Executing actions

Summary

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.

Description

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:
Intuitively seen, it can be expected that transition may be performed by the result value of interview.moreAnswersNeeded() from the following code.

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

Let's examine more perfect example.

```
<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>
```

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>
```

Action outcome event mapping

Actions often invoke methods on plain Java objects. When action-state and decision-state are called, the value returned by this method, the value returned by this value should be able to use to transit the status. Since the transition is occurred by event, the value returned by method should be mapped to Event object.
Following table explains how the Event object is mapped according to the value type commonly returned.
Expression of event identifier mapped by method return type

Type returned by result	Event value mapped
java.lang.String	String value
java.lang.Boolean	yes(corresponding to true), no(corresponding to false)
java.lang.Enum Enum	Enum name
Other type	success

Example.moreAnswersNeeded() method return type is expected to be boolean and can be mapped to yes or no accordingly.

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

Action implementations

While writing action code as POJO logic is the most common, there are several other action implementation options.
Sometimes, it is required to create the action code that should be accessed to flow context sometimes. At this time, POJO is called or flowRequestContext can be delivered with EL variable.
Instead, Action interface should be implemented or MultiAction default class can be inherited.

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) {
  ...
        }

}
```

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

```
<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().efaultText("No room is
available at this hotel").build());

                        return "error";
                }
        }
}
```

Control business exception when using MultiAction

While following example is same as the previous example functionally, it is implemented in MultiAction instead of POJO action.
Method can be configured according to Event ${methodName}(RequestContext) restriction and more stronger type of stability is provided compared to freedom of POJO.

```
<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();
                }
        }
}
```

Other Action execution examples

on-start

```
<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

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

on-exit

```
<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

```
<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

```
<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

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

Named actions

```
<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>
```

Streaming actions

Following example shows that printBoardingPassAction is called from flow and is the example of implementing when printing with PDF.
Implement actual pdf related source in doExecute() method of PrintBoardingPassAction inheriting AbstractAction and return success().

```
<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();
  }
}
```

Reference

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