covers Spring 3.0

Spring IN ACTION

THIRD EDITION

Craig Walls

MANNING

Table of Contents

8 Working with Spring Web Flow

This chapter covers

- Creating conversational web applications
- Defining flow states and actions
- Securing web flows

One of the strangely wonderful things about the internet is that it's so easy to get lost. There are so many things to see and read. The hyperlink is at the core of the internet's power. But at the same time it's no wonder they call it *the web*. Just like webs built by spiders, it traps anyone who happens to crawl across it.

I'll confess: one reason why it took me so long to write this book is because I once got lost in an endless path of Wikipedia links.

There are times when a web application must take control of a web surfer's voyage, leading the user step by step through the application. The quintessential example of such an application is the checkout process on an e-commerce site. Starting with the shopping cart, the application leads you through a process of entering shipping details, billing information, and ultimately an order confirmation.

Spring Web Flow is a web framework that enables development of elements following a prescribed flow. In this chapter, we're going to explore Spring Web Flow and see how it fits into the Spring web framework landscape.

Chapter 8. Working with Spring Web Flow. Spring in Action, Third Edition, ISBN: 9781935182351

It's possible to write a flowed application with any web framework. I've even seen a Struts application that had a certain flow built into it. But without a way to separate the flow from the implementation, you'll find that the definition of the flow is scattered across the various elements that make up the flow. There's no one place to go to fully understand the flow.

Spring Web Flow is an extension to Spring MVC that enables development of flow-based web applications. It does this by separating the definition of an application's flow from the classes and views that implement the flow's behavior.

As we get to know Spring Web Flow, we're going to take a break from the Spitter example and work on a new web application for taking pizza orders. We'll use Spring Web Flow to define the order process.

The first step to working with Spring Web Flow is to install it within your project. Let's start there.

8.1 Installing Spring Web Flow

Although Spring Web Flow is a subproject of the Spring Framework, it isn't part of the Spring Framework proper. Therefore, before we can get started building flow-based applications, we'll need to add Spring Web Flow to our project's classpath.

You can download Spring Web Flow from the project's website (http://www.springframework.org/webflow). Be sure to get the latest version (as I write this, that's version 2.2.1). Once you've downloaded and unzipped the distribution zip file, you'll find the following Spring Web Flow JAR files in the dist directory:

- org.springframework.binding-2.2.1.RELEASE.jar
- org.springframework.faces-2.2.1.RELEASE.jar
- org.springframework.js-2.2.1.RELEASE.jar
- org.springframework.js.resources-2.2.1.RELEASE.jar
- org.springframework.webflow-2.2.1.RELEASE.jar

For our example, we'll only need the *binding* and *webflow* JAR files. The others are for using Spring Web Flow with JSF and JavaScript.

8.1.1 Configuring Web Flow in Spring

Spring Web Flow is built upon a foundation of Spring MVC. That means that all requests to a flow first go through Spring MVC's `DispatcherServlet`. From there, a handful of special beans in the Spring application context must be configured to handle the flow request and execute the flow.

Several of the web flow beans are declared using elements from Spring Web Flow's Spring configuration XML namespace. Therefore, we'll need to add the namespace declaration to the context definition XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:flow="http://www.springframework.org/schema/webflow-config"
```

```
xsi:schemaLocation="http://www.springframework.org/schema/webflow-config
  http://www.springframework.org/schema/webflow-config/
  ➥spring-webflow-config-2.0.xsd
  http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
```

With the namespace declaration in place, we're now ready to start wiring up web flow beans, starting with the flow executor.

WIRING A FLOW EXECUTOR

As its name implies, the *flow executor* drives the execution of a flow. When a user enters a flow, the flow executor creates and launches an instance of the flow execution for that user. When the flow pauses (such as when a view is presented to the user), the flow executor also resumes the flow once the user has taken some action.

The `<flow:flow-executor>` element creates a flow executor in Spring:

```
<flow:flow-executor id="flowExecutor"
                    flow-registry="flowRegistry" />
```

Although the flow executor is responsible for creating and executing flows, it's not responsible for loading flow definitions. That responsibility falls to a flow registry, which we'll create next. Here, the flow registry is referred to by its ID: `flowRegistry`.[1]

CONFIGURING A FLOW REGISTRY

A *flow registry*'s job is to load flow definitions and make them available to the flow executor. We can configure a flow registry in the Spring configuration with the `<flow:flow-registry>` element like this:

```
<flow:flow-registry id="flowRegistry"
        base-path="/WEB-INF/flows">
   <flow:flow-location-pattern value="*-flow.xml" />
</flow:flow-registry>
```

As declared here, the flow registry will look for flow definitions under the /WEB-INF/flows directory, as specified in the `base-path` attribute. Per the `<flow:flow-location-pattern>` element, any XML file whose name ends with *-flow.xml* will be considered a flow definition.

All flows are referred to by their IDs. Using the `<flow:flow-location-pattern>` as we have, the flow ID will be the directory path relative to the `base-path`—or the part of the path represented with the double asterisk. Figure 8.1 shows how the flow ID is calculated in this scenario.

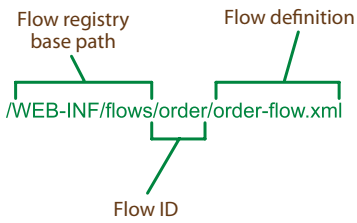

Figure 8.1 When using a flow location pattern, the path to the flow definition file relative to the base path will be used as the flow's ID.

[1] The `flow-registry` attribute is explicitly set here, but it wasn't necessary to do so. If it's not set, then it defaults to `flowRegistry`.

Alternatively, you could leave the `base-path` attribute off and explicitly identify the flow definition file's location:

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

Here, the `<flow:flow-location>` element is used instead of `<flow:flow-location-pattern>`. The `path` attribute directly points at the /WEB-INF/flows/springpizza.xml file as the flow definition. When configured this way, the flow's ID is derived from the base name of the flow definition file; *springpizza* in this case.

If you'd like to be even more explicit about the flow's ID, then you can set it with the `id` attribute of the `<flow:flow-location>` element. For example, to specify `pizza` as the flow's ID, configure the `<flow:flow-location>` like this:

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

HANDLING FLOW REQUESTS

As we saw in the previous chapter, `DispatcherServlet` typically dispatches requests to controllers. But for flows, we'll need a `FlowHandlerMapping` to help `DispatcherServlet` know that it should send flow requests to Spring Web Flow. The `FlowHandlerMapping` is configured in the Spring application context like this:

```
<bean class="org.springframework.webflow.mvc.servlet.FlowHandlerMapping">
  <property name="flowRegistry" ref="flowRegistry" />
</bean>
```

As you can see, the `FlowHandlerMapping` is wired with a reference to the flow registry so it knows when a request's URL maps to a flow. For example, if we have a flow whose ID is `pizza`, then `FlowHandlerMapping` will know to map a request to that flow if the request's URL pattern (relative to the application context path) is `/pizza`.

Whereas the `FlowHandlerMapping`'s job is to direct flow requests to Spring Web Flow, it's the job of a `FlowHandlerAdapter` to answer that call. A `FlowHandlerAdapter` is equivalent to a Spring MVC controller in that it handles requests coming in for a flow and processes those requests. The `FlowHandlerAdapter` is wired as a Spring bean like this:

```
<bean class="org.springframework.webflow.mvc.servlet.FlowHandlerAdapter">
  <property name="flowExecutor" ref="flowExecutor" />
</bean>
```

This handler adapter is the bridge between `DispatcherServlet` and Spring Web Flow. It handles flow requests and manipulates the flow based on those requests. Here, it's wired with a reference to the flow executor to execute the flows that it handles requests for.

We've configured all of the beans and components that are needed for Spring Web Flow to work. What's left is to actually define a flow. We'll do that soon enough. But first, let's get to know the elements that are put together to make up a flow.

8.2 The components of a flow

In Spring Web Flow, a flow is defined by three primary elements: states, transitions, and flow data.

States are points in a flow where something happens. If you imagine a flow as being like a road trip, then states are the towns, truck stops, and scenic stops along the way. Instead of picking up a bag of Doritos and a Diet Coke, a state in a flow is where some logic is performed, some decision is made, or some page is presented to the user.

If flow states are like the points on a map where you might stop during a road trip, then *transitions* are the roads that connect those points. In a flow, you get from one state to another by way of a transition.

As you travel from town to town, you may pick up some souvenirs, memories, and empty snack bags along the way. Similarly, as a flow progresses, it collects some data: the current condition of the flow. I'm tempted to refer to it as the state of the flow, but the word *state* already has another meaning when talking about flows.

Let's take a closer look at how these three elements are defined in Spring Web Flow.

8.2.1 States

Spring Web Flow defines five different kinds of state, as shown in table 8.1.

The selection of states provided by Spring Web Flow makes it possible to construct virtually any arrangement of functionality into a conversational web application. Though not all flows will require all of the states described in table 8.1, you'll probably end up using most of them at one time or another.

Table 8.1 Spring Web Flow's selections of states

State type	What it's for
Action	Action states are where the logic of a flow takes place.
Decision	Decision states branch the flow in two directions, routing the flow based on the outcome of evaluation flow data.
End	The end state is the last stop for a flow. Once a flow has reached its end state, the flow is terminated.
Subflow	A subflow state starts a new flow within the context of a flow that is already underway.
View	A view state pauses the flow and invites the user to participate in the flow.

In a moment we'll see how to piece these different kinds of states together to form a complete flow. But first, let's get to know how each of these flow elements are manifested in a Spring Web Flow definition.

VIEW STATES

View states are used to display information to the user and to offer the user an opportunity to play an active role in the flow. The actual view implementation could be any of the views supported by Spring MVC, but is often implemented in JSP.

Within the flow definition XML file, the `<view-state>` element is used to define a view state:

```
<view-state id="welcome" />
```

In this simple example, the `id` attribute serves a dual purpose. It identifies the state within the flow. Also, because no view has been specified otherwise, it specifies `welcome` as the logical name of the view to be rendered when the flow reaches this state.

If you'd rather explicitly identify another view name, then you can do so with the `view` attribute:

```
<view-state id="welcome" view="greeting" />
```

If a flow presents a form to the user, you may want to specify the object to which the form will be bound. To do that, set the `model` attribute:

```
<view-state id="takePayment" model="flowScope.paymentDetails"/>
```

Here we've specified that the form in the `takePayment` view will be bound to the flow-scoped `paymentDetails` object. (We'll talk more about flow scopes and data in a moment.)

ACTION STATES

Whereas view states involve the users of the application in the flow, action states are where the application itself goes to work. Action states typically invoke some method on a Spring-managed bean and then transition to another state depending on the outcome of the method call.

In the flow definition XML, action states are expressed with the `<action-state>` element. Here's an example:

```
<action-state id="saveOrder">
  <evaluate expression="pizzaFlowActions.saveOrder(order)" />
  <transition to="thankYou" />
</action-state>
```

Although it's not strictly required, `<action-state>` elements usually have an `<evaluate>` element as a child. The `<evaluate>` element gives an action state something to do. The `expression` attribute is given an expression that's evaluated when the state is entered. In this case, `expression` is given a SpEL[2] expression which

[2] Starting with version 2.1.0, Spring Web Flow uses the Spring Expression Language, but can optionally use OGNL or the Unified EL if you'd prefer.

indicates that the `saveOrder()` method should be called on a bean whose ID is `pizzaFlowActions`.

DECISION STATES

It's possible for a flow to be purely linear, stepping from one state to another without taking any alternate routes. But more often a flow branches at one point or another, depending on the flow's current circumstances.

Decision states enable a binary branch in a flow execution. A decision state will evaluate a Boolean expression and will take one of two transitions, depending on whether the expression evaluates to `true` or `false`. Within the XML flow definition, decision states are defined by the `<decision-state>` element. A typical example of a decision state might look like this:

```
<decision-state id="checkDeliveryArea">
  <if test="pizzaFlowActions.checkDeliveryArea(customer.zipCode)"
      then="addCustomer"
      else="deliveryWarning" />
</decision-state>
```

As you can see, the `<decision-state>` element doesn't work alone. The `<if>` element is the heart of a decision state. It's where the expression is evaluated. If the expression evaluates to `true`, then the flow will transition to the state identified by the `then` attribute. But if it's `false`, then the flow will transition to the state named in the `else` attribute.

SUBFLOW STATES

You probably wouldn't write all of your application's logic in a single method. Instead, you'd probably break it up into multiple classes, methods, and other structures.

In the same way, it's a good idea to break flows down into discrete parts. The `<subflow-state>` element lets you call another flow from within an executing flow. It's analogous to calling a method from within another method.

A `<subflow-state>` might be declared as follows:

```
<subflow-state id="order" subflow="pizza/order">
  <input name="order" value="order"/>
  <transition on="orderCreated" to="payment" />
</subflow-state>
```

Here, the `<input>` element is used to pass the order object as input to the subflow. And, if the subflow ends with an `<end-state>` whose ID is `orderCreated`, then the flow will transition to the state whose ID is `payment`.

But I'm getting ahead of myself. We haven't talked about the `<end-state>` element or transitions yet. But we'll look at transitions soon in section 8.2.2. As for end states, that's what we'll look at next.

END STATES

Eventually all flows must come to an end. And that's what they'll do when they transition to an end state. The `<end-state>` element designates the end of a flow and typically appears like this:

```
<end-state id="customerReady" />
```

When the flow reaches an `<end-state>`, the flow ends. What happens next depends on a few factors:

- If the flow that's ending is a subflow, then the calling flow will proceed from the `<subflow-state>`. The `<end-state>`'s ID will be used as an event to trigger the transition away from the `<subflow-state>`.
- If the `<end-state>` has its `view` attribute set, the specified view will be rendered. The view may be a flow-relative path to a view template, prefixed with `externalRedirect:` to redirect to some page external to the flow, or prefixed with `flowRedirect:` to redirect to another flow.
- If the ending flow isn't a subflow and no `view` is specified, then the flow simply ends. The browser ends up landing on the flow's base URL, and with no current flow active, a new instance of the flow begins.

It's important to realize that a flow may have more than one end state. Since the end state's ID determines the event fired from a subflow, you may want to end the flow through multiple end states to trigger different events in the calling flow. Even in flows that aren't subflows, there may be several landing pages that follow the completion of a flow, depending on the course that the flow took.

Now that we've looked at the various kinds of states in a flow, we should take a moment to look at how the flow travels between states. Let's see how to pave some roads in a flow by defining transitions.

8.2.2 Transitions

As I've already mentioned, transitions connect the states within a flow. Every state in a flow, with the exception of end states, should have at least one transition so that the flow will know where to go once that state has completed. A state may have multiple transitions, each one representing a different path that could be taken upon completion of the state.

A transition is defined by the `<transition>` element, a child of the various state elements (`<action-state>`, `<view-state>`, and `<subflow-state>`). In its simplest form, the `<transition>` element identifies the next state in the flow:

```
<transition to="customerReady" />
```

The `to` attribute is used to specify the next state in the flow. When `<transition>` is declared with only a `to` attribute, the transition is the default transition for that state and will be taken if no other transitions are applicable.

More commonly transitions are defined to take place upon some event being fired. In a view state, the event is usually some action taken by the user. In an action state, the event is the result of evaluating an expression. In the case of a subflow state, the event is determined by the ID of the subflow's end state. In any event (no pun intended), you can specify the event to trigger the transition by specifying it in the `on` attribute:

```
<transition on="phoneEntered" to="lookupCustomer"/>
```

In this example, the flow will transition to the state whose ID is `lookupCustomer` if a `phoneEntered` event is fired.

The flow can also transition to another state in response to some exception being thrown. For example, if a customer record can't be found, you may want the flow to transition to a view state that presents a registration form. The following snippet shows that kind of transition:

```
<transition
    on-exception=
           "com.springinaction.pizza.service.CustomerNotFoundException"
    to="registrationForm" />
```

The `on-exception` attribute is much like the `on` attribute, except that it specifies an exception to transition on instead of an event. In this case, a `CustomerNotFoundException` will cause the flow to transition to the `registrationForm` state.

GLOBAL TRANSITIONS

After you've created a flow, you may find that there are several states that share some common transitions. For example, I wouldn't be surprised to find the following `<transition>` sprinkled all over a flow:

```
<transition on="cancel" to="endState" />
```

Rather than repeat common transitions in multiple states, you can define them as global transitions by placing the `<transition>` element as a child of a `<global-transitions>` element. For example:

```
<global-transitions>
  <transition on="cancel" to="endState" />
</global-transitions>
```

With this global transition in place, all states within the flow will have an implicit `cancel` transition.

We've talked about states and transitions. Before we get busy writing flows, let's look at flow data, the remaining member of the web flow triad.

8.2.3 Flow data

If you've ever played one of those old text-based adventure games, you know that as you move from location to location, you occasionally find objects laying around that you can pick up and carry with you. Sometimes you need an object right away. Other times, you may carry an object around through the entire game not knowing what it's for—until you get to that final puzzle and find that it's useful after all.

In many ways, flows are like those adventure games. As the flow progresses from one state to another, it picks up some data. Sometimes that data is only needed for a little while (maybe just long enough to display a page to the user). Other times, that data is carried around through the entire flow and is ultimately used as the flow completes.

DECLARING VARIABLES

Flow data is stored away in variables that can be referenced at various points in the flow. It can be created and accumulated in several ways. The simplest way to create a variable in a flow is by using the `<var>` element:

```
<var name="customer" class="com.springinaction.pizza.domain.Customer"/>
```

Here, a new instance of a `Customer` object is created and placed into the variable whose name is `customer`. This variable will be available to all states in a flow.

As part of an action state or upon entry to a view state, you may also create variables using the `<evaluate>` element. For example:

```
<evaluate result="viewScope.toppingsList"
   expression="T(com.springinaction.pizza.domain.Topping).asList()" />
```

In this case, the `<evaluate>` element evaluates an expression (a SpEL expression) and places the result in a variable named `toppingsList` that's view-scoped. (We'll talk more about scopes in a moment.)

Similarly, the `<set>` element can set a variable's value:

```
<set name="flowScope.pizza"
   value="new com.springinaction.pizza.domain.Pizza()" />
```

The `<set>` element works much the same as the `<evaluate>` element, setting a variable to the resulting value from an evaluated expression. Here, we're setting a flow-scoped `pizza` variable to a new instance of a `Pizza` object.

You'll see more specifics on how these elements are used in an actual flow when we get to section 8.3 and start building a real working web flow. But first, let's see what it means for a variable to be flow-scoped, view-scoped, or use some other scope.

SCOPING FLOW DATA

The data carried about in a flow will have varying lifespans and visibility, depending on the scope of the variable it's kept in. Spring Web Flow defines five scopes, as described in table 8.2.

Table 8.2 Spring Web Flow's selections of states

Scope	Lifespan and visibility
Conversation	Created when a top-level flow starts and destroyed when the top-level flow ends. Shared by a top-level flow and all of its subflows.
Flow	Created when a flow starts and destroyed when the flow ends. Only visible within the flow it was created by.
Request	Created when a request is made into a flow and destroyed when the flow returns.
Flash	Created when a flow starts and destroyed when the flow ends. It's also cleared out after a view state renders.
View	Created when a view state is entered and destroyed when the state exits. Visible only within the view state.

When declaring a variable using the `<var>` element, the variable is always flow-scoped within the flow defining the variable. When using `<set>` or `<evaluate>`, the scope is specified as a prefix for the `name` or `result` attribute. For example, to assign a value to a flow-scoped variable named `theAnswer`:

```
<set name="flowScope.theAnswer" value="42"/>
```

Now that we've seen all of the raw materials of a web flow, it's time to piece them together into a full-blown, fully functional web flow. As we do, keep your eyes peeled for examples of how to store data away in scoped variables.

8.3 Putting it all together: the pizza flow

As I mentioned earlier in this chapter, we're taking a break from the Spitter application. Instead, we've been asked to build out an online pizza ordering application where hungry web visitors can order their favorite Italian pie.[3]

As it turns out, the process of ordering a pizza can be defined nicely in a flow. We'll start by building a high-level flow that defines the overall process of ordering a pizza. Then we'll break that flow down into subflows that define the details at a lower level.

8.3.1 Defining the base flow

A new pizza chain, Spizza,[4] has decided to relieve the load on their stores' telephones by allowing customers to place orders online. When the customer visits the Spizza website, they'll identify themselves, select one or more pizzas to add to their order, provide payment information, and then submit the order and wait for the pizza to arrive, hot and fresh. Figure 8.2 illustrates this flow.

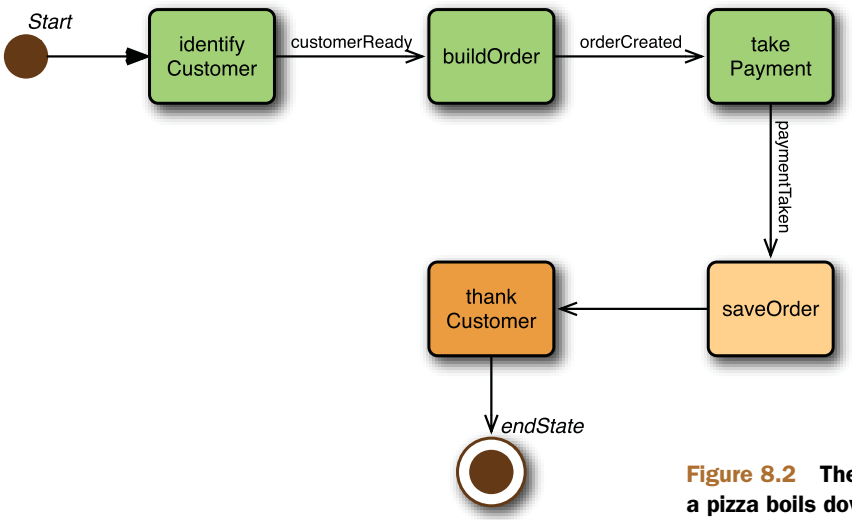

Figure 8.2 The process of ordering a pizza boils down to a simple flow.

[3] In truth, I couldn't think of any good way of working a flow into the Spitter application. So rather than shoehorn a Spring Web Flow example into Spitter, we're going to go with the pizza example.

[4] Yes, I know...there's a real Spizza pizza place in Singapore. This isn't that one.

The boxes in the diagram represent states and the arrows represent transitions. As you can see, the overall pizza flow is simple and linear. It should be easy to express this flow in Spring Web Flow. The only thing that makes it interesting is that the first three states can be more involved than suggested by a simple box.

The following shows the high-level pizza order flow as defined using Spring Web Flow's XML-based flow definition.

Listing 8.1 The pizza order flow, defined as a Spring Web Flow

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

  <var name="order"
       class="com.springinaction.pizza.domain.Order"/>

  <subflow-
     state id="identifyCustomer" subflow="pizza/customer">
    <output name="customer" value="order.customer"/>
    <transition on="customerReady" to="buildOrder" />
  </subflow-state>

  <subflow-state id="buildOrder" subflow="pizza/order">
    <input name="order" value="order"/>
    <transition on="orderCreated" to="takePayment" />
  </subflow-state>

  <subflow-state id="takePayment" subflow="pizza/payment">
    <input name="order" value="order"/>
    <transition on="paymentTaken" to="saveOrder"/>
  </subflow-state>

  <action-state id="saveOrder">
    <evaluate expression="pizzaFlowActions.saveOrder(order)" />
    <transition to="thankCustomer" />
  </action-state>

  <view-state id="thankCustomer">
    <transition to="endState" />
  </view-state>

  <end-state id="endState" />

  <global-transitions>
    <transition on="cancel" to="endState" />
  </global-transitions>
</flow>
```

Call customer subflow

Call order subflow

Call payment subflow

Save order

Thank customer

Global cancel transition

The first thing you see in the flow definition is the declaration of the `order` variable. Each time the flow starts, a new instance of `Order` is created. The `Order` class, as shown next, has properties for carrying all of the information about an order, including the customer information, the list of pizzas ordered, and the payment details.

Listing 8.2 An `Order` carries all of the details pertaining to a pizza order

```
package com.springinaction.pizza.domain;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

public class Order implements Serializable {
   private static final long serialVersionUID = 1L;

   private Customer customer;
   private List<Pizza> pizzas;
   private Payment payment;

   public Order() {
      pizzas = new ArrayList<Pizza>();
      customer = new Customer();
   }

   public Customer getCustomer() {
      return customer;
   }

   public void setCustomer(Customer customer) {
      this.customer = customer;
   }

   public List<Pizza> getPizzas() {
      return pizzas;
   }

   public void setPizzas(List<Pizza> pizzas) {
      this.pizzas = pizzas;
   }

   public void addPizza(Pizza pizza) {
      pizzas.add(pizza);
   }

   public float getTotal() {
      return 0.0f;
   }

   public Payment getPayment() {
      return payment;
   }

   public void setPayment(Payment payment) {
      this.payment = payment;
   }
}
```

The main portion of the flow definition is made up of the flow states. By default, the first state in the flow definition file is also the first state that will be visited in the flow. In this case, that's the `identifyCustomer` state (a subflow state). But if you'd like, you can explicitly identify any state as the starting state by setting the `start-state` attribute in the `<flow>` element:

```
<?xml version="1.0" encoding="UTF-8"?>
<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"
  start-state="identifyCustomer">
...
</flow>
```

Identifying a customer, building a pizza order, and taking payment are activities that are too complex to be crammed into a single state. That's why we'll define them later in more detail as flows in their own right. But for the purposes of the high-level pizza flow, these activities are expressed with the `<subflow-state>` element.

The `order` flow variable will be populated by the first three states and then saved in the fourth state. The `identifyCustomer` subflow state uses the `<output>` element to populate the `order`'s `customer` property, setting it to the output received from calling the customer subflow. The `buildOrder` and `takePayment` states take a different approach, using `<input>` to pass the `order` flow variable as input so that those subflows can populate the `order` internally.

After the order has been given a customer, some pizzas, and payment details, it's time to save it. The `saveOrder` state is an action state that handles that task. It uses `<evaluate>` to make a call to the `saveOrder()` method on the bean whose ID is `pizzaFlowActions`, passing in the order to be saved. When it's finished saving the order, it transitions to `thankCustomer`.

The `thankCustomer` state is a simple view state, backed by the JSP file at /WEB-INF/flows/pizza/thankCustomer.jsp, as shown next.

Listing 8.3 A JSP view that thanks the customer for their order

```
<html xmlns:jsp="http://java.sun.com/JSP/Page">
  <jsp:output omit-xml-declaration="yes"/>
  <jsp:directive.page contentType="text/html;charset=UTF-8" />

  <head><title>Spizza</title></head>

  <body>
    <h2>Thank you for your order!</h2>

    <![CDATA[
    <a href='${flowExecutionUrl}&_eventId=finished'>Finish</a>
   ]]>
    </body>
</html>
```

Fire finished event

The "thank you" page thanks the customer for their order and gives a link for the customer to finish the flow. This link is the most interesting thing on the page because it shows one way that a user can interact with the flow.

Spring Web Flow provides a `flowExecutionUrl` variable, which contains the URL for the flow, for use in the view. The Finish link attaches an `_eventId` parameter to the URL to fire a `finished` event back to the web flow. That event sends the flow to the end state.

At the end state, the flow ends. Since there are no further details on where to go after the flow ends, the flow will start over again at the `identifyCustomer` state, ready to take another pizza order.

That covers the general flow for ordering a pizza. But there's more to the flow than what we see in listing 8.1. We still need to define the subflows for the `identifyCustomer`, `buildOrder`, and `takePayment` states. Let's build those flows next, starting with the one that identifies the customer.

8.3.2 Collecting customer information

If you've ordered a pizza before, you probably know the drill. The first thing they ask for is your phone number. Aside from giving them a way to call you if the delivery driver can't find your house, the phone number also serves as your identification to the pizza shop. If you're a repeat customer, they can use that phone number to look up your address so that they'll know where to deliver your order.

For a new customer, the phone number won't turn up any results. So the next information that they'll ask for is your address. At this point the pizzeria knows who you are and where to deliver your pizzas. But before they ask you what kind of pizza you want, they need to check to make sure that your address falls within their delivery area. If not, then you'll have to come in and pick up the pizza yourself.

The initial question and answer period that begins every pizza order can be illustrated with the flow diagram in figure 8.3.

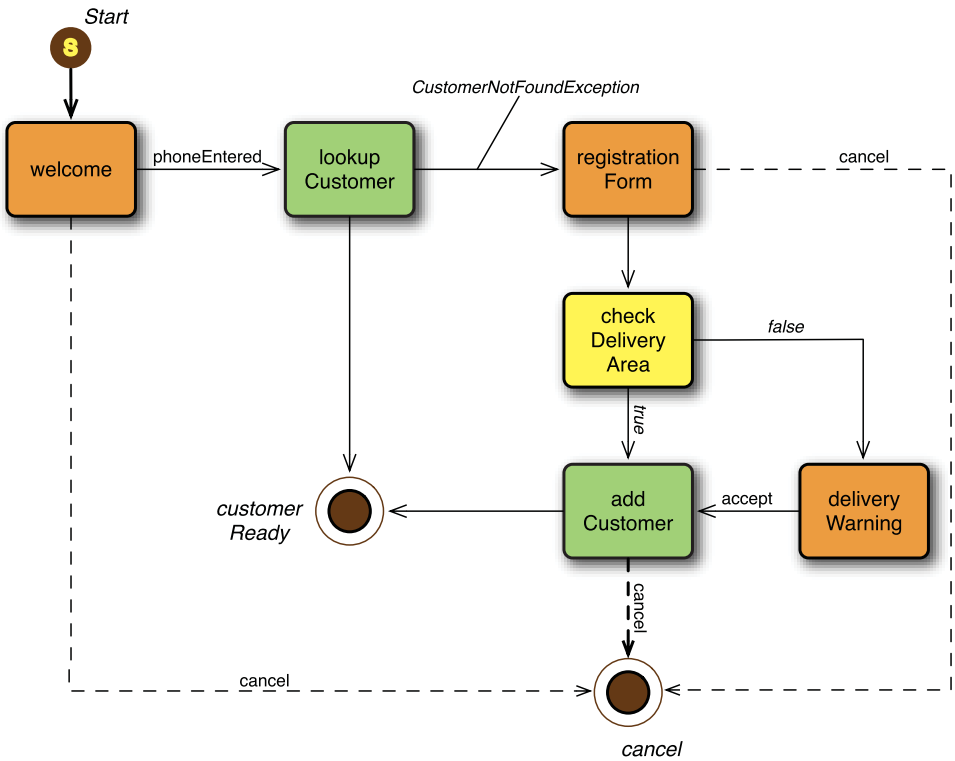

Figure 8.3 The flow for identifying a customer has a few more twists than the pizza flow.

This flow is more interesting than the top-level pizza flow. This flow isn't linear and branches in a couple of places depending on different conditions. For example, after looking up the customer, the flow could either end (if the customer was found) or transition to a registration form (if the customer was not found). Also, at the `checkDeliveryArea` state, the customer may or may not be warned that their address isn't in the delivery area.

The following shows the flow definition for identifying the customer.

Listing 8.4 Identifying the hungry pizza customer with a web flow

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

  <var name="customer" class="com.springinaction.pizza.domain.Customer"/>

  <view-state id="welcome">                                    <-- Welcome customer
    <transition on="phoneEntered" to="lookupCustomer"/>
  </view-state>

  <action-state id="lookupCustomer">                           <-- Look up customer
    <evaluate result="customer" expression=
        "pizzaFlowActions.lookupCustomer(requestParameters.phoneNumber)" />
    <transition to="registrationForm" on-exception=
        "com.springinaction.pizza.service.CustomerNotFoundException" />
    <transition to="customerReady" />
  </action-state>

  <view-state id="registrationForm" model="customer">          <-- Register new customer
    <on-entry>
      <evaluate expression=
          "customer.phoneNumber = requestParameters.phoneNumber" />
    </on-entry>
    <transition on="submit" to="checkDeliveryArea" />
  </view-state>

  <decision-state id="checkDeliveryArea">                      <-- Check delivery area
    <if test="pizzaFlowActions.checkDeliveryArea(customer.zipCode)"
        then="addCustomer"
        else="deliveryWarning"/>
  </decision-state>

  <view-state id="deliveryWarning">                            <-- Show delivery warning
    <transition on="accept" to="addCustomer" />
  </view-state>

  <action-state id="addCustomer">                              <-- Add customer
    <evaluate expression="pizzaFlowActions.addCustomer(customer)" />
    <transition to="customerReady" />
  </action-state>

  <end-state id="cancel" />
  <end-state id="customerReady">
    <output name="customer" />
  </end-state>
```

```
  <global-transitions>
    <transition on="cancel" to="cancel" />
  </global-transitions>
</flow>
```

This flow introduces a few new tricks, including our first use of the `<decision-state>` element. Also, since it's a subflow of the `pizza` flow, it expects to receive an `Order` object as input.

As before, let's break down this flow definition state by state, starting with the `welcome` state.

ASKING FOR A PHONE NUMBER

The `welcome` state is a fairly straightforward view state that welcomes the customer to the Spizza website and asks them to enter their phone number. The state itself isn't particularly interesting. It has two transitions: one that directs the flow to the `lookupCustomer` state if a `phoneEntered` event is fired from the view, and another `cancel` transition, defined as a global transition, that reacts to a `cancel` event.

Where the `welcome` state gets interesting is in the view itself. The welcome view is defined in /WEB-INF/flows/pizza/customer/welcome.jspx, as shown next.

Listing 8.5 Welcoming the customer and asking for their phone number

```
<html xmlns:jsp="http://java.sun.com/JSP/Page"
      xmlns:form="http://www.springframework.org/tags/form">
  <jsp:output omit-xml-declaration="yes"/>
  <jsp:directive.page contentType="text/html;charset=UTF-8" />

  <head><title>Spizza</title></head>

  <body>
    <h2>Welcome to Spizza!!!</h2>

    <form:form>
      <input type="hidden" name="_flowExecutionKey"
             value="${flowExecutionKey}"/>

      <input type="text" name="phoneNumber"/><br/>

      <input type="submit" name="_eventId_phoneEntered"
             value="Lookup Customer" />
    </form:form>
  </body>
</html>
```

Flow execution key

Fire phoneEntered event

This simple form prompts the user to enter their phone number. But the form has two special ingredients that enable it to drive the flow.

First note the hidden `_flowExecutionKey` field. When a view state is entered, the flow pauses and waits for the user to take some action. The flow execution key is given to the view as a sort of "claim ticket" for the flow. When the user submits the form, the flow execution key is sent along with it in the `_flowExecutionKey` field and the flow resumes where it left off.

Also pay special attention to the submit button's name. The *_eventId_* portion of the button's name is a clue to Spring Web Flow that what follows is an event that

should be fired. When the form is submitted by clicking that button, a `phoneEntered` event will be fired, triggering a transition to `lookupCustomer`.

LOOKING UP THE CUSTOMER

After the welcome form has been submitted, the customer's phone number is among the request parameters and is ready to be used to look up a customer. The `lookupCustomer` state's `<evaluate>` element is where that happens. It pulls the phone number off of the request parameters and passes it to the `lookupCustomer()` method on the `pizzaFlowActions` bean.

The implementation of `lookupCustomer()` is not important right now. It's sufficient to know that it will either return a `Customer` object or throw a `CustomerNotFoundException`.

In the former case, the `Customer` object is assigned to the `customer` variable (per the `result` attribute) and the default transition takes the flow to the `customerReady` state. But if the customer can't be found, then a `CustomerNotFoundException` will be thrown and the flow will transition to the `registrationForm` state.

REGISTERING A NEW CUSTOMER

The `registrationForm` state is where the user is asked for their delivery address. Like other view states we've seen, it'll render a JSP view. The JSP file is shown next.

Listing 8.6 Registering a new customer

```
<html xmlns:c="http://java.sun.com/jsp/jstl/core"
     xmlns:jsp="http://java.sun.com/JSP/Page"
     xmlns:spring="http://www.springframework.org/tags"
     xmlns:form="http://www.springframework.org/tags/form">

  <jsp:output omit-xml-declaration="yes"/>
  <jsp:directive.page contentType="text/html;charset=UTF-8" />

  <head><title>Spizza</title></head>

  <body>
    <h2>Customer Registration</h2>

    <form:form commandName="customer">
      <input type="hidden" name="_flowExecutionKey"
             value="${flowExecutionKey}"/>
      <b>Phone number: </b><form:input path="phoneNumber"/><br/>
      <b>Name: </b><form:input path="name"/><br/>
      <b>Address: </b><form:input path="address"/><br/>
      <b>City: </b><form:input path="city"/><br/>
      <b>State: </b><form:input path="state"/><br/>
      <b>Zip Code: </b><form:input path="zipCode"/><br/>
      <input type="submit" name="_eventId_submit"
             value="Submit" />
      <input type="submit" name="_eventId_cancel"
             value="Cancel" />
    </form:form>
    </body>
</html>
```

This isn't the first form we've seen in our flow. The `welcome` view state also displayed a form to the customer. That form was simple and had only a single field. It was easy enough to pull that field's value from the request parameters. The registration form, on the other hand, is more involved.

Instead of dealing with the fields one at a time through the request parameters, it makes more sense to bind the form to a `Customer` object—let the framework do all of the hard work.

CHECKING THE DELIVERY AREA

After the customer has given their address, we need to be sure that they live within the delivery area. If Spizza can't deliver to them, then we should let them know and advise them that they'll need to come in and pick up the pizzas themselves.

To make that decision, we use a decision state. The `checkDeliveryArea` decision state has an `<if>` element that passes the customer's Zip code into the `checkDeliveryArea()` method on the `pizzaFlowActions` bean. That method will return a Boolean value: `true` if the customer is in the delivery area, `false` otherwise.

If the customer is in the delivery area, then the flow transitions to the `addCustomer` state. If not, then the customer is taken to the `deliveryWarning` view state. The view behind the `deliveryWarning` is /WEB-INF/flows/pizza/customer/deliveryWarning.jspx and is shown next.

Listing 8.7 Warning a customer that pizza can't be delivered to their address

```
<html xmlns:jsp="http://java.sun.com/JSP/Page">
  <jsp:output omit-xml-declaration="yes"/>
  <jsp:directive.page contentType="text/html;charset=UTF-8" />

  <head><title>Spizza</title></head>

  <body>
        <h2>Delivery Unavailable</h2>

        <p>The address is outside of our delivery area. You may
        still place the order, but you will need to pick it up
        yourself.</p>

        <![CDATA[
        <a href="${flowExecutionUrl}&_eventId=accept">
                                Continue, I'll pick up the order</a> |
        <a href="${flowExecutionUrl}&_eventId=cancel">Never mind</a>
        ]]>
  </body>
</html>
```

The key flow-related items in deliveryWarning.jspx are the two links that offer the customer a chance to continue with the order or to cancel. Using the same `flowExecurtionUrl` variable that we used in the `welcome` state, these links will trigger either an `accept` event or a `cancel` event in the flow. If an accept event is sent, then the flow will transition to the `addCustomer` state. Otherwise, the global cancel transition will be followed and the subflow will transition to the `cancel` end state.

We'll talk about the end states in a moment. First, let's take a quick look at the `addCustomer` state.

STORING THE CUSTOMER DATA

By the time the flow arrives at the `addCustomer` state, the customer has entered their address. For future reference, that address needs to be stored away (probably in a database). The `addCustomer` state has an `<evaluate>` element that calls the `addCustomer()` method on the `pizzaFlowActions` bean, passing in the `customer` flow variable.

Once the evaluation is complete, the default transition will be taken and the flow will transition to the end state whose ID is `customerReady`.

ENDING THE FLOW

Normally a flow's end state isn't that interesting. But in this flow, there's not just one end state, but two. When a subflow ends, it fires a flow event that's equivalent to its end state's ID. If the flow only has one end state, then it'll always fire the same event. But with two or more end states, a flow can influence the direction of the calling flow.

When the customer flow goes down any of the normal paths, it'll ultimately land on the end state whose ID is `customerReady`. When the calling pizza flow resumes, it'll receive a `customerReady` event, which will result in a transition to the `buildOrder` state.

Note that the `customerReady` end state includes an `<output>` element. This element is a flow's equivalent of Java's `return` statement. It passes back some data from a subflow to the calling flow. In this case, the `<output>` is returning the `customer` flow variable so that the `identifyCustomer` subflow state in the pizza flow can assign it to the order.

On the other hand, if a `cancel` event is triggered at any time during the customer flow, it'll exit the flow through the end state whose ID is `cancel`. That will trigger a `cancel` event in the pizza flow and result in a transition (via the global transition) to the pizza flow's end state.

8.3.3 Building an order

After the customer has been identified, the next step in the main flow is to figure out what kind of pizzas they want. The order subflow, as illustrated in figure 8.4, is where the user is prompted to create pizzas and add them to the order.

As you can see, the `showOrder` state is the centerpiece of the order subflow. It's the first state that the user sees upon entering the flow and it's the state that the user is sent to after adding a new pizza to the order. It displays the current state of the order and offers the user a chance to add another pizza to the order.

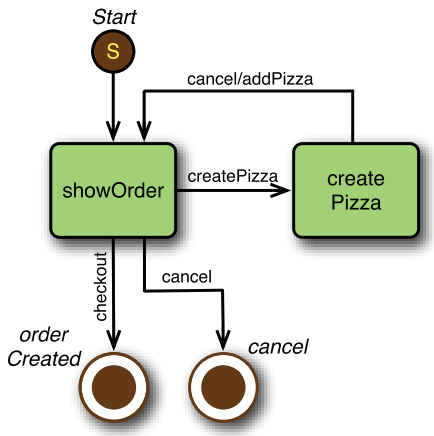

Figure 8.4 Pizzas are added via the order subflow.

Upon choosing to add a pizza to the order, the flow transitions to the `createPizza` state. This is another view state that gives the user a selection of pizza sizes and toppings to build a pizza with. From here the user may add a pizza or cancel. In either event the flow transitions back to the `showOrder` state.

From the `showOrder` state, the user may choose to either submit the order or cancel the order. Either choice will end the order subflow, but the main flow will go down different paths depending on which choice is made.

The following shows how the diagram translates into a Spring Web Flow definition.

Listing 8.8 The order subflow view states to display the order and to create a pizza

```
<?xml version="1.0" encoding="UTF-8"?>
<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="order" required="true" />

  <view-state id="showOrder">
    <transition on="createPizza" to="createPizza" />
    <transition on="checkout" to="orderCreated" />
    <transition on="cancel" to="cancel" />
  </view-state>

  <view-state id="createPizza" model="flowScope.pizza">
    <on-entry>
      <set name="flowScope.pizza"
          value="new com.springinaction.pizza.domain.Pizza()" />

      <evaluate result="viewScope.toppingsList"  expression=
                  "T(com.springinaction.pizza.domain.Topping).asList()" />
    </on-entry>
    <transition on="addPizza" to="showOrder">
      <evaluate expression="order.addPizza(flowScope.pizza)" />
    </transition>
    <transition on="cancel" to="showOrder" />
  </view-state>

  <end-state id="cancel" />
  <end-state id="orderCreated" />
</flow>
```

This subflow will actually operate on the `Order` object created in the main flow. Therefore, we need a way of passing the `Order` from the main flow to the subflow. As you'll recall from listing 8.1, we used the `<input>` element there to pass the `Order` into the flow. Here we're using it to accept that `Order` object. If you think of this subflow as being analogous to a method in Java, the `<input>` element used here is effectively defining the subflow's signature. This flow requires a single parameter called `order`.

Next we find the `showOrder` state, a basic view state with three different transitions: one for creating a pizza, one for submitting the order, and another to cancel the order.

The `createPizza` state is more interesting. Its view is a form that submits a new `Pizza` object to be added to the order. The `<on-entry>` element adds a new `Pizza` object to flow scope to be populated when the form is submitted. Note that the `model` of this view state references the same flow-scoped `Pizza` object. That `Pizza` object will be bound to the create pizza form, shown next.

Listing 8.9 Adding pizzas to an order with an HTML form bound to a flow-scoped object

```
<div xmlns:form="http://www.springframework.org/tags/form"
     xmlns:jsp="http://java.sun.com/JSP/Page">

  <jsp:output omit-xml-declaration="yes"/>
  <jsp:directive.page contentType="text/html;charset=UTF-8" />

    <h2>Create Pizza</h2>
    <form:form commandName="pizza">
      <input type="hidden" name="_flowExecutionKey"
          value="${flowExecutionKey}"/>

      <b>Size: </b><br/>
  <form:radiobutton path="size"
                    label="Small (12-inch)" value="SMALL"/><br/>
  <form:radiobutton path="size"
                    label="Medium (14-inch)" value="MEDIUM"/><br/>
  <form:radiobutton path="size"
                    label="Large (16-inch)" value="LARGE"/><br/>
  <form:radiobutton path="size"
                    label="Ginormous (20-inch)" value="GINORMOUS"/>
      <br/>
      <br/>

      <b>Toppings: </b><br/>
      <form:checkboxes path="toppings" items="${toppingsList}"
                       delimiter="&lt;br/&gt;"/><br/><br/>

      <input type="submit" class="button"
          name="_eventId_addPizza" value="Continue"/>
      <input type="submit" class="button"
          name="_eventId_cancel" value="Cancel"/>
    </form:form>
</div>
```

When the form is submitted via the Continue button, the size and topping selections will be bound to the `Pizza` object and the `addPizza` transition will be taken. The `<evaluate>` element associated with that transition indicates that the flow-scoped `Pizza` object should be passed in a call to the order's `addPizza()` method before transitioning to the `showOrder` state.

There are two ways to end the flow. The user can either click the Cancel button on the `showOrder` view or they can click the Checkout button. Either way, the flow transitions to an `<end-state>`. But the `id` of the end state chosen determines the event triggered on the way out of this flow, and ultimately determines the next step in the main flow. The main flow will either transition on `cancel` or will transition on `orderCreated`.

In the former case, the outer flow ends; in the latter case, it transitions to the `takePayment` subflow, which we'll look at next.

8.3.4 Taking payment

It's not common to get a free pizza, and the Spizza pizzeria wouldn't stay in business long if they let their customers order pizzas without providing some form of payment. As the pizza flow nears an end, the final subflow prompts the user to enter payment details. This simple flow is illustrated in figure 8.5.

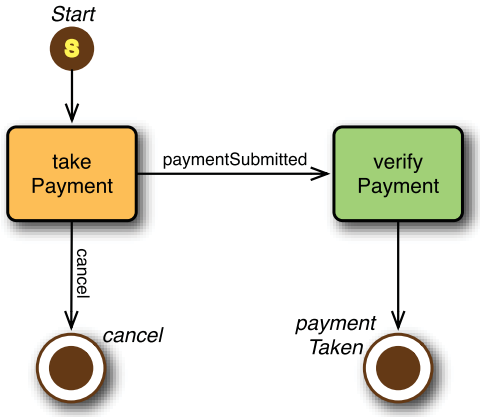

Figure 8.5 The final step in placing a pizza order is to take payment from the customer through the payment subflow.

Like the order subflow, the payment subflow also accepts an `Order` object as input using the `<input>` element.

As you can see, upon entering the payment subflow, the user arrives at the `takePayment` state. This is a view state where the user can indicate that they'll pay by credit card, check, or cash. Upon submitting their payment information, they're taken to the `verifyPayment` state, an action state that verifies that their payment information is acceptable.

The payment subflow is defined in XML as shown.

Listing 8.10 The payment subflow has one view state and one action state.

```
<?xml version="1.0" encoding="UTF-8"?>
<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="order" required="true"/>

  <view-state id="takePayment" model="flowScope.paymentDetails">
    <on-entry>
      <set name="flowScope.paymentDetails"
          value="new com.springinaction.pizza.domain.PaymentDetails()" />

      <evaluate result="viewScope.paymentTypeList" expression=
                "T(com.springinaction.pizza.domain.PaymentType).asList()" />
    </on-entry>
    <transition on="paymentSubmitted" to="verifyPayment" />
    <transition on="cancel" to="cancel" />
  </view-state>

  <action-state id="verifyPayment">
    <evaluate result="order.payment" expression=
        "pizzaFlowActions.verifyPayment(flowScope.paymentDetails)" />
    <transition to="paymentTaken" />
  </action-state>
```

```
  <end-state id="cancel" />
  <end-state id="paymentTaken" />
</flow>
```

As the flow enters the `takePayment` view state, the `<on-entry>` element sets up the payment form by first using a SpEL expression to create a new `PaymentDetails` instance in flow scope. This will effectively be the backing object for the form. It also sets the view-scoped `paymentTypeList` variable to a list containing the values of the `PaymentTypeenum` (shown in listing 8.11). Here, SpEL's `T()` operator is used to get the `PaymentType` class so that the static `toList()` method can be invoked.

Listing 8.11 The `PaymentType` enumeration defines customer's choices for payment.

```
package com.springinaction.pizza.domain;

import static org.apache.commons.lang.WordUtils.*;

import java.util.Arrays;
import java.util.List;

public enum PaymentType {
  CASH, CHECK, CREDIT_CARD;

  public static List<PaymentType> asList() {
    PaymentType[] all = PaymentType.values();
    return Arrays.asList(all);
  }

  @Override
  public String toString() {
    return capitalizeFully(name().replace('_', ' '));
  }
}
```

Upon being presented with the payment form, the user may either submit a payment or cancel. Depending on the choice made, the payment subflow either ends through the `paymentTaken<end-state>` or the `cancel<end-state>`. As with other subflows, either `<end-state>` will end the subflow and return control to the main flow. But the `id` of the `<end-state>` taken will determine the transition taken next in the main flow.

Now we've stepped all of the way through the pizza flow and its subflows. We've seen a lot of what Spring Web Flow is capable of. Before we move past the Web Flow topic, let's take a quick look at what's involved with securing access to a flow or any of its states.

8.4 Securing web flows

In the next chapter, we'll see how to secure Spring applications using Spring Security. But while we're on the subject of Spring Web Flow, let's quickly look at how Spring Web Flow supports flow-level security when used along with Spring Security.

States, transitions, and even entire flows can be secured in Spring Web Flow by using the `<secured>` element as a child of those elements. For example, to secure access to a view state, you might use `<secured>` like this:

```
<view-state id="restricted">
  <secured attributes="ROLE_ADMIN" match="all"/>
</view-state>
```

As configured here, access to the view state will be restricted to only users who are granted ROLE_ADMIN access (per the `attributes` attribute). The `attributes` attribute takes a comma-separated list of authorities that the user must have to gain access to the state, transition, or flow. The `match` attribute can be set to either `any` or `all`. If it's set to `any`, then the user must be granted at least one of the authorities listed in `attributes`. If it's set to `all`, then the user must have been granted all of the authorities.

You may be wondering how a user is granted the authorities checked for by the `<secured>` element. For that matter, how does the user even log in to the application in the first place? The answers to those questions will be addressed in the next chapter.

8.5 Summary

Not all web applications are freely navigable. Sometimes, a user must be guided along, asked appropriate questions, and led to specific pages based on their responses. In these situations, an application feels less like a menu of options and more like a conversation between the application and the user.

In this chapter, we've explored Spring Web Flow, a web framework that enables development of conversational applications. Along the way, we built a flow-based application to take pizza orders. We started by defining the overall path that the application should take, starting with gathering customer information and concluding with the order being saved in the system.

A flow is made up of several states and transitions that define how the conversation will traverse from state to state. As for the states themselves, they come in one of several varieties: action states that perform some business logic, view states that involve the user in the flow, decision states that dynamically direct the flow, and end states that signify the end of a flow. In addition, there are subflow states, which are themselves defined by a flow.

Finally, we saw hints of how access to a flow, state, or transition can be restricted to users who are granted specific authorities. But we deferred conversation of how the user authenticates to the application and how the user is granted those authorities. That's where Spring Security comes in, and Spring Security is what we'll explore in the next chapter.