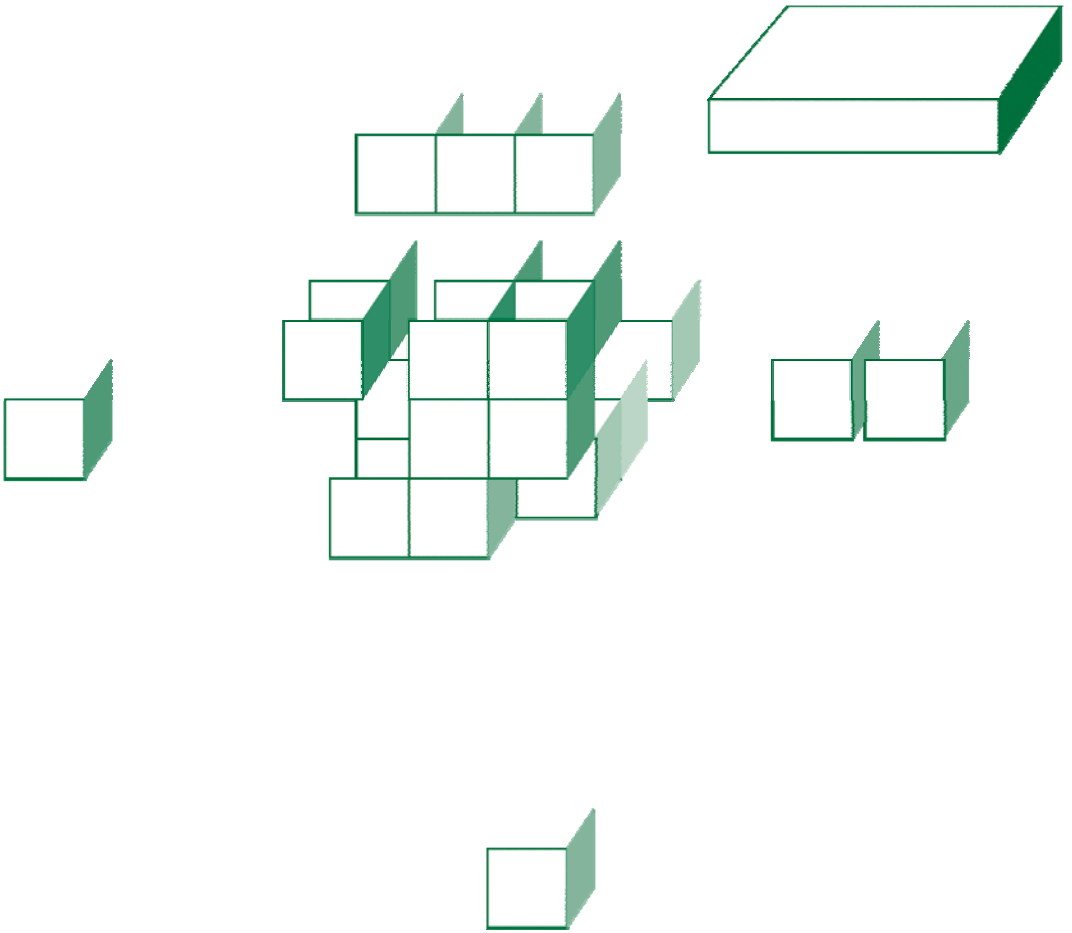


# CHAPTER 5

## BINDING



## OBJECTIVES

*After completing "Binding," you will be able to:*

- Define binders to take control of model binding details, including required fields and choice of data converters.
- Report Web Flow error messages to your JSP views.
- Implement and install custom converters.
- Wrap existing **PropertyEditor** implementations to create Web Flow converters.
- Implement a more externally-configurable conversion service so that desired converters can be plugged using Spring bean definitions and autowiring.

# Binding to a Model

---

- We've seen that it's a snap to connect a view state to a model, and get a match-and-bind heuristic that works well for most cases.
- We can be more explicit, if we like, using a `<binder>` element.

```
<view-state id="myPage" model="myBean" >
  <binder>
    <binding property="prop1" />
    <binding property="prop2" />
    <binding property="prop3" required="true" />
  </binder>
  ...

```

- `<binding>` elements under the binder state exactly what **properties** should be bound from HTTP request parameters; by default Web Flow has been binding any possible match between parameter name and property name.
  - Once a `<binder>` is in force, only properties declared in it will be bound; others will be ignored even if there are matching inputs.
  - We can also state that certain values are **required**. Web Flow will enforce this rule and generate an error message when needed.
- As with binding in general, the rules spelled out in a `<binder>` apply to the transfer of user input into the model.
    - Flow of information out to a new page will not be constrained, for instance, by the list of bound properties; the JSP will “see” the whole model bean as usual.

# The MessageContext

---

- Web Flow manages a **MessageContext** for each request.
- The full data structure goes pretty deep, but basically it's a list of **Message** objects, each with a **severity**, a **source**, and **text**.
  - The source will usually be a model property name; it can be **null**.
- This will gather messages related to several processes:
  - **Binding**, as when a supplied value can't be converted to the type of a target property
  - **Required fields** not found in the HTTP request
  - **Validation** errors.
- The first two of these happen automatically; you must put validation errors in the context manually when implementing a validator.

# Message Bundles

---

- Web Flow takes a nice, simple approach to finding message maps: it looks for a file **messages.properties** in the flow directory.
- This is a standard messages/properties file.
- It can contain keys of a few forms:
  - A key composed of **model**, **property**, and **error key** will be sought by the flow executor for that specific case.
  - A key which is just an **error key** will be used as a fallback if no bean- and property-specific key is found.
- There are two important (key?) error keys:
  - The typical binding failure is a type mismatch, and the message generated in this case will have the error key **typeMismatch**.
  - If a required field is not found, the error key will be **required**.
- So we find message properties files with keys such as:

```
employee.firstName.required  
product.price.typeMismatch
```

```
required  
typeMismatch
```

# Reporting Error Messages

---

- Thus far we've been reporting errors through a crude mechanism, which is just a matter of string-dumping the message context:

```
${flowRequestContext.messageContext}
```

- This is informative, but of course not user-friendly.

- There are a few ways to penetrate the model, but the most intuitive of these is a simple list of messages exposed as the **allMessages** property.

- We can do a couple important things with this list:

- We can report all errors, for instance at the top of a page:

```
<c:forEach var="error" items=
  "${flowRequestContext.messageContext.allMessages}"
>
  <p>${error.text}</p> <!-- or whatever we like -->
</c:forEach>
```

- We can place error messages that relate to a single field somewhere near that form input, so as to direct the user's attention to the problem and let them fix it:

```
<c:forEach var="error" items=
  "${flowRequestContext.messageContext.allMessages}"
>
  <c:if test="${error.source == 'justOneField'}" >
    <span class="errorMessage" />${error.text}</span>
  </c:if>
</c:forEach>
```

- The latter approach is encoded in a custom tag `<cc:error>` for both the LandUse and Wholesale applications.

# Requiring Input Fields

- In **Examples/Wholesale/Step3**, the **buildFeed** view gets more particular about how it binds inputs.

- See **docroot/define/define.xml**:

```
<view-state id="buildFeed" model="feed" >  
  <binder>  
    <binding property="order.product"  
      required="true" />  
    <binding property="order.price"  
      required="true" />  
    <binding property="order.quantity"  
      required="true" />  
  </binder>  
  ...
```

- **Build and test this version of the application.**

- Choose a feed, and try to add an item without supplying input:

The screenshot shows a form with three input fields: 'Item:', 'Price:', and 'Quantity:'. Each field is empty. To the right of each field is a red error message: 'The product name is required.', 'The price is required.', and 'The quantity is required.' respectively. At the bottom of the form are three buttons: 'Add', 'Edit Selected', and 'Remove Selected'.

- While you're at it, try putting in a non-numeric value for one of the number fields:

The screenshot shows the same form as above, but the 'Price:' field now contains the text 'five'. The error message for the Price field has changed to 'Must be a number.' in red. The other error messages remain the same. The 'Add', 'Edit Selected', and 'Remove Selected' buttons are still present at the bottom.

- How are the errors reported to the page?
  - See `docroot/define/buildFeed.jsp`: the table includes a third column whose cells are populated by the `<cc:error>` tag.

```
<tr>
  <td>Item:</td>
  <td><form:input path="order.product" /></td>
  <td><cc:error path="order.product" /></td>
</tr>
...
```

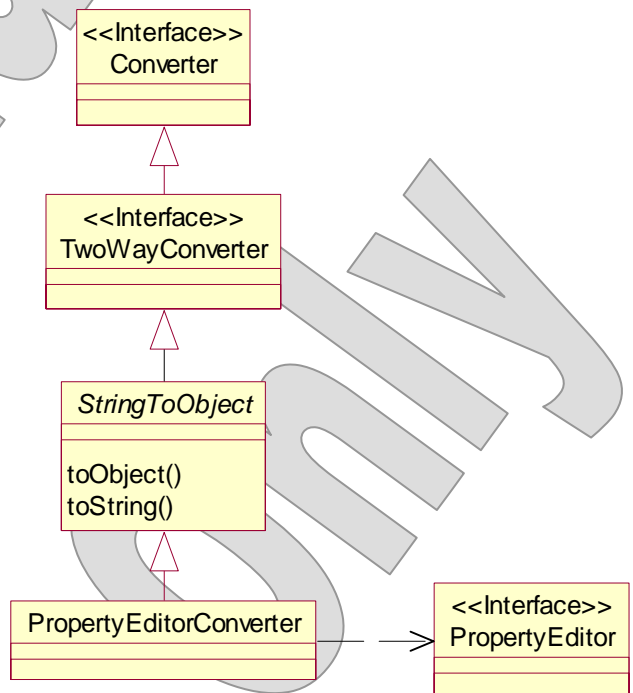
- This tag (found in `docroot/WEB-INF/tags/cc/error.tag`) runs the filtering loop that we saw just before this example – but it tests for a **path** value which is a parameter to the tag, so that it's reusable.
- Finally, see `docroot/define/messages.properties` for the sources of these messages:

```
typeMismatch=Must be a number.
required=The {0} is required.
...
order.product=product name
order.price=price
order.quantity=quantity
```

- Note that the property names are translated before being plugged in to the **required** message.

# Converters

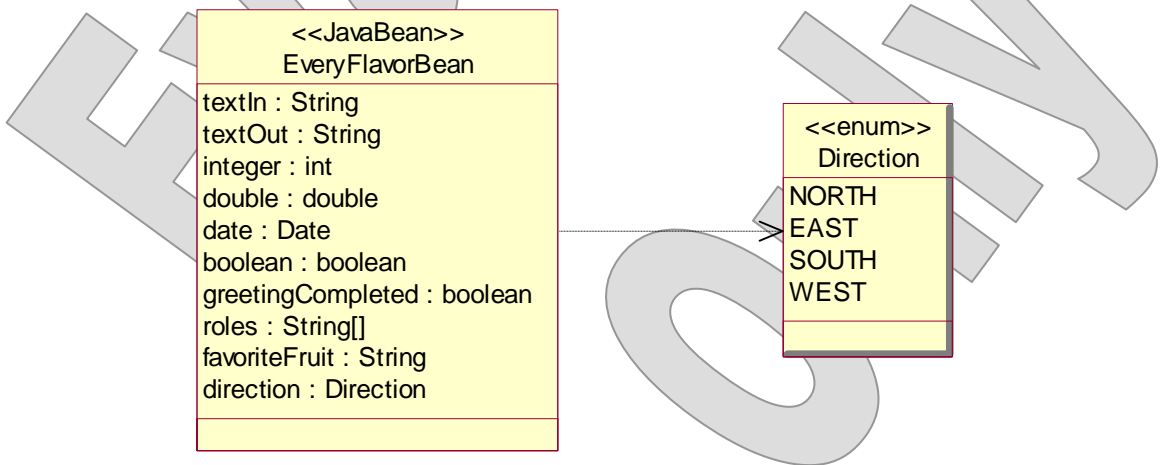
- Web Flow uses a flexible system of **converters** to carry out binding of HTTP request parameters to model properties.
- There are default binders for just about everything you'd expect in the way of standard HTML form inputs: numeric types, booleans, dates, etc.
  - One glaring omission is support for binding HTML checkboxes to boolean properties – more on this later in the chapter.
- You can build your own converters as well – and, check your watches, here's that moment where we start to look at the actual Web Flow API!
- The base interface is **Converter**, but for most purposes it's easiest to do one of two things – either of which is pretty simple:
  - Extend **StringToObject**, which fills in a lot of the interface and just needs to know your target class and how to parse to it – and how to format it, if the object's **toString** method isn't enough.
  - Instantiate a **PropertyEditorConverter**, which adapts the **PropertyEditor** interface that is used elsewhere in Spring.



- In **Examples/Binding/Step1**, a one-page application illustrates common HTML form widgets and their default bindings to JavaBean properties in Web Flow.
- The definition in **docroot/UI/UI.xml** binds a variable of type **EveryFlavorBean** as the model for an HTML form:

```
<view-state id="UI" model="bean" >  
  <var name="bean"  
    class="cc.webflow.EveryFlavorBean" />  
  <transition on="limit" >  
    <evaluate expression="bean.narrow()" />  
  </transition>  
  <transition on="submit" >  
    <evaluate expression="bean.update()" />  
  </transition>  
</view-state>
```

- This class offers several properties of several common types:



- **docroot/UI/UI.jsp** lays out a simple three-column form, with labels, input widgets, and possible error messages.
  - The widgets are bound to bean properties in both directions.
  - Here's the table row for the **textIn** property:

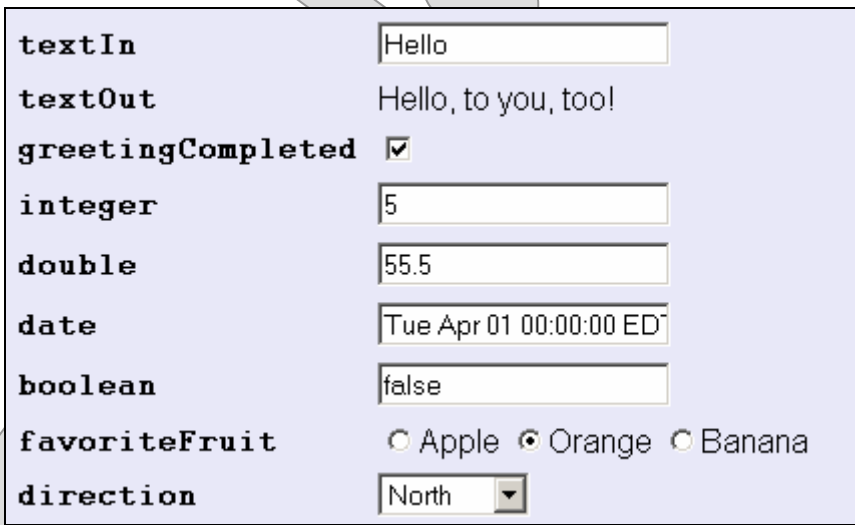
```
<tr>
  <td class="code" >textIn</td>
  <td><input type="text" name="textIn"
    value="{bean.textIn}" /></td>
  <td><cc:error path="textIn" /></td>
</tr>
```

- Build and test the application at the following URL:  
<http://localhost:8080/Binding>

The screenshot shows a web form titled "WebFlow Binding" with the following fields and controls:

- textIn**: A text input field.
- textOut**: A text area containing "Nothing yet ...".
- greetingCompleted**: A checkbox.
- integer**: A text input field containing "0".
- double**: A text input field containing "0.0".
- date**: A text input field.
- boolean**: A text input field containing "false".
- favoriteFruit**: Radio buttons for "Apple", "Orange", and "Banana".
- direction**: A dropdown menu showing "<select>".
- roles**: A text input field and a button labeled "Limit to one role".
- A "Submit" button at the bottom.

- Try filling in some values for the controls:
  - Enter “Hello” in the **textIn** field.
  - Put sensible number values in the **integer** and **double** fields.
  - Enter “2009-04-01” for the **date**; this is the standard parsing format for Web Flow’s default date converter.
  - Pick any **favoriteFruit** and any **direction**.
  - Don’t bother with **roles**; we’ll look at that a bit later.
- Click **Submit ...**



A screenshot of a web form with the following fields and values:

<b>textIn</b>	Hello
<b>textOut</b>	Hello, to you, too!
<b>greetingCompleted</b>	<input checked="" type="checkbox"/>
<b>integer</b>	5
<b>double</b>	55.5
<b>date</b>	Tue Apr 01 00:00:00 ED
<b>boolean</b>	false
<b>favoriteFruit</b>	<input type="radio"/> Apple <input checked="" type="radio"/> Orange <input type="radio"/> Banana
<b>direction</b>	North

- We see that all the values were bound cleanly and were echoed back to the page.
- Also, some logic in the bean – the **update** method invoked by the **submit** transition – has set the **greetingCompleted** property, in response to a matching test on **textIn**.
- But, the date value has been produced in a fuller format – this is just the **toString** method on **java.util.Date**.

- Click **Submit** again, and we run into some issues:

<code>textIn</code>	<input type="text" value="Hello"/>	
<code>textOut</code>	Hello, to you, too!	
<code>greetingCompleted</code>	<input checked="" type="checkbox"/>	typeMismatch on greetingCompleted
<code>integer</code>	<input type="text" value="5"/>	
<code>double</code>	<input type="text" value="55.5"/>	
<code>date</code>	<input type="text" value="Tue Apr 01 00:00:00 EDT"/>	typeMismatch on date
<code>boolean</code>	<input type="text" value="false"/>	

- One of these is simple enough: we have a mismatch between the input and output formats we're using for dates.
  - We entered a value in yyyy-mm-dd format, as expected by the standard date converter.
  - Our page produced the longer format using a simple JSP expression.
  - When we re-submit the page – even with no further changes to the date value – the converter can't parse it.

- The other is more surprising: a checked HTML checkbox will submit a value that Web Flow cannot bind to a boolean property!
  - There is a standard boolean converter.
  - But there’s also a major oversight in the way it’s coded: it insists on string representations “true” and “false”, and these are the only strings it will convert successfully.
  - But the HTTP request parameter will have the value “on”.
- Try the string “on” in the **boolean** field, and you’ll get the same result:

<code>textIn</code>	<input type="text" value="Hello"/>	
<code>textOut</code>	Hello, to you, too!	
<code>greetingCompleted</code>	<input checked="" type="checkbox"/>	<span style="color: red;">typeMismatch on greetingCompleted</span>
<code>integer</code>	<input type="text" value="5"/>	
<code>double</code>	<input type="text" value="55.5"/>	
<code>date</code>	<input type="text" value="Wed Apr 01 00:00:00 EC"/>	<span style="color: red;">typeMismatch on date</span>
<code>boolean</code>	<input type="text" value="false"/>	<span style="color: red;">typeMismatch on boolean</span>

- Notice that the value bounces back to “false”, too – because the binding failed, the prior value of the property still obtains.

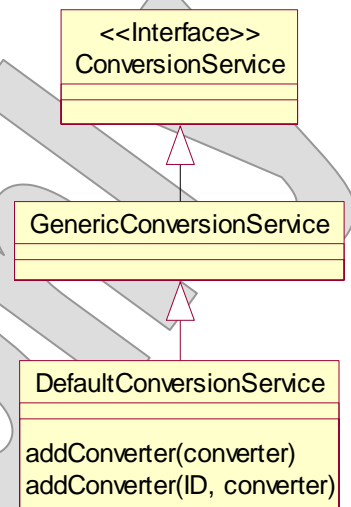
- Clear that checkbox, and remove the date value completely, so that we'll be able to focus on another part of the page.
- Fill in a value for the **roles** field: “one two three” or something like that.
  - This is supposed to be parsed as a whitespace- or comma-separated list, and converted to an array of strings.
- Click **Submit** and see that we have no binding errors.
- But, click **Limit to One Role**, and see that the string is unchanged.
  - The logic here – in the **narrow** method on the bean class – is supposed to cut the list down to include just the first element of whatever was submitted.
  - Nothing doing ... and this gives us a hint that the standard converter is not working quite as we'd hoped.
- Look in the server console to see the actual binding of your input to the string array:

```
Roles:  
one two three
```

  - The code in **setRoles** produces each role value on a separate line.
  - So we see that the whole string is just being set as the one and only element in the string array – not bad for a default converter, but not what we're looking for.
- We'll fix all three of these conversion issues in this chapter's lab.

# Registering a Converter

- Implementing a converter is generally easy to do.
- What's actually a bit more work is putting your custom converter into play for a web flow.
- Web Flow applies converters in one of two ways:
  - Any property type can have a **default converter** registered for it; this is how everything's been happening so far.
  - A **<binding>** can specify a **converter** by unique ID, so that alternatives are available for different properties of the same type – or for the same property as rendered in different situations.
- To plug into either of these features, you need to implement a **ConversionService** and configure it in the flow registry.
- We saw this Spring configuration in the previous chapter.
- The implementation is straightforward: the method **addDefaultConverters** should:
  - Call the superclass implementation!
  - Call **addConverter** to pass in each custom converter as a default for its target type
  - Call **addConverter** with both the converter and an ID, to make that converter available for explicit use in a **<binding>**.



- We'll review the conversion configuration that's already in place for the LandUse application, and add a second custom converter.
  - Do your work in **Demos/Conversion**.
  - The completed demo is in **Examples/LandUse/Step6**.
- The configuration in **docroot/WEB-INF/LandUse-servlet.xml** plugs our **ConversionService** into a services object, which in turn is available to the flow registry.

```
<bean
  id="conversionService"
  class=
    "gov.usda.usfs.landuse.web.ConversionService"
/>

<webflow:flow-builder-services
  id="flowBuilderServices"
  conversion-service="conversionService"
/>

<webflow:flow-registry
  id="flowRegistry"
  flow-builder-services="flowBuilderServices"
>
```

- Our implementation just registers a single converter, as the default for **Calendar** instances.
  - See `src/gov/usda/usfs/landuse/web/ConversionService.java`:

```
public class ConversionService
  extends DefaultConversionService
{
  @Override
  public void addDefaultConverters ()
  {
    super.addDefaultConverters ();
    addConverter (new DateConverter ("M/d/yy"));
  }
}
```

- In case you were wondering: yes, Web Flow already has a **Date** converter in place by default.
  - Our bean just happens to use **Calendar** instead of **Date** – perhaps not the finest design choice that’s ever been made, but typical of the sorts of adaptations that a web application will require.
  - Even if we used **Dates**, there’s still the question of how we might want them to be formatted; so it’s common to have a default and then alternative converters so that `<binding>`s can choose the right input formatting case-by-case.

- The converter itself just extends **StringToObject** and applies formatting using the **java.text** package.
  - See `src/gov/usda/usfs/landuse/web/DateConverter.java`:

```
public class DateConverter
  extends StringToObject
{
  ...
  public DateConverter (String format)
  {
    super (Calendar.class);
    setDateFormat (format);
  }

  public String toString (Object calendar)
  {
    return formatter.format
      (((Calendar) calendar).getTime ());
  }

  public Object toObject (String text, Class cls)
  {
    try
    {
      Calendar value = Calendar.getInstance ();
      value.setTime (formatter.parse (text));
      return value;
    }
    ...
  }
  ...
}
```

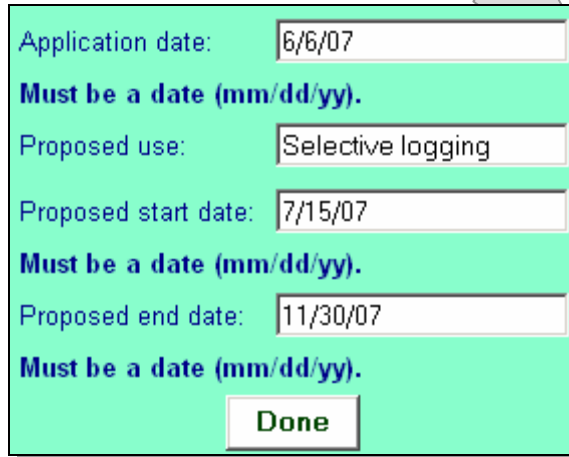
- So that's what's given us our date parsing thus far.
  1. Build and test the application to review the runtime behavior.  
`http://localhost:8080`
  2. If you provide a date that doesn't parse, you get an error message:

The image shows two screenshots from a web application. The top screenshot is a form with three input fields: 'Proposed use:' containing 'Silver mining', 'Proposed start date:' containing '2/1/07', and 'Proposed end date:' containing 'June 17, 2009'. A 'Done' button is visible below the fields, with a mouse cursor pointing at it. The bottom screenshot shows a single input field for 'Proposed end date:' containing '7/30/08', with an error message below it: 'Must be a date (mm/dd/yy)'.

3. But, remember that a missing date is acceptable, thus far; there's nothing in the application that forbids a **null** value.
4. You may want to take a moment to review the messages in `docroot/landUse/messages.properties`; there are no surprises there, but you can see the specific type-mismatch message.  
`typeMismatch=Must be a date (mm/dd/yy).`
5. First, try removing the conversion service from `LandUse-servlet.xml` – just cut one line to the clipboard:

```
<webflow:flow-registry  
  id="flowRegistry"  
  flow-builder-services="flowBuilderServices"  
>
```

6. Rebuild and test. You see the difference immediately – just click **Done** for any proposal, even without changing any dates:



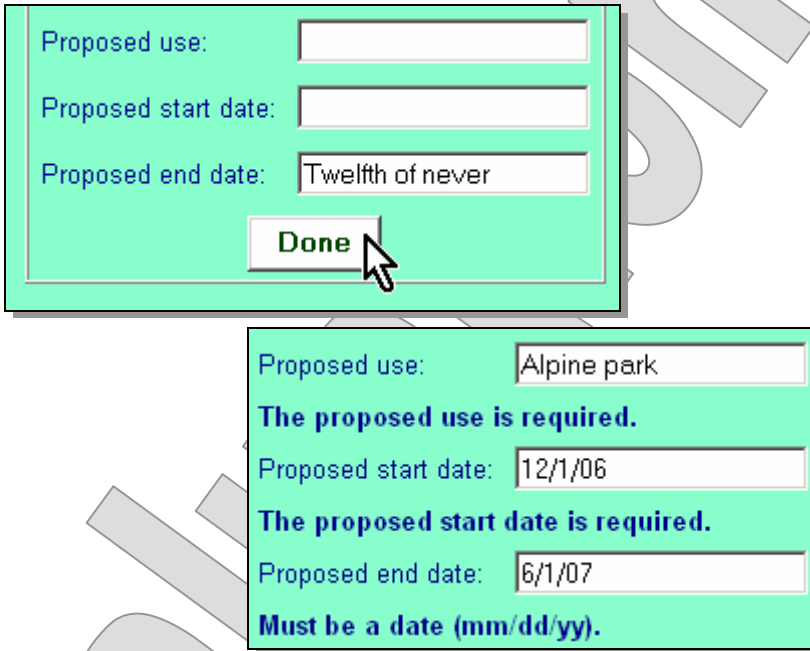
The screenshot shows a web form with the following fields and messages:

- Application date:  (highlighted in green)
- Must be a date (mm/dd/yy).
- Proposed use:
- Proposed start date:  (highlighted in green)
- Must be a date (mm/dd/yy).
- Proposed end date:  (highlighted in green)
- Must be a date (mm/dd/yy).
- A **Done** button is located at the bottom of the form.

- There is no converter for this property type, and so we get a type mismatch error. (A quick criticism: Web Flow could be more precise in its reporting here.)
7. Put that converter service back in place by pasting the deleted line back into **LandUse-servlet.xml**.
8. Open **docroot/landUse/landUse.xml**, and let's add a binder to the **detail** state:

```
<view-state id="detail" model="proposal" >
  <binder>
    <binding property="affectedParcel"
      required="true" />
    <binding property="applicant" required="true" />
    <binding property="applicationDate"
      required="true" />
    <binding property="proposedUse" required="true"/>
    <binding property="useStart" required="true" />
    <binding property="useEnd" required="true" />
  </binder>
```

9. Build and test the results. Now, a missing date is also unacceptable – as is any missing value in this form.



10. Try adding comment; as you saw at the end of the previous chapter, something still isn't right with this view state, and now we're getting a pretty good idea what that might be. You can't add the comment; it just lands you back on the page to try again and again.

11. Let's start by adding some error reporting to the page. Open `docroot/landUse/addComment.jsp`, and find the input form. Add a new cell to each of the first three rows of the table, like this:

```
<tr>
  <td>Your name:</td>
  <td><input type="text" name="contributor"
    value="{comment.contributor}" /></td>
  <td><cc:error path="contributor" /></td>
</tr>
...
```

12. Build and test again, and now we're getting better (if not entirely accurate!) feedback. So we've learned two things:

A screenshot of a web form with a light green background. It contains three input fields: "Your name:" (text box), "Recommendation:" (dropdown menu with "Neutral" selected), and "Comment:" (text area). To the right of the "Recommendation:" field, there is a blue error message: "Must be a date (mm/dd/yy)." The error message is positioned to the right of the dropdown menu.

- There's a problem binding recommendation values, even though they're clearly valid.
- Our error reporting needs some work!

13. Fixing the first problem will mask the second, so let's put a better error message in place right away. Open **messages.properties** and add a specific message for this field:

```
newComment.recommendation.typeMismatch= (one line)  
Must be one of Accept, Reject, or Neutral.  
typeMismatch=Must be a date (mm/dd/yy).
```

14. Build and test again, and see your error message:

A screenshot of the same web form as in the previous image. The error message has been updated to: "Must be one of Accept, Reject, or Neutral." The error message is now positioned to the right of the "Recommendation:" dropdown menu.

15. Now let's fix the root problem: no converter for the enumerated type **Comment.Recommendation**. Well, actually, Web Flow can handle enums okay – as long as the input is the exact string representation of one of the enumerated values. But this isn't usually good practice; rather, we want to separate the Java all-caps convention from the user-friendly presentation that's in place for the form.

- Another way to deal with this, and in fact probably a better one, would be to set **value** attributes in the `<option>` tags that did line up correctly as "ACCEPT", "REJECT", and "NEUTRAL".
- But we'll instead take the opportunity to plug in a custom validator of our own, and see that we can work it this way, too.

16. Open **ConversionService.java** again, and add code to the bottom of the **addDefaultConverters** method:

```
import org.springframework.binding (one line)
    .convert.converters.PropertyEditorConverter;
...
public void addDefaultConverters ()
{
    ...
    addConverter (new DateConverter ("M/d/yy"));
    addConverter (new PropertyEditorConverter
        (new RecommendationPropertyEditor (),
        Comment.Recommendation.class));
}
```

- This approach allows us to re-use an existing class, the **RecommendationPropertyEditor**, rather than writing a whole new converter class just for this bit of logic.

17. Build and test, and you should see that you can now add comments to a proposal, which means that the new converter is working.

Contributor	Recommendation
Sierra Club	<a href="#">REJECT</a>
Ernest DeLallo	<a href="#">REJECT</a>
Electronics Association	<a href="#">ACCEPT</a>
Franklin Amory	<a href="#">ACCEPT</a>
New contributor	<a href="#">NEUTRAL</a>

18. As a final exercise, you might want to add a `<binder>` to the `addComment` view state, as we should be requiring all fields here, just the way we do on most of our other forms.

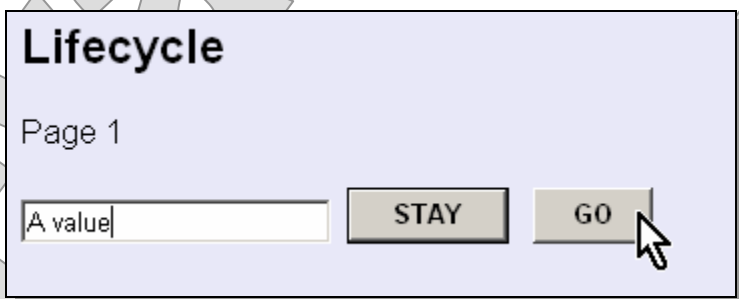
Your name:	<input type="text"/>	The contributor name is required.
Recommendation:	<input type="text" value="Neutral"/>	
Comment:	<input type="text"/>	The comment text is required.

- The Lifecycle application has been updated, in **Examples/Lifecycle/Step2**, to include traces of type conversion.
  - A custom converter and conversion service are now in place.
  - The converter – see `src/cc/webflow/Converter.java` – traces calls to `convertSourceToTargetClass`.
- Build and test the application at the following URL:

`http://localhost:8080/Lifecycle`

```
**** Starting flow
****   Entering state: page1
****   Rendering view: page1
****   Method called: bean.getValue()
```

- Enter a value in the text field and click **Go**.



- See the converter being invoked just before `Bean.setValue`:

```
****   Method called:
****       converter.convertSourceToTargetClass()
****   Method called: bean.setValue()
****   Making transition: from page1 to page2
****   Exited state: page1
****   Entering state: page2
****   Rendering view: page2
****   Method called: bean.getValue()
```

# A Declarative Approach

---

- The best approach we've seen so far to putting custom converters in place leaves something to be desired.
  - Subclassing **DefaultConversionService** and overriding **addDefaultConverters** means scripting our choices of custom converters in a Java class.
  - In other words, it's a purely **programmatic** solution.
- Much of Spring focuses on a more declarative style of development – especially where infrastructure is concerned.
- To wit: it would be nice to have a conversion service that would wire, or even auto-wire, to declared converters as Spring beans.
- This is trickier than it seems at first.
  - We can define an injectable **converters** property for our service.
  - But it turns out that **addDefaultConverters** is called by the base class **constructor**.
  - This is questionable Spring practice, as it means the method will be called before all properties have been set.
  - Put the opposite way, by the time our injectable list of converters has been initialized, our opportunity by way of overriding **addDefaultConverters** will already have passed by.
- So, to get a declarative solution, we have to take a step back ...

- In **Examples/LandUse/Step7**, the application has been refactored to use a more flexible implementation of **ConversionService**.
- Our old custom **gov.usda.usfs.landuse.web.ConversionService** is gone.
- In its place is **cc.webflow.ConversionService**.
  - It has an injectable property **converterList**, which is a **List<TwoWayConverter>**.
  - Since overriding **addDefaultConverters** wouldn't fly, this class takes a more direct role in the conversion process itself: it overrides **executeConversion**, and so is involved each time Web Flow wants to convert a value.
  - This logic is naturally a bit more complex, as we now have to assure not just that our converters are added to a list managed by the base class, but that they are actually used appropriately.
- The code is shown on the following page – with a great deal of exception-handling and tracing code left out so we can see the spine of the algorithm more clearly.
  - Basically, we check if the source type and target class **match** any of our converters; if so we call that converter.
  - Then we see if the source and target types match any of our converters **in reverse** – since Web Flow uses converters also to render strings from objects, and so forth.
  - Failing that, we just hand off to the base class and let one of the standard converters do the job.

```
@Override
public Object executeConversion
    (Object source, Class targetClass)
    throws ConversionException
{
    for (TwoWayConverter candidate : converterList)

        if (candidate.getTargetClass ()
            .equals (targetClass) &&
            candidate.getSourceClass ()
                .isInstance (source))
            try
            {
                return candidate.convertSourceToTargetClass
                    (source, targetClass);
            }
            catch (Exception ex) { ... }

        else if ((candidate.getSourceClass ()
            .equals (targetClass) ||
            targetClass == Object.class) &&
            candidate.getTargetClass ()
                .isInstance (source))
            try
            {
                return candidate.convertTargetToSourceClass
                    (source, targetClass);
            }
            catch (Exception ex) { ... }

    return super.executeConversion
        (source, targetClass);
}
```

- By declaring a bean of this type in the application context, we put ourselves in a position to declare our converters as well.
  - See `docroot/WEB-INF/LandUse-servlet.xml`:

```
<bean class=
  "gov.usda.usfs.landuse.web.DateConverter"
>
  <constructor-arg value="M/d/yy" />
</bean>

<bean class="org.springframework.binding
  .convert.converters.PropertyEditorConverter" >
  <constructor-arg>
    <bean class="gov.usda.usfs.landuse.web
      .RecommendationPropertyEditor" />
  </constructor-arg>
  <constructor-arg>
    <bean class="java.lang.Class"
      factory-method="forName" >
      <constructor-arg value="gov.usda.usfs.landuse
        .Comment$Recommendation" />
    </bean>
  </constructor-arg>
</bean>

<bean
  id="conversionService"
  class="cc.webflow.ConversionService"
  autowire="byType"
/>
```

- The declaration of a **PropertyEditorConverter** that wraps our existing **RecommendationPropertyEditor** is a bit of a mouthful – but even this does work in Spring’s configuration vocabulary.

**Suggested time: 30 minutes**

In this lab you will implement a few custom converters and plug them into the Binding application that we saw at the beginning of the chapter:

- A **BetterBooleanConverter** to read that checkbox correctly
- A **StringArrayConverter** to support the **roles** property
- A **DateConverter** to put a consistent date format in place

Detailed instructions are found at the end of the chapter.

## SUMMARY

- **The Web Flow system for conversion is simple to use, and we get pretty good results across the board.**
- **Custom converters are easy to implement, and moderately easy to install via a conversion service.**
  - The system of registering the conversion service is a little quirky.
  - It also tilts toward a programmatic style of registration, where a declarative one would have some real advantages.
  - A little extra code can put that declarative approach in play.
- **Converters are used for every type conversion in Web Flow – not just for reading HTTP request parameters.**
- **So custom converters must be written thoroughly, with the needs of EL expressions as well as HTTP request binding in mind.**