

Exercise 4 (ver. Spring)

REST Service

Author: Mariusz Fraś

1 Objectives of the exercise

The purpose of the exercise is:

1. Mastering the basics of the technique of implementing REST web services – Part A.
2. Mastering the technique of supporting HATEOAS – Part B.

The exercise consists in:

- Building a REST-based service that provides CRUD operations.
 - The service will handle HTTP requests (GET, POST, ... methods) that reference resources using specific URLs.
 - The service operates on data in JSON format.
- Extending service with HATEOS capabilities
- Testing the service with Postman (or alike) or console client.

2 Środowisko wytwórcze

Aplikacja jest opracowana dla następującego środowiska:

- Windows 10 (or higher) Operating System (also possible MacOS or Linux).
- IntelliJ.IDEA programming platform with proper plugins – here is assumed Ultimate Edition with Spring framework boundled (Spring Boot, Spring Initializr,...).

W ćwiczeniu będzie opracowana aplikacji z użyciem frameworka Spring. Spring Framework wspiera tworzenie aplikacji REST (i RESTful) w szerokim zakresie – m.in. z użyciem wzorca MVC.

3 Exercise – Part A Basic Rest Service

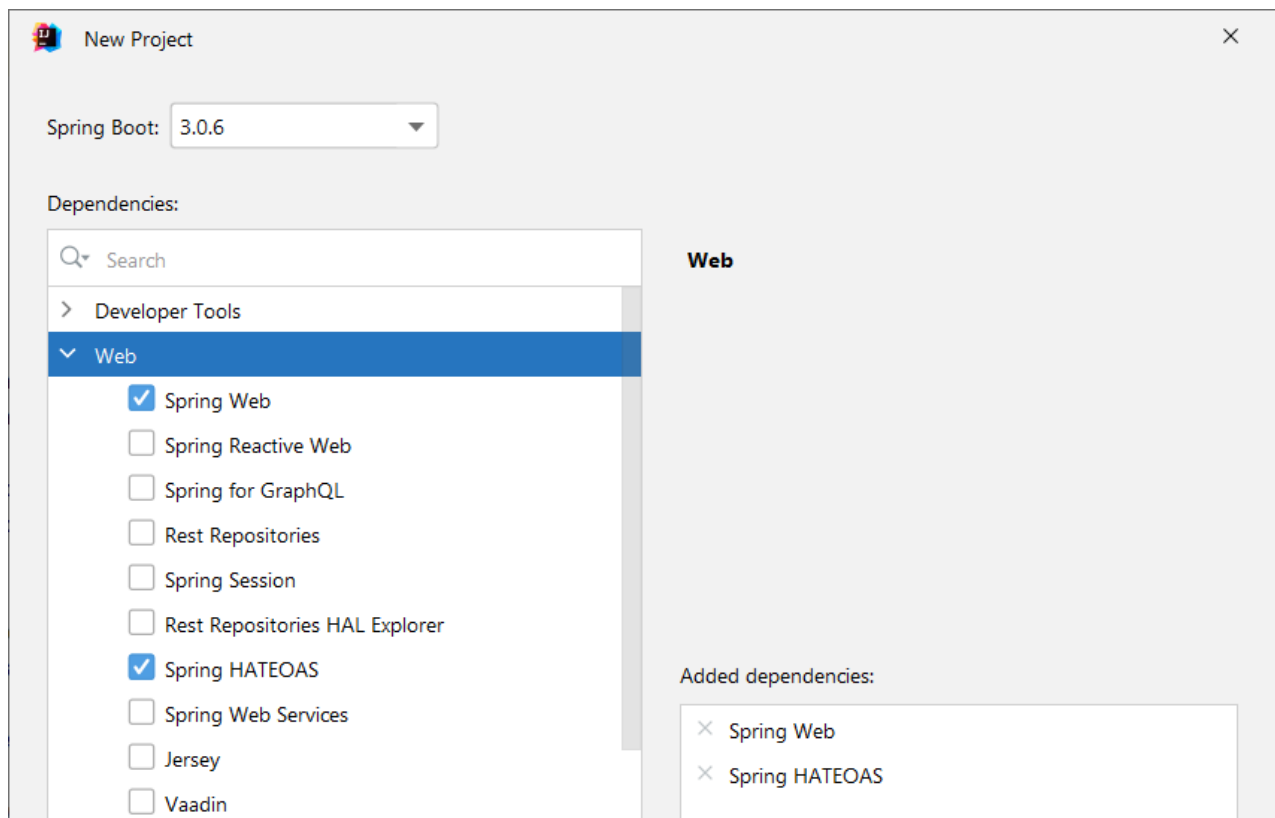
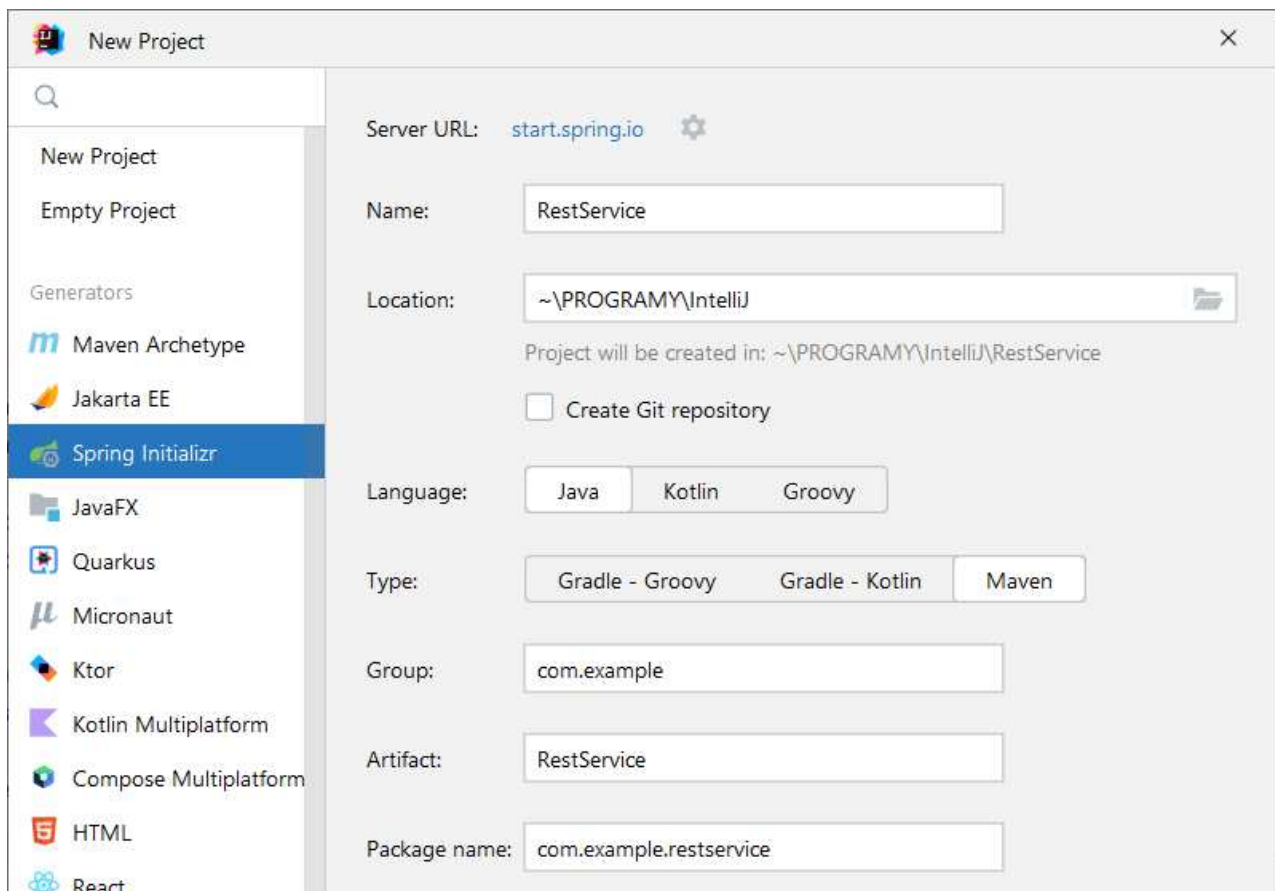
3.1 Struktura projektu

IntelliJ.IDEA Ultimate pozwala wykorzystać mechanizm Spring (Spring Initializr) do wygenerowania szablonu projektu aplikacji. Poniższe komponenty POM uzyskuje się stosując opcje pokazane na kolejnych obrazkach.

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>3.0.5</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>
...
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-hateoas</artifactId>
  </dependency>
</dependencies>
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

spring-boot-starter-web , **spring-boot-starter-hateoas** oraz plugin **spring-boot-maven-plugin** dostarczają klasy i mechanizm budowania kodu z pomocą adnotacji (annotations) – do implementacji podstawowych aplikacji webowych (w tym REST), oraz do wsparcia własności HATEOAS (w szczególności obsługi HAL).

- Wygeneruj szablon aplikacji SpringBoot z opcjami **Spring Web** i **Spring HATEOAS**.



3.2 Podstawowy kod serwisu

3.2.1 Model / Repozytorium danych

- Zdefiniuj własne klasy dla obsługi modelu danych. Klasy są takie jak w ćwiczeniu 3. Tu są to:

- **Person** – klasa element danych
 - zawiera pola **int id**, **String name** i **int age**,
 - zawiera gettery i settery,
 - opcjonalnie konstruktor parametrowy.
 - **PersonRepository** – interfejs dla repozytorium
 - definiuje metody CRUD:


```
List<Person> getAllPersons();
Person getPerson(int id) throws PersonNotFoundException;
Person updatePerson(Person person) throws PersonNotFoundException;
boolean deletePerson(int id) throws PersonNotFoundException;
Person addPerson(Person person) throws BadRequestEx;
```
 - **PersonRepositoryImpl** – klasa repozytorium przechowywania danych.
 - zawiera listę danych


```
private List<Person> personList;
```
 - zawiera implementacje metod interfejsu – np.:


```
public Person getPerson(int id) {
    for (Person thePerson : personList) {
        if (thePerson.getId() == id) {
            return thePerson;
        }
    }
    throw new PersonNotFoundException(id);
}
```
- Zaimplementuj pozostałe metody
- opcjonalnie zawiera konstruktor inicjujący listę kilkoma elementami.

Dla metod klasy można zdefiniować zwracany status http odpowiedzi za pomocą adnotacji **@ResponseStatus(code = XXX)** gdzie XXX to status odpowiedzi HTTP.

- Zdefiniuj klasy wyjątków:
 - Zdefiniuj klasy dla wyjątków występujących w interfejsie repozytorium.
 - Oznacz klasy adnotacją specyfikującą odpowiedni kod błędu odpowiedzi. Np.:


```
@ResponseStatus(code = HttpStatus.NOT_FOUND)
public class PersonNotFoundException extends RuntimeException {
    public PersonNotFoundException() {
        super("The specified person does not exist");
    }
    public PersonNotFoundException(int id) {
        super(String.valueOf(id));
    }
}
```

3.2.2 Metody mapowane dla żądań REST

Kontrolery pozwalają mapować żądania REST (definiowane metodą HTTP i URI) do implementowanych metod klasy **kontrolera**.

Do zmiennych parametrów ścieżki URI żądania można odnieść się adnotacją **@PathVariable**. Adnotacja **@RequestBody** dla parametru metody pozwala odnieść się do danych przekazanych w żądaniu (zwykle POST lub PUT) – czyli serializuje przekazywane dane JSON do obiektu Javy (POJO).

- Zdefiniuj klasę kontrolera (tu: **PersonController**). W klasie musi być:
 - oznaczenie klasy jako kontrolera REST,
 - definicja metod dla odpowiednich operacji CRUD serwisu,
 - mapowanie żądań REST do poszczególnych metod ze specyfikacją URI i metody HTTP,
 - tu również zainicjowana zmienna repozytorium.

- Oznacz klasę adnotacją kontrolera:

```
@RestController
public class PersonController {
    ...
}
```

- Zdefiniuj metody dla żądań z adnotacjami mapowania dla URI i metody HTTP

```
@RequestMapping(value = "XXX", method = RequestMethod.YYY)
```

gdzie **XXX**- odpowiednia ścieżka względna, **YYY** – odpowiednia metoda

lub skrótowo:

```
@YYMapping("XXX")
```

Np. dla pobierania elementu danych listy:

(np. tu GET http://host_address/persons/{id})

```
@GetMapping("/persons/{id}")
```

```
public Person getPerson(@PathVariable int id) {
    System.out.println("...called GET"); //dla śledzenia działania
    return dataRepo.getPerson(id);
}
```

gdzie: **{id}** – specyfikuje zmienną w URI, **@PathVariable** – definiuje odniesienie do zmiennej.

- Zdefiniuj i oznacz analogicznie wszystkie metody dla żądań CRUD:

GET http://host_address/persons/ – pobranie całej listy

DELETE http://host_address/persons/{id} – usunięcie elementu listy

POST http://host_address/persons/ (lub POST http://host_address/persons/{id}) – dodanie (przekazanie w żądaniu) elementu do listy

PUT http://host_address/persons/{id} – modyfikacja elementu listy

- o Dla metody POST i PUT uzyskaj w metodzie przekazywane dane specyfikując parametr metody adnotacją **@RequestBody** – np.:

```
public Person addPerson(@RequestBody Person person) {
    ...
}
```

- o Dodaj i zainicjuj zmienna repozytorium:

```
private PersonRepository dataRepo = new PersonRepositoryImpl();
```

Kontroler obsługi wyjątków pozwala w jednym miejscu przechwytywać rzucane wyjątki i odpowiednio definiować odpowiedzi w konkretnych sytuacjach. Klasa jest z adnotacją **@ControllerAdvice** lub **@RestControllerAdvice** (nieco upraszcza kod) a metody obsługujące dany wyjątek z adnotacją **@ExceptionHandler(...)**.

- Zdefiniuj klasę obsługi wyjątków (tu o nazwie **FaultController**).

- o Dodaj w projekcie klasę i oznacz ją adnotacją **@ControllerAdvice**:

```
@ControllerAdvice
public class FaultController {
    ...
}
```

- o Zdefiniuj w klasie metody-hendlery dla wyjątków i dodaj adnotacje

- handlera wyjątku: **@ExceptionHandler**

- odpowiedniego kodu odpowiedzi: **@ResponseStatus** (i stała klasy **HttpStatus**),

- zwrotu odpowiedzi do body odpowiedzi na żądanie: **@ResponseBody**.

Np. dla błędu odwołania się do elementu którego nie ma:

```
@ResponseBody
@ExceptionHandler(PersonNotFoundException.class)
@ResponseStatus(value = HttpStatus.NOT_FOUND)
String PNFEHandler(PersonNotFoundException e) {
    return HttpStatus.NOT_FOUND + " - The person of ID="
        + e.getMessage() + " DOES NOT EXIST";
}
```

Zdefiniuj metody dla pozostałych wyjątków.

- Uruchom serwis.

- o Przetestuj działanie z użyciem programu Postman.
- o Wykonaj żądania dla wszystkich metod.
- o Wykonaj żądania generujące wyjątki.

3.2.3 Specyfikacja statusu i nagłówek odpowiedzi

Dla metody kontrolera można zdefiniować podstawowy zwracany status HTTP odpowiedzi za pomocą adnotacji **@ResponseStatus(code = XXX)** gdzie XXX to status odpowiedzi HTTP (np. `HttpStatus.NotFound`).

Można też adaptacyjnie implementować wymagany status odpowiedzi.

Klasa **ResponseEntity<T>** pozwala specyfikować **status odpowiedzi HTTP**, **nagłówki HTTP**, a także **body** odpowiedzi (jak i inne elementy). Można jej użyć aby w konkretnych przypadkach ustawiać wymagany status. W tym celu należy:

- zmienić zwracany typ metody – np.:

```
ResponseEntity<Person> addPerson(@RequestBody Person person) {...}
```

- ustawić odpowiednio składowe zwracanego obiekty klasy **ResponseEntity<T>** - np.:

```
return ResponseEntity
    .status(HttpStatus.CREATED)
    .header(HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION_JSON_VALUE)
    .body(person);
```

Gdy nie potrzeba zwracać żadnych danych można użyć konstrukcji bez body. Np. dla statusu udanego kasowania: z kodem 204 No Content:

```
return ResponseEntity.noContent().build();
```

- Zaimplementuj odpowiednio zwracane statusy dla metod serwisu.
- Przetestuj działanie z użyciem programu Postman.

3.3 Kod Serwisu dla wsparcia HATEOAS

3.3.1 Wiadomości błędu w formacie RFC7807

Hendlery obsługi wyjątków mają zwracać w komunikaty zawierające pola wg RFC8707. Tu wyspecyfikujemy trzy pola: **Status**, **Title** i **Details**. Typ treści (Content-Type) ma być **application/problem+json**.

Wykorzystana zostanie klasa **ResponseEntity<T>** pozwalająca specyfikować **status odp. HTTP**, **nagłówki HTTP** i **body** odpowiedzi.

Dla metod-handlerów klasy kontrolera wyjątków (ControllerAdvice):

- Zmień typ zwracanych wartości na **ResponseEntity<?>**. Np.:

```
ResponseEntity<?> PNFEHandler(PersonNotFoundException e) {
    ...}
```
- Zmień zwracany obiekt na odpowiednio skonfigurowany **ResponseEntity**.
 Np. dla błędu **PersonNotFound** (404 Not Found) zwróć:

```
ResponseEntity.status(HttpStatus.NOT_FOUND)
    .header(HttpHeaders.CONTENT_TYPE,
        MediaType.HTTP_PROBLEM_DETAILS_JSON_VALUE)
    .body(Problem.create()
        .withStatus(HttpStatus.NOT_FOUND)
        .withTitle(HttpStatus.NOT_FOUND.name())
        .withDetail([...tu odpowiednia Wiadomosc...]));
```

3.3.2 Reprezentacja danych (zwracane informacje/dane) wg formatu HAL

HAL definiuje jeden ze sposobów wsparcia w wiadomościach HATEOAS dodając sekcję **_links** do wiadomości (odpowiedzi). Spring wspiera to rozwiązanie za pomocą klasy

EntityModel<T> oraz **CollectionModel<EntityModel<T>>**. Metody **linkTo(...)** oraz **with...(())** dodają odpowiednie pozycje w zwracanym obiekcie JSON.

Dla metod-klasy głównego kontrolera (RestController):

- Zmień typ zwracanych wartości na **EntityModel<Person>**. lub **EntityModel<?>**
Np.dla: GET `http://host_address/persons/{id}` które zwraca Person:

```
public EntityModel<Person> getPerson(@PathVariable int id) {
    ...
}
```

- Zmień zwracany obiekt na odpowiednio skonfigurowany EntityModel.

Np. dla żądania jak wyżej zwróć:

```
EntityModel.of(thePerson,
    LinkTo(methodOn(PersonController.class).getPerson(id)).withSelfRel(),
    LinkTo(methodOn(PersonController.class).delPerson(id)).withRel("delete"),
    LinkTo(methodOn(PersonController.class).getAllPersons()).withRel("list all")
);
```

Linkowanie do konkretnej metody kontrolera wygeneruje odpowiednie URI przekazywane w body odpowiedzi

W przypadku listy (kolekcji) elementów należy użyć **CollectionModel** oraz strumieniowo dodać do każdego zwracanego elementu w kolekcji odpowiednie linki z użyciem metody **linkTo(,,)**. Np dla listy z ćwiczenia, dla metody **getAllPersons()**:

- Zmień typ zwracanych wartości na **CollectionModel<EntityModel<Person>>**.

```
public CollectionModel<EntityModel<Person>> getAllPersons() {
    ...
}
```

- Zdefiniuj zmienną (tu **persons**) zwracanego typu i strumieniowo wygeneruj listę danych typu **EntityModel**:

```
List<EntityModel<Person>> persons =
    dataRepo.getAllPersons().stream().map(    person ->
        EntityModel.of(person,
            LinkTo(methodOn(PersonController.class)
                .getPerson(person.getId())).withSelfRel(),
            LinkTo(methodOn(PersonController.class)
                .delPerson(person.getId())).withRel("delete"),
            LinkTo(methodOn(PersonController.class)
                .getAllPersons()).withRel("list all")
        )
    ).collect(Collectors.toList());
```

- Zwróć ten utworzony obiekt dodając linki dla całej kolekcji:

```
CollectionModel.of(persons,
    LinkTo(methodOn(PersonController.class)
        .getAllPersons()).withSelfRel());
```


- Uruchom serwis.
 - Przetestuj działanie z użyciem programu Postman.
 - Wykonaj żądania dla wszystkich metod.
 - Wykonaj żądania generujące wyjątki.

Przykładowe odpowiedzi na żądania powinny mieć podobną zawartość jak poniżej:

1) Odpowiedź z błędem:

HTTP/1.1 404 No Found

Content-Type: application/problem+json

```
...
{
  "title": "NOT_FOUND",
  "status": 404,
  "detail": "The person of ID=22 DOES NOT EXIST"
}
```

2) Odpowiedź w formacie HAL:

HTTP/1.1 200 OK

Content-Type: application/json

```
...
{
  "id": 2,
  "name": "Adam Adamski",
  "age": 10,
  "_links": {
    "self": {
      "href": http://localhost:8080/persons/2
    },
    "delete": {
      "href": http://localhost:8080/persons/2
    },
    "list all": {
      "href": http://localhost:8080/persons
    }
  }
}
```

3.3.3 HATEAOS Extensions

HATEOAS wymaga dostarczenia w reprezentacji danych kierujących działaniem klienta – w uproszczeniu, informacji dostarczającej wiedzy jak i można wejść w interakcję z serwisem w zależności od aktualnego stanu aplikacji.

Jednym z elementów implementacji jest generowanie w odpowiedzi metadanych (tu linków z opisem) adekwatnych do stanu danych. W ćwiczeniu dodamy dodatkowy atrybut statusu elementu listy i w zależności od niego będą generowane różne odpowiedzi.

Niech Person może być w stanie nieaktywności NOT_ACTIVE (np. chory lub na urlopie), w stanie dostępności ACTIVE (możliwy do wynajęcia) lub wynajęty HIRED.

Zakładamy, że osobę można wynająć (*hire*) tylko jeśli jest ACTIVE, a zrezygnować z wynajmu (*vacate*) tylko jeśli jest HIRED. Taką informację dostarczymy w odpowiednio implementując odpowiedzi w formancie HAL na żądania (dla uproszczenia będą uwzględnione tylko dwa statusy: ACTIVE i HIRED).

- Dodaj do klasy Person pole statusu typu

```
enum PersonStatus {  
    ACTIVE,  
    NOT_ACTIVE,  
    HIRED  
}
```

czyli:

```
public class Person {  
    ...  
    private PersonStatus status;  
    ...  
}
```

oraz odpowiednie getter i setter.

- Dodaj w projekcie klasę wyjątku **ConflictEx** – podobna jak wcześniejsze klasy ale dla sygnalizacji błędu 409 Conflict
- Zdefiniuj w kontrolerze wyjątków handler zwracający odpowiednią odpowiedź w formacie RFC7807 z odpowiednim statusem:

```
@ResponseBody  
@ExceptionHandler(ConflictEx.class)  
@ResponseStatus(HttpStatus.CONFLICT)  
ResponseEntity<?> ConflictHandler(ConflictEx e) {  
    ...  
}
```

- Zdefiniuj w kontrolerze dwie dodatkowe metody:

```
@PatchMapping("/persons/{id}/hire")  
public ResponseEntity<?> hirePerson(@PathVariable int id) {...}
```

oraz

```
@PatchMapping("/persons/{id}/vacate")  
public ResponseEntity<?> vacatePerson(@PathVariable int id) {...}
```

Zwróć uwagę na zastosowaną regułę REST:

dla operacji na danych w URI najpierw jest ścieżka do zasobu, potem ew. dalsze elementy

Tu, do modyfikacji części danych stosujemy metodę PATCH (nie jest to konieczne wymagane, w implementacji możemy uprościć operacje i wywoływać zamianę całego elementu listy)

- W metodzie **hirePerson(...)**
 - pobierz element z repozytorium i sprawdź jego status
 - jeśli status jest ACTIVE to
 - zmień status na HIRED
 - zwróć ResponseEntity ze statusem HttpStatus.OK oraz body z linkami:
 - selfRel – do samego elementu
 - "vacate" – do rezygnacji z wynajmu – czyli:


```
LinkTo(methodOn(PersonController.class).vacatePerson(thePerson.getId()))
                    .withRel("vacate")
```
 - "list all" – do całej listy
 - Czyli z wynajmu można zrezygnować, ale osoby nie można wynająć ani usunąć (zakładamy że wynajęcie blokuje usuwanie)
 - jeśli status jest inny to wyrzucamy wyjątek konfliktu operacji:


```
throw new ConflictEx("You CAN'T hire a person with status "
                    +thePerson.getStatus());
```
- Zdefiniuj analogicznie metodę **vacatePerson(...)** ale w body zamieść linki
 - **selfRel** – do samego elementu
 - **"hire"** – analogicznie ale do wynajęcia (wolna osobę można ponownie wynająć)
 - **"delete"** – do usuwania
 - **"list all"** – do całej listy
- W metodzie **getPerson(...)** zmodyfikuj kod tak aby zwracana reprezentacja danych była adekwatna do stanu (statusu osoby), t.j:
 - Najpierw zdefiniuj zmienną typu EntityModel<T>


```
EntityModel<Person> em = EntityModel.of(thePerson);
```
 - Dodaj do **em** link **withSelfRel()**
 - Sprawdź bieżący status osoby i odpowiednio dodaj linki:
 - dla ACTIVE linki **"delete"** i **"hire"**
 - dla HIRED link **"vacate"**
 - na końcu dodaj link "list all"

Dla uproszczenia nie modyfikujemy zwracanych reprezentacji danych we wszystkich przypadkach. Docelowo serwis powinien uwzględniać wskazane własności we wszystkich przypadkach.

- Uruchom serwis i przetestuj działanie z użyciem programu Postman.
Wyniki operacji powinny być podobne do następujących:

a) `getPerson` lub `vacatePerson`

```
{
  "id": 2,
  "name": "Beata Babacka",
  "age": 10,
  "status": "ACTIVE",
  "_links": {
    "self": {
      "href": "http://localhost:8080/persons/2"
    },
    "delete": {
      "href": "http://localhost:8080/persons/2"
    },
    "hire": {
      "href": "http://localhost:8080/persons/2/hire"
    },
    "list_all": {
      "href": "http://localhost:8080/persons"
    }
  }
}
```

b) `hirePerson`

```
{
  "id": 2,
  "name": "Beata Babacka",
  "age": 10,
  "status": "HIRED",
  "_links": {
    "self": {
      "href": "http://localhost:8080/persons/2"
    },
    "vacate": {
      "href": "http://localhost:8080/persons/2"
    },
    "list_all": {
      "href": "http://localhost:8080/persons"
    }
  }
}
```

c) `hirePerson` gdy jest już wynajeta

```
{
  "title": "CONFLICT",
  "status": 409,
  "detail": "You CAN'T hire a person with status HIRED"
}
```

4 Exercise – Part II

The detailed and final requirements for Part II are set by the teacher.

A. Develop or prepare to develop a program according to the teacher's instructions.