Spring - ogólnie

Główna zasada działania:

- Spring używa **@Autowired** do automatycznego wstrzykiwania zależności. Jeśli w kontenerze Springa istnieje dokładnie jeden bean pasujący do potrzeby, jest on wstrzykiwany. W przypadku wielu beanów o tej samej klasie, używa się **@Qualifier** do wskazania odpowiedniego beana.
- **@ComponentScan** pozwala Springowi automatycznie znaleźć i zarejestrować beany w kontenerze IoC. Skanowanie odbywa się na podstawie adnotacji (@Component, @Service, @Repository, @Controller) w wskazanych pakietach.

Inversion of Control (IoC), czyli odwrócenie sterowania, to zasada, według której nie nasz kod, ale zewnętrzny system (w tym przypadku Spring) kontroluje przepływ programu.

Dependency Injection (DI), czyli wstrzykiwanie zależności, to sposób na dostarczenie do naszego kodu wszystkiego, czego potrzebuje do działania.
W praktyce oznacza to, że gdy piszemy kod aplikacji, definiujemy tylko, co ma się dziać, a Spring Boot dba o to, jak to się stanie. Przykładowo, jeśli nasza aplikacja potrzebuje dostępu do bazy danych, po prostu informujemy Spring Boot, a on zajmuje się nawiązaniem połączenia i zarządzaniem nim.

Adnotacje:

```
@Autowired: Automatyczne wstrzykiwanie zależności przez Spring. Może być stosowane na polach, setterach, konstruktorach.

@Qualifier: Określa, który dokładnie bean ma być wstrzyknięty, gdy istnieje więcej niż jeden kandydat.

@Component: Ogólna adnotacja wskazująca, że dana klasa jest beanem zarządzanym przez Spring.

@Service: Specjalizacja @Component wskazująca, że klasa pełni rolę serwisu w warstwie biznesowej.

@Repository: Specjalizacja @Component dla warstwy dostępu do danych, może oferować dodatkowe funkcje, np. obsługę wyjątków specyficznych dla bazy danych.

@Controller: Specjalizacja @Component dla warstwy prezentacji, w szczególności dla MVC.

@RestController: Połączenie @Controller i @ResponseBody, wskazuje kontroler obsługujący REST API.

@Bean: Metoda w klasie konfiguracyjnej oznaczona tą adnotacją tworzy i zwraca bean, który jest zarządzany przez kontener Springa.

@Configuration: Wskazuje klasę, która zawiera definicje beanów.

@ComponentScan: Określa pakiety do przeszukania w celu znalezienia beanów przez Springa.
```

Bardziej “ludzki” opis adnotacji:

- **@Component:** Oznacza klasę jako komponent zarządzany przez Springa, coś jak gotowy do użycia blok budynku.
- **@Service**: Specjalny typ komponentu dla logiki biznesowej. To część programu, gdzie pisze się większość instrukcji co aplikacja ma robić, jakie operacje wykonać, np. dodanie nowego użytkownika, obliczenie czegoś.
- **@Repository:** Specjalny typ komponentu do komunikacji z bazą danych - to miejsce, gdzie zapisuje się i odczytuje informacje z bazy danych, np. informacje o użytkownikach, produktach itp.
- **@Controller:** Komponent do obsługi żądań HTTP, część systemu odpowiadająca za interakcje z użytkownikiem.
- **@Autowired:** Pozwala Springowi automatycznie dostarczać potrzebne komponenty do innych komponentów.
- @**RestController** to specjalny typ Controllera, który nie zwraca stron internetowych, ale proste dane, np. w formacie tekstowym lub JSON (który wygląda jak zestaw informacji w nawiasach klamrowych).
- @**RequestParam** służy do pobierania danych z adresu strony, ale tych po znaku zapytania, na przykład w /szukaj?fraza=kotki "fraza" to byłby "RequestParam".
- @**RequestBody** to kawałek informacji, które wysyłasz na serwer, na przykład kiedy wypełniasz formularz na stronie.
- @**PathVariable** to sposób na pobranie części tego, co wpiszesz w adresie strony, np. jeśli masz adres /uzytkownik/123, to "123" może być takim "PathVariable", czyli zmienną, którą program może użyć.

Spring Web

Spring Web to część frameworka Spring, która umożliwia tworzenie aplikacji internetowych. Umożliwia ona przyjmowanie żądań HTTP od użytkowników (np. kiedy ktoś wchodzi na stronę internetową lub używa aplikacji mobilnej) i zwracanie odpowiedzi (np. strona HTML, JSON).

Adnotacje:

```
@Controller:
- Oznacza klasę jako kontroler w modelu MVC, który obsługuje żądania HTTP.
- Używany dla tradycyjnych aplikacji webowych, gdzie widokiem może być strona JSP, Thymeleaf itp.

@RestController:
- Połączenie @Controller i @ResponseBody.
- Używany dla usług RESTful, gdzie odpowiedź z metody kontrolera jest automatycznie serializowana do
JSON lub XML.

@RequestMapping oraz jej odmiany (@GetMapping, @PostMapping, @PutMapping, @DeleteMapping):
- Mapuje żądania HTTP na metody kontrolerów.
- Określa URL, metodę HTTP i opcjonalnie nagłówki, parametry, typy zawartości itp.

@PathVariable:
- Wstrzykuje wartości z części ścieżki URL do metody kontrolera.
- Przydatne w REST API, gdzie część URL reprezentuje zasób lub identyfikator.

@RequestParam:
- Wstrzykuje wartości parametrów zapytania (query parameters) do metody kontrolera.

@RequestBody:
- Wstrzykuje treść żądania HTTP bezpośrednio do obiektu parametru metody kontrolera.
- Używane w usługach RESTful przyjmujących JSON lub XML.

@ResponseBody:
- Wskazuje, że wartość zwrócona przez metodę kontrolera ma być użyta jako treść odpowiedzi HTTP.
- Wymagane, gdy nie używamy @RestController.
```

Kody HTTP:

- **1xx (Informacyjne):** Są to odpowiedzi wstępne, wskazujące, że żądanie zostało otrzymane i proces jest kontynuowany.
- **2xx (Sukces):** Kody te wskazują, że żądanie zostało pomyślnie otrzymane, zrozumiane i zaakceptowane.
- **3xx (Przekierowanie):** Kody te oznaczają, że do realizacji żądania potrzebne jest podjęcie dalszych działań przez agenta użytkownika.
- **4xx (Błąd klienta):** Kody te są używane, gdy żądanie zawiera błędną składnię lub nie może być spełnione.
- **5xx (Błąd serwera):** Wskazują one, że serwer nie zdołał spełnić pozornie prawidłowego żądania.

Kody błędów:

- **Informacyjne:**
  - 100 - Continue
  - 101 - Switching Protocols
  - 103 - Early Hints
- **Sukcesu:**
  - **200 - OK** 
  - 201 - Created
  - 206 - Partial Content
- **Przekierowania:**
  - **301 - Moved permanently** 
  - **302 - Found**
  - 304 - Not modified
- **Błędu klienta:**
  - **400 - Bad request**
  - 401 - Unauthorized
  - 402 - Payment Required 
  - 403 - Forbidden
  - **404 - Not found**
  - **405 - Method not allowed** 
  - **408 - Request timeout**
  - **410 - Gone** 
  - 418 - I'm a teapot
  - 420 - Enhance Your Calm 
  - 425 - Too Early
  - 429 - Too Many Requests
  - 451 - Unavailable For Legal Reasons 
- **Błąd serwera**
  - **500 - Internal Server Error**
  - **501 - Not Implemented**
  - **502 - Bad gateway**
  - **503 - Service unavailable** 
  - **504 - Gateway timeout**

Ważne, mało popularne

Żądania:

- **GET (@GetMapping):** Żąda danych z określonego zasobu.
- **POST (@PostMapping):** Przesyła dane do przetworzenia do określonego zasobu.
- **PUT (@PutMapping):** Zastępuje wszystkie obecne reprezentacje docelowego zasobu przesłaną zawartością.
- **DELETE (@DeleteMapping):** Usuwa wszystkie obecne reprezentacje docelowego zasobu wskazanego przez URL.
- **HEAD:** Podobnie jak GET, ale przekazuje tylko linię stanu i sekcję nagłówka.
- **PATCH:** Stosuje częściowe modyfikacje zasobu.

Mockito


mohito

Mockito - biblioteka która jest używana głównie do "mockowania", czyli symulowania zachowań obiektów w testach jednostkowych. Pozwala na określenie, jak atrapy mają się zachować - możemy na przykład ustawić, jakie dane mają zwracać, gdy są wywoływane z określonymi argumentami, albo sprawdzić, czy określone metody zostały wywołane w trakcie testu.

Metody:

```
mock(Class<T> classToMock): Tworzy atrapę (mock) dla danej klasy.

spy(T object): Tworzy szpiega dla obiektu, zachowując jego oryginalne zachowanie.

when(T methodCall): Określa zachowanie mocka w odpowiedzi na wywołanie metody.

verify(T mock): Sprawdza, czy na mocku wykonano określone wywołania.

thenReturn(T value): Określa wartość zwracaną przez mock w odpowiedzi na wywołanie metody.

thenThrow(Throwable... toBeThrown): Określa wyjątek (lub wyjątki), który ma być rzucony przez mock.

doReturn(T toBeReturned): Alternatywny sposób określenia zwracanej wartości, używany gdy metoda jest
void lub ma innych mocków jako argumenty.

doThrow(Throwable... toBeThrown): Alternatywny sposób określenia wyjątku do rzucenia przez mock.

doAnswer(Answer<?> answer): Określa odpowiedź mocka za pomocą niestandardowego zachowania.
```

Adnotacje:

```
@Mock: Tworzy atrapę (mock) dla zadeklarowanego pola klasy.

@Spy: Tworzy szpiega dla zadeklarowanego pola klasy.

@InjectMocks: Automatycznie wstrzykuje mocki i szpiegi do testowanego obiektu.

@Captor: Tworzy ArgumentCaptor dla zadeklarowanego pola.

@RunWith(MockitoJUnitRunner.class): Uruchamia testy z inicjalizacją mocków (nie wymaga ręcznego
wywołania MockitoAnnotations.initMocks(this)).
```

Mock, Spy, ArgumentCaptor - różnice

```
Mock: Atrapa obiektu, którego wszystkie metody są domyślnie stubowane. Odpowiedzi na wywołania muszą być zdefiniowane w teście.

Spy: Kopiuje rzeczywisty obiekt, ale pozwala na nadpisywanie (stubowanie) wybranych metod. Pozostałe metody zachowują swoje rzeczywiste zachowanie.

ArgumentCaptor: Używany do przechwytywania argumentów przekazanych do metod mocków. Pozwala to na dokładne sprawdzenie, czy metody zostały wywołane z oczekiwanymi argumentami.
```

Argument matchery

Określają które wywołania metody powinny zostać obsłużone przez Mockito

```
any(): Matcher akceptujący dowolny argument.

eq(T value): Matcher porównujący argumenty z określoną wartością.

refEq(T value): Matcher porównujący referencje obiektów.

same(T value): Matcher sprawdzający, czy argument to ta sama instancja, co określona wartość.

isNull(): Matcher akceptujący wartości null.

notNull(): Matcher odrzucający wartości null.

any(Class<T> type): Matcher akceptujący dowolny argument danego typu.
```

JPA

JPA pomaga w mapowaniu obiektów Java na tabele baz danych w sposób, który jest zarządzany przez kontekst trwałości, co znacząco upraszcza operacje CRUD (Create, Read, Update, Delete) na danych.

Funkcje i metody:

```
EntityManager: Główny interfejs JPA służący do zarządzania cyklem życia obiektów encji, ich tworzeniem,
 usuwaniem oraz odpytywaniem bazy danych.

- persist(Object entity): Zapisuje obiekt do bazy danych.
- merge(Object entity): Aktualizuje stan obiektu w bazie danych.
- remove(Object entity): Usuwa obiekt z bazy danych.
- find(Class<T> entityClass, Object primaryKey): Wyszukuje obiekt w bazie danych za pomocą klucza
głównego.
- Entity: Klasa Java oznaczona adnotacją @Entity, reprezentująca tabelę w bazie danych, gdzie każda
instancja klasy odpowiada wierszowi w tabeli.

----

@Id: Adnotacja stosowana do oznaczenia pola klasy encji jako klucza głównego.

@GeneratedValue: Określa strategię generowania unikalnych wartości dla klucza głównego.

- Query: Umożliwia tworzenie i wykonywanie zapytań w bazie danych, zarówno w języku JPQL (Java
Persistence Query Language) jak i SQL.
```

Baza H2:

H2 to lekka baza danych w pamięci. Ważne rzeczy:

- Łatwa konfiguracja. nic nie trzeba pobierać
- Po restarcie dane są tracone.
- **Wsparcie dla standardowego SQL**: H2 wspiera większość standardowych funkcji SQL.
- **Wbudowany interfejs webowy**: Dostępny przez przeglądarkę interfejs użytkownika do zarządzania bazą danych.
- **Szybkość**: Jako baza danych w pamięci, H2 jest bardzo szybka.

Spring Test

Polegają one na testowaniu najmniejszych części kodu, takich jak metody i klasy, w izolacji od reszty systemu.

Klasy i metody:

MockMvc: Klasa używana do testowania kontrolerów Spring MVC poprzez programowanie wywołań żądań i asercji na odpowiedziach.

TestRestTemplate: Umożliwia testowanie endpointów RESTowych.
@Test: Adnotacja JUnit określająca metodę jako testową.

Assertions: Klasa z metodami służącymi do asercji, np. assertEquals, assertTrue, itd.

Adnotacje:

@SpringBootTest: Ładuje pełny kontekst aplikacji, używane w testach integracyjnych, ale może być również używane w testach jednostkowych do konfiguracji testowych właściwości.

@DataJpaTest: Ładuje kontekst związany z JPA, używane do testowania repozytoriów.

@WebMvcTest: Ładuje kontekst związany z MVC, używane do testowania kontrolerów webowych.

@JsonTest: Ładuje kontekst potrzebny do testowania JSONów.

@RestClientTest: Ładuje kontekst potrzebny do testowania klientów REST.

@MockBean: Tworzy i rejestruje mock danego beana w kontekście aplikacji Springa, pozwalając na izolację testowanego komponentu od jego zależności.

@InjectMocks: Wstrzykuje mocki (stworzone przez Mockito) do testowanego obiektu.

@ExtendWith(SpringExtension.class): Integruje Springa z JUnit 5, umożliwiając użycie Springowego kontekstu w testach.

//LEGACY poniżej

można zignorować dalsze slajdy

Spring Web:

- **Controller** to część Springa, która zajmuje się odbieraniem informacji, które wpisujesz w przeglądarce, np. adres strony internetowej.

```
@Controller
public class MyPageController {
    @GetMapping("/mojaStrona")
    public String showMyPage() {
        return "mojaStrona";
    }
}

@Controller odpowiada za przekazanie nazwy widoku mojaStrona, który ma być wyświetlony jako strona internetowa.
```

- **RestController** to specjalny typ Controllera, który nie zwraca stron internetowych, ale proste dane, np. w formacie tekstowym lub JSON (który wygląda jak zestaw informacji w nawiasach klamrowych).

```
@RestController
public class MyRestController {
    @GetMapping("/mojeDane")
    public Map<String, String> getMyData() {
        return Collections.singletonMap("klucz", "wartość");
    }
}

@RestController ułatwia zwracanie danych JSON, w tym przykładzie zwracany jest obiekt Map, który zostanie przekształcony na JSON.
```

- **PathVariable** to sposób na pobranie części tego, co wpiszesz w adresie strony, np. jeśli masz adres /uzytkownik/123, to "123" może być takim "PathVariable", czyli zmienną, którą program może użyć.

```
    @GetMapping("/uzytkownik/{id}")
    public String getUser(@PathVariable String id) {
        return "Id użytkownika to: " + id;
    }

    @PathVariable jest używana do pobrania id użytkownika bezpośrednio z URL.
```

- **RequestBody** to kawałek informacji, które wysyłasz na serwer, na przykład kiedy wypełniasz formularz na stronie.

```
@PostMapping("/uzytkownik")
public String addUser(@RequestBody User user) {
    return "Dodano użytkownika o imieniu: " + user.getName();
}

@RequestBody jest używane w żądaniach POST, aby pobrać dane użytkownika przesłane w ciele żądania,
np. w formacie JSON.
```

- **RequestParam** służy do pobierania danych z adresu strony, ale tych po znaku zapytania, na przykład w /szukaj?fraza=kotki "fraza" to byłby "RequestParam".

```
@GetMapping("/wyszukaj")
public String search(@RequestParam(name = "fraza") String searchPhrase) {
    return "Wyszukiwanie frazy: " + searchPhrase;
}

@RequestParam pozwala na pobranie wartości fraza z query parametru w URL
```

- **Service (rodzaj @Component)** to część programu, gdzie pisze się większość instrukcji co aplikacja ma robić, jakie operacje wykonać, np. dodanie nowego użytkownika, obliczenie czegoś.

```
@Service
public class UserService {
    public String getUserName(Long userId) {
        return "Nazwa użytkownika dla ID: " + userId;
    }
}

W @Service definiuje się logikę aplikacji.
```

- **Repository** to miejsce, gdzie zapisuje się i odczytuje informacje z bazy danych, np. informacje o użytkownikach, produktach itp.

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
}

@Repository to interfejs używany do komunikacji z bazą danych, tutaj rozszerzający JpaRepository, który
 dostarcza podstawowe operacje bazodanowe.
```

//TODO - ta kartka pojdzie nizej, tu na samą góre dać “ogólne” adnotacje z springa aby je rozumieć przed czytaniem tych z Spring Web

Spring Test: opisy od wykładowcy

@SpringBootTest - (oznaczenie miejsca gdzie będziemy pisać testy). Stawia kontekst springowy i pozwala uruchamiać w nim testy. Posiadanie kontekstu pozwala nam między innymi na dociągnie poprzez adnotację @Autowired beanów zdefiniowanych w ramach naszej aplikacji. W przypadku niepoprawnej konfiguracji naszego kontekstu (n.p. błędnie zdefiniowanych beanów) te testy również nie zadziałają, ponieważ kontekst springowy nie wystartuje. Kontekst ten jest cachowany między wywołaniami testów, także w innych klasach. Dlatego zmiany w jego konfiguracji będą miały wpływ także na inne testy.

@AutoconfiguredMockMVC - stosowana razem ze @SpringBootTest, jak sama nazwa wskazuje autokonfiguruje beana o nazwie MockMVC. Użycie ten adnotacji na klasie pozwala nam wstrzyknąć beana MockMvc do klasy testowej, a następnie definiować wykonanie zapytań restowych do naszego kontekstu. W przypadku tego sposobu nie posiadamy serwera testowego a jedynie wywołujemy najwyższe warstwy kodu springa, które odpowiadają za przyjęcie i przetworzenie zapytania restowego.

@LocalServerPort możemy użyć, jeśli do adnotacji @SpringBootTest dodamy (webEnvironment = WebEnvironment.RANDOM_PORT). Służy do pobrania wartości portu, na jakim działa tymczasowy serwer aplikacji. W tej sytuacji poza kontekstem springa uruchamiany jest tymczasowy serwer, który możemy wykorzystać w naszych testach. Aby zapytanie trafiło na serwer, a nie tak jak w przypadku MockMvc do kodu należy użyć beana **TestRestTemplate** wstrzykniętego przy pomocy adnotacji @Autowired. Następnie należy zdefiniować zapytanie restowe przy użyciu jego metod. Co istotne, w przeciwieństwie do mockMvc musimy podać tu całego url do naszej aplikacji, a więc http://localhost:8080 + ścieżka do konkretnego endpointu.

@WebMvcTest - tej adnotacji używamy samodzielnie, możemy ustawić jej parametr controllers, w którym definiujemy Controllery/Resourcey używane w adnym teście. Ograniczy to postawienie kontekstu tylko i wyłącznie do danego kontrollera. Użycie tej adnotacji uruchamia jedynie webową warstwę kontekstu springa (przede wszystkim Controllery/Resourcey). Powoduje to, że jeśli bean webowy korzysta z innego beana (np. serwisu) musi tego beana zamockować za pomocą adnotacji **@MockBean** (istnieje także **@SpyBean** jednak użycie tej adnotacji nie spowoduje utworzenia nowego beana). Zachowanie tych adnotacji jest analogiczne jak adnotacji **@Mock** i **@Spy** z Mockito z tą różnicą, że są one włączane do kontekstu springa. Użycie tej adnotacji pozwala nam korzystać ze skonfigurowanego beana MockMvc.

@DataJpaTest - uruchamia nam bazę testową in memory h2 o ile znajduje się w dependencies aplikacji i wykonuje operacje tworzenia struktur bazy danych. Pozwala testować wszystkie beany z warstwy JPA, a więc głównie te oznaczone adnotacją **@Repository**.

MPR

“Zdać na 3 i wyjebane”

- **Autowired** wstrzykuje zależności “dodaje obiekty” na podstawie nazwy

```
2 usages
private final StudentService studentService;
@Autowired
public studentController(StudentService studentService) {
    this.studentService = studentService;
}
```

- **Component** oznacza, ze ta klasa jest komponentem dla autowired
- **Bean** działa razem z @Component i oznacza się nim metody

```
@Component
public class StudentService {
    1 usage
    @Bean
    public List<Student> getStudents(){
        return List.of(
                new Student(
                        id: 1L,
                        name: "Karol",
                        email: "s27092@pjwstk.edu.pl",
                        LocalDate.of( year: 2003, Month.MAY,  dayOfMonth: 15),
                        age: 20));
    }
}
```

- **RequestMapping** nie wiem jak to wytłumaczyć, ale jak sie wpisze (path=api/students) to to teraz bedzie dostepne w localhost:8080/api/students
- **GetMapping** odpowiada zapytaniu HTTP GET. Tak samo:
  @PostMapping @PutMapping @PatchMapping @DeleteMapping

```
@RestController
@RequestMapping(path = "api/v1/student")
public class studentController {

    2 usages
    private final StudentService studentService;
    @Autowired
    public studentController(StudentService studentService) {
        this.studentService = studentService;
    }

    @GetMapping
    public List<Student> getStudents(){
        return studentService.getStudents();
    }
}
```

Miscellaneous:

- **Qualifier** - wraz z **@Autowired** pozwala na rozróżnienie między **@Bean** tego samego typu

- **Configuration** - oznaczenie dla klasy konfiguracyjnej w Spring. Nie wiem czy cos to robi chyba po prostu oznacza, cos jak @FunctionalInterface