Przedmiot: Platformy programowania

III Informatyka, studia stacjonarne

Laboratorium 4

Formatowanie, walidacja i niestandardowe wiązanie danych.

- Formatowanie danych za pomocą adnotacji,
- Wykrywanie błędów w formularzu i ich wyświetlanie,
- Błędy wiązania danych,
- Wykorzystanie Bean Validation API do walidacji danych,
- Formatowanie danych z użyciem edytorów właściwości i obiektów formatujących.

Przydatne linki:

1. http://www.baeldung.com/javax-validation
2. http://docs.jboss.org/hibernate/validator/6.0/reference/en-US/html_single/
3. https://mvnrepository.com/ - repozytorium MAVENA!!!!!

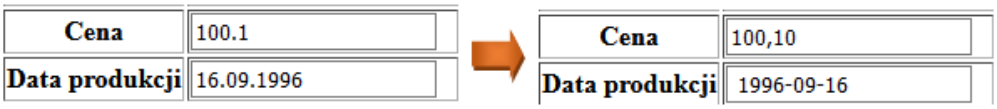
UWAGA!!!: Potrzebny będzie artefakt `spring-boot-starter-validation`

Formatowanie danych za pomocą adnotacji

Jeśli chcemy, w formularzu obsługiwać dane w pewnym formacie tekstowym, musimy go określić. Typowymi danymi wymagającymi formatowania mogą być daty i liczby. Możemy tego dokonać za pomocą adnotacji `@DateTimeFormat` i `@NumberFormat` umieszczanych nad polami komponentu JB powiązanymi z formularzem lub ich metodami dostępowymi.

```
@NumberFormat(pattern = "#.00")
private Float price;

@DateTimeFormat(pattern = "yyyy-MM-dd")
private LocalDate productionDate;
```



Jeśli chcemy skorzystać z typów pól z HTML5, takich jak *date* …

```
<input type="date" th:field="*{productionDate}"/>
```

…to pojawia się również problem poprawnej konwersji danych

Ponadto, adnotacje możemy umieszczać bezpośrednio przed argumentem metody kontrolera, którego wartość ma powstać na podstawie parametru żądania lub zmiennej URL.

```
@PostMapping("/process-date")
public void localDate(@DateTimeFormat(iso = DateTimeFormat.ISO.DATE)
 LocalDate localDate) {
    // ...
}
```

Zadanie 1 (1pkt). Przy użyciu adnotacji obsłuż formatowanie dat i liczb w formularzu

Wyrażenia Thymeleaf do obsługi informacji o błędach

Kontrola poprawności danych wprowadzanych przez użytkownika do pól formularzy, jest kluczowa dla każdej aplikacji. Błędy związane z przetwarzaniem danych z formularza, są zwracane przez wyrażenie `#fields.errors('nazwa_pola')`. Z kolei wyrażenie `#fields.hasErrors('nazwa_pola')` umożliwia sprawdzenie, czy dla danego pola są dostępne jakiekolwiek informacje o błędach.

Użycie wyrażeń Thymeleaf w celu wyświetlenia informacji o błędzie dla pola productionDate obiektu polecenia *vehicle*.

```
<table>
<form th:method="POST" th:object="${vehicle}"
th:action="@{/vehicleForm.html}">
      ...
      <tr>
         <th>Data Produkcji:</th>
         <td>
         <input type="date" th:field="*{productionDate}"/>
    <p th:if="${#fields.hasErrors('productionDate')}" th:each="err :
${#fields.errors('productionDate')}" th:text="${err}"></p>
         </td>
      </tr>
      ...
   </form>
</table>
```

Interfejsy `BindingResult` i `Errors`

Wyniki związane z walidacją danych formularza są dostępne w obiektach `BindingResult` i `Errors`. Aby użyć wybranego obiektu należy umieścić go jako parametr metody kontrolera obsługującej formularz (**koniecznie musi on być umieszczony zaraz po obiekcie polecenia**). W przypadku napotkania błędów, zwykle realizuje się powtórne wyświetlenie formularza.

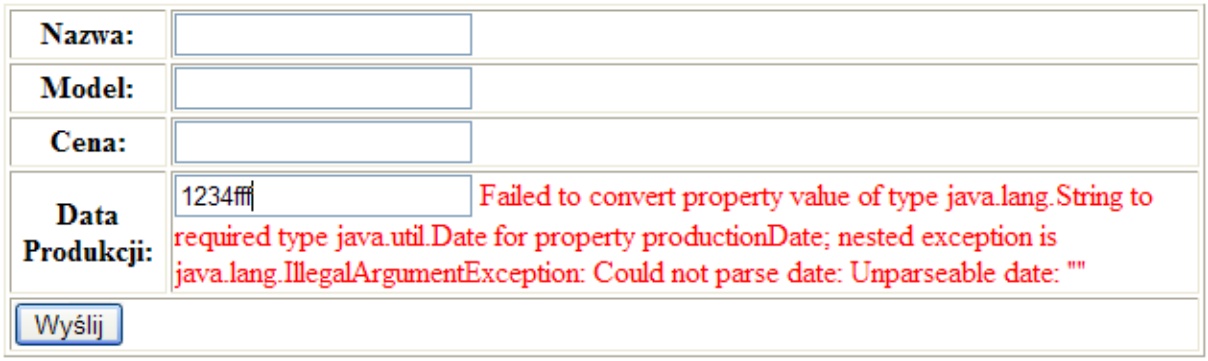
Interpretacja wyników związanych z walidacją formularza

```
@RequestMapping(method=RequestMethod.POST)
public String processForm(@ModelAttribute("vehicle") Vehicle v,
BindingResult result){

    if(result.hasErrors()){
        return "vehicleForm";
    }
    return "successView";
}
```

Błędy wiązania danych

Użytkownik w pole formularza typu **`text`**, może wprowadzić dowolny ciąg znaków. Jeżeli takie pole jest powiązane z właściwością **obiektu polecenia** (*czyli obiektu przechwytującego dane z formularza*), która jest np., typu `Integer`, `Float` lub `Date`, może pojawić się błąd związany z konwersją typów – nie każdy łańcuch da się przekształcić np., na typ `Float`.



Jak można zauważyć na powyższym zrzucie ekranu, Spring (a dokładnie komponent klasy **`DataBinder`**) napotkał problem z wiązaniem danych, gdyż wprowadzonego przez użytkownika ciągu znaków, nie udało się przekonwertować na obiekt klasy `Date`. Komunikat ten jest jednak mało czytelny dla przeciętnego użytkownika.

Aby zamienić systemowe komunikaty, związane z błędami wiązania danych, na bardziej przyjazne, należy:

1. W katalogu `resources` utworzyć plik właściwości (`.properties`) np., o nazwie `errors-messages.properties`, a w nim ... dla każdej właściwości obiektu polecenia, która może sprawić problemy podczas konwersji danych, dodać wpis w postaci `klucz`=`wartość` w następującym ogólnym formacie:

   `typBłędu.`**`nazwa_obiektu_polecenia_w_modelu`**`.właściwość`=`opis błędu`

   np.

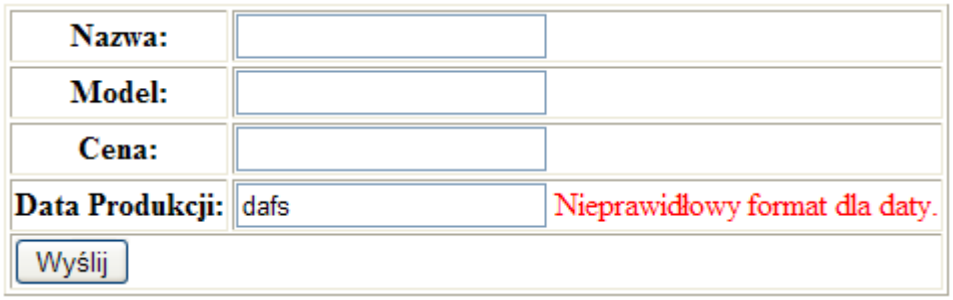
   `typeMismatch.`**`vehicle`**`.productionDate`=`Nieprawidłowy format daty`

   tutaj przedrostek `typeMismatch` *oznacza typ błędu związany z konwersją danych.*

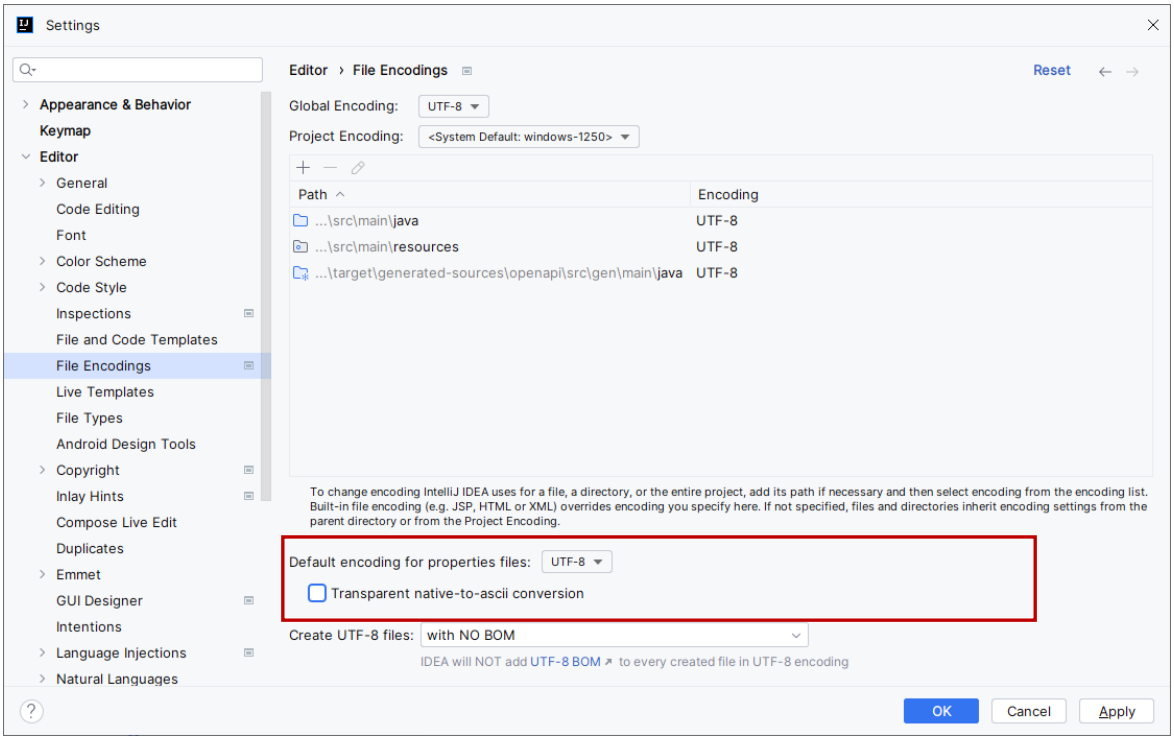
2. W pliku `application.properties` wskazać plik z komunikatami.

   `spring.messages.basename``=`**`nazwa_pliku_bez_rozszerzenia`**

Jeśli istnieje potrzeba wskazania wielu plików z komunikatami, ich nazwy oddzielamy przecinkiem. W ten sposób można dodać np., etykiety pól i wykorzystywać je w wyrażeniach Thymeleaf.



Kodowanie plików właściwości (*.properties*) można ustawić w **File | Settings** a następnie zakładka **Editor | File Encodings**. Należy tutaj wybrać UTF-8.



Zadanie 2 (2pkt). Wyświetlanie błędów związanych z wiązaniem danych,

1. Dodaj do aplikacji obsługę wiadomości dotyczących problemów wiązania danych,
2. Wartości komunikatów powinny być umieszczone w pliku właściwości,

Walidacja danych sterowana adnotacjami

Spring Framework posiada własne mechanizmy walidacji bazujące na interfejsie `Validator` [1]i klasie narzędziowej `ValidationUtils`. Nie mniej jednak Spring oferuje także wsparcie dla standardu JSR-303 API (Bean Validation 1.0) [2], JSR-349 API (Bean Validation 1.1) i JSR-380 API(Bean Validation 2.0), które służą do walidacji danych z użyciem adnotacji. Specyfikacje tych standardów znajdziesz pod adresem http://beanvalidation.org/specification/

Oto niektóre z nich:

Adnotacja	**Opis**
`@Email`	sprawdza czy pole jest poprawnym adresem email
`@NotBlank`	sprawdza czy pole jest wypełnione znakami białymi (spacjami, itd.)
`@NotEmpty`	sprawdza czy pole jest pustym łańcuchem znaków. Należy stosować dla łańcuchów znaków.
`@NotNull`	sprawdza czy wartość pola nie jest `null`. Należy stosować dla właściwości, które nie są łańcuchami znaków, ani nie są też typami prostymi. Czyli należy stosować dla pól typu `Integer`, `Float`, itd.
`@Min`	sprawdza, czy wartość pola jest większa równa wskazanej.
`@Max`	sprawdza, czy wartość pola jest mniejsza równa wskazanej.
`@Pattern(regex=)`	sprawdza zawartość pola jest zgodna z wyrażeniem regularnym. W Java do obsługi wyrażeń regularnych służy klasa `Pattern`, o której

[1] https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#validation
[2] http://docs.jboss.org/hibernate/validator/6.0/reference/en-US/pdf/hibernate_validator_reference.pdf

	możesz więcej dowiedzieć tutaj: https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html, natomiast o zasadach tworzenia wyrażeń tutaj: http://www.vogella.com/tutorials/JavaRegularExpressions/article.html
`@Future`	sprawdza czy data jest przyszła.
`@Past`	sprawdza czy dana jest przeszła.
`@Size(min=, max=)`	sprawdza czy pole jest rozmiarów, od `min` do `max`. Dotyczy to również długości list.
`@URL(protocol=, host=, port=)`	sprawdza, czy pole jest prawidłowym `URL`-em
`@AssertTrue` i `@AssertFalse`	Sprawdzają czy flaga jest ustawiona lub nie. Przydatne przy „Zapoznałem się z regulaminem?" lub „Czy jesteś pełnoletni?".

Ponadto, *Hibernate Validator* rozszerza[3] Bean Validation Api o dodatkowe adnotacje. M.in. `@CreditCardNumber`, `@Currency`, `@Range`, zlokalizowane w pakiecie *org.hibernate.validator.constraints*. W pakiecie *org.hibernate.validator.constraints.pl* mamy specyficzne dla naszego kraju `@PESEL`, `@NIP` i `@REGON`.

`@Valid` – pozwala na automatyczne wywołanie walidatora dla wskazanego obiektu polecenia. Walidacja będzie wykonywana kaskadowo.

Przykładowa walidacja danych z wykorzystaniem adnotacji z Bean Validation API.

```
public class Vehicle implements Serializable{

   @NotBlank
   @Size(min=5, max=20, message="Size must be between {min} and {max}
marks")
   private String name;

   @NotBlank
   private String model;

   @Min(100) @Max(1000)
   private Double price;

   @Past
   private Date productionDate;

   @Valid
   private ComplexProperty complexProperty;

}
```

Uwaga: Aby włączyć automatyczną walidację danych na poziomie adnotacji należy w metodzie obsługującej formularz, dodać adnotację `@Valid` bezpośrednio przed parametrem, którym jest obiekt polecenia. Walidacja zostanie przeprowadzona przez domyślny walidator danych.

Uwaga: W powyższym przykładzie użyto atrybutu *message* adnotacji. Takie podejście nie jest zalecane, gdyż nie powinno umieszczać się tekstu w kodzie źródłowym programu. Zamiast tego należy umieścić odpowiedni komunikat w pliku z komunikatami.

[3] http://docs.jboss.org/hibernate/validator/6.0/reference/en-US/html_single/#validator-defineconstraints-hv-constraints

Przykład komunikatu:

```
Size.vehicle.name=Nazwa musi składać się z 5-20 znaków
```

lub lepiej

```
Size.vehicle.name=Nazwa musi składać się z {2}-{1} znaków
```

gdzie {2} i {1} odpowiadają argumentom *min* i *max* (indeksy argumentów wynikają z rozmieszczenia alfabetycznego) adnotacji `@Size`.

Zadanie 3 (2pkt). Walidacja sterowana adnotacjami

1. Do swojego głównego komponentu JB dodaj adnotacje z Bean Validation API – każda właściwość wyświetlana w formularzu ma mieć ograniczenia, wykorzystaj różne adnotacje
2. Dokonaj odpowiednich wpisów w pliku z komunikatami o błędach,

Niestandardowe operacje towarzyszące związaniu danych

Spring Framework (jak wiele innych frameworków) zapewnia automatyczne wiązanie danych z formularzy z właściwościami obiektu polecenia (*komponentem powiązanym z formularzem*). W procesie wiązania parametrów formularza z polami obiektu polecenia następuje automatyczna konwersja typów danych. Należy zauważyć, że wszystkie parametry formularza są przesyłane jako tekst. Następnie parametry formularza są zamieniane na typy języka Java `int`, `long`, `float` itd. i odwrotnie. W Spring Framework wiązaniem danych zajmuje się komponent **`WebDataBinder`**.

Jeśli dane z formularza muszą być obsłużone **niestandardowo** należy, w komponencie `WebDataBinder`, dokonać rejestracji obiektów, które się tym zajmą. W tym celu należy w kontrolerze dodać metodę (o dowolnej nazwie) z argumentem typu `WebDataBinder` i adnotacją **`@InitBinder`**. Adnotacja ta posiada opcjonalny atrybut **`value`**, który może przyjmować następujące wartości:

- *nazwa obiektu polecenia w modelu,*
- *nazwa parametru żądania, przechwytywanego w argumencie metody,*

Jeśli nie uzupełnimy wartości value to metoda opatrzona adnotacją **@InitBinder** zostanie wywołana dla wszystkich parametrów żądań i wszystkich atrybutów modelu.

Niestandardowa obsługa związana z wiązaniem danych może dotyczyć m.in.:

- *wykluczeń parametrów,*
- *walidacji danych,*
- *konwersji danych między typami za pomocą formaterów danych lub edytorów właściwości.*

Wykluczenia parametrów zapobiega to atakom typu ***Http parameter pollution***[4]. W poniższym przykładzie parametry żądania *createdDate* i *id*, mimo że mogą zostać przesłane, to nie zostaną związane z obiektem polecenia.

```
@InitBinder ("vehicle")
public void initBinder(WebDataBinder binder) {
    binder.setDisallowedFields("createdDate","id");//ważne ze względu na
bezpieczeństwo aplikacji
    ...
}
```

[4] https://niebezpiecznik.pl/post/ataki-http-parameter-contamination-i-pollution-hpc-oraz-hpp/

Walidacja danych – mimo, że Spring Framework wspiera *Bean Validation API*, to też dostarcza własnego mechanizmu związanego z walidacją bazującego na interfejsie `Validator` oraz klasie narzędziowej `ValidationUtils`. Aby skorzystać z tego mechanizmu należy:

1. Zaimplementować interfejs `Validator`.

```
public class CustomVehicleValidator implements Validator {

    @Override//Wspierana klasa
    public boolean supports(Class<?> aClass) {
        return Vehicle.class.isAssignableFrom(aClass);
    }

    @Override//Logika związana z poprawnością danych w obiekcie
    public void validate(Object o, Errors errors) {

        ValidationUtils.rejectIfEmpty(errors, "model",
"Empty.vehicle.model");
        var vehicle = (Vehicle) o;
        if (vehicle.getPrice() < 0) {
            errors.rejectValue("price", "Negative.vehicle.price");
        } else if (vehicle.getPrice() > 100) {
            errors.rejectValue("price", "Toomuch.vehicle.price");
        }
    }
}
```

2. Zarejestrować obiekt walidatora

```
@InitBinder("vehicle")
public void initBinder(WebDataBinder binder) {
    binder.addValidators(new CustomVehicleValidator());
    ...
}
```

Zadanie 4 (2pkt). Utwórz i zarejestruj springowy walidator, który uniemożliwi wprowadzenie identycznych wartości do dwóch wybranych przez ciebie pól formularza.

Formatery danych[5] służą do przekształcania danych pochodzących z formularzy (czyli łańcuchów znaków), w bardziej skomplikowane obiekty(np., `Date`) i odwrotnie.
Można je wykorzystać zamiast formaterów sterowanych adnotacjami opisanych wcześniej.

Rejestrowanie formaterów danych w obiekcie DataBinder

```
@InitBinder
public void initFormatters(WebDataBinder binder) {
    //formatowanie cen
    var currencyStyleFormatter = new CurrencyStyleFormatter();
    currencyStyleFormatter.setCurrency(Currency.getInstance("USD"));
    binder.addCustomFormatter(currencyStyleFormatter, "price");
    //formatowanie dat
    binder.addCustomFormatter(new DateFormatter("dd-MM-yyyy"));
}
```

Formatery danych można też rejestrować globalnie i to je głównie odróżnia od edytorów danych.
Można także utworzyć własną klasę formatującą poprzez implementację interfejsu `Formatter` i jej metod `parse` i `print`.

Implementacja niestandardowego formatera danych.

[5] https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#format-Formatter-SPI

```
public class RangeFormater implements Formatter<Range> {
    @Override//zamiana obiektu na tekst
    public String print(Range range, Locale locale) {
        if (range == null) {
            return "";
        }
        return String.format("%.0f-%.0f", range.getFrom(), range.getTo());
    }
    @Override//zamiana tekstu na obiekt
    public Range parse(String formatted, Locale locale) throws
ParseException {
        if (formatted.length() == 0) {
            return null;
        }
        var tokens = formatted.split("-");
        if(tokens.length>1){
            return new Range(Float.valueOf(tokens[0]),
Float.valueOf(tokens[1]));
        }else{
            var maxSpeed = 220f;
            return new Range(Float.valueOf(tokens[0]), maxSpeed);
        }
    }
}
```

Edytory właściwości [6]są podobne do formaterów i są od nich starszą koncepcją pochodzącą ze specyfikacji Java Beans. Formatery i edytory właściwości są uruchamiane przed operacją wiązania danych (z formularza) z właściwościami obiektu polecenia.

Wykorzystanie edytora klasy java.util.Date do obsługi właściwości formularza.

```
@InitBinder
public void initBinder(WebDataBinder binder) {
   SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
   dateFormat.setLenient(false);
   binder.registerCustomEditor(Date.class, new
CustomDateEditor(dateFormat, false));
}
```

Można zaimplementować własny editor właściwości poprzez rozszerzenie klasy `PropertyEditorSupport`.

Implementacja niestandardowego edytora danych.

```
public class CustomTypeEditor extends PropertyEditorSupport {

    @Override//zamiana obiektu na tekst
    public String getAsText() {
        var mojTyp = (CustomType) getValue();
        return mojTyp == null ? "" : mojTyp.getName();
    }

    @Override//zamiana tekstu na obiekt
    public void setAsText(String text) throws IllegalArgumentException {
        var mojTyp = new CustomType ();
        mojTyp.setName(text.toUpperCase());
        setValue(mojTyp);
    }
}
```

[6] https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-beans-conversion

Zadanie 5(3pkt). **Implementacja klasy formatującej dane lub edytora właściwości.**

1. Zaimplementuj kolejny komponent JB, który będzie zagregowaną własnością głównego komponentu np., *FormatPizzy*, *Adres*, *Rozmiar*, a w nim kilka właściwości np. `width`, `height`, `thickness`. , czy *ulica*, *nrBudynku*, *kodPocztowy, miejscowość*.
2. W głównym komponencie JB dodaj pole typu z punktu 1, metody dostępowe oraz twórz instancję nowego pola w bezargumentowym konstruktorze głównego JB.
3. Zaimplementuj *formatter* lub *edytor właściwości* tak, aby do pojedynczego pola formularza można było w wpisać ciąg znaków, który zostanie zamieniony na obiekt. Przykładowe formaty
    a. *wys x szer x gr*
    b. *średnica x grubość,*
    c. *ulica ; nrBudynku ; kodPocztowy ; miejscowość*