Spring MVC formularze

v3.1

Obsługa formularzy

Jak obsłużyć formularz

Dane z formularzy możemy obsłużyć pobierając je za pomocą adnotacji **@RequestParam** lub bezpośrednio z obiektu **HttpServletRequest**.

Wygodniejszym sposobem jest skorzystanie z udostępnionej przez Springa biblioteki znaczników form oraz bindowania formularzy do obiektów.

Przykładowe POJO

Dodajemy proste POJO Student

```
public class Student {
    private String firstName;
    private String lastName;
    public Student(){}
    public Student(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    @Override
    public String toString() {
        return "Student [firstName=" + firstName + ",
                    lastName=" + lastName + "]";
    }
}
```

Definiujemy prostą klasę, której będziemy używać omawiając prace z formularzami.

Obsługa za pomocą parametrów

Definiujemy formularz html:

```
<form method="post">
    First name: <input type="text" name="firstName"><br>
    Last name: <input type="text" name="lastName"><br>
    <input type="submit" value="Submit">
</form>
```

Dodajemy akcję jego wyświetlania:

```
@RequestMapping(value = "/simple", method = RequestMethod.GET)
public String simple() {
    return "form/registerSimple";
}
```

Obsługa za pomocą parametrów

Dodajemy akcję jego przetwarzania:

```
@RequestMapping(value = "/simple", method = RequestMethod.POST)
public String processSimple(@RequestParam String firstName,
                            @RequestParam String lastName, Model model) {
    Student student = new Student(firstName, lastName);
    model.addAttribute("student", student);
    return "form/success";
}
```

Na podstawie parametrów pobranych przy użyciu adnotacji **@RequestParam** tworzymy obiekt klasy **Student**, który następnie przekazujemy do widoku.

Możemy również wykonać dowolne inne operacje na naszych parametrach, niekoniecznie trzeba łączyć je z obiektem.

Bindowanie danych

Mapowaniem(**bindowaniem**) danych określamy zachowanie, gdy wartości z pól formularza są automatycznie ustawiane w konkretnych polach obiektu.

Np. dla naszej klasy **Student** wartość atrybutu **firstName** zostanie automatycznie wstawiona wartość z

```
<input name="firstName">
```

W tym celu wykorzystujemy bibliotekę znaczników:

```
<%@ taglib prefix="form"
    uri="http://www.springframework.org/tags/form" %>
```

Bibliotekę tą załączamy w naszych plikach jsp analogicznie jak to miało miejsce w przypadku **jstl**.

Taglib form

Omawiany wcześniej formularz z wykorzystaniem biblioteki znaczników będzie wyglądał następująco:

```
<form:form method="post"
           modelAttribute="student">
    <form:input path="firstName" />
    <form:input path="lastName" />
    <input type="submit" value="Save">
</form:form>
```

Atrybut:

```
modelAttribute="student"
```

określa, że dane z formularza są wiązane z modelem **Student**. Spotkać można również przykłady zawierające atrybut commandName - od wersji 4.3 nie jest zalecany.

Atrybut:

```
path="firstName"
```

określa powiązanie z właściwością modelu.

Wyświetlanie formularza

Do widoku, który wyświetla formularz przekazujemy atrybut o nazwie **student** - jest to nowy obiekt klasy **Student**.

```
@RequestMapping(value = "/register",method = RequestMethod.GET)
public String showRegistrationForm(Model model) {
    model.addAttribute("student", new Student());
    return "form/registerForm"; }
```

Możemy również przekazać uprzednio wypełniony danymi obiekt np.

```
@RequestMapping(value = "/register", method = RequestMethod.GET)
public String showRegistrationForm(Model model) {
    Student student = new Student("Jan","Kowalski");
    model.addAttribute("student", student);
    return "form/registerForm"; }
```

W taki sposób przekazujemy dane pobrane z bazy w celu ich edycji.

Możemy również ustawić wartości domyślne.

Obsługa formularza

W akcji możemy już korzystać z obiektu student.

```
@RequestMapping(value = "/register", method = RequestMethod.POST)
public String processForm(@ModelAttribute Student student) {
    System.out.println(student.getFirstName());
    return "form/success";
}
```

Obiekt klasy **Student** będzie wypełniony danymi z formularza - w tym celu oznaczamy obiekt adnotacją **@ModelAttribute**.

Taki obiekt możemy już zapisać do bazy danych przy pomocy poznanego wcześniej **Entity Managera** oraz metody **persist**.

Definiujemy metodę dostępną pod tym samym adresem, ale tylko przy pomocy metody POST.

Warto zapoznać się z dokumentacją: https://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#mvc-ann-modelattrib-method-args

@ModelAttribute

Jeżeli adnotację **@ModelAttribute** zastosujemy do metody kontrolera, zwracane przez nią wartości będą dostępne dla wszystkich widoków tego kontrolera.

```
@ModelAttribute("languages")
public List<String> checkOptions() {
    String a[] = new String[] {"java", "php", "ruby", "python"};
    return Arrays.asList(a);
}
```

Zostanie ona wywołana przy okazji uruchamiania każdej akcji kontrolera.

Dane te możemy np. przy użyciu **JSTL** wyświetlić za pomocą pętli:

```
<c:forEach items="${languages}" var="lang">
        ${lang}<br>
</c:forEach>
```

Lub skorzystać z tagu **<form:select>** - opiszemy go w następnej części.

Tagi tekstowe

W celu powiązania danych z formularza z obiektem, stosujemy specjalne tagi.

Dla wartości typu **String** możemy wykorzystać następujące tagi:

(**<form:input>**, **<form:password>** **<form:textarea>**), np.:

```
<form:input path="firstName"/>
<form:password path="password"/>
<form:textarea path="notes" rows="3"
               cols="20"/>
```

Zostaną one zmapowane na atrybuty naszej klasy:

```
public class Student {
    private String firstName;
    private String password;
    private String notes;
    //pozostałe elementy: gettery,
    //settery, konstruktory
}
```

Wiązanie odbywa się po nazwie określonej w atrybucie tagu - **path** oraz nazwie atrybutu z klasy.

form:checkbox

Przy pomocy tagu checkbox możemy bindować dane do pola typu **boolean** naszego obiektu.

```
<form:checkbox path="receiveMessages"/>
```

```
public class Student {
    private String firstName;
    private String password;
    private String notes;
    private boolean receiveMessages;
}
```

tag checkbox

Używając wielu tagów **<form:checkbox>** możemy bindować dane do tablicy lub kolekcji.

```
Php: <form:checkbox path="skills"
                    value="php"/>
Java: <form:checkbox path="skills"
                     value="java"/>
Ruby: <form:checkbox path="skills"
                     value="ruby"/>
Python: <form:checkbox path="skills"
                       value="python"/>
```

```
public class Student {
    private String firstName;
    private String password;
    private String notes;
    private boolean receiveMessages;
    private String[] skills;
}
```

W ten sposób możemy bindować dane do wszystkich kolekcji/tablic, które są określone w encji.

tag checkbox

Częstym przypadkiem użycia jest generowanie tagów checkbox na podstawie danych otrzymanych z kontrolera, służy do tego konstrukcja:

```
<form:checkboxes items="${skills}" path="skills" />
```

Dla atrybutu **items** podstawiamy przekazaną z kontrolera wartość.

path to nazwa atrybutu z obiektu.

tag checkbox

Za pomocą metody kontrolera opatrzonej adnotacją **@ModelAttribute** przekazujemy dodatkowe dane przy wywołaniu każdej jego akcji.

```
@ModelAttribute("skills")
public Collection<String> skills() {
    List<String> skills = new ArrayList<String>();
    skills.add("Java");
    skills.add("Php");
    skills.add("python");
    skills.add("ruby");
    return skills;
}
```

tag checkbox

Tag ten zawiera również dodatkowe atrybuty przydatne podczas wyświetlania określonych właściwości w przypadku gdy elementy są obiektami naszych własnych typów:

```
<form:checkboxes  path="skills" items="${skills}"
                  itemLabel="name" itemValue="id" />
```

- **itemLabel** - atrybut naszej klasy, który wyświetlamy dla użytkownika
- **itemValue** - atrybut naszej klasy, który przekazujemy jako wartość

tag checkbox

Tworzymy klasę **Skill**:

```
public class Skill {
    private Integer id;
    private String name;
    //pozostałe elementy: gettery, settery, konstruktory
}
```

Przekazujemy listę jej obiektów:

```
@ModelAttribute("skills")
public Collection<Skill> skills() {
    List<Skill> skills = new ArrayList<Skill>();
    skills.add(new Skill(1, "Java"));
    skills.add(new Skill(2, "PHP"));
    skills.add(new Skill(3, "Ruby"));
    return skills;
}
```

tag radiobutton

Do generowania elementu html typu **radio** służy tag:

```
<form:radiobutton path="valueToBind" value="bindedValue"/>
```

np.

```
Male: <form:radiobutton path="sex" value="M"/>
Female: <form:radiobutton path="sex" value="F"/>
```

Tag ten analogicznie posiada wariant, który służy do obsługi wartości otrzymanych z kontrolera:

```
<form:radiobuttons items="${skills}" path="mainSkill" />
```

tag select

Kolejnym tagiem jest select

```
<form:select path="country" items="${countryItems}" />
```

Możemy określić by była to lista z możliwością wielokrotnego wyboru:

```
<form:select path="fruit" items="${fruit}" multiple="true"/>
```

Aby dodać dodatkową wartość, możemy wykorzystać dodatkowo tag:

```
<form:options/>
```

np:

```
<form:select path="book">
    <form:option value="-" label="--Please Select--"/>
    <form:options items="${books}"/>
</form:select>
```

tag select

Kolejnym tagiem jest select

```
<form:select path="country" items="${countryItems}" />
```

Możemy określić by była to lista z możliwością wielokrotnego wyboru:

```
<form:select path="fruit" items="${fruit}" multiple="true"/>
```

Aby dodać dodatkową wartość, możemy wykorzystać dodatkowo tag:

```
<form:options/>
```

np:

```
<form:select path="book">
    <form:option value="-" label="--Please Select--"/>
    <form:options items="${books}"/>
</form:select>
```

Dodaje dodatkową opcję.

tag select

Kolejnym tagiem jest select

```
<form:select path="country" items="${countryItems}" />
```

Możemy określić by była to lista z możliwością wielokrotnego wyboru:

```
<form:select path="fruit" items="${fruit}" multiple="true"/>
```

Aby dodać dodatkową wartość, możemy wykorzystać dodatkowo tag:

```
<form:options/>
```

np:

```
<form:select path="book">
    <form:option value="-" label="--Please Select--"/>
    <form:options items="${books}"/>
</form:select>
```

Tworzy opcje na podstawie wartości przekazanej z kontrolera.

tag hidden

Dostępny jest również tag służący do przekazywania wartości ukrytych dla użytkownika

```
<form:hidden path="id" value="12345" />
```

Tag ten będzie potrzebny np. w formularzu edycji wartości z bazy danych do przechowywania identyfikatora oraz w przypadku gdy chcemy przekazać parametr w sposób nie widoczny dla użytkownika.

Wykonaj zadania z działu

Formularze

Praca z encjami

Dane z formularzy bardzo często zamierzamy utrwalić do ponownego ich wykorzystania w przyszłości.

W tym celu wystarczy zbindować formularz z odpowiednią **Encją**.

Dodając odpowiednie adnotacje wykorzystamy w tym celu nasze przykładowe **POJO**.

```
@Entity
public class Student {
    private String firstName;
    private String lastName;
    public Student(){}
}
```

Praca z encjami

Aby utrwalić dane z naszego formularza wywołujemy metodę **save**, której argumentem jest obiekt, który chcemy zapisać.

Pamiętajmy aby w dowolny, znany nam sposób, wstrzyknąć odpowiednią klasę (serwis/repozytorium/dao) zajmującą się zapisem do bazy danych.

```
@Autowired
private  StudentDao studentDao;
@RequestMapping(value = "/register", method = RequestMethod.POST)
public String processForm(@ModelAttribute Student student) {
    studentDao.saveStudent(student);
    return "form/success";
}
```

Bindowanie zaawansowane

W przypadku danych typów **int**, **boolean**, **String** zostaną one automatycznie zbindowane do odpowiednich właściwości naszego obiektu.

W przypadku gdy obiekt posiada referencje do obiektów naszych własnych klas nie zostaną one automatycznie zbindowane ponieważ **Spring** nie wie jak przekształcić przesłaną w formie tekstowej wartość na odpowiedni obiekt.

Bindowanie zaawansowane

Przykład omawianej definicji encji:

```
@Entity
public class Student {
    private String firstName;
    private String lastName;
    @ManyToOne
    @JoinColumn(name = "student_group_id")
    private StudentGroup studentGroup;
    public StudentGroup getStudentGroup() {
        return studentGroup;
    }
    public void setStudentGroup(StudentGroup studentGroup) {
        this.studentGroup = studentGroup;
    }
}
```

Bindowanie zaawansowane

Korzystając z tagu:

```
<form:select path="studentGroup" items="${groups}"
             itemValue="id" itemLabel="name"/>
```

Możemy wyświetlić listę grup do wyboru dla odpowiedniego studenta.

Aby dostępne grupy wyświetliły się prawidłowo należy je oczywiście odpowiednio do naszego widoku przekazać.

W celu przekazania dodatkowych danych wykorzystamy metodę kontrolera z omawianą adnotacją **@ModelAttribute**:

```
@ModelAttribute("groups")
public Collection<StudentGroup> populateGroups() {
//odpowiednia metoda zwracająca kolekcje
    return this.groups.findGroups();
}
```

Bindowanie zaawansowane

Próba zapisu obiektu student spowoduje wystąpienie błędu.

Oznacza to, że **Spring** nie wie jak przekształcić przekazaną wartość tekstową na obiekt odpowiedniego typu.

Field error in object 'student' on field 'studentGroup': rejected value [11]; codes ... convert property value of type 'java.lang.String' to required type 'pl.coderslab.model.StudentGroup' for property 'studentGroup'; nested exception is java.lang.IllegalStateException: **Cannot convert value** of type **'java.lang.String'** to required type **'pl.coderslab.model.StudentGroup'** for property **'studentGroup'** : no matching editors or conversion strategy found]

Bindowanie zaawansowane

Prostym sposobem aby rozwiązać ten problem jest modyfikacja atrybutu **path** w następujący sposób:

```
<form:select itemValue="id" itemLabel="name"
             path="studentGroup.id" items="${groups}"/>
```

Dodając do atrybutu **path** - który reprezentuje referencję do obiektu sufiks **.id** :

```
studentGroup.id
```

Przy takim rozwiązaniu Spring poradzi sobie prawidłowo z bindowaniem.

Użycie konwertera

Spring udostępnia interfejs dzięki, któremu możemy stworzyć klasy definiujące sposób konwersji odpowiednich typów: https://docs.spring.io/spring/docs/current/spring-framework-reference/html/validation.html#core-convert

Dzięki temu możemy określić sposób w jaki pobrany przez nas parametr typu **String** ma zostać zamieniony na obiekt odpowiedniego typu.

Przykład konwertera

```
public class StudentGroupConverter implements Converter<String, StudentGroup> {
    @Autowired
    private  StudentGroupRepository groups;
    @Override
    public StudentGroup convert(String source) {
        StudentGroup group = groups.findById(Integer.parseInt(source));
        return  group;
    }
}
```

Konwerter zawiera jedną metodę, która przy pomocy przekazanego repozytorium lub Dao pobiera obiekt danego typu.

Rejestracja konwertera

Utworzony konwerter należy dodać do rejestru, w pliku konfiguracji:

```
@Override
public void addFormatters(FormatterRegistry registry) {
    registry.addConverter(getStudentGroupConverter());
}
@Bean
public StudentGroupConverter getStudentGroupConverter() {
    return new StudentGroupConverter();
}
```

Warto również zapoznać się z formaterami: https://docs.spring.io/spring/docs/current/spring-framework-reference/html/validation.html#format-Formatter-SPI

Generyczny konwerter

Naturalnym krokiem rozwoju będzie utworzenie bazowych Encji oraz generycznego konwertera.

Więcej na ten temat:

https://docs.spring.io/spring/docs/current/spring-framework-reference/html/validation.html#core-convert-ConverterFactory-SPI
https://digitaljoel.nerd-herders.com/2011/06/15/spring-converterfactory-implementation/

Warto również zapoznać się z artykułem:

http://springinpractice.com/2012/01/07/making-formselect-work-nicely-using-spring-3-formatters/

Wykonaj zadania z działu

Formularze - Encje