CHAPTER 5

Advanced Web Flow Concepts

This chapter continues in the same trend as the previous one but covers more advanced topics. The basic flow definition artifacts explained in the previous chapter are essential to most flow definitions. In this chapter, you'll learn about additional functionality that will give you more power and flexibility when expressing your flows. Some of the subjects covered follow:

- OGNL, Spring Web Flow's default expression language
- Data binding and validation
- Reusing flow definitions as coarse-grained application modules with the help of subflow states

The previous chapter set up the foundation of the "enter payment" use case of the Spring Bank sample application. This chapter will complete the flow definition, filling in the missing parts.

Let's start with an investigation of OGNL, an expression language leveraged by Spring Web Flow.

OGNL

Object Graph Navigation Language (OGNL) is Spring Web Flow's default expression language and is available at `http://www.ognl.org`. OGNL gets and sets properties of Java objects. Along with basic object graph navigation and bean property access, OGNL also supports powerful advanced features such as projections and list selections.

Using OGNL expressions from inside a flow definition avoids writing boilerplate Java code that takes care of trivial tasks such as getting or setting values in one of Spring Web Flow's scopes. For instance, an `Action` implementation like this

```
public class ExampleAction extends MultiAction {

  public Event setFoo(RequestContext context) throws Exception {
    String bar = context.getRequestParameters().get("bar");
    Foo foo = (Foo)context.getFlowScope().get("foo");
    foo.setBar(bar);
    return success();
  }
}
```

can be reformulated in terms of a reusable set action and two OGNL expressions (the `set` action will be explained in more detail later):

```
<set attribute="${foo.bar} " scope="flow"
  value="${requestParameters.bar} "/>
```

In this example, `${foo.bar}` and `${requestParameters.bar}` are OGNL expressions. The `value` expression will be evaluated against `RequestContext`, like most expressions in Spring Web Flow, and translates to `RequestContext.getRequestParameters().get("bar")`. The `attribute` expression will be evaluated against the flow scope attribute map, as indicated by the specified `scope`, making it equivalent to `((Foo)AttributeMap.get("foo")).setBar(. . .)`. Notice how OGNL shields you from low-level details such as type casts. In many cases, this makes simple property access easier and much more readable.

OGNL EXPRESSION DELIMITERS

In the `${foo.bar}` example, the OGNL expressions are delimited using `${. . .}`. This is only necessary if Spring Web Flow does not know whether or not something is an expression. In the case of a set action, the `value` expression and `attribute` expression are always interpreted as OGNL expressions, so the example could have been coded as follows:

```
<set attribute="foo.bar" scope="flow"
  value="requestParameters.bar"/>
```

A typical example where the use of `${. . .}` delimiters is required is embedding expressions into a larger string. For instance

```
externalRedirect:http://www.google.be/search?q=${flowScope.queryString}
```

can obviously not be rewritten as

```
externalRedirect:http://www.google.be/search?q=flowScope.queryString
```

which would be interpreted as one big constant string (this example is taken from the "View Selections" section).

A comprehensive guide to all of the features supported by the OGNL expression language can be found on the OGNL web site: `http://www.ognl.org`. Instead of reproducing that material here, I will only illustrate those OGNL functionalities most relevant to Spring Web Flow. Refer to the OGNL language reference for more details.

■ **Caution** OGNL is very powerful and even allows simple programming constructs like variable or function declarations. Don't abuse these features to start *programming in XML*. Cases requiring nontrivial logic are better implemented using Java and a custom action invoked from your flow definition.

One important issue to keep in mind is refactoring. The more "code" you have in OGNL expressions, the harder the expression becomes to refactor. For instance, IDEs currently do not update an OGNL expression like `foo.bar` when you would rename the `bar` property of `Foo` to something else.

OGNL by Example

Let's study OGNL's more interesting functionality by looking at some simple examples. Consider the Java class in Listing 5-1.

Listing 5-1. *A Person POJO*

```
public class Person {

  private String firstName;
  private String lastName;
  private Person friend;

  public String getFirstName() {
    return firstName;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public String getLastName() {
    return lastName;
  }
```

```
  public void setLastName(String lastName) {
    this.lastName = lastName;
  }

  public Person getFriend() {
    return friend;
  }

  public Person setFriend(Person friend) {
    this.friend = friend;
  }

  public String sayHello() {
    return "Hello " + getFirstName() + " " + getLastName() + "!";
  }

  public void rename(String firstName, String lastName) {
    setFirstName(firstName);
    setLastName(lastName);
  }
}
```

Accessing the `firstName` property of a `Person` object can be done using a simple `${firstName}` expression. Most OGNL expressions can be used to both get (read) and set (write) a property value. When getting the value, the `${firstName}` expression is equivalent to `getFirstName()`, while it translates into `setFirstName(firstName)` when updating the value. OGNL also supports direct access to the public fields of an object, avoiding the need for getters and setters. You can chain together property names using the `.` (dot) operator. For instance, `${friend.firstName}` translates to `getFriend().getFirstName()` or `getFriend().setFirstName(firstName)`, depending on whether you're reading or writing the property.

■ **Caution** Unlike some other expression languages, for instance the JSP expression language, OGNL does not silently ignore `null` values. Consider the case where the `friend` property of a `Person` object is `null`; evaluating the `${friend.firstName}` expression against that object would result in an exception.

OGNL has another little quirk you should be aware of. When you try to set the value of a property to an object that cannot be coerced into the type of that property, OGNL will fail silently and set the property value to `null`. For instance, trying to set the `friend` property to a `java.util.Date` object will result in the `friend` property being set to `null`. Strangely enough, and inconsistent with `Object` typed properties, OGNL will generate an exception when a value cannot be converted to a primitive type number (e.g., `int` or `long`).

When evaluating an OGNL expression against a map, the property name is taken to be the key of an entry in the map. So `${person.firstName}` evaluated against a map set up like this

```
Person person = ...;
Map map = new HashMap();
map.put("person", person);
```

would return the value of the `firstName` property of the `person` object in the map.

OGNL also allows indexed access to elements in arrays or lists using the `[]` syntax. For instance, `${list[0].firstName}` would resolve to the `firstName` property of the first `Person` object in the following list:

```
Person person = ...;
List list = new LinkedList();
list.add(person);
```

OGNL can also be used to invoke *public* methods on objects. To invoke a method, simply add parentheses to the method name. The expression `${sayHello()}` invokes the `sayHello()` method defined by the `Person` class. Method arguments can be specified in between the method parentheses. For instance, `${rename(friend.firstName, friend.lastName)}` calls the `rename(firstName, lastName)` method, passing in the name of the `friend` property. Static methods can also be called using the following format: `@class@method(args)`, for instance `${@java.lang.Math@random()}`.

From time to time, it is useful to express a constant value in an OGNL expression. OGNL supports the following constants:

- Using `null` resolves to the Java `null` literal.
- String literals can be placed between either single or double quotes, for instance, `${rename('John', 'Doe')}`. They support Java character escaping using a backslash (`\`).
- Character literals can be specified using single quotes. Using Java character escapes is again supported.
- The Boolean values `true` and `false` are recognized.
- Numeric constants (integers, floats, etc.) can be defined using the same syntax as in Java. OGNL also supports two extra possibilities however. Using the `B` suffix creates a `BigDecimal` (e.g., `13.4B`), while the `H` (huge) suffix results in a `BigInteger`.

New objects can be created in an expression as usual, using the `new` operator. There is one caveat with this, however: you have to specify the fully qualified class name. Constructor arguments can be passed in just like in a normal method invocation.

OGNL has special syntax to allow easy construction of collection types such as arrays, lists, and maps, for instance:

```
${ { new com.ervacon.sample.Person(), new com.ervacon.sample.Person() }  }
```

This creates a `java.util.List` containing two `Person` objects. Note the extra curly braces, indicating *collection construction syntax*. Similarly, you can also create arrays and maps, as in this example:

```
${new int[] {1, 2, 3 }  }
${#{"person" : new com.ervacon.sample.Person() }  }
```

The first line creates an integer array with three elements: `1`, `2`, and `3`. The second line instantiates a `java.util.Map` containing a single `Person` object indexed as `person`.

Most Java operators are supported by OGNL, including the ternary conditional operator. Here are a few examples:

```
${age > 18 ? "adult" : "minor" }
${order.shippingRequired or (order.weight gt 100) }
${firstName == "John" and lastName != "Doe" }
${firstName.equals("John") }
${firstName not in {"John", "Jack" }  }
```

Of particular interest are the `or`, `and`, `lt`, `gt`, `lte`, `gte`, and `not` synonyms for `||`, `&&`, `<`, `>`, `<=`, `>=`, and `!`, making OGNL expressions XML friendly. OGNL also has a convenient `in` operator that you can use to test whether or not an object is contained in a collection.

Now that you have a good idea of what OGNL expressions are capable of, let's see how Spring Web Flow leverages them in flow definitions.

OGNL in Action

Both the XML and Java flow definition syntax allow you to use OGNL expressions. The XML flow builder will automatically parse an expression string into an `Expression` object. The `AbstractFlowBuilder` superclass used by Java flow builders offers `expression(expressionString)` and `settableExpression(expressionString)` methods that create appropriate `Expression` objects. A *settable expression* is an expression that can be used to set the value of a property, for instance:

```
RequestContext context = ...;
Expression e = expression("flowScope.person.sayHello()");
String helloMessage = (String)e.evaluate(context, null);
```

```
SettableExpression se =
  settableExpression("flowScope.person.firstName");
se.evaluateToSet(context, "Bob", null);
```

Notice that the expression strings in this example do not use the optional `${. . .}` delimiters. The entire strings will be interpreted as OGNL expressions.

Note The `expression(expressionString)` and `settableExpression(expressionString)` methods use the `ExpressionParser` available from the flow service locator (as `getFlowServiceLocator().getExpressionParser()`) to parse the expression strings.

Transition Matching Criteria

As a first example of using OGNL expressions in a flow definition, let's take a look at transition matching criteria. The previous chapter explained that a `TransitionCriteria` object is responsible for matching a transition as eligible for execution. The `TransitionCriteria` interface is very simple (see Listing 5-2).

Listing 5-2. *The TransitionCriteria Interface*

```
package org.springframework.webflow.engine;

public interface TransitionCriteria {

  public boolean test(RequestContext context);
}
```

So far, I have only illustrated the `EventIdTransitionCriteria`, which matches on the last event that occurred in the flow, available as `RequestContext.getLastEvent()`. Spring Web Flow also supports a number of other `TransitionCriteria` types, as detailed in Table 5-1.

Table 5-1. *Transition Criteria Encodings*

Encoded	Description
`eventId`	This encoding is converted into an `EventIdTransitionCriteria` object matching the last event that occurred in the flow, as returned by `RequestContext.getLastEvent()`.
*	This `WildcardTransitionCriteria` matches on everything: it always returns `true`. Not specifying any transition criteria is equivalent to using wildcard transition criteria.

Table 5-1. *Transition Criteria Encodings (Continued)*

Encoded	Description
`${. . .}`	This encoding will be converted into a `BooleanExpressionTransitionCriteria` object that evaluates the specified expression against the `RequestContext`. The result of the expression should be of `Boolean` type and will be returned to the caller. The expression can use the `#result` shorthand notation to refer to the ID of the last event that occurred. So `#result` is equivalent to `RequestContext.getLastEvent().getId()`.
`bean:id`	This encoding will obtain and use the identified `TransitionCriteria` bean from the Spring application context defining the flow definition. This allows you to easily plug in custom `TransitionCriteria` implementations. The `bean:id` form is mainly targeted at XML flow builders. Java flow builders always have the option of directly instantiating a `TransitionCriteria` implementation of choice and using that.

Notice that using `${. . .}` is required in the case of transition criteria expressions, to distinguish them from the default `eventId` criteria.

The value of the `on` attribute of the `<transition>` element of an XML flow definition supports all the encoded forms mentioned in Table 5-1, for instance:

```
<transition on="next" to="showConfirmPayment"/>
<transition on="${#result=='next' and flowScope.person.minor} "
  to="showMinorWarning"/>
<transition on="*" to="nextState"/>
<transition to="nextState"/>
<transition on="bean:myCriteria" to="nextState"/>
```

Likewise, the `on(transitionCriteriaExpression)` method of the `AbstractFlowBuilder` also supports all of these encoded forms:

```
transition(on("next"), to("showConfirmPayment"))
transition(on("${#result=='next' and flowScope.person.minor} ",
  to("showMinorWarning"))
transition(on("*"), to("nextState"))
transition(on("bean:myCriteria"), to("nextState"))
```

Using OGNL expressions to express transition criteria gives you a lot of flexibility defining transitions that are only matched in particular situations. The expression can take any information available in the request context into account. For instance, the `${#result=='next' and flowScope.person.minor}` expression illustrated previously looks at a `Person` object in flow scope. The transition criteria expressed by this OGNL expression could also have been coded directly in Java as follows:

```
public class SampleTransitionCriteria implements TransitionCriteria {

  public boolean test(RequestContext context) {
```

```
    String result = context.getLastEvent().getId();
    Person person = (Person)context.getFlowScope().get("person");
    return result.equals("next") && person.isMinor();
  }
}
```

This implementation can be deployed into a Spring application context and referenced using the `bean:id` notation.

■ **Tip** Remember that transition criteria expressions are evaluated against the `RequestContext`.

Target State Resolvers

OGNL expressions can also be used to specify the target state of a transition. As the previous chapter already mentioned, a `Transition` uses a `TargetStateResolver` strategy to determine the state to transition to *at flow execution time*. The `TargetStateResolver` interface is simple and defines only a single method (Listing 5-3).

Listing 5-3. *The TargetStateResolver Interface*

```
package org.springframework.webflow.engine;

public interface TargetStateResolver {

  public State resolveTargetState(Transition transition,
      State sourceState, RequestContext context);
}
```

Besides the `RequestContext`, a `TargetStateResolver` also has access to the transition for which it is resolving the target state and the source state of that transition.

Be default, the target state will be looked up by ID, but several other encoded forms are also supported, as detailed in Table 5-2.

Table 5-2. *Target State Resolver Encodings*

Encoded	Description
`stateId`	This statically identifies the state to transition to *by ID*.
`${...}`	This encoder calculates the ID of the target state of the transition at flow execution time by evaluating a given OGNL expression against the `RequestContext`. The expression should return a `String`, the ID of the target state.
`bean:id`	This loads a `TargetStateResolver` bean from the Spring application context defining the flow definition. It allows you to plug in custom `TargetStateResolver` implementations.

Using `${. . .}` delimiters is again required to distinguish an OGNL expression from a static state ID.

XML flow definitions can use these encoded forms in the value of the `to` attribute of the `<transition>` element:

```
<transition on="next" to="showConfirmPayment"/>
<transition on="next" to="${flowScope.nextStateId} "/>
<transition on="cancel" to="bean:cancelStateIdCalculator"/>
```

Java flow builders can do the same thing using the `to(targetStateIdExpression)` helper method:

```
transition(on("next"), to("showConfirmPayment"))
transition(on("next"), to("${flowScope.nextStateId} "))
transition(on("cancel"), to("bean:cancelStateIdCalculator"))
```

Instead of using the `bean:id` form, Java flow builder can also directly instantiate and use a `TargetStateResolver` implementation.

■ **Tip** Remember that transition target state expressions are evaluated against the `RequestContext`.

Using OGNL for both transition matching criteria and the target state calculation gives you a lot of flexibility. Chapter 10, "Real-World Use Cases," shows an example that defines a global `back` transition that transitions the flow back to the previous view state.

Decision States

Having the ability to use arbitrary OGNL expressions to specify transition matching criteria allows you to define a conditional branch in your flow, for instance:

```
<transition on="${#result=='next' and flowScope.person.minor} "
  to="showMinorWarning"/>
<transition on="${#result=='next' and !flowScope.person.minor} "
  to="processPerson"/>
```

This is obviously a very common requirement. Spring Web Flow defines a special state type, the *decision state*, to express this kind of conditional branching in a much more readable way:

```
<decision-state id="checkMinor">
  <if test="flowScope.person.minor" then="showMinorWarning"
    else="processPerson"/>
</decision-state>
```

Decision states make *routing decisions*, moving the flow to one of several possible target states. They often sit in between other state types, for instance, view states or action states. Here is a more complete example showing a decision state routing the flow to one of two possible paths:

```
<view-state ...>
  ...
  <transition on="next" to="checkMinor"/>
</view-state>

<decision-state id="checkMinor">
  <if test="flowScope.person.minor" then="showMinorWarning"
    else="processPerson"/>
</decision-state>

<view-state id="showMinorWarning" view="minorWarning">
  <transition on="confirm" to="processPerson"/>
</view-state>

<action-state id="processPerson">
  ...
</action-state>
```

A decision state is implemented by the `TransitionableState` subclass `org.springframework.webflow.engine.DecisionState`. A decision state can contain any number of `if` elements, defining an if-then-else structure that will be translated into transitions with corresponding matching criteria. The `test` attribute of the `if` element defines the transition criteria as an OGNL expression (so there is no need for `${. . .}`). The `then` and `else` attributes identify the target state and can use OGNL expressions to do so, just like normal transitions. A transition using a wildcard (*) matching criteria will be used to represent the `else` of the if-then-else structure. When the flow enters a decision state, the first matching transition is used to transition on to the next state. Consequently, it is only useful to have an `else` on the last `if`, for instance:

```
<decision-state id="switchOnPersonType">
  <if test="flowScope.person.minor" then="processMinor"/>
  <if test="flowScope.person.adult" then="processAdult"/>
  <if test="flowScope.person.elderly" then="processElderly"
    else="showProcessingError"/>
</decision-state>
```

■ **Note** If no matching transition can be found, an exception will be generated.

XML flow builders can use the `<decision-state>` element and nested `<if>` elements to define a decision state, as illustrated in the previous paragraphs. Java flow builders can use any of the `addDecisionState(. . .)` variants provided by the `AbstractFlowBuilder` to add a decision state to the flow, for instance:

```
addDecisionState("checkMinor",
    on("${flowScope.person.minor} "),
    "showMinorWarning", "processPerson");
```

Notice that this code uses the general purpose `on(transitionCriteriaExpression)` method, making use of the `${. . .}` expression delimiters required. Alternatively, a Java flow builder can directly configure a decision state with a number of transitions:

```
addDecisionState("switchOnPersonType",
    new Transition[] {
        transition(on("${flowScope.person.minor} "), to("processMinor")),
        transition(on("${flowScope.person.adult} "), to("processAdult")),
        transition(on("${flowScope.person.elderly} "), to("processElderly")),
        transition(on("*"), to("showProcessingError")) } );
```

Just as any other transitionable state in a web flow, decision states can have entry and exit actions.

■ **Note** The `test` expression of an `if` element in a decision state is evaluated against the `RequestContext`. This was to be expected, since `test` is really just a transition-matching criteria expression.

Similarly, OGNL expressions in the `then` and `else` attributes of an `if` element are also evaluated against the `RequestContext`. They are nothing more than expressions resolving a transition target state.

Set Actions

The "OGNL" section introduced the set action. Spring Web Flow provides `SetAction` as a convenient way to set a value in a particular scope. `SetAction` leverages OGNL expressions for calculating the value to set and actually setting it:

```
<set attribute="${foo.bar} " scope="flow"
  value="${requestParameters.bar} "/>
```

The `value` expression is evaluated against the `RequestContext`, while the `attribute` expression will be evaluated against the identified scope map. If you do not explicitly

specify one of the available scopes (`request`, `flash`, `flow`, or `conversation`), the request scope will be used by default. Recall that use of the `${. . .}` delimiter around the expressions is optional, because the values of the `attribute` and `value` attributes are always expected to be expressions.

`SetAction` is an action like any other Spring Web Flow action. You can use it anywhere you can use a normal action, for instance, in action states or as part of a transition. `SetAction` always returns the `success` event, which you can ignore by turning the action into a *named action* as described in the "Action States" section of the previous chapter. Here is an example illustrating this:

```
<set attribute="showWarning" value="true" name="setShowWarning"/>
```

XML flow definitions can define set actions using the convenient `<set>` element, as shown previously. Java flow builders have no direct support for set actions and will have to instantiate the `SetAction` class directly before adding it to an action state or transition:

```
SetAction set = new SetAction(
    settableExpression("showWarning"),
    ScopeType.REQUEST,
    expression("true"));
addActionState("sampleActionState",
    name("setShowWarning", set),
    ...);
```

Notice how this code uses the `settableExpression(expressionString)` and `expression(expressionString)` helper methods.

Bean Invoking Actions

Using actions to bridge the web tier and middle tier of your application was discussed in the "Implementing Actions" section of the previous chapter. That section showed a `PaymentAction` in the context of the "enter payment" use case of the Spring Bank sample application. This action obtains a `Payment` object from flow scope and passes it on to the `PaymentProcessingEngine`. As you can imagine, this kind of straightforward bridging code is very common. Spring Web Flow provides a bean invoking action, making this kind of bridging trivial. A *bean invoking action* can directly invoke any public method on any bean available in the application context defining the flow definition or in the flow local application context. This has two important benefits:

- It avoids a proliferation of lots of trivial `Action` implementations invoking methods on application (service) layer beans, like `PaymentAction`.

- The invoked bean can be a POJO (plain old Java object) and does not need to implement any Spring Web Flow interfaces. As a result, your code is less dependent on the framework, which is always a good thing.

Let's use such a bean invoking action to rework `enterPayment-flow`, dropping `PaymentAction` along the way. The following example shows a bean invoking action definition roughly equivalent to the `PaymentAction` developed in the previous chapter:

```
<bean-action bean="paymentProcessingEngine" method="submit">
  <method-arguments>
    <argument expression="${conversationScope.payment} "/>
  </method-arguments>
</bean-action>
```

This action will invoke the `submit(payment)` method directly on the `paymentProcessingEngine` bean in the application context, passing in the `Payment` object stored in conversation scope as a method argument.

Bean invoking actions are implemented using an `AbstractBeanInvokingAction` subclass. Since they implement the `Action` interface, like any other action, they can be used anywhere a normal action can be used, for instance in action states, as state entry actions, or as part of transition execution. Just like all other actions, bean invoking actions can also be *named*, as explained in the previous chapter.

You can think of a bean invoking action as a decorator that adapts a public method on a POJO to the `Action` contract. To make this work, a bean invoking action needs three key pieces of information, as illustrated in Figure 5-1:

- An exact specification of the method to invoke, and the POJO (bean) to invoke it on. In Java, methods are described using a *method signature*.
- A specification of how result objects returned by the invoked method should be exposed to the flow execution.
- A strategy to create an action result event based on the outcome of the method invocation.

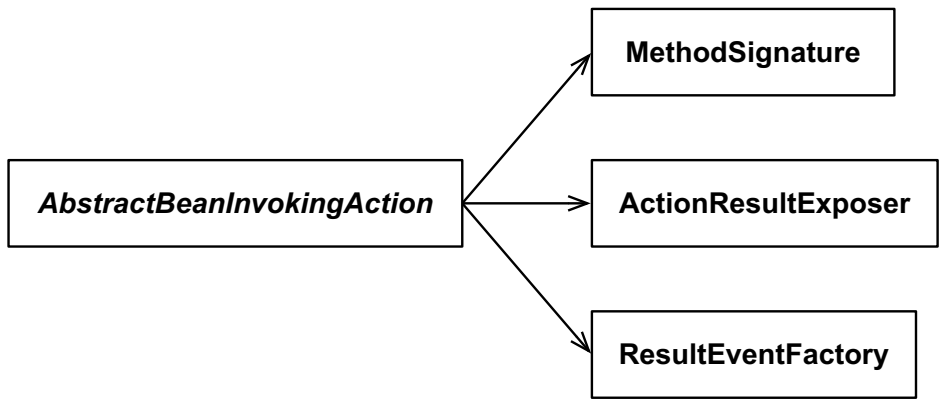

Figure 5-1. *Bean invoking action class diagram*

Let's investigate these three parts in more detail.

POJO Method Signatures

XML flow definitions identify the bean to invoke methods on using the `bean` attribute of the `<bean-action>` element. The value of this attribute is the ID of a bean available from the Spring application context defining the flow definition. The name of the method to invoke is specified using the `method` attribute, while a nested `<method-arguments>` element specifies the arguments to be passed to the method. If the invoked method does not declare any parameters, the `<method-arguments>` can be omitted.

The `expression` attribute of `<argument>` elements, nested inside the `<method-arguments>` element, allows you to supply argument values to the method when it is invoked. The argument value is obtained by evaluating the provided OGNL expression against the `RequestContext`. The order of the `<argument>` elements should correspond with the order of the method parameters declared by the method. For instance, take the `rename` method defined by a fictitious `PersonService` interface, as shown in Listing 5-4.

Listing 5-4. *PersonService*

```
public interface PersonService {

  public void rename(Person person, String firstName, String lastName);
  public void notifyPersonsBornOn(Date date);
  public Person getPerson(Long id);
  public PersonStatus getPersonStatus(Long id);
}
```

This `rename` method can be invoked using the following bean invoking action declaration:

```
<bean-action bean="personService" method="rename">
  <method-arguments>
    <argument expression="${flowScope.person} "/>
    <argument expression="${'John'} "/>
    <argument expression="${'Doe'} "/>
  </method-arguments>
</bean-action>
```

This, of course, assumes that a `PersonService` bean with the ID `personService` is defined in the application context.

Java flow builders can use variants of the `action(. . .)` helper method to set up a bean invoking action. A method signature (modeled using the `MethodSignature` class) can be created using the `method(signatureString)` helper. Here is an example creating

a bean invoking action invoking the `rename` method on the `personService` bean, as discussed before:

```
action("personService",
    method("rename(${flowScope.person} , ${'John'} , ${'Doe'} )"));
```

Notice how the argument expressions are embedded in the string-encoded method signature.

Spring Web Flow also supports automatic type conversion of method arguments. For instance, if a method declares a parameter of type `java.util.Date`, but the argument value is of type `java.lang.String`, Spring Web Flow can automatically convert the string into a `Date`. To enable this automatic type conversion, you need to explicitly specify the declared method parameter type using the `parameter-type` attribute, for instance:

```
<bean-action bean="personService" method="notifyPersonsBornOn">
  <argument expression="${requestParameters.date} "
    parameter-type="java.util.Date"/>
</bean-action>
```

Similarly, a Java flow builder can specify the parameter type by including it in the string-encoded method signature:

```
action("personService",
    method("notifyPersonsBornOn( \
      java.util.Date ${requestParameters.date} )"));
```

This type conversion functionality leverages the conversion service that will be discussed in "The Conversion Service" section.

■ **Tip** Remember that method argument value expressions are evaluated against the `RequestContext`.

Action Result Exposers

Java methods can have return values. Bean invoking actions will expose the value returned by the invoked bean method to the flow execution using `ActionResultExposer`. If the method does not return any value (it is declared `void`), nothing needs to happen of course.

The `ActionResultExposer` strategy simply puts the method return value into a specified scope using a given name. XML flow definitions can configure the `ActionResultExposer` using the `<method-result>` element nested inside the `<bean-action>` element, like so:

```
<bean-action bean="personService" method="getPerson">
  <method-arguments>
```

```
    <argument expression="${flowScope.personId} "/>
  </method-arguments>
  <method-result name="person" scope="flow"/>
</bean-action>
```

If you do not explicitly specify the `scope`, request scope will be used as a default. Similarly, Java flow builders can use the `action(beanId, methodSignature, resultExposer)` method, passing in an `ActionResultExposer` object configured using the `result(resultName, resultScope)` helper method. Here is an example equivalent to the XML fragment shown previously:

```
action("personService",
    method("getPerson(${flowScope.personId} )"),
    result("person", ScopeType.FLOW));
```

If the invoked POJO method does not return a result object but instead throws an exception, that exception will be propagated by the bean invoking action. In the "Handling Exceptions" section, you will learn how to handle these exceptions.

Result Event Factories

Recall that action execution methods return `Event` objects. The final aspect of adapting a normal POJO method to the `Action` interface is interpreting the method return value and returning an appropriate `Event`. This is the responsibility of a `ResultEventFactory` strategy.

Spring Web Flow will automatically decide which `ResultEventFactory` to use based on the declared return type of the invoked POJO method. The recognized return types and their corresponding event IDs are detailed in Table 5-3.

Table 5-3. *Mapping Method Return Types to Event IDs*

Return Type	Event ID	Description
`Boolean` or `boolean`	`yes` or `no`	A `yes` event indicates a `true` return value, while `no` means `false`.
`LabeledEnum`	The enum label	`LabeledEnum` instances, as defined by the Spring Framework, are recognized. An event will be generated having the *label* of the `LabeledEnum` as `id`. The actual `LabeledEnum` instance returned by the method will be included as payload in the event, keyed as `result`. Note that a Spring `LabeledEnum` is a JDK 1.3- and 1.4-compatible precursor to the Java 5 `enum`.
`Enum`	The enum name	A Java 5 enum (`java.lang.Enum`) return value will be mapped to an event having the *name* of the enum as its `id`. The `Enum` object itself will be included in the event as payload, indexed as `result`.

Table 5-3. *Mapping Method Return Types to Event IDs (Continued)*

Return Type	Event ID	Description
`String`	The string	When the invoked method returns a `String`, that string will be used as the event `id`.
`Event`	The event	In the odd case where the invoked method returns a Spring Web Flow `Event`, the event is used as-is.
Anything else	`success`	If the invoked method is declared `void` or has an unrecognized return type, the `success` event is generated. The returned object, if any, is included as payload in the generated event, keyed as `result`.

If the declared method return type is recognized, but the actual return value of the method at runtime is `null`, a *null event* will be generated with event ID `null`.

There is no direct way to influence the `ResultEventFactory` logic from inside a flow definition. The mapping shown previously has proven to be sufficient in the majority of cases. The following example illustrates a simple action state calling the imaginary `getPersonStatus(id)` method of the `PersonService`:

```
<action-state id="queryPersonStatus">
  <bean-action bean="personService" method="getPersonStatus">
    <method-arguments>
      <argument expression="${flowScope.personId} "/>
    </method-arguments>
  </bean-action>
  <transition on="MINOR" to="processMinor" />
  <transition on="ADULT" to="processAdult" />
  <transition on="ELDERLY" to="processElderly" />
  <transition on="null" to="showProcessingError" />
</action-state>
```

The `getPersonStatus(id)` returns a `PersonStatus` enum:

```
public enum PersonStatus {

  MINOR, ADULT, ELDERLY;
}
```

■ **Note** Notice that this example shows that action states can fulfill a similar *routing* requirement like the decision states discussed in the "Decision States" section.

Evaluate Actions

Bean actions can invoke any method on a bean defined in a Spring application context. *Evaluate actions* are similar but invoke a method on any object available via the `RequestContext`. While bean invoking actions invoke the identified method themselves, evaluate actions use OGNL's method invocation capability, as discussed in the "OGNL by Example" section. More generally, evaluate actions can evaluate any OGNL expression against the request context and expose the result to the flow execution.

Evaluate actions are implemented by the `EvaluateAction` class, a subclass of `AbstractAction`. Since evaluate actions are normal actions, you can use them anywhere you can use a normal action. Evaluate actions can also be given a name, turning them into *named actions*.

An XML flow definition can define an evaluate action using the `<evaluate-action>` element. The `expression` attribute of this element specifies the OGNL expression to evaluate:

```
<evaluate-action expression="flowScope.person.rename( \
  flowScope.person.friend.firstName, \
  flowScope.person.friend.lastName)"/>
```

Note that you don't need to use the `${. . .}` delimiters since the value of the `expression` attribute is assumed to be an expression. Java flow builders can again use one of the overloaded variants of the `action(. . .)` helper method to add an evaluate action to the flow. The `expression(expressionString)` method can be used to parse the expression:

```
action(expression("flowScope.person.rename( \
  flowScope.person.friend.firstName, \
  flowScope.person.friend.lastName"));
```

Objects resulting from OGNL expression evaluation can be exposed to the flow using an `ActionResultExposer`, just like with bean invoking actions. XML flow definitions can configure the `ActionResultExposer` using the `<evaluation-result>` element, similar to the `<method-result>` element used for bean actions (consult the "Action Result Exposers" section for more details):

```
<evaluate-action expression="flowScope.person.sayHello()">
  <evaluation-result name="helloMessage" scope="flow"/>
</evaluate-action>
```

This will put the message returned by the `sayHello()` method into flow scope, keyed as `helloMessage`. If you do not explicitly specify the `scope`, request scope will be used. Java flow builders can use the `result(resultName, resultScope)` helper method to initialize an `ActionResultExposer` and pass that into the `action(expression, resultExposer)` method:

```
action(expression("flowScope.person.sayHello()"),
  result("helloMessage", ScopeType.FLOW));
```

The event returned by the `EvaluateAction` is determined using `ResultEventFactory`, just like in the case of bean invoking actions. Refer to the "Result Event Factories" section for exact details on how expression evaluation results will be mapped to action result events. The mapping detailed in Table 5-3 will be applied to the type of object resulting from expression evaluation.

■ **Tip** Remember that evaluate actions evaluate an expression against the `RequestContext`.

Evaluate actions are deceptively simple. Using them to invoke methods on stateful beans stored in one of the flow execution scopes provides an object-oriented, stateful programming model for web applications. Instead of storing only data in the flow execution scopes, and manipulating that data using actions, you can now put real *objects* into the flow execution scopes, combining data and behavior. A flow can trigger the behavior of such stateful objects using an evaluate action. This leads to a much more object-oriented style of programming web applications.

Flow Variables

When using evaluate actions, objects that you invoke methods on need to be available through the `RequestContext`, typically in one of the flow execution scopes. You could use a normal `Action` to initialize such an object, but Spring Web Flow offers a more explicit approach: flow variables.

Flow variables, implemented as subclasses of the `FlowVariable` class, will be initialized when a flow session starts. They are essentially factories for the initial value of the variable, as returned by the `createVariableValue(context)` hook method. Two concrete `FlowVariable` implementations are provided, and they are shown in Figure 5-2.

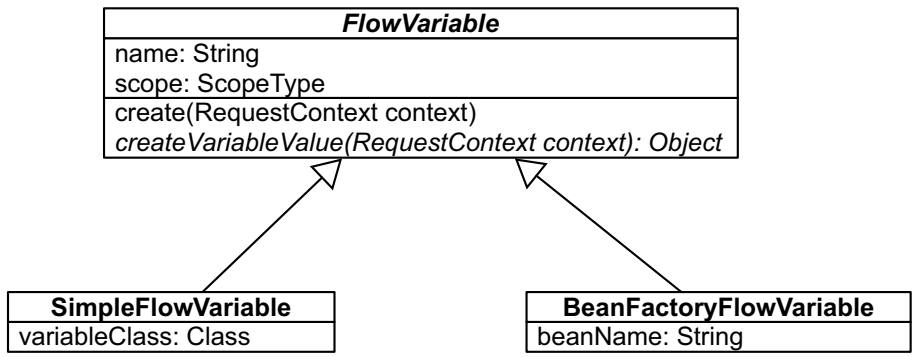

Figure 5-2. *Flow variable class diagram*

`SimpleFlowVariable` will instantiate a configured class using the default constructor each time `createVariableValue(context)` is called (for instance, when a new flow session is spawned). A `BeanFactoryFlowVariable` will look up a *prototype* bean using a configured ID in a Spring `BeanFactory`, typically the application context containing the flow definition. The identified bean must be configured as a prototype bean. This is enforced to avoid threading-related issues, as there might be multiple concurrent flow sessions for a flow definition defining a flow variable.

An XML flow definition can define a flow variable using the `<var>` element. The `name` attribute of this element is required and specifies the name used to index the variable value in the configured `scope` (if no scope is explicitly specified, flow scope is assumed by default):

```
<var name="person" scope="conversation"/>
```

This defines a `BeanFactoryFlowVariable`, where the bean ID defaults to the variable name. You can also explicitly specify the bean ID using the `bean` attribute:

```
<var name="person" bean="initialPerson" scope="conversation"/>
```

Defining `SimpleFlowVariable` is done using the `class` attribute instead of the `bean` attribute. The value of the class attribute is the fully qualified class name of the class to instantiate.

```
<var name="person" class="com.ervacon.sample.Person"
  scope="conversation"/>
```

■ **Caution** Specifying both a `bean` and `class` attribute results in a `BeanFactoryFlowVariable` instance.

Since flow variables are an optional part of a flow definition, Java flow builders need to override the `buildVariables()` method. The `AbstractFlowBuilder` superclass of Java flow builders offers no direct support for flow variables. Instead, Java flow builders have to directly instantiate flow variables and add them to the flow under construction, available using `getFlow()`:

```
public void buildVariables() throws FlowBuilderException {
  getFlow().addVariable(new BeanFactoryFlowVariable(
      "person", "initialPerson",
      getFlowServiceLocator().getBeanFactory(), ScopeType.CONVERSATION));
  getFlow().addVariable(new SimpleFlowVariable(
      "person", Person.class, ScopeType.CONVERSATION));
}
```

Notice how the bean factory containing the flow definition can be retrieved using `getFlowServiceLocator().getBeanFactory()`. Because Java flow builders directly instantiate the flow variable, they are not limited to using `SimpleFlowVariable` and `BeanFactoryFlowVariable`. You can also directly subclass `FlowVariable` and use your own implementation.

Accessing Scoped Beans

Spring 2 introduced the concept of bean *scopes*. In previous versions of the Spring Framework, beans could only be singletons or prototypes. Spring 2 also supports web-application–related scopes like request, session, and global session. A bean declared to be in session scope will live in the HTTP session:

```
<bean id="person" class="com.ervacon.sample.Person" scope="session"/>
```

To be able to use Spring 2's web-related scopes, you need to add `ServletRequestListener` to your `web.xml` deployment descriptor (consult the Spring reference documentation for more details):

```
<web-app version="2.4" ...>

  ...

  <listener>
    <listener-class>
      org.springframework.web.context.request.RequestContextListener
    </listener-class>
  </listener>

  ...

</web-app>
```

Since web-scoped beans are normal beans as far as Spring Web Flow is concerned, you can use a bean invoking action to invoke methods on those beans, for instance:

```
<bean-action bean="person" method="sayHello">
  <method-result name="helloMessage" scope="flow"/>
</bean-action>
```

This is roughly equivalent to the following evaluate action:

```
<evaluate-action expression="externalContext.sessionMap.person.sayHello()">
  <evaluation-result name="helloMessage" scope="flow"/>
</evaluate-action>
```

The main difference between the two approaches is that a Spring 2 web-scoped bean will automatically be created the first time it is accessed, while an evaluate action does not do that.

■ **Tip** Using bean invoking actions to invoke methods on Spring 2 web-scoped beans provides an elegant way for a flow to share information with other parts of the application that do not live inside the flow execution, for instance, normal Spring Web MVC `Controller` implementations.

Furthermore, Spring Web Flow 2 enhances the scoping functionality in Spring 2 with support for the flow execution scopes defined by Spring Web Flow.

The Conversion Service

Spring Web Flow type conversion support was already mentioned briefly in the "POJO Method Signatures" section, when talking about passing arguments to POJO methods. Internally, Spring Web Flow uses a powerful and flexible conversion service to do these kinds of type conversions. A *conversion service* is an implementation of the `org.springframework.binding.convert.ConversionService` interface.

By default, Spring Web Flow uses a conversion service containing converters that can convert from string representations to common Java types. Table 5-4 details the conversions that are supported out of the box.

Table 5-4. *Default Conversions from Text to Target Type*

Target Type	Supported String Encodings
`Class`	A fully qualified class name (such as `java.lang.Integer`) and a recognized type alias (such as `integer`) are converted into the corresponding `Class` object. Using the `type:` prefix, you can explicitly indicate that the type is referenced by an alias, while the `class:` prefix indicates a fully qualified class name.
`Number`	This translates a number string recognized by the `java.text.NumberFormat` class or understood by the `decode(string)` method of the target type into the corresponding `Number` object. All `Number` subtypes are supported: `Integer`, `Short`, `Byte`, `Long`, `Float`, `Double`, `BigInteger`, and `BigDecimal`.
`Boolean`	Strings `true`, `on`, `yes`, and `1` will result in Boolean `true`, while `false`, `off`, `no`, and `0` are converted into Boolean `false`.
`LabeledEnum`	The string representation is assumed to be the label of a `LabeledEnum`, which will be looked up. `LabeledEnum` is compatible with JDK 1.3 and 1.4 and is a precursor to the Java 5 enum provided by the Spring Framework.