

Laboratorium 5. *Spring Boot* i *JPA*

Cel zajęć

Realizacja zadań z niniejszego laboratorium umożliwi studentom poznanie elementów konfiguracji projektu *Spring* z wykorzystaniem startera *Spring Boot* [4, 18] i metod pracy z bazą danych za pomocą repozytorium *JPA* [7, 16, 22].

Zakres tematyczny

- Konfiguracja projektu *Spring Boot* (plik konfiguracyjny ***pom.xml*** oraz ***application.properties***, adnotacje do konfiguracji stosowane w startowej klasie projektu).
- Implementacja aplikacji współpracującej z danymi z bazy *H2* (lub *MySQL*) za pomocą interfejsu *Spring Data JPA* i *Hibernate*:
 - definicja klasy encji do obsługi zadań, podstawowe adnotacje w klasie encji i konfiguracja dostępu do bazy danych,
 - definicja interfejsu repozytorium dziedziczącego po interfejsie ***CrudRepository***,
 - wykorzystanie gotowych metod repozytorium do obsługi zadań w aplikacji typu *CRUD* (dodawanie nowych zadań, wyświetlanie listy istniejących, usuwanie wskazanego zadania),
 - implementacja metod wyszukiwania danych za pomocą zapytań wbudowanych lub metod z adnotacją ***@Query***.

Wprowadzenie

Projekt *Spring Boot* ułatwia start z projektem *Spring* i eliminuje on potrzebę pracochłonnego tworzenia konfiguracji w plikach *XML*. Gotową aplikację można utworzyć i uruchomić za pomocą jednej klasy startowej. Dzięki o wiele prostszej konfiguracji, *Spring Boot* nadaje się do projektów studenckich i szybkiego prototypowania aplikacji.

Plikiem konfiguracyjnym środowiska *Spring Boot* jest plik ***application.properties***. Większość konfiguracji tworzy się przy pomocy adnotacji *Spring Boot*, pozostała część znajduje się w pliku ***application.properties***. W *Spring Boot* można korzystać z adnotacji auto-konfiguracji, skanowania w poszukiwaniu komponentów oraz z możliwości definiowania dodatkowych konfiguracji dla klasy startowej aplikacji, poprzedzonej specjalną adnotacją ***@SpringBootApplication***.

Praca z danymi zarówno w projektach *Spring*, jaki i *Spring Boot*, może być realizowana na różne sposoby (podstawowy interfejs *JDBC*, biblioteka *JdbcTemplate*, gotowe interfejsy repozytoriów). Interfejs *JDBC* i *JdbcTemplate* wykorzystano w poprzednich laboratoriach. Nie są to jednak rozwiązania idealne, ponieważ trzeba ręcznie budować zapytania *SQL*, które często się powtarzają, szczególnie w przypadku podstawowych zapytań typu *CRUD*.

Hibernate jest rozwiązaniem tego problemu, ponieważ:

- pozwala automatycznie mapować obiekty języka *Java* na wiersze w bazie danych,
- pozwala odczytywać rekordy z bazy danych i automatycznie tworzyć z nich obiekty w języku *Java*.

Wykorzystując *Hibernate* teoretycznie nie trzeba mieć większego pojęcia o poprawnym konstruowaniu zapytań w języku *SQL*, ponieważ *Hibernate* buduje je sam. *Hibernate* jest najpopularniejszą biblioteką *ORM* do mapowania obiektowo-relacyjnego w Javie. Specyfikacją *JEE* do mapowania obiektowo-relacyjnego jest *JPA* (*Java Persistence API*). Specyfikacja oznacza, że nie jest to żadna biblioteka, a jedynie zbiór definicji i interfejsów, utworzonych przez grono ekspertów, których celem było wprowadzenie do Javy standardu mapowania obiektowo relacyjnego. Główną implementacją standardu *JPA* (*Java Persistence API*) jest obecnie właśnie *Hibernate*. Z kolei *Spring Data JPA* to niewielka biblioteka upraszczająca pracę z *JPA* poprzez automatyczne tworzenie kodu repozytoriów. Centralnym interfejsem dostarczanym przez bibliotekę jest *CrudRepository* i rozszerzający go interfejs *JpaRepository*.

Zadanie 5.1. Konfiguracja projektu w *Spring Boot*

Utwórz projekt Maven zwykłej aplikacji (*New Project* → *Java with Maven* → *Java Application*) o nazwie np. *pai_springboot*. Do pliku *pom.xml* w folderze **Project Files** dodaj element **<parent>** oraz zależność z informacją o tworzonej aplikacji *Web* z wykorzystaniem startera *Spring Boot*. Przy dodawaniu zależności nie podawaj teraz wersji, gdyż wersje poszczególnych artefaktów wyspecyfikowane są w projekcie nadrzędnym, zdefiniowanym jako element **<parent>**. Gotowy plik *pom.xml* przedstawia przykład 5.1 (z *Java* v15, gdzie wersję Javy wskazano kolorem czerwonym).

Przykład 5.1. Plik *pom.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
```

```

<modelVersion>4.0.0</modelVersion>
<groupId>com.bp</groupId>
<artifactId>SpringBoot1</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>jar</packaging>
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <maven.compiler.source>15</maven.compiler.source>
  <maven.compiler.target>15</maven.compiler.target>
</properties>
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.4.5</version>
</parent>
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
</dependencies>
</project>

```

Po uzupełnieniu *pom.xml*, zbuduj projekt i sprawdź, jakie zależności zostały do niego dodane (folder *Dependencies*). Następnie w głównym pakiecie projektu (w tym przykładzie *bp.pai_springboot*) utwórz główną klasę *Main*, której zadaniem będzie uruchomienie aplikacji i obsługa żądań *HTTP* (Przykład 5.2).

Przykład 5.2. Klasa Main

```

package bp.pai_springboot;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
@EnableAutoConfiguration
public class Main {
    public static void main(String[] args) {
        SpringApplication.run(Main.class, args);
    }

    @RequestMapping("/")
    @ResponseBody
    public String mainPage() {
        return "Hello Spring Boot!";
    }
}

```

Utworzona klasa **Main** posiada dwie adnotacje:

- **@EnableAutoConfiguration** z *org.springframework.boot.autoconfigure*, dzięki której aplikacja dokona samokonfiguracji według domyślnych wartości, załaduje potrzebne moduły itp.,
- **@Controller** z pakietu *org.springframework.stereotype* informuje, że klasa obsługuje żądania *HTTP*.

Ważnym składnikiem tej klasy jest zwykła metoda **main()**, wywoływana na początku każdego programu. W niej aplikacja uruchamiana jest za pomocą jednej linii:

```
SpringApplication.run(Main.class, args);
```

Drugą metodą klasy **Main** jest metoda **mainPage()** z dwoma adnotacjami:

- **@RequestMapping("/")** wskazuje, że żądanie z przeglądarki o stronę główną będzie obsługiwała właśnie ta metoda,
- **@ResponseBody** wskazuje, że metoda zwróci ciało odpowiedzi, przesłane do przeglądarki w formacie tekstu (w tym przypadku jest to łańcuch *String* z napisem *Hello Spring Boot*).

Aby uruchomienie aplikacji *Spring Boot* nie stwarzało problemów, w pliku **pom.xml** warto wskazać klasę główną. W tym celu w istniejącym już elemencie **<properties>** dodaj wpis:

```
<start-class>bp.pai_springboot.Main</start-class>
```

wskazujący na pakiet i klasę startową z metodą **main()**.

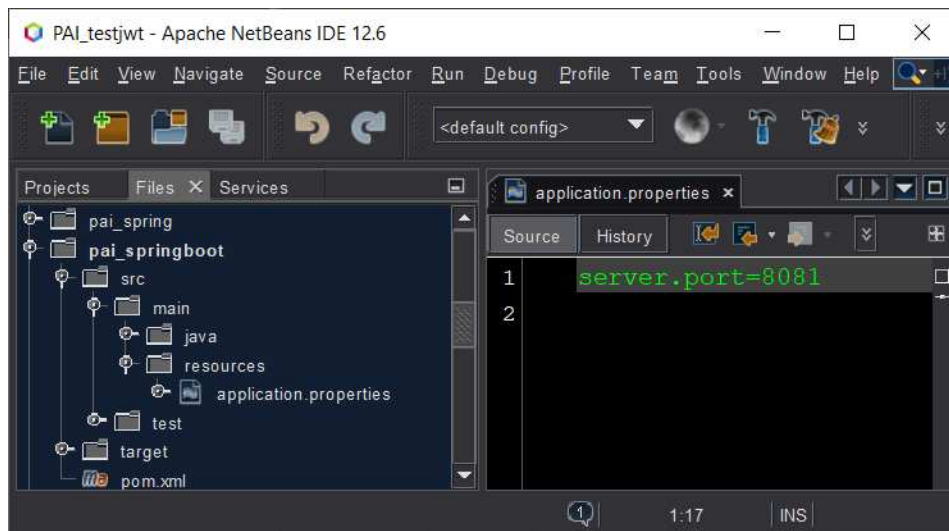
Uruchom projekt, co spowoduje także próbę uruchomienia serwera *TomcatEmbeddedServletContainer on port(s): 8080 (http)*. *Spring Boot* uruchamia serwer *Tomcat (Embedded Tomcat)* za nas i wdraża tam aplikację. Dzięki temu nie trzeba już korzystać z serwera zewnętrznego (jak do tej pory), ponieważ konfiguracja startera *Spring Boot* dostarcza wbudowany serwer *Tomcat*, na którym aplikacja jest wdrażana i uruchamiana. Jeśli próba uruchomienia skończy się komunikatem o błędzie (konflikt z portami): „*Identify and stop the process that's listening on port 8080 or configure this application to listen on another port.*”, to należy skonfigurować serwer tak, aby nasłuchiwał na innym porcie niż domyślny 8080.

SpringBoot korzysta z dwóch podstawowych plików konfiguracyjnych:

- **pom.xml** – znany już, podstawowy plik konfiguracyjny dla projektu *Maven*,
- **application.properties** – plik konfiguracyjny do wprowadzenia dodatkowych ustawień.

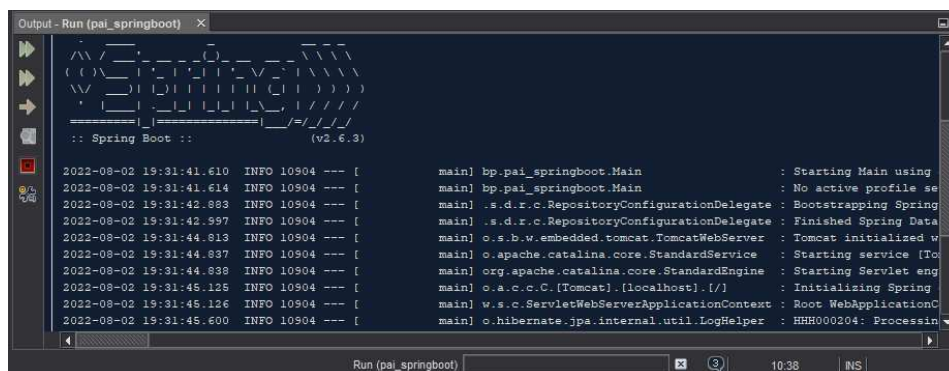
Utwórz plik konfiguracyjny **application.properties** (nowy plik z kategorii *Other* i *Empty file*), w którym będą dodawane kolejne elementy związane z ustawieniami aplikacji. Taki plik najlepiej jest umieścić w katalogu **resources** (Rys. 5.1 – widok w zakładce **Files**). Jeśli katalog **resources** nie istnieje, to go utwórz. W **application.properties** dodaj jedno polecenie:

```
server.port=8081
```



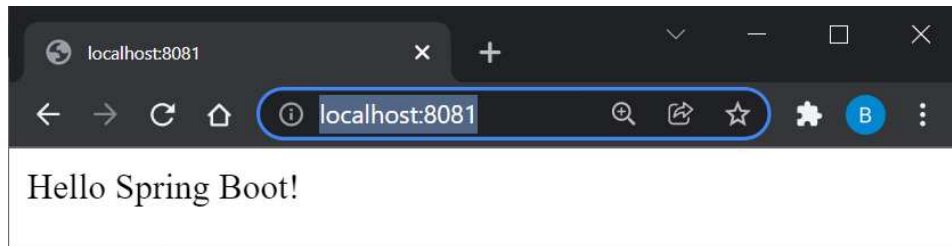
Rys. 5.1. Lokalizacja pliku **application.properties**

Po prawidłowym zbudowaniu i uruchomieniu aplikacji, wynik w oknie **Output** powinien być podobny do tego z rysunku 5.2.



Rys. 5.2. Komunikaty dla prawidłowo uruchomionego projektu **Spring Boot**

Po wpisaniu w oknie przeglądarki adresu URL: <http://localhost:8081/> widok strony w przeglądarce powinien być postaci jak na rysunku 5.3.



Rys. 5.3. Efekt działania pierwszej aplikacji *Spring Boot*

5.1.1. Oddzielenie kontrolera

W przypadku bardzo prostej aplikacji takie podejście z klasą *Main* i dodatkowymi metodami, obsługującymi żądania *HTTP*, sprawdza się, jednak pomieszczone są tutaj różne warstwy aplikacji. Aby kod był czytelniejszy, należy wydzielić metody do obsługi żądań *HTTP* do oddzielnej klasy kontrolera (jak w poprzednim laboratorium).

W pakiecie projektu utwórz pakiet *controllers* na kontrolery i zdefiniuj tam klasę *PageController*, która powinna zawierać część kodu z klasy *Main* i dodatkową metodę do obsługi adresu, np.: */hello* (Przykład 5.3).

Przykład 5.3. Klasa *PageController* w pakiecie *controllers*

```
package bp.pai_springboot.controllers;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class PageController {

    @RequestMapping("/")
    @ResponseBody
    public String mainPage() {
        return "Hello Spring Boot from mainPage() method!";
    }
    @RequestMapping("/hello")
    @ResponseBody
    public String pageTwo() {
        return "Hello Spring Boot from pageTwo() method!";
    }
}
```

W pliku z klasą **Main** usuń fragment kodu przeniesiony do kontrolera oraz adnotację **@Controller** i zbędne deklaracje importu, a z kolei dodaj adnotację **@ComponentScan**, która każe aplikacji przeszukiwać inne klasy w poszukiwaniu adnotacji, takich jak np. **@Controller**. Jeśli klasy są w innych pakietach, należy je wskazać w atrybucie adnotacji, przykładowo:

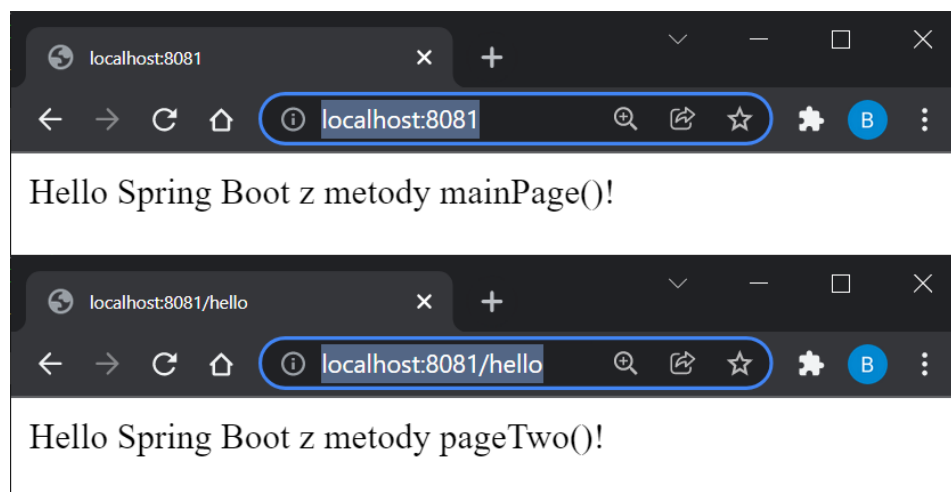
```
@ComponentScan({"bp.pai_springboot.controllers",  
                "bp.pai_springboot.innypakiet"})
```

Klasę **Main** po zmianach przedstawia przykład 5.4.

Przykład 5.4. Klasa Main po modyfikacji

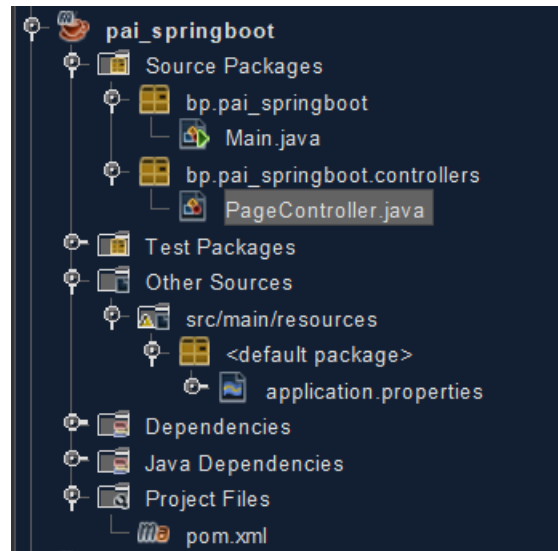
```
package bp.pai_springboot;  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;  
import org.springframework.context.annotation.ComponentScan;  
  
@EnableAutoConfiguration  
@ComponentScan  
public class Main {  
    public static void main(String[] args) {  
        SpringApplication.run(Main.class, args);  
    }  
}
```

Po zatrzymaniu serwera *Tomcat* (*Run* → *Stop Build/Run*) i ponownym uruchomieniu aplikacji, w przeglądarce można podejrzeć wynik działania dwóch akcji, które zostały zdefiniowane w kontrolerze (Rys. 5.4).



Rys. 5.4. Wynik działania dwóch metod kontrolera

Strukturę plików projektu pokazano na rysunku 5.5.



Rys. 5.5. Końcowa struktura projektu

5.1.2. Eksport aplikacji do pliku *war* lub *jar*

Aplikacja może być wyeksportowana do pliku *war* tak, aby można było ją uruchomić na zewnętrznym serwerze *Tomcat*. Jednak ciekawą opcją jest utworzenie tak zwanego *fatJar*, czyli pliku *jar* z wszystkimi zależnościami i kontenerem *Tomcat*, który można uruchomić przy pomocy polecenia:

```
java -jar plik.jar
```

W tym celu dodaj (po sekcji zależności `<dependencies>`) do pliku *pom.xml* wpis:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <executions>
        <execution>
          <goals>
            <goal>repackage</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```


Uruchomienie projektu spowoduje teraz utworzenie odpowiedniego pliku *jar* w katalogu *target*. Więcej na ten temat można znaleźć na stronie: <http://docs.spring.io/spring-boot/docs/current/reference/html/build-tool-plugins-maven-plugin.html>.

Zadanie 5.2. Spring Boot i Spring Data JPA

Spring Data to jeden z kluczowych dodatków do *Spring*, który umożliwia integrację aplikacji z bazą danych. Domyślna implementacja bazuje na *Hibernate*.

W tym zadaniu zostanie utworzona aplikacja typu *CRUD*, współpracująca z bazą danych *H2* za pomocą *Spring Data JPA*. Potrzebne będą:

- dodatkowe zależności,
- klasa encji, czyli klasa *POJO* z odpowiednimi adnotacjami, która zostanie wykorzystana do utworzenia tabeli w bazie danych,
- konfiguracja bazy danych,
- repozytorium, czyli interfejs z definicjami operacji, które można wykonać na klasie encji (dodawanie, pobieranie, usuwanie, modyfikowanie, wyszukiwanie).

5.2.1. Dodatkowe zależności

Spring Data JPA łączy się z wieloma różnymi bazami danych (typu *embedded* jak *H2*, relacyjnymi jak *MySQL*, *Oracle* itp.). Najszybszym jednak sposobem na sprawdzenie działania kodu w akcji jest użycie bazy *H2* [1] w trybie *memory*. Jest ona uruchamiana wraz z aplikacją i przetrzymywana w całości w pamięci (domyślnie), więc po zrestartowaniu aplikacji – baza zostanie usunięta. Baza ta może być również zapisywana do pliku. Aby dodać bazę *H2* do projektu, wystarczy dodać następującą zależność do pliku *pom.xml*:

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
</dependency>
```

Dodatkowo *Spring* musi wiedzieć, że aplikacja używa interfejsu *JPA*. W tym celu należy dodać do pliku *pom.xml* kolejną zależność:

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

UWAGA! Zawsze pamiętaj o przebudowaniu projektu po dodaniu dodatkowych zależności do pliku *pom.xml* lub dokonaniu innych modyfikacji.

5.2.2. Klasa encji

Następny krok to utworzenie klasy encji, która będzie odwzorowana na rekord w tabeli bazy danych. Każde pole obiektu klasy encji będzie odwzorowane jako oddzielna kolumna w tabeli w bazie. W klasie encji można także opisać relacje pomiędzy obiektami, które zostaną odwzorowane w bazie danych za pomocą kluczy obcych. Instancja obiektu jest odwzorowywana w bazie danych jako pojedynczy wiersz (rekord) w odpowiedniej tabeli bazy danych.

Utworzenie klas encji spowoduje, że *Spring* sam (za pośrednictwem *Hibernate*) przy uruchomieniu aplikacji, połączy się ze wskazaną w konfiguracji bazą danych (domyślnie *H2* w trybie memory), sprawdzi strukturę bazy i dokona potrzebnych modyfikacji (utworzy nieistniejące tabele, doda pola do tabel itp.).

W głównym folderze projektu utwórz kolejny pakiet *entities* (lokalizacja jak na rysunku 5.8), a w nim klasę encji *Zadanie*. Klasa posiada wszystkie pola prywatne: *nazwa*, *opis*, *budżet* i *czy zostało wykonane* (Przykład 5.5).

UWAGA! Importy klas wskazanych jako adnotacje *@Entity*, *@Column*, *@Id*, *@GeneratedValue* powinny pochodzić z pakietu *javax.persistence*.

Przykład 5.5. Klasa encji *Zadanie*

```
package bp.pai_springboot.entities;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Lob;
```

```
@Entity
public class Zadanie{
    @GeneratedValue
    @Id
    private Long id;

    @Column
    private String nazwa;

    @Column
    @Lob
    private String opis;
```

```
@Column
private Double koszt;

@Column
private Boolean wykonane=false;

public Zadanie() {
    this.koszt = 2000.0;
    this.nazwa="Zadanie";
    this.opis="Zadanie do wykonania";
}

//nadpisana metoda toString
@Override
public String toString() {
    return "Encja Zadanie{ id=" + id + ", " + nazwa + ", " +
        opis + ", koszt=" + koszt + ", wykonane=" + wykonane +
        "}";
}
//dodaj metody get i set
}
```

Do klasy *Zadanie* dodaj jeszcze metody *get* i *set*.

W klasie encji *Zadanie*:

- **@Entity** – adnotacja informuje, że obiekt klasy będzie odwzorowany na rekord tabeli w bazie danych.
- **@Id** – definiuje pole, które będzie odwzorowane na klucz główny (unikatowy identyfikator rekordu) w tabeli, w przykładzie występuje z adnotacją **@GeneratedValue**, co oznacza, że wartość ta powinna zostać wygenerowana automatycznie.
- **@Column** – informuje, że pole jest kolumną tabeli. Adnotację tę można pominąć, jeśli nazwa pola klasy i kolumny w bazie danych są takie same. Jeśli są różne, to dodaje się atrybut **name** jako parametr adnotacji.
- **@Lob** – informuje, że w tym polu będą przechowywane duże obiekty (*Large Object*). Na przykład, pole typu *String* z samą adnotacją **@Column** w *SQL* zostanie utworzone domyślnie jako *VARCHAR (255)*. Jeżeli dołączona zostanie adnotacja **@Lob** pole to będzie typu *TEXT*.

5.2.3. Konfiguracja bazy danych

Po uruchomieniu aplikacji struktura bazy danych zostanie automatycznie utworzona. Istniejący już plik konfiguracyjny *application.properties* wykorzystaj do konfiguracji pracy z bazą danych, dodając do niego wpisy:

```
spring.jpa.properties.hibernate.hbm2ddl.auto=update
#spring.datasource.url=jdbc:h2:file:./bazaDanych
spring.jpa.show-sql = true
```

Pierwszy wiersz informuje o strategii generowanego schematu bazy danych (schemat bazy danych zostanie utworzony i później aktualizowany automatycznie). Drugi wiersz (z komentarzem #) dotyczy ewentualnej lokalizacji bazy danych *H2* w pliku (domyślnie baza jest w trybie *memory*). *H2* umożliwia tworzenie lokalnych baz danych, a struktura bazy danych może znajdować się w systemie plików. Uruchamia się ona razem z aplikacją. Ostatnie polecenie pozwala zobaczyć na konsoli zapytania *SQL* (budowane przez *Hibernate*), wykorzystane do utworzenia bazy danych, dodania rekordów, pobrania ich z bazy, itp.

Na rysunku 5.7 zaznaczono przykładowe polecenia *SQL* wykorzystane w przykładzie.

5.2.4. Repozytorium *CRUD*

Model definiuje strukturę danych, a repozytoria definiują jakie operacje można wykonać na danych. Podstawowe operacje *CRUD* (ang. *Create, Read, Update, Delete*) udostępnia sam interfejs *JPA*, inne operacje, jak wyszukiwanie po polach, trzeba dodać samodzielnie. Najprostsze repozytoria tworzone są jako **interfejsy**. Cały kod, który potrzebny jest do wykonania akcji, jest generowany przez *Spring*.

W kolejnym pakiecie *repositories* (lokalizacja jak na Rys. 5.8) utwórz interfejs repozytorium (nowy plik z kategorii *interface*) o nazwie **ZadanieRepository**, który udostępni standardowe operacje (np. *CRUD*) wykonywane na klasie encji **Zadanie** (Przykład 5.6). Interfejs **ZadanieRepository** dziedziczy po gotowym repozytorium **CrudRepository**. Dodatkowe operacje na danych udostępnia interfejs **JpaRepository**, który rozszerza możliwości **CrudRepository** (sprawdź jakie).

Przykład 5.6. Interfejs repozytorium dla klasy encji **Zadanie**

```
package bp.pai_springboot.repositories;

import bp.pai_springboot.entities.Zadanie;
import org.springframework.data.repository.CrudRepository;

public interface ZadanieRepository
    extends CrudRepository<Zadanie, Long>{
}
```

UWAGA! Definiując interfejs repozytorium, poza typem encji należy wskazać typ pola, które jest mapowane na klucz podstawowy w bazie danych. W przykładzie wskazano: **<Zadanie, Long>**.

5.2.5. Zastosowanie repozytorium

Ostatnim krokiem jest „wstrzyknięcie” obiektu **repozytorium** do klasy kontrolera i wykorzystanie jego gotowych metod, na przykład:

- `save(Zadanie t)` – zapisuje obiekt **Zadanie t** do bazy danych,
- `Zadanie findOne(Long id)` – znajduje i zwraca obiekt **Zadanie** o podanym **id**,
- `Iterable findAll()` – pobiera wszystkie rekordy z tabeli bazy danych i zwraca je w postaci kolekcji **Iterable**,
- `long count()` – zwraca liczbę rekordów tabeli **zadanie**,
- `void delete(Long id)` – usuwa z obiekt o zadanym **id**,
- `void delete(Zadanie t)` – usuwa wskazany obiekt **Zadanie** z tabeli w bazie danych,
- `deleteAll()` – usuwa wszystkie rekordy z tabeli.

Aby poinformować *Spring*, gdzie znajduje się repozytorium, przed klasą **Main** dodaj adnotację **@EnableJpaRepositories** z parametrem **basePackagesClasses** (Przykład 5.7). Pamiętaj też o dodaniu kolejnego importu dla klasy **EnableJpaRepositories**.

Przykład 5.7. Zastosowanie repozytorium – klasa Main

```
@EnableAutoConfiguration
@ComponentScan({"bp.pai_springboot.controllers"})
@EnableJpaRepositories({"bp.pai_springboot.repositories"})

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

UWAGA! Jeśli pakiety z kontrolerami i interfejsem repozytorium są umieszczone jako bezpośrednie podpakiety głównego pakietu projektu, to nie trzeba dodawać parametrów do odpowiednich adnotacji i można uprościć je do postaci:

```
@ComponentScan
@EnableJpaRepositories
```

Utworzony wcześniej kontroler **PageController** wykorzystaj teraz do pracy z danymi z bazy. Do klasy kontrolera dodaj deklarację obiektu repozytorium, poprzedzoną odpowiednią adnotacją **@Autowired**:

```
@Autowired
public ZadanieRepository rep;
```

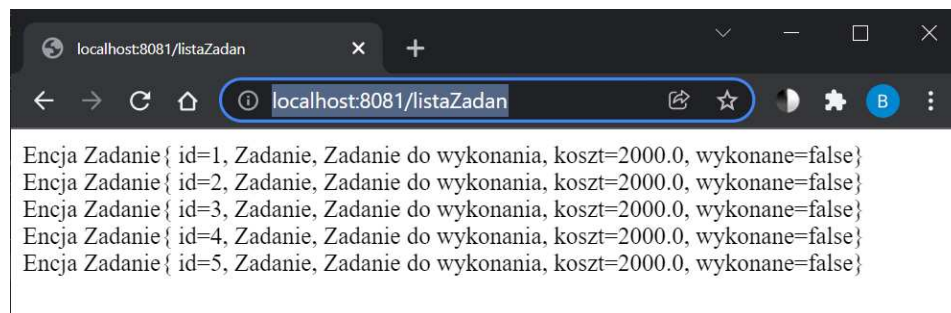
Adnotacja **@Autowired** umożliwia wykorzystanie mechanizmu programowania obiektowego, znanego jako „**wstrzykiwanie zależności**” (ang. *dependency injection – DI*).

Następnie dodaj nową metodę **listaZadan()**, która zdefiniuje i zapisze obiekt klasy **Zadanie** do bazy danych oraz zwróci w odpowiedzi do klienta listę aktualnych rekordów pobranych z bazy danych (Przykład 5.8).

Przykład 5.8. Dodatkowa metoda w kontrolerze PageController

```
@RequestMapping("/listaZadan")
@ResponseBody
public String listaZadan() {
    StringBuilder odp = new StringBuilder();
    Zadanie zadanie = new Zadanie();
    //korzystając z obiektu repozytorium zapisujemy zadanie do bazy
    rep.save(zadanie);
    //korzystając z repozytorium pobieramy wszystkie zadania z bazy
    for(Zadanie i: rep.findAll()) {
        odp.append(i).append("<br>");
    }
    return odp.toString();
}
```

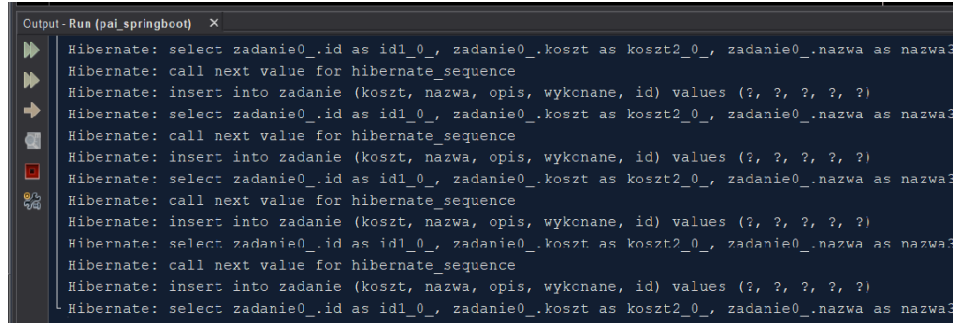
Po uruchomieniu aplikacji i przejściu na stronę **http://localhost:8081/listaZadan** powinno się wyświetlić dodane zadanie. Po kilkukrotnym odświeżeniu strony wynik będzie postaci jak na rysunku 5.6.



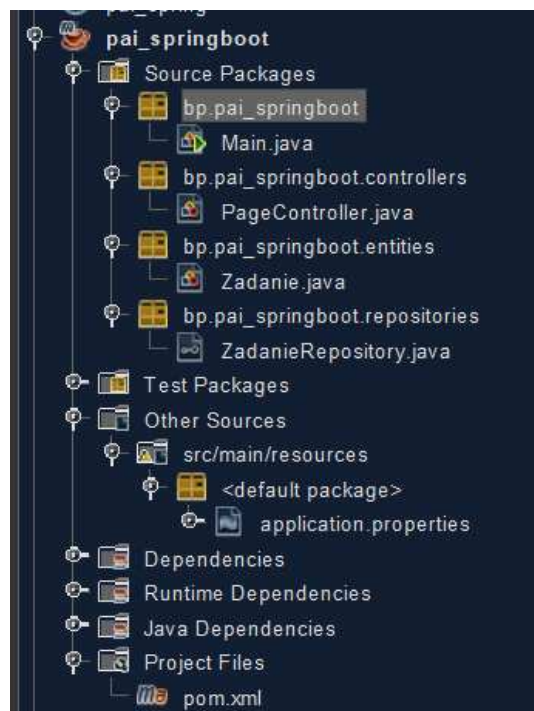
Rys. 5.6. Wynik pracy z bazą H2

W okienku *Output* w *NetBeans* można zaobserwować zapytania do bazy generowane przez *Hibernate* (Rys. 5.7).

Struktura projektu w *NetBeans* pokazana jest na rysunku 5.8.



```
Output - Run (pai_springboot) X
Hibernate: select zadanie0_.id as id1_0_, zadanie0_.koszt as koszt2_0_, zadanie0_.nazwa as nazwa3
Hibernate: call next value for hibernate_sequence
Hibernate: insert into zadanie (koszt, nazwa, opis, wyknane, id) values (?, ?, ?, ?, ?)
Hibernate: select zadanie0_.id as id1_0_, zadanie0_.koszt as koszt2_0_, zadanie0_.nazwa as nazwa3
Hibernate: call next value for hibernate_sequence
Hibernate: insert into zadanie (koszt, nazwa, opis, wyknane, id) values (?, ?, ?, ?, ?)
Hibernate: select zadanie0_.id as id1_0_, zadanie0_.koszt as koszt2_0_, zadanie0_.nazwa as nazwa3
Hibernate: call next value for hibernate_sequence
Hibernate: insert into zadanie (koszt, nazwa, opis, wyknane, id) values (?, ?, ?, ?, ?)
Hibernate: select zadanie0_.id as id1_0_, zadanie0_.koszt as koszt2_0_, zadanie0_.nazwa as nazwa3
Hibernate: call next value for hibernate_sequence
Hibernate: insert into zadanie (koszt, nazwa, opis, wyknane, id) values (?, ?, ?, ?, ?)
Hibernate: select zadanie0_.id as id1_0_, zadanie0_.koszt as koszt2_0_, zadanie0_.nazwa as nazwa3
Hibernate: call next value for hibernate_sequence
Hibernate: insert into zadanie (koszt, nazwa, opis, wyknane, id) values (?, ?, ?, ?, ?)
Hibernate: select zadanie0_.id as id1_0_, zadanie0_.koszt as koszt2_0_, zadanie0_.nazwa as nazwa3
```

Rys. 5.7. Polecenia *Hibernate* widoczne w okienku *Output*

Rys. 5.8. Struktura projektu po wykonaniu zadania 5.2

Zadanie 5.3. MySQL i Spring Boot

W projekcie z zadania 5.2 wykorzystano *Spring Data JPA*, *Hibernate* oraz bazę danych *H2*. Aby zmienić konfigurację do pracy z serwerem *MySQL*:

1. Dodaj do pliku *pom.xml* zależność (i zakomentuj zależność dla *H2*):

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
</dependency>
```

2. Do pliku **application.properties** dodaj konfigurację dla *MySQL* (oraz zakomentuj dla *H2*):
- ```
#Jeśli jest problem z tworzeniem tabeli w mysql - dodaj:
#spring.jpa.hibernate.ddl-auto=update
spring.datasource.url =
jdbc:mysql://localhost:3306/test?serverTimezone=UTC&useUnicode
=yes&characterEncoding=UTF-8
spring.datasource.username = root
spring.datasource.password =
Strategia nazewnictwa dla Hibernate (Naming strategy)
spring.jpa.hibernate.naming-strategy =
org.hibernate.cfg.ImprovedNamingStrategy
Określenie dialektu SQL pozwala Hibernate generować
odpowiednią składnię SQL dla wskazanej bazy danych:
spring.jpa.properties.hibernate.dialect =
org.hibernate.dialect.MySQL5Dialect
```

Reszta plików projektu pozostaje bez zmian. Tabele i kolumny będą tworzone zgodnie z definicjami pól w encjach (klasach z adnotacją *@Entity*) tak samo jak dla bazy *H2*. Po zrestartowaniu aplikacji i jej uruchomieniu z konfiguracją dla *MySQL*, dane o zadaniach będą utrwalone w bazie *test* w tabeli *zadanie*.

**UWAGA!** Pamiętaj o uruchomieniu serwera *MySQL* przed uruchomieniem aplikacji z zadaniami.

### Zadanie 5.4. Metody wyszukiwania w repozytorium

Rozszerz działanie aplikacji o kolejne metody:

- do generowania dodatkowych zadań testowych,
- do usuwania wskazanego zadania,
- do wyszukiwania zadań według zadanego kryterium.

Dzięki dziedziczeniu po interfejsie *CrudRepository* (lub *JpaRepository*) można korzystać z gotowych metod do wykonywania podstawowych operacji typu *CRUD*. Jednak aby wyszukiwać dane, należy do repozytorium dodać dodatkowe metody do filtrowania danych według zadanego kryterium.

- a) Korzystając z istniejącej już metody kontrolera *listaZadan()* (lub nowej pomocniczej metody) wygeneruj więcej zadań, np. za pomocą instrukcji (sprawdź, czy nowe rekordy pojawiają się w bazie danych *test* w tabeli *zadanie*):

```
Zadanie z;
double k=1000;
boolean wyk=false;
for (int i=1;i<=10;i++){
```



```
z = new Zadanie();
z.setNazwa("zadanie "+i);
z.setOpis("Opis czynnosci do wykonania w zadaniu "+i);
z.setKoszt(k);
z.setWykonane(wyk);
wyk=!wyk;
k+=200.50;
rep.save(z);
}
```

- b) Do kontrolera dodaj nową metodę **delete**, do usuwania rekordu w odpowiedzi na zapytanie postaci **http://localhost:8081/delete/5**. Przetestuj działanie tej metody.
- c) Do repozytorium **ZadanieRepository** dodaj kolejne metody do wyszukiwania rekordów w tabeli **zadanie**:
- **findByWykonane(boolean)** – zwraca rekordy z wykonanymi lub niewykonanymi już zadaniami;
  - **findByKosztLessThan(double)** – zwraca rekordy zadań, których koszt jest mniejszy niż zadany;
  - **findByKosztBetween(double, double)** – zwraca rekordy zadań, których koszt należy do wskazanego przedziału wartości.

Do wyszukiwania wykorzystaj *Spring Data JPA* i tzw. **zapytania wbudowane**. Zapytania wbudowane umożliwiają tworzenie zapytań w oparciu o mechanizmy składania zapytań z nazwy akcji (np. **findBy**) oraz części własnej w postaci nazw pól encji. Więcej szczegółów i przykłady zapytań wbudowanych i metod z adnotacją **@Query** można znaleźć na stronie:

<https://www.javappa.com/kurs-spring/spring-data-jpa-zapytania-wbudowane>

Przetestuj działanie tych metod, dodając do kontrolera odpowiednie akcje.

**UWAGA!** W celu przekazania wielu parametrów do akcji kontrolera, należy w **@RequestMapping** podać je w {} w adresie *URL*, np.:

```
@RequestMapping("/koszt/{min}/{max}")
```

### Zadanie 5.5. Wyszukiwanie w bazie *world*

W oparciu o *Spring Boot* utwórz nową aplikację, która będzie korzystała z bazy **world** i tabeli **country**. Zadaniem aplikacji ma być obsługa wyszukiwania:

- krajów z danego kontynentu,
- krajów o liczbie ludności z zadanego przedziału,
- krajów danego kontynentu o powierzchni z zadanego przedziału.

Tak jak w zadaniu poprzednim, akcje kontrolera powinny zwracać typ *String* i być poprzedzone adnotacją *@ResponseBody*. Na potrzeby tego zadania w definicji klasy encji *Country* wystarczy dodać tylko te pola, które są potrzebne w metodach wyszukiwania. Pola powinny być deklarowane odpowiednio jako typ *String*, *Double* (lub *BigDecimal*) i nie trzeba stosować adnotacji *@Column* poprzedzających nazwy pól.

**UWAGA 1.** W pliku *application.properties* należy zmienić domyślną strategię nazewnictwa dla *Hibernate* na (należy wpisać w jednym wierszu!):

```
spring.jpa.hibernate.naming.physical-
strategy=org.hibernate.boot.model.naming.PhysicalNamingStrategyS
tandardImpl
```

Pozwoli to właściwie zmapować nazwę pól składających się z wielu wyrazów (np. *surfaceArea* w klasie encji *Country*) na nazwę kolumny *SurfaceArea* w tabeli. Brak tego ustawienia (ustawienie domyślne) powoduje tworzenie nowej kolumny o nazwie *surface\_area*, co stwarza problemy z działaniem metod wyszukiwania opartych na zapytaniach wbudowanych.

**UWAGA 2.** Jeśli nie uda się skorzystać z zapytania wbudowanego, zawsze można zastosować własne metody z adnotacją *@Query*. Przykłady takich zapytań można znaleźć na wskazanej już wcześniej stronie:

<https://www.javappa.com/kurs-spring/spring-data-jpa-zapytania-wbudowane>

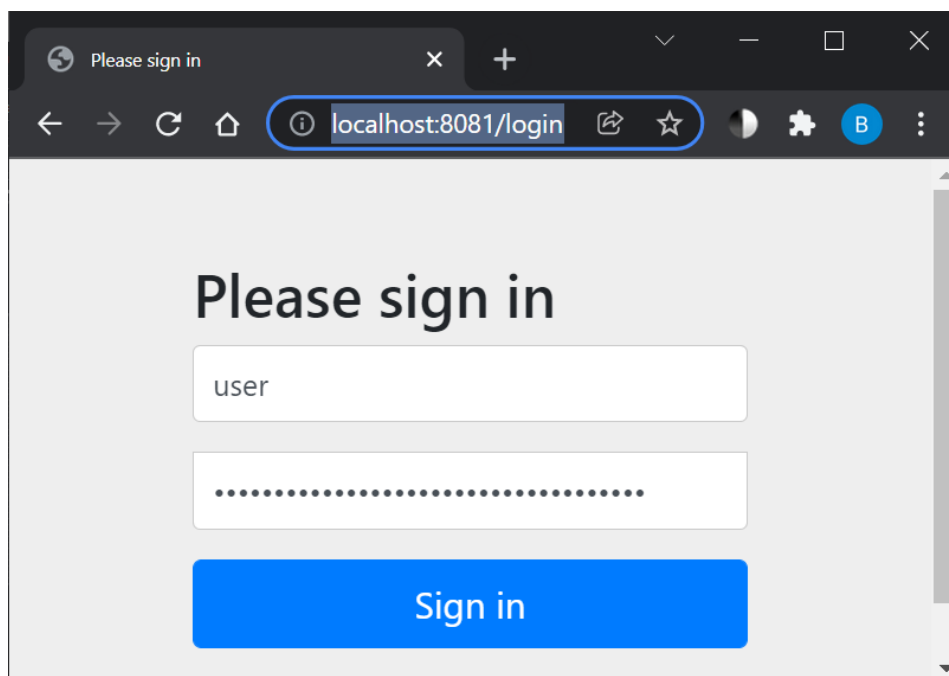
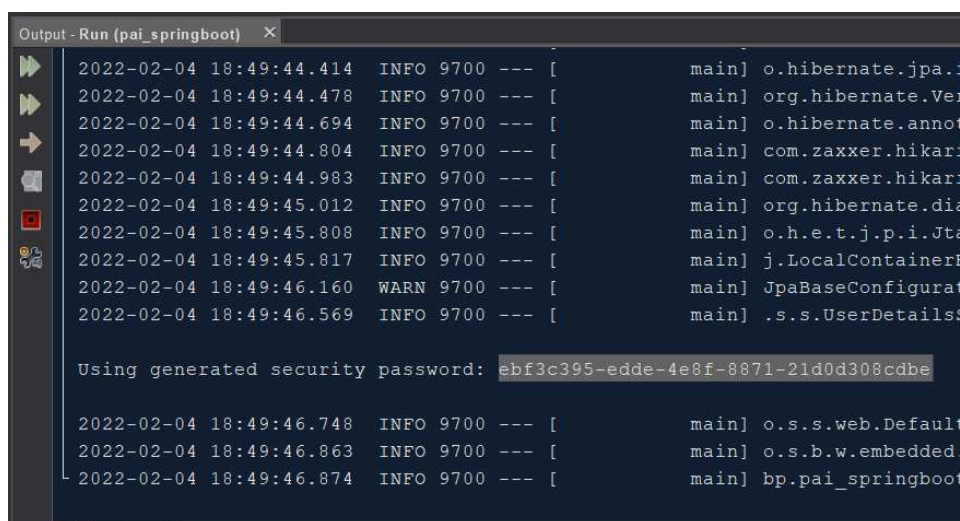
## Zadanie 5.6. Wstęp do *Spring Security*

*Spring* udostępnia gotowy mechanizm autentykacji i autoryzacji użytkowników, z którego można skorzystać, dodając do *pom.xml* zależność do *Spring Security* (Przykład 5.9).

### Przykład 5.9. Zależność dla *Spring Security*

```
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Po przebudowaniu projektu z dodaną zależnością dla *Spring Security* oraz po wejściu na adres strony z listą zadań, nastąpi przekierowanie do strony logowania (Rys. 5.9). Domyślna nazwa użytkownika to *user*, a hasło jest generowane losowo przez serwer i wypisywane na konsolę (Rys. 5.10).

Rys. 5.9. Domyslny formularz logowania dla *Spring Security*Rys. 5.10. Losowo wygenerowane hasło dla użytkownika *user*

Wpisanie poprawnych danych pozwoli ponownie zobaczyć strony. W przypadku podania nieprawidłowych danych system poprosi o nie kolejny raz. Anulowanie spowoduje wyświetlenie domyślnej strony błędu.

Dane logowania można zmienić, dodając do pliku konfiguracyjnego ***application.properties*** kolejne wiersze z ustawieniami, np.:

```
spring.security.user.name=beata
spring.security.user.password=beata
spring.security.user.roles=admin
```

Bardziej zaawansowana konfiguracja uwierzytelniania użytkownika będzie tematem kolejnych laboratoriów.