

## Informatyka, Aplikacje internetowe i mobilne, semestr 5

### Programowanie aplikacji Webowych

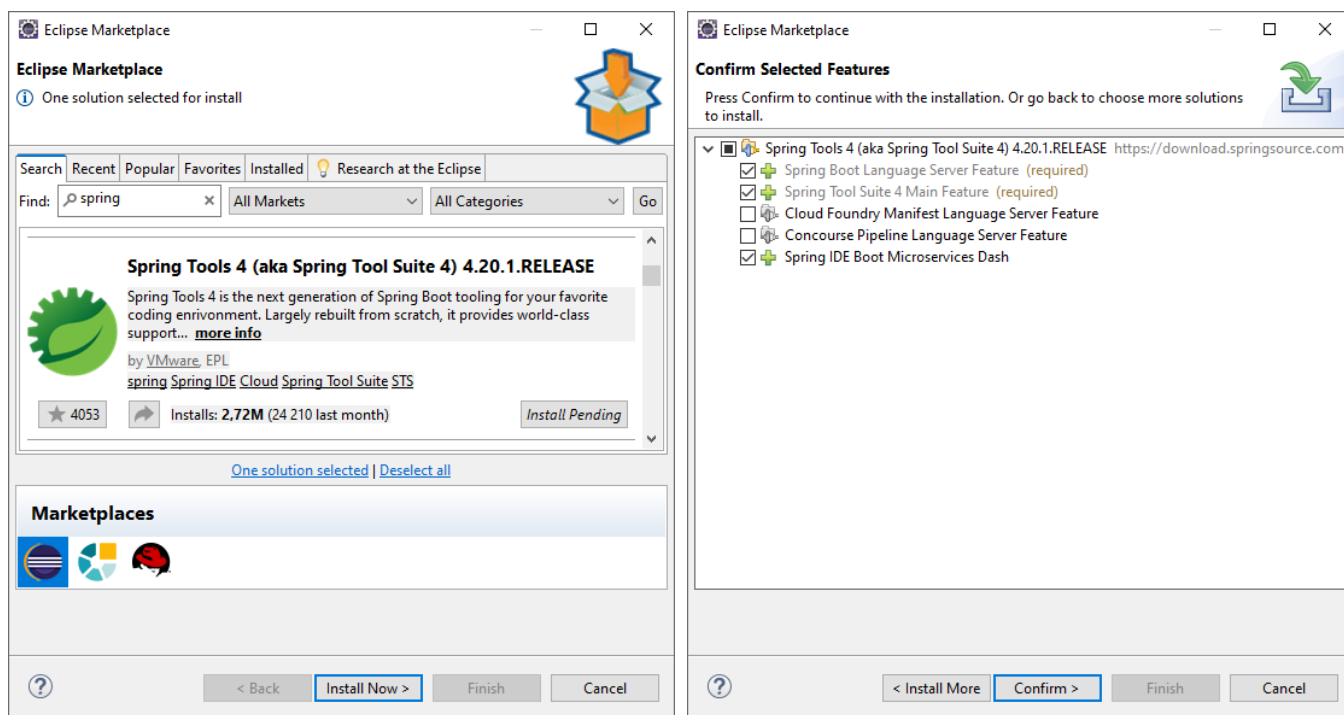
#### Laboratorium nr 6 i 7

#### Tworzenie aplikacji Webowej SpringBooks z wykorzystaniem frameworka Spring Boot

Utwórz aplikację Web SpringBooks umożliwiającą zarządzanie książkami z wykorzystaniem frameworka Spring Boot oraz bazy danych PostgreSQL.

### 1. Eclipse IDE – dodatki

Uzupełnij Eclipse IDE o pakiet Spring Tools 4. Pakiet Spring Tools 4 ułatwia pracę z frameworkiem Spring Boot w Eclipse dostarczając gotowe kreatory i szablony. Wybierz z menu Help | Eclipse Marketplace | Spring Tools 4 | Install. Ustawienia elementów można pozostawić bez zmian, wymagane jest zaakceptowanie licencji. W kolejnych oknach trzeba potwierdzić zaufanie do produktów Oracle Corporation i Spring Builds. Natomiast po zakończeniu instalacji trzeba zrestartować Eclipse IDE.



### 2. Projekt aplikacji SpringBooks

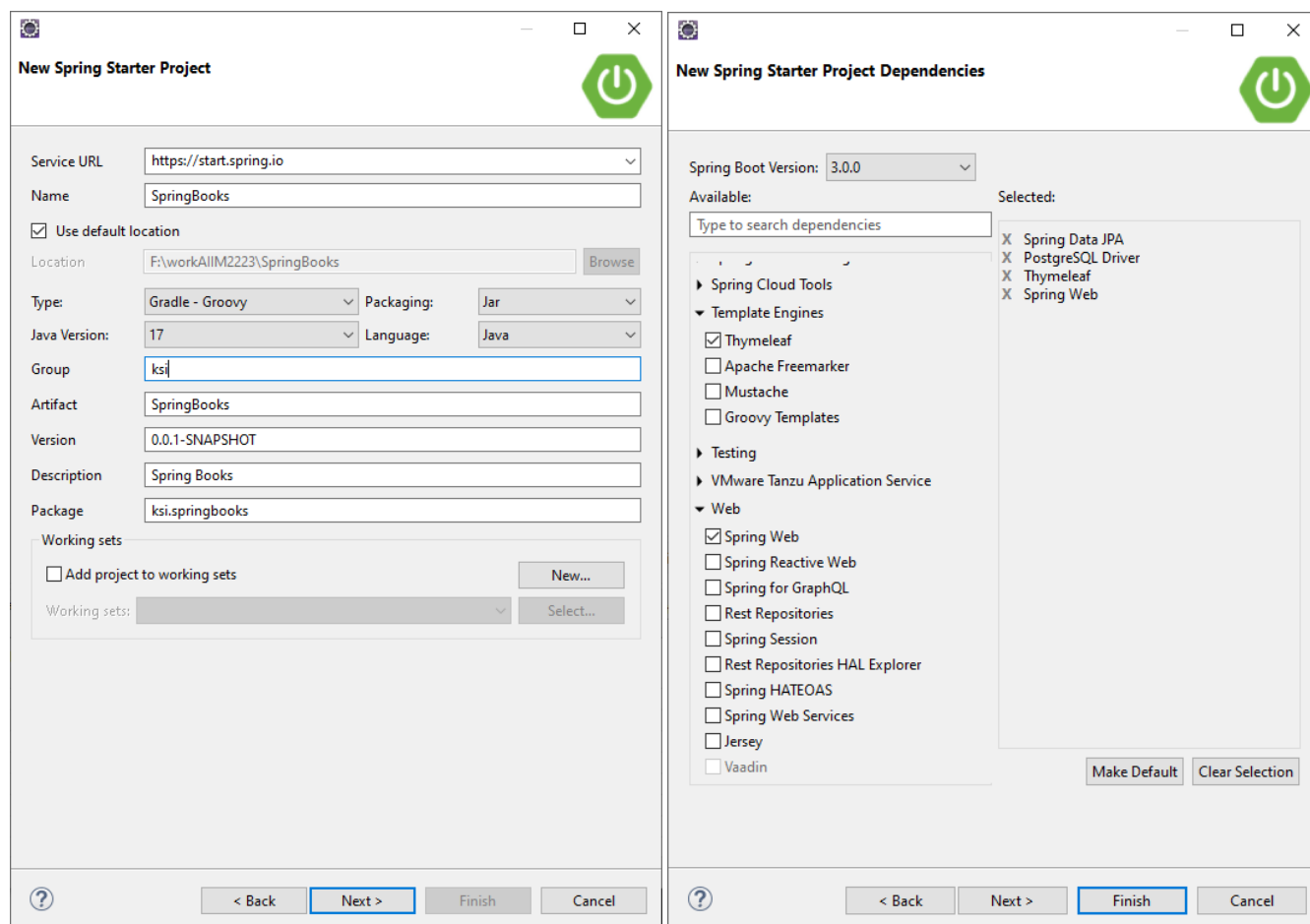
Utwórz projekt Eclipse dla aplikacji o nazwie SpringBooks korzystającej z frameworka Spring Boot wybierając z menu File | New | Other | Spring Boot | Spring Starter Project. Ustawienia kreatora opisano i przedstawiono na obrazkach poniżej.

W pierwszym oknie New Spring Starter Project ustaw Name i Artifact jako SpringBooks, typ projektu – Gradle - Groovy, Java Version – 17, Packaging – War, Group – ksi, Description – Spring Books, Package – ksi.springbooks.

Eclipse umożliwia obecnie zbudowanie projektu na podstawie Maven albo Gradle w języku Kotlin lub Groovy. Domyślnie ustawiony jest ten ostatni. Pozostawimy to ustawienie bez zmian. Jeśli biblioteki ściągane są po raz pierwszy zajmuje to trochę czasu.

W kolejnym oknie New Spring Starter Project Dependencies pojawiają się wersje Spring Boot, wybierz ostatnią stabilną, np. 3.1.5. Samodzielnie trzeba wybrać niezbędne zależności. W naszym przypadku, jak widać na obrazku, będzie to:

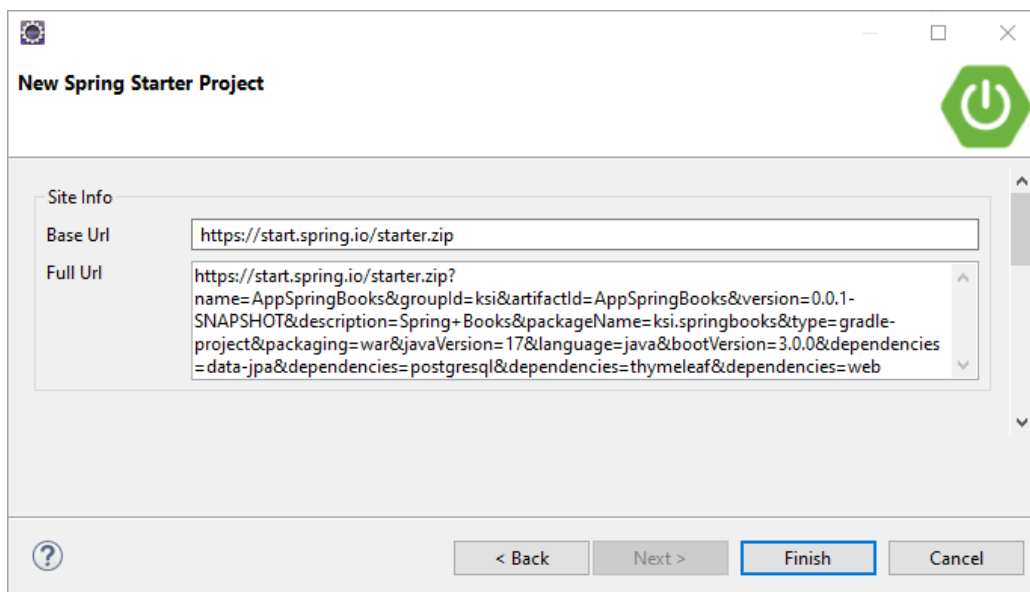
Web | Spring Web  
SQL | PostgreSQL Driver  
SQL | Spring Data JPA  
Template Engines | Thymeleaf



The first screenshot shows the 'New Spring Starter Project' dialog. The 'Name' field is set to 'SpringBooks' and the 'Artifact' field is also set to 'SpringBooks'. The 'Group' field is set to 'ksi' and the 'Package' field is set to 'ksi.springbooks'. The 'Type' is set to 'Gradle - Groovy' and the 'Java Version' is set to '17'. The 'Description' is set to 'Spring Books'.

The second screenshot shows the 'New Spring Starter Project Dependencies' dialog. The 'Spring Boot Version' is set to '3.0.0'. The 'Available' list includes 'Spring Cloud Tools', 'Template Engines', 'Testing', 'VMware Tanzu Application Service', and 'Web'. The 'Selected' list includes 'Spring Data JPA', 'PostgreSQL Driver', 'Thymeleaf', and 'Spring Web'.

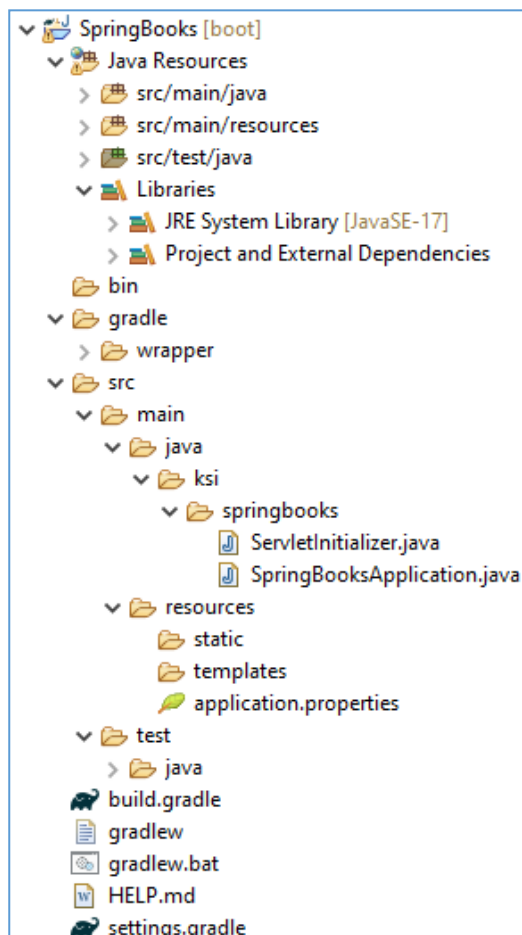
W ostatnim oknie kreatora New Spring Starter Project sprawdź zawartość pola Full URL. Kreator aplikacji Spring Boot w celu wygenerowania szkieletu aplikacji korzysta z usługi serwisu start.spring.io którą wywołuje z ustawionymi przez nas parametrami.



Zakończ korzystnie z kreatora. Wygenerowanie struktury projektu wymaga ściągnięcia dodatkowych bibliotek i zabiera trochę czasu. W efekcie otrzymujemy strukturę zgodną ze strukturą projektu Gradle przedstawioną na obrazku.

Sprawdź zawartość pliku `SpringBooksApplication.java`. Została tam wygenerowana klasa reprezentująca całą aplikację z główną metodą uruchomieniową `main` oraz dołączonymi pakietami konfiguracyjnymi. Jest tylko jedna taka klasa w aplikacji i musi być opisana adnotacją `@SpringBootApplication`.

Sprawdź zawartość pliku **build.gradle**. Znajdują się tam ustawienia zgodne z projektem Gradle oraz dołączone zależności.



W kodzie **build.gradle** sekcja zależności (dependencies) ma postać:

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
    implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
    implementation 'org.springframework.boot:spring-boot-starter-web'
    runtimeOnly 'org.postgresql:postgresql'
```

```
providedRuntime 'org.springframework.boot:spring-boot-starter-tomcat'  
testImplementation 'org.springframework.boot:spring-boot-starter-test'  
}
```

Projekt Spring Boot jest budowany w oparciu o jeden servlet zarządzający tzw. dyspozytor. Dyspozytor (DispatcherServlet) udostępnia wspólny algorytm przetwarzania żądań i organizuje współpracę z komponentami, które wykonują pozostałe prace. Pakiet spring-boot-starter-web dostarcza automatyczną deklarację, mapowanie i konfigurację servletu dyspozytora. W aplikacjach Spring Boot nie trzeba tworzyć pliku web.xml (poza nielicznymi wyjątkami). Servlet jest rejestrowany programowo za pomocą klasy ServletInitializer dziedziczącej z SpringBootServletInitializer.

Pakiet spring-boot-starter-data-jpa wspomaga współpracę z relacyjnymi bazami danych korzystając z interfejsów Spring Data JPA i JPA oraz systemu ORM Hibernate. Integracja Hibernate ze Spring jest jednym z częściej spotykanych rozwiązań w aplikacjach Web.

Pakiet spring-boot-starter-thymeleaf to implementacja silnika szablonów Thymeleaf. Thymeleaf umożliwia tworzenie elementów interfejsu użytkownika po stronie serwera czyli frontendu po stronie backendu. Zawiera dwa tzw. dialekty: Standard i SpringStandard. SpringStandard umożliwia integrację ze Spring MVC, poza tym dialekty są identyczne. Thymeleaf zawiera zestaw zdefiniowanych elementów XML / HTML / XHTML, umożliwia również tworzenie własnych elementów użytkownika.

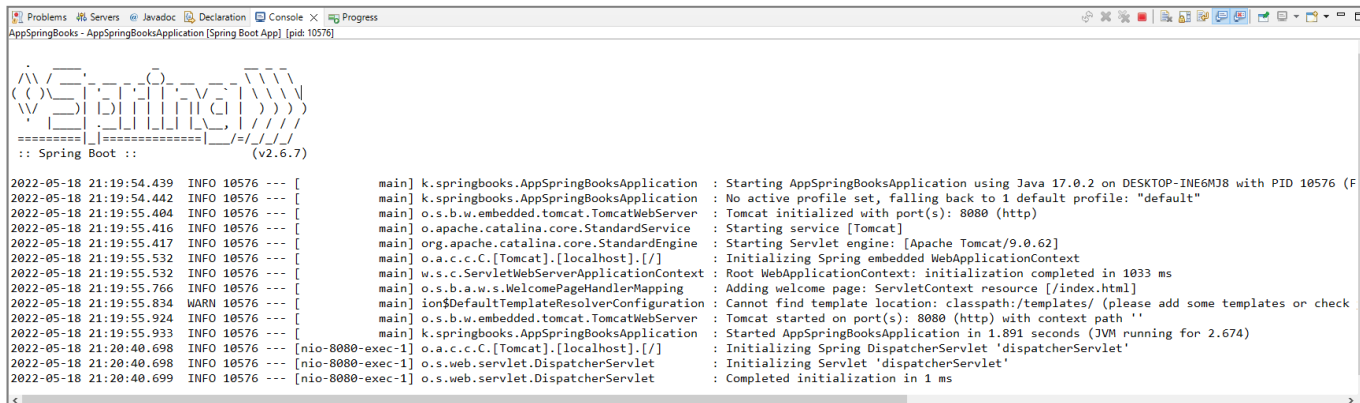
### 3. Sprawdzenie działania projektu aplikacji Spring Boot

Przeprowadź proste sprawdzenie możliwości uruchomienia aplikacji generując plik **index.html** z tytułem i nagłówkiem: Spring Books Application. W tym celu wygeneruj statyczny plik HTML 5 o nazwie **index.html** w folderze **src/main/resources/static**. Ustaw tytuł i nagłówek h1 z napisem Spring Books Application.

Wybierz projekt, a następnie z podręcznego menu Run As | Spring Boot App. Podczas uruchamiania aplikacji, w konsoli Eclipse pojawi się logo ASCII Spring Boot oraz seria komunikatów mówiąca o stanie uruchomienia aplikacji. Eclipse zgłosi błąd ponieważ dołączyliśmy do projektu sterownik bazy danych, natomiast nie mamy jeszcze skonfigurowanego połączenia. Żeby pominąć łączenie z bazą danych w pliku aplikacji dołącz do adnotacji `@SpringBootApplication` parametr wyłączający użycie konfiguracji dla danych. Powinna ona mieć postać:

```
@SpringBootApplication(exclude = {DataSourceAutoConfiguration.class})
```

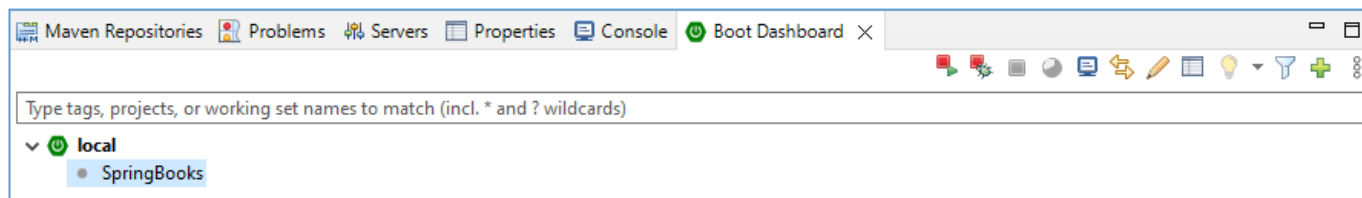
Ponownie uruchom aplikację. Zauważ, że aplikacja uruchomiona została za pomocą Tomcata na porcie 8080 (nie jest to tożsame z tomcatem na zakładce servers). Wywołaj w oddzielnej przeglądarce adres <http://localhost:8080>. Powinna zostać wyświetlona zawartość domyślnej statycznej strony **index.html**.



Zauważ, że aplikacja została uruchomiona z wykorzystaniem innego serwera Tomcat (dostarczonego razem ze Spring Boot), a Eclipse nie otworzył automatycznie okna przeglądarki. Żeby zobaczyć działanie aplikacji wpisz w przeglądarce localhost:8080. Nie można zarządzać wprost instancją uruchomionego Tomcata wraz z naszą aplikacją, gdyż na zakładce Servers status Tomcata jest widoczny jako Stopped. Serwer i aplikację można zatrzymać w konsoli lub na pasku narzędziowym wybierając Stop.



Sterowanie aplikacją wspomaga okno Boot Dashboard, które zostało dodane razem ze Spring Tools 4. Można je włączyć, korzystając z pokazanej na obrazku ikonki. Zakładka pojawi się w dolnej części okna Eclipse. Udostępnia m.in. listę aplikacji Spring oraz ikonki do uruchomienia projektu i przeglądarki z aplikacją.



## 4. Połączenie z bazą danych PostgreSQL

Zdefiniuj połączenie z bazą danych PostgreSQL w pliku application.properties.

Plik konfiguracyjny aplikacji jest obecnie pusty. Dodaj do tego pliku następujące ustawienia:

```
# parametry poziomu logowania/raportowania aplikacji
logging.level.org.springframework=ERROR
spring.sql.init.mode=always

# parametry dla PostgreSQL
spring.sql.init.platform=postgres
spring.jpa.properties.hibernate.default_schema=books
spring.datasource.url=jdbc:postgresql://localhost:5432/<nazwa_bd>
spring.datasource.username=<nazwa_u>
spring.datasource.password=<haslo_u>

# parametry dla JPA
spring.jpa.hibernate.ddl-auto=update
```

Parametry zostały podzielone na trzy grupy: poziomu logowania/raportowania aplikacji, dla PostgreSQL i dla JPA. W pierwszej grupie ustawiony został poziom raportowania błędów (ERROR) oraz informacja, że Spring Boot ma zawsze automatycznie tworzyć schemat osadzonego źródła danych (always). W drugiej grupie PostgreSQL znajdują się informacje na temat dostępu do bazy danych. Natomiast w trzeciej JPA, ustawiona została automatyczna aktualizacja schematu w bazie danych oraz informacja na temat używanej platformy bazy danych

Uzupełnij dane dotyczące Twojej bazy danych.

Usuń informację o wykluczeniu używania źródła danych z adnotacji `@SpringBootApplication` w pliku uruchomieniowym `SpringBooksApplication.java`

Uruchom aplikację Spring Books, żeby sprawdzić, czy w konfiguracji `application.properties` nie ma błędów. Jeśli błędów nie ma, aplikacja uruchomi się poprawnie. Nadal będzie dostęp tylko do statycznej strony dla adresu: `http://localhost:8080`, ale dodatkowo przy uruchomieniu sprawdzone zostanie połączenie z systemem bazy danych PostgreSQL.

Zatrzymaj aplikację.

## 5. Baza danych w obiektach Java

Utworzymy klasy Java reprezentujące tabele w bazie danych. Będą miały nazwy: `Book`, `Publisher` i `Category`. Żeby klasa reprezentowała tabelę musi zostać zaopatrzona w adnotację `@Entity`.

Dodaj do projektu pakiet `ksi.springbooks.models` i utwórz w nim klasę `Book`. Dodaj adnotację `@Entity` oraz pola zgodnie z kodem poniżej. Dodatkowe dwie adnotacje `@Id` oraz `@GeneratedValue` są potrzebne dla pola `idb`, które zawiera identyfikator i jest kluczem głównym tabeli. Adnotacja `@Id` deklaruje klucz podstawowy nie generując wartości. Adnotacja `@GeneratedValue` określa strategię generowania klucza czyli wartości pola `id`.

Specyfikacja JPA obsługuje 4 różne strategie generowania klucza podstawowego (**GenerationType.\*: Auto, Identity, Sequence, Table**), które programowo generują wartości klucza podstawowego lub wykorzystują funkcje bazy danych, takie jak automatycznie zwiększane kolumny lub sekwencje. Wykorzystany tu parametr `GenerationType.IDENTITY` opiera się na generowaniu po stronie bazy danych nowej wartości przy każdej operacji wstawiania nowego rekordu. Z punktu widzenia bazy danych jest to bardzo wydajne, ponieważ kolumny autoinkrementacji są wysoce zoptymalizowane i nie wymagają żadnych dodatkowych instrukcji.

Dodanie kodu wymaga dodatkowo importu odpowiednich bibliotek.

```
@Entity
public class Book {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long idb;
    private String title;
}
```

Użycie powyższych adnotacji umożliwia standard JPA ORM, z którego korzysta Spring. Znacznie ułatwia to obsługę bazy danych z poziomu aplikacji.

Wygeneruj dla klasy `Book` konstruktor z nadklasy oraz zestawy metod `get` i `set` dla wszystkich pól.

Połącz się z bazą danych, którą podałeś w konfiguracji połączenia w pliku `application.properties` i załóż tam schemat `books`. Następnie uruchom aplikację, W bazie powinna pojawić się tabela `book` zgodna z definicją w modelu `Book`.

Skonstruuj model encji `Publisher` dla tabeli `publisher` z polami `idp` (klucz główny generowany automatycznie w bazie danych) oraz `name` i `address`. Tabela ta łączy się z tabelą `book` w relacji jeden do wielu. Zdefiniowanie w modelach relacji wymaga dodania do pól kolejnych adnotacji, tutaj są to `@OneToMany` oraz `@ManyToOne`.

Ponieważ książka może mieć tylko jednego wydawcę, dodaj do modelu `Book`, pole `publisher` zgodnie z następującym schematem.

```
@ManyToOne
private Publisher publisher;
```

Jednocześnie w modelu `Publisher` trzeba zadeklarować, że może on być w relacji z wieloma książkami, będzie je reprezentować lista klasy `List<Book>`. Dodaj do modelu `Publisher` pole `books` z adnotacją definiującą relację:

```
@OneToMany
@JoinColumn(name="publisher_idp")
private List<Book> books;
```

Analogicznie zdefiniuj model encji `Category` z polami `idc` (klucz główny generowany automatycznie w bazie danych) oraz `description`. Połącz tabelę odpowiednimi adnotacjami relacji.

Uruchom aplikację i sprawdź czy w bazie danych pojawiły się odpowiednie tabele.

Żeby dodać do nich dane skopiuj dołączony do materiałów plik z poleceniami dodania danych testowych **data.sql**. Umieść plik w katalogu `resources` i ponownie uruchom aplikację.

Po wygenerowaniu danych zmień nazwę pliku **data.sql** albo go usuń.

## 6. Definiowanie struktury aplikacji

Struktura aplikacji Web składa się z łańcucha powiązanych obiektów/klas. Podstawowe elementy takiego łańcucha to kontroler (`controller`), usługa (`service`) i interfejs do repozytorium (`repository`). Na przykład dla części aplikacji obsługującej książki będzie to:

`BookController` – klasa z adnotacją `@Controller`

`BookService` – klasa z adnotacją `@Service`, która zostanie wstrzyknięta do kontrolera

`BookRepository` – interfejs rozszerzający `JpaRepository` z adnotacją `@Repository`, który zostanie wstrzyknięty do `BookService`.

Działanie aplikacji polega na tym, że odpowiednia metoda kontrolera (wybrana na podstawie zdefiniowanego mapowania) przejmie żądanie. Następnie uruchamia ona metodę serwisu odpowiedzialną za przetworzenie danych. Ta metoda wywołuje następnie odpowiednią metodę repozytorium pobierającą, bądź zapisującą dane w bazie.

### Repozytoria



W folderze `springbooks` utwórz nowy podkatalog `repositories`. Wygeneruj w nim interfejs dla książki o nazwie `BookRepository` rozszerzający interfejs `JpaRepository<Book, Long>`. Dodaj adnotację `@Repository`. Kod interfejsu powinien mieć postać:

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import ksi.springbooks.models.Book;

@Repository
public interface BookRepository extends JpaRepository<Book, Long> {

}
```

Korzystając z `BookRepository` jako rozszerzenia `JpaRepository` aplikacja będzie miała dostęp do metod umożliwiających pracę z relacją `book` w bazie danych, np.: `findAll()`, `findById(id)`, `save()`, `deleteById(id)`. Można tu również deklarować własne metody.

Analogicznie wygeneruj interfejsy repozytoriów dla dwóch pozostałych klas.

### Usługi - serwisy

W katalogu `springbooks` utwórz nowy podkatalog `services`. Wygeneruj w nim klasę `BookService` z pustym konstruktorem. Zgodnie z opisaną powyżej strukturą klasa ta będzie potrzebowała adnotacji `@Service` oraz wstrzyknięcia interfejsu `BookRepository`. Wstrzyknięcia można dokonać definiując pole z adnotacją `@Autowired`. Po dodaniu odpowiednich adnotacji i bibliotek, kod klasy powinien mieć postać:

```
package ksi.springbooks.services;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import ksi.springbooks.repositories.BookRepository;

@Service
public class BookService {
    @Autowired
    private BookRepository repository;

    public BookService() {
        super();
    }
}
```

Do klasy dodaj metody do obsługi tabeli `books`. Metody te można wygenerować korzystając z menu podręcznego i wybierając `Source | Generate Delegate Methods`. Zaznacz metody `findAll()`, `save()`, `findById()` oraz `deleteById()` i uruchom generowanie.

Analogicznie wygeneruj klasy usług dla pozostałych dwóch modeli `Publisher` i `Category`.

### Kontrolery



W katalogu `springbooks` utwórz podkatalog `controllers`. Wygeneruj w nim klasę o nazwie `BookController`. Opisz ją adnotacją `@Controller` i wstrzyknij do niej obiekt klasy `BookService`. Kod powinien mieć następującą postać:

```
package ksi.springbooks.controllers;

import ksi.springbooks.models.Book;
import ksi.springbooks.services.BookService;

@Controller
public class BookController {

    @Autowired
    private BookService service;

    public BookController() {
    }
}
```

W klasie kontrolera zaprojektujemy akcje odzwierciedlające odpowiednie działania aplikacji: wyświetlanie listy książek i wyświetlanie formularza do dodawania książek.

Do kodu kontrolera `BookController` dodaj akcję `viewBooksList`, która utworzy listę wszystkich książek z bazy i przekaże tę listę jako atrybut modelu do widoku/szablonu strony HTML. Żeby wygenerować listę skorzystaj z metody `findAll()` klasy usług `BookService`. Jako parametr metody-akcji ustaw model klasy `Model`. Jako wynik metody ustaw `String` z nazwą widoku w postaci strony HTML. Natomiast, żeby akcja została wykonana na żądanie URL dodaj do niej adnotację `@RequestMapping`. Przykład kodu metody-akcji `viewBooksList` można zobaczyć poniżej.

```
@RequestMapping("books_list")
public String viewBooksList(Model model){
    List<Book> lb=service.findAll();
    model.addAttribute("lb", lb);
    return "books_list";
}
```

Do kontrolera można dodawać różne metody oraz adnotacje. Na przykład metoda do wyświetlania formularza, do dopisywania danych nowych książek lub do zapisywania danych na temat książek w bazie danych.

Dodaj do klasy metodę wyświetlającą formularz do wpisywania nowych książek o nazwie `showFormNewBook`. Dodaj do metody adnotację `@RequestMapping`. Jako parametr metody ustaw model. W kodzie metody wygeneruj nowy obiekt książki i przekaż go jako atrybut do szablonu widoku. W wyniku przekaż łańcuch znaków z nazwą widoku, w którym będzie zdefiniowany formularz. Kod takiej metody może mieć postać:

```
@RequestMapping("/new_book")
public String showFormNewBook(Model model) {
    Book nb = new Book();
    model.addAttribute("book", nb);
    return "new_book";
}
```

```
}
```

Dodaj do klasy kolejną metodę `saveBook`. Dodaj do niej adnotację `@PostMapping`, która definiuje pobranie danych metodą `post`. W parametrze metody umieść drugą adnotację `@ModelAttribute`, wiążącą ten parametr lub wartość zwracaną przez metodę z nazwanym atrybutem modelu, który będzie używany w widoku. W efekcie, podobnie jak poprzednia metoda, przekieruje użytkownika do wybranej strony `.html` aplikacji, tutaj jest to strona `books_list.html`.

```
@PostMapping(value="/save_book")
public String saveBook(@ModelAttribute("book") Book book) {
    service.save(book);
    return "redirect:/books_list";
}
```

## 7. Tworzenie widoków aplikacji korzystając z silnika szablonów Thymeleaf

Thymeleaf umożliwia tworzenie elementów interfejsu użytkownika po stronie serwera (backendu). Korzystanie z szablonów wymaga (oprócz dołączenia bibliotek) wygenerowania pliku `.html` z dołączoną przestrzenią nazw `thymeleaf`:

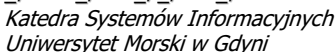
```
xmlns:th="http://www.thymeleaf.org"
```

Jak widać znaczniki Thymeleaf w szablonie będą się rozpoczynały od `th`.

### Widok listy książek

W katalogu `resources/templates` utwórz plik `books_list.html` zgodny z szablonem HTML 5. Do znacznika `html` dodaj podane powyżej deklaracje przestrzeni nazw niezbędnej dla użycia silnika szablonów Thymeleaf. W części `head` wpisz w znaczniku `title` nazwę strony Books list. W części `body` zdefiniuj `div` z następującą zawartością:

```
<div>
  <h2>Books List</h2>
  <a href="/new_book">Create New Book</a>
  <br/><br/>
  <table>
    <thead>
      <tr>
        <th>Book Id</th>
        <th>Title</th>
        <th>Publisher</th>
        <th>Category</th>
      </tr>
    </thead>
    <tbody>
      <tr th:each="book : ${lb}">
        <td th:text="${book.idb}">Book ID</td>
        <td th:text="${book.title}">Title</td>
        <td th:text="${book.publisher.name}">Publisher</td>
        <td th:text="${book.category.description}">Category</td>
      </tr>
    </tbody>
  </table>
</div>
```



```
<a th:href="@{'/edit_book/' + ${book.idb}}">Edit</a>  
    &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&~  
    <a th:href="@{'/delete_book/' + ${book.idb}}">Delete</a>  
  </td>  
</tr>  
</tbody>  
</table>  
</div>
```

Zauważ, że w dokumencie, oprócz `/new_book`, są już akcje `/edit_book`, oraz `/delete_book`, które trzeba będzie dodać w późniejszym etapie do kontrolera `BookController`. Oprócz tego będzie potrzebna wcześniej zdefiniowana akcja `/save_book`, która zapewni działanie formularza HTML podczas operacji edycji/zmiany danych książki lub dodawania nowej książki.

Uruchom aplikację Spring za pomocą Run As / Spring Boot App.

Jeśli pojawi się komunikat błędu dotyczący próby ponownego załadowania danych z pliku **data.sql** - związanych z powtórzeniem wartości klucza, to zmień w katalogu resources nazwę pliku z **data.sql** na **rem\_data.sql** i ponownie uruchom aplikację Spring Boot App.

W przeglądarce internetowej wywołaj adres URL: [http://localhost:8080/books\\_list](http://localhost:8080/books_list). Powinna pojawić się strona jak na obrazku poniżej.

## Books List

[Create New Book](#)

| Book ID | Title   | Publisher | Category   |                      |                        |
|---------|---|-----------|------------|----------------------|------------------------|
| 1       | Java. Podstawy                                | Helion    | Java       | <a href="#">Edit</a> | <a href="#">Delete</a> |
| 2       | Projektownie serwisów WWW. Standardy sieciowe | Helion    | WWW        | <a href="#">Edit</a> | <a href="#">Delete</a> |
| 3       | Zrozumieć JavaScript                          | Helion    | JavaScript | <a href="#">Edit</a> | <a href="#">Delete</a> |
| 4       | Head first Java                               | OREILLY   | Java       | <a href="#">Edit</a> | <a href="#">Delete</a> |
| 5       | HTML5. Komponenty                             | PWN       | HTML       | <a href="#">Edit</a> | <a href="#">Delete</a> |
| 6       | Wydajny JavaScript                            | PWN       | JavaScript | <a href="#">Edit</a> | <a href="#">Delete</a> |

## Widok formularza do dodawania książek

Wygeneruj nowy plik `new_book.html`. Dołącz przestrzenie nazw do znacznika `html` i ustaw `title` jako `Create new book`. W części `body` umieść `div` z formularzem jak poniżej.

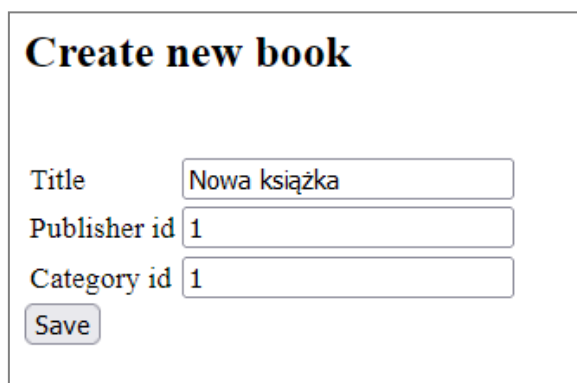
```
<div>
  <h2>Create new book</h2>
  <br />
  <form action="#" th:action="@{/save_book}" th:object="${book}" method="post">
    <table>
      <tr>
        <td>Title</td>
        <td><input type="text" th:field="*{title}" /></td>
      </tr>
    </table>
  </form>
</div>
```

```

    </tr>
    <tr>
      <td>Publisher id</td>
      <td><input type="text" th:field="*{publisher.idp}" /></td>
    </tr>
    <tr>
      <td>Category id</td>
      <td><input type="text" th:field="*{category.idc}" /></td>
    </tr>
  </table>
  <button type="submit">Save</button>
</form>
</div>

```

Formularz może wyglądać tak jak na obrazku poniżej.



Uruchom aplikację i przetestuj działanie formularza dodając dane nowej książki (pamiętaj, że jako id Publisher oraz Category trzeba podać odpowiednie liczby).

Więcej informacji na temat Thymeleaf można znaleźć na przykład po adresie:

<https://www.thymeleaf.org/doc/articles/standarddialect5minutes.html>

## 8. Edycja i usuwanie danych

Dodanie do aplikacji kolejnych możliwości wymaga zdefiniowania odpowiednich akcji kontrolera BookController. Każda z metod opisujących akcję ma swoją charakterystyczną konstrukcję. Pozostałe dwie metody do obsługi formularza w aplikacji SpringBooks to edycja i usuwanie danych. Metoda do edycji może mieć następującą postać:

```

@RequestMapping("/edit_book/{idb}")
public ModelAndView showEditFormBook(@PathVariable(name = "idb") Long idb) {
    ModelAndView mav = new ModelAndView("edit_book");
    Optional<Book> eb = service.findById(idb);
    mav.addObject("book", eb);
    return mav;
}

```

Metoda do usuwania danych może mieć postać:

```
@RequestMapping("/delete_book/{idb}")
public String deleteBook(@PathVariable(name = "idb") Long idb) {
    service.deleteById(idb);
    return "redirect:/books_list";
}
```

Akcje w powyższym kodzie wykorzystują metody usługi BookService: `deleteById()` oraz `findById()` wygenerowane wcześniej.

W kodzie zostały użyte dodatkowe adnotacje `@PathVariable` i `@ModelAttribute` oraz klasa `ModelAndView`. Adnotacja `@PathVariable` wskazuje jaki parametr metody powiązany jest ze zmienną szablonu URI. W naszym przypadku jest to `idb`. Klasa `Spring ModelAndView` umożliwia kontrolerowi korzystanie zarówno z modelu, jak i widoku w postaci jednej wartości (obiektu). Obiekt ten jest również wynikiem działania metody.

Edycja książki wymaga zdefiniowania szablonu widoku z formularzem `edit_book.html`. Skonstruuj formularz do edycji danych książek. Skorzystaj z szablonu do dodawania danych nowej książki wprowadzając odpowiednie zmiany. Tytuł `title` zmień na `Edit book`. Dodaj do formularza pole ustawione na tylko do odczytu zawierające id książki.

```
<tr>
    <td>Book Id:</td>
    <td><input type="text" th:field="*{idb}" readonly="readonly" /></td>
</tr>
```

Sprawdź działanie aplikacji.

## 9. Sortowanie danych

Posortuj listę książek według tytułów.

Sortowanie można wykonać na kilka sposobów. Jednym z prostych rozwiązań jest dodanie deklaracji metody o odpowiedniej nazwie w interfejsie `BookRepository`. Dodaj do interfejsu deklarację metody do sortowania malejąco według id:

```
List<Book> findByIdOrderByIdbDesc();
```

Wywołaj ją w metodzie `findAll()` klasy `BookService`, zamiast

```
return repository.findAll();
```

wprowadź:

```
return repository.findByIdOrderByIdbDesc();
```

Przetestuj działanie metody uruchamiając ponownie aplikację. Sprawdź, czy książki zostały odpowiednio posortowane.

Posortuj książki według tytułów od A do Z. Zadeklaruj w interfejsie `BookRepository` dodatkową deklarację metody: `List<Book> findByIdOrderByTitleAsc();`

Dokonaj odpowiednich zmian w metodzie `findAll()` klasy `BookService` i przetestuj działanie aplikacji.

Zadeklaruj i wywołaj inną metodę do sortowania, tym razem malejąco Desc według nazwy wydawcy. Ponieważ modele zostały połączone relacjami, można korzystać z pól innych modeli poprzedzając je nazwą klasy modelu. Metoda może mieć postać:

```
List<Book> findByOrderByPublisherNameDesc();
```

Można również odwoływać się do kilku kolumn. Deklaracja metody do sortowania rosnąco według nazwy wydawcy i tytułu książki może mieć postać:

```
List<Book> findByOrderByPublisherNameAscTitleAsc();
```

Skonstruuj widok strony books\_list\_sort z listą książek posortowaną według nazw kategorii i tytułów.

Nie zawsze można wykorzystać automatyczne nazwy. Na przykład pola ze znakiem podkreślenia ( \_ ) w nazwie nie mogą być wołane w ten sposób. Natomiast zapytania do bazy danych można dołączyć do modelu Book w postaci adnotacji @NamedNativeQuery. Trzeba je umieścić przed definicją klasy, ale za adnotacją @Entity. Zapytanie może mieć postać:

```
@NamedNativeQuery(name = "Book.findAllByTitleNativeSQL",  
query = "SELECT * FROM books.book ORDER BY title ",  
resultClass = Book.class)
```

Trzeba jeszcze dodać do interfejsu BookRepository deklarację zgodnie z nazwą metody:

```
List<Book> findAllByTitleNativeSQL();
```

oraz wywołać tę metodę w usłudze.

Jeszcze inny sposób to zdefiniowanie zapytania za pomocą adnotacji @Query bezpośrednio w repozytorium.

## 10. Menu i styl

Dodaj do aplikacji menu skonstruowane w Thymeleaf oraz podstawowy arkusz stylów. Wprowadzenie do tematu podziału stron w Thymeleaf można znaleźć na przykład pod adresem:

<https://www.baeldung.com/spring-thymeleaf-fragments#views>

## 11. Dostosowanie aplikacji SpringBooks

- Dodaj do aplikacji możliwość zarządzania danymi wydawców modelu Publisher oraz danymi kategorii modelu Category uzupełniając kod o odpowiednie kontrolery (dodaj repozytoria i usługi jeśli nie zostały dodane wcześniej).
- Dodaj do formularzy z danymi książek pola select do wyboru wydawcy i kategorii z listy dostępnych w bazie danych. Wprowadzenie można znaleźć pod adresem:  
<https://www.baeldung.com/thymeleaf-select-option>
- Oprogramuj sprawdzanie poprawności danych po stronie serwera.
- Inne według własnych pomysłów. (Opcja)

## **12. Opublikowanie aplikacji na serwerze foka**

Wyeksportuj aplikację do pliku .war wybierając projekt i opcję Export |WAR file. Opublikuj aplikację na swoim egzemplarzu Tomcat na serwerze foka.

### **Sprawozdanie**

W sprawozdaniu w systemie Sprawer wyślij link do działającej aplikacji wystawionej na serwerze foka oraz folder main i plik build.gradle z projektu ze źródłami spakowany do pliku .zip. W przypadku, gdy projekt nie pochodzi z Eclipse IDE, proszę dodatkowo w komentarzu napisać w jakim IDE był realizowany.