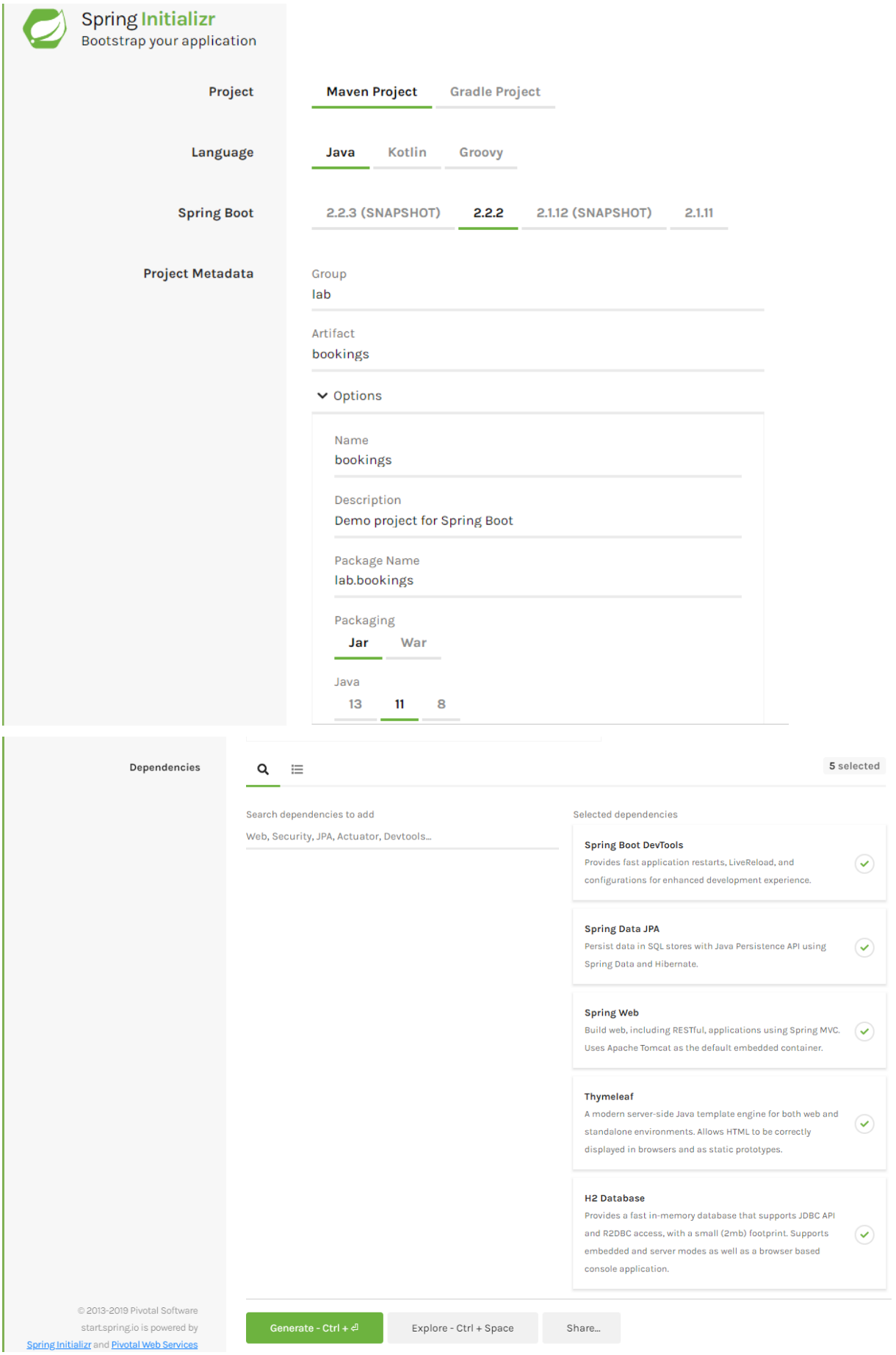
Spring: JPA + Spring MVC + Thymeleaf

Celem ćwiczenia jest wykorzystanie frameworka Spring do stworzenia aplikacji internetowej do rezerwacji apartamentów. Projekt zostanie stworzony przy użyciu Spring Boot, z wykorzystaniem Spring MVC jako frameworka aplikacji webowej, JPA do dostępu do bazy danych oraz Thymeleaf dla interfejsu użytkownika. Założonym IDE jest IntelliJ IDEA (Community Edition), ale można użyć dowolnego innego IDE wspierającego projekty Maven lub Gradle, ponieważ nie będziemy polegać na kreatorach do tworzenia komponentów aplikacj.

1. Wygeneruj projekt Spring Boot wykorzystując Spring Initializr:
    a) Otwórz adres Spring Initializr w przeglądarce: https://start.spring.io/
    b) Wybierz Maven jako narzędzie do budowania projektu. (Jeśli chcesz, możesz wybrać Gradle. Niektóre szczegóły dotyczące importu projektu do IDE będą wtedy inne.)
    c) Wybierz Javę jako język projektu.
    d) Wybierz najnowszą dostępną produkcyjną (nie SNAPSHOT) wersję Spring Boot.
    e) W sekcji Project Metadata wprowadź “lab” jako Group i “bookings” jako Artifact.
    f) Rozwiń sekcję Options. Upewnij się, że wybrane jest pakowanie do JAR. Wybierz wersję Javy, która najlepiej pasuje do wersji Javy, którą posiadasz zainstalowaną w swoim systemie.
    g) W sekcji Dependencies wprowadź: DevTools, Web, Thymeleaf, JPA i H2

       H2 jest lekką bazą danych SQL, wystarczającą do celów deweloperskich, a DevTools jest częścią Spring Boot upraszczającą tworzenie aplikacji poprzez oferowanie automatycznego restartu aplikacji i ponownego ładowania przeglądarki podczas przebudowy projektu po dokonaniu zmian w kodzie aplikacji.
    h) Kliknij przycisk Generate, aby wygenerować i pobrać archiwum zip zawierające projekt startowy.

    Uwaga: Pełne ustawienia projektu prezentują poniższe zrzuty ekranu.


Spring Initializr
Bootstrap your application
Project
Maven Project
Gradle Project
Language
Java
Kotlin
Groovy
Spring Boot
2.2.3 (SNAPSHOT)
2.2.2
2.1.12 (SNAPSHOT)
2.1.11
Project Metadata
Group
lab
Artifact
bookings
Options
Name
bookings
Description
Demo project for Spring Boot
Package Name
lab.bookings
Packaging
Jar
War
Java
13
11
8
Dependencies
5 selected
Search dependencies to add
Web, Security, JPA, Actuator, Devtools...
Selected dependencies
Spring Boot DevTools
Provides fast application restarts, LiveReload, and configurations for enhanced development experience.
Spring Data JPA
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.
Spring Web
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
Thymeleaf
A modern server-side Java template engine for both web and standalone environments. Allows HTML to be correctly displayed in browsers and as static prototypes.
H2 Database
Provides a fast in-memory database that supports JDBC API and R2DBC access, with a small (2mb) footprint. Supports embedded and server modes as well as a browser based console application.
© 2013-2019 Pivotal Software
start.spring.io is powered by
Spring Initializr and Pivotal Web Services
Generate - Ctrl + ⏎
Explore - Ctrl + Space
Share...

2. Zaimportuj wygenerowany projekt do IntelliJ IDEA:
    a) Rozpakuj projekt.
    b) Otwórz IntelliJ IDEA.
    c) Na ekranie powitalnym kliknij przycisk Open i wybierz katalog projektu.
    d) Zobacz zawartość pliku konfiguracyjnego projektu (pom.xml dla projektu Maven, build.gradle dla projektu Gradle). Zlokalizuj startery Spring Boot wśród zależności projektu.
    e) Zlokalizuj główną klasę aplikacji Spring Boot i obejrzyj jej zawartość.

    Uwaga: Po otwarciu projektu, IDE zsynchronizuje i pobierze zależności projektowe. Może to zająć trochę czasu.
3. Utwórz aplikację webową typu "Hello World" aby zobaczyć Spring Boot w działaniu:
    a) Utwórz plik index.html w katalogu src/main/resources/templates (Jest to domyślna lokalizacja dla szablonów Thymeleaf.)
    b) Umieść następującą zawartość w utworzonym pliku:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8"/>
    <title>Welcome page</title>
</head>
<body>
Hello World!
</body>
</html>
```

    c) Utwórz nowy pakiet dla kontrolerów Spring MVC jako węzeł potomny pakietu zawierającego główną klasę aplikacji. Pełna nazwa pakietu powinna brzmieć lab.bookings.controllers.

    Notka: Dzięki adnotacji Spring Boot na klasie głównej, wszystkie pakiety, które znajdują się poniżej pakietu klasy głównej w hierarchii pakietów zostaną automatycznie zeskanowane w poszukiwaniu komponentów Spring (beanów).

    d) W nowym pakiecie utwórz klasę i nazwij ją IndexController. Umieść w niej następujący kod:

```
@Controller
public class IndexController {

    @RequestMapping("/")
    public String index() {
        return "index";
    }
}
```

    Notka: Klasa będzie działać jako bean Springa i kontroler Spring MVC dzięki adnotacji @Controller. Kontroler posiada jedną metodę akcji zwracającą nazwę widoku do wyświetlenia. W Spring MVC nazwy widoków są tłumaczone na ścieżki systemu plików przez komponenty (beany) określane jako view resolver. Spring Boot dostarcza domyślny view resolver dla Thymeleaf automatycznie po dodaniu do zależności projektu startera Thymeleaf.

    e) Uruchom aplikację (np. poprzez kliknięcie prawym przyciskiem myszy w głównym oknie edytora klasy).
    f) Przejdź do następującego adresu w przeglądarce: http://localhost:8080/.

4. Zmień odwzorowanie metody akcji kontrolera na adres "/welcome". Przebuduj aplikację i sprawdź w przeglądarce, czy nowy adres URL strony powitalnej działa.
5. Zmodyfikuj aplikację tak, aby pozdrawiała użytkownika po imieniu zakodowanym na stałe w kontrolerze (na razie):
    a) Zmodyfikuj metodę akcji tak, aby dodawała łańcuch "Anonymous" jako atrybut "name" do modelu.

```
    @RequestMapping("/welcome")
    public String index(Model model) {
        model.addAttribute("name","Anonymous");
        return "index";
    }
```

    b) Zmodyfikuj szablon widoku tak, aby wyświetlane było przekazane imię:

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8"/>
    <title>Welcome page</title>
</head>
<body>
Hello <span th:text="${name}">Stranger</span>!
</body>
</html>
```

    Zwróć uwagę na dodanie przestrzeni nazw Thymeleaf oraz na sposób użycia Expression Language w celu odwołania się do modelu.

    c) Przebuduj aplikację (Ctrl+F9 w Intellij IDEA) i zaobserwuj efekt dokonanych zmian w przeglądarce.

    Jeśli podmiana aplikacji „na gorąco" ("hot swap") nie powiedzie się, zawsze możesz uruchomić aplikację od nowa (Shift+F10 w Intellij IDEA).

    d) Otwórz szablon Thymeleaf w przeglądarce bezpośrednio z systemu plików (nie przez aplikację). Powinieneś zauważyć jedną z zalet Thymeleaf w porównaniu z JSP: domyślne wartości dla elementów powiązanych.
6. Zmodyfikuj aplikację tak, aby witała użytkownika za pomocą imienia podanego jako parametr URL:
    a) Zmodyfikuj metodę akcji tak, aby pobierała imię użytkownika jako parametr zapytania przed dodaniem go jako atrybutu do modelu.

```
@RequestMapping("/welcome")
public String index(@RequestParam(required = false) String name,
                    Model model) {
    if (name == null || name.equals(""))
        name = "Anonymous";
    model.addAttribute("name", name);
    return "index";
}
```

    b) Przebuduj aplikację.
    c) Przetestuj aplikację w przeglądarce sprawdzając jej zachowanie z podanym i niepodanym parametrem "name" w adresie URL żądania.

7. Utwórz nowy pakiet i nazwij go “lab.bookings.models”. Będzie on zawierał encje dla aplikacji, którą zamierzamy zbudować.
8. W nowo utworzonym pakiecie stwórz klasę “Apartment”. Dodaj do niej adnotację JPA @Entity Utwórz trzy prywatne pola: id (Long), name (String) oraz capacity (int). Oznacz pole id jako identyfikator encji poprzez adnotację @Id. Wygeneruj gettery i settery dla dodanych pól.

```
// package and imports

@Entity
public class Apartment {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;
    private int capacity;

    // getters & setters
}
```

9. Uwtórz nowy pakiet “lab.bookings.repositories” a w nim interfejs “ApartmentRepository”, który będzie rozszerzał interfejs generyczny CrudRepository<Apartment, Long>.

```
public interface ApartmentRepository
       extends CrudRepository<Apartment, Long> {
}
```

Notka: Spring dostarcza generyczny interfejs CrudRepository w celu obsługi operacji CRUD dla danej encji. Posiada on dwa typy jako parametry: typ encji oraz typ jej identyfikatora. Implementacja interfejsu w aplikacji nie jest wymagana – Spring dostarczy funkcjonalność automatycznie. Zabieg służy uporządkowaniu kodu oraz wspomaga podpowiedzi w narzędziach programistycznych.

10. Utwórz kontroler obsługujący apartamenty według następujących kroków:
    a) Dodaj nową klasę ApartmentController w tym samym pakiecie co poprzednio tworzony kontroler.
    b) Oznacz klasę przy pomocy adnotacji jako kontroler Spring MVC.
    c) Dodaj adnotację @RequestMapping ustawiającą odwzorowanie kontrolera na ścieżkę “/apartments”.
    d) Stwórz prywatne pole “repository” typu ApartmentRepository.
    e) Utwórz publiczny konstruktor przyjmujący jeden argument posiadający taką samą nazwę jak i typ co dodane wcześniej pole. Ustaw w nim wartość pola “repository” na wartość przekazaną parametrem.
    f) Dodaj akcję getAll(), która będzie obsługiwała zapytania metodą GET (bez podawania ścieżki URL specyficznej dla metody). Powinna ona:
        i. Pobrać kolekcję wszystkich apartamentów z repozytorium, a następnie dodać je do modelu jako atrybut “apartments”.
        ii. Utworzyć nową instancję klasy Apartment i dodać ją do modelu pod domyślną nazwą (taką samą jak nazwa klasy zaczynająca się małą literą). Ten obiekt będzie powiązany z formularzem dodawania nowego apartamentu.
        iii. Zwrócić “apartments” jako nazwę widoku.
    g) Utwórz akcję create() do obsługi żądania POST (bez podawania ścieżki URL specyficznej dla metody). Powinna ona:

i. Przyjmować instancję klasy Apartment jako jej argument. (Dane o nowym apartamencie będą pochodzić z formularza HTML. Spring MVC zbierze dane z formularza i przekaże je do metody akcji w formie obiektu.

ii. Zapisać dostarczoną instancję klasy Apartment w bazie danych poprzez repozytorium.

iii. Zwrócić łańcuch znaków “redirect:/apartments”, który poinstruuje Spring MVC aby przekierował przeglądarkę do strony z wylistowanymi apartamentami.

```
@Controller
@RequestMapping("/apartments")
public class ApartmentController {
    private ApartmentRepository repository;

    public ApartmentController(ApartmentRepository repository) {
        this.repository = repository;
    }

    @GetMapping()
    public String getAll(Model model) {
        model.addAttribute(new Apartment());
        model.addAttribute("apartments", repository.findAll());
        return "apartments";
    }

    @PostMapping()
    public String create(Apartment apartment) {
        repository.save(apartment);
        return "redirect:/apartments";

    }
}
```

11. Stwórz widok (szablon Thymeleaf) z formularzem do dodawania apartamentu oraz tabelą prezentującą istniejące apartamenty:

a) Utwórz plik apartments.html w folderze src/main/resources/templates.

b) Dodaj formularz HTML powiązany atrybutami Thymeleaf (th:object oraz th:field) z instancją encji Apartment przekazaną w modelu.

c) Dodaj tabelę HTML powiązaną atrybutami Thymeleaf (th:each oraz th:text) z kolekcją encji Apartment przekazaną w modelu.

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8"/>
    <title>Apartments</title>
</head>
<body>
<form th:object="${apartment}" method="post">
    <label for="name">Name: </label>
    <input id="name" type="text" required th:field="*{name}"/>
    <label for="capacity">Capacity: </label>
    <input id="capacity" type="number"
           th:field="*{capacity}"/>
    <input type="submit" value="Add"/>
</form>

<table border="1">
```

```
    <thead>
    <tr>
        <th>Id</th>
        <th>Name</th>
        <th>Capacity</th>
    </tr>
    </thead>
    <tbody>
    <tr th:each="a: ${apartments}">
        <td th:text="${a.id}">1</td>
        <td th:text="${a.name}">Fabulous</td>
        <td th:text="${a.capacity}">4</td>
    </tr>
    </tbody>
</table>
</body>
</html>
```

d) Przebuduj projekt.

e) Przejdź w przeglądarce do adresu http://localhost:8080/apartments i dodaj kilka apartamentów.

Notka: Znajdująca się w pamięci instancja bazy H2 została skonfigurowana i wystartowana przez Spring Boot. Ponadto JPA w aplikacji zostało skonfigurowane aby łączyć się z tą instancją.

f) Odśwież stronę w przeglądarce po dodaniu apartamentu. Czy formularz został wysłany ponownie?

Notka: Zaimplementowaliśmy wzorzec Post-Redirect-Get, który zapobiega ponownemu wysłaniu formularza po odświeżeniu strony.

12. Dodaj walidację pola capacity na poziomie modelu danych (encji)

a) W encji Apartment wykorzystaj adnotację @Min aby ustawić 1 jako minimalną wartość pola. Jako komunikat błędu ustaw: “Value must be a positive integer”.

b) Zmodyfikuj akcję POST kontrolera tak, aby instancja encji Apartment była zapisana tylko, jeżeli jest poprawna. W przeciwnym wypadku przekieruj użytkownika do widoku apartamentów.

```
    @PostMapping()
    public String createTable(@Valid Apartment apartment,
                              Errors errors, Model model) {
        if (errors.hasErrors()) {
            model.addAttribute("apartments", repository.findAll());
            return "apartments";
        } else {
            repository.save(apartment);
            return "redirect:/apartments";
        }
    }
```

Notka: Argumenty akcji kontrolera w Spring MVC nie są walidowane automatycznie. Adnotacja @Valid jest wymagana aby uruchomić Bean Validation. Argument Errors musi znaleźć się na liście argumentów, aby błędy zostały przekazane z kontrolera do widoku.

c) Dodaj następującą linię na końcu formularza w widoku aby wyświetlić błędy modelu.

```
<p th:if="${#fields.hasErrors('capacity')}"
   th:errors="*{capacity}">Capacity Error</p>
```

d) Przebuduj projekt i przetestuj walidację.

Notka: Moglibyśmy dodać walidację HTML5 na poziomie przeglądarki (atrybut min dla pola INPUT typu liczbowego) aby uniemożliwić wysyłkę niepoprawnych wartości. Niemniej jednak sprawdzenie poprawności danych wejściowych w warstwie logiki biznesowej również powinno być wykonane.

13. Utwórz klasę encji do reprezentacji rezerwacji:

a) Dodaj klasę Booking w tym samym pakiecie co klasa Apartment.

b) Zastąp wygenerowany kod klasy poniższym:

```
@Entity
public class Booking {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @ManyToOne
    private Apartment apartment;

    @NotNull
    @DateTimeFormat(iso = DateTimeFormat.ISO.DATE)
    private LocalDate fromDate;

    private LocalDate toDate;

    @Min(value = 1)
    @Max(value = 366)
    private Integer numDays = 1;

    @Min(value = 1)
    private Integer numGuests = 1;

    @NotEmpty
    private String firstName;

    @NotEmpty
    private String lastName;
}
```

Notka: Adnotacja @DateTimeFormat jest wymagana aby Spring wiedział w jaki sposób przekonwertować łańcuch zanków pochodzący z interfejsu użytkownika do daty w formacie Java. Adnotacja nie będzie potrzebna dla pola zawierającego koniec rezerwacji, gdyż użytkownik zostanie poproszony o liczbę dni zamiast daty.

c) Wygeneruj gettery i settery dla pól klasy Booking.

d) Z uwagi na to, że data końca rezerwacji nie znajdzie się w formularzu rezerwacji, dodaj w klasie encji metodę oznaczoną adnotacją @PrePersist. Jej zadaniem będzie obliczenie daty końca rezerwacji przed zapisaniem encji w bazie danych:

```
    @PrePersist
    private void onPrePersist() {
        toDate = fromDate.plusDays(numDays);
    }
```

e) Data końca rezerwacji jest niezbędna do wyszukania dostępnych apartamentów do wynajęcia przed zapisaniem rezerwacji w bazie danych. Ze względu na to, zamień wygenerowany getter dla pola daty końca rezerwacji poniższym:

```
    public LocalDate getToDate() {
        return fromDate.plusDays(numDays);
    }
```

14. Na chwilę obecną posiadamy relację jednokierunkową wiele do jednego (adnotacja @ManyToOne) z nowo utworzonej encji Booking do encji Apartment. Dodaj poniższe pole do encji Apartment aby uczynić relację dwukierunkową. Następnie wygeneruj getter i setter dla dodanego pola.

```
    @OneToMany(mappedBy = "apartment")
    private List<Booking> bookings;
```

15. Utwórz interfejs repozytorium (BookingRepository) dla encji Booking analogicznie do utworzonego wcześniej dla encji Apartment.

16. Dodaj następującą metodę do ApartmentRepository:

```
@Query("select a from Apartment a " +
       "where a.capacity >= :numGuests and " +
       "not exists (select b from a.bookings b " +
       "  where :startDay < b.toDate and :endDay > b.fromDate) " +
       "order by a.capacity")
List<Apartment> getFreeApartments(@Param("numGuests") Integer numGuests,
                                  @Param("startDay") LocalDate startDay,
                                  @Param("endDay") LocalDate endDay);
```

Notka: Metoda korzysta z zapytania JPQL w celu znalezienia dostępnych apartamentów spełniających kryteria podane przez użytkownika. Upewnij się, że rozumiesz wszystkie składowe zapytania.

17. Utwórz kontroler BookingController do obsługi rezerwacji przy pomocy poniższego kodu. Przeanalizuj go.

```
@Controller
@RequestMapping("/bookings")
public class BookingController {
    private BookingRepository bookingRepository;
    private ApartmentRepository apartmentRepository;

    public BookingController(BookingRepository bookingRepository,
                             ApartmentRepository apartmentRepository) {
        this.bookingRepository = bookingRepository;
        this.apartmentRepository = apartmentRepository;
    }

    @GetMapping
    public String getAll(Model model) {
        model.addAttribute(new Booking());
        model.addAttribute("bookings", bookingRepository.findAll());
        return "bookings";
    }

    @PostMapping
    public String create(@Valid Booking booking, Errors errors,
                         Model model) {
        String view;
        if (errors.hasErrors()) {
            model.addAttribute("bookings",
                               bookingRepository.findAll());
            view = "bookings";
        } else {
            int numGuests = booking.getNumGuests();
            LocalDate startDay = booking.getFromDate();
            LocalDate endDay = booking.getToDate();
            List<Apartment> availableApartments =
apartmentRepository.getFreeApartments(numGuests, startDay, endDay);
```

```
            if(availableApartments.size() > 0) {
                booking.setApartment(availableApartments.get(0));
                bookingRepository.save(booking);
                view = "redirect:/bookings";
            } else {
                model.addAttribute("noApartmentAvailable", true);
                model.addAttribute("bookings",
                                   bookingRepository.findAll());
                view = "bookings";
            }
        }
        return view;
    }
}
```

Notka: Metoda POST musi być gotowa nie tylko na obsługę encji Booking naruszających ograniczenia deklaratywne (obsługa analogiczna do poprzednio utworzonego kontrolera), ale również na brak dostępnych apartamentów spełniających podane kryteria. W przypadku wystąpienia takiej sytuacji rezerwacja nie może zostać zapisana w bazie danych, a informacja o problemie powinna zostać przekazana do widoku poprzez ustawienie odpowiedniej flagi w modelu.

18. Utwórz widok do dodawania i wyświetlania rezerwacji (bookings.html) przy pomocy poniższego kodu. Odszukaj fragment odpowiedzialny za wyświetlanie informacji o braku dostępnych apartamentów spełniających kryteria użytkownika.

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8"/>
    <title>Bookings</title>
</head>
<body>
<form th:object="${booking}" method="post">
    <label>Last Name</label>
    <input type="text" th:field="*{lastName}" required/>
    <label>First Name</label>
    <input type="text" th:field="*{firstName}" required/>
    <label>Number of guests</label>
    <input type="number" min="1" th:field="*{numGuests}"/>
    <label>From</label>
    <input type="date" th:field="*{fromDate}" required/>
    <label>Number of days</label>
    <input type="number" min="1" th:field="*{numDays}"/>
    <input type="submit" value="Add"/>
    <p th:if="${noApartmentAvailable == true}">
       No apartment available for the selection criteria
    </p>
</form>

<table border="1">
    <thead>
    <tr>
        <th>Id</th>
        <th>Last Name</th>
        <th>Fist Name</th>
        <th>From</th>
        <th>To</th>
        <th>Number of days</th>
        <th>Number of guests</th>
```

```
        <th>Apartment</th>
        <th>Apartment capacity</th>
    </tr>
    </thead>
    <tbody>
    <tr th:each="b: ${bookings}">
        <td th:text="${b.id}"></td>
        <td th:text="${b.lastName}"></td>
        <td th:text="${b.firstName}"></td>
        <td th:text="${b.fromDate}"></td>
        <td th:text="${b.toDate}"></td>
        <td th:text="${b.numDays}"></td>
        <td th:text="${b.numGuests}"></td>
        <td th:text="${b.apartment.name}"></td>
        <td th:text="${b.apartment.capacity}"></td>
    </tr>
    </tbody>
</table>
</body>
</html>
```

19. Przebuduj aplikację.
20. Przetestuj aplikację w przeglądarce poprzez dodanie paru rezerwacji. Zweryfikuj również jej zachowanie w przypadku braku dostępnych apartamentów z powodu konfliktu czasowego z istniejącymi rezerwacjami.

Zadanie do samodzielnego wykonania

Zaimplementuj mechanizm usuwania rezerwacji.

Notka: Nie wykorzystuj zapytania GET do tego celu. Jedną z możliwości jest wysłanie żądania POST z id rezerwacji na dedykowany URL np. „bookings/delete”, inną wykorzystanie dedykowanej metody i żądania DELETE (symulowanego za pomocą POST przez Thymeleaf gdyż DELETE nie jest deklaratywnie obsługiwany przez formularze HTML).