Rendering Views

Summary

This chapter shows you how to use the view-state element to render views within a flow.

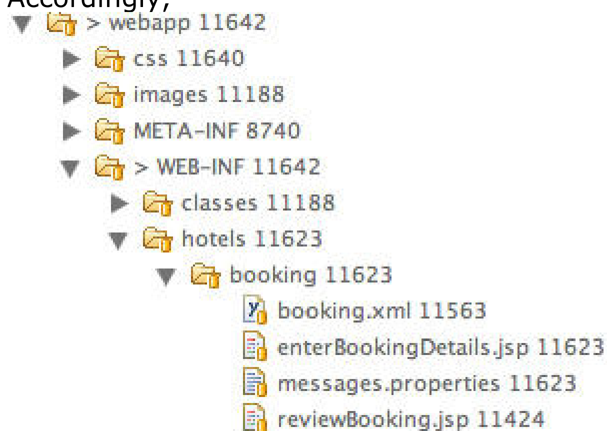
Description

Defining view states

The view-state creates the related view as default and waits for the user to respond through the screen. Below view-state has the enterBookingDetails ID. Since there’s no separate view configuration, the ID refers to view.

```
<view-state id="enterBookingDetails">
        <transition on="submit" to="reviewBooking" />
</view-state>
```

Accordingly,



enterBookingDetails.jsp in the directory where booking.xml(or booking-flow.xml) under directory exists, automatically works as a view.
Or view=”/WEB-INF/hotels/booking/enterBookingDetails.jsp” can be set explicitly using absolute path. We’ll explain again in the following.

Specifying view identifiers

Use the view attribute to specify the id of the view to render explicitly.

- Flow relative view ids

```
<view-state id="enterBookingDetails" view="bookingDetails.xhtml">
```

- Absikyte view ides

```
<view-state id="enterBookingDetails" view="/WEB-INF/hotels/booking/bookingDetails.jsp>
```

- Logical view ids: with Spring MVC’s view framework

```
<view-state id="enterBookingDetails" view="bookingDetails">
```

View scope

The view scope is the variable maintained in the view-state. The view scope is useful when the same view should be shown multiple times as Ajax request.

- Declare view variable using var tag.

```
<var name="searchCriteria" class="com.mycompany.myapp.hotels.SearchCriteria" />
```

- Allocate to viewScope variable

```
<on-render>
        <evaluate expression="bookingService.findHotels(searchCriteria)"
result="viewScope.hotels"/>
</on-render>
```

Treat Object in the view scope

Following code shows that screen ID draws view state screen of searchResults, but call bookingService.findHotels(searchCriteria) method after drawing, and shows the screen after saving the results as a hotel in viewScope.
When next or previous event occurs in the screen, eval(searchCriteria.nextPage()/previousPage()) occurs, and spreads the results to the designated area as a fragment.

```
<view-state id="searchResults">
        <on-render>
                <evaluate expression="bookingService.findHotels(searchCriteria)"
                        result="viewScope.hotels" />
        </on-render>
        <transition on="next">
                <evaluate expression="searchCriteria.nextPage()" />
                <render fragments="searchResultsFragment" />
        </transition>
        <transition on="previous">
                <evaluate expression="searchCriteria.previousPage()" />
                <render fragments="searchResultsFragment" />
        </transition>
</view-state>
```

Let's explain in details below.

Executing render actions

To execute certain actions before showing the view, use on-render.

```
<on-render>
  <evaluate expression="bookingService.findHotels(searchCriteria)" result="viewScope.hotels" />
</on-render>
```

Model Binding

```
<view-state id="enterBookingDetails" model="booking">
```

Following actions appear for the model designated when view event occurs.

1. view-to-model binding.
2. Validate model efficiency

Performing type conversion

Implementing a converter

To implement your own Converter, implement the org.springframework.binding.convert.converters.TwoWayConverter interface.

```
protected abstract Object toObject(String string, Class targetClass) throws Exception;
protected abstract String toString(Object object) throws Exception;
```

Example of implementation.

```
public class StringToMonetaryAmount extends StringToObject {
        public StringToMonetaryAmount() {
        super(MonetaryAmount.class);
  }
  @Override
  protected Object toObject(String string, Class targetClass) {
        return MonetaryAmount.valueOf(string);
  }
  @Override
  protected String toString(Object object) {
        MonetaryAmount amount = (MonetaryAmount) object;
        return amount.toString();
  }
}
```

Converter already implement is located at org.springframework.binding.convert.converters.

Registering a Converter

Redefine addDefaultConverts() method inheriting org.springframework.binding.convert.service.DefaultConversionService. For details, refer to configuring using ConversionService in the system configuration.

Suppressing binding

Use the bind attribute to suppress model binding and validation for particular view events.

```
<view-state id="enterBookingDetails" model="booking">
        <transition on="proceed" to="reviewBooking">
        <transition on="cancel" to="bookingCancelled" bind="false" />
</view-state>
```

Specifying bindings explicitly

Use the binder element to configure the exact set of model bindings usable by the view as below.

```
<view-state id="enterBookingDetails" model="booking">
        <binder>
                <binding property="creditCard" />
                <binding property="creditCardName" />
                <binding property="creditCardExpiryMonth" />
                <binding property="creditCardExpiryYear" />
        </binder>
        <transition on="proceed" to="reviewBooking" />
        <transition on="cancel" to="cancel" bind="false" />
</view-state>
```

If not designated as binder, binds all properties. Can designate converter using converter.

```
<view-state id="enterBookingDetails" model="booking">
        <binder>
                <binding property="checkinDate" converter="shortDate" />
                <binding property="checkoutDate" converter="shortDate" />
                <binding property="creditCard" />
```

```
                <binding property="creditCardName" />
                <binding property="creditCardExpiryMonth" />
                <binding property="creditCardExpiryYear" />
        </binder>
        <transition on="proceed" to="reviewBooking" />
        <transition on="cancel" to="cancel" bind="false" />
</view-state>
```

Validating a model

Model validation is supported by enforcing constraints programmatically on the Web Flow.

Validation in Program

First method is to define the validation logic in model object.
At the time when it is gone to model from view-stat(view-state postback lifecycle), Web Flow automatically calls validate method.

```
<view-state id="enterBookingDetails" model="booking">
        <transition on="proceed" to="reviewBooking">
</view-state>
```

validate{view-state name) code in Booking class can be seen as shown below. (method name: validate + EnterBookingDetails)

```
public class Booking {
        private Date checkinDate;
        private Date checkoutDate;
  ...
        public void validateEnterBookingDetails(ValidationContext context) {
                MessageContext messages = context.getMessages();
                if (checkinDate.before(today())) {
                        messages.addMessage(new  MessageBuilder().error().source("checkinDate").

                         defaultText("Check in date must be a future date").build());
                } else if (!checkinDate.before(checkoutDate)) {
                         messages.addMessage(new MessageBuilder().error()
                                                                     .source("checkoutDate")
                                                                     .defaultText("Check out date must
be later than check in date")
                                                                     .build());
                }
        }
}
```

When event for enterBookingDetails occurs, validateEnterBookingDetails is called automatically.
Define method name as validate$<state>.

Implementing a Validator

It can be defined as a separate object called a Validator. To do this, first create a class whose name has the pattern ${model}Validator. . Then define a public method with the name validate${state}. The class name below is Bokking+Validator, and the method name is validate+EnterBookingDetails.

```
@Component
public class BookingValidator {
        public void validateEnterBookingDetails(Booking booking, ValidationContext context) {
```

```
		MessageContext messages = context.getMessages();
		if (booking.getCheckinDate().before(today())) {
			messages.addMessage(new MessageBuilder().error()
									.source("checkinDate")
									.defaultText("Check in date must be
a future date")
									.build());
		} else if (!booking.getCheckinDate().before(booking.getCheckoutDate())) {
			messages.addMessage(new MessageBuilder().error()
									.source("checkoutDate")
									.defaultText("Check out date must
be later than check in date")
									.build());
		}
	}
}
```

Error object of spring mvc can be received.

ValidationContext

During validation, it enables to access MessageContext as well as various objects.

No Validation

May not perform validation by setting validate="false"

```
<view-state id="chooseAmenities" model="booking">
	<transition on="proceed" to="reviewBooking">
	<transition on="back" to="enterBookingDetails" validate="false" />
</view-state>
```

Execute View transition

Target of transition can be the request to show some views that is called 'fragments' when executing (1)other view, (2)current view again, (3)action, (4)control Ajax events.

Transition actions

```
<transition on="submit" to="bookingConfirmed">
	<evaluate expression="bookingAction.makeBooking(booking, messageContext)" />
</transition>
public class BookingAction {
	public boolean makeBooking(Booking booking, MessageContext context) {
		try {
			bookingService.make(booking);
			return true;
		} catch (RoomNotAvailableException e) {
			context.addMessage(builder.error().defaultText("No room is available at
this hotel").build());
			return false;
		}
	}
}
```

Global transitions

```
<global-transitions>
	<transition on="login" to="login">
	<transition on="logout" to="logout">
</global-transitions>
```

Event handlers

```
<transition on="event">
	<!-- Handle event -->
</transition>
```

Rendering fragments

Use the render element within a transition to request partial re-rendering of the current view after handling the event:

```
<transition on="next">
	<evaluate expression="searchCriteria.nextPage()" />
	<render fragments="searchResultsFragment" />
</transition>
```

Specify multiple elements to re-render by separating them with a comma delimiter.

Use Message

MessageContext is API used to save the message during flow execution.
General message or message supported with internationalization can be used.
Message level can be designated and the supported level includes info, warning and error. Use MessageBuilder to add message.

- Add general message

```
MessageContext context = ...
MessageBuilder builder = new MessageBuilder();
context.addMessage(builder.error().source("checkinDate").defaultText("Check in date must be a future date").build());
context.addMessage(builder.warn().source("smoking").defaultText("Smoking is bad for your health").build());
context.addMessage(builder.info().defaultText("We have processed your reservation - thank you and enjoy your stay").build());
```

- Add message supported with internationalization

```
MessageContext context = ...
MessageBuilder builder = new MessageBuilder();
context.addMessage(builder.error().source("checkinDate").code("checkinDate.notFuture").build());
context.addMessage(builder.warn().source("smoking").code("notHealthy").resolvableArg("smoking").build());
```

Using message bundles

Use the MessageSource on the Spring to define the message bundle. Simply manage as the property file.

```
#messages.properties
checkinDate=Check in date must be a future date
notHealthy={0} is bad for your health
reservationConfirmation=We have processed your reservation - thank you and enjoy your stay
```

View and flow can access with resourceBundle EL variable.

```
<h:outputText value="#{resourceBundle.reservationConfirmation}" />
```

Understanding system generated messages

It is possible to desginate messages on the exceptions created in the system. For example, if an error occurs at type conversion, message can be designated through typeMismatch.

booking.checkinDate.typeMismatch=The check in date must be in the format yyyy-mm-dd.

Displaying popups

When you want to rending the modal pop-up dialogue with a view, configure popup=”true” within the view-state.

<view-state id="changeSearchCriteria" view="enterSearchCriteria.xhtml" popup="true">

In particular, client code is not required at all for showing the popup if it is used with spring java script. SWF redirects the client request as popup.

View backtracking

As default, you can return to previous view-state with a back button. It is possible to configure this using the history.

- Set to 'discard' to prevent backtracking.

<transition on="cancel" to="bookingCancelled" history="discard">

- Set to 'invalidate' to prevent all views shown before as well as current view from backtracking.

<transition on="confirm" to="bookingConfirmed" history="invalidate">

Reference

- Spring Web Flow reference 2.0.x
- Spring Web-Flow Framework Reference beta with Korean (by Park Chan Wook)