Projekt pt. ***„Jeden Uniwersytet – Wiele Możliwości. Program Zintegrowany”***

Cezary Siwoń

Java i Spring

Opracowanie na potrzeby projektu

Jeden Uniwersytet – Wiele Możliwości. Program Zintegrowany

Spring

Spring jest framework'iem, w którym programista uzupełnia kod wywoływany przez zbudowany szkielet aplikacji webowej (często razem z serwerem HTTP)

Ważne elementy

- kontener IoC (Inversion of Control)
- mechanizm odpowiedzialny za **dependency injection**
- część odpowiedzialną za konfigurację

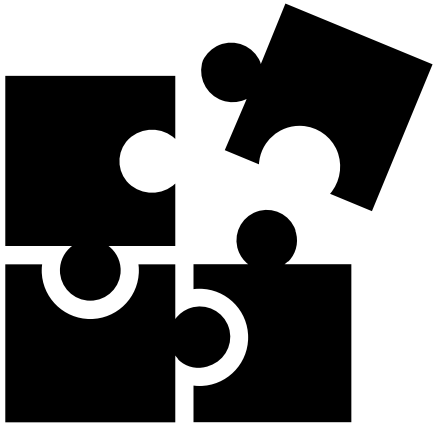
BIURO PROJEKTU
Uniwersytet Śląski w Katowicach
40-007 Katowice, ul. Bankowa 12, p. 2.9

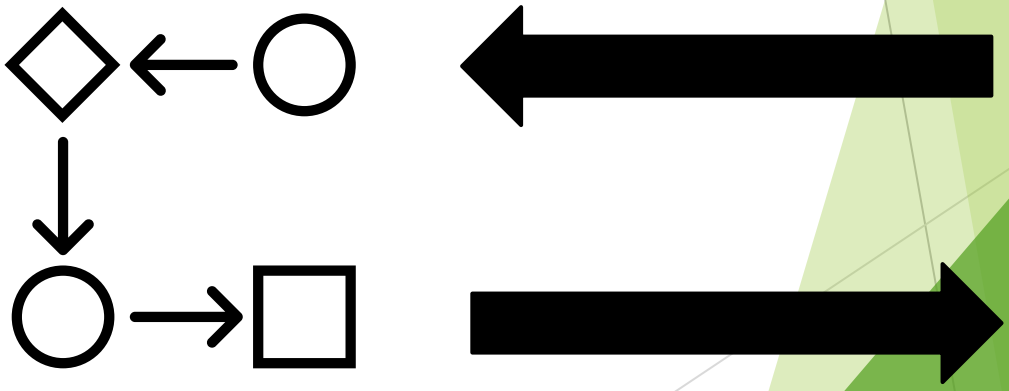
UNIWERSYTET ŚLĄSKI
BIURO DS. PROJEKTÓW OGÓLNOUCZELNIANYCH

www.zintegrowane.us.edu.pl
☎ 32 359 21 73

Co to jest framework?

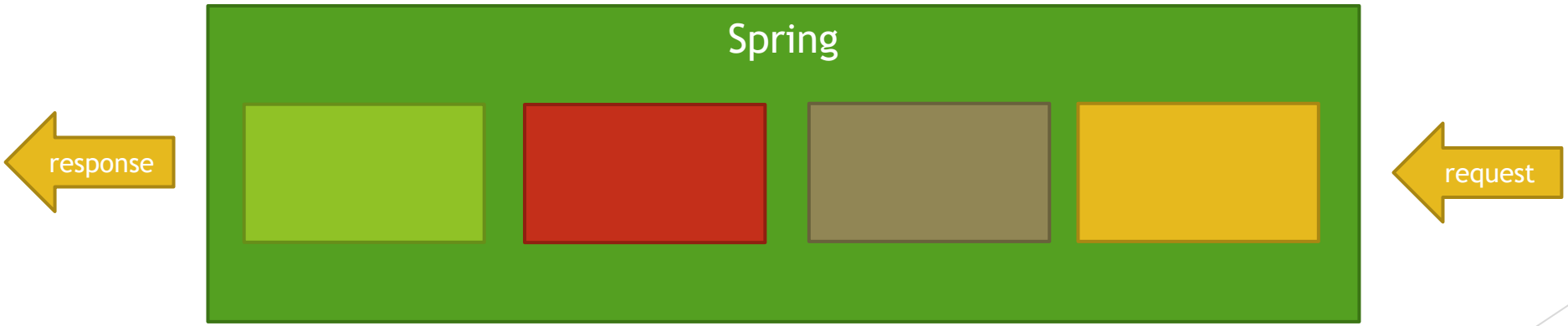
- Framework to pusta aplikacja, w której programista uzupełnia kod, który jest wywoływany przez szkielet.
- Framework może narzucać pewien przepływ przetwarzania.





Spring jako framework webowy

- Główne zastosowanie Spring'a to aplikacje webowe, ale nie tylko.
- W przypadku tworzenia aplikacji webowych działanie Spring można porównać do taśmy produkcyjnej, w które możemy dodawać kolejne etapy przetwarzania, konfigurowanie obecnych itd.



Podstawowe moduły

Spring umożliwia dodawanie modułów, które są zależnościami aplikacji. Można dowolnie konfigurować potrzebne składowe:

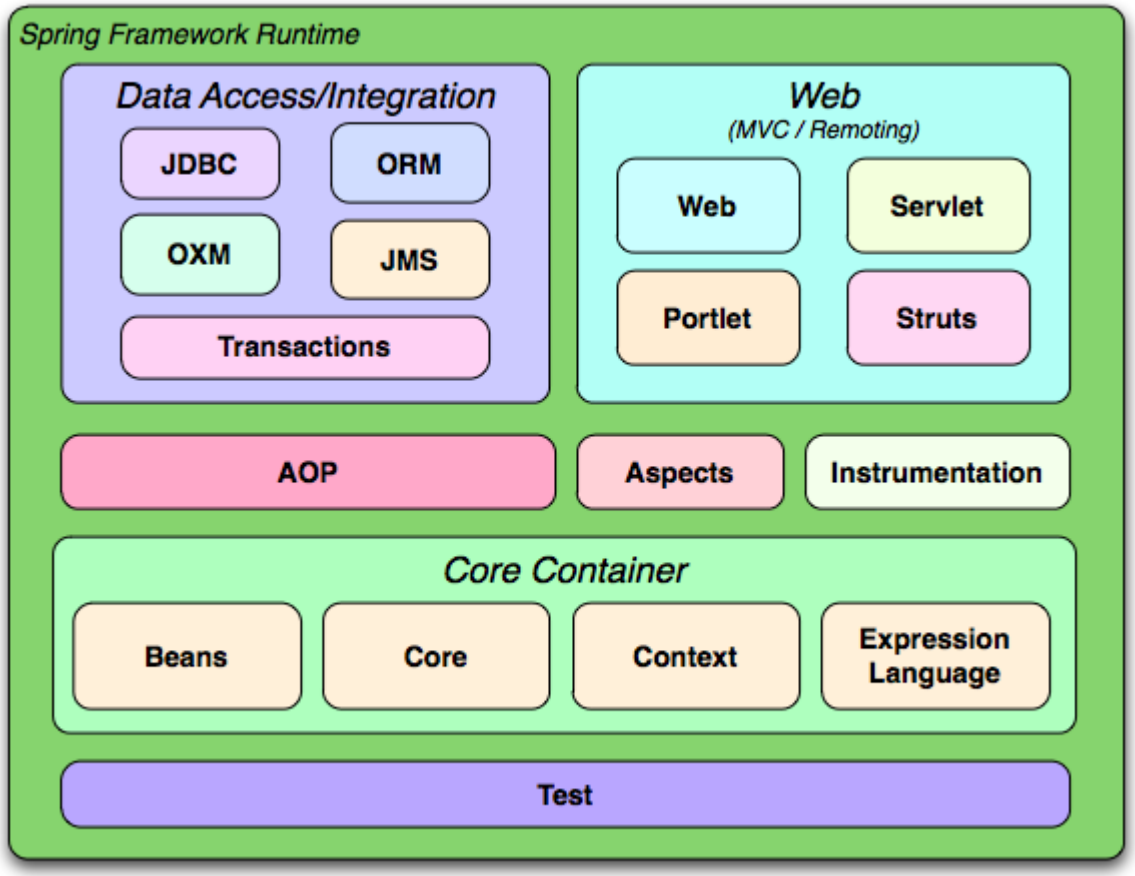
- spring-web
- spring-webmvc
- spring-core
- spring-context
- spring-aop
- jakarta.annotation-api
- tomcat-embed-core
- I wiele innych

BIURO PROJEKTU
Uniwersytet Śląski w Katowicach
40-007 Katowice, ul. Bankowa 12, p. 2.9

UNIWERSYTET ŚLĄSKI
BIURO DS. PROJEKTÓW OGÓLNOUCZELNIANYCH

www.zintegrowane.us.edu.pl
☎ 32 359 21 73

Spring - architektura



Bean

- To prosty obiekt z polami – właściwościami <Property>
- Dostęp do właściwości odbywa się metodami get<Property> i/lub set<Property>
- Klasa musi posiadać domyślny, bezargumentowy konstruktor
- Klasa powinna być serializowana

```
public class Person implements Serializable {
    private String name;

    private LocalDate birthDate;

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public LocalDate getBirthDate() {
        return birthDate;
    }
    public void setBirthDate(LocalDate birthDate) {
        this.birthDate = birthDate;
    }
    public int getAge(){
        return LocalDate.now().getYear() - this.birthDate.getYear();
    }
}
```

Spring Core

Bean

Bean to obiekt, który jest zarządzany przez kontener IoC. Rolą kontenera IoC jest zarządzanie beanami czyli ich tworzeniem i udostępnianiem w oznaczonych miejscach kodu. Programista nie jest bezpośrednio odpowiedzialny za tworzenie lub usuwanie takiego obiektu.

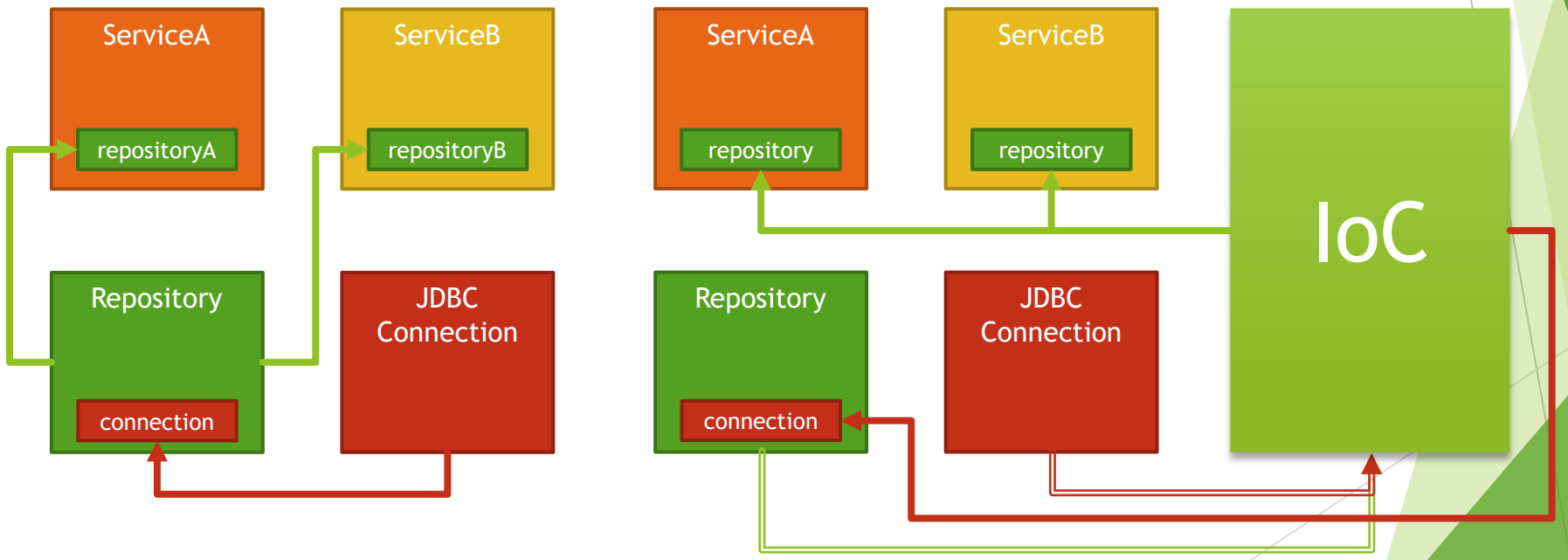
Beany są singletonami, czyli jest tworzona i udostępniana dokładnie jedna instancja klasy beanu.

Metody wskazania klasy, która posłuży do tworzenia beanów:

- Wpis do pliku xml (dzisiaj już nie stosowany)
- Dodanie adnotacji (tzw. stereotypów)
  - @Component
  - @Service
  - @Repository
  - @Controller
- Dodanie metody z adnotacją @Bean w klasie @Configuration, która zwraca obiekt będący beanem

Kontener IoC

Zadaniem kontenera IoC jest usunięcie problemu tworzenia potrzebnych obiektów bezpośrednio w klasach, które tego potrzebują.



Stereotypy

Wszystkie stereotypy dodają klasę do kontenera i działają bardzo podobnie:

- `@Component` - oznacza generyczny komponent
- `@Service` - funkcjonalnie nie różni się niczym od adnotacji `@Component`. Jest jedynie informacją, że klasa reprezentuje warstwę serwisową, tzn. taką, która zawiera logikę biznesową.
- `@Repository` - reprezentuje warstwę dostępu do bazy danych (tzn. klasę, która wykorzystuje, np. instancję EntityManager do wykonania zapytań do relacyjnej bazy danych). Od adnotacji @Component różni się jedynie tym, że w przypadku wystąpienia błędu na warstwie bazodanowej, możemy otrzymać bardziej szczegółowe informacje w wyjątkach.
- `@Controller` - działa tak jak `@Component`, ale bean, nad którym znajduję się ta adnotacja, trafia również do tzw. `WebApplicationContext`, tzn. do tej części kontekstu, która reprezentuje obiekty tworzące warstwę webową aplikacji.

Dependency injection

Wstrzykiwanie zależności to technika dostarczania obiektu przez kontener IoC w odpowiednim miejscu klasy.

Metody DI:

- Przez konstruktor – obiekt przekazywany jest w konstruktorze
- Przez pole – obiekt jest przypisywany do pola
- Przez seter – obiekt jest przekazywany przez argument setera

BIURO PROJEKTU
Uniwersytet Śląski w Katowicach
40-007 Katowice, ul. Bankowa 12, p. 2.9

UNIWERSYTET ŚLĄSKI
BIURO DS. PROJEKTÓW OGÓLNOUCZELNIANYCH

www.zintegrowane.us.edu.pl
☎ 32 359 21 73

Brak DI

```
public class Repository {
}

public class ServiceA {
    Repository repository;

    public ServiceA(Repository repository){
        this.repository = repository;

}

public class ServiceB {
    Repository repository;

    public ServiceB(Repository repository){
        this.repository = repository;

}
```

Ten kod dostarcza wymaganych obiektów do tworzenia następnych

```
public class Application {
    public static void main(String[] args) {
        Repository repository = new Repository();
        ServiceA serviceA = new ServiceA(repository);
        ServiceB serviceB = new ServiceB(repository);
    }
}
```

Dependency injection

```
@Repository
public class Repository {
}

@Service
public class ServiceA {
    Repository repository;
    public ServiceA(Repository repository){
        this.repository = repository;
}

@Service
public class ServiceB {
    Repository repository;
    public ServiceB(Repository repository){
        this.repository = repository;

}
```

Ten kod zostanie wygenerowany na podstawie adnotacji

```
public class Application {
    public static void main(String[] args) {
        Repository repository = new Repository();
        ServiceA serviceA = new ServiceA(repository);
        ServiceB serviceB = new ServiceB(repository);
    }
}
```

Wstrzykiwanie przez konstruktor

- W tej metodzie obiekt klasy beanu jest parametrem konstruktora.
- Nie jest wymagana adnotacja **`@Autowired`**, choć często się ją umieszcza zwyczajowo.
- Zalecana metoda DI. Wskazana jest deklaracja pola ze słowem **`final`**.

```
public class QuizController {
    final QuizService quizService;

    public QuizController(QuizServiceJpa quizService) {
        this.quizService = quizService;
    }
```

Wstrzykiwanie przez seter

- W tej metodzie obiekt klasy beanu jest parametrem setera.
- Wymagana jest adnotacja **`@Autowired`**.
- Kontener dostarczają wymaganego bean wywołuje metodę setera.

```
public class QuizController {
    QuizService quizService;

    @Autowired
    public void setQuizService(final QuizService quizService) {
        this.quizService = quizService;
    }
```

Wstrzykiwanie przez pole

- Wymagana jest adnotacja **`@Autowired`** nad polem.
- Obecnie nie jest już to zalecana metoda

```
public class QuizController {

    @Autowired
    QuizService quizService;
```

Zasięg - scope

- Beany są singletonami, czyli tworzona jest dokładnie jedna instancja klasy, która jest wstrzykiwana przez DI.
- Jeśli istnieje potrzeba stworzenia większej liczby instancji beanu to należy zmienić jego zasięg – scope:

```
@Scope("prototype")
```

- Domyślny zasięg większości beanów jest **`singleton`**
- Dla aplikacji webowej dostępne są inne zasięgi:
  - **`request`** – instancja dla każdego żądania HTTP
  - **`session`** – instancja dla nowej sesji HTTP
  - **`application`** – instancja dla aplikacji
  - **`websocket`** – instancja dla gniazda

Zasięg - scope

- Adnotacje do definiowania konkretnych zasięgów:
  - `@RequestScope`
  - `@SessionScope`
  - `@ApplicationScope`

BIURO PROJEKTU
Uniwersytet Śląski w Katowicach
40-007 Katowice, ul. Bankowa 12, p. 2.9

UNIWERSYTET ŚLĄSKI
BIURO DS. PROJEKTÓW OGÓLNOUCZELNIANYCH

www.zintegrowane.us.edu.pl
☎ 32 359 21 73

Kwalifikatory

- W przypadku kilku implementacji interfejsu wstrzykiwanego beanu należy wskazać, która implementacja ma zostać wstrzyknięta.
- Do tego służą adnotacje:
  - `@Qualifier`
  - `@Primary`
- W przypadku zastosowania obu adnotacji pierwszeństwo ma `@Qualifier`
- Można także wskazać konkretną implementację w przypadku wstrzykiwania przez konstruktor, określając typ implementacji wstrzykiwanego beana

@Qualifier
wybór po stronie klienta

```java
@Service("MemoryService")
public class MemoryContactService implements ContactService{
        ...
}
```

```java
@Service("ConstantService")
public class ConstantContactService implements ContactService{
        ...
}
```

```java
public ContactController(@Qualifier("MemoryService") ContactService contactService){
    ...
}
```

```java
@Qualifier("MemoryService")
@Autowired
private ContactService contactService;
```

@Primary
wybór po stronie implementacji

```
@Service("MemoryService")
@Primary
public class MemoryContactService implements ContactService{
        ...
}
```

```
@Service("ConstantService")
public class ConstantContactService implements ContactService{
        ...
}
```

```
@Autowired
private ContactService contactService;
```

Wybór w konstruktorze

```
@Service("MemoryService")
@Primary
public class MemoryContactService implements ContactService{
        ...
}
```

```
@Service("ConstantService")
public class ConstantContactService implements ContactService{
        ...
}
```

```
private final ContactService service;
public HomeController(MemoryContactService service) {
    this.service = service;
}
```

Lista beanów

- Możliwe jest też wstrzyknięcie listy pasujących beanów

```
@Autowired
private List<ContactService> services;
```

- Kolejność wstrzykniętych beanów można określić adnotacją @Order

```
@Service("MemoryService")
@Order(1)
public class MemoryContactService implements ContactService{
        ...
}
```

```
@Service("ConstantService")
@Order(2)
public class ConstantContactService implements ContactService{
        ...
}
```

Metoda main

- Aplikacja Spring'a jest typową aplikacją z metoda **`main`**
- Sama metoda **`main`** zawiera zazwyczaj jedną linijkę uruchamiająca całą aplikację, ale to co jest uruchamiane zależy do dołączonych zależności w pom.xml
- Adnotacja nad klasą jest połączeniem funkcji trzech adnotacji:
  - `@SpringBootConfiguration`
  - `@EnableAutoConfiguration`
  - `@ComponentScan`

```
@SpringBootApplication
public class QuizApplication extends SpringBootServletInitializer {

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

Metoda main

- `@SpringBootConfiguration`

Adnotacja `@SpringBootConfiguration` działa tak samo jak adnotacja `@Configuration`. Oznacza to, że w jej wnętrzu możemy definiować inne beany.

- `@EnableAutoConfiguration`

Adnotacja `@EnableAutoConfiguration`, włącza mechanizm automatycznej konfiguracji. Dzięki jej istnieniu aplikacja tworzy pewien kontekst, na podstawie konfiguracji i zależności w pliku pom.xml.

Metoda main

- `@ComponentScan`

Adnotacja `@ComponentScan` jest adnotacją informującą, w jakich pakietach powinny być szukane beany, które trafią do kontekstu. Domyślnie adnotacja ta w poszukiwaniu komponentów, skanuje aktualną paczkę (i paczki w jej wnętrzu).

Domyślnie skanowane są wszystkie klasy w bieżącym pakiecie aplikacji.

```
package pl.sda.quiz;
...

import java.util.List;
@SpringBootApplication
@ComponentScan(basePackageClasses = "pl.sda")
public class QuizApplication extends SpringBootServletInitializer {
   public static void main(String[] args) {
      SpringApplication.run(QuizApplication.class, args);
   }
```

Klasy w tym pakiecie nie są domyślnie dostępne w aplikacji.

Metoda main – wersja konsolowa

```
@SpringBootApplication
public class ConsoleApplication implements CommandLineRunner {

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

    @Override
    public void run(String... args) throws Exception {
        System.out.println("Hello from Spring");
    }
}
```

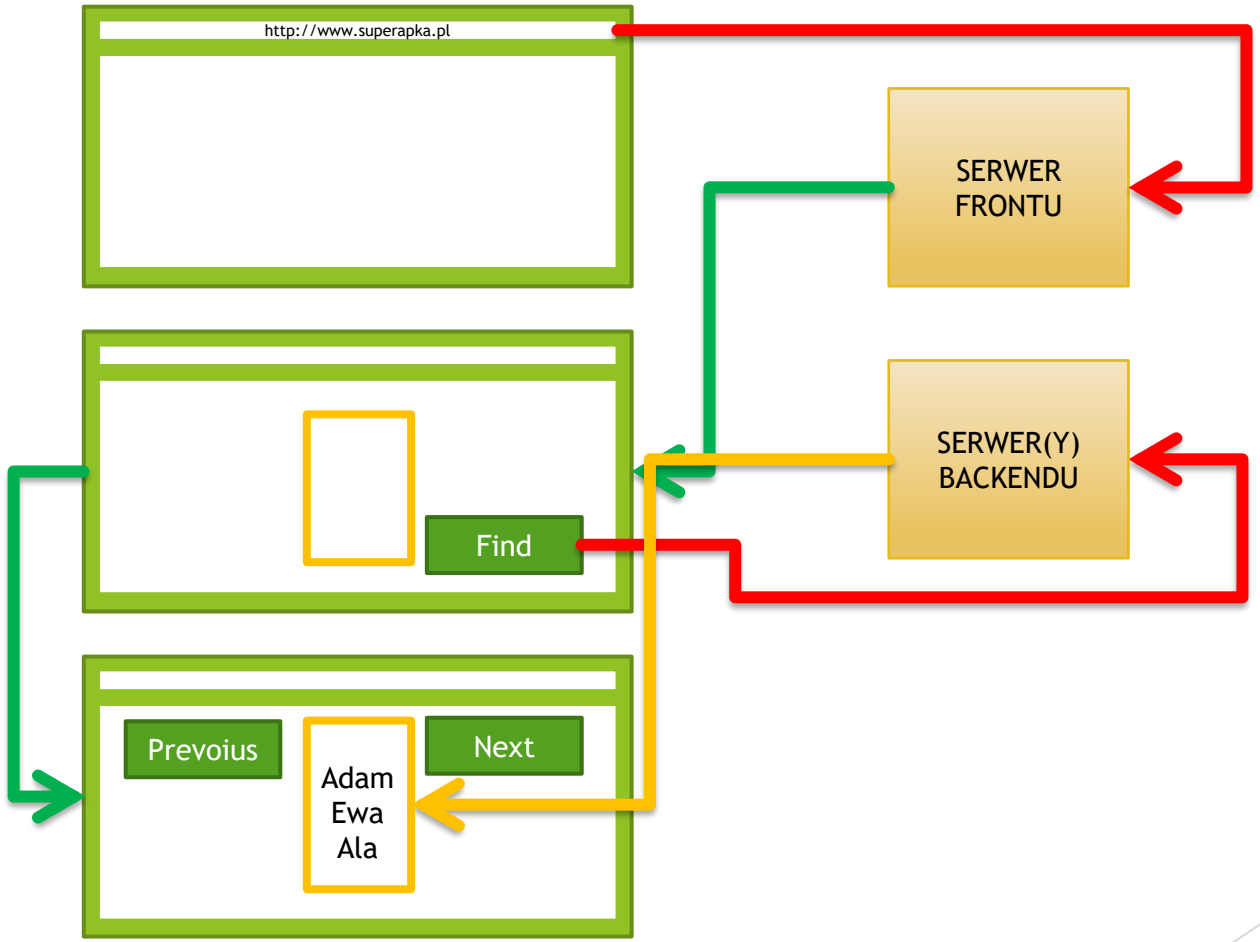
Interfejs, którego metoda jest wywoływana po uruchomieniu kontekstu Spring'a.

W application.properties należy dodać właściwość wyłączającą typ aplikacji webowej

```
spring.main.web-application-type=none
```

Spring REST

Architektura SPA – Single Page Application



BIURO PROJEKTU
Uniwersytet Śląski w Katowicach
40-007 Katowice, ul. Bankowa 12, p. 2.9

UNIWERSYTET ŚLĄSKI
BIURO DS. PROJEKTÓW OGÓLNOUCZELNIANYCH

www.zintegrowane.us.edu.pl
☎ 32 359 21 73

REST wstęp

- Serwisy REST zwracają tylko dane w otwartym formacie tekstowym JSON, ewentualnie XML
- Dane zwracane przez serwis REST pochodzą z jasno określonego zasobu
- REST posługuje się niepisanym standardem formułowanych żądań, w których:
  - nazwy ednpointów powinny odzwierciedlać zasób
  - metoda żądania HTTP określa rodzaj operacji na tym zasobie
- Metody żądań to:
  - `GET` – pobranie zasobu
  - `POST` – przesłanie w celu zapisania w serwisie zasobu
  - `PUT`, `PATCH` – edycja zasobu
  - `DELETE` – usunięcie zasobu

BIURO PROJEKTU
Uniwersytet Śląski w Katowicach
40-007 Katowice, ul. Bankowa 12, p. 2.9

UNIWERSYTET ŚLĄSKI
BIURO DS. PROJEKTÓW OGÓLNOUCZELNIANYCH

www.zintegrowane.us.edu.pl
☎ 32 359 21 73

REST

- **REST** – **Re**presentational **S**tate **T**ransfer – zmiana stanu poprzez reprezentację - styl architektury oprogramowania opierający się o zbiór wcześniej określonych reguł opisujących jak definiowane są zasoby, a także umożliwiających dostęp do nich. Został on zaprezentowany przez Roya Fieldinga w 2000 roku. (cytat z Wiki)

REST

Po ludzku

REST to metoda definiowania dostępu do zasobów, na których wykonujemy typowego CRUD’a:

- C – create: tworzenie czyli dodanie nowego, niezarejestrowanego zasobu np. użytkownika
- R – read: odczyt zasobu na podstawie identyfikatora (taki numerem użytkownika) lub parametrów wyszukiwania
- U – update: modyfikacja istniejącego zasobu na podstawie jego identyfikatora
- D – delete: usunięcie istniejącego zasobu na podstawie identyfikatora.

Konwencje REST

- **URI** endpointów **REST-owych** budowane są wg poniższych reguł**:**
- Zazwyczaj część restowa URI zaczyna się od słowa **api** np.:

  http://api.superapka.pl/api

  ale może tam znaleźć się bardziej rozbudowana ścieżka, wskazująca na sekcje, oddział itd.np.:

  http://api.example.com/api/louvre/

Konwencje REST

- Udostępniony zasób przez REST powinien być rzeczownikiem w liczbie mnogiej np.:

  http://api.superapka.pl/api/users

  przy czym zasoby mogą być zorganizowane hierarchicznie np.:

  http://api.superapka.pl/api/users/authors

- W URI stosujemy znak dywizu (myślnika) `-`, nie stosujemy znaku podkreślenia (podłoga, undescore)!
- Preferujemy małe litery
- Nie dodajemy rozszerzeń plików

Konwencje REST

- Drugim ważnym składnikiem REST-u jest stosowanie odpowiedniej metody HTTP do operacji CRUDu:
  - GET – pobieranie zasobu czyli tylko odczyt
  - POST – utworzenie nowego zasobu z nadaniem mu nowego identyfikatora (powinien zwrócić UIR wskazujące na nowo utworzony zasób)
  - PUT – modyfikacja istniejącego zasobu na podstawie identyfikatora (napisanie starego obiektu)
  - PATCH – modyfikacja jednej właściwości zasobu na podstawie identyfikatora
  - DELETE – usunięcie zasobu o podanym identyfikatorze

JSON

JSON – JavaScript Object Notation

Dane przesyłane w żądaniach i odpowiedziach REST-owych mają forma JSON, który jest wzorowany na notacji obiektów w JavaScript. Są jednak pewne różnice:

- Nazwy właściwości (pól) muszą być w podwójnych apostrofach (i tylko takie stosuje się w JSON):
  { ”field”: ”value”}
- Typami pól mogą być:
  - string - łańcuch np. ”abcd”
  - number - liczba np. 1234, 134.56
  - object - obiekt JSON {”date”:”2019.10.11”}
  - array - tablica, [1, 2, 4]
  - boolean - wartość logiczna np. true, false
  - null

JSON

- Nie wolno wstawiać przecinka za ostatnim polem (w JS dopuszczalne)
- Data w JSON to łańcuch, który trzeba sparsować do obiektu Date w JS
- Teoretycznie polem JSON nie może być metoda (funkcja), ale można kod funkcji zapisać do stringa i potem wykonać eval(kod), ale tego NIE ROBIMY!!!

Model dojrzałości Richardsona

- Brak definicji nowego protokołu, tunelowanie w ramach istniejącego protokołu
- Definicja hierarchicznego dostępu do zasobów
- Czasowniki HTTP definiują operacje na zasobach a status odpowiedzi informują o powodzeniu lub błędach
- Linki do zasobów - HATEOAS (Hypermedia As The Engine of Application State)

HTTP status

`200 OK`

- Powodzenie oznaczające w zależności od metody:
- GET: Zasób został wysłany
- PUT or POST: operacje zostały wykonane a wynik został wysłany.

`201 Created`

- Potwierdzenie wykonania żądań POST lub PUT.

`202 Accepted`

- Operacja została zaakceptowana ale jeszcze nie wykonana

`204 No Content`

- Brak zawartości w ciele odpowiedzi

HTTP status

`300 Multiple choices`

- Wiele możliwości

`301 Moved Permamently`

- Zasób trwale przeniesiony

`302 Found`

- Znaleziono – żądany zasób jest chwilowo dostępny pod innym adresem, a przyszłe odwołania do zasobu powinny być kierowane pod adres pierwotny

`303 See other`

- Zobacz inne – odpowiedź na żądanie znajduje się pod innym URI i tam klient powinien się skierować. To jest właściwy sposób przekierowywania w odpowiedzi na żądanie metodą POST.

...

HTTP status

`400 Bad Request`

- Niezrozumiałe ciało żądania

`401 Unauthorized`

- Wykonanie żądanie wymaga autoryzacji

`403 Forbidden`

- Klient nie ma praw do wykonania żądania

`404 Not Found`

- Brak takie zasobu

REST kontroler

- Kontroler serwisu REST można utworzyć adnotacjami:
  - `@Controller`
  - `@RestController` – kombinacja adnotacji `@Controller` i `@ResponseBody`
- Adnotacje metod kontrolera odpowiedzialne z obsługę endpointów
  - `GET` - `@GetMapping`
  - `POST` - `@PostMapping`
  - `PUT` - `@PutMapping`
  - `PATCH` - `@PatchMapping`
  - `DELETE` - `@DeleteMapping`
- Alternatywnie można stosować starszą notację:

```
@RequestMapping(method = RequestMethod.GET, path = "/api/hello")
```

@PathVariable

Częścią ścieżki jest zmienna, określająca instancję, numer lub identyfikator zasobu:

`host/api/quizzes/3`

dostęp do tej zmiennej umożliwia adnotacja umieszczona w metodzie kontrolera

```
@GetMapping("/api/quizzes/{id}")
@RequestBody
public Quiz findQuizById(@PathVariable(name = "id") final Long id) {
    return quizService.findById(id);
}
```

ResponseEntity

- Klasa generyczna reprezentująca odpowiedź żądania
- Zawiera wszystkie niezbędne informacje jak status, nagłówki i ciało, które można dowolnie ustawiać
- Typem parametrycznym jest klasa obiektu, który zostanie przesłany w ciele odpowiedzi:

```
@GetMapping("/quiz/json")
public ResponseEntity<SimpleQuiz> quizJson() {
    return ResponseEntity
            .status(HttpStatus.OK)
            .contentType(MediaType.APPLICATION_JSON)
            .body(SimpleQuiz.builder()
                    .question("Kiedy powstała Java")
                    .answer1("2000")
                    .build()
            );
}
```

@ResponseBody

Służy do generowania odpowiedzi z domyślnie ustawionym statusem i nagłówkami, adekwatnie do zawartości i metody żądania.

Metoda kontrolera z tą adnotacją zwraca w ciele odpowiedzi zwracany obiekt:

```
@GetMapping("/quiz/json")
@ResponseBody
public SimpleQuiz quizJson() {
    return SimpleQuiz.builder()
                    .question("Kiedy powstała Java")
                    .answer1("2000")
                    .build();
}
```

@RequestBody

- Metody kontrolera, które są odpowiedzialne za obsługę żądań typu POST, PUT lub PATCH, muszą odczytywać ciało żądania, w którym znajduje się przesyłany obiekt np. w postaci JSON.
- Adnotacja powoduje skonwertowania ciała do obiektu klasy opisującej dane w ciele żądania

```
@ResponseBody
@RequestMapping(value = "/api/quizzes/{index}", method = RequestMethod.PUT)
public void updateQuiz(@RequestBody final Quiz quiz, @PathVariable final Integer index) {

}
```

Ustawianie statusu

- Adnotacja nad metodą kontrolera pozwala na ustawienie domyślnego statusu wszystkich odpowiedzi

`@ResponseStatus(HttpStatus.CREATED)`

- Lepiej nie umieszczać tej adnotacji nad metodami zwracającymi `ResponseEntity`!

```
@ResponseBody
@ResponseStatus(HttpStatus.NO_CONTENT)
@RequestMapping(value = "/api/quizzes/{index}", method = RequestMethod.PUT)
public void updateQuiz(@RequestBody final Quiz quiz, @PathVariable final Integer index) {

}
```

Przykład – klasa obiektu

```
@Builder
@NoArgsConstructor
@AllArgsConstructor
@Data
public class Book implements Serializable {
    private String title;
    private String author;
}
```

Kontroler REST

```
@RestController
@RequestMapping("/api/v1/books")
public class RestBookController {
    public List<Book> books = new ArrayList<>(
            List.of(
                    Book.builder().title("Java").author("Bloch").build(),
                    Book.builder().title("Spring Boot").author("Pitoval").build(),
                    Book.builder().title("Hibernate").author("Vance").build()
            ));
...
```

Przykład – metody GET

```
@GetMapping("{id}")
public ResponseEntity<Book> getBook(@PathVariable Integer id) {
    try {
        return ResponseEntity.ok(books.get(id));
    } catch (IndexOutOfBoundsException e) {
        return ResponseEntity.of(Optional.empty());
    }
}
```

```
@GetMapping("")
public List<Book> getBooks() {
    return books;
}
```

Przykład – metoda POST

```
@PostMapping("")
public ResponseEntity<Book> addBook(@RequestBody Book book) {
    books.add(book);
    return ResponseEntity.ok(book);
}
```

Przykład – metoda DELETE

```
@DeleteMapping("{id}")
public ResponseEntity<Book> deleteBook(@PathVariable Integer id) {
    try {
        books.remove(id.intValue());
        return ResponseEntity.ok().build();
    } catch (IndexOutOfBoundsException e) {
        return ResponseEntity.of(Optional.empty());
    }
}
```

Przykład – metoda PUT

```
@PutMapping("/{id}")
public ResponseEntity<Book> putBook(@PathVariable Integer id, @RequestBody Book book){
    try {
        books.set(id, book);
        return ResponseEntity.ok(books.get(id));
    } catch (IndexOutOfBoundsException e) {
        return ResponseEntity.of(Optional.empty());
    }
}
```

Przykład – metoda Patch

```
@PatchMapping(path = "{id}", consumes = "application/merge-patch+json")
public ResponseEntity<Book> patchBook(@PathVariable Integer id, @RequestBody JsonPatch
docPatch) {
    final Book book = books.get(id);
    ObjectMapper mapper = new ObjectMapper();
    final JsonNode jsonNode = mapper.convertValue(book, JsonNode.class);
    try {
        final JsonNode pathed = docPatch.apply(jsonNode);
        final Book book1 = mapper.treeToValue(pathed, Book.class);
        return ResponseEntity.ok(book1);
    } catch (JsonPatchException | JsonProcessingException e) {
        System.out.println(e);
        return (ResponseEntity<Book>) ResponseEntity.internalServerError();
    }
}
```

Metoda patch

- Obsługa metody PATCH wymaga zależności

```
implementation 'com.github.java-json-tools:json-patch:1.13'
implementation 'com.fasterxml.jackson.datatype:jackson-datatype-jsr310:2.13.4:'
```

- Żądania powinny mieć ustawiony nagłówek `Content-Type`

`application/json-patch+json`

lub

`application/merge-patch+json`

Patch - operacje

Operacja	Opis
`add`	Dodaje lub ustawia właściwość, dodaje element tablicy
`remove`	Usuwa właściwość lub element tablicy
`replace`	Usuwa właściwość lub element tablicy dodając nową wartość w miejsce usuniętej
`move`	Usuwa właściwość lub element tablicy i przenosi wartość do miejsca przeznaczenia
`copy`	Dodaje nową właściwość lub element tablicy z wartością pobraną ze źródła
`test`	Zwraca status sukcesu jeśli wartość w ścieżce jest równa z dostarczoną wartością.

Patch – przykład add

Ustawienie nowej wartości właściwości AccountNumber

Dodanie nowej płatności do listy Payments, znak '-' oznacza wstawienie elementu na koniec listy

```
PATCH  http://localhost:5291/api/students/payments/1
Content-Type: application/json-patch+json

[{
    "op": "add",
    "path": "/AccountNumber",
    "value": "33333333"
  },
  {
    "op": "add",
    "path": "/payments/-",
    "value": 100
  }]
```

```
public class StudentPayments{
    public int studentId;
    public string accountNumber;
    public List<double> payments;
}
```

Przykład opisu remove

```
[
  {
    "op": "remove",
    "path": "/accountNumber"
  },
  {
    "op": "remove",
    "path": "/payments/0"
  }
]
```

Usunięcie właściwości powoduje ustawienie jej wartości na null.

Indeks usuwanego elementu tablicy. listy

Przykład operacji replace

```
[
  {
    "op": "replace",
    "path": "/accountNumber",
    "value": "11111"
  },
  {
    "op": "replace",
    "path": "/payments/0",
    "value": "10000"
  }
]
```

Zamiana wartości właściwości.

Indeks zmienianego elementu tablicy, listy

Nowa wartość

Przykład operacji move

```
[
  {
    "op": "move",
    "from": "/payments/0",
    "path": "/payments/1"
  }
]
```

Ścieżka źródła

Ścieżka przeznaczenia

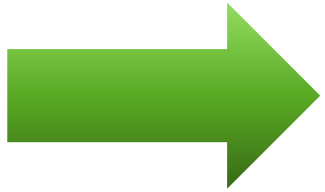
Przykład operacji copy

```
[
  {
    "op": "copy",
    "from": "/payments/2",
    "path": "/payments/0"
  }
]
```

Ścieżka źródła

Ścieżka przeznaczenia, kopiowana wartość jest wstawiana w podane miejsce!!!

```
{
  "StudentId": 1,
  "AccountNumber": "12345679000",
  "Payments": [
    10.0,
    20.0,
    30.0
  ]
}
```



```
{
  "StudentId": 1,
  "AccountNumber": "12345679000",
  "Payments": [
    30.0,
    10.0,
    20.0,
    30.0
  ]
}
```

Przykład operacji test

```
PATCH http://localhost:5291/api/Students/payments/1
Content-Type: application/json-patch+json

[
  {
    "op": "test",
    "path": "/payments/0",
    "value": "30"
  },
  {
    "op": "replace",
    "path": "/payments/0",
    "value": "3000"
  }
]
```

Test, czy pierwsza płatność jest równa 30

Jeśli test się powiedzie, to zostaną wykonane pozostałe operacje – zmiana płatności na 3000

Niepowodzenie testu kończy się błędem, a żadna z pozostałych operacji nie jest wykonywana!

```
{
  "StudentPayments": [
    "The current value '3000' at position '0' is not equal to the test value '30'."
  ]
}
```

Walidacja

- Należy dodać zależność:

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

- Adnotacje walidacji te są zdefiniowane w standardzie JSR-380, część z nich to:

`@Null, @NotNull, @Email, @Min, @Max, @Size, @AssertTrue, @AssertFalse`

Walidacja

- Adnotacje walidacji należy umieścić nad polami klasy obiektu walidowanego.
- Walidować należy wszystkie obiekty przychodzące z „zewnątrz” jak np. DTO (Data Transfer Object)
- Walidację przeprowadza się adnotacjami:
  - `@Valid`
  - `@Validated`

Walidacja - przykład

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Message {
    @NotNull
    private String from;
    @NotNull
    private String to;
    @NotNull
    @Length(min=5, max = 200)
    private String message;
}
```

Jeśli obiekt `message` nie spełnia warunków walidacji to zostanie zgłoszony wyjątek `MethodArgumentNotValidException`. Wyjątek można obsłuży dowolną metodą np. `ExceptionHandler`’em

```
@ResponseStatus(HttpStatus.BAD_REQUEST)
@PostMapping("/messages")
public Message newMessage(@Valid final Message message) {
    return new Message("from", "to","Hello");
}
```

Stronicowanie i HATEOAS

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

Stronicowanie wspiera repozytorium REST, które tworzy automatycznie kontroler REST z obsługą zasobu (ebooks):

```
@RepositoryRestResource(path = "ebooks", collectionResourceRel = "ebooks")
public interface PagedBookRepository extends PagingAndSortingRepository<Book, Long> {
    @RestResource(path = "author")
    Page<Book> findBookByAuthor(@Param("name") String author, Pageable p);
}
```

localhost:9000/ebooks?page=0&size=2

localhost:9000/ebooks/search/author?name=Bloch&sort=title

localhost:9000/ebooks/search/author?name=Bloch

Fundusze Europejskie
Wiedza Edukacja Rozwój

Projekt pt.

Stronicowanie i HATEOAS

BIURO PROJEKTU
Uniwersytet Śląski w Katowicach
40-007 Katowice, ul. Bankowa 12, p. 2.9

```
{
  "_embedded": {
    "ebooks": [
      {
        "title": "Java",
        "author": "Bloch",
        "_links": {
          "self": {
            "href": "http://localhost:9000/ebooks/0"
          },
          "book": {
            "href": "http://localhost:9000/ebooks/0"
          }
        }
      },
      {
        "title": "Effective Java",
        "author": "Bloch",
        "_links": {
          "self": {
            "href": "http://localhost:9000/ebooks/1"
          },
          "book": {
            "href": "http://localhost:9000/ebooks/1"
          }
        }
      }
    ]
  },
  "_links": {
    "first": {
      "href": "http://localhost:9000/ebooks?page=0&size=2"
    },
    "self": {
      "href": "http://localhost:9000/ebooks?page=0&size=2"
    },
    "next": {
```

localhost:9000/ebooks?page=0&size=2

Generowanie linków

- W zwykłych kontrolerach można też dodawać linki stosując poniższą bibliotekę

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

- Typami zwracanymi są CollectionModel lub EntityModel, która dodają linki do wysyłanej odpowiedzi z kolekcją obiektów lub obiektem

```
@GetMapping("")
public CollectionModel<Book> getBooks() {
    return CollectionModel.of(
     books,
     linkTo(methodOn(RestBookController.class).getBooks()).withSelfRel()
    );
}
```

Generowanie linków

```
@GetMapping("{id}")
public EntityModel<Book> getBook(@PathVariable Integer id) throws Exception {
    try {
        return EntityModel.of(
     books.get(id),
     linkTo(methodOn(RestBookController.class).getBook(id)).withSelfRel(),
     linkTo(methodOn(RestBookController.class).getBooks()).withRel("books")
                 );
    } catch (IndexOutOfBoundsException e) {
        throw new NotFoundException("Book not found");
    }
}
```

Obsługa wyjątków

- Zgłoszenie wyjątku w metodzie kontrolera zostaje przejęte przez odpowiedniego handlera, który obsługuje daną rodzinę wyjątków i generuje odpowiedź z informacją o błędzie

```
@ExceptionHandler(NotFoundException.class)
@ResponseStatus(HttpStatus.NOT_FOUND)
public Map<String, String> employeeNotFoundHandler(NotFoundException ex) {
    HashMap<String, String> result = new HashMap<String, String>();
    result.put("message", ex.getMessage());
    return result;
}
```

```
public class NotFoundException extends Exception{
    public NotFoundException(String bookNotFound) {
        super(bookNotFound);
    }
}
```

Obsługa wyjątków

- Obsługa lokalna wewnątrz klasy kontrolera

```
@ResponseStatus(HttpStatus.BAD_REQUEST)
@ExceptionHandler(InvalidQuizIdException.class)
public Error handleQuizException(final Exception exception) {
    return new Error(exception.getMessage());
}
```

```
@NoArgsConstructor
@AllArgsConstructor
@Data
public class Error {
    private String message;
}
```

- Obsługa globalna w osobnej klasie

```
@RestControllerAdvice
@ResponseBody
@Slf4j
public class QuizErrorHandler {
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    @ExceptionHandler(InvalidQuizIdException.class)
    public Error handleQuizException(final Exception exception) {
        log.debug("something bad has happened...");
        return new Error(exception.getMessage());
    }
}
```

Format danych REST –negocjacja

- Domyślnie dane są zwracane w postaci JSON
- Można nad metodą określić atrybutem `produces` formaty produkowane w kontrolerze:

```
@GetMapping( value = "/{id}", produces = {MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE})
public ResponseEntity<Book> readBook(@PathVariable long id){
    Book book = new Book();
    book.setId(id);
    book.setAuthor("Autor");
    book.setTitle("Tytuł");
    return ResponseEntity.of(id < 20 ? Optional.of(book) : Optional.empty());
}
```

- W klasie modelu należy podać adnotację XmlRootElement

```
@XmlRootElement
public class Book {
    private long id;
    ...
}
```

Format danych REST - negocjacja

- Klient określa rodzaj zwracanych danych nagłówkiem

```
Accept: application/xml
```

- Na podstawie powyższego nagłówka serwer zwraca dane w żądanym formacie

Spring Security

Spring Security

- Zależność dołączająca system zabezpieczeń:

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

Automatyczna konfiguracja mechanizmów:

- automatycznie generowany użytkownik wraz z hasłem
- wsparcie dla Basic Authentication dla REST API
- automatycznie wygenerowana strona umożliwiająca logowanie i wylogowanie
- włączone zabezpieczenie przed atakami CSRF (tzw. Cross-site request forgery)
- włączone zabezpieczenie przez atakiem Session Fixation

Security ustawienia

- Przy braku jawnej konfiguracji
  - Użytkownik - `user`
  - Hasło podane w logach np.:

```
Using generated security password: d14a5f60-6789-4e0b-870f-26ad01d6628e
```

- Konfiguracja w pliku `application.properties`:

```
spring.security.user.name=
spring.security.user.password=
```

Security - Basic Authentication

- Domyślny sposób uwierzytelniania
- W każdym żądaniu musi się znaleźć nagłówek w postaci:

```
Basic user:password
```

- Dane są enkodowane algorytmem Base64

Security konfiguracja

- Konfiguracja security polega na definicji klasy z andotacją `@Configuration` rozszerzającej `WebSecurityConfigurerAdapter`

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
                .httpBasic()
                .realmName("Realm")
                .and()
                .csrf()
                .disable()
                .headers()
                .frameOptions()
                .disable()
                .and()
                .authorizeRequests()
                .antMatchers("/api/quiz/**",
"/api/quizzes/**").hasRole("USER")
                .antMatchers("/api/register").permitAll()
                .antMatchers("/api/register/**").hasRole("USER")
                .antMatchers("h2-console/**").permitAll()
                .anyRequest().permitAll()
                .and();
    }
}
```

Security konfiguracja

- Domyślnie wszystkie ścieżki wymagają uwierzytelniania
- Metody budowniczego zabezpieczeń
  - `antMatchers` – określa dla ścieżki (lub dodatkowo metody HTTP) poziom dostępu, np.:
    - `authenticated` - wymaga autentykacji
    - `permitAll` – dostępny dla wszystkich
    - `hasRole` – dostępny po autentykacji dla użytkowników o podanej roli
  - Ponadto `antMatchers` pozwala zdefiniować specjalne znaki, które potrafią dopasować grupę ścieżek. Te znaki to:
    - $ - zastępuje jeden, dowolny znak
    - * - zastępuje jeden lub wiele znaków, do następnego znaku /
    - ** - zastępuje jeden lub wiele znaków

Security konfiguracja

- `mvcMatchers` - alternatywa dla antMatchers, która pozwala definiować ścieżki w stylu Spring MVC (tzn. wykorzystując dodatkowo klamry)
- `anyRequest` - definiuje wymagany poziom dostępu dla pozostałych, niezdefiniowanych ścieżek
- `formLogin` - logowanie przez formularz logowania (domyślną stronę lub zdefiniowaną przez FormLoginConfigurer)
- `logout` - wsparcie dla wylogowania
- `csrf` – konfigurowanie mechanizmu CSRF
- `headers` – konfigurowanie nagłówków powiązanych z warstwą bezpieczeństwa

Security
użytkownik w pamięci

Obiekt **AuthenticationManagerBuilder** może definiować listę użytkowników przechowywanych w pamięci za pomocą metody **inMemoryAuthentication**:

- zdefiniowanie nazwy użytkownika za pomocą metody **withUser**
- ustawienie hasła użytkownika za pomocą metody **password**
- zdefiniowania ról użytkownika wykorzystując metodę **roles**
- zdefiniować kolejnego użytkownika wykorzystując metodę **and**

Security
użytkownik w pamięci

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(final AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication()
                .withUser("user1").password("Admin_1").roles("ADMIN")
                .and()
                .withUser("user2").password("Admin_2").roles("SUPER_ADMIN");
    }
}
```

Security
hasła

- Hasła nie mogą być jawnie porównywane, muszą być haszowane za pomocą enkodera
- Haszowanie nie jest odwracalne, nie można odzyskać hasła z otrzymanego ciągu
- Do każdego haszowanego hasła dodawany jest losowy ciąg tzw. salt, który powoduje, że dwa identyczne hasła nie maja identycznych ciągów.

Security encoder

- Należy wskazać stosowany algorytm haszowanie definiując bean PasswordEncoder
- Dostępne algorytmy:
  - bcrypt BCryptPasswordEncoder
  - ldap LdapShaPasswordEncoder
  - MD5 MessageDigestPasswordEncoder("MD5")
  - noop NoOpPasswordEncoder
  - pbkdf2 Pbkdf2PasswordEncoder
  - scrypt SCryptPasswordEncoder
  - SHA-1 MessageDigestPasswordEncoder("SHA-1")
  - SHA-256 MessageDigestPasswordEncoder("SHA-256")
  - sha256 StandardPasswordEncoder
  - argon2 Argon2PasswordEncoder

```
@Bean
public PasswordEncoder encoder() {
    return new BCryptPasswordEncoder();
}
```

Security encoder

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(final AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication()
                .withUser("user1").password("{bcrypt}$2a$10$w6AoSvH7ezHwQomG19Oj
                 Oev3AZuu/UWkROR5CjW1fXZjvP/dy9mEq").roles("ADMIN")
                .and()
                .withUser("user2").password("{MD5}{WtFrETqej9qZsjAWIlXhM03PxD5xK
                 vdcpr3T9R/BHl8=}2aff96e65139a682af29072ffcef19d1").roles("SUPER_ADMIN");
    }
}
```

Serwis własnych użytkowników

```java
@Entity
@Table(name = "users")
public class User implements UserDetails, Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @EqualsAndHashCode.Include
    private long id;

    ...
    @Override
    public boolean isAccountNonExpired() {
        return enabled;
    }
    @Override
    public boolean isAccountNonLocked() {
        return enabled;
    }
    @Override
    public boolean isCredentialsNonExpired() {
        return enabled;
    }
    @Override
    public boolean isEnabled() {
        return enabled;
    }
    public boolean isVerified() {
        return verified;
    }
}
```

Encja opisująca naszych użytkowników musi implementować interfejs UserDetails.

Użytkownik zostanie zalogowany jeśli te metody zwracają true.

Serwis własnych użytkowników

```
@Repository
public interface UserRepository extends CrudRepository<User, Long> {
    User findByEmail(String username);
}
```

Repozytorium encji użytkowników.

Serwis własnych użytkowników

```
@Service
public class UserDetailsServiceJpa implements UserDetailsService {
    private final UserRepository userRepository;

    @Autowired
    public UserDetailsServiceJpa(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        return Optional.ofNullable(userRepository.findByEmail(username))
                .orElseThrow(() ->
         new UsernameNotFoundException(String.format("Użytkownik %s nie został znaleziony ", username)));
    }
}
```

Serwis dostarczający informacji o użytkownikach musi implementować intefejs UserDetailsService, w którym jest tylko jedna metoda ładująca użytkownika na podstawie nazwy użytkownika.

Serwis własnych użytkowników

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    private final UserDetailsService userDetailsService;

    @Autowired
    public SecurityConfig(UserDetailsServiceJpa userDetailsService) {
        this.userDetailsService = userDetailsService;
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth
                .userDetailsService(userDetailsService)
                .passwordEncoder(encoder());
    }
```

Klasa konfiguracyjna posiada dostęp do serwisu użytkowników przez wstrzyknięty serwis.

W tej metodzie wskazujemy, że Spring ma korzystać z naszego serwisu użytkowników.

Dostęp do użytkownika w kontrolerze

```
@PostMapping("/add")
public String saveProduct(@AuthenticationPrincipal User user) {
  product.setPublisher(user);
  productService.create(product);
  return "redirect:/products/index";
}
```

Metod kontrolera, która wymaga zalogowanego użytkownika.

Najlepszy dostęp do zalogowanego użytkownika zapewnia dodanie parametru z adnotacją @AuthenticationPrincipal.

Uzyskanie dostępu do zalogowanego użytkownika

Pozostałe metody na uzyskanie użytkownika:

- Wstrzyknięcie obiektu `Principal` do metody kontrolera.
- Wstrzyknięcie obiektu `Authentication` do metody kontrolera.
- Użycie `SecurityContextHolder`, aby pobrać kontekst bezpieczeństwa.

Podstawowa konfiguracja dla aplikacji REST

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
            .httpBasic()
            .realmName("realm")
            .and()
            .csrf()
            .disable()
            .headers()
            .and()
            .authorizeRequests()
            .antMatchers(HttpMethod.POST, "/api/**").hasRole("USER")
            .antMatchers(HttpMethod.DELETE, "/api/**").hasRole("USER")
            .antMatchers(HttpMethod.PUT, "/api/**").hasRole("USER")
            .antMatchers(HttpMethod.PATCH, "/api/**").hasRole("USER")
            .antMatchers(HttpMethod.GET, "/api/**").permitAll()
            .and()
            .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
}
```

Włączenie autoryzacji typu BASIC

Wyłączenie generowania tokenów zabezpieczających przez CSRF

Wyłączenie tworzenia sesji gdyż REST jest bezstanowy.

Spring Data

Moduły

- Spring Data commons
- Spring Data JDBC
- Spring Data JDBC Ext
- Spring Data JPA
- Spring Data LDAP
- Spring Data MongoDB
- Spring Data Redis
- Spring Data REST
- Spring Data for Apache Cassandra
- oraz wiele innych także dostarczanych przez społeczność

JDBC Template - konfiguracja

```
@Configuration
public class AppJdbcConfiguration {

    @Bean
    public DataSource dataSource(){
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("org.hsqldb.jdbc.JDBCDriver");
        dataSource.setUrl("jdbc:hsqldb:file/c:\\data\\books");
        dataSource.setUsername("sa");
        dataSource.setPassword("");
        return dataSource;
    }
    ...
}
```

```
@Configuration
public class AppJdbcConfiguration {
    @Bean
    public DataSource dataSource(){
        return new EmbeddedDatabaseBuilder()
                .setType(EmbeddedDatabaseType.HSQL)
                .setName("books")
                .addScript("/sql/schema.sql")
                .addScript("/sql/data.sql")
                .build();
    }
    @Bean
    public JdbcTemplate jdbcTemplate(DataSource dataSource) {
        return new JdbcTemplate(dataSource);
    }
}
```

JDBC – przykładowe skrypty i model

/sql/schema.sql

```
create table books (
    id int identity primary key,
    title varchar(30),
    author varchar (20)
);
```

/sql/data.sql

```
insert into books(title, author) values('Bloch','Java');
insert into books(title, author) values('Freeman','C#');
```

```
@Data
@Builder
public class Book {
    private int id;
    private String title;
    private String author;
}
```

JDBC Template - select

```
@Repository
public class BookRepository {
    private final JdbcTemplate jdbcTemplate;

    public BookRepository(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    public List<Book> findAll() {
        return jdbcTemplate.query("select id, title, author from books", (row, n) ->
                {
                    return Book.builder()
                            .id(row.getInt("id"))
                            .author(row.getString("author"))
                            .title((row.getString("title")))
                            .build();
                }
        );
    }
}
```

JDB Template - insert

```
public long insert(final Book book) {
    return jdbcTemplate.update("insert into books (title, author) values(?, ?)"
                               , book.getTitle(), book.getAuthor());
}
```

```
public long insertBook(final Book book){
    Map<String, Object> values = new HashMap<>();
    values.put("title", book.getTitle());
    values.put("author", book.getAuthor());
    return new SimpleJdbcInsert(jdbcTemplate)
            .withTableName("books")
            .execute(values);
}
```

Jdbc Template – delete i select

```
public int delete(int id){
    return jdbcTemplate.update("delete from BOOKS where id = ?", p -> p.setInt(1, id));
}
```

```
public Book findById(int id) {
    return this.jdbcTemplate.queryForObject(SELECT_BY_ID,
            (row, n) -> {
                return Book.builder()
                        .id(row.getInt("id"))
                        .title(row.getString("title"))
                        .author(row.getString("author"))
                        .build();
            },
            new Object[]{id});
}
```

```
public Map<String ,Object> findBy(int id){
    return jdbcTemplate.queryForMap("select * from books where id = ?", id);
}
```

JdbcTemplate – Named Parameters

```
public int countByAuthor(String author){
    var parameters = new MapSqlParameterSource();
    parameters.addValue("author", author);
    NamedParameterJdbcTemplate template = new NamedParameterJdbcTemplate(jdbcTemplate);
    return template.queryForObject("select count(*) from books where author=:author",
                        parameters, Integer.class);
}
```

Spring JPA

Tradycyjna konfiguracja persistence.xml

To jest tzw. jednostka perzystencji, która jest parametrem fabryki

META-INF/persistence.xml

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
             version="2.0">
    <persistence-unit name="users" transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
        <properties>
            <property name="hibernate.hbm2ddl.auto" value="create"/>
            <property name="javax.persistence.jdbc.driver" value="org.hsqldb.jdbc.JDBCDriver"/>
            <property name="javax.persistence.jdbc.user" value="sa"/>
            <property name="javax.persistence.jdbc.password" value=""/>
            <property name="javax.persistence.jdbc.url" value="jdbc:hsqldb:mem:users"/>
            <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect"/>
            <property name="hibernate.id.new_generator_mappings" value="false"/>
        </properties>
    </persistence-unit>
</persistence>
```

SPRING JPA – zależności i konfiguracja

pom.xml

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

update: zmiana struktury zgodnie z definicją encji
create: tworzenie schematu bez wywoływania drop do zakończeniu
create-drop: tworzenie schematu i usuwanie po wyjściu
none: brak zmian

application.properties

```
spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://${MYSQL_HOST:localhost}:3306/db_example
spring.datasource.username=springuser
spring.datasource.password=ThePassword
spring.datasource.driver-class-name =com.mysql.jdbc.Driver
#spring.jpa.show-sql: true
```

Zmienna środowiskowa przechowująca adres serwera bazy danych

Model

```java
@Entity
@Table(name = "users")
@Builder
@Getter
@Setter
@EqualsAndHashCode(onlyExplicitlyIncluded = true)
@ToString
@NoArgsConstructor
@AllArgsConstructor
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @EqualsAndHashCode.Include
    private long id;

    @Column(name = "name", nullable = false)
    private String username;

    private String email;

    @CreationTimestamp
    private Timestamp created;
}
```

Adnotacje Lombok’a

Adnotacje JPA

Adnotacja Hibernate

Repozytoria

- Spring oferuje wsparcie w postaci generowanych klas repozytoriów na podstawie interfejsu generycznego.
- Dostawca persystencji jest tworzony przez Springa na podstawie konfiguracji. Jednocześnie zablokowany jest dostęp do EntityManagera, udostępniając tylko jego wersję proxy, która nie obsługuje transakcji!
- Klasy repozytoriów
  - `CrudRepository<T, ID>`
  - `PagingAndSortingRepository<T, ID>`
  - `JpaRepository<T, ID>`

JPQL

JPQL - Jakarta Persistence Query Language

- Wersja SQL posługująca się encjami i ich polami zamiast tabel i kolumn
- Obowiązkowo encje w zapytaniach muszą posiadać aliasy
- Można korzystać z polimorfizmu np. w miejscu encji bazowej można wstawić encję pochodną.
- Wynikiem zapytań mogą być tylko encje, ich kolekcje lub typy proste.

JPQL - Przykłady

```
SELECT b FROM Book b

FROM Book

SELECT b FROM Book b WHERE b.title = 'H2G2’

SELECT CASE
    b.editor WHEN 'Apress’ THEN b.price * 0.5 ELSE b.price * 0.8
    END
FROM Book b

SELECT c FROM Customer c WHERE c.age > 18 ORDER BY c.age DESC
```

Repozytorium – zapytania natywne

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

    List<Book> findBookByAuthor(String author);

    @Query(value = "select count(*) from books where author = ?1", nativeQuery = true)
    int countBooksByAuthor(String author);

    @Query(value = "select title, author from books", nativeQuery = true)
    List<Map<String ,Object>> getTitlesAndAuthors();
}
```

PagingAndSortingRepository

```
@Repository
public interface TodoPagingRepository extends PagingAndSortingRepository<EntityTodo, Long> {
}
```

```
final TodoPagingRepository pagingRepository;
```

```
final Page<EntityTodo> page = pagingRepository.findAll(PageRequest.of(0, 10));
```

Serwisy

Serwisy

- Serwisy to klasy wykonujące logikę biznesową, czyli funkcje ważne z punktu widzenia biznesu czy koncepcji aplikacji.
- Serwis powinien posługiwać się klasami domenowymi
- Stereotyp do definiowania serwisów

@Service

- Do tworzenia serwisu nie jest wymagane definiowanie interfejsu, ale zazwyczaj interfejs jest tworzony

Serwis - przykład

```java
public interface BookService {
    Book save(BookDto dto);
    List<Book> findAll();
}
```

```java
@Service
public class JpaBookService implements BookService{
    private final BookRepository bookRepository;

    public JpaBookService(BookRepository bookRepository) {
        this.bookRepository = bookRepository;
    }
    @Override
    public Book save(BookDto dto) {
        return BookMapper.mapFromEntity(bookRepository.save(BookMapper.mapToEntity(dto)));
    }
    @Override
    public List<Book> findAll() {
        return bookRepository.findAll()
                .stream()
                .map(BookMapper::mapFromEntity)
                .collect(Collectors.toList());
    }
}
```

Data Transfer Object

Data Transfer Object

- Dane przychodzące w żądaniu HTTP często nie są wystarczające na utworzenie obiektu domenowego
- Dane te mają też inne formaty niż wymagane w obiekcie domenowym
- Aby Spring otrzymane dane żądania przekształcił na obiekt definiuje się klasę tzw. Data Transfer Object, która zawiera tylko pola żądania

```
title:
date:
publisher:
```

```
class BookDto{
    String title;
    String date;
    String publisher;
}
```

```
class Book{
    long id;
    String title;
    LocalDate date;
    Publisher publisher;
}
```

Mapper

- Rolą mapera jest przepakowanie danych z klasy Dto do klasy domenowej lub klasy encji.
- Można tworzyć własne mappery np. jako klasę z metoda statycznymi, które przyjmują obiekt przekształcany i zwracają obiekt docelowy:

```
public static BookEntity mapToEntity(BookDto dto);
```

- Można też stosować gotowe rozwiązania jak np. MapStruct

Mapper - przykład

```java
@Data
@Builder
public class BookDto {
    @Length(min = 1, max = 120,
            message = "Tytuł nie może być pusty i dłuży od 120 znaków!")
    private String title;
    @Length(min = 3, max = 15)
    private String author;
}
```

```java
@Entity
@Table(name = "book")
@Builder
@Getter
@Setter
@ToString
@NoArgsConstructor
@AllArgsConstructor
public class EntityBook {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    @Column(nullable = false)
    private String title;

    @Column(nullable = true)
    private String author;
}
```

```java
public class BookMapper {
    public static EntityBook mapToEntity(BookDto dto){
        return EntityBook.builder()
                .author(dto.getAuthor())
                .title(dto.getTitle())
                .build();
    }

    public static Book mapFromEntity(EntityBook entity){
        return Book.builder()
                .id(entity.getId())
                .title(entity.getTitle())
                .author(entity.getAuthor())
                .build();
    }
}
```

```java
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class Book {
    private long id;
    private String title;
    private String author;
}
```

Konfiguracja

Pliki konfiguracyjne

- Spring obsługuje pliki konfiguracyjne w formatach:
  - properties – application.properties
  - Yaml - application.yaml
  - XML - persistence.xml

Własne dane konfiguracyjne

application.properties

```
page.size=20
page.start=1
theme=standard
```

```
@Configuration
public class AppConfig {
    @Value("${page.size}")
    public int pageSize;

    @Value("${page.start}")
    public int pageStart;
}
```

```
@EnableWebMvc
public class MyApplication {
   final AppConfig config;


   ...

   config.pageSize
}
```

Klasa właściwości

application.properties

```
pl.sdacademy.group.prop-a=prop_A_value
pl.sdacademy.group.prop-b=17
```

```
@Component // lub //@EnableConfigurationProperties(PropertiesGroup.class)
@ConfigurationProperties(prefix = "pl.techelf.group")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class PropertiesGroup {
    private String propA;
    private Integer propB;
}
```

```
@Controller
public class HomeController {
    @Autowired
    PropertiesGroup group;

    @GetMapping("/")
    @ResponseBody
    public String home(@RequestParam String name){
        return "Hello " + name + " " + group.getPropA();
    }
}
```

Konfiguracja na podstawie zmiennych środowiskowych

application.properties

```
spring.datasource.user=${DATABASE_USER}
spring.datasource.password=${DATABASE_PASSWORD}
```

```
C:\>SET DATABASE_USER=admin
C:\>SET
DATABASE_USER=admin
```

Zmienna środowiskowa przechowująca nazwę użytkownika lub hasło

BIURO PROJEKTU
Uniwersytet Śląski w Katowicach
40-007 Katowice, ul. Bankowa 12, p. 2.9

UNIWERSYTET ŚLĄSKI
BIURO DS. PROJEKTÓW OGÓLNOUCZELNIANYCH

www.zintegrowane.us.edu.pl
☎ 32 359 21 73

Profile

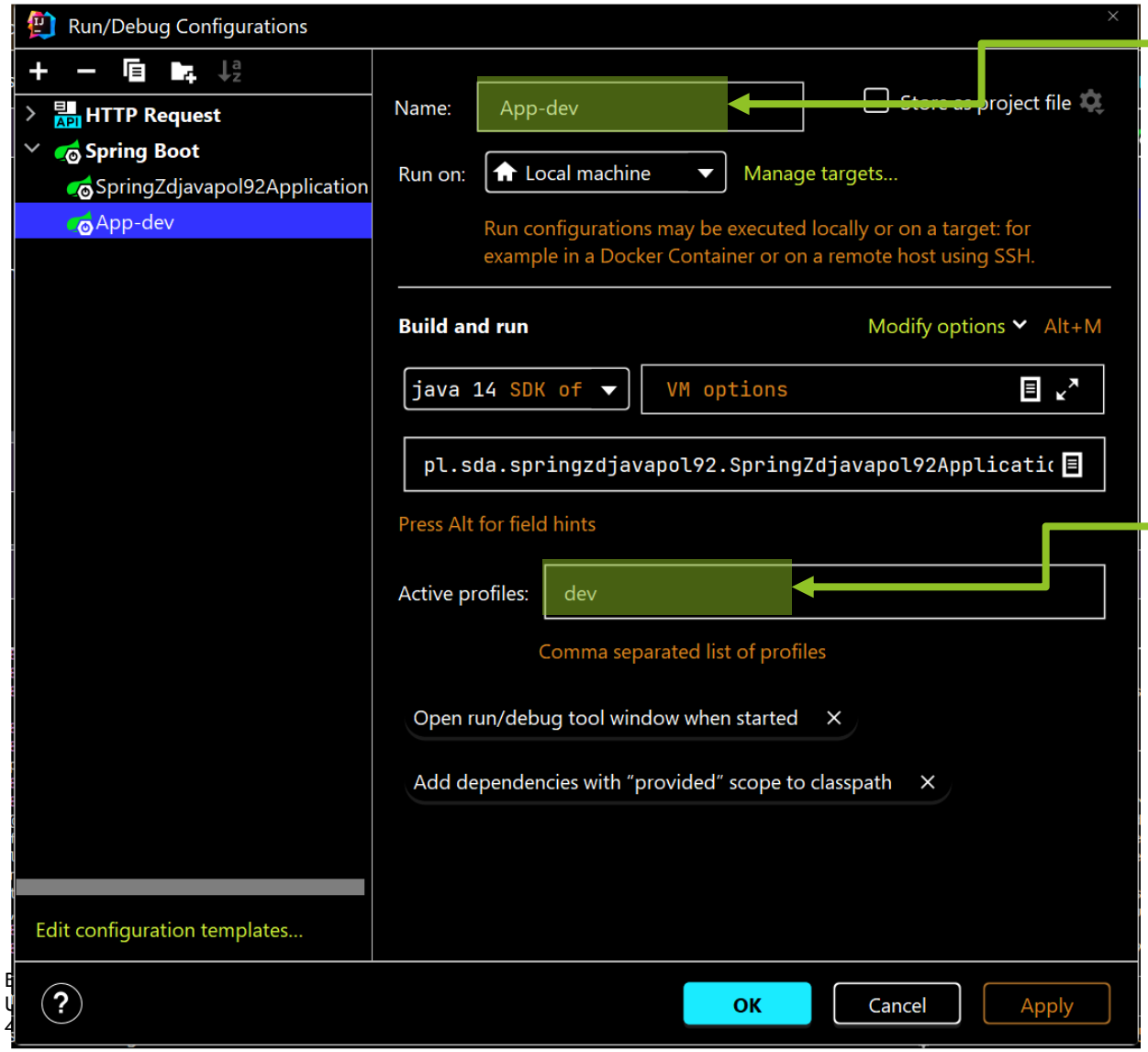
- Można tworzyć wiele konfiguracji i zapisywać je w osobnych plikach, które tworzą profile np.:

```
application-dev.properties    - dev
application-prod.properties   - prod
application.properties        - default
application-local.properties  - local
```

- Podczas kompilacji i uruchomienia aplikacji można wskazać z którego profilu korzystamy

```
-Dspring,profiles.active=dev
```

*Projekt pt. **„Jeden Uniwersytet – Wiele Możliwości. Program Zintegrowany”***



Nazwa konfiguracji uruchomieniowej

Aktywny profil

Wstrzykiwanie implementacji w zależności od profilu

- Tworząc implementacje tego samego interfejsu można wskazać która powinna być wstrzykiwana w zależności od profilu
- Adnotacja `@Profile({”profil1”, ”profil2”,..})` wskazuje, dla których profili kontener powinien wstrzyknąć daną implementację.

Wstrzykiwanie implementacji w zależności od profilu

```java
@Service
@Profile({"dev"})
public class DevBookService implements BookService{

    private Map<Long, EntityBook> books = new HashMap<>();
    private AtomicLong indexies = new AtomicLong(0);

    @Override
    public Book save(BookDto dto) {
        ...
    }

    @Override
    public List<Book> findAll() {
        ...
    }
}
```

```java
@Service
@Profile({"default", "prod"})
public class JpaBookService implements BookService{
    private final BookRepository bookRepository;

    public JpaBookService(BookRepository bookRepository) {
        this.bookRepository = bookRepository;
    }


    @Override
    public Book save(BookDto dto) {
        ...
    }

    @Override
    public List<Book> findAll() {
        ...
    }
}
```