Spring + Thymeleaf: layout i walidacja formularzy

Przygotowanie projektu

- Pobierz dołączony do ćwiczenia projekt
- Rozpakuj, otwórz w Netbeans
- Projekt jest przygotowany dla Javy 1.7 - jeśli pracujesz na 1.8, zmień wersję w `pom.xml`

Kalkulator - przygotowany kod

Przyjrzyj się zawartości projektu `lab8`. Znajdziesz w nim gotowy, działający kalkulator:

- klasa kontrolera `CalculatorController` - obsługuje trzy akcje: `add`, `multiply` i `calculate`;
- klasa formularza `CalculatorForm`;
- dwa pliki widoków odpowiadające za:
  - `index.html` - wyświetlenie formularzy,
  - `result.html` - wyświetlenie wyniku obliczeń.

Wydzielenie logiki z kontrolera do usługi

- W tej chwili cała logika (przeprowadzanie obliczeń) jest umieszczona w samym kontrolerze. Tak być nie powinno - wydzielimy logikę do osobnej klasy `CalculatorService`.

Tworzenie usługi CalculatorService

- Do pakietu `wizut.tpsi.calculator` dodaj klasę `CalculatorService`. Umieść w niej metody `add`, `subtract`, `multiply` i `calculate`:

```
public int add(int a, int b) {
    return a + b;
}

public int subtract(int a, int b) {
    return a - b;
}

public int multiply(int a, int b) {
    return a * b;
}

public int calculate(int a, int b, String operation) {
```

```
    ...
}
```

- Do metody `calculate()` przenieśmy logikę z kontrolera, z metody `doCalculations()`:
  - metoda jako argumenty dostaje dwie liczby i operację;
  - spradzamy operację i wywołujemy odpowiednią metodę `add()`, `subtract()` lub `multiply()`;
  - w przypadku nieznanej operacji, rzucamy wyjątkiem.

```
public int calculate(int x, int y, String operation) {
    switch(operation) {
        case "+":
            return add(x, y);
        case "-":
            return subtract(x, y);
        case "*":
            return multiply(x, y);
    }
    throw new IllegalArgumentException("Nieznana operacja: " + operation);
}
```

Modyfikacja kontrolera CalculatorController

- Musimy teraz zmodyfikować kontroler tak, by korzystał z usługi `CalculatorService`.
- Dodaj do kontrolera pole, w którym przychowamy obiekt klasy `CalculatorService`:

```
@Controller
public class CalculatorController {

    private CalculatorService calculatorService;

    ...
```

- Na razie to pole pozostaje puste - dostarczeniem do niego obiektu klasy `CalculatorService` zajmiemy się za chwilę.

Zadanie 1

Zmodyfikuj kod wszystkich akcji, tak by wykorzystywały wywołania usługi `calculatorService`, zamiast samodzielnie wykonywać obliczenia; na przykład w metodzie `add()`:

```
int result = form.getX() + form.getY();
```

zmień na:

```
int result = calculatorService.add(form.getX(), form.getY());
```

Podobnie przerób metody `multiply()` i `doCalculations()`.

Wstrzykiwanie zależności

- Pozostaje kwestia stworzenia obiektu klasy `CalculatorService`.
- Aby zachować luźne wiązanie między klasami, zrzucimy na Spring Framework odpowiedzialność za utworzenie obiektu usługi i dostarczenie go do kontrolera. Użyjemy do tego wstrzykiwania zależności (*Dependency Injection*).
- W tym celu:
  - z klasy `CalculatorService` zrobimy komponent zarządzany przez Spring Framework (adnotacja `@Service`),
  - w klasie `CalculatorController` wstrzykniemy komponent (adnotacja `@Autowired`).
- W kodzie `CalculatorService`, przed deklaracją klasy, dodaj adnotację `@Service`:

```
@Service
public class CalculatorService {
    ...
}
```

- W kodzie kontrolera `CalculatorController`, przed deklaracją pola z obiektem usługi, dodaj adnotację `@Autowired`:

```
@Controller
public class CalculatorController {

    @Autowired
    private CalculatorService calculatorService;

    ...
}
```

- Sprawdź działanie aplikacji - wszystkie formularze powinny działać tak jak na początku.

Szablony layoutu w Thymeleaf

- Podzielimy teraz naszą aplikację na podstrony:
  - strona z operacją dodawania,
  - strona z mnożeniem,
  - strona z uniwersalnym kalkulatorem.

- Do projektu dodamy szablon `layout.html`, zawierający szkielet wszystkich podstron (elementy wspólne, np. sekcja `head` ze stylami CSS, ładowaniem JavaScript itp.; nagłówek z linkami nawigacyjnymi, stopka, itp.).
- W katalogu szablonów `templates` dodaj nowy plik `layout.html`. Umieść w nim następujący kod:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/web/thymeleaf/layout">
    <head>
        <title layout:title-pattern="MiniMatlab 0.1 alpha - $CONTENT_TITLE">MiniMatlab
        <meta charset="UTF-8" />
    </head>
    <body>
        <div class="header">
            <h1>MiniMatlab 0.1 alpha</h1>
        </div>

        <div class="navMenu">
            <a href="/add">Dodawanie</a> |
            <a href="/multiply">Mnożenie</a> |
            <a href="/calculate">Uniwersalny kalkulator</a>
        </div>

        <div layout:fragment="content">
            Tu będzie wstawiona właściwa zawartość widoku.
        </div>
    </body>
</html>
```

- Zwróć uwagę na następujące fragmenty:
  - deklaracja przestrzeni nazw `xmlns:layout` - z niej pochodzić będę atrybuty Thymeleaf definiujące layout;
  - element `title` - w miejsce znacznika `$CONTENT_TITLE` będzie wstawiony tytuł konkretnej podstrony, zdefiniowany w konkretnym widoku;
  - element `div` z atrybutem `layout:fragment` - w tym miejscu będzie wstawiona zawartość konkretnego widoku (dla konkretnej akcji/podstrony).
- Musimy teraz zmodyfikować szablon `index.html`, tak by korzystał z powyższego layoutu. W pliku `index.html`:
  - dodaj deklarację layoutu w elemencie `html`:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
```

```
xmlns:th="http://www.thymeleaf.org"
xmlns:layout="http://www.ultraq.net.nz/web/thymeleaf/layout"
layout:decorator="layout">
```

  - całą zawartość strony opakuj w element `section` z odnośnikiem do layoutu:

```
<body>
    <section layout:fragment="content">
        ... tutaj cała wcześniejsza zawartość body ...
    </section>
</body>
```

  - usuń nagłówek z tytułem strony (`<h1>...`) - tę część mamy w szablonie layoutu.

- Sprawdź działanie aplikacji. Linki do podstron jeszcze nie działają, ale formularze obliczeń powinny działać jak wcześniej.

Zadanie 2

- Przerób stronę `result.html` tak by też korzystała z layoutu.

Obsługa linków do podstron

- Aby poprawnie obsłużyć linki do podstron, musimy zmienić mapowanie w kontrolerze:
  - istniejące metody obsługujące adresy `/add`, `/multiply` i `/calculate` przypiszemy do obsługi tylko żądań `POST` - użyjemy ich do obsługi przesłanych formularzy, nie do samego wyświetlenia formularzy;
  - dodamy nowe metody, przypisane do tych samych adresów (`/add`, `/multiply` i `/calculate`), ale tylko do żądań `GET` - użyjemy ich do do samego wyświetlenia formularzy.
- Zmień adnotacje mapujące metody `addNumbers()`, `multiplyNumbers()` i `doCalculations()`: zamiast `@RequestMapping(...)` wstaw `@PostMapping(...)`.
- Dodaj nowe metody do obsługi żądań `GET` i wyświetlenia formularzy:

```
@GetMapping("/add")
public String addForm(CalculatorForm form) {
    return "add";
}

@GetMapping("/multiply")
public String multiplyForm(CalculatorForm form) {
    return "multiply";
}
```

```
@GetMapping("/calculate")
public String calculateForm(CalculatorForm form) {
    return "calculate";
}
```

- Pozostaje nam przygotowanie widoków `add.html`, `multiply.html` i `calculate.html` oraz usunięcie formularzy ze strony głównej.

Zadanie 3

- Przenieś zawartość `index.html` do trzech osobnych podstron: `add.html`, `multiply.html`, `calculate.html`. W każdej z nich umieść odpowiedni formularz, tylko jeden. Każda z podstron powinna korzystać z layoutu.
- Na stronie `index.html` dodaj komunikat powitalny, usuń za to wszystkie formularze.

Obsługa błędów w formularzach

- Nasza aplikacja nie jest w tej chwili zabezpieczona przed podaniem błędnych danych. Spróbuj zamiast liczby wpisać w formularzu `krowa` - co się stanie?
  - Spring spróbował dopasować dane z żądania do obiektu klasy `CalculatorForm`;
  - nie udało się to z powodu błędnego typu danych - Spring rzucił więc wyjątkiem i przekierował na standardową stronę błędu.
- Pora obsłużyć błędy walidacji - jeśli użytkownik wprowadzi błędne dane, chcemy powrócić do formularza, podświetlić błędnie wypełnione pola na czerwono i wyświetlić komunikat o błędzie.
- Aby samodzielnie obsłużyć błędy walidacji, musimy wstrzyknąć do metody w kontrolerze obiekt `BindingResult`:

```
@PostMapping("/add")
public String addNumbers(Model model, CalculatorForm form, BindingResult binding) {
    ...
}
```

- Zachowanie kontrolera zasadniczo się zmieni - Spring spróbuje automatycznie wypełnić obiekt formularza (czyli `CalculatorForm form`), ale jeśli mu się to nie uda, nie rzuci wyjątkiem. Błędne pola w `CalculatorForm` pozostaną niewypełnione, a obiekt `BindingResult binding` będzie zawierał informacje o błędach.
- Sprawdźmy zatem, czy wprowadzono poprawne dane. Jeśli nie - wyświetlimy jeszcze raz formularz. Jeśli tak - wykonamy resztę kodu akcji, jak do tej pory.

```
@PostMapping("/add")
public String addNumbers(Model model, CalculatorForm form, BindingResult binding) {
    if(binding.hasErrors()) {
        return "add"; // powrót do formularza
    }

    // reszta kodu akcji add
    // ...
}
```

- Sprawdź działanie aplikacji. Co się teraz stanie, jeśli wpiszemy błędne dane?
- Musimy jeszcze zmodyfikować widok, tak by wyświetlał informacje o błędnych danych. Przejdź do pliku `add.html`.
- Zacznijmy od powiązania formularza z obiektem `CalculatofForm` przekazanym z kontrolera:
  - do elementu `form` dodaj atrybut `th:object`,
  - z pól usuń atrybuty `name` i `value`, zamiast nich dodaj `th:field`.

```
<form action="/add" method="post" th:object="${calculatorForm}">
    x = <input type="text" th:field="*{x}" />
    y = <input type="text" th:field="*{y}" />
    <input type="submit" value="Dodaj" />
</form>
```

- Teraz Thymeleaf będzie świadom tego, z jakiego obiektu pochodzą dane formularza. Automatycznie też może skorzystać z informacji o błędach w obiekcie `BindingResult`. Aby powiązanie formularza z `CalculatofForm` działało prawidłowo, musimy jeszcze dodać parametr `CalculatofForm form` w kontrolerze, do metody wyświetlającej formularz:

```
@GetMapping("/add")
public String addForm(CalculatorForm form) {
    return "add";
}
```

- Dodajmy teraz podświetlenie na czerwono błędów - do błędnych pól dodamy klasę CSS `error`. Atrybut `th:errorClass` oznacza, że Thymeleaf poszuka w `BindingResult` informacji o błędach. Jeśli znajdzie błąd w danym polu, doda do niego wskazaną klasę CSS.

```
<form action="/add" method="post" th:object="${calculatorForm}">
    x = <input type="text" th:field="*{x}" th:errorClass="error" />
    y = <input type="text" th:field="*{y}" th:errorClass="error"/>
    <input type="submit" value="Dodaj" />
</form>
```

- Musimy też dodać definicję klasy `error` do CSS. Możemy to zrobić w pliku szablonu `layout.html` - w ten sposób CSS będzie dostępny dla wszystkich podstron. W pliku `layout.html`, w sekcji `<head>`, dodaj kod:

```
<style>
    input.error { background-color: pink }
</style>
```

- Sprawdź działanie aplikacji.
- Na razie tylko podświetlamy błędne pola - ostatni krok to wyświetlenie komunikatów o błędach.
- Komunikaty o błędzie również mogą zostać wyciągnięte automatycznie z `BindingResult`. Dodajmy do formularza dwa elementy `span`:
  - wyświetlane warunkowo - tylko wtedy, jeżeli dane pole zawiera błędy (atrybut `th:if`);
  - wyświetlające komunikaty o błedach dla danego pola (atrybut `th:errors`).

```
<form action="/add" method="post" th:object="${calculatorForm}">
    x = <input type="text" th:field="*{x}" th:errorClass="error" />
    <span th:if="${#fields.hasErrors('x')}" th:errors="*{x}">Błędne dane</span>

    y = <input type="text" th:field="*{y}" th:errorClass="error"/>
    <span th:if="${#fields.hasErrors('y')}" th:errors="*{y}">Błędne dane</span>

    <input type="submit" value="Dodaj" />
</form>
```

- Sprawdź działanie aplikacji. Jak widać standardowy komunikat o błędzie nie jest zbyt przyjazny. Własne komunikaty możemy umieścić w pliku konfiguracyjnym:
  - utwórz plik o nazwie `messages.properties` (w `src/main/resources`, obok pliku `application.properties`);
  - umieść w tym pliku własne komunikaty o błędach:

```
typeMismatch.java.lang.Integer=Wymagana liczba
```

Zadanie 4

- Uporządkuj szablon `add.html` - umieść pola w osobnych wierszach, błędy walidacji obok nich.
- W taki sam spobó obsłuż błędy na pozostałych stronach - `multiply` i `calculate`.