



**WYŻSZA SZKOŁA BANKOWA**  
**Warszawa**

# Programowanie aplikacji w Java

**Maciej Gowin**

**Zjazd 9 - dzień 2**

## Spring w praktyce: @Valid

Do tej pory walidacja odbywała się na poziomie kodu oraz zwracanie błędu na podstawie logiki zapisanej explicite w kodzie.

Framework Spring wprowadza automatyzację tego procesu poprzez adnotacje, która pozwala na odseparowanie logiki biznesowej od logiki walidacji danych wejściowy.

# Spring w praktyce: @Valid

Aby dodać wsparcie dla walidacji realizowanej poprzez adnotację `@Valid` zaimportować zależność:

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

W momencie, gdy Spring Boot odnajdzie argument metody opatrzone adnotacją `@Valid` załadowany zostanie automatycznie mechanizm walidacji oparty o JSR 300 wraz z referencyjną implementacją Hibernate.

```
<dependency>
  <groupId>org.hibernate.validator</groupId>
  <artifactId>hibernate-validator</artifactId>
</dependency>
```

# Spring w praktyce: @Valid

Adnotacja `@Valid` sprawdza zgodność przesłanego modelu na podstawie dodatkowych adnotacji opisujących przesyłane dane.

```
@PostMapping("/customers/{id}")
public CustomerDto updateCustomer(@PathVariable("id") Integer id,
                                  @Valid @RequestBody CustomerDto customerDto) {
    /* ... */
}
```

```
public class CustomerDto {

    private int id;

    @NotEmpty(message = "First name must be not empty")
    private String firstName;

    @NotBlank(message = "Last name must be not blank")
    private String lastName;
}
```

# Spring w praktyce: @Valid

W przypadku błędów walidacji Spring Boot automatycznie zgłosi błąd `MethodArgumentNotValidException`, który możemy później obsłużyć na poziomie `@ExceptionHandler`.

Informacje o błędach przesyłane w obiekcie `BindingResult`, do którego mamy dostęp z poziomu: `MethodArgumentNotValidException.getBindingResults`.

# Spring w praktyce: @Valid

Możemy też przechwycić błędy na poziomie metody. Zostaną one automatycznie wstrzyknięcie do obiektu `BindingResult`.

W takim przypadku nie zostanie zgłoszony błąd `MethodArgumentNotValidException`.

```
@PostMapping("/customers")
public CustomerDto addCustomer(@Valid @RequestBody CustomerDto customerDto,
                               BindingResult bindingResult) {
    /* ... */
}
```

## Programowanie: przykład 89

Użycie wbudowanego mechanizmu walidacji dla niepustych pól łańcuchów znaków.

# Spring w praktyce: @Valid

Istnieje szereg wbudowanych adnotacji opisujących sposoby walidacji dostępnych w pakiecie `javax.validation.constraints`.

Do najczęściej używanych należą: `@NotNull`, `@NotEmpty`, `@NotBlank`, `@Size`, `@Min`, `@Max`, `@Past`, `@Future`.

Oczywiście istnieje możliwość zdefiniowania własnej adnotacji oraz metodę obsługującą jej zachowanie.



# Spring w praktyce: @Valid

Do definicji logiki walidacji używamy interfejsu `ConstraintValidator`, który jest aktywowany na podstawie danej adnotacji dla danego typu.

```
public class AdultValidator implements ConstraintValidator<Adult, LocalDate> {  
    @Override  
    public void initialize(final Adult constraintAnnotation) {  
    }  
  
    @Override  
    public boolean isValid(final LocalDate value, final ConstraintValidatorContext context) {  
        /* ... */  
    }  
}
```

# Spring w praktyce: @Valid

Istotne jest podanie walidatora podczas definicji adnotacji. W czasie działania programu walidator sprawdza, czy dana adnotacja ma zdefiniowane pola `message`, `group` oraz `payload`.

```
@Target({ FIELD, METHOD, PARAMETER, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Constraint(validatedBy = AdultValidator.class)
public @interface Adult {

    String message() default "Must be above 18";

    Class<?>[] groups() default {

    };

    Class<? extends Payload>[] payload() default {

    };
}
```

## Programowanie: przykład 90

Użycie wbudowanego mechanizmu walidacji do definicji adnotacji sprawdzającej, czy dana osoba jest osobą pełnoletnią.

# Spring w praktyce: RestTemplate

Jeżeli nasz system komunikuje się z innym serwisem, konieczna może być wymiana danych przy pomocy protokołu HTTP. Jest to szczególnie istotne w przypadku przekazywania danych pomiędzy dwoma serwisami RESTful.

Niskopoziomowa implementacja wywołań zapytań jest możliwa, lecz uciążliwa. Framework Spring dostarcza użyteczną klasę `RestTemplate`, która wspiera komunikację pomiędzy serwisami.

# Spring w praktyce: RestTemplate

Klasa `RestTemplate` dostarcza szereg metod realizujących zapytania HTTP oraz pozwalające na automatyczną konwersję modelu do obiektów.

Do podstawowych należą:

- `getForObject` , `getForEntity` - wywołanie metody GET, pobranie danych
- `postForObject` , `postForEntity` - wywołanie metody POST, wysłanie oraz pobranie danych
- `exchange` - wywołanie dowolnej metody HTTP na podstawie parametry, opcjonalne wysłanie oraz pobranie danych

# Spring w praktyce: @Value

Adnotacja `@Value` może zostać użyta do wstrzyknięcia wartości zdefiniowanych jako zmienne konfiguracyjne.

Zmienne najczęściej definiowane są w plikach konfiguracyjnych – dostarczanych razem z aplikacją lub też z zewnątrz podczas uruchomienia.

Wartości mogą być definiowane na poziomie pól lub parametrów konstruktorów oraz metod.

```
@Value("${myApplication.myProperty}")  
private String myProperty;
```

# Spring w praktyce: @Value

Wyrażenia użyte wraz z adnotacją `@Value` wspierają SpEL (Spring Expression Language).

SpEL jest językiem wyrażeń, który pozwala na przeszukiwanie oraz operacje na wartościach podczas uruchomienia.

```
myApplication.letters = a,b,c  
myApplication.letterPerKey = valuesMap={key1: 'a', key2: 'b', key3: 'c'}
```

```
@Value("#{ '${myApplication.letters}'.split(',') }")  
private List<String> letters;
```

```
@Value("#{ ${myApplication.letterPerKey} }")  
private Map<String, String> letterPerKey;
```

```
@Value("#{ ${myApplication.letterPerKey}.key1 }")  
private String key1Letter;
```

# Programowanie: przykład 91

Użycie `RestTemplate` do załadowanie domyślnego lotniska dla kraju użytkownika na podstawie zewnętrznego serwisu.

Do pobrania domyślnego kodu lotniska może zostać wykorzystany serwis:

```
https://www.ryanair.com/api/locate/3/countries
```



# Spring: JPA

Spring wspiera standard JPA wraz z interfejsem `EntityManager`. Szereg zależności zależnych od JPA oraz wstępna konfiguracja dostępna jest poprzez:

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

Jako implementacji JPA framework Spring używa biblioteki Hibernate, która jest dodawana domyślnie.

# Spring w praktyce: EntityManager

W standardzie JPA do obsługi zapytań używamy interfejsu `EntityManager`.

Przykładowym rozszerzeniem tego interfejsu jest interfejs `Session` dostarczany wraz z implementacją przez bibliotekę Hibernate.

Podczas użycia implementacji `EntityManager` realizowanej przez Hibernate istnieje możliwość wyłuskania `Session` poprzez metodę `EntityManager.unwrap`.

# Spring w praktyce: EntityManager

W przypadku pracy z `EntityManager` do wstrzyknięcia zależności w ziarnach użyjemy specyficznej dla standardu JPA adnotacji `@PersistenceContext`.

Jest ona rozpoznawana przez kontener oraz powoduje wstrzyknięcie nowej instancji dla każdego z wątków, ponieważ klasa ta nie jest bezpieczna wielowątkowo.

```
@Repository
public class CustomerRepository {

    @PersistenceContext
    private EntityManager entityManager;

    /* ... */
}
```

# Spring w praktyce: EntityManager

Należy pamiętać, że operacje modyfikujące dane powinny być wykonywane w obrębie transakcji. Nie musimy definiować ich z poziomu kodu metod.

Spring wspiera adnotacje `@Transactional` opisującą transakcję będącą częścią standardu JPA.

```
@Transactional
public Customer addCustomer(Customer customer) {
    entityManager.persist(customer);
    return customer;
}
```

# Spring w praktyce: EntityManager

W przypadku użycia auto-konfiguracji Spring Boot zainicjalizuje wszystkie wymagane ziarna konieczne do stworzenia ziarna `EntityManager`.

Do pełnej konfiguracji niezbędne będzie ustawienie z góry ustalonych zmiennych konfiguracyjnych.

```
spring.datasource.url = jdbc:mysql://localhost:3306/sample
spring.datasource.username = root
spring.datasource.password = root
spring.datasource.driver-class-name = com.mysql.cj.jdbc.Driver

spring.jpa.database-platform = org.hibernate.dialect.MySQL5Dialect
spring.jpa.hibernate.ddl-auto = update
```

# Spring w praktyce: DataSource

Wewnętrznie użyta jest klasa `DataSource` definiująca połączenie. Do tworzenia instancji `EntityManager` wykorzystywany jest klasa `EntityManagerFactory`.

W przypadku wyłączenia auto-konfiguracji konieczne byłoby stworzenie hierarchii zależności.

# Spring w praktyce: DataSource

```
@Configuration
public class ApplicationConfiguration {

    @Bean
    public DataSource dataSource() {
        HikariConfig config = new HikariConfig();
        /* ... */
        return new HikariDataSource(config);
    }

    @Bean
    public PlatformTransactionManager transactionManager() {
        JpaTransactionManager transactionManager = new JpaTransactionManager(entityManagerFactory());
        transactionManager.setDataSource(dataSource());
        /* ... */
        return transactionManager;
    }

    @Bean
    public EntityManagerFactory entityManagerFactory() {
        LocalContainerEntityManagerFactoryBean entityFactory = new LocalContainerEntityManagerFactoryBean();
        entityFactory.setDataSource(dataSource());
        /* ... */
        return entityFactory.getObject();
    }
}
```

# Spring w praktyce: @EntityScan

Adnotacja `@EntityScan` wspiera wyszukiwanie i rejestrację wszystkich klas definiujących encje JPA używane przez ORM.

Użycie adnotacji spowoduje przeszukanie danego pakietu w poszukiwaniu klas opartych o `@Entity`. Podobnie jak to było w przypadku konfiguracji Hibernate.

```
@EntityScan("pl.wsb.programowaniejava.maciejgowin")
```



## Programowanie: przykład 92

Połączenie z bazą danych przy pomocy JPA oraz użycie `EntityManager` .

# Spring w praktyce: JpaRepository

Spring Data JPA wprowadza interfejs `JpaRepository` pozwalający na tworzenie użytecznych ziaren definiujących operacje CRUD na bazie danych, takie jak:

- wyszukiwanie: `List<T> findAll()`, `Optional<T> findById(ID)`,
- zapisywanie: `T save(T)`, `T saveAndFlush(T)`,
- usuwanie: `void delete(T)`, `void deleteById(ID)`.

Dla dowolnej encji możemy zdefiniować interfejs, który jest rozszerzeniem `JpaRepository` oraz będzie zarejestrowany jako ziarno.

```
@Repository
public interface CustomerCrudRepository extends JpaRepository<Customer, Integer> {
}
```

# Spring w praktyce: JpaRepository

Domyślnie Spring Boot włącza wsparcie dla repozytoriów JPA oraz wyszukuje definicji w pakiecie, w którym zlokalizowany jest klasa opisana adnotacją `@SpringBootApplication` .

Jeżeli repozytoria JPA zdefiniowane są w innym pakiecie możemy zdefiniować alternatywne pakiety poprzez `@EnableJpaRepositories` .

# Spring w praktyce: @Query

Na poziomie `JpaRepository` możemy dodawać też spersonalizowane zapytania do bazy poprzez adnotację `Query`, którą definiujemy na poziomie metody.

Do definicji zapytań używamy JPQL.

```
@Repository
public interface CustomerCrudRepository extends JpaRepository<Customer, Integer> {

    @Query("FROM Customer ORDER BY firstName")
    List<Customer> findAllOrderByFirstName();

    @Query("FROM Customer WHERE lastName = :lastName")
    List<Customer> findAllByLastName(@Param("lastName") String lastName);
}
```

## Programowanie: przykład 93

Połączenie z bazą danych przy pomocy JPA oraz użycie `JpaRepository` .

# Spring Boot w praktyce: Actuator

Actuator pozwala na monitorowanie aplikacji.

Dla przykładu: jeżeli nasza aplikacja korzysta z bazy danych, rozwiązanie to pozwala na sprawdzenie połączenie oraz ustalenie statusu aplikacji na podstawie stanu bazy.

Spring Boot udostępnia wsparcie dla Actuator poprzez zależność.

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

# Spring Boot: Actuator

Actuator wprowadza wbudowane ścieżki domyślnie udostępnione pod prefiksem `/actuator`. Między innymi:

- `/health` : podsumowanie statusu aplikacji,
- `/info` : informacje o systemie, mogą być konfigurowane oraz rozszerzane,
- `/env` : podsumowanie zmiennych systemowych,
- `/metrics` : szczegóły metryk aplikacji,
- `/logfile` : logi aplikacyjne.

Domyślnie dostępne są: `/health` oraz `/info`.

# Spring Boot: Actuator

Actuator konfigurowany jest przez szereg zmiennych, które pozwalają na definiowanie dostępnych funkcji oraz ich konfigurację.

```
management.endpoints.web.exposure.include = health, info  
management.endpoint.health.show-details = ALWAYS
```



# Spring Boot: Actuator oraz /health

Ścieżka `/actuator/health` zwraca status aplikacji, który jest sumą statusów zdefiniowanych komponentów dostarczających statusów elementów aplikacji. Moduły Spring mogą dostarczać kolejne komponenty, które rejestrowane są automatycznie.

Aby zdefiniować własny komponent sprawdzający status interesującego nas elementu, należy dostarczyć ziarno implementujące interfejs `HealthIndicator`.

```
@Component
public class CustomHealthIndicator implements HealthIndicator {

    @Override
    public Health health() {
        /* ... */
    }
}
```

# Spring Boot: Actuator oraz /info

Ścieżka `/actuator/info` zwraca informacje o aplikacji. Możliwe jest rozszerzenie informacji np. o jej nazwę oraz rezultaty procesu budowania.

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <version>2.2.2.RELEASE</version>
  <executions>
    <execution>
      <goals>
        <goal>build-info</goal>
        <goal>repackage</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

# Spring Boot: Actuator oraz personalizacja

Możliwe jest zdefiniowanie własnej ścieżki dla danej funkcjonalności poprzez adnotację `@Endpoint` .

```
@Component
@Endpoint(id = "copyright")
public class CopyrightEndpoint {

    @ReadOperation
    public Map<String, String> copyright() {
        return Map.of(
            "author", "Maciej Gowin",
            "year", Year.now().toString());
    }
}
```

## Programowanie: przykład 94

Użycie moduły monitoringu przy pomocy Spring Actuator wraz z definicją własnych elementów.

# Spring Security

Moduł Spring Security pozwala na automatyzację procesu uwierzytelniania oraz autoryzacji.

Zabezpieczenie aplikacji składa się z dwóch etapów:

- uwierzytelniania (ang. authentication): potwierdzenie tożsamości danego podmiotu oraz sprawdzenie, czy dany podmiot jest tym, za którego się podaje,
- autoryzacja (ang. authorization): nadanie podmiotowi dostępu do zasobu.

# Spring Security

Moduł ten jest operaty o konwencji, dzięki którym przy minimalnej konfiguracji dostarczana jest podstawowa funkcjonalność.

Istnieje szereg sposobów definiowania przepływu oraz przy użyciu jakich mechanizmów, uwierzytelnianie i autoryzacja będą realizowane.

# Spring Boot w praktyce: Security

Główną zależnością pozwalającą na aktywowanie mechanizmu bezpieczeństwa w aplikacji Spring Boot jest:

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

Konfiguracja procesu jest w głównej mierze oparta o nadpisanie `WebSecurityConfigurerAdapter`.

## Programowanie: przykład 95

Użycie Spring Security do zabezpieczenia zasobów aplikacji.



## Programowanie: zadanie 32

Przeanalizuj moduły `spring-jdbc` oraz `spring-boot-starter-jdbc`. Przetestuj połączenie z bazą danych za pomocą `JdbcTemplate`.