

# Spring MVC Form Handling

**“Code with Passion!”**



# Topics

- 2-phase form submission handling
- Command/form objects
- @ModelAttribute
- Validation
- Data binding
- Form tags
- Redirection (in form submission handling)

# 2-Phase Form Submission Handling

# Form Display & Submission Handling

- It is a two-phase process
- Phase 1: Handle initial form display request (GET request)
  - > (1.1) Creation and initialization of Command object
  - > (1.2) Display the form page
- Phase 2: Handle the form submission (POST request)
  - > (2.1) Data binding and type conversion
  - > (2.2) Validation
  - > (2.3) Business logic handling with input data
  - > (2.4) Redirect

# Phase 1: Handle initial form display request

```
@Controller
@RequestMapping(value="/account")
public class AccountController {
```

```
// Phase 1: Handle initial form display request
```

```
@RequestMapping(method=RequestMethod.GET)
public String getcreateForm(Model model) {
```

```
// (1.1) Create command object "account"
model.addAttribute("account", new Account());
```

```
// (1.2) Return a logical view "account/createForm", which results in
// displaying "account/createForm.jsp"
return "account/createForm";
```

```
}
```

"account" is now accessible in a form page

# Phase 1: ./account/createForm.jsp Displayed

```
<form:form modelAttribute="account" action="account" method="post">
  <fieldset>
    <legend>Account Fields</legend>
    <p>
      <form:label      for="name" path="name"
                        cssErrorClass="error">Name</form:label><br/>
      <form:input path="name" /> <form:errors path="name" />
    </p>
    <p>
      <form:label for="balance" path="balance"
                  cssErrorClass="error">Balance</form:label><br/>
      <form:input path="balance" /> <form:errors path="balance" />
    </p>
    <p>
      <form:label for="equityAllocation" path="equityAllocation"
                  cssErrorClass="error">Equity Allocation</form:label><br/>
      <form:input path="equityAllocation" /> <form:errors path="equityAllocation" />
    </p>
    <p>
      <form:label for="renewalDate" path="renewalDate"
                  cssErrorClass="error">Renewal Date</form:label><br/>
      <form:input path="renewalDate" /> <form:errors path="renewalDate" />
    </p>
    <p>
      <input type="submit" />
    </p>
  </fieldset>
</form:form>
```

## Phase 2: Form is submitted

```
<form:form modelAttribute="account" action="account" method="post">
  <fieldset>
    <legend>Account Fields</legend>
    <p>
      <form:label      for="name" path="name"
                        cssErrorClass="error">Name</form:label><br/>
      <form:input path="name" /> <form:errors path="name" />
    </p>
    <p>
      <form:label for="balance" path="balance"
                  cssErrorClass="error">Balance</form:label><br/>
      <form:input path="balance" /> <form:errors path="balance" />
    </p>
    <p>
      <form:label for="equityAllocation" path="equityAllocation"
                  cssErrorClass="error">Equity Allocation</form:label><br/>
      <form:input path="equityAllocation" /> <form:errors path="equityAllocation" />
    </p>
    <p>
      <form:label for="renewalDate" path="renewalDate"
                  cssErrorClass="error">Renewal Date</form:label><br/>
      <form:input path="renewalDate" /> <form:errors path="renewalDate" />
    </p>
    <p>
      <input type="submit" />
    </p>
  </fieldset>
</form:form>
```

**sends the post request  
to /account URL**

# Phase 2: Handle form submission

```
@Controller
@RequestMapping(value="/account")
public class AccountController {
```

...

```
// Phase 2: Handle form submission
@RequestMapping(method=RequestMethod.POST)
// (2.1) Data Binding and type conversion
// (2.2) Validation are performed by Spring framework before this method gets called
public String create(@Valid Account account, BindingResult result) {
    if (result.hasErrors()) {
        return "account/createForm";
    }
    // (2.3) Some business logic handling
    this.accounts.put(account.assignId(), account);
    // (2.4) Redirect
    return "redirect:/account/" + account.getId();
}
```



# Lab:

Exercise 1: Simple form  
4946\_spring4\_mvc\_form.zip



# Command/Form Objects

# Model Attribute Gets Created In Initial Form Display Request Handling

```
// Handle initial form request
```

```
@RequestMapping(method=RequestMethod.GET)
```

```
public String getcreateForm(Model model) {
```

```
    // "Account" object will be accessible as form object in the "account/createForm.jsp"
```

```
    model.addAttribute("account", new Account());
```

```
    return "account/createForm";
```

```
}
```

```
// Handle form submission
```

```
@RequestMapping(method=RequestMethod.POST)
```

```
public String create(Account account, BindingResult result) {
```

```
    // If there is an error either in validation or data binding,
```

```
    // display the "account/createForm.jsp" page again.
```

```
    if (result.hasErrors()) {
```

```
        return "account/createForm";
```

```
    }
```

```
    // If there is no error, add the newly created account into
```

```
    // the "accounts" table and then redirect to the /account/{id},
```

```
    // which will be handled in the "getView(..)" method below.
```

```
    this.accounts.put(account.assignId(), account);
```

```
    return "redirect:/account/" + account.getId();
```

```
}
```

# Model Attribute is used when Form page gets displayed in Phase 1

```
<form:form modelAttribute="account" action="account" method="post">
  <fieldset>
    <legend>Account Fields</legend>
    <p>
      <form:label for="name" path="name" cssErrorClass="error">Name</form:label><br/>
      <form:input path="name" /> <form:errors path="name" />
    </p>
    <p>
      <form:label for="balance" path="balance" cssErrorClass="error">Balance</form:label><br/>
      <form:input path="balance" /> <form:errors path="balance" />
    </p>
    <p>
      <form:label for="equityAllocation" path="equityAllocation" cssErrorClass="error">Equity Allocation</form:label><br/>
      <form:input path="equityAllocation" /> <form:errors path="equityAllocation" />
    </p>
    <p>
      <form:label for="renewalDate" path="renewalDate" cssErrorClass="error">Renewal Date</form:label><br/>
      <form:input path="renewalDate" /> <form:errors path="renewalDate" />
    </p>
    <p>
      <input type="submit" />
    </p>
  </fieldset>
</form:form>
```

# Form Objects

- Form objects bind HTTP request parameters to bean properties
  - > With type conversion and validation

# Form Object gets used in Form Submission Handling

```
// Handle initial form request
@RequestMapping(method=RequestMethod.GET)
public String getcreateForm(Model model) {

    model.addAttribute("account", new Account());
    return "account/createForm";
}

// Handle form submission
@RequestMapping(method=RequestMethod.POST)
// Form values are bound to the properties of "Account" form object.
public String create(Account account, BindingResult result) {
    // If there is an error either in validation or data binding,
    // display the "account/createForm.jsp" page again.
    if (result.hasErrors()) {
        return "account/createForm";
    }

    // If there is no error, add the newly created account into
    // the "accounts" table and then redirect to the /account/{id},
    // which will be handled in the "getView(..)" method below.
    this.accounts.put(account.assignId(), account);
    return "redirect:/account/" + account.getId();
}
```

**@ModelAttribute**

# @ModelAttribute

- *@ModelAttribute* has two usage scenarios in controllers
- Usage #1 - annotating a method for creating model attribute
  - > *@ModelAttribute* provides reference data for the model
- Usage #2 - annotating a method parameter for accessing existing model attribute (if it exists already) or create a model attribute (if it does not exist)
  - > *@ModelAttribute* maps a model attribute to the specific, annotated method parameter



# @ModelAttribute annotating a method (Usage Model #1) for creating model attr

- Used to create a Model attribute through a method
- *@ModelAttribute* (“<attribute-name>”) annotated methods are executed before the chosen *@RequestMapping* annotated handler method.
  - > They effectively pre-populate the implicit model with specific attributes, often loaded from a database
  - > It could be then accessible in a view
- (Example in the following slide)

# @ModelAttribute annotating a method (Usage Model #1)

// Create “subjectList” model attribute from the return value  
// of the “populateSubjectList()” method. The “subjectList” model  
// attribute is then used in the view as reference data.

```
@ModelAttribute("subjectList")
public List<String> populateSubjectList() {

    //Data referencing for web framework checkboxes
    List<String> subjectList = new ArrayList<String>();
    subjectList.add("Math");
    subjectList.add("Science");
    subjectList.add("Art");
    subjectList.add("Music");

    return subjectList;
}
```

```
@ModelAttribute("insuranceCommand")
public InsuranceCommand getInsuranceCommand(){
    InsuranceCommand insuranceCommand = new InsuranceCommand();
    return insuranceCommand;
}
```

# @ModelAttribute annotating a method (Usage Model #1)

```
<tr>  
  <td>Favorite Subject:</td>  
  <td><form:checkboxes items="${subjectList}"  
    path="subject" /></td>  
  <td><form:errors path="subject" cssClass="error" /></td>  
</tr>
```

Mozilla Firefox

File Edit View History Bookmarks ScrapBook Tools Help

http://localhost:8080/mvc\_form\_formtags/

http://localh...orm\_formtags/

### Spring's form tags example

Name:

Address :

Password :

Confirm Password :

Subscribe to newsletter? : ☐

Favorite Subject: ☒ Math ☐ Science ☐ Art ☐ Music ☐ Sports

Sex : ☒ Male ☐ Female

Choose a number: ☐ Number 1 ☐ Number 2 ☐ Number 3 ☐ Number 4 ☐ Number 5

Country:

Spring Experience: 

- Spring Core
- Spring MVC
- Spring WebFlow
- Spring Batch

Done

## @ModelAttribute annotating a method parameter (Usage Model #2)

- If model attribute already exists, you can access it
  - > Example: once a model attribute is created via Usage model #1, such an attribute can then be accessed through @ModelAttribute annotated handler method parameters in a handler method, potentially with binding and validation applied to it
- Otherwise (if the model attribute does not exist), it gets created and you can access it

# @ModelAttribute annotating a method parameter (Usage Model #2)

```
@RequestMapping(value="{id}", method=RequestMethod.GET)
public String getView(@PathVariable Long id,
    // Access "subjectList" pre-existing model attribute as "subjectList2"
    @ModelAttribute("subjectList") List<String> subjectList2,
    Model model) {

    Account account = this.accounts.get(id);
    if (account == null) {
        throw new ResourceNotFoundException(id);
    }
    model.addAttribute("account", account);
    model.addAttribute("subjectList2", subjectList2);

    //return "account/view";                // Display account detail in form format
    return "account/view2";                // Display account detail in table format
}
```

# Lab:

Exercise 6: @ModelAttribute  
for method parameter  
4946\_spring4\_mvc\_form.zip



# Validation

# Invoking a Custom Validator

- Choices of configuring Validator instance invoked when a *@Valid* method argument is encountered
  - > Choice #1 - Call *binder.setValidator(Validator)* within a *@Controller*'s *@InitBinder* callback
    - > Allows you to configure a Validator instance per *@Controller* class
  - > Choice #2 - Explicitly call *validate()* method of a Validator class
    - > Allows you to call a validator method at the location you want
  - > Choice #3 - Call *setValidator(Validator)* on the global *WebBindingInitializer*
    - > Allows you to configure a Validator instance across all *@Controllers*



# Choice #1 - Per Controller Validation

```
@Controller
public class MyController {

    @InitBinder
    protected void initBinder(WebDataBinder binder) {
        binder.setValidator(new FooValidator());
    }

}
```

```
public class FooValidator implements Validator{

    public void validate(Object target, Errors errors) {
        // Validation logic
        if (validationFailed()){
            errors.rejectValue("fieldName", errorCode);
        }
    }

}
```

## Choice #2 – Call Validator Explicitly

```
public String create(@Valid Account account, BindingResult result) {  
    // Perform validation directly within the method  
    accountValidator2.validate(account, result);  
  
    if (result.hasErrors()) {  
        return "account/createForm";  
    }  
    this.accounts.put(account.assignId(), account);  
    return "redirect:/account/" + account.getId();  
}
```

## Choice #3 - Global Validation

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:mvc="http://www.springframework.org/schema/mvc"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/mvc
    http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">

  <mvc:annotation-driven validator="globalValidator"/>

</beans>
```

# Spring 3 Enhancements in Validation

- Beginning with Spring 3, Spring MVC has the ability to automatically validate *@Controller* inputs.
  - > In previous versions, it was up to the developer to manually invoke validation logic
- To trigger validation of a *@Controller* input, simply annotate the input argument with *@Valid* (from JSR 303)

```
@Controller
public class MyController {

    @RequestMapping(method=RequestMethod.POST)
    public void processFoo(@Valid Account account,
                          BindingResult result) { /* ... */ }
```

# Domain Class with Validation Annotation

```
public class Account {  
    private Long id;  
  
    @NotNull  
    @Size(min=1, max=25)  
    private String name;  
  
    @NotNull  
    @NumberFormat(style=Style.CURRENCY)  
    private BigDecimal balance = new BigDecimal("1000");  
  
    @NotNull  
    @NumberFormat(style=Style.PERCENT)  
    private BigDecimal equityAllocation = new BigDecimal(".60");  
  
    @DateTimeFormat(style="S-")  
    @Future  
    private Date renewalDate = new Date(new Date().getTime() + 31536000000L);  
  
    public Long getId() {  
        return id;  
    }  
}
```

# Controller Class

```
@Controller
@RequestMapping(value="/account")
public class AccountController {

    private Map<Long, Account> accounts =
        new ConcurrentHashMap<Long, Account>();

    // Handle initial form request
    @RequestMapping(method=RequestMethod.GET)
    public String getCreateForm(Model model) {
        model.addAttribute(new Account());
        return "account/createForm";
    }

    // Handle form request
    @RequestMapping(method=RequestMethod.POST)
    public String create(@Valid Account account, BindingResult result) {
        if (result.hasErrors()) {
            return "account/createForm";
        }
        this.accounts.put(account.assignId(), account);
        return "redirect:/account/" + account.getId();
    }
}
```

# Lab:

Exercise 2: Validator 1

Exercise 3: Validator 2

4946\_spring4\_mvc\_form.zip



# Data Binding



# What is Data Binding?

- Data binding binds user input to be dynamically bound to the domain model of an application (or whatever objects you use to process user input)

# BindingResult as Handler Argument

- org.springframework.validation.BindingResult
  - > The *BindingResult* parameter have to follow the model object that is being bound immediately

```
// Handle form submission
@RequestMapping(method=RequestMethod.POST)
public String create(Account account, BindingResult result) {
    // If there is an error either in validation or data binding,
    // display the "account/createForm.jsp" page again.
    if (result.hasErrors()) {
        return "account/createForm";
    }
}
```

...

# Customizing Data Binding

- There are two choices for customizing request parameters with PropertyEditors through WebDataBinder (Data binding)
  - > Choice #1: Annotate the handler method with *@InitBinder* (Simpler and preferred)
  - > Choice #2: Externalize your configuration by providing a custom *WebBindingInitializer*.

# Choice #1: Using `@InitBinder`

- Annotating controller methods with `@InitBinder` allows you to configure web data binding directly within your controller class
  - > `@InitBinder` identifies methods that initialize the `WebDataBinder` that will be used to populate command and form object arguments of annotated handler methods.

# Choice #1: Using @InitBinder

- Methods annotated with @InitBinder receive WebDataBinder as argument, to which you register custom binders

@Controller

```
public class MyFormController {
```

**@InitBinder**

```
public void initBinder(WebDataBinder binder) {
```

```
    SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");  
    dateFormat.setLenient(false);
```

```
    // Register Spring-provided CustomDateEditor binder class
```

```
    // It converts "yyyy-MM-dd" string to Date object
```

```
    binder.registerCustomEditor(Date.class,  
                                new CustomDateEditor(dateFormat, false));
```

```
    // Register custom binder class
```

```
    // It converts integer string entered by a user to PetType object
```

```
    binder.registerCustomEditor(PetType.class, new PetTypeEditor(this.clinic));
```

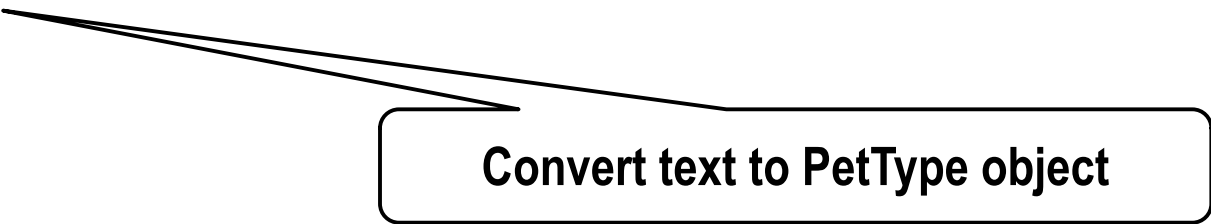
```
}
```

```
// ...
```

```
}
```

# Custom Binder Class

```
public class PetTypeEditor extends PropertyEditorSupport {  
    private final Clinic clinic;  
  
    public PetTypeEditor(Clinic clinic) {  
        this.clinic = clinic;  
    }  
  
    @Override  
    public void setAsText(String text) throws IllegalArgumentException {  
        for (PetType type : this.clinic.getPetTypes()) {  
            if (type.getName().equals(text)) {  
                setValue(type);  
            }  
        }  
    }  
}
```



Convert text to PetType object

## Choice #2: Custom WebBindingInitializer

- Provide a custom implementation of the *WebBindingInitializer* interface
- Provide a custom bean configuration for an AnnotationMethodHandlerAdapter, thus overriding the default configuration
  - > Note: AnnotationMethodHandlerAdapter is deprecated in Spring 3.2 in favor of RequestMappingHandlerAdapter

# Custom WebBindingInitializer

```
<bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter">  
  <property name="cacheSeconds" value="0" />  
  <property name="webBindingInitializer">  
    <bean  
      class="org.springframework.samples.petclinic.web.ClinicBindingInitializer" />  
    </property>  
  </bean>
```



# Custom WebBindingInitializer

```
public class ClinicBindingInitializer implements WebBindingInitializer {  
  
    @Autowired  
    private Clinic clinic;  
  
    public void initBinder(WebDataBinder binder) {  
        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");  
        dateFormat.setLenient(false);  
        binder.registerCustomEditor(Date.class,  
                                     new CustomDateEditor(dateFormat, false));  
        binder.registerCustomEditor(String.class, new StringTrimmerEditor(false));  
        binder.registerCustomEditor(PetType.class, new PetTypeEditor(this.clinic));  
    }  
}
```

# Lab:

Exercise 4: InitBinder  
4946\_spring4\_mvc\_form.zip



# Form Tags

# Using Spring's Form Tag Library

- As of version 2.0, Spring provides a comprehensive set of data binding-aware tags for handling form elements when using JSP and Spring MVC.
- Each tag provides support for the set of attributes of its corresponding HTML tag counterpart, making the tags familiar and intuitive to use.
- The tag-generated HTML is HTML 4.01/XHTML 1.0 compliant.

# Tag Library Integration with Spring MVC

- Unlike other form/input tag libraries, Spring's form tag library is integrated with Spring MVC, giving the tags access to the command object and reference data your controller deals with

# “form” tag (from Spring Form Tag Library)

Let's assume we have a domain object called “command”. It is a JavaBean with properties such as firstName and lastName.

We will use it as the form backing object of our form controller which returns form.jsp.

```
<form:form>
<table>
  <tr>
    <td>First Name:</td>
    <td><form:input path="firstName" /></td>
  </tr>
  <tr>
    <td>Last Name:</td>
    <td><form:input path="lastName" /></td>
  </tr>
  <tr>
    <td colspan="2">
      <input type="submit" value="Save Changes" />
    </td>
  </tr>
</table>
</form:form>
```

# “form” tag

The preceding JSP assumes that the variable name of the form backing object is 'command'.

If you have put the form backing object into the model under another name (definitely a best practice), then you can bind the form to the named variable.

```
<form:form commandName="user">
  <table>
    <tr>
      <td>First Name:</td>
      <td><form:input path="firstName" /></td>
    </tr>
    <tr>
      <td>Last Name:</td>
      <td><form:input path="lastName" /></td>
    </tr>
    <tr>
      <td colspan="2">
        <input type="submit" value="Save Changes" />
      </td>
    </tr>
  </table>
</form:form>
```

# “checkboxes” tag

This tag renders multiple HTML 'input' tags with type 'checkbox'.

```
<form:form>
  <table>
    <tr>
      <td>Interests:</td>
      <td>
        <%-- Property is of an array or of type java.util.Collection --%>
        <form:checkboxes path="preferences.interests" items="${interestList}"/>
      </td>
    </tr>
  </table>
</form:form>
```



# “radiobuttons” tag

This tag renders multiple HTML 'input' tags with type 'radio'.

```
<tr>  
  <td>Sex:</td>  
  <td><form:radiobuttons path="sex" items="${sexOptions}"/></td>  
</tr>
```

# “password” tag

This tag renders an HTML 'input' tag with type 'password' using the bound value.

```
<tr>
  <td>Password:</td>
  <td>
    <form:password path="password" />
  </td>
</tr>
```

# “select” tag

This tag renders an HTML 'select' element. It supports data binding to the selected option as well as the use of nested option and options tags.

```
<tr>  
  <td>Skills:</td>  
  <td><form:select path="skills" items="${skills}"/></td>  
</tr>
```

# Lab:

Exercise 5: Form Tags  
4946\_spring4\_mvc\_form.zip



# **Redirect (in Form Submission Handling)**

# Post Redirect Get (PRG)

- Issue an HTTP redirect back to the client, before the view is rendered.
- Use cases
  - > case #1: To eliminate the possibility of the user submitting the form data multiple times through refreshing
  - > case #2: When one controller has been called with POST'ed data, and the response is actually a delegation to another controller (for example on a successful form submission)
- Return “redirect:<destination-URL>”
  - > The `UrlBasedViewResolver` will recognize this as a special indication that a redirect is needed. The rest of the view name will be treated as the redirect URL.

# Example Redirect

// Inside of a controller

// Handle initial form request

```
@RequestMapping(method=RequestMethod.GET)
public String getcreateForm(Model model) {
    model.addAttribute(new Account());
    return "account/createForm";
}
```

// Handle form request

```
@RequestMapping(method=RequestMethod.POST)
public String create(@Valid Account account, BindingResult result) {
    if (result.hasErrors())
        return "account/createForm";
    }
    this.accounts.put(account.assignId(), account);
    return "redirect:/account/" + account.getId();
}
```

# Lab:

Exercise 7: Redirection and Refreshing a page  
4946\_spring4\_mvc\_form.zip

