9.5 and later)
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQL95Dialect
# Validation or export of schema DDL to the database (create, create-drop, validate, update, none)
spring.jpa.hibernate.ddl-auto=update
# Logging JPA Queries
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true

# Spring Security
# HTTP authentication credentials
spring.security.user.name=admin
spring.security.user.password=admin
```

10. IMPLEMENTACJA USŁUGI

10.1. Użyjemy uwierzytelniania typu *Basic Authentication* – powszechnie stosowanego i prostego zabezpieczenia usługi przed niepożądanym dostępem. Zastosowanie uwierzytelniania tego typu bez użycia protokołu szyfrowania np. TLS (rozwiniecie protokołu SSL) nie zapewnia ochrony przekazywanych danych (można np. podejrzec przesyłane w *Base64* login i hasło). Innym często wykorzystywanym i bardziej zaawansowanym mechanizmem zabezpieczającym jest *JWT (JSON Web Token)*, który również można by zastosować w naszej usłudze.

Login i hasło wykorzystywane w *Basic Authentication* zostały zdefiniowane w pliku *application.properties* (p. 9.3) jako wartości parametrów *spring.security.user.name* i *spring.security.user.password*. Pozostaje jeszcze utworzyć w pakiecie *com.project.rest.config* klasę *SecurityConfig* dziedziczącą z *WebSecurityConfigurerAdapter*.

Należy jedynie nadpisać metodę *configure*, tak jak to zostało przedstawione poniżej. Proszę zwrócić uwagę, na adnotację *@Configuration*, która wskazuje Springowi klasę konfiguracyjną. Podczas uruchamiania aplikacji framework Spring skanuje pliki poszukując w nich różnych adnotacji i na ich podstawie podejmuje odpowiednie działania m.in. tworzy obiekty, wstrzykuje zależności, konfiguruje mechanizmy itd.

```
package com.project.config;

import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .csrf().disable()
            .authorizeRequests()
            .anyRequest().authenticated()
            .and()
            .httpBasic();
    }
}
```

10.2. Utworzenie klas encyjnych odwzorowujących bazodanowe tabele zostało opisane w punktach od 4.1 do 4.4. Można oczywiście do *src/main/java* naszego projektu przekopiować pakiet modelu utworzony w ramach poprzedniego zadania. W klasach dodamy też kilka nowych adnotacji – do automatycznej walidacji (*javax.validation.constraints.**) oraz określającą sposób odwzorowania w JSON-ie dwukierunkowych relacji zastosowanych w modelu. Bez tej ostatniej adnotacji próba pobrania projektu z zadaniami spowodowałaby zapętlenie podczas mapowania obiektów do formatu JSON, tak jak przedstawiono poniżej. Adnotacja *@JsonIgnoreProperties({"projekt"})* przed listą zadań pozwoli pominąć podczas mapowania obiektów zmienną *projekt* klasy *Zadanie* i tym samym rozwiąże problem. Istnieje jeszcze kilka innych adnotacji eliminujących zapętlenia m.in. *@JsonManagedReference* i *@JsonBackReference*, *@JsonView* oraz *@JsonIdentityReference* i *@JsonIdentityInfo*.

```
{
  "projectId": 8,
  "nazwa": "Rozpoznawanie znaków drogowych",
  "opis": "Projekt i implementacja wielowarstwowej sieci neuronowej.",
  "dataCzasUtworzenia": "2020-04-17T22:25:35.258",
  "dataCzasModyfikacji": "2020-04-17T22:25:35.258",
  "dataOddania": "2020-06-15",
  "zadania": [
    {
      "zadanieId": 9,
      "nazwa": "Przygotowanie zbioru testowego.",
      "opis": "Utworzenie obrazków niskiej rozdzielczości w formacie PNG.",
      "kolejnosc": 1,
      "dataCzasDodania": "2020-04-17T22:25:35.319",
      "projekt": {
        "projectId": 8,
        "nazwa": "Rozpoznawanie znaków drogowych",
        "opis": "Projekt i implementacja wielowarstwowej sieci neuronowej.",
        "dataCzasUtworzenia": "2020-04-17T22:25:35.258",
        "dataCzasModyfikacji": "2020-04-17T22:25:35.258",
        "dataOddania": "2020-06-15",
        "zadania": [
          {
            "zadanieId": 9,
            .....
```

Poniżej przykład klasy *Projekt*. Dodaj w pozostałych klasach encyjnych analogiczne adnotacje.

```
package com.project.model;

...
import javax.validation.constraints.NotNull;
import javax.validation.constraints.NotBlank;
import com.fasterxml.jackson.annotation.JsonIgnoreProperties;

@Entity
@Table(name = "projekt")
public class Projekt {

    ...    //lepszym rozwiązaniem jest przechowywanie komunikatów poza kodem źródłowym np. w plikach *.properties
    @NotBlank(message = "Pole nazwa nie może być puste!")
    @Size(min = 3, max = 50, message = "Nazwa musi zawierać od {min} do {max} znaków!")
    @Column(nullable = false, length = 50)
    private String nazwa;

    ...

    @OneToMany(mappedBy = "projekt")
    @JsonIgnoreProperties({"projekt"})
    private List<Zadanie> zadania;

    ...
}
```

10.3. Utworzenie repozytoriów. Podstawowych metod bazodanowych nie trzeba samodzielnie implementować, wystarczy tylko utworzenie interfejsu dziedziczącego z *JpaRepository<T, ID>* (gdzie: *T* – klasa encyjna, *ID* – typ identyfikatora).

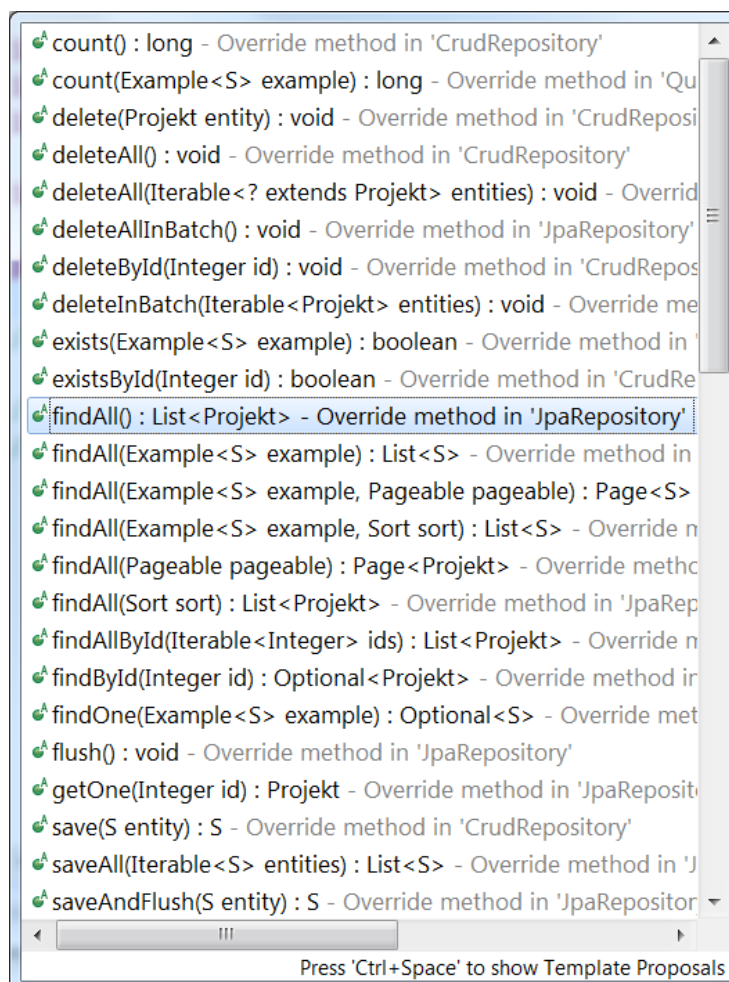
```
package com.project.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import com.project.model.Projekt;

public interface ProjektRepository extends JpaRepository<Projekt, Integer> {

}
```

Samo zdefiniowanie powyższego interfejsu pozwala na korzystanie ze wszystkich przedstawionych na obrazku metod.



W przypadku, gdy potrzebne są bardziej zaawansowane metody bazodanowe możemy użyć np. zapytań wbudowanych w nazwę metody – np. `findBy{query}` lub adnotacji `@Query(...)`. Więcej informacji na ten temat można znaleźć na stronie: <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods>. Utwórz w projekcie pakiet `com.project.repository` i dodaj do niego trzy poniższe interfejsy – `ProjektRepository`, `ZadanieRepository` i `StudentRepository`.

```
package com.project.repository;
import java.util.List;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.JpaRepository;
import com.project.model.Projekt;

public interface ProjektRepository extends JpaRepository<Projekt, Integer> {
    Page<Projekt> findByNazwaContainingIgnoreCase(String nazwa, Pageable pageable);
    List<Projekt> findByNazwaContainingIgnoreCase(String nazwa);

    // Metoda findByNazwaContainingIgnoreCase definiuje zapytanie
    // SELECT p FROM Projekt p WHERE upper(p.nazwa) LIKE upper(:%nazwa%)
}

package com.project.repository;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import com.project.model.Zadanie;

public interface ZadanieRepository extends JpaRepository<Zadanie, Integer> {
    //dwukropkiem oznacza się parametry zapytania
    @Query("SELECT z FROM Zadanie z WHERE z.projekt.projektId = :projektId")
    Page<Zadanie> findZadaniaProjektu(@Param("projektId") Integer projektId, Pageable pageable);

    @Query("SELECT z FROM Zadanie z WHERE z.projekt.projektId = :projektId")
    List<Zadanie> findZadaniaProjektu(@Param("projektId") Integer projektId);
}

package com.project.repository;
import java.util.Optional;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.JpaRepository;
import com.project.model.Student;

public interface StudentRepository extends JpaRepository<Student, Integer> {
    Optional<Student> findByNrIndeksu(String nrIndeksu);
    Page<Student> findByNrIndeksuStartsWith(String nrIndeksu, Pageable pageable);
    Page<Student> findByNazwiskoStartsWithIgnoreCase(String nazwisko, Pageable pageable);

    // Metoda findByNrIndeksuStartsWith definiuje zapytanie
    // SELECT s FROM Student s WHERE s.nrIndeksu LIKE :nrIndeksu%

    // Metoda findByNazwiskoStartsWithIgnoreCase definiuje zapytanie
    // SELECT s FROM Student s WHERE upper(s.nazwisko) LIKE upper(:nazwisko%)
}
```

10.4. Utworzenie serwisów. Powyższe repozytoria definiują bezpośredni dostęp do danych poprzez bazodanowe zapytania, są podstawowymi elementami, a w zasadzie interfejsami tzw. warstwy persystencji. Zwykle chcemy opakować naszą interakcję z bazą danych w warstwę pośrednią – serwisy, które tworzą tzw. warstwę logiki biznesowej. Serwis zwykle korzysta z niższej warstwy persystencji, ale może także używać innych klas z tej samej warstwy. W naszej aplikacji utworzymy serwisy domenowe, których metody będą wywoływane przez kontrolery. Oczywiście wszystkie implementacje serwisów będą również hermetyzowane przy pomocy interfejsów. Utwórz interfejs `ProjektService` w pakiecie `com.project.service`, następnie dodaj klasę `ProjektServiceImpl` z jego implementacją. W podobny sposób utwórz pozostałe serwisy dla zadań i studentów.


```

package com.project.service;

import java.util.Optional;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import com.project.model.Projekt;

public interface ProjektService {

    Optional<Projekt> getProjekt(Integer projektId);
    Projekt setProjekt(Projekt projekt);
    void deleteProjekt(Integer projektId);
    Page<Projekt> getProjekty(Pageable pageable);
    Page<Projekt> searchByNazwa(String nazwa, Pageable pageable);

}

```

W poniższej klasie mamy kilka adnotacji wymagających wyjaśnienia np.:

- *@Service* - adnotacja oznacza, że klasa będzie zarządzana przez Springa, pełni taką samą rolę co adnotacja *@Component* z dodatkowym wskazaniem na klasę warstwy logiki biznesowej.
- *@Autowired* – oznacza, że Spring zajmie się wstrzyknięciem instancji klasy *ProjektRepository* do zmiennej *projektRepository* (wcześniej oczywiście też sam utworzy obiekt tej klasy). W tym przypadku jest to tzw. wstrzykiwanie zależności poprzez konstruktor. Parametr wejściowy konstruktora określa klasę obiektu, który przeznaczony jest do wstrzyknięcia. Jeżeli Springowi nie uda się utworzyć żadnego obiektu, który ma być wstrzyknięty lub będzie możliwość utworzenia kilku takich obiektów to pojawi się błąd podczas uruchamiania aplikacji (tego typu problemy można również rozwiązywać za pomocą specjalnych adnotacji). W najnowszych wersjach frameworku adnotacja przed konstruktorem nawet może być pomijana, ponieważ wstrzykiwanie przez konstruktor jest działaniem domyślnym.

```

package com.project.service;

import java.util.Optional;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.stereotype.Service;
import com.project.model.Projekt;
import com.project.repository.ProjektRepository;

@Service
public class ProjektServiceImpl implements ProjektService {

    private ProjektRepository projektRepository;

    @Autowired
    public ProjektServiceImpl(ProjektRepository projektRepository) {
        this.projektRepository = projektRepository;
    }

    @Override
    public Optional<Projekt> getProjekt(Integer projektId) {
        return projektRepository.findById(projektId);
    }

    @Override
    public Projekt setProjekt(Projekt projekt) {
        //TODO
        return null;
    }

    @Override
    public void deleteProjekt(Integer projektId) {
        //TODO
    }

    @Override
    public Page<Projekt> getProjekty(Pageable pageable) {
        //TODO
        return null;
    }

}

```

```

@Override
public Page<Projekt> searchByNazwa(String nazwa, Pageable pageable) {
    //TODO
    return null;
}
}

```

Jeżeli w jakiejś metodzie serwisu modyfikującej dane wykonywanych jest kilka operacji bazodanowych i chcemy mieć gwarancję, że w przypadku niepowodzenia którejkolwiek z nich pozostałe zmiany zostaną wycofane to musimy skorzystać z adnotacji *@Transactional*. Dzięki niej wszystkie operacje uruchamiane wewnątrz metody zostaną wykonane w jednej transakcji bazodanowej (adnotację można wstawiać też przed nazwą klasy, wtedy wszystkie jej metody zostaną uwzględnione).

W naszym modelu nie ustawialiśmy kaskadowości operacji, która określa co ma się dzieć z zależnymi encjami podczas modyfikacji obiektu głównego np. możemy określić co zrobimy z zadaniami projektu w momencie jego usunięcia (mamy do dyspozycji kilka opcji m.in.: *CascadeType.PERSIST*, *CascadeType.MERGE*, *CascadeType.REMOVE*, *CascadeType.ALL*). W obecnej wersji podczas próby usunięcia projektu z zadaniami dostaniemy błąd (nie jest to oczywiście błąd, który trzeba naprawiać, często nawet unika się kaskadowego usuwania danych) np.:

[org.postgresql.util.PSQLException](#): BŁĄD: modyfikacja lub usunięcie na tabeli "projekt" narusza klucz obcy "fkauptfb9wjo0o9fu7bm2s5tifid" tabeli "zadanie"
Szczegóły: Klucz (projekt_id)=(7) ma wciąż odwołanie w tabeli "zadanie".

...

Zatem jeżeli nie dodamy odpowiedniego ustawienia *CascadeType* do adnotacji *@OneToMany* to będziemy musieli podczas usuwania projektu zadbać o wcześniejsze usunięcie jego zadań. W takim przypadku zwykle chcemy mieć gwarancję, że usuniemy wszystko albo nic. Poniższa, przykładowa metoda to zapewnia dzięki przetwarzaniu transakcyjnemu.

```

@Service
public class ProjektServiceImpl implements ProjektService {

    private ProjektRepository projektRepository;
    private ZadanieRepository zadanieRepository;

    @Autowired // w tej wersji konstruktora Spring wstrzyknie dwa repozytoria
    public ProjektServiceImpl(PjektRepository projektRepository, ZadanieRepository zadanieRepo) {
        this.projektRepository = projektRepository;
        this.zadanieRepository = zadanieRepo;
    }

    ...

    @Override
    @Transactional
    public void deleteProjekt(Integer projektId) {
        for (Zadanie zadanie : zadanieRepository.findZadaniaProjektu(projektId)) {
            zadanieRepository.delete(zadanie);
        }
        projektRepository.deleteById(projektId);
    }
}

```

Do realizacji transakcji wykorzystywany jest springowy moduł programowania aspektowego, powyższy kod jest równoważny fragmentowi:

```

EntityManager etx = entityManager.getTransaction();
try {
    etx.begin();
    deleteProjekt(projektId); //wywołanie metody oznaczonej adnotacją @Transactional
    etx.commit();
} catch (Exception e) {
    etx.rollback();
    throw e;
}

```

10.5. Utworzenie kontrolerów. Proszę zwrócić uwagę w poniższej tabeli na adresy tzw. endpointów, jak widać do różnych akcji używamy zawsze tego samego głównego adresu, ale różnych metod HTTP oraz parametrów dołączanych do adresu (wartość identyfikatora zasobu umieszczana jest bezpośrednio w adresie, na jego końcu, a po znaku zapytania w formacie *klucz=wartość* definiuje się dodatkowe parametry np. stronicowania, sortowania, wyszukiwania). Takie podejście nie jest obligatoryjne, ale jest praktyką rekomendowaną.

ADRES ZASOBU: http://localhost:8080/api/projekty		
AKCJA	METODA HTTP	ADRES ENDPOINTA
POBIERANIE PROJEKTU (metoda kontrolera: <i>getProjekt</i>)	GET	.../api/projekty/{projektId}
POBIERANIE PROJEKTÓW (metoda kontrolera: <i>getProjekty</i>)	GET	.../api/projekty[?page=0&size=10&sort=nazwa]
WYSZUKANIE PROJEKTÓW (metoda kontrolera: <i>getProjektyByNazwa</i>)	GET	.../api/projekty?nazwa=wartosc[&page=0&size=10&sort=nazwa]
UTWORZENIE PROJEKTU (metoda kontrolera: <i>createProjekt</i>)	POST	.../api/projekty
MODYFIKACJA PROJEKTU (metoda kontrolera: <i>updateProjekt</i>)	PUT	.../api/projekty/{projektId}
USUWANIE PROJEKTU (metoda kontrolera: <i>deleteProjekt</i>)	DELETE	.../api/projekty/{projektId}

Przeanalizuj poniższy kod źródłowy kontrolera, zwróć uwagę na adnotacje i komentarze. Utwórz pakiet *com.project.rest.controller* i zdefiniuj w nim trzy kontrolery dla projektu, zadania i studenta.

```
package com.project.rest.controller;

import java.net.URI;
import javax.validation.Valid;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.servlet.support.ServletUriComponentsBuilder;
import com.project.model.Projekt;
import com.project.service.ProjektService;

// dzięki adnotacji @RestController klasa jest traktowana jako zarządzany
// przez Springa REST-owy kontroler obsługujący sieciowe żądania
@RestController
@RequestMapping("/api") // adnotacja @RequestMapping umieszczona w tym miejscu pozwala definiować
public class ProjektRestController { // część wspólną adresu, wstawianą przed wszystkimi poniższymi ścieżkami

    private ProjektService projektService; //serwis jest automatycznie wstrzykiwany poprzez konstruktor

    @Autowired
    public ProjektRestController(ProjektService projektService) {
        this.projektService = projektService;
    }

    // PRZED KAŻDĄ Z PONIŻSZYCH METOD JEST UMIESZCZONA ADNOTACJA (@GetMapping, PostMapping, ... ), KTÓRA OKREŚLA
    // RODZAJ METODY HTTP, A TAKŻE ADRES I PARAMETRY ŻĄDANIA
```



```
//Przykład żądania wywołującego metodę: GET http://localhost:8080/api/projekty/1
@GetMapping("/projekty/{projektId}")
ResponseBody<Projekt> getProjekt(@PathVariable Integer projektId) { // @PathVariable oznacza, że wartość
    return ResponseEntity.of(projektService.getProjekt(projektId)); // parametru przekazywana jest w ścieżce
}

// @Valid włącza automatyczną walidację na podstawie adnotacji zawartych
// w modelu np. NotNull, Size, NotEmpty itp. (z javax.validation.constraints.*)
@PostMapping(path = "/projekty")
ResponseBody<Void> createProjekt(@Valid @RequestBody Projekt projekt) { // @RequestBody oznacza, że dane
    // projektu (w formacie JSON) są
    // przekazywane w ciele żądania
    Projekt createdProjekt = projektService.setProjekt(projekt);
    URI location = ServletUriComponentsBuilder.fromCurrentRequest() // link wskazujący utworzony projekt
        .path("/{projektId}").buildAndExpand(createdProjekt.getProjektId()).toUri();
    return ResponseEntity.created(location).build(); // zwracany jest kod odpowiedzi 201 - Created
    // z linkiem location w nagłówku
}

@PutMapping("/projekty/{projektId}")
public ResponseEntity<Void> updateProjekt(@Valid @RequestBody Projekt projekt,
                                           @PathVariable Integer projektId) {
    return projektService.getProjekt(projektId)
        .map(p -> {
            projektService.setProjekt(projekt);
            return new ResponseEntity<Void>(HttpStatus.OK); // 200 (można też zwracać 204 - No content)
        })
        .orElseGet(() -> ResponseEntity.notFound().build()); // 404 - Not found
}

@DeleteMapping("/projekty/{projektId}")
public ResponseEntity<Void> deleteProjekt(@PathVariable Integer projektId) {
    return projektService.getProjekt(projektId).map(p -> {
        projektService.deleteProjekt(projektId);
        return new ResponseEntity<Void>(HttpStatus.OK); // 200
    }).orElseGet(() -> ResponseEntity.notFound().build()); // 404 - Not found
}

//Przykład żądania wywołującego metodę: http://localhost:8080/api/projekty?page=0&size=10&sort=nazwa,desc
@GetMapping(value = "/projekty")
Page<Projekt> getProjekty(Pageable pageable) { // @RequestHeader HttpHeaders headers - jeżeli potrzebny
    return projektService.getProjekty(pageable); // byłby nagłówek, wystarczy dodać drugą zmienną z adnotacją
}

// Przykład żądania wywołującego metodę: GET http://localhost:8080/api/projekty?nazwa=webowa
// Metoda zostanie wywołana tylko, gdy w żądaniu będzie przesyłana wartość parametru nazwa.
@GetMapping(value = "/projekty", params="nazwa")
Page<Projekt> getProjektyByNazwa(@RequestParam String nazwa, Pageable pageable) {
    return projektService.searchByNazwa(nazwa, pageable);
}
}
```

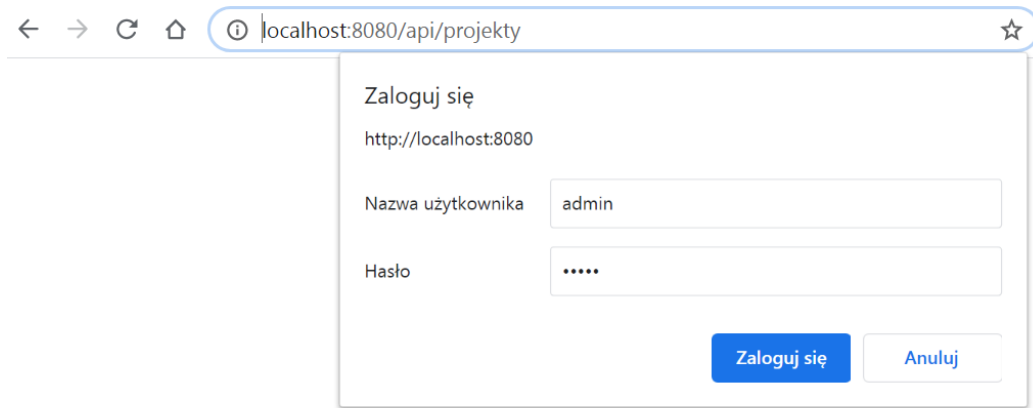
10.6. Próba uruchomienia aplikacji za pomocą klasy *ProjectRestApiApplication* zakończy się błędem, ponieważ Spring podczas inicjalizacji poszukuje adnotacji w pakiecie klasy uruchomieniowej oraz wszystkich jego podpakietach. Chociaż w parametrze adnotacji *@SpringBootApplication* można wskazywać pakiety z adnotacjami to w naszym przypadku najprostszym rozwiązaniem problemu będzie zmiana nazwy pakietu przenosząca klasę *ProjectRestApiApplication* poziom wyżej. W widoku *Project Explorer* zaznacz główną ikonkę pakietu *com.project.rest*, następnie wciśnij prawy przycisk myszy i wybierz z menu *Refactor -> Rename*. W polu tekstowym okienka pozostaw *com.project* i kliknij *OK*. Aby uruchomić aplikację kliknij prawym przyciskiem myszki wewnątrz okna z kodem źródłowym lub na ikonke klasy *ProjectRestApiApplication* i wybierz *Run As -> Java Application*.

```
package com.project;

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

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

Sprawdź, czy w konsoli środowiska Eclipse nie zostały wyświetlone jakieś błędy, powinna być też informacja o uruchomieniu serwera np. *Tomcat started on port(s): 8080 (http) with context path "*. Jeżeli aplikacja została poprawnie uruchomiona otwórz przeglądarkę i wpisz adres, który wywoła metodę pobierającą dane wszystkich projektów tj. <http://localhost:8080/api/projekty>.



Po podaniu nazwy i hasła (patrz p. 10.1) otrzymamy odpowiedź:

```
{"content": [], "pageable": {"sort": {"sorted": false, "unsorted": true, "empty": true}, "offset": 0, "pageNumber": 0, "pageSize": 20, "unpaged": false, "paged": true}, "totalPages": 0, "totalElements": 0, "last": true, "size": 20, "number": 0, "sort": {"sorted": false, "unsorted": true, "empty": true}, "numberOfElements": 0, "first": true, "empty": true}
```

W bazie nie są jeszcze przechowywane żadne dane, więc składnik *content* jest pusty. Struktura i pozostałe elementy odpowiedzi służą do stronicowania. Przekazywane wartości są ustawiane przez użyte w metodach kontrolera obiekty klas *Page* i *Pageable*. Jeżeli zmienimy, tak jak poniżej, zwracany typ metody kontrolera to w odpowiedzi otrzymamy tylko puste nawiasy prostokątne.

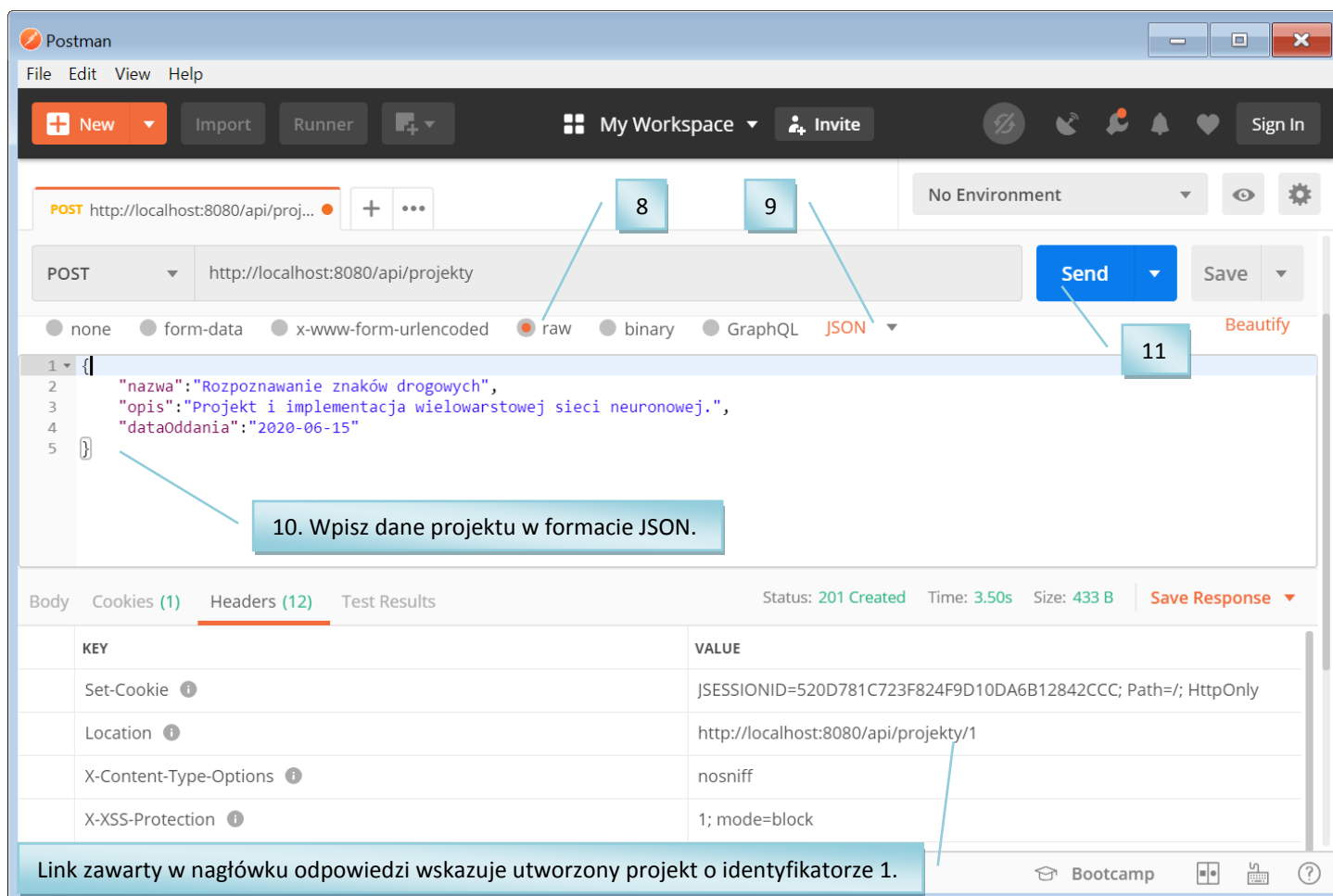
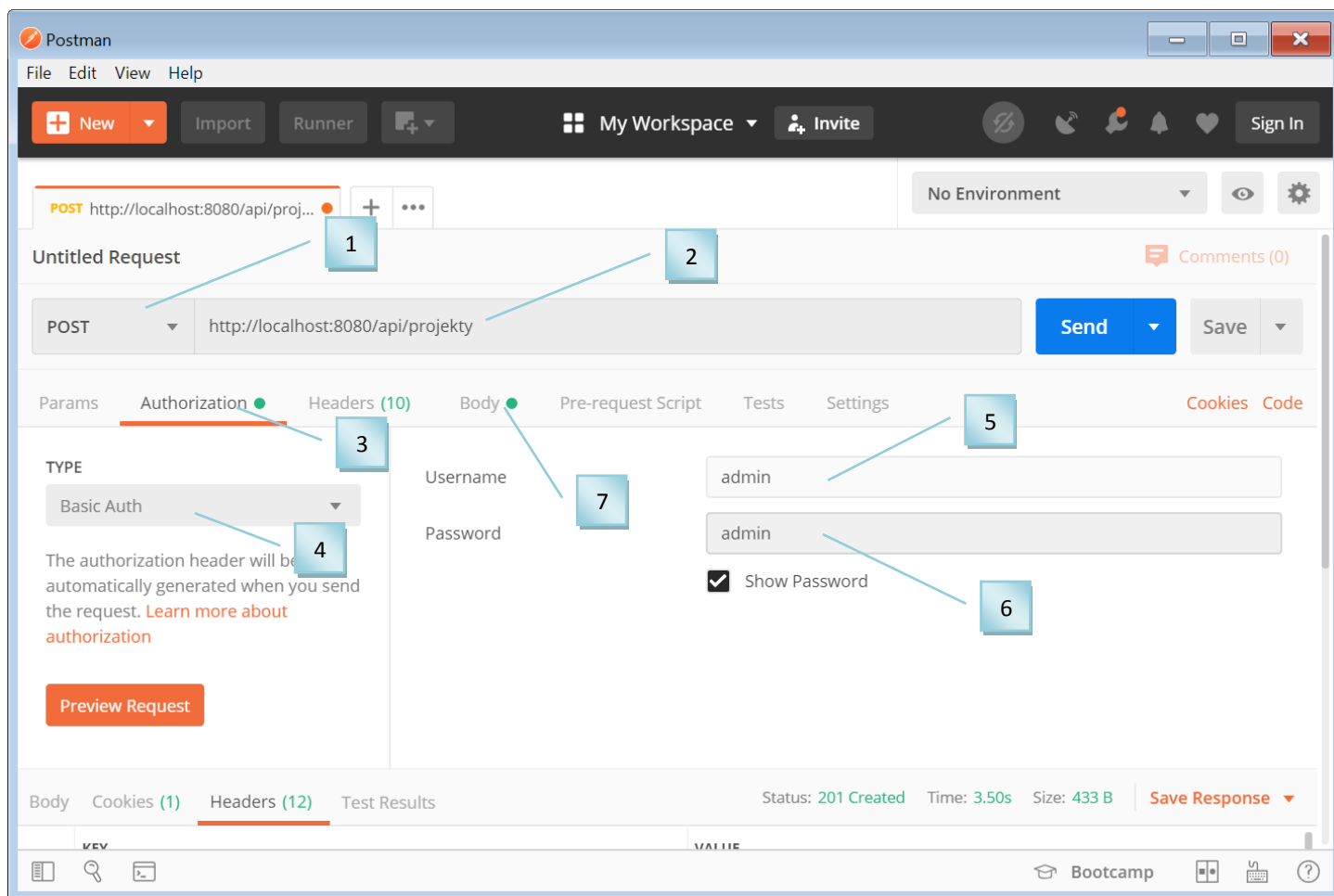
```
@GetMapping(value = "/projekty")
List<Projekt> getProjekty() {
    return projektService.getProjekty(PageRequest.of(0, 20)).getContent();
}
```

11. TESTOWANIE REST API

11.1. Ściągnij darmową wersję aplikacji *Postman* (<https://www.postman.com/downloads/>), za pomocą której przetestujesz endpointy kontrolerów. Na początek utwórz kilka projektów w bazie danych. Dla wybranego projektu utwórz też parę zadań. Później sprawdź pobieranie danych z dodatkowymi parametrami stronicowania i wyszukiwania dołączając np.

...?nazwa=jakieś_wyszukiwane_słowo_lub_jego_fragment&page=0&size=10&sort=nazwa,desc

Na koniec usuń projekt z przypisanymi zadaniami.



11.2. Testowanie kontrolerów można zautomatyzować za pomocą testów korzystających ze springowej biblioteki MockMVC. Na początek edytuj plik *project-rest-api/build.gradle* i w bloku *test* dodaj fragment włączający drukowanie komunikatów w konsoli. Dodaj też bloki *compileJava* i *compileTestJava* jeżeli w środowisku Eclipse ustawione jest windowsowe kodowanie znaków. Następnie kliknij prawym przyciskiem myszki na głównej ikonce projektu i wybierz *Gradle -> Refresh Gradle Project*.

```
...
compileJava {
    options.encoding = 'windows-1250'
}                                     // potrzebne, gdy edytor kodu źródłowego środowiska
                                     // Eclipse ma ustawione windowsowe kodowanie
compileTestJava{
    options.encoding = 'windows-1250'
}

...
test {
    useJUnitPlatform() //aktywacja natywnego wsparcia JUnit 5 (od wersji 4.6 Gradle'a)
    testLogging {
        showStandardStreams = true //ustawia drukowanie komunikatów w konsoli
    }
}
```

W *src\test\java* utwórz pakiet *com.project.rest* i dodaj do niego klasę *ProjektRestControllerTest*. Następnie przekopiuj przedstawioną poniżej klasę. Przeanalizuj jej zawartość, uruchom testy i sprawdź wydruki w konsoli środowiska Eclipse.

```

package com.project.rest;

import static org.hamcrest.Matchers.containsString;
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotNull;
import static org.junit.jupiter.api.Assertions.assertTrue;
import static org.mockito.ArgumentMatchers.any;
import static org.mockito.Mockito.times;
import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.verifyNoMoreInteractions;
import static org.mockito.Mockito.when;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.post;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.put;
import static org.springframework.test.web.servlet.result.MockMvcResultHandlers.print;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.header;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.jsonPath;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.util.Collections;
import java.util.Optional;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.TestInfo;
import org.mockito.ArgumentCaptor;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.json.JacksonTester;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageImpl;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Pageable;
import org.springframework.data.domain.Sort;
import org.springframework.http.MediaType;
import org.springframework.security.test.context.support.WithMockUser;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.MvcResult;
import org.springframework.web.bind.MethodArgumentNotValidException;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.SerializationFeature;
import com.fasterxml.jackson.datatype.jsr310.JavaTimeModule;
import com.project.model.Projekt;
import com.project.service.ProjektService;

@SpringBootTest
@AutoConfigureMockMvc
@WithMockUser(username = "admin", password = "admin")
public class ProjektRestControllerTest {
    // Uwaga! Test wymaga poniższego konstruktora w klasie Projekt, dodaj jeżeli nie został jeszcze zdefiniowany.
    // public Projekt(Integer projektId, String nazwa, String opis, LocalDateTime dataCzasUtworzenia, LocalDate
    //                                                                 dataOddania){
    // ...
    //}

    // --- URUCHAMIANIE TESTÓW ---
    // ABY URUCHOMIĆ TESTY KLIKNIJ NA NAZWIE KLASY PRAWYM PRZYCISKIEM
    // MYSZY I WYBIERZ Z MENU 'Run As' -> 'Gradle Test' LUB PO USTAWIENIU
    // KURSORA NA NAZWIE KLASY WCIŚNIJ SKRÓT 'CTRL+ALT+X' A PÓŹNIEJ 'G'
    // MOŻNA RÓWNIEŻ ANALOGICZNIE URUCHAMIAĆ POJEDYNCZE METODY KLIKAJĄC
    // WCZEŚNIEJ NA ICH NAZWĘ

    private final String apiPath = "/api/projekty";

    @MockBean
    private ProjektService mockProjektService; //tzw. mock (czyli obiekt, którego używa się zamiast rzeczywistej
    //implementacji) serwisu wykorzystywany przy testowaniu kontrolera

    @Autowired
    private MockMvc mockMvc;

    private JacksonTester<Projekt> jsonTester;

```



```

@Test
public void getProjekt() throws Exception {
    Projekt projekt = new Projekt(1, "Nazwa1", "Opis1", LocalDateTime.now(), LocalDate.of(2020, 6, 7));
    Page<Projekt> page = new PageImpl<>(Collections.singletonList(projekt));
    when(mockProjektService.getProjekt(any(Pageable.class))).thenReturn(page);

    mockMvc.perform(get(apiPath).contentType(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk())
        .andExpect(jsonPath("$.content[*]").exists()) //content[*] - oznacza całą zawartość tablicy content
        .andExpect(jsonPath("$.content.length()").value(1))
        .andExpect(jsonPath("$.content[0].projektId").value(projekt.getProjektId()))
        .andExpect(jsonPath("$.content[0].nazwa").value(projekt.getNazwa()));

    verify(mockProjektService, times(1)).getProjekt(any(Pageable.class));
    verifyNoMoreInteractions(mockProjektService);
}

@Test
public void getProjekt() throws Exception {
    Projekt projekt = new Projekt(2, "Nazwa2", "Opis2", LocalDateTime.now(), LocalDate.of(2020, 6, 7));
    when(mockProjektService.getProjekt(projekt.getProjektId()))
        .thenReturn(Optional.of(projekt));

    mockMvc.perform(get(apiPath +("/{projektId})", projekt.getProjektId()).accept(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk())
        .andExpect(jsonPath("$.projektId").value(projekt.getProjektId()))
        .andExpect(jsonPath("$.nazwa").value(projekt.getNazwa()));

    verify(mockProjektService, times(1)).getProjekt(projekt.getProjektId());
    verifyNoMoreInteractions(mockProjektService);
}

@Test
public void createProjekt() throws Exception {
    Projekt projekt = new Projekt(null, "Nazwa3", "Opis3", null, LocalDate.of(2020, 6, 7));
    String jsonProjekt = jacksonTester.write(projekt).getJson();
    projekt.setProjektId(3);
    when(mockProjektService.setProjekt(any(Projekt.class))).thenReturn(projekt);

    mockMvc.perform(post(apiPath).content(jsonProjekt).contentType(MediaType.APPLICATION_JSON)
        .accept(MediaType.ALL))
        .andExpect(status().isCreated())
        .andExpect(header().string("location", containsString(apiPath + "/" + projekt.getProjektId())));
}

@Test
public void createProjektEmptyName() throws Exception {
    Projekt projekt = new Projekt(null, "", "Opis4", null, LocalDate.of(2020, 6, 7));
    MvcResult result = mockMvc.perform(post(apiPath)
        .content(jacksonTester.write(projekt).getJson())
        .contentType(MediaType.APPLICATION_JSON)
        .accept(MediaType.ALL))
        .andExpect(status().isBadRequest())
        .andReturn();

    verify(mockProjektService, times(0)).setProjekt(any(Projekt.class));

    Exception exception = result.getResolvedException();
    assertNotNull(exception);
    assertTrue(exception instanceof MethodArgumentNotValidException);
    System.out.println(exception.getMessage());
}

```

```

@Test
public void updateProjekt() throws Exception {
    Projekt projekt = new Projekt(5, "Nazwa5", "Opis5", LocalDateTime.now(), LocalDate.of(2020, 6, 7));
    String jsonProjekt = jsonTester.write(projekt).getJson();

    when(mockProjektService.getProjekt(projekt.getProjektId())).thenReturn(Optional.of(projekt));
    when(mockProjektService.setProjekt(any(Projekt.class))).thenReturn(projekt);

    mockMvc.perform(put(apiPath + "/" + projekt.getProjektId())
        .content(jsonProjekt)
        .contentType(MediaType.APPLICATION_JSON)
        .accept(MediaType.ALL))
        .andDo(print())
        .andExpect(status().isOk());

    verify(mockProjektService, times(1)).getProjekt(projekt.getProjektId());
    verify(mockProjektService, times(1)).setProjekt(any(Projekt.class));
    verifyNoMoreInteractions(mockProjektService);
}

/**
 * Test sprawdza czy żądanie o danych parametrach stronicowania i sortowania
 * spowoduje przekazanie do serwisu odpowiedniego obiektu Pageable, wcześniej
 * wstrzykniętego do parametru wejściowego metody kontrolera
 */
@Test
public void getProjektyAndVerifyPageableParams() throws Exception {
    Integer page = 5;
    Integer size = 15;
    String sortProperty = "nazwa";
    String sortDirection = "desc";
    mockMvc.perform(get(apiPath)
        .param("page", page.toString())
        .param("size", size.toString())
        .param("sort", String.format("%s,%s", sortProperty, sortDirection)))
        .andExpect(status().isOk());

    ArgumentCaptor<Pageable> pageableCaptor = ArgumentCaptor.forClass(Pageable.class);
    verify(mockProjektService, times(1)).getProjekty(pageableCaptor.capture());

    PageRequest pageable = (PageRequest) pageableCaptor.getValue();
    assertEquals(page, pageable.getPageNumber());
    assertEquals(size, pageable.getPageSize());
    assertEquals(sortProperty, pageable.getSort().getOrderFor(sortProperty).getProperty());
    assertEquals(Sort.Direction.DESC, pageable.getSort().getOrderFor(sortProperty).getDirection());
}

@BeforeEach
public void before(TestInfo testInfo) {
    System.out.printf("-- METODA -> %s\n", testInfo.getTestMethod().get().getName());

    ObjectMapper mapper = new ObjectMapper(); // ustawienie formatu daty i czasu w komunikatach
    mapper.registerModule(new JavaTimeModule()); // JSON-a dla zmiennych typu LocalDate i LocalDateTime
    mapper.disable(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS);
    JacksonTester.initFields(this, mapper);
}

@AfterEach
public void after(TestInfo testInfo) {
    System.out.printf("<- KONIEC -- %s\n", testInfo.getTestMethod().get().getName());
}
}

```

11.3. Możesz też wygenerować wersję uruchomieniową usługi REST – plik JAR. W widoku *Gradle Tasks* kliknij prawym przyciskiem myszki na *assemble* (z *project-rest-api/build*) i wybierz *Run Gradle Tasks*. Utworzony plik można znaleźć w katalogu *project-rest-api\build\libs*. Usługę najlepiej uruchamiać za pomocą konsoli, aby można łatwo zamykać proces bezokienkowej aplikacji. W tym celu można utworzyć plik wsadowy np. *run-project-rest-api.bat* z poniższą zawartością:

```

::można wskazać prywatne JRE, np. z udostępnionego pakietu
set path="C:\eclipse-2018-09\eclipse\jdk\bin"
::można też wskazać katalog z plikiem JAR, jeżeli plik BAT jest przechowywany w innym miejscu
cd C:\eclipse-2018-09\workspace\project-rest-api\build\libs
java -jar project-rest-api-0.0.1-SNAPSHOT.jar

```

PRZYDATNE SKRÓTY

CTRL + SHIFT + L – pokazuje wszystkie dostępne skróty

CTRL + SHIFT + F – formatowanie kodu

SHIFT + ALT + R – zmiana nazwy klasy, metody lub zmiennej itp., trzeba wcześniej ustawić kursor na nazwie

SHIFT + ALT + L – utworzenie zmiennej z zaznaczonego fragmentu kodu

SHIFT + ALT + M – utworzenie metody z zaznaczonego fragmentu kodu

CTRL + ALT + STRZAŁKA W GÓRĘ – skopiowanie linijki i wklejenie w bieżącym wierszu

CTRL + ALT + STRZAŁKA W DÓŁ – skopiowania bieżącej linijki i wklejenie poniżej

CTRL + SHIFT + O – automatyczne dodawanie i porządkowanie sekcji importów

CTRL + 1 – „zrób to co chcę zrobić”, m.in. sugestie rozwiązań bieżącego problemu

CTRL + Q – przejście do miejsca ostatniej modyfikacji

F11 – debugowanie aplikacji

CTRL + F11 – uruchomienie aplikacji

CTRL + M – powiększenie/zmniejszenie widoku w perspektywie

Ustawienie kursora np. na wywołaniu metody, typie zmiennej, klasie importu itp. i wciśnięcie **F3** powoduje przejście do kodu źródłowego wywoływanej metody, klasy zmiennej, klasy importu itd.