5. Executing actions

5.1. Introduction

This chapter shows you how to use the `action-state` element to control the execution of an action at a point within a flow. It will also show how to use the `decision-state` element to make a flow routing decision. Finally, several examples of invoking actions from the various points possible within a flow will be discussed.

5.2. Defining action states

Use the `action-state` element when you wish to invoke an action, then transition to another state based on the action's outcome:

```
<action-state id="moreAnswersNeeded">
    <evaluate expression="interview.moreAnswersNeeded()" />
    <transition on="yes" to="answerQuestions" />
    <transition on="no" to="finish" />
</action-state>
```

The full example below illustrates a interview flow that uses the action-state above to determine if more answers are needed to complete the interview:

```
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/webflow
                          http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd">

    <on-start>
        <evaluate expression="interviewFactory.createInterview()" result="flowScope.interview" />
    </on-start>

    <view-state id="answerQuestions" model="questionSet">
        <on-entry>
            <evaluate expression="interview.getNextQuestionSet()" result="viewScope.questionSet" />
        </on-entry>
        <transition on="submitAnswers" to="moreAnswersNeeded">
            <evaluate expression="interview.recordAnswers(questionSet)" />
        </transition>
    </view-state>

    <action-state id="moreAnswersNeeded">
        <evaluate expression="interview.moreAnswersNeeded()" />
        <transition on="yes" to="answerQuestions" />
        <transition on="no" to="finish" />
    </action-state>

    <end-state id="finish" />

</flow>
```

5.3. Defining decision states

Use the `decision-state` element as an alternative to the action-state to make a routing decision using a convenient if/else syntax. The example below shows the

`moreAnswersNeeded` state above now implemented as a decision state instead of an action-state:

```
<decision-state id="moreAnswersNeeded">
    <if test="interview.moreAnswersNeeded()" then="answerQuestions" else="finish" />
</decision-state>
```

5.4. Action outcome event mappings

Actions often invoke methods on plain Java objects. When called from action-states and decision-states, these method return values can be used to drive state transitions. Since transitions are triggered by events, a method return value must first be mapped to an Event object. The following table describes how common return value types are mapped to Event objects:

Table 5.1. Action method return value to event id mappings

Method return type	Mapped Event identifier expression
java.lang.String	the String value
java.lang.Boolean	yes (for true), no (for false)
java.lang.Enum	the Enum name
any other type	success

This is illustrated in the example action state below, which invokes a method that returns a boolean value:

```
<action-state id="moreAnswersNeeded">
    <evaluate expression="interview.moreAnswersNeeded()" />
    <transition on="yes" to="answerQuestions" />
    <transition on="no" to="finish" />
</action-state>
```

5.5. Action implementations

While writing action code as POJO logic is the most common, there are several other action implementation options. Sometimes you need to write action code that needs access to the flow context. You can always invoke a POJO and pass it the flowRequestContext as an EL variable. Alternatively, you may implement the `Action` interface or extend from the `MultiAction` base class. These options provide stronger type safety when you have a natural coupling between your action code and Spring Web Flow APIs. Examples of each of these approaches are shown below.

Invoking a POJO action

```
<evaluate expression="pojoAction.method(flowRequestContext)" />
```

```
public class PojoAction {
    public String method(RequestContext context) {
        ...
    }
}
```

Invoking a custom Action implementation

```
<evaluate expression="customAction" />
```

```
public class CustomAction implements Action {
    public Event execute(RequestContext context) {
        ...
    }
}
```

Invoking a MultiAction implementation

```
<evaluate expression="multiAction.actionMethod1" />
```

```
public class CustomMultiAction extends MultiAction {
    public Event actionMethod1(RequestContext context) {
        ...
    }

    public Event actionMethod2(RequestContext context) {
        ...
    }

    ...
}
```

5.6. Action exceptions

Actions often invoke services that encapsulate complex business logic. These services may throw business exceptions that the action code should handle.

Handling a business exception with a POJO action

The following example invokes an action that catches a business exception, adds a error message to the context, and returns a result event identifier. The result is treated as a flow event which the

calling flow can then respond to.

```
<evaluate expression="bookingAction.makeBooking(booking, flowRequestContext)" />
```

```
public class BookingAction {
   public String makeBooking(Booking booking, RequestContext context) {
       try {
           BookingConfirmation confirmation = bookingService.make(booking);
           context.getFlowScope().put("confirmation", confirmation);
           return "success";
       } catch (RoomNotAvailableException e) {
           context.addMessage(new MessageBuilder().error().
               .defaultText("No room is available at this hotel").build());
           return "error";
       }
   }
}
```

Handling a business exception with a MultiAction

The following example is functionally equivlant to the last, but implemented as a MultiAction instead of a POJO action. The MultiAction requires its action methods to be of the signature `Event ${methodName}(RequestContext)`, providing stronger type safety, while a POJO action allows for more freedom.

```
<evaluate expression="bookingAction.makeBooking" />
```

```
public class BookingAction extends MultiAction {
   public Event makeBooking(RequestContext context) {
       try {
           Booking booking = (Booking) context.getFlowScope().get("booking");
           BookingConfirmation confirmation = bookingService.make(booking);
           context.getFlowScope().put("confirmation", confirmation);
           return success();
       } catch (RoomNotAvailableException e) {
           context.getMessageContext().addMessage(new MessageBuilder().error().
               .defaultText("No room is available at this hotel").build());
           return error();
       }
   }
}
```

5.7. Other Action execution examples

on-start

The following example shows an action that creates a new Booking object by invoking a method on a service:

```
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/webflow
                          http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd">

    <input name="hotelId" />
```

```
    <on-start>
        <evaluate expression="bookingService.createBooking(hotelId, currentUser.name)"
                  result="flowScope.booking" />
    </on-start>

</flow>
```

on-entry

The following example shows a state entry action that sets the special `fragments` variable that causes the view-state to render a partial fragment of its view:

```
<view-state id="changeSearchCriteria" view="enterSearchCriteria.xhtml" popup="true">
    <on-entry>
        <render fragments="hotelSearchForm" />
    </on-entry>
</view-state>
```

on-exit

The following example shows a state exit action that releases a lock on a record being edited:

```
<view-state id="editOrder">
    <on-entry>
        <evaluate expression="orderService.selectForUpdate(orderId, currentUser)"
                  result="viewScope.order" />
    </on-entry>
    <transition on="save" to="finish">
        <evaluate expression="orderService.update(order, currentUser)" />
    </transition>
    <on-exit>
        <evaluate expression="orderService.releaseLock(order, currentUser)" />
    </on-exit>
</view-state>
```

on-end

The following example shows the equivalent object locking behavior using flow start and end actions:

```
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/webflow
                          http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd">

    <input name="orderId" />

    <on-start>
        <evaluate expression="orderService.selectForUpdate(orderId, currentUser)"
                  result="flowScope.order" />
    </on-start>

    <view-state id="editOrder">
        <transition on="save" to="finish">
            <evaluate expression="orderService.update(order, currentUser)" />
        </transition>
    </view-state>

    <on-end>
        <evaluate expression="orderService.releaseLock(order, currentUser)" />
    </on-end>

</flow>
```

on-render

The following example shows a render action that loads a list of hotels to display before the view is rendered:

```
<view-state id="reviewHotels">
    <on-render>
        <evaluate expression="bookingService.findHotels(searchCriteria)"
                  result="viewScope.hotels" result-type="dataModel" />
    </on-render>
    <transition on="select" to="reviewHotel">
        <set name="flowScope.hotel" value="hotels.selectedRow" />
    </transition>
</view-state>
```

on-transition

The following example shows a transition action adds a subflow outcome event attribute to a collection:

```
<subflow-state id="addGuest" subflow="createGuest">
    <transition on="guestCreated" to="reviewBooking">
        <evaluate expression="booking.guestList.add(currentEvent.attributes.newGuest)" />
    </transition>
</subfow-state>
```

Named actions

The following example shows how to execute a chain of actions in an action-state. The name of each action becomes a qualifier for the action's result event.

```
<action-state id="doTwoThings">
    <evaluate expression="service.thingOne()">
        <attribute name="name" value="thingOne" />
    </evaluate>
    <evaluate expression="service.thingTwo()">
        <attribute name="name" value="thingTwo" />
    </evaluate>
    <transition on="thingTwo.success" to="showResults" />
</action-state>
```

In this example, the flow will transition to `showResults` when `thingTwo` completes successfully.

Streaming actions

Sometimes an Action needs to stream a custom response back to the client. An example might be a flow that renders a PDF document when handling a print event. This can be achieved by having the action stream the content then record "Response Complete" status on the ExternalContext. The responseComplete flag tells the pausing view-state not to render the response because another object has taken care of it.

```
<view-state id="reviewItinerary">
    <transition on="print">
        <evaluate expression="printBoardingPassAction" />
    </transition>
</view-state>
```

```
public class PrintBoardingPassAction extends AbstractAction {
    public Event doExecute(RequestContext context) {
        // stream PDF content here...
        // - Access HttpServletResponse by calling context.getExternalContext().getNativeResponse();
        // - Mark response complete by calling context.getExternalContext().recordResponseComplete();
        return success();
    }
}
```

In this example, when the print event is raised the flow will call the printBoardingPassAction. The action will render the PDF then mark the response as complete.

Handling File Uploads

Another common task is to use Web Flow to handle multipart file uploads in combination with Spring MVC's `MultipartResolver`. Once the resolver is set up correctly as described here and the submitting HTML form is configured with `enctype="multipart/form-data"`, you can easily handle the file upload in a transition action. Given a form such as:

```
<form:form modelAttribute="fileUploadHandler" enctype="multipart/form-data">
        Select file: <input type="file" name="file"/>
        <input type="submit" name="_eventId_upload" value="Upload" />
</form:form>
```

and a backing object for handling the upload such as:

```
package org.springframework.webflow.samples.booking;

import org.springframework.web.multipart.MultipartFile;

public class FileUploadHandler {

    private transient MultipartFile file;

    public void processFile() {
                //Do something with the MultipartFile here
    }

    public void setFile(MultipartFile file) {
        this.file = file;
    }
}
```

you can process the upload using a transition action as in the following example:

```
<view-state id="uploadFile" model="uploadFileHandler">
    <var name="fileUploadHandler" class="org.springframework.webflow.samples.booking.FileUploadHandler" />
    <transition on="upload" to="finish" >
        <evaluate expression="fileUploadHandler.processFile()"/>
    </transition>
    <transition on="cancel" to="finish" bind="false"/>
</view-state>
```

The `MultipartFile` will be bound to the `FileUploadHandler` bean as part of the normal form binding process so that it will be available to process during the execution of the transition action.