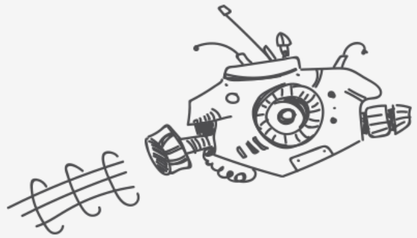
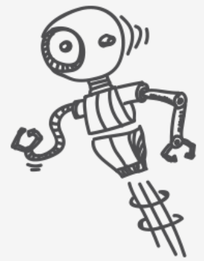
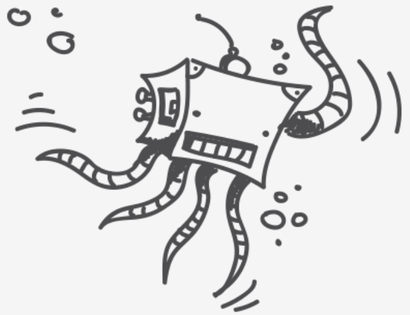


Coders Lab
SZKOŁA IT

Walidacja

v3.3



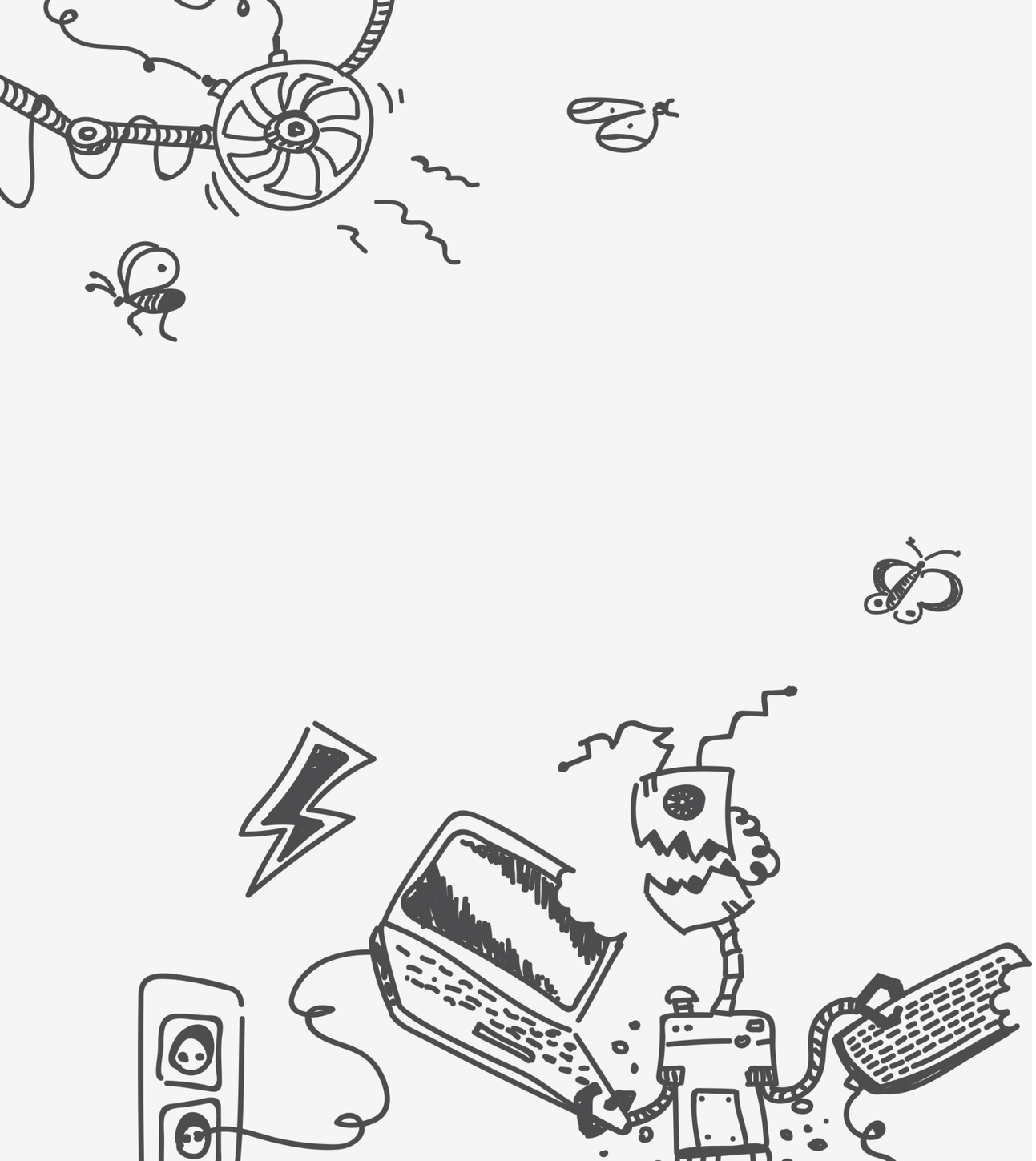


Plan

01 Wprowadzenie do walidacji

02 Walidacja formularzy

03 Własny walidator



Wprowadzenie do walidacji

Walidacja

Walidacja - jest to sprawdzanie poprawności danych.

Co i w jakim zakresie walidujemy zależy od naszych wymagań biznesowych.

Sprawdzać powinniśmy np.:

- Wypełnienie danych - np. przy rejestracji wymagamy podania imienia.
- Czy zbiór wartości jest prawidłowy - np. imię nie może zawierać cyfr.
- Czy format danych jest prawidłowy - np. kod pocztowy.
- Dane identyfikacyjne - np. pesel ma określoną sumę kontrolną oraz liczbę znaków.

Walidacja

Nasze encje powinny spełniać określone warunki, np.:

- → cena ma być zawsze liczbą dodatnią.
- → wiadomość nie może być dłuższa niż 140 znaków.

Jest to możliwe dzięki **walidacji.**

Możemy wykorzystać obsługę API Java Validation JSR 380 znaną również jako **Bean Validation 2.0.** - a dokładnie implementację Hibernate-validator.

JSR 380 - to specyfikacja dotycząca sprawdzania poprawności ziaren specyfikacje mają ustandaryzować opis walidacji za pomocą adnotacji.

https://beanvalidation.org/2.0/

https://jcp.org/en/jsr/detail?id=380

Zależności

Pierwszym krokiem do skorzystania z adnotacji przypisanych do encji jest dołączenie odpowiedniej biblioteki do naszego projektu.

Wyszukujemy w repozytorium mavena: **hibernate-validator**.

https://mvnrepository.com/artifact/org.hibernate/hibernate-validator/

Następnie dodajemy zależność w pliku **pom.xml**, np:

```
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>6.0.17.Final</version>
</dependency>
```

Adnotacje walidacji

W celu nałożenia ograniczeń lub warunków jakie mają spełniać encje wykorzystujemy adnotacje:

- **@AssertFalse** musi być typu Boolean i przyjąć wartość false.
- **@AssertTrue** musi być typu Boolean i przyjąć wartość true.
- **@DecimalMax** musi być liczbą typu BigDecimal, której wartość jest mniejsza bądź równa podanej wartości typu BigDecimal.
- **@DecimalMin** musi być liczbą typu BigDecimal, której wartość jest większa lub równa podanej wartości typu BigDecimal.
- **@Digits** musi być liczbą posiadającą określoną liczbę cyfr.
- **@Future** wartością oznaczonego elementu musi być data z przyszłości.

Adnotacje walidacji

- **@NotNull** nie może być null.
- **@Null** musi być null.
- **@Past** musi być data z przeszłości.
- **@Max** musi być liczbą, której wartość jest mniejsza od danej wartości lub jej równa.
- **@Min** musi być liczbą, której wartość jest większa od danej wartości lub jej równa.

Adnotacje walidacji

- **@Pattern** wartość oznaczonego elementu musi spełniać warunek określony wyrażeniem regularnym.
- **@Size** String, kolekcja lub tablica, której długość mieści się w podanym zakresie.
- **@Range** wartość mieści się w podanym zakresie.

Więcej na temat wbudowanych reguł znajdziesz w dokumentacji:

https://docs.jboss.org/hibernate/stable/validator/reference/en-US/html_single/#validator-defineconstraints-spec.

Dodatkowe adnotacje

Wykorzystywany przez nas hibernate validator (6.0.17.Final) zawiera również inne przydatne walidatory, których specyfikacja **JSR** nie uwzględnia:

- **@PESEL**
- **@NIP**
- **@REGON**
- **@Email**
- **@URL**

- **@NotBlank** - sprawdza czy String po usunięciu białych znaków nie jest pusty
- **@NotEmpty** dla kolekcji sprawdza czy nie równa się null oraz czy size >0

Przykładowa encja

```
@Entity
public class Person {
    @NotNull
    @Size(min = 2, max = 30)
    private String firstName;
    @NotNull
    @Size(min = 2, max = 30)
    private String lastName;
}
```

Dodanie wiadomości do walidacji

Często chcemy wyświetlić wiadomość, gdy encja nie przejdzie walidacji. Wykorzystać możemy w tym przypadku atrybut adnotacji **message**.

```
public class Person {
    @Min(value =18, message="Musi mieć przynajmniej 18 lat")
    private int age;
}
```

Plik tłumaczeń

Zamiast ustawiać wiadomość błędu walidacji możemy zdefiniować odpowiedni plik tłumaczeń i w nim określić komunikaty.

W tym celu tworzymy plik **ValidationMessages.properties** w lokalizacji: src/main/resources

W pliku umieszczamy wpisy w oddzielnych wierszach, oddzielone znakiem równości, schematycznie:

klucz.do.tlumaczenia = Przetłumaczony wpis

np.

```
javax.validation.constraints.Size.message = Nie Nie Nie
javax.validation.constraints.Min.message = Nasz komunikat
org.hibernate.validator.constraints.pl.PESEL.message = Niepoprawny Pesel
```

Pierwszy człon jest odpowiednikiem pakietu, w którym znajduje się adnotacja.

Problemy kodowania

W przypadku wystąpienia problemów z kodowaniem znaków wystarczy skorzystać z ustawień w **IntelliJ** opisanych w poniższym artykule.

https://www.jetbrains.com/help/idea/properties-files.html

Po zmianie ustawień, może być dodatkowo wymagane ponowne wpisanie komunikatów zawierających problematyczne znaki.

Jednocześnie od wersji 9 Javy kodowanie plików `properties` zostało zmienione na `UTF-8` z wcześniejszego `ISO-8859-1`. Problem ten więc nie powinien w przyszłości występować.

https://docs.oracle.com/javase/9/intl/internationalization-enhancements-jdk-9.htm

Plik tłumaczeń

Aby sprawdzić pod jakim kluczem dostępne jest tłumaczenie, które chcemy zmienić, wystarczy przejść do kodu adnotacji.

W naszym IDE z wciśniętym klawiszem Ctrl klikamy w adnotację.

Wycinek kodu adnotacji Min

```
@Retention(RUNTIME)
@Documented
@Constraint(validatedBy = { })
public @interface Min {
    String message() default "{javax.validation.constraints.Min.message}";
}
```

zawarty w nawiasach klamrowych ciąg znaków będzie szukanym przez nas kluczem.

Plik tłumaczeń

Możemy zdefiniować pliki tłumaczeń dla konkretnych języków.

W tym celu tworzymy plik **ValidationMessages_pl.properties** w lokalizacji: src/main/resources

Spring automatycznie skorzysta z odpowiedniego pliku w zależności od ustawień lokalizacyjnych.

Aby ustawić domyślny język Polski należy w **klasie konfiguracji** zdefiniować ziarno Springa:

```
@Bean(name="localeResolver")
public LocaleContextResolver getLocaleContextResolver() {
    SessionLocaleResolver localeResolver = new SessionLocaleResolver();
    localeResolver.setDefaultLocale(new Locale("pl","PL"));
    return localeResolver; }
```

Plik tłumaczeń

Możemy zdefiniować pliki tłumaczeń dla konkretnych języków.

W tym celu tworzymy plik **ValidationMessages_pl.properties** w lokalizacji: src/main/resources

Spring automatycznie skorzysta z odpowiedniego pliku w zależności od ustawień lokalizacyjnych.

Aby ustawić domyślny język Polski należy w **klasie konfiguracji** zdefiniować ziarno Springa:

```
@Bean(name="localeResolver")
public LocaleContextResolver getLocaleContextResolver() {
    SessionLocaleResolver localeResolver = new SessionLocaleResolver();
    localeResolver.setDefaultLocale(new Locale("pl","PL"));
    return localeResolver; }
```

Tworzymy obiekt typu **SessionLocaleResolver** dla naszej aplikacji - oznacza to, że informacje będą trzymane w sesji.

Plik tłumaczeń

Możemy zdefiniować pliki tłumaczeń dla konkretnych języków.

W tym celu tworzymy plik **ValidationMessages_pl.properties** w lokalizacji: src/main/resources

Spring automatycznie skorzysta z odpowiedniego pliku w zależności od ustawień lokalizacyjnych.

Aby ustawić domyślny język Polski należy w **klasie konfiguracji** zdefiniować ziarno Springa:

```
@Bean(name="localeResolver")
public LocaleContextResolver getLocaleContextResolver() {
    SessionLocaleResolver localeResolver = new SessionLocaleResolver();
    localeResolver.setDefaultLocale(new Locale("pl","PL"));
    return localeResolver; }
```

Ustawiamy domyślne **locale** dla naszej aplikacji.

Walidacja danych – wywołanie walidacji

Aby wywołać walidację w akcji kontrolera na utworzonym obiekcie musimy wstrzyknąć zależność do walidatora

```
@Autowired
Validator validator;
```

Aby możliwe było wstrzykiwanie obiektu **Validator** musimy zdefiniować poniższe ziarno w **klasie konfiguracji**:

```
@Bean
public Validator validator() {
    return new LocalValidatorFactoryBean();
}
```

LocalValidatorFactoryBean jest dla Springa implementacją interfejsu **javax.validation.Validator**.

Walidacja danych – wywołanie walidacji

Na wstrzykniętym obiekcie **validator** wywołujemy metodę **validator.validate(obToValidate);**

Metoda ta zwraca zbiór **(Set)** obiektów typu **ConstraintViolation**, zawierają one informacje o błędach które wystąpiły.

```
Set<ConstraintViolation<Person>> violations = validator.validate(p2);
```

Zbiór ten jest parametryzowany klasą, której obiekty sprawdzamy.

Wywołując metodę **violations.isEmpty()** sprawdzamy czy wystąpiły błędy.

ConstraintViolation posiada metody:

getPropertyPath() - pobierającą nazwę atrybutu dla którego wystąpił błąd.

getMessage() - pobierającą wiadomość z opisem błędu.

Walidacja danych – przykład

```
@Autowired
Validator validator;
@RequestMapping("/validate")
@ResponseBody
public String validateTest() {
    Person p2 = new Person();
    Set<ConstraintViolation<Person>> violations = validator.validate(p2);
    if (!violations.isEmpty()) {
        for (ConstraintViolation<Person> constraintViolation : violations) {
            System.out.println(constraintViolation.getPropertyPath() + " "
                               + constraintViolation.getMessage()); }
    } else {
        // save object
    }
    return "validateResult";
}
```

Walidacja danych – przykład

```
@Autowired
Validator validator;
@RequestMapping("/validate")
@ResponseBody
public String validateTest() {
    Person p2 = new Person();
    Set<ConstraintViolation<Person>> violations = validator.validate(p2);
    if (!violations.isEmpty()) {
        for (ConstraintViolation<Person> constraintViolation : violations) {
            System.out.println(constraintViolation.getPropertyPath() + " "
                               + constraintViolation.getMessage()); }
    } else {
        // save object
    }
    return "validateResult";
}
```

Wstrzykujemy walidator.

Walidacja danych – przykład

```
@Autowired
Validator validator;
@RequestMapping("/validate")
@ResponseBody
public String validateTest() {
    Person p2 = new Person();
    Set<ConstraintViolation<Person>> violations = validator.validate(p2);
    if (!violations.isEmpty()) {
        for (ConstraintViolation<Person> constraintViolation : violations) {
            System.out.println(constraintViolation.getPropertyPath() + " "
                               + constraintViolation.getMessage()); }
    } else {
        // save object
    }
    return "validateResult";
}
```

Wywołujemy metodę **validate()** przypisując jej wynik do zbioru zawierającego potencjalne błędy.

Walidacja danych – przykład

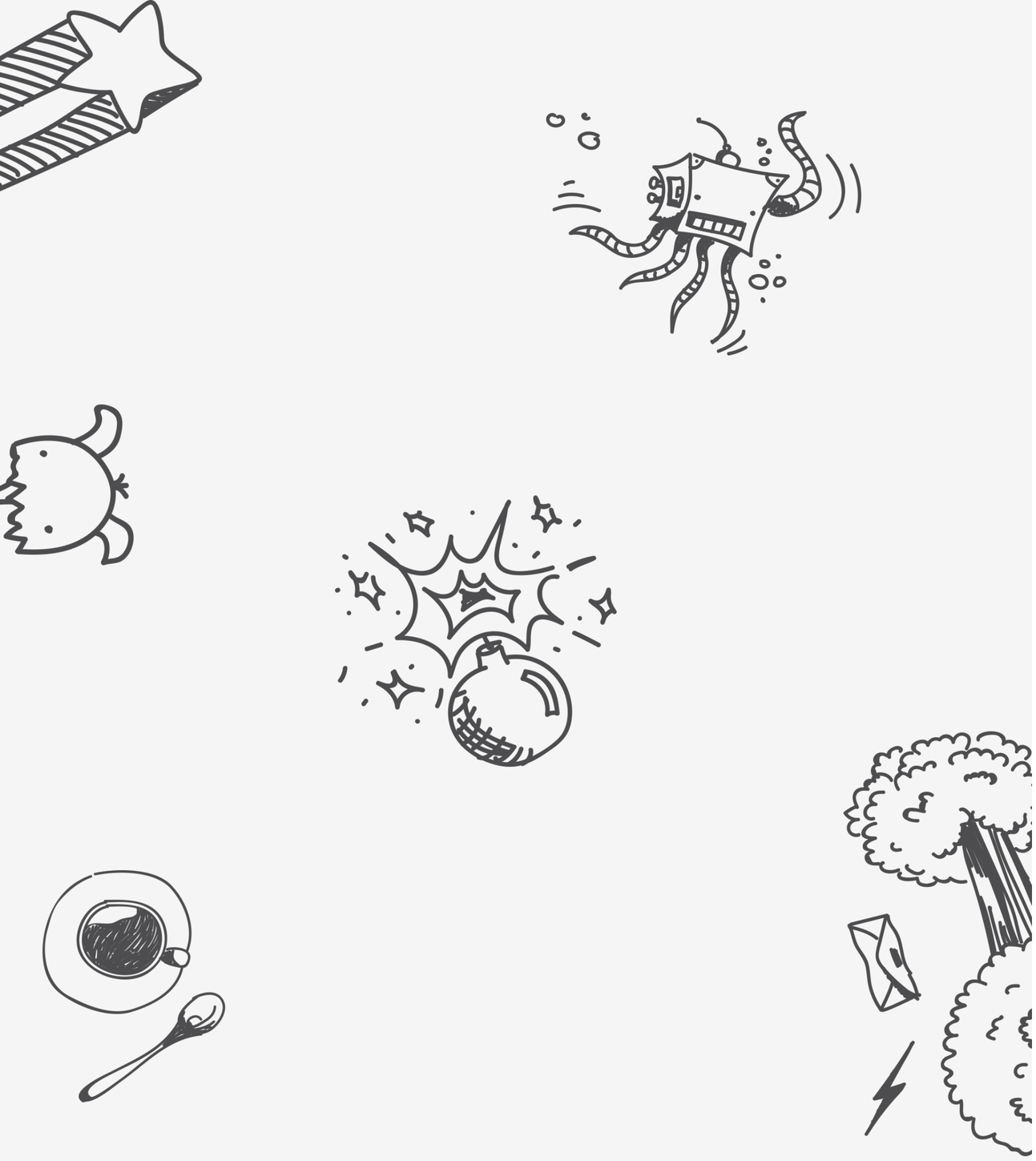
```
@Autowired
Validator validator;
@RequestMapping("/validate")
@ResponseBody
public String validateTest() {
    Person p2 = new Person();
    Set<ConstraintViolation<Person>> violations = validator.validate(p2);
    if (!violations.isEmpty()) {
        for (ConstraintViolation<Person> constraintViolation : violations) {
            System.out.println(constraintViolation.getPropertyPath() + " "
                               + constraintViolation.getMessage()); }
    } else {
        // save object
    }
    return "validateResult";
}
```

Sprawdzamy czy wystąpiły błędy.

Walidacja danych – przykład

```
@Autowired
Validator validator;
@RequestMapping("/validate")
@ResponseBody
public String validateTest() {
    Person p2 = new Person();
    Set<ConstraintViolation<Person>> violations = validator.validate(p2);
    if (!violations.isEmpty()) {
        for (ConstraintViolation<Person> constraintViolation : violations) {
            System.out.println(constraintViolation.getPropertyPath() + " "
                               + constraintViolation.getMessage()); }
    } else {
        // save object
    }
    return "validateResult";
}
```

Dla wszystkich błędów wyświetlamy nazwę atrybutu klasy, który nie przeszedł walidacji oraz komunikat.



Czas
na zadania

Wykonaj zadania z działu:
Walidacja

Walidacja formularzy

Walidacja formularzy

Omówiona we wcześniejszym rozdziale metoda walidacji - jest przydatna, w przypadku gdy dane do naszych obiektów otrzymujemy z api lub wczytujemy z innego źródła, np. pliku.

W tym rozdziale opiszemy jak walidować dane wprowadzane przez użytkowników za pomocą formularzy.

Walidacja formularzy

Aby dodać walidację do **akcji kontrolera** obsługującej zapis formularza wystarczy opatrzyć zapisywany obiekt adnotacją **@Valid**:

```
public String processRegistration(@Valid Person person, BindingResult result) { }
```

Jako atrybut metody dodajemy również obiekt typu **BindingResult** - to do niego zostaną zbindowane ewentualne informacje o błędach.

Sprawdzenia czy występują błędy wykonujemy za pomocą metody:

```
result.hasErrors()
```

wywołanej na obiekcie typu **BindingResult**, który przyjmujemy jako argument.

Walidacja danych – przykład akcji

```
@RequestMapping(value = "/add", method = RequestMethod.POST)
public String processRegistration(@Valid Person person, BindingResult result) {
    if (result.hasErrors()) {
        return "person/add";
    }
    personDao.save(person);
    return "redirect:/success";
}
```

Walidacja danych – przykład akcji

```
@RequestMapping(value = "/add", method = RequestMethod.POST)
public String processRegistration(@Valid Person person, BindingResult result) {
    if (result.hasErrors()) {
        return "person/add";
    }
    personDao.save(person);
    return "redirect:/success";
}
```

Jeżeli wystąpi błąd, wyświetlamy ponownie formularz dodawania. W formularzu tym będą już wcześniej wypełnione dane.

Walidacja danych – przykład akcji

```
@RequestMapping(value = "/add", method = RequestMethod.POST)
public String processRegistration(@Valid Person person, BindingResult result) {
    if (result.hasErrors()) {
        return "person/add";
    }
    personDao.save(person);
    return "redirect:/success";
}
```

Jeżeli wystąpi błąd, wyświetlamy ponownie formularz dodawania. W formularzu tym będą już wcześniej wypełnione dane.

Zapis do bazy za pomocą metody klasy **Dao** .

Walidacja danych – przykład akcji

```
@RequestMapping(value = "/add", method = RequestMethod.POST)
public String processRegistration(@Valid Person person, BindingResult result) {
    if (result.hasErrors()) {
        return "person/add";
    }
    personDao.save(person);
    return "redirect:/success";
}
```

Jeżeli wystąpi błąd, wyświetlamy ponownie formularz dodawania. W formularzu tym będą już wcześniej wypełnione dane.

Zapis do bazy za pomocą metody klasy **Dao** .

Jeżeli dane są prawidłowe - przekierowujemy do innej strony.

Walidacja danych – przykład akcji

Rozważmy poniższy formularz:

```
@RequestMapping(value = "/add", method = RequestMethod.GET)
    public String showForm(Model model) {
      model.addAttribute("student", new Person());
    return "person/add";
}
```

oraz jego obsługę:

```
@RequestMapping(value = "/add", method = RequestMethod.POST)
public String saveProposition(@Valid Person student, BindingResult result) {
    if (result.hasErrors()) {
        return "person/add";
    }
     // save person
    return "redirect:/person/list";
}
```

Walidacja danych – przykład akcji

W naszym przykładzie otrzymamy błąd ponieważ **Spring** w przypadku wystąpienia błędów walidacji automatycznie utworzy w modelu zmienną o nazwie **person** podczas gdy oczekiwana przez nas ma nazwę **student**.

W takim przypadku musimy jawnie wskazać nazwę przy pomocy adnotacji **ModelAttribute** w poniższy sposób:

```
@RequestMapping(value = "/add", method = RequestMethod.POST)
public String saveProposition(@ModelAttribute("student") @Valid Person student,
            BindingResult result) {
    if (result.hasErrors()) {
        return "person/add";
    }
     // save person
    return "redirect:/person/list";
}
```

Zmienne w modelu

W przypadku problemów z bindowaniem danych przydatny może okazać się omawiany już sposób z wyświetlaniem danych z modelu:

```
@RequestMapping(value = "/test-model")
@ResponseBody
public void getAllFromMap(Model model) {
        model.asMap().forEach((k, v) -> logger.debug(k + ": " + v));
}
```



```
private final Logger logger = LoggerFactory.getLogger(HomeController.class);
```

Zmienne w modelu

W przypadku problemów z bindowaniem danych przydatny może okazać się omawiany już sposób z wyświetlaniem danych z modelu:

```
@RequestMapping(value = "/test-model")
@ResponseBody
public void getAllFromMap(Model model) {
        model.asMap().forEach((k, v) -> logger.debug(k + ": " + v));
}
```

Do wyświetlenia danych użyliśmy metody debug klasy **Logger**.

```
private final Logger logger = LoggerFactory.getLogger(HomeController.class);
```

Kolejność atrybutów akcji kontrolera

W większości przypadków kolejność parametrów przekazywanych do kontrolera nie ma znaczenia.

W tym jednak przypadku trzeba zapamiętać że jest inaczej.

Obiekt klasy **BindingResult** musi być bezpośrednio po obiekcie który jest sprawdzany. Poniższy kod (gdzie BindingResult znajduje się przed obiektem klasy Person) wywoła nam błąd.

```
public String processRegistration( BindingResult result, Person person) {
    //....
}
```


Błąd w dosyć oczywisty sposób opisuje co jest nie tak:

HTTP Status 500 - Request processing failed; nested exception is java.lang.IllegalStateException: An Errors/BindingResult argument is expected to be declared immediately after the model attribute,

Wyświetlanie błędów w formularzach

Aby dodać wyświetlanie błędu dla konkretnego pola należy w formularzu dodać tag `<form:errors>`

Przykładowo dla pola age definiujemy go następująco:

```
<form:errors path="age" />
```

Przykład pola formularza oraz pola do wyświetlania błędu:

```
<form:input path="age" />
<form:errors path="age"
             cssClass="error" />
```

atrybut `cssClass` - określa klasę **css** jaką otrzyma wygenerowany element typu **span** zawierający informację o błędzie:

```
<span id="age.errors" class="error">
  Musi mieć przynajmniej 18 lat</span>
```

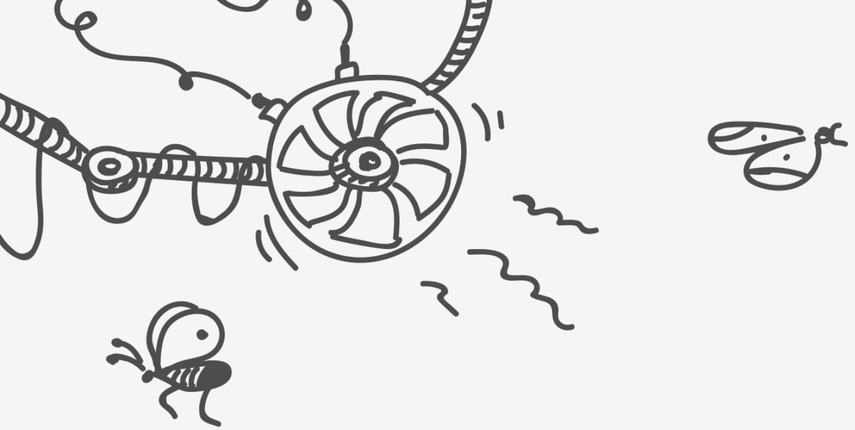
Wyświetlanie błędów w formularzach

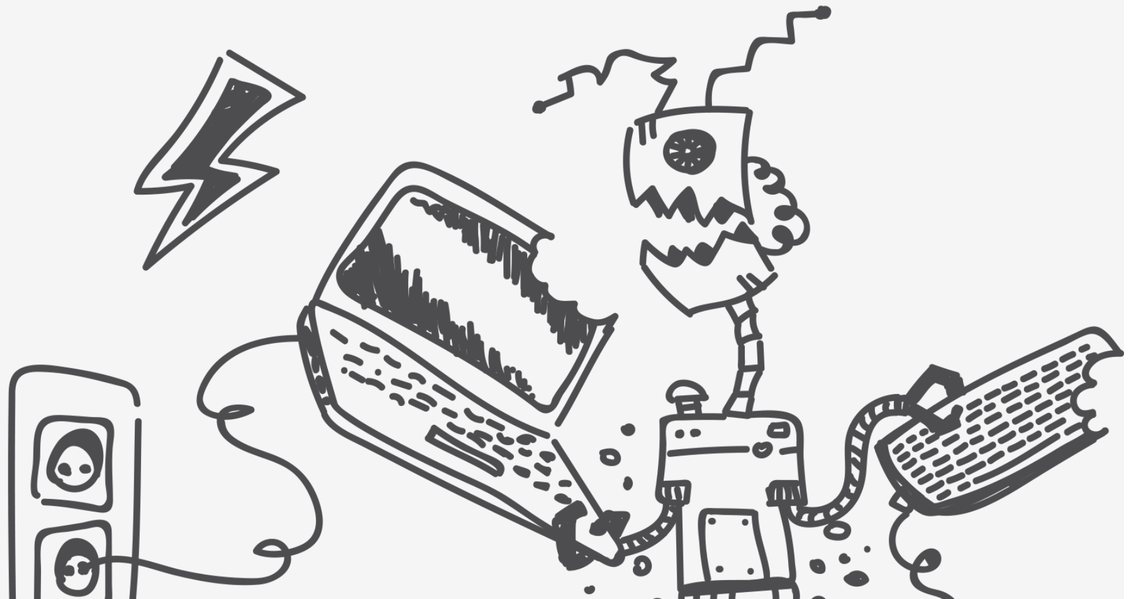
Aby zmienić element w jakim wyświetlają się komunikaty o błędach możemy dodatkowo zdefiniować atrybut o nazwie `element`.

```
<form:errors path="age" cssClass="error" element="div" />
```

Możemy również dodać tag wyświetlający wszystkie błędy w jednym miejscu

```
<form:errors path="*" />
```





Własny walidator

Własny walidator

Mimo wielu zdefiniowanych walidatorów, często zachodzi potrzeba do definicji naszych własnych ograniczeń.

Pierwszym krokiem jest zdefiniowanie **adnotacji**.

Następnie napiszemy klasę naszego walidatora.

Adnotacji tej użyjemy do oznaczenia pola w naszej encji, analogicznie jak to miało miejsce ze zdefiniowanymi już adnotacjami.

Tworzenie adnotacji

Adnotacje tworzymy bardzo podobnie jak interfejsy, należy pamiętać o znaku `@` przed słowem kluczowym **interface**.

```
package pl.coderslab.validator;
@Target({ElementType.METHOD, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface StartWith {
}
```

Za pomocą adnotacji **@Target** określamy element, nad którym możemy umieścić adnotację:

- **ElementType.METHOD** - nad metodą
- **ElementType.TYPE** - nad klasą lub interfejsem

Za pomocą adnotacji **@Retention** określamy moment wykrywania adnotacji:

RetentionPolicy.RUNTIME - wykrywane w trakcie wykonywania programu.

Własny walidator - adnotacja

Kompletny przykład

```
package pl.coderslab.validator;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import javax.validation.Constraint;
import javax.validation.Payload;
@Constraint(validatedBy = StartWithValidator.class)
@Target({ElementType.METHOD, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface StartWith {
        String message() default "{startWith.error.message}";
        Class<?>[] groups() default {};
        Class<? extends Payload>[] payload() default {}; }
```

Własny walidator - adnotacja

Kompletny przykład

```
package pl.coderslab.validator;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import javax.validation.Constraint;
import javax.validation.Payload;
@Constraint(validatedBy = StartWithValidator.class)
@Target({ElementType.METHOD, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface StartWith {
        String message() default "{startWith.error.message}";
        Class<?>[] groups() default {};
        Class<? extends Payload>[] payload() default {}; }
```

Podajemy klasę, która będzie realizować właściwą walidację.

Własny walidator - adnotacja

Kompletny przykład

```
package pl.coderslab.validator;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import javax.validation.Constraint;
import javax.validation.Payload;
@Constraint(validatedBy = StartWithValidator.class)
@Target({ElementType.METHOD, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface StartWith {
        String message() default "{startWith.error.message}";
        Class<?>[] groups() default {};
        Class<? extends Payload>[] payload() default {}; }
```

Nazwa Adnotacji.

Własny walidator - adnotacja

Kompletny przykład

```
package pl.coderslab.validator;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import javax.validation.Constraint;
import javax.validation.Payload;
@Constraint(validatedBy = StartWithValidator.class)
@Target({ElementType.METHOD, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface StartWith {
        String message() default "{startWith.error.message}";
        Class<?>[] groups() default {};
        Class<? extends Payload>[] payload() default {}; }
```

Wiadomość wyświetlana w przypadku wystąpienia błędu. Możemy jej wartość umieścić w omawianych plikach z tłumaczeniami.

Własny walidator - adnotacja

Kompletny przykład

```
package pl.coderslab.validator;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import javax.validation.Constraint;
import javax.validation.Payload;
@Constraint(validatedBy = StartWithValidator.class)
@Target({ElementType.METHOD, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface StartWith {
        String message() default "{startWith.error.message}";
        Class<?>[] groups() default {};
        Class<? extends Payload>[] payload() default {}; }
```

Te elementy są wymagane, aczkolwiek nie będziemy ich wykorzystywać.

Klasa walidatora

Tworząc walidację wskazaliśmy nazwę klasy, w której będzie realizowana walidacja.

Aby klasa mogła pełnić rolę walidatora musi implementować interfejs **ConstraintValidator**.

Interfejs ten posiada metodę `isValid`, która jest wywoływana przy wywołaniu metody `validate` np. podczas walidacji w kontrolerze:

```
Person p2 = new Person();
Set<ConstraintViolation<Person>> violations = validator.validate(p2);
```

Metoda `isValid` zostanie również wywołana niejawnie w momencie bindowania danych z formularza.

Metoda isValid

Metoda `isValid` przyjmuje 2 parametry:

Pierwszy jej argument jest wartością do walidacji.

Drugi parametr może posłużyć do dodania dodatkowego komunikatu błędu oraz wyłączenia domyślnego komunikatu.

Link do dokumentacji: https://docs.jboss.org/hibernate/validator/4.3/reference/en-US/html/validator-customconstraints.html#example-constraint-validator

Klasa walidatora

Przykładowa implementacja:

```
package pl.coderslab.validator;
import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;
public class StartWithValidator implements ConstraintValidator<StartWith, String> {
    @Override
    public void initialize(StartWith constraintAnnotation) {
    }
    @Override
    public boolean isValid(String value, ConstraintValidatorContext context) {
        return value.startsWith("A");
    }
}
```

Klasa walidatora

Przykładowa implementacja:

```
package pl.coderslab.validator;
import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;
public class StartWithValidator implements ConstraintValidator<StartWith, String> {
    @Override
    public void initialize(StartWith constraintAnnotation) {
    }
    @Override
    public boolean isValid(String value, ConstraintValidatorContext context) {
        return value.startsWith("A");
    }
}
```

Wskazujemy wcześniej określoną **adnotację** oraz typ danych jakiego będzie dotyczyć adnotacja.

Klasa walidatora

Przykładowa implementacja:

```
package pl.coderslab.validator;
import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;
public class StartWithValidator implements ConstraintValidator<StartWith, String> {
    @Override
    public void initialize(StartWith constraintAnnotation) {
    }
    @Override
    public boolean isValid(String value, ConstraintValidatorContext context) {
        return value.startsWith("A");
    }
}
```

Za pomocą tej metody możemy pobrać parametry określone w naszej adnotacji. Omówimy to rozwijając naszą adnotację.

Klasa walidatora

Przykładowa implementacja:

```
package pl.coderslab.validator;
import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;
public class StartWithValidator implements ConstraintValidator<StartWith, String> {
    @Override
    public void initialize(StartWith constraintAnnotation) {
    }
    @Override
    public boolean isValid(String value, ConstraintValidatorContext context) {
        return value.startsWith("A");
    }
}
```

W tej metodzie umieszczamy logikę walidacji, zwracając wartość typu boolean.

Dodanie walidacji

Aby dodać nowy walidator wystarczy odpowiednio opatrzyć pole naszej encji utworzoną adnotacją:

```
@NotNull
@Size(min=2, max=30)
@StartWith()
private String firstName;
```

Parametr adnotacji

Wewnątrz definicji adnotacji możemy określić jakie parametry może ona przyjmować.

Deklarację przedstawiamy schematycznie następująco:

```
Typ nazwaParametru();
```

Np.

```
public @interface StartWith {
    String version();
}
```

Dodając adnotację do elementu określamy ją następująco:

```
@StartWith(version="1.0")
int rating;
```

Parametr adnotacji

Szczególny przypadek stanowi adnotacja, która ma jeden parametr, możemy wtedy pominąć jego nazwę, a przyjmuje on domyślnie nazwę **value**.

```
public @interface StartWith {
    String value();
}
```

Dodając adnotację do elementu określamy ją następująco:

```
@StartWith("annotValue")
int rating;
```

Dane adnotacji mogą być uzyskiwane od nich poprzez wywołanie nazwy metody w niej określonej, np.

```
@Override
public void initialize(StartWith
    constraintAnnotation) {
    this.start =
        constraintAnnotation.value();
}
```

Wykorzystywane są w tym celu mechanizmy refleksji lub narzędzia przetwarzania w fazie kompilacji - zagadnień tych nie omawiamy w trakcie kursu.

Parametr adnotacji

Szczególny przypadek stanowi adnotacja, która ma jeden parametr, możemy wtedy pominąć jego nazwę, a przyjmuje on domyślnie nazwę **value**.

```
public @interface StartWith {
    String value();
}
```

Dodając adnotację do elementu określamy ją następująco:

```
@StartWith("annotValue")
int rating;
```

Dane adnotacji mogą być uzyskiwane od nich poprzez wywołanie nazwy metody w niej określonej, np.

```
@Override
public void initialize(StartWith
    constraintAnnotation) {
    this.start =
        constraintAnnotation.value();
}
```

Wywołujemy metodę `value()` .

Wykorzystywane są w tym celu mechanizmy refleksji lub narzędzia przetwarzania w fazie kompilacji - zagadnień tych nie omawiamy w trakcie kursu.

Parametr adnotacji

Aby uczynić nasz walidator bardziej przydatnym dodamy możliwość definiowania parametru adnotacji.

W naszym przypadku jest to ciąg znaków, od którego ma się zaczynać adnotowany atrybut.

```
@Constraint(validatedBy = StartWithValidator.class)
@Target({ElementType.METHOD, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface StartWith {
    String value();
    String message() default "{startWith.error.message}";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

Parametr adnotacji

Aby uczynić nasz walidator bardziej przydatnym dodamy możliwość definiowania parametru adnotacji.

W naszym przypadku jest to ciąg znaków, od którego ma się zaczynać adnotowany atrybut.

```
@Constraint(validatedBy = StartWithValidator.class)
@Target({ElementType.METHOD, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface StartWith {
    String value();
    String message() default "{startWith.error.message}";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

W tym celu adnotację uzupełniamy o możliwość dodania wartości typu String.

Walidator - parametr z adnotacji

Modyfikujemy klasę walidatora:

```
public class StartWithValidator implements ConstraintValidator<StartWith, String>{
    private String start;
    @Override
    public void initialize(StartWith constraintAnnotation) {
        this.start = constraintAnnotation.value(); }
    @Override
    public boolean isValid(String value, ConstraintValidatorContext context) {
        return value.startsWith(start); }
}
```

Wykorzystujemy metodę `initialize` do pobrania parametru z adnotacji oraz modyfikujemy metodę `isValid` by z niego skorzystała.

Dodanie adnotacji

```
@StartWith("A")
private String firstName;
```

Grupy walidacji

W zależności od kontekstu różne dane powinny być walidowane.

Jako przykład można podać szkic artykułu w systemie CMS, dodając go nie wymagamy wypełnienia wszystkich pól. Możemy np. wymagać jedynie pola tytuł.

Należy pamiętać że podczas wywoływania walidacji, pola z grupy domyślnej są zawsze sprawdzane. Z tego powodu grupa ta powinna składać się z elementów wymaganych zawsze.

Podobnie sprawa będzie wyglądać w przypadku zamówienia w sklepie internetowym.

Zachowanie takie możemy uzyskać, definiując grupy walidacji.

Dzięki grupom walidacji możemy decydować, jakie ograniczenia mają być nakładane w zależności od sytuacji.

Innych danych rejestracyjnych możemy wymagać od firmy, a innych od użytkownika indywidualnego.

Tworzenie grupy

Pierwszym krokiem jest utworzenie interfejsu:

```
public interface ValidationGroupName {}
```

Może on mieć dowolną nazwę.

Jego nazwa będzie jednocześnie nazwą naszej grupy walidacji.

Następnie w encji dla poszczególnych pól określamy jakiej grupy walidacji ona dotyczy:

```
public class Person2 {
    @NotNull
    @Size(min = 3, max = 20)
    private String firstName;
    @NotNull(groups =
        ValidationGroupName.class)
    private String lastName;
 }
```

Tworzenie grupy

Możemy zdefiniować więcej niż jedną grupę:

```
public class Person2 {
    @NotNull
    @Size(min = 3, max = 20)
    private String firstName;
    @NotNull(groups={ValidationGroupName.class, Default.class})
    private String lastName;
 }
```

Default - jest nazwą domyślnej grupy, która jest wywoływana podczas walidacji.

Walidacja określonej grupy

Aby walidować określoną grupę podajemy jej nazwę w metodzie **validate** jako drugi parametr.

```
Person p2 = new Person();
Set<ConstraintViolation<Person>> violations =
    validator.validate(p2,ValidationGroupName.class);
if (!violations.isEmpty()) {
  for (ConstraintViolation<Person> constraintViolation : violations) {
      System.out.println(constraintViolation.getPropertyPath() + " "
                         + constraintViolation.getMessage());
  }
} else {
       // save object
}
```

Walidacja określonej grupy

Aby walidować określoną grupę podajemy jej nazwę w metodzie **validate** jako drugi parametr.

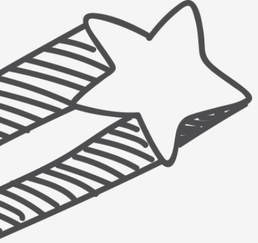
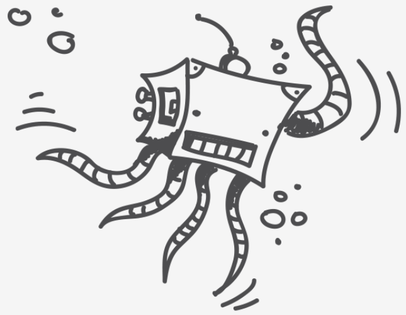
```
Person p2 = new Person();
Set<ConstraintViolation<Person>> violations =
    validator.validate(p2,ValidationGroupName.class);
if (!violations.isEmpty()) {
  for (ConstraintViolation<Person> constraintViolation : violations) {
      System.out.println(constraintViolation.getPropertyPath() + " "
                         + constraintViolation.getMessage());
  }
} else {
        // save object
}
```

Wywołujemy metodę `validate` określając jako drugi parametr grupę walidacji `ValidationGroupName.class` jeżeli będziemy chcieli określić domyślną grupę jako drugi parametr podajemy `Default.class` .

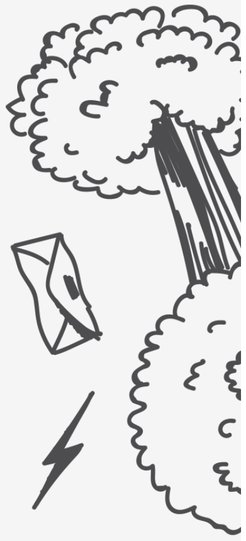
Walidacja określonej grupy - formularz

Aby walidować określoną grupę w formularzu korzystamy z adnotacji **@Validated**.

```
@RequestMapping(value = "/add", method = RequestMethod.POST)
public String processRegistration(@Validated({Default.class}) Person person,
                                  BindingResult result) {
    if (result.hasErrors()) {
        return "person/add";
    }
    personRepository.save(person);
    return "redirect:/success";
}
```





Czas
na zadania

Wykonaj zadania z działu:
Walidacja Formularze