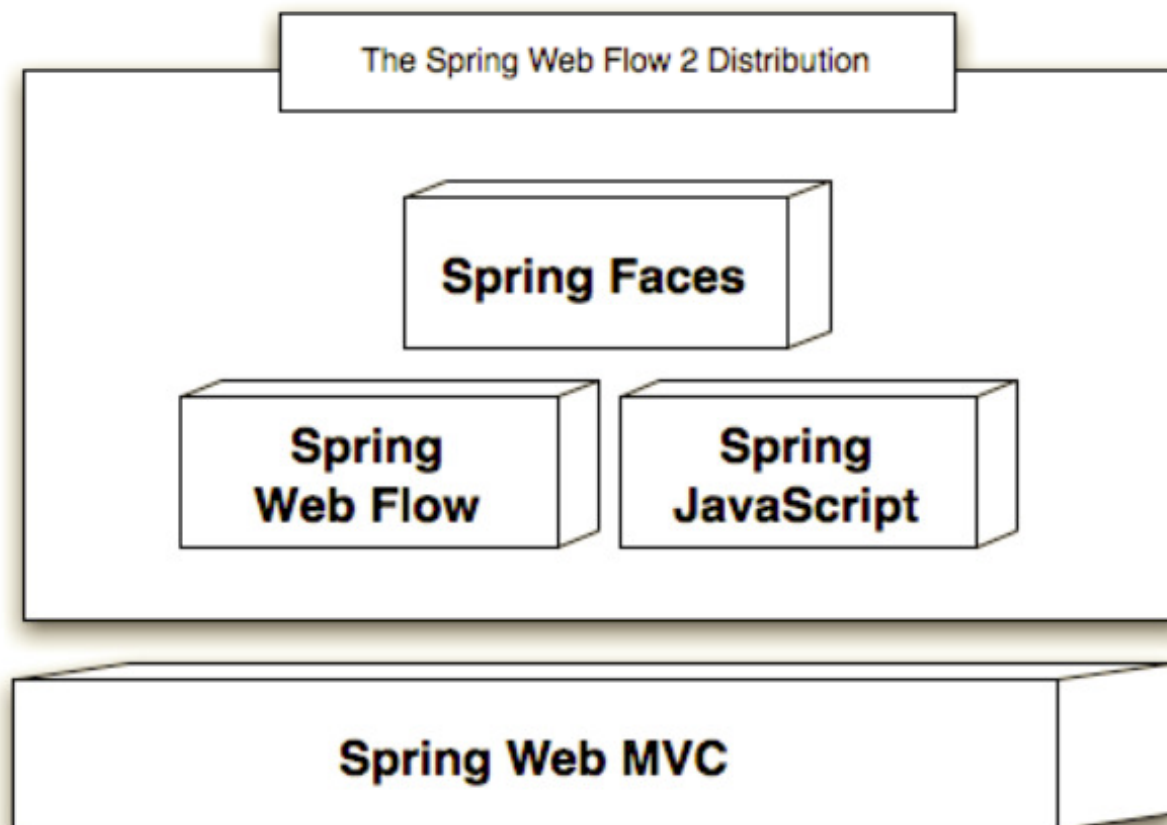


Spring Web Flow is the project in the Spring Portfolio that focuses on providing the infrastructure for building and running rich web applications. As a Spring project, Web Flow builds on the Spring Web MVC framework to provide:

- A domain-specific-language for defining reusable controller modules called [flows](#)
- An advanced controller engine for managing conversational state
- First-class support for using Ajax to construct rich user interfaces
- First-class support for using JavaServerFaces with Spring

The modules of the Web Flow 2 distribution and their relationship with the Spring Framework are illustrated below:

What's in Web Flow 2



Spring Web MVC

The Spring Web MVC framework, a module of the Spring Framework distribution, provides the foundation for developing web applications with Spring using the proven ModelViewController paradigm. Each of the modules of the Web Flow distribution builds on this foundation.

Spring Web Flow

The Web Flow module is a MVC extension that allows you to define Controllers using a [domain-specific-language](#). This language is designed to model user interactions that require several requests into the server to complete, or may be invoked from different contexts.

Spring JavaScript

[Spring JavaScript](#) is a JavaScript abstraction framework that makes it easy to write unobtrusive JavaScript to progressively enhance a web page with behavior. The framework consists of a public JavaScript API along with an implementation that builds on the Dojo Toolkit. Spring.js aims to simplify the use of Dojo for common enterprise scenarios while retaining its full-power for advanced use cases.

Spring JavaScript can work with *any* server-side framework. The Web Flow 2 distribution includes convenient integration between Spring JavaScript and Spring Web MVC for processing Ajax requests.

Spring Faces

The [Spring Faces](#) module contains Spring's support for JavaServerFaces. This support allows you to use JSF as a View technology within a familiar Spring MVC and Web Flow Controller environment. With this architectural approach, you combine the benefits of the JSF UI component model with the benefits of a Web MVC architecture. Spring Faces also includes a lightweight component library built on Spring JavaScript for declaratively enabling Ajax and client-side validation behaviors in a progressive manner.

WebFlow

- In traditional web application development, developers often manage their UI flows programmatically, so these flows are hard to maintain and reuse.
- Spring Web Flow offers a flow definition language that can help separate UI flows from presentation logic in a highly configurable way, so the flows can be easily changed and reused.
- Spring Web Flow supports not just Spring Web MVC, but also Spring Portlet MVC and other web application frameworks such as Struts and JSF.

Managing a Simple UI Flow with Spring Web Flow

Problem

You would like to manage a simple UI flow in a Spring MVC application using Spring Web Flow.

Solution

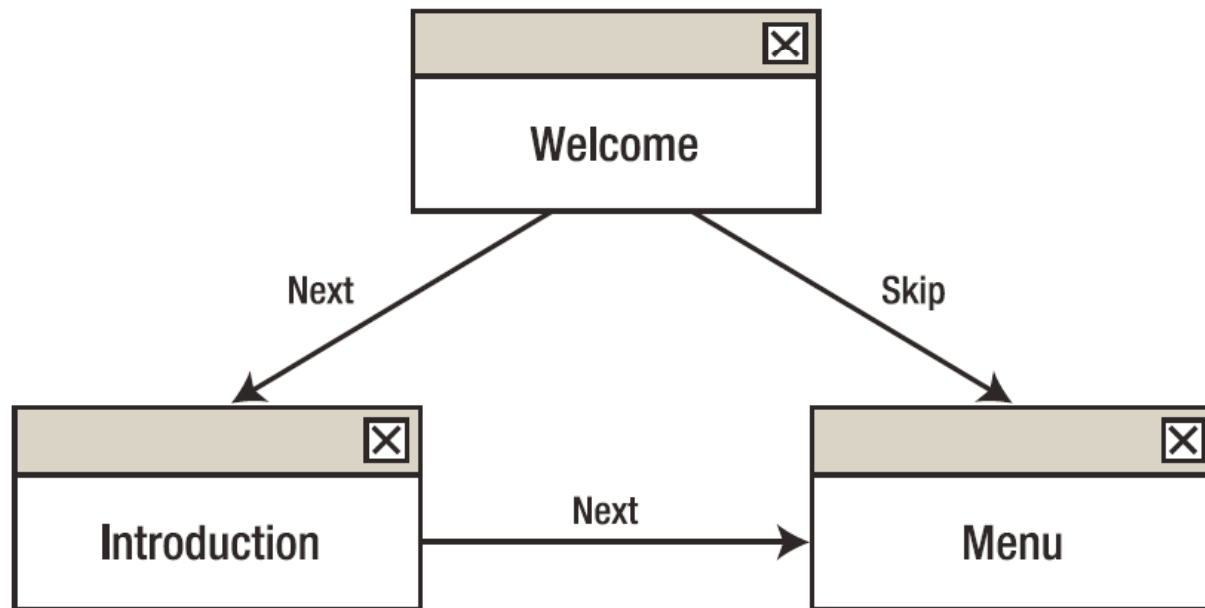
Spring Web Flow allows you to model UI activities as *flows*. It supports defining a flow either by Java or by XML. XML-based flow definitions are widely used due to the power and popularity of XML. You can also easily modify your XML-based flow definitions without recompiling your code. Moreover, Spring IDE supports Spring Web Flow by offering a visual editor for you to edit XML-based flow definitions.

A flow definition consists of one or more *states*, each of which corresponds to a step in the flow. Spring Web Flow builds in several state types, including view state, action state, decision state, subflow state, and end state. Once a state has completed its tasks, it fires an *event*. An event contains a source and an event ID, and perhaps some attributes. Each state may contain zero or more *transitions*, each of which maps a returned event ID to the next state.


When a user triggers a new flow, Spring Web Flow can auto-detect the start state of that flow (i.e., the state without transitions from other states), so you don't need to specify the start state explicitly. A flow can terminate at one of its defined end states. This marks the flow as ended and releases resources held by the flow.

How It Works

Suppose you are going to develop an online system for a library. The first page of this system is a welcome page. There are two links on this page. When a user clicks on the Next link, the system will show the library introduction page. There's another Next link on this introduction page, clicking which will show the menu page. If a user clicks the Skip link on the welcome page, the system will skip the introduction page and show the menu page directly. This welcome UI flow is illustrated in Figure 15-1. This example will show you how to develop this application with Spring MVC and use Spring Web Flow to manage the flow.



In the library introduction page, you would like to show the recent books which this library purchased recently. They are queried from a back-end using the following bookDAO class




```
package com.me.flow;

import java.util.List;
import java.util.ArrayList;

public class BookDAO
{
    public List<String> getBooks()
    {
        List<String> books = new ArrayList<String>();
        books.add("Spring MVC Programming");
        books.add("Oracle 10g");
        books.add("Microsoft SQL Server");
        books.add("Java EE Programming");
        return books;
    }
}
```


In the web deployment descriptor (i.e., web.xml), you register ContextLoaderListener to load the root application context at startup, and also Spring MVC's DispatcherServlet for dispatching requests.



```
<web-app>
<context-param>
<param-name>contextConfigLocation</param-name>
<param-value>/WEB-INF/library-service.xml</param-value>
</context-param>
<listener>
<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
<servlet>
<servlet-name>library</servlet-name>
<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>library</servlet-name>
<url-pattern>/flow/*</url-pattern>
</servlet-mapping>
</web-app>
```

ContextLoaderListener will load the root application context from the configuration file you specify in the contextConfigLocation context parameter.

This configuration file declares beans in the DAO layer.



```
<?xml version="1.0" encoding="UTF-8"?>
```


```
<beans>
```

```
<!-- Root Context: defines shared resources visible to all other web components -->
```

```
<bean name="bookDao" class="com.me.flow.BookDAO"/>
```


```
</beans>
```


In order to separate the configurations of Spring MVC and Spring Web Flow, you can centralize Spring Web Flow's configurations in another file
Then create this file with the following contents:



```
<beans >
<bean name="/" class="org.springframework.webflow.mvc.servlet.FlowController">
<property name="flowExecutor" ref="flowExecutor" />
</bean>
<webflow:flow-executor id="flowExecutor" />
<webflow:flow-registry id="flowRegistry">
<webflow:flow-location path="/WEB-INF/flows/welcome.xml" />
</webflow:flow-registry>
</beans>
```

Creating Web Flow Definitions



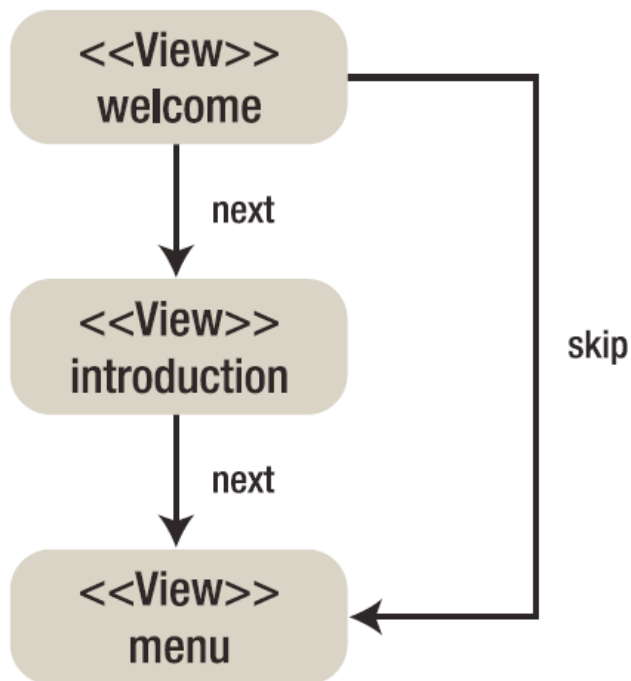
```
<flow >
<view-state id="welcome">
  <transition on="next" to="introduction" />
  <transition on="skip" to="menu" />
</view-state>

<view-state id="introduction">
  <on-render>
    <evaluate expression="bookDao.getBooks()" result="requestScope.books" />
  </on-render>
  <transition on="next" to="menu" />
</view-state>

<view-state id="menu" />
</flow>
```

You can use the `<on-render>` element to trigger an action for a view state before its view renders. Spring Web Flow supports using an expression of Unified EL or OGNL to invoke a method. For more about Unified EL and OGNL, please refer to the article “Unified Expression Language,” at <http://java.sun.com/products/jsp/reference/techart/unifiedEL.html>, and the OGNL language guide, at <http://www ognl.org/>. The preceding expression is valid for both Unified EL and OGNL. It invokes the `getHolidays()` method on the `libraryService` bean and stores the result in the `holidays` variable in the request scope.

The flow diagram for this welcome flow is illustrated in Figure 15-2.



Modeling Web Flows with Different State Types

Problem

You would like to model various types of UI activities as web flows to execute in Spring Web Flow.

Solution

In Spring Web Flow, each step of a flow is denoted by a state. A state may contain zero or more transitions to the next states according to an event ID. Spring Web Flow provides several built-in state types for you to model web flows. It also allows you to define custom state types.

Table 15-1 shows the built-in state types in Spring Web Flow.

Table 15-1. *Built-In State Types in Spring Web Flow*

State Type	Description
View	Renders a view for a user to participate in the flow (e.g., by displaying information and gathering user input). The flow's execution pauses until an event is triggered to resume the flow (e.g., by a hyperlink click or a form submission).
Action	Executes actions for the flow, such as updating a database and gathering information for displaying.
Decision	Evaluates a Boolean expression to decide which state to transition to next.
Subflow	Launches another flow as a subflow of the current flow. The subflow will return to the launching flow when it ends.
End	Terminates the flow, after which all flow scope variables become invalid.

How It Works

Suppose you are going to build a web flow for library users to search books. First, a user has to enter the book criteria in the criteria page. If there's more than one book matching the criteria, they will be displayed in the list page. In this page, the user can select a book to browse its details in the details page. However, if there's exactly one book matching the criteria, its details will be shown directly in the details page, without going through the list page. This book search UI flow is illustrated in Figure 15-3.

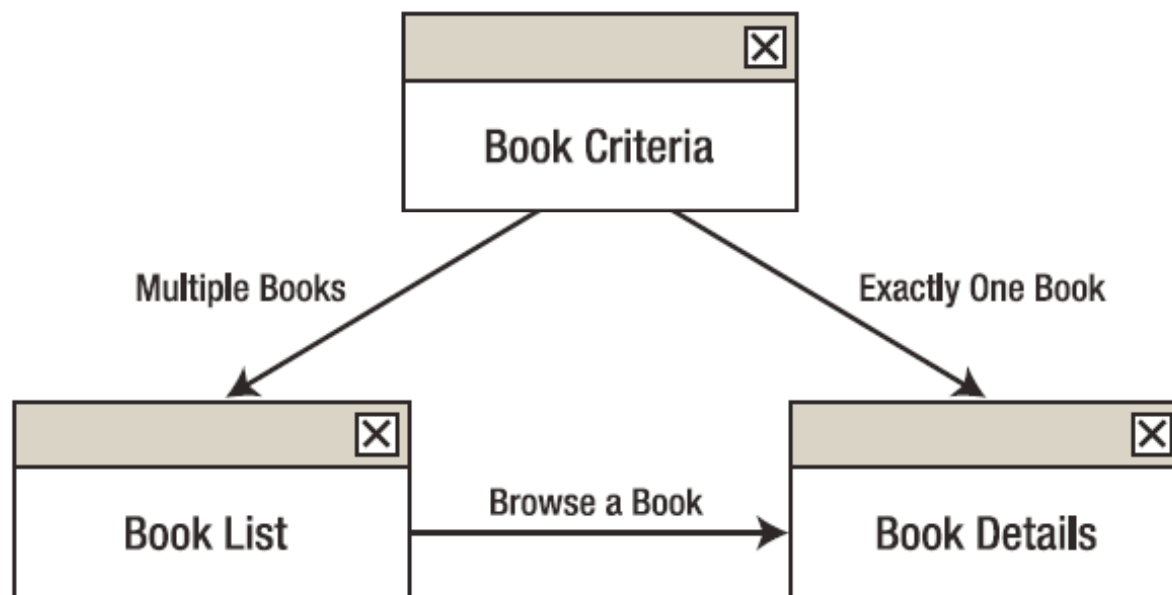


Figure 15-3. *The book search UI flow*

First of all, let's create the domain class `Book`. It should implement the `Serializable` interface, as its instances may need to be persisted in sessions.

Now you can deploy this application and test this simplified book search flow with the URL <http://localhost:8080/library/flow/bookSearch>. The current flow diagram for this book search flow is illustrated in Figure 15-4.

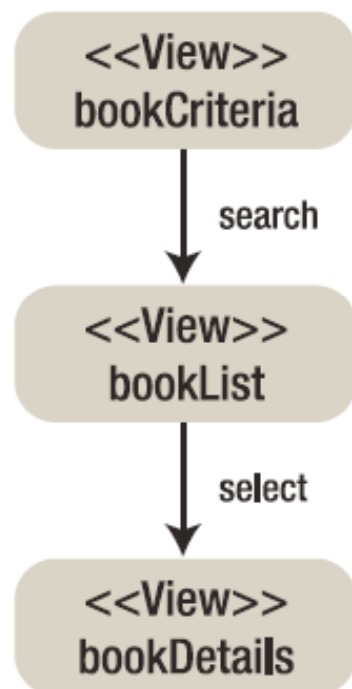


Figure 15-4. *The flow diagram for the book search flow with view states only*

Defining Action States

Although you can include the search action in the `bookCriteria` view state, this action cannot be reused for other states that require book searching. The best practice is to extract reusable actions into stand-alone action states. An action state simply defines one or more actions to perform in a flow, and these actions will be performed in the declared order. If an action returns an Event object containing an ID that matches a transition, the transition will take place immediately without performing the subsequent actions. But if all of the actions have been performed without a matching transition, the success transition will take place.

For the purposes of reuse, you can extract the book search action in a `searchBook` state. Then modify the transition of the `bookCriteria` state to go through this state.

```
<flow ...>
  <view-state id="bookCriteria">
    <on-render>
      <evaluate expression="bookCriteriaAction.setupForm" />
    </on-render>
    <transition on="search" to="searchBook">
      <evaluate expression="bookCriteriaAction.bindAndValidate" />
    </transition>
  </view-state>
  <action-state id="searchBook">
    <evaluate expression="bookService.search(bookCriteria)"
      result="flowScope.books" />
    <transition on="success" to="bookList" />
  </action-state>
  ...
</flow>
```

The current flow diagram for this book search flow is illustrated in Figure 15-5.

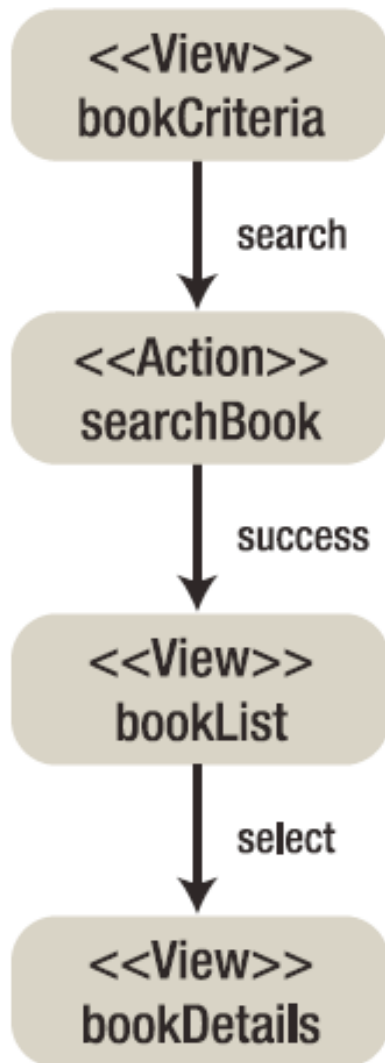


Figure 15-5. *The flow diagram for the book search flow with an action state*

Defining Decision States

Now you are going to satisfy the book search flow's requirement: if there's more than one search result, display them in the list page; otherwise, show its details directly in the details page without going through the list page. For this purpose, you need a decision state that can evaluate a Boolean expression to determine the transition:

```
<flow ...>
  ...
  <action-state id="searchBook">
    <evaluate expression="bookService.search(bookCriteria)"
      result="flowScope.books" />
    <transition on="success" to="checkResultSize" />
  </action-state>

  <decision-state id="checkResultSize">
    <if test="books.size() == 1" then="extractResult" else="bookList" />
  </decision-state>

  <action-state id="extractResult">
    <set name="flowScope.book" value="books.get(0)" />
    <transition on="success" to="bookDetails" />
  </action-state>
  ...
</flow>
```

The success transition of the searchBook state has been changed to checkResultSize, a decision state that checks if there's exactly one search result. If true, it will transition to the extractResult action state to extract the first and only result into the flow scope variable book. Otherwise, it will transition to the bookList state to display all search results in the list page. The current flow diagram is illustrated in Figure 15-6.

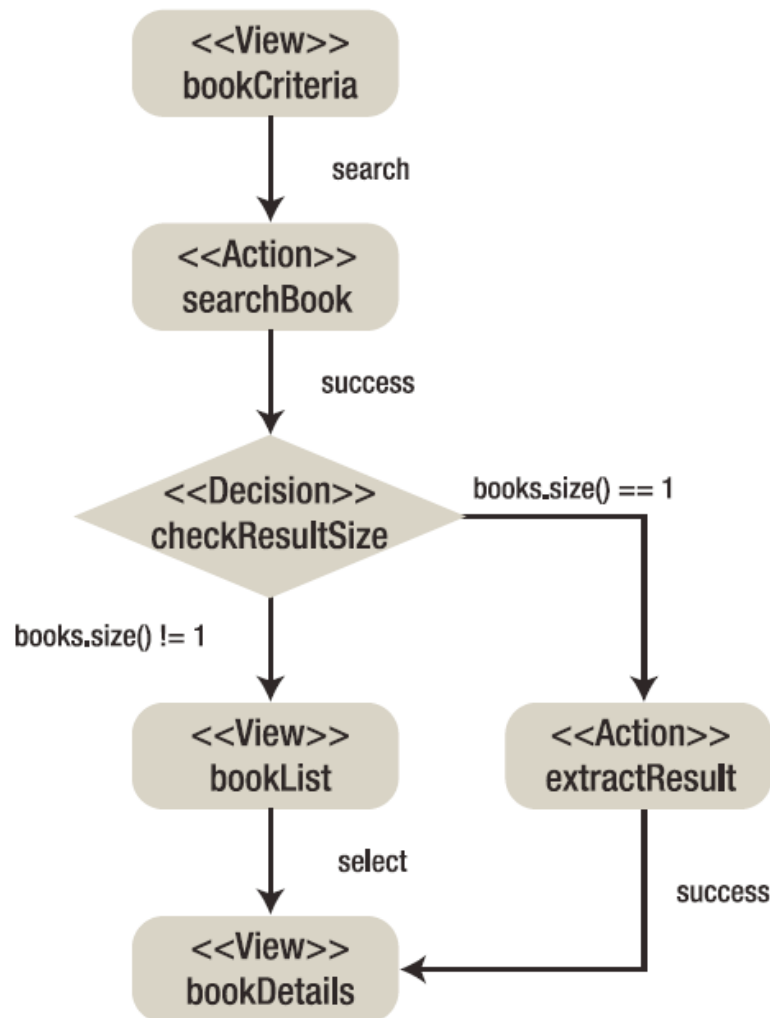


Figure 15-6. *The flow diagram for the book search flow with a decision state*

Defining End States

The basic requirement of the book search flow has been finished. However, you might be asked to provide a New Search link in both the book list page and the book details page for starting a new search. You can simply add the following link to both pages:

```
<a href="${flowExecutionUrl}&_eventId=newSearch">New Search</a>
```

As you can see, this link will trigger a `newSearch` event. Instead of transitioning to the first view state, `bookCriteria`, it's better to define an end state that restarts this flow. An end state will invalidate all flow scope variables to release their resources.

```
<flow ...>
  ...
  <view-state id="bookList">
    <transition on="select" to="bookDetails">
      <evaluate expression="bookService.findByIsbn(requestParameters.isbn)"
        result="flowScope.book" />
    </transition>
    <transition on="newSearch" to="newSearch" />
  </view-state>

  <view-state id="bookDetails">
    <transition on="newSearch" to="newSearch" />
  </view-state>

  <end-state id="newSearch" />
</flow>
```

By default, an end state restarts the current flow to the start state, but you can redirect to another flow by specifying the flow name with `flowRedirect` as the prefix in the end state's view attribute. The current flow diagram is illustrated in Figure 15-7.

By default, an end state restarts the current flow to the start state, but you can redirect to another flow by specifying the flow name with `flowRedirect` as the prefix in the end state's `view` attribute. The current flow diagram is illustrated in Figure 15-7.

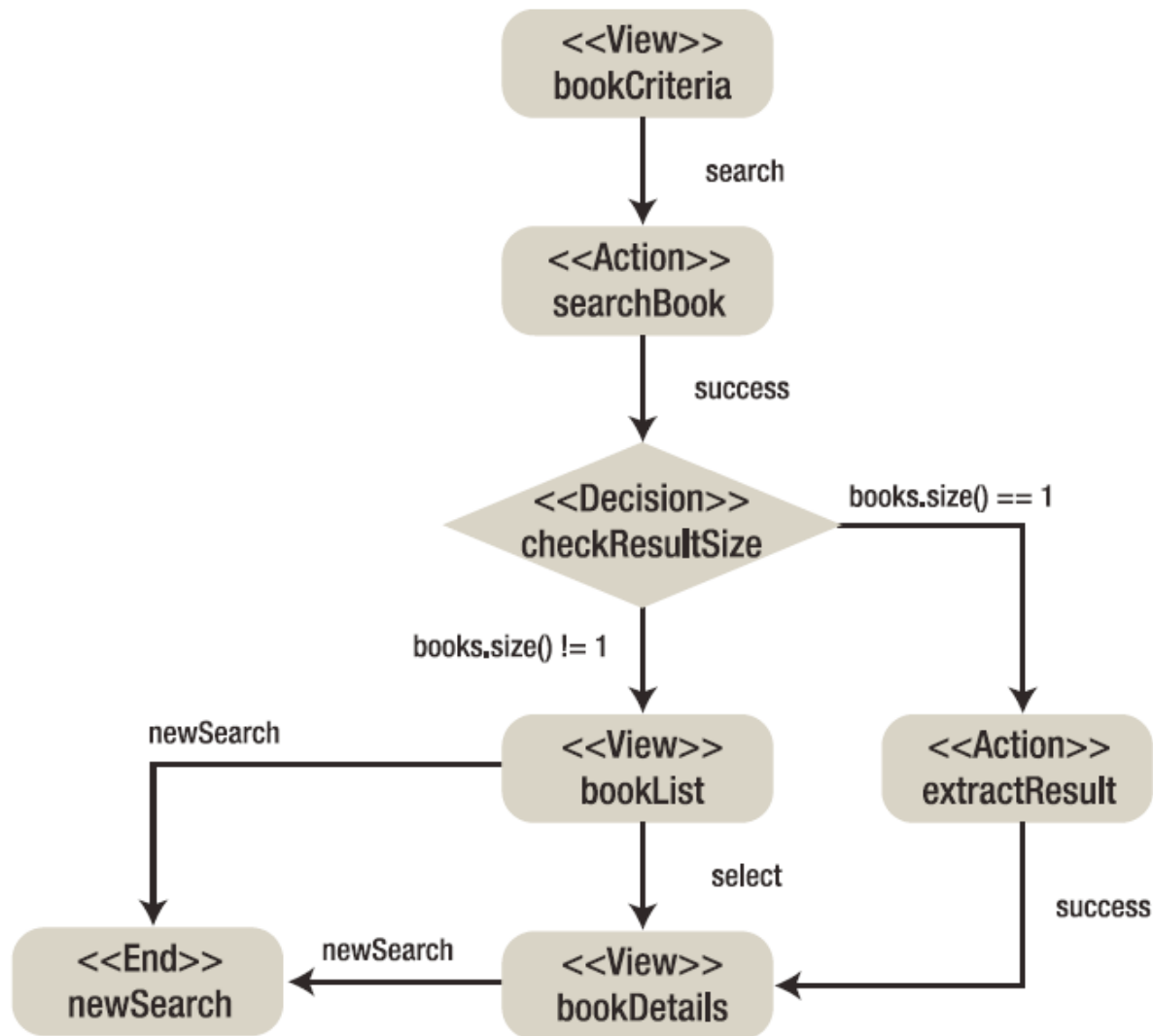


Figure 15-7. The flow diagram for the book search flow with an end state

Defining Subflow States

Suppose you have another web flow that also requires showing a book's details. For reuse purposes, you extract the bookDetails state into a new web flow that can be called by other flows as a subflow. First of all, you have to add the flow definition's location to the flow registry in library-webflow.xml and create the flow definition XML file as well:

```
<webflow:flow-registry id="flowRegistry">
    ...
    <webflow:flow-location path="/WEB-INF/flows/bookDetails/bookDetails.xml" />
</webflow:flow-registry>
```

Then you move the bookDetails view state to this flow and bookDetails.jsp to this directory. As the bookDetails state has a transition to the newSearch state, you also define it in this flow as an end state.

```
<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="book" value="flowScope.book" />

    <view-state id="bookDetails">
        <transition on="newSearch" to="newSearch" />
    </view-state>

    <end-state id="newSearch" />
</flow>
```

The book instance to be shown is passed as an input parameter to this flow with the name book, and it is stored in the flow scope variable book. Note that the flow scope is only visible to the current flow.

Then you define a subflow state in the bookSearch flow that launches the bookDetails flow to show a book's details:

```
<subflow-state id="bookDetails" subflow="bookDetails">  
  <input name="book" value="flowScope.book" />  
  <transition on="newSearch" to="newSearch" />  
</subflow-state>
```

In this subflow state definition, you pass the book variable in the flow scope of the bookSearch flow to the bookDetails subflow as an input parameter. When the bookDetails subflow ends in its newSearch state, it will transition to the newSearch state of the parent flow, which happens to be the newSearch end state of the bookSearch flow in this case. The current flow diagram is illustrated in Figure 15-8.

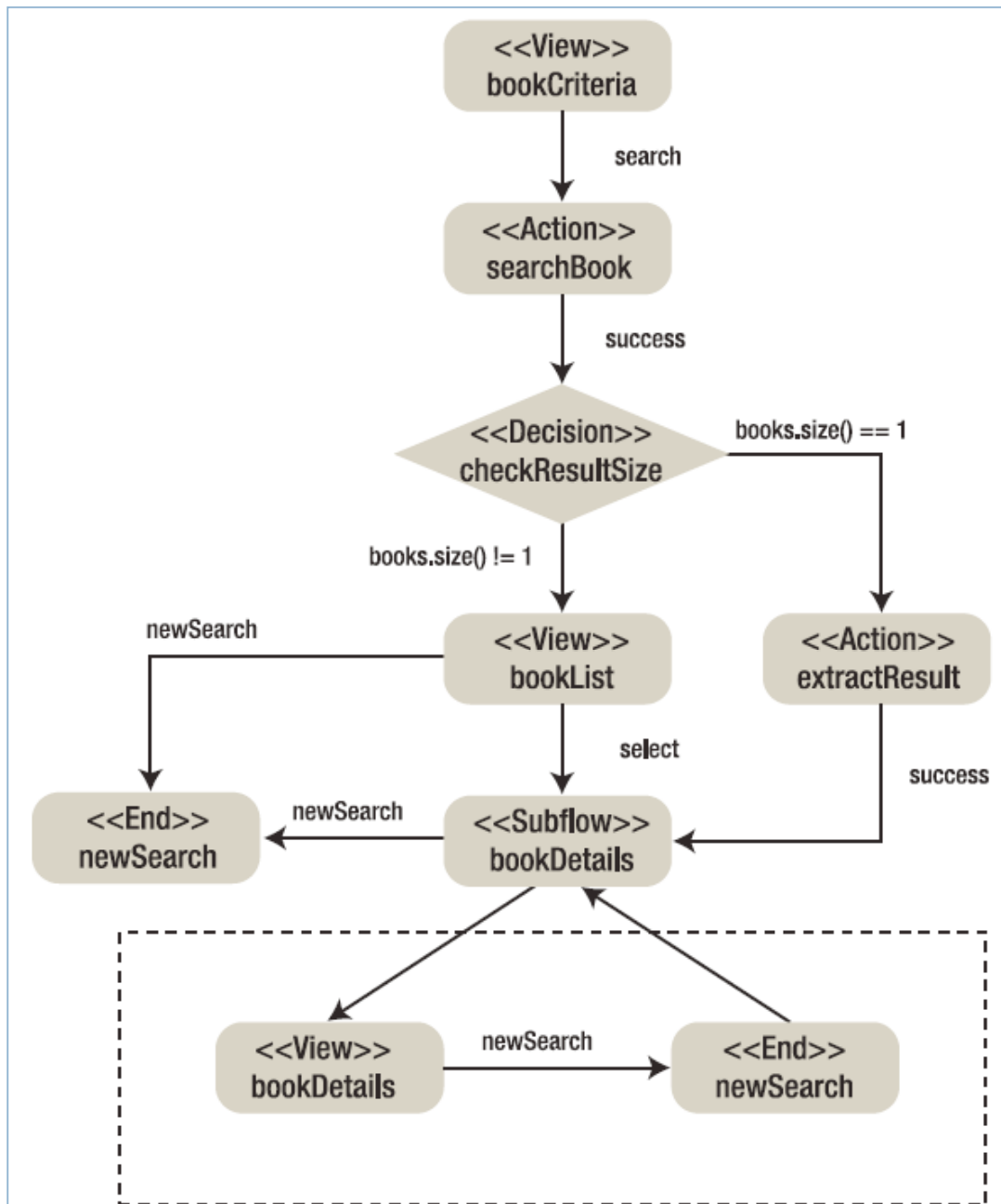


Figure 15-8. The flow diagram for the book search flow with a subflow