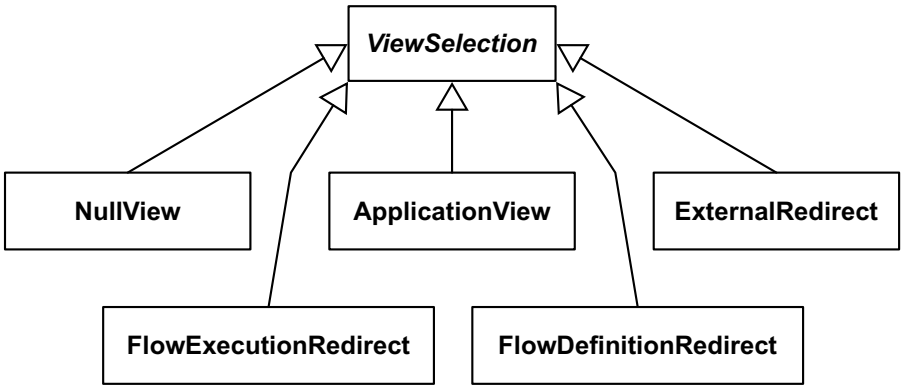


Figure 5-4. *View selection types*

Data Binding and Validation

Among the important functionalities provided by most web application frameworks are data binding and validation. Data binding is the process of mapping incoming request parameters to the model, potentially converting the `String` request parameters into richly typed objects, such as `AccountNumber` objects. Validation is the process of verifying the model information after it has been updated by the data binding process, ensuring it is valid and consistent. Data binding and validation are typically the responsibility of the controller in the MVC triad.

As an MVC controller, Spring Web Flow provides advanced data binding and validation support, leveraging the `DataBinder` machinery provided by the Spring Framework. The Spring `DataBinder` uses the JavaBean conventions to perform data binding. For instance, an HTML form input field named `debitAccount`

```
<input type="text" name="debitAccount"/>
```

will bind onto a `debitAccount` property of the form-backing object, a `Payment` object in the case of the "enter payment" web flow:

```
public class Payment implements Serializable {

  private Account debitAccount = new Beneficiary();
```

```
  public Account getDebitAccount() {
    return debitAccount;
  }

  public void setDebitAccount(Account debitAccount) {
    this.debitAccount = debitAccount;
  }
}
```

To convert the `String`-valued `debitAccount` request parameter into an `Account` object, the `DataBinder` will use a `java.beans.PropertyEditor`. Property editors typically implement `setAsText(text)` and `getAsText()` methods to do to-string and from-string conversions:

```
public class AccountNumberEditor extends PropertyEditorSupport {

  public void setAsText(String text) throws IllegalArgumentException {
    setValue(StringUtils.hasText(text) ? new AccountNumber(text) : null);
  }

  public String getAsText() {
    return getValue() == null ? "" : getValue().toString();
  }
}
```

The `DataBinder` will automatically detect property editors located in the same package as the subject class, with a class name having the `Editor` suffix, for instance, `AccountNumberEditor` for `AccountNumber` objects. You can also manually register property editors with a `DataBinder` (which is a `PropertyEditorRegistry`) by implementing a `PropertyEditorRegistrar`:

```
public class PaymentPropertyEditorRegistrar
  implements PropertyEditorRegistrar {

  private AccountRepository accountRepository;

  public void setAccountRepository(AccountRepository accountRepository) {
    this.accountRepository = accountRepository;
  }
```

```
  public void registerCustomEditors(PropertyEditorRegistry registry) {
    registry.registerCustomEditor(Account.class, "debitAccount",
      new PropertyEditorSupport() {
        public void setAsText(String text)
            throws IllegalArgumentException {
          setValue(accountRepository.getAccount(new AccountNumber(text)));
        }
      } );
  }
}
```

The preceding registrar registers a property editor for the `debitAccount` property of the form-backing object, loading the identified `Account` object from the repository (database). Out of the box, Spring provides several reusable property editor implementations, such as a `CustomDateEditor`, which can be configured with a `java.text.DateFormat` to define the string representation of the dates.

■ **Note** In Spring Web Flow 1, the `DataBinder` is not aware of the Spring Web Flow conversion service. As a result, it will not use the converters registered with the conversion service during data binding.

Once data binding has completed, the form-backing object can be validated using an `org.springframework.validation.Validator` implementation:

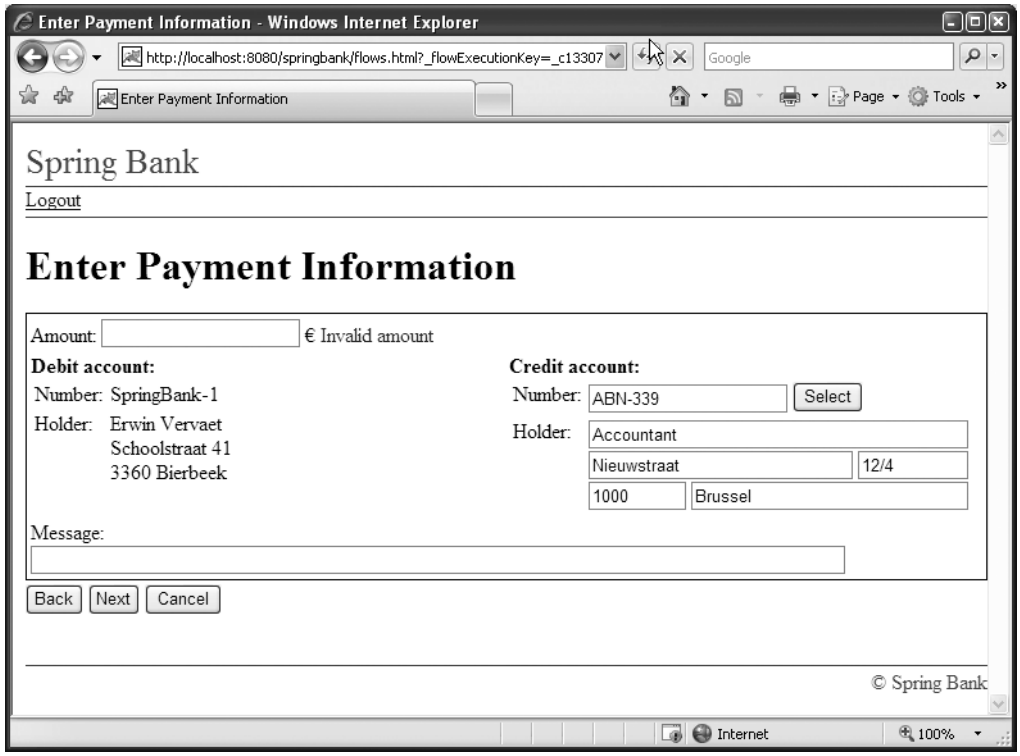
```
public class PaymentValidator implements Validator {

  public boolean supports(Class clazz) {
    return Payment.class.isAssignableFrom(clazz);
  }

  public void validate(Object obj, Errors errors) {
    Payment payment = (Payment)obj;
    if (payment.getAmount() == null ||
        payment.getAmount().compareTo(new BigDecimal(0)) < 0) {
      errors.rejectValue("amount", "error.invalidPaymentAmount",
        "The payment amount is invalid");
    }
```

```
    ValidationUtils.rejectIfEmpty(errors, "debitAccount.number",
      "error.debitAccountRequired", "The debit account is required");
    ValidationUtils.rejectIfEmpty(errors, "creditAccount.number",
      "error.creditAccountRequired", "The credit account is required");
  }
}
```

The following screen shot below shows the Enter Payment Information page, displaying a validation error generated by the preceding validator:



Notice that a Spring `Validator` is not web specific and can be reused in other parts of the system. During the data binding and validation process, all errors that occur are recorded in an `Errors` object. This object can be exposed to a view for rendering, for instance, using the Spring form tag library (introduced in Spring 2.x) or the Spring binding tag library.

The Spring DataBinder has support for binding indexed properties on collection types such as arrays, lists, or maps. For instance, a request parameter named map['key'].prop maps to the prop property of the object indexed as key in a map. The map itself is the map property of the form-backing object. Similar logic applies to a parameter named list[3].prop.

The FormAction

A web flow can easily use the Spring DataBinder machinery with the help of a specialized action: the FormAction. A FormAction manages a form-backing object, and an associated Errors object, in one of the flow execution scopes.

When directly using the org.springframework.webflow.action.FormAction class, you have to configure the type of the form-backing object using the formObjectClass property. The FormAction will instantiate a new form-backing object using the default constructor whenever required. Alternatively, you can subclass FormAction and override the createFormObject(context) hook method, typically loading a form object instance from a backing data store:

```
protected Object createFormObject(RequestContext context)
    throws Exception {
  Long objectId = context.getFlowScope().getLong("objectId");
  return serviceLayer.loadObject(objectId);
}
```

All useful properties of the FormAction are listed in Table 5-5.

Table 5-5. *FormAction Properties*

Property	Description
formObjectClass	The class of the form object; required unless createFormObject(context) is overridden in a subclass
formObjectName	The name of the form-backing object; defaults to formObject, or is calculated by convention based on the form object class (for instance, a Payment object will be named payment)
formObjectScope	The flow execution scope to store the form-backing object in; defaults to flow scope
formErrorsScope	The flow execution scope to store the Errors object in; defaults to flash scope
propertyEditorRegistrar	PropertyEditorRegistrar used to register custom property editors with the DataBinder
validator	Validator implementation to use

Configuring `FormAction` for use in the `enterPayment-flow` is simple; just set up an appropriate bean definition:

```
<bean id="formAction"
  class="org.springframework.webflow.action.FormAction">
  <property name="formObjectClass"
    value="com.ervacon.springbank.domain.Payment"/>
  <property name="formObjectName" value="payment"/>
  <property name="formObjectScope" value="CONVERSATION"/>
  <property name="formErrorsScope" value="CONVERSATION"/>
  <property name="propertyEditorRegistrar">
    <bean class="com.ervacon.springbank.web.PaymentPropertyEditorRegistrar"
      autowire="byName"/>
  </property>
  <property name="validator">
    <bean class="com.ervacon.springbank.domain.PaymentValidator"/>
  </property>
</bean>
```

A `Payment` object is used as a form-backing object and stored in conversation scope. The associated `Errors` instance is also stored in conversation scope.

■ **Tip** `FormAction` beans are typically specific to a web flow and are preferably defined in the flow local application context. Refer to the "Flow Local Bean Definitions" section in Chapter 4 for more information.

When defining the scope of the form-backing object and `Errors` object, you typically want to stick to one of two combinations:

- *The form object in flow scope and the* `Errors` *object in flash scope*: This is the default. Using these settings you will get undo behavior when pressing the Back button; the previous page is redisplayed and all future edits have been undone.
- *Both the form object and* `Errors` *object in conversation scope*: In this case, all the edits to the form-backing object will be maintained, even when pressing the browser's Back button.

Other combinations are, of course, also possible. Unless you have a real requirement for special behavior however, use one of the two combinations mentioned in the preceding list.

■ **Tip** Give this a try! Configure the `FormAction` bean used by the "enter payment" flow to use one of the combinations listed in this section, and see how the flow reacts when you click the browser's Back button on the Confirm Payment page.

Recall that you can obtain the completed Spring Bank sample application from the Ervacon Subversion repository available at `https://svn.ervacon.com/public/spring/samples/trunk/springbank`.

Form-Handling Methods

`FormAction` is a `MultiAction` subclass hat defines five action execution methods for handling HTML input forms. Let's take a look at each of these methods:

setupForm(context)

Before a form can be properly displayed, the `DataBinder` needs to be initialized, possibly registering custom property editors, and the form-backing object needs to be created or loaded. These tasks are the responsibility of the `setupForm(context)` action execution method, which always signals the `success` event.

■ **Tip** Normally, you should always make the `setupForm(context)` action a render action of your view state. This ensures that the form handling machinery and form-backing object are properly initialized before the form view is rendered.

bindAndValidate(context)

Bind incoming request parameters to the form-backing object and then validate the form object. Validation is only done when a `Validator` has been registered with the `FormAction` using the `validator` property.

The `success` event is signaled when binding and validation were successful; otherwise, `error` is signaled.

bind(context)

This is similar to `bindAndValidate(context)`, but it performs only data binding, not validation.

It will signal the `success` event when data binding succeeded or `error` otherwise (for instance, when there are type conversion problems).

validate(context)

Validate the form-backing object using the configured `Validator`, signaling `success` or `error`, depending on the outcome of validation.

■ **Tip** The `bindAndValidate(context)`, `bind(context)`, and `validate(context)` actions are best used when linked to a transition of a view state, as *transition execution criteria*, for instance, a `submit` transition that does `bindAndValidate(context)`. This way, the transition will roll back if the data binding or validation fails, reentering the view state to display the errors. If data binding and validation succeed, the flow moves on to the next state.

---

resetForm(context)

Use this to reset the form-backing object, recreating or reloading it. The associated `Errors` object will also be reinitialized. Always signals the `success` event.

Using the action methods provided by the `FormAction`, you can define a view state that does powerful and flexible form handling, typically called a *form state*. The following example shows the complete definition of the `showEnterPaymentInfo` form state of the `enterPayment-flow`:

```
<view-state id="showEnterPaymentInfo" view="enterPaymentInfo">
  <render-actions>
    <action bean="formAction" method="setupForm"/>
  </render-actions>
  <transition on="next" to="showConfirmPayment">
    <action bean="formAction" method="bindAndValidate"/>
  </transition>
</view-state>
```

Before the `enterPaymentInfo` view is rendered, the form-handling machinery will be set up with a call to `setupForm(context)`. Every time the form is submitted along with the `next` event, data binding and validation will take place with the help of `bindAndValidate(context)`. Notice that the global `cancel` transition that was added to the flow definition before does not do any data binding or validation. This is exactly what you typically want: canceling the flow shouldn't fail because of validation errors!

Besides the `DataBinder` initialization and setup of the form-backing object, as done by the `setupForm(context)` action, there is typically an additional task to execute before a form can be displayed: loading reference data. For instance, consider a form that includes a drop-down with predefined values, loaded from the database. The `FormAction` has no

support for loading reference data. Instead, you can just add an additional render action to your flow to load the reference data, for instance:

```
<render-actions>
  <action bean="formAction" method="setupForm"/>
  <bean-action bean="accountRepository" method="getAccounts">
    <method-arguments>
      <argument expression="externalContext.sessionMap.user.clientId"/>
    </method-arguments>
    <method-result name="accounts"/>
  </bean-action>
</render-actions>
```

This example uses a bean invoking action to load an `accounts` list from the account repository. Keep in mind that the `FormAction` is a multiaction, making it possible to add additional action execution methods in a subclass. Such an extra action method could also be used to load reference data.

■ **Note** A flow can potentially deal with multiple different form-backing objects. In this case, just define multiple `FormAction` implementations and configure them with a different form object name.

Piecemeal Validation

By default, the `Validator` implementation registered with the `FormObject` will be used to validate the entire form object. In a flow, you typically want piecemeal validation on intermediate steps and complete validation at the end of the flow.

Spring Web Flow allows piecemeal validation by specifying the `validatorMethod` attribute of the `bindAndValidate(context)` or `validate(context)` actions:

```
<action bean="formAction" method="bindAndValidate">
  <attribute name="validatorMethod" value="validateFirstStep"/>
</action>
```

This will invoke the `validateFirstStep(formObject, errors)` method on the registered `Validator`. You can add any number of piecemeal validation methods to the validator, as long as they adhere to the following signature:

```
public void methodName(FormObjectClass formObject, Errors errors);
```

■ **Note** Notice that the signature of a piecemeal validation method defines a parameter of the class of the form object, not just `Object` like in the case of the `Validator` interface. This saves you from doing an extra type cast in each piecemeal validation method.

Subclassing FormAction

If you need to customize input form handling beyond what is possible using the properties of the `FormAction`, you can subclass `FormAction` and override one of the defined hook methods—a form of the Template Method design pattern (Gamma et al 1995).

Four hook methods are available; they are explained in the following sections.

createFormObject(context)

This method was already mentioned previously. The default implementation instantiates the form object class using the default constructor. If you want to load an existing form-backing object, override this method and add the necessary code.

initBinder(context, binder)

This hook allows advanced configuration of the `DataBinder`. For instance, using the `initDirectFieldAccess()` method of the `DataBinder`, you can have the `DataBinder` bind directly onto public fields of the form object, instead of requiring property getters and setters.

Another typical example of `DataBinder` configuration is setting the allowed fields for automatic data binding. This prevents a malicious user from injecting values into your form-backing object by adding request parameters to a request.

registerPropertyEditors(context, registry)

Overriding this method is an alternative to configuring `FormAction` with a `PropertyEditorRegistrar`. You can register any required custom property editors with the `DataBinder`. The default implementation just delegates to the registered `PropertyEditorRegistrar`, if any. In general, it is preferred to use a `PropertyEditorRegistrar`.

■ **Caution** Do not use the `initBinder(context, binder)` method to register custom property editors with the data binder. The `initBinder(context, binder)` method will only be called when a new data binder is created. As a result, property editors registered in this method will not be available in all situations. Instead, override `registerPropertyEditors(context, registry)`, or configure the `propertyEditorRegistrar` property of the `FormAction`.

validationEnabled(context)

Use this hook to determine, based on information available in the `RequestContext`, whether or not validation should occur in the current request. The default implementation always returns `true`.

This concludes the discussion of the data binding and validation support available in Spring Web Flow. Keep in mind that data binding and validation are not a core part of Spring Web Flow. In some situations, particularly when integrating with JSF, it is better to have the JSF components do basic data binding and validation. `FormAction` validation support can still prove useful for *cross validation*: ensuring that the combination of selected field values is valid.

Let's now move on to another important topic: reusing flows as subflows.

Subflows

An important topic still left to cover is modularity. How does Spring Web Flow tackle the modularity concern identified in Chapter 1, "Introducing Spring Web Flow"? Ideally, the framework would make it possible to capture use cases as coarse-grained application modules, having a well-defined input-output contract to allow black box reuse. Spring Web Flow allows exactly this. In Spring Web Flow, a flow definition *is* a reusable module. The `enterPayment-flow` we have been developing is a coarse-grained representation of the "enter payment" use case in the Spring Bank sample application. You have already seen flows used as top-level modules, launched directly in response to a request targeted at the flow controller and containing a `_flowId` request parameter. This section will explain how flows can be reused as subflows, from inside other flows.

Here is a useful analogy to keep in mind when discussing subflows: a subflow is similar to a method call in Java, where one method invokes another method, for instance:

```
public void foo() {
  bar();
}
```

As you can see, method `foo()` calls into method `bar()`. Similarly, a `foo-flow` can call into another flow, `bar-flow`, using a subflow state:

```
<subflow-state id="launchBar" flow="bar-flow">
  <transition on="end" to="someState"/>
</subflow-state>
```

A subflow state is an instantiation of the `org.springframework.webflow.engine.SubflowState` class, a subclass of `TransitionableState`. Since a subflow state is transitionable, it can contain any number of transition definitions. These transitions will fire