3
The Basics of Spring Web Flow 2

The central element of Spring Web Flow 2 (SWF2) is the flow. A flow is described inside a flow definition. There are two ways to describe a flow:

- The declarative definition: Usage of XML with XSD (XML Schema Definition) grammar (`http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd`).
- The programmatic definition: Usage of Java.

In a standard Spring Web Flow 2 distribution, the `http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd` XSD is inside the `org.springframework.webflow-2.x.x.RELEASE.jar` library. The concrete location inside the mentioned library is the `org/springframework/webflow/engine/model/builder/xml` directory.

The usage of XML is more than recommended because the usage of Java for the flow definition has no direct support through a tool. Additionally, it is just not relevant for daily usage. If you want to use Java, you have to deal directly with the internal API of Spring Web Flow. We set our focus to show you the application development with Spring Web Flow and therefore we don't show the programmatic definition of a flow. In the case of the usage of the declarative approach, you can change the flow without changing your code for your application.

Continuations – Restore and Resume

A **continuation** represents the rest of the computation given a point in the computation. Another word for "rest of the computation" is control state, meaning the data structures and code needed to complete a computation. Most languages implement the data structure as a variant of the stack and the code as just a pointer to the current instructions. There are also programming languages that represent the control data as a heap-allocated object (see the reference: `http://en.wikipedia.org/wiki/Continuation`).

One of the important reasons for using the Spring Web Flow 2 framework is the fact that a use case mostly consists of more than one page. Therefore, there is a need for an interruption, maybe in case you need to wait for the input of the user. At the point of interruption, the system has to serialize the instances between the requests. Continuations are in order for the mechanism to restore and resume at a later point. Additionally, it helps to make interactions inside a web application easier. The Spring Web Flow framework does the work and therefore we have nothing to do with the infrastructure for continuations.

Elements of a flow

We mentioned before that the description of a flow is mostly done in an XML file. We named it the declarative definition. The concrete naming of that file is **flow definition file**.

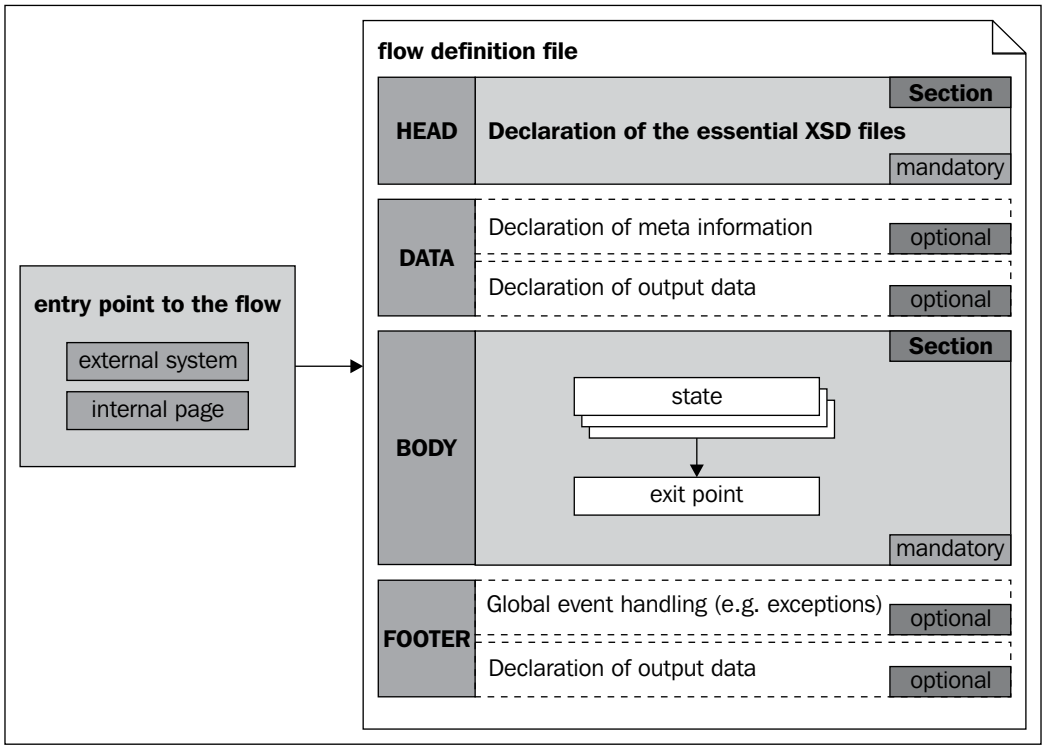
Remark: For a better understanding of the internals, it could be helpful to check the source code from Spring Web Flow 2.

You can think of a flow definition as a state machine, because it consists of states and transitions between them. Besides these two concepts, there are actions which trigger the transitions between the states.

A flow definition file consists of the following main elements:

- Section head
- Section data (optional section)
- Section body
- Section footer (optional section)

We can visualize the central block of that flow definition file in the following figure:



The entry point to the flow

In the previous figure, there is a section which is not directly described through the flow definition file, **the entry point to the flow**. There are two general types of entry points:

- Requests from an external system
- Requests from an internal page

A flow can have more than one entry point, because an entry point is just a request to the URI of the flow. In our example, the entry for the `add` flow is in the `view-issues.jsp` page.

Section head

The flow definition file is based on an XSD file. It is important to use the XSD file of version 2.0 because of the fact that the XSD has dramatically changed from version 1.0 to version 2.0.

One of the significant changes from version 1.0 to version 2.0 of the Spring Web Flow framework is that the file size of a flow definition file is reduced to about 50%. For developers who come from the old version, there is an automatic tool for upgrading from the old to the new version. To use the automatic upgrade feature, just execute the following command in the command prompt:

```
java org.springframework.webflow.upgrade.
WebFlowUpgrader flow-to-upgrade.xml
```

The URLs for both the versions of the XSD files are shown in the following table.

Version	URL
1.0	`http://www.springframework.org/schema/webflow /spring-webflow-1.0.xsd`
2.0	`http://www.springframework.org/schema/webflow /spring-webflow-2.0.xsd`

The main element of the definition file is `flow`. To use the `webflow` elements, you have to declare the namespace (with the help of the mentioned XSD file). See the line below for a short example of the namespace definition.

```
<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">
```

If you are using an editor (e.g. Eclipse) with an integrated validation for the syntax of the XML file, you will need access to the Internet to validate against the declared XSD schema in the header of the flow definition file.

Section data

There are two types of data that could be provided or handled inside a flow. There is some metadata for a flow. Additionally, there is a way of providing a flow with some input data from outside.

The metadata of a flow

The metadata of a flow is the section where you can specify attributes for configuring your flow, for example you configure whether the data inside the flow is persisted or not. The metadata is not directly related to the concrete use case which is described through the specific flow.

Persistence context

Most of the applications today are a variation of the classical CRUD (Create – Read – Update – Delete) pattern. That means each application has to handle information (or data) in a specialized way. For that issue, Spring Web Flow provides the persistence context which helps you to simplify the database access.

If you are interested in more information about ORM (Object-relational mapping), start with the article at `http://en.wikipedia.org/wiki/Object-relational_mapping`.

The nature of web programming is to manage concurrency in a multi-user environment. Therefore, there is often the necessity to manage more than one user and therefore we are confronted with more than one read/write process on the underlying data layer. One cornerstone in the management of concurrency is the usage of transactions. The new Spring Web Flow release offers possibilities to handle the mentioned transactions for the developer.

The transactions may be demarcated in presentation layer or service and DAO layers, even inside a database in the form of stored procedure.

In the latest release of Spring Web Flow 2, there is one implementation of the Persistence Context—the so-called *FlowScoped Persistence Context*.

More implementations for the Persistence Context are scheduled for the next major release of Spring Web Flow. There is one issue for view state persistence context, which can be tracked with the `http://jira.springframework.org/browse/SWF-577` URL. Another issue is for the conversation-scoped persistence context, which can be traced with the `http://jira.springframework.org/browse/SWF-567` URL.

FlowScoped Persistence Context

The concept of flow-managed persistence is inspired by the Hibernate Long Session. The changes are only committed at the end of the flow. This pattern is often used in conjunction with an **optimistic locking strategy** to protect the integrity of data modified in parallel by multiple users.

Before we can start working with FlowScoped persistence, we have to mark the flow description (see the following listing which shows the flow configuration) with `persistence-context`, or you can set the `persistence-context` attribute to `true`. It doesn't matter which of the two ways you choose. The authors think that the first one is more intuitive.

Direct usage of persistence-context

```
<?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">
    <persistence-context />
</flow>
```

Usage of the persistence-context attribute

```
<?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">
    <attribute name="persistence-context" value="true" />
</flow>
```

Additionally, we have to write the configuration of the beans to the configuration of the flow (often done in the `applicationContext.xml` file). In that mentioned configuration file, we have to define a listener for the underlying persistence technology. If you want to use the JPA (Java Persistence API), you have to define a `JpaFlowExcecutionListener` (see the listing below), or for Hibernate you have to define `HibernateFlowExecutionListener`.

```
<webflow:flow-executor id="flowExecutor" flow-registry="flowRegistry">
    <webflow:flow-execution-listeners>
```

```
        <webflow:listener ref="jpaFlowExecutionListener" />
    </webflow:flow-execution-listeners>
</webflow:flow-executor>

<bean id="jpaFlowExecutionListener"
     class="org.springframework.webflow.persistence.
JpaFlowExecutionListener">
    <constructor-arg ref="entityManagerFactory" />
    <constructor-arg ref="transactionManager" />
</bean>
```

Just for completion, we show the definitions for two beans, `entityManagerFactory` and `transactionsManager`.

```
   <!-- Drives transactions using local JPA APIs -->
   <bean id="transactionManager" class="org.springframework.orm.jpa.
JpaTransactionManager">
      <property name="entityManagerFactory" ref="entityManagerFactory"
/>
   </bean>

   <!-- Creates a EntityManagerFactory for use with the Hibernate JPA
provider and a simple in-memory data source populated with test data
-->
   <bean id="entityManagerFactory" class="org.springframework.orm.jpa.
LocalContainerEntityManagerFactoryBean">
      <property name="dataSource" ref="dataSource" />
      <property name="jpaVendorAdapter">
         <bean class="org.springframework.orm.jpa.vendor.
HibernateJpaVendorAdapter" />
      </property>
   </bean>
   <!-- Deploys a in-memory "booking" datasource populated -->
   <bean id="dataSource" class="org.springframework.jdbc.datasource.
DriverManagerDataSource">
      <property name="driverClassName" value="org.hsqldb.jdbcDriver"
/>
      <property name="url" value="jdbc:hsqldb:mem:flowtrac" />
      <property name="username" value="sa" />
      <property name="password" value="" />
   </bean>
```

It is worth mentioning that HSQLDB (`http://hsqldb.org/`) is more than simply a memory DataSource. It is a relational database which is completely compatible with Java.

At the end of the flow, we have to trigger the commit of the data changes. For that, we have the `commit` attribute, which can be set to `true`.

```
<end-state id="transactionConfirmed" commit="true" />
```

Those who are familiar with the JPA might now ask about the EntityManager (see the reference to `entityManagerFactory` in `applicationContext.xml`). Its instance will be created at the start of the flow with a listener. That EntityManager can be referenced in the flow configuration (use the `persistenceContext` variable). Additionally, each access on data, which is managed through the Spring framework, is using that EntityManager.

The internal realization of the concept of Persistence Context is done through `FlowExecutionListener` (package: `org.springframework.webflow.execution`).

Section input

It is possible to provide a flow with some data. The data can be categorized into two sections:

- Variables which are created when the flow starts. These types of variables are called flow instance variables. These variables are changed typically in a programmatic way.
- Inputs which are provided from the caller of the flow. These variables are set through the request of the flow.

There is a difference between the definition of the flow (in XML or Java) and the programming of the flow. You can use XML or Java for the flow definition but with both of them you describe (or program) the flow with a specific DSL (**Domain Specific Language**). The name of that specific language is flow definition language. For the access of data in the different scopes (see the following section for the description of the different scopes), a special syntax is necessary.

If you want to read more about the concept of Domain Specific Language, start with the article at `http://en.wikipedia.org/wiki/Domain-specific_programming_language`.

Programming in a flow

There are four places where you need more than simple XML in a flow. These are:

- access of the data provided from the client (see the *Input* section)
- access to data structures in the different scopes (see the *The scopes* section)
- invoking actions, especially methods on declared beans
- creating some control structures (for example, for defining a decision)

Spring Web Flow 2 uses an EL (Expression Language) for the programming logic inside the flow. The framework supports two different types of expression languages. These variants are described in the subsequent table:

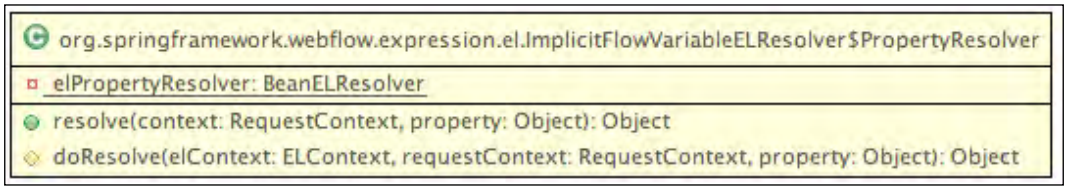
Name	Description
Unified EL	The Unified EL is used by default in Spring Web Flow. As implementation library, the JBoss EL (download it from: `http://www.springsource.com/repository/app/bundle/detail? nme=com.springsource.org.jboss.el`) is used. For more information on Unified EL, visit `http://java.sun.com/products/jsp/reference/techart/unifiedEL.html`.
OGNL	OGNL is the acronym for Object-Graph Navigation Language. It is used as the default EL in version 1 of Spring Web Flow. For more information about OGNL, visit `http://www.ognl.org`.

Most of the syntax of both the implementations is the same. It is recommended to use the standard and not the proprietary syntax of OGNL (only if you need some special things from that). Inside the DSL of the flow definition, there can be two types of expressions. The difference is whether the expression has a delimiter (e.g. `${}` or `#{}`) or not. The expressions without a delimiter are called **standard eval expressions**. An example is `action.execute()`. If you use a delimiter, a run-time exception from type `java.lang.IllegalArgumentException` is thrown. The second type of expression is the template expression. For that, you have to use a delimiter. For example, `my-${name.locale}.xml`. The result of the template is text. Most of the attributes in the latest Spring Web Flow 2 schema accept expressions. Check the schema at `http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd` to see whether an attribute accepts it or not. The following implicit variables are available inside the EL expressions:

- `currentEvent`
- `currentUser`
- `externalContext`

- `flowExecutionContext`
- `flowExecutionUrl`
- `flowRequestContext`
- `messageContext`
- `resourceBundle`
- `requestParameters`

The internal implementation of these variables is done with the help of the `ImplicitFlowVariableElResolver` class inside the `org.springframework.webflow.expression.el` package. Dependent on the variable, there are specific implementations of the internal `PropertyResolver` class (see the class diagram below).



Now we will describe the variables and their usage:

Variable currentEvent:

With the `currentEvent` variable, you can access data of the current event, which is last processed by the active flow. The instance of the event is from the `Event` class (package `org.springframework.webflow.execution`).

Variable currentUser:

With the `currentUser` variable, you have access to `Principal` (the principal is the authenticated user) who is authenticated through the web container. For example, with `principal.name`, you access the name of the principal.

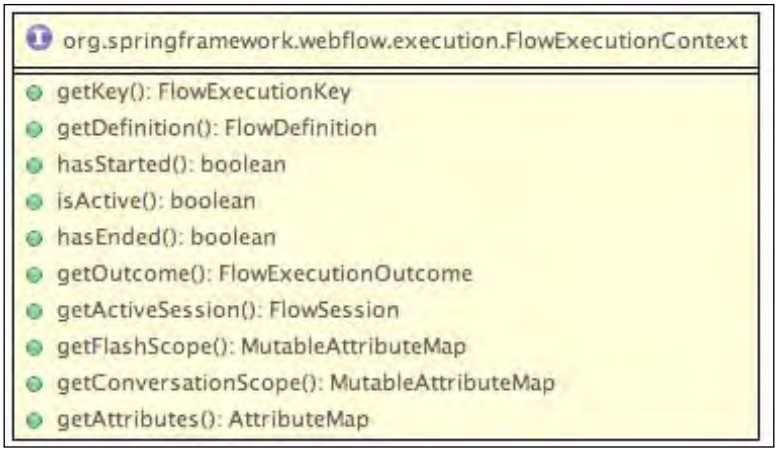
Variable externalContext:

Sometimes it is necessary to interact with the environment (e.g. the session object). For that case, you can use the `externalContext` variable.The instance is from type `ExternalContext` (package `org.springframework.webflow.context`). For a complete overview, see the following class diagram. You can understand that the ExternalContext is a façade which provides access to an external system that has called the Spring Web Flow system.

In case a servlet calls the Spring Web Flow system, you get an instance from the `ServletExternalContext` type (package `org.springframework.webflow.context.servlet`). It is worth mentioning that there is a subclass for `ServletExternalContext` which is used in the case of Spring MVC. Its name is `MvcExternalContext` (package: `org.springframework.webflow.context.mvc.servlet`). In the case of a portlet environment, the concrete class is `PortletExternalContext` (package: `org.springframework.webflow.context.portlet`).

Variable flowExecutionContext:

The `flowExecutionContext` variable allows you to get access to an instance of the underlying `FlowExecutionContext` (package `org.springframework.webflow.execution`). See the following diagram for the methods:

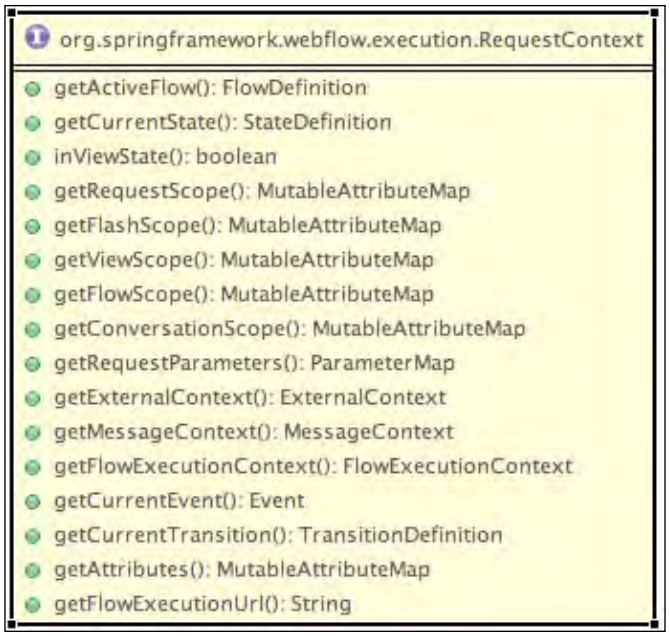


Variable flowExecutionUrl:

The developer has access to the context-relative URL of the flow with the help of the `flowExecutionUrl` variable. In the default implementation, the URL is created inside the `createFlowExecutionUrl` method of the `DefaultFlowUrlHandler` class (package `org.springframework.webflow.context.servlet`). In the default implementation, the flow execution key will be attached to the URI of the request as a parameter (name: `execution`).

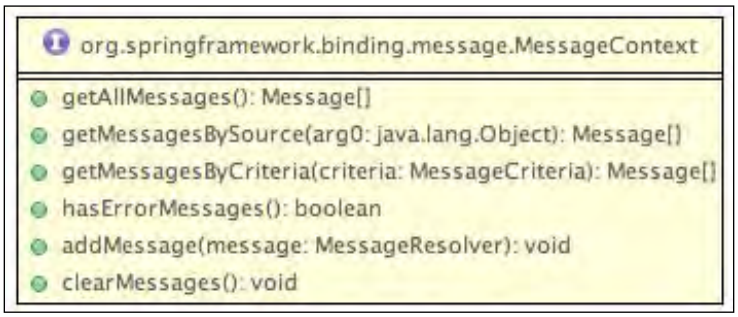
Variable flowRequestContext:

With the `flowRequestContext` variable, the developer gets access to the actual instance of the `RequestContext` inside the `org.springframework.webflow.execution` package. The methods are shown in the following diagram:



Variable messageContext:

The `messageContext` variable gives us the privilege to create and retrieve messages from the flow execution (e.g. error and success messages). The instance is from the `MessageContext` type in the `org.springframework.binding.message` package. For the methods, view the following diagram:



Variable resourceBundle:

To access a message inside the `messages.properties` file inside the `flow` directory, use the `resourceBundle` EL variable, for example `resourceBundle.name`. The following line shows a small example for the usage:

```
<set name="flashScope.ErrorMessage" value="resourceBundle.
myErrorMessage" />
```

It's possible to use that EL variable inside the views, too. See the example below for a demonstration:

```
<h:outputText value="#{resourceBundle.myErrorMessage}" />
```

Variable requestParameters:

The `requestParameters` variable offers the possibility to access the parameters of the incoming request, for example `requestParameters.issueId`.

The scopes

The data inside a flow can occur in different scopes. In short, a scope describes the duration for which the data is valid. Now, we introduce the different scopes for a better understanding.

Name	Description
`flow`	On the start of a flow, the `flow` scope is allocated and it is destroyed after the flow ends. Therefore, the `flow` scope lives as long as the flow and is the right place if you want to store data that should be accessible during the complete flow execution. The objects in the flow scope have to implement `java.io.Serializable`. To access the flow scope in the flow definition, use the EL variable `flowScope`. It's not possible to access the data in the subflow.
`view`	On the start of a `view-state` (states are explained later in the chapter; for understanding it is important to know that the lifecycle of a flow is fragmented into different states), the `view` scope is created. That scope is destroyed after the `view-state` ends. It is important to know that the view scope is only accessible inside a `view-state`. The objects in the `view` scope have to implement `java.io.Serializable`. To access the `view` scope in the flow definition, use the EL variable viewScope.
`request`	Variables which come in a request are stored inside the `request` scope. That scope is created when the flow is called and destroyed after the flow returns. To access the `request` scope in the flow definition, use the EL variable `requestScope.`

Name	Description
`flash`	The `flash` scope itself is created on the start of the flow. The cleanup is done after every view rendering. The destruction is done at the end of the flow. The objects in the `flash` scope have to implement `java.io.Serializable`. To access the `flash` scope in the flow definition, use the EL variable `flashScope`.
`conversation`	A conversation (and therefore the flow) spans a complete flow. It is created on the flow start and destroyed at the end of the flow. The `conversation` scope is valid inside the subflows too. That is the only difference between the `conversation` and `flow` scope. To access the `conversation` scope in the flow definition, use the EL variable `conversationScope`.

For a fast overview of the duration of the scopes, have a look at the following table:

Scope	Created	Cleaned	Destroyed
request scope	call of a flow	-	flow returns
view scope	enter view state	-	exit the view
flash scope	flow start	after a view render	end of flow
flow scope	flow start	-	flow end
conversation scope	(top-level) flow start	-	flow end

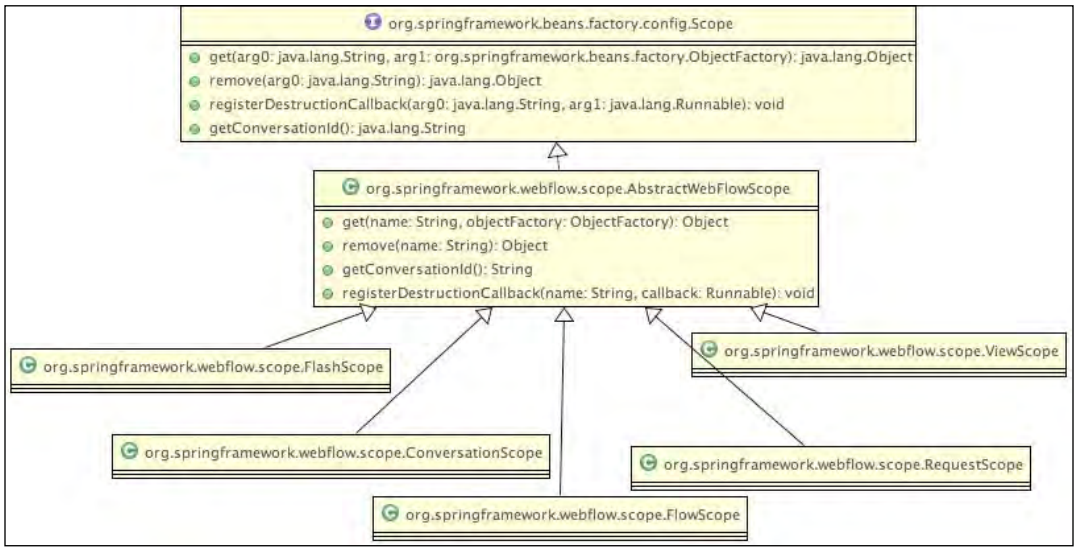
In some situations, you often need an instance from a class in a clean state. Therefore, if the cleanup of the instance is as fast as the creation of a new instance, it could be preferable to clean the instance. In that case, you lower the memory footprint of your application.

The scopes are sections to store the data. How to access these data is described in the following section.

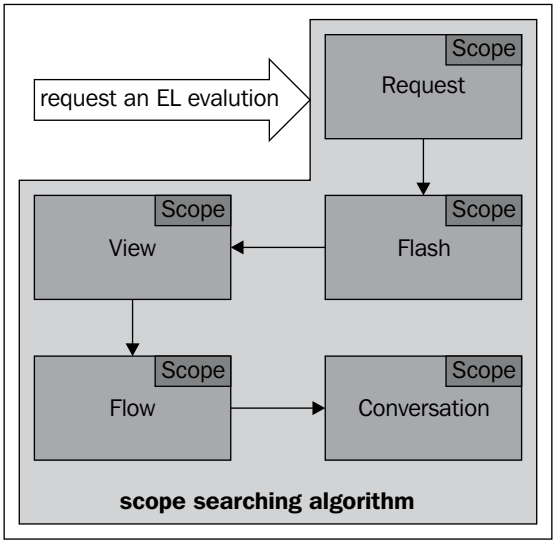
Resolving variables

There are two ways to access the variables. You can address a variable with `Scope` as its prefix (e.g. `flashScope.name`) or without a prefix (e.g. `name`). If you use a prefix, the specified scope is searched for the variable. If no scope prefix is specified, a scope search algorithm (implemented in `org.springframework.webflow.expression.el.ScopeSearchingElResolver`) is used. In simple words, a scope (base interface is `org.springframework.beans.factory.config.scope.Scope`) is just a special `java.util.Map`. All scopes extend the abstract class named `org.springframework.webflow.scope.AbstractWebFlowScope`.

For a complete overview, we illustrated this in the class diagram below:



The scope searching algorithm is visualized in the figure below:



Request is not HttpServletRequest

If we talk about a request inside a flow, we talk about a single call (inside a thread) by an external actor. The representation for a request is the `RequestContext` class inside the `org.springframework.webflow.execution` package. If you work with Spring MVC, don't confuse it with the `RequestContext` class inside the `org.springframework.web.servlet.support` package. The actual instance (request specific) of `RequestContext` can be accessed through the `RequestContextHolder` class in the `org.springframework.webflow.execution` package. The call is `RequestContext context = RequestContextHolder.getRequestContext()`.

The flow instance variables

To handle data inside the flow, it is necessary to have a variable. Such a variable is defined in the data section of the flow. The name of the XML tag is `var`. You can define as many flow instance variables as you want. The limitation is only the memory of your machine.

Name	Description
`Name`	The name of the instance variable. Please take care that the name is unique, otherwise you can get unpredictable errors.
`Class`	The class which should be used for the variable. It is important that the class implements the `java.io.Serializable` marker interface because it is saved between the requests of a flow and therefore the instances are serialized. Additionally, it is important that the class should have a public default constructor, because the default implementation uses it to instantiate `SimpleInstantiationStrategy` (package: `org.springframework.beans.factory.support`) and that strategy searches for a public default constructor with the `clazz.getDeclaredConstructor((Class[]) null)` call.

In the following example, a flow instance variable with the name `issue` and the `flowtrac.core.model.issue.IssueImpl` class is shown.

```
<var name="issue" class="flowtrac.core.model.issue.IssueImpl" />
```

It is important to know that if you use a complex object which has some transient member variables which are annotated with `@Autowired`, these variables are rewired after the flow resumes.

The variables are created on the start of the flow. It is not mandatory to define flow instance variables.

Assign a value to a scope variable

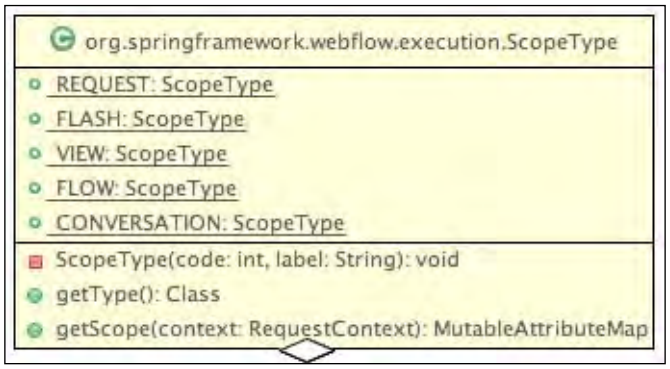
Instead of defining the flow instance variables with the `var` tag, there is an option to set a specific value inside the scope. For that, you can use `set`.

```
<set name="requestScope.issueId" value="requestParameters.id"
type="long" />
```

To advise Spring Web Flow that a conversion is essential, you have to specify the `type` attribute. The framework checks the value and does a type conversion if it is necessary.

Access the value of a scope

If you want to access the value of a scope inside the code, you have to use the `ScopeType` class inside the `org.springframework.webflow.execution` package. That class contains enumerations for each type of scope (see the following figure). To access the attributes of a scope, there is a method called `getScope` with the `RequestContext` parameter.



For convenience, we implemented a static method called `getFromScope` which can be placed in the helper class (in our example, we named it `ScopeValueHelper`). With that method, it is easy to obtain a value of the specific scope. You have to provide the name of the value, the type of the value, and the scope.

```
   public static <T> T getFromScope(final String name, final T
typeToConvert, final ScopeType scope)   {
      if (scope == null)  {
         throw new ScopeValueHelperException("A scope must be set.");
      }
      RequestContext context = RequestContextHolder.
getRequestContext();
      if (context == null)  {
         throw new ScopeValueHelperException("Could not retrieve the
```

```
context of the actual request. Please take care that you use that
class inside a flow.");
      }
      T value = null;
      MutableAttributeMap map = null;
      map = scope.getScope(context);
      if (map == null)   {
         throw new ScopeValueHelperException("The scope " + scope + "
has no value map.");
      }
      try  {
         value = (T) map.get(name);
      } catch (ClassCastException cce) {
         throw new ScopeValueHelperException("The value in the scope
is not from the correct type.",cce);
      }
      return value;
   }
```

An example for the usage of that method is shown in the following line:

```
boolean create = ScopeValueHelper.getFromScope("create", create =
false, ScopeType.REQUEST);
```

Additionally, we added a method which searches in all scopes (using the same scope searching algorithm like Spring Web Flow). The name of the method is also `getFromScope`; however you only have to provide the name of the value and the type. The implementation of the algorithm is very easy. We just have a list with the scopes inside (see below).

```
static List<ScopeType> scopeSequence;
static  {
   scopeSequence = new LinkedList<ScopeType>();
   scopeSequence.add(ScopeType.REQUEST);
   scopeSequence.add(ScopeType.FLASH);
   scopeSequence.add(ScopeType.VIEW);
   scopeSequence.add(ScopeType.FLOW);
   scopeSequence.add(ScopeType.CONVERSATION);
}
```

The list is used for searching like the scope searching algorithm does. The implementation of the method is shown below.

```
   public static <T> T getFromScope(final String name, T
typeToConvert)  {
      RequestContext context = RequestContextHolder.
getRequestContext();
```

```
      if (context == null)  {
         throw new ScopeValueHelperException("Could not retrieve the
context of the actual request. Please take care that you use that
class inside a flow.");
      }
      T value = null;
      MutableAttributeMap map = null;
      for (ScopeType type: scopeSequence)   {
         map = type.getScope(context);
         if (map == null)   {
            throw new ScopeValueHelperException("The scope " + type +
" has no value map.");
         }
         try  {
            value = (T) map.get(name);
         } catch (ClassCastException cce) {
            throw new ScopeValueHelperException("The value in the
scope is not from the correct type.",cce);
         }
         if (value != null)   {
            return value;
         }
      }
      return null;
   }
```

The usage is similar to the usage of the other `getFromScope` method. For completeness, we show it now:

```
boolean create = ScopeValueHelper.getFromScope("create", create =
false);
```

Inputs

It is possible to provide a flow with some external data. For example, you specify the parameter in a GET or POST request. For that case, you have to use the `input` tag. Besides the name, you have to specify the `id`. That `id` is the name of the provided variable.

For example, we have a link for editing the issues:

```
<a href="/flowtrac-web/flowtrac/issue/add?id=${issue.id}" />
```

As you can see, the flow is started with the `id` parameter. To use that parameter, we define an `input` parameter with the name `id`.

```
<input name="id" />
```

Additionally, it is possible to specify the type of the input parameter. The name of the attribute is `type`. For example:

```
<input name="id" type="long" />
```

If an input value does match the specified type, a type conversion will be attempted. The conversion is done in `org.springframework.webflow.engine.builder.model.FlowModelBuilder`. If the conversion could not be done, the `org.springframework.binding.convert.ConversionExecutionException` run-time exception is thrown. The last possible (and optional too) attribute is `value`. With this parameter, you can map the input parameter to a value in a specific scope, as shown below.

```
<input name="id" value="flowScope.issue.id" />
```

The following table sums up the information.

Name	Description
`Id`	The name of the input variable.
`Type`	The type of the input variable. If you specify this, an automatic type conversion is attempted. The attribute is optional.
`Value`	With that optional attribute, you can map an input parameter to a flow variable.

The states

There are six states that can occur inside a flow definition. The states are:

- `start-state`
- `action-state`
- `view-state`
- `decision-state`
- `subflow-state`
- `end-state`

It is possible for a state to inherit configuration information from another state. For that, use the `parent` attribute. For more information about inheritance in the flow definition, see the section about inheritance.

The start-state

When the flow starts, there must be a state which should be executed first. For that, you have a choice between an explicit `start-state` and an implicit `start-state`. In our example, the first state which occurs is the `add` view-state.

```
<view-state id="add" model="issue">
```

In the above case, we choose the implicit way. That means that if you do not specify the `start-state`, the first state which occurs in the flow definition file is the `start-state`. The other way is the explicit way. For that, you have to specify on the `flow` tag the `start-state` attribute. An example for that is shown below:

```
<flow start-state="add" />
```

As you can see above, the `start-state` is no concrete state, it is only a contract. And therefore, there is no explicit state. If you have developed applications with Spring Web Flow 1, you might remember that there was a tag called `start-state`.

The action-state and execution of business logic

If you want to do more than navigate, you have to execute some business logic. For that, in many frameworks the concept of actions exists, like in Spring Web Flow 2. There are two ways to express the evaluation of an action. There is the `action-state` tag. With that, you have an extra state for executing an action. To minimize your flow definition, it is recommended to use the `evaluate` element inside your `view-state`.

The tag for executing an action is `evaluate`. The attributes for the tag are described in the following table.

Attribute	Description
`expression`	The expression to evaluate. Here, you can execute an action on each bean which is accessible inside the flow. You have to use the expression language for expressing what to execute. That attribute is mandatory.
`result`	With that optional attribute, you have the possibility to assign the result to a variable in a specific scope.
`resultType`	The type of the result.

Additionally, it is possible to provide the `evaluate` tag with attributes. That can be done with the `attribute` child tag . But first, an example for the usage of `evaluate` without `attribute`.

```
<evaluate expression="issueService.findById(id, true)"
result="flowScope.issue" />
```

For an example of the usage of the `attribute` child element for `evaluate`, we provide the second parameter for the `findById` method as `attribute`.

```
<evaluate expression="issueService.findById(id)" result="flowScope.
issue">
  <attribute name="create" value="true" type="boolean"></attribute>
</evaluate>
```

The attributes are provided inside `RequestContext`. That means we need access to the actual instance of `RequestContext` inside the code. For convenience, we wrapped the `ScopeValueHelper` helper class inside the `flowtrac.swf.extension.scope` package.

```
public static <T> T getFromRequestContext(final String name, T
typeToConvert) {
   //Get the attributes from the actual request
   MutableAttributeMap map = RequestContextHolder.getRequestContext().
getAttributes();
   if (map == null) {
      throw new AttributeHelperException("Could not retrieve the
attributes from the actual request.");
   }
   T value = null;
   try  {
      value = (T) map.get(name);
   } catch (ClassCastException cce) {
      throw new AttributeHelperException("The value in the request is
not from the correct type.",cce);
   }
   return value;
}
```

With this method, we can now easily access the attributes in the action methods of the code. For an example, we show the following lines of the implementation of the `findById` method.

```
public Issue findById(Long id) {
      boolean create = ActionAttributeHelper.get("create", create =
false);
      return findById(id, create);
   }
```

The previous example uses the `create` attribute which is provided through the flow definition file. In the following table, we want to give an overview of the points where you could execute an action.

Point of Execution	Usage

At the start of the flow	There is a tag with the name `on-start`.
	In the case of our demo application, we used that way to search an issue. The following line shows an example for that:
	`<on-start>`
	`  <evaluate expression="issueService.findById(id, true)" result="flowScope.issue" />`
	`</on-start>`

At the entry of a state	Inside a state, there is the `on-entry` tag.
	If you use that element, the code inside the `on-entry` tag is executed first.
	A small example:
	`<view-state …>`
	`  <on-entry>`
	`<evaluate …>`
	`  </on-entry>`
	`</view-state>`

At the time of rendering a view	Inside a state, there is the `on-render` tag.
	If you use that element, the code inside the `on-render` tag is executed just before the view of the concerned.
	A small example:
	`<view-state …>`
	`  <on-render>`
	`<evaluate …>`
	`  </on-render>`
	`</view-state>`
	The difference between the `on-entry` tag and the `on-render` tag is that the latter one is executed every time the view is rendered. A use case could be that every time you render a view you want to create a new instance from the database.

At the time a transition is executed	It is possible to execute an action on time when a transition is executed. We just show a small example for that:
	`<transition on="store" to="issueStore">`
	`<evaluate expression="persistenceContext.persist(issue)" />`
	`</transition>`
	If the method returns `false`, the transition is not executed and the old view is presented again.

Point of Execution	Usage

On the exit of a state	Inside a state, there is the `on-exit` tag.
	If you use that element, the code inside the `on-exit` tag is executed when the state exits.
	A small example:
	`<view-state …>`
	`  <on-exit>`
	`<evaluate …>`
	`  </on-exit>`
	`</view-state>`

At the end of a flow	There is a tag with the name `on-end`.
	In the case of our demo application, we used that way to add some data to the issue. The following line shows an example for that:
	`<on-end>`
	`  <evaluate expression="itemInformationService.enrich(issue)" />`
	`</on-end>`

Details of a transition

A transition is the process of transfer from one state to another. Transitions are triggered through events. There are two types of transitions internally:

- Transitions inside an `action-state`
- Transitions inside a `view-state`

Transitions inside an action state and a view state are both equally relevant for you as a developer.

You can understand a transition as a glue for the views, because to navigate from one view to another a transition is needed.

We describe all parameters in the following table. We also mention if a parameter can only occur in one state.

Attribute name	Description	Available in
`bind`	Indicates whether model binding should occur before the state is transferred or not. The value is optional and is of type `boolean` (`bind="true"`). Without model binding, the validation is prevented, but you have to bind the values manually if you want to use the values further in your use case. The default value is `true`.	`view-state`
`history`	This attribute's value (optional) sets the history policy for this transition. That means that this setting decides whether you can use the back button of the browser or not. The following values are possible: • `preserve` (default value) • `discard` • `invalidate` The `preserve` value means that back-tracking to the current state (after the transition) is possible. The `discard` value prevents back-tracking to the actual state and `invalidate` prevents back-tracking to that state and any previously entered `view-state`.	`view-state`
`on`	The name of the outcome on which the transition is triggered. It is simple text.	Both
`on-exception`	The name of the exception (fully qualified, for example `java.lang.NumberFormatException`) on which the transition is triggered. It is not possible to mix `on` and `on-exception`.	Both
`to`	The name of the destination state.	Both

The view-state

The following table shows the attributes of a `view-state`:

Attribute	Description
`id` **(mandatory)**	The identifier for the state. This identifier must be unique inside the flow.
`parent` **(optional)**	The name of the parent of the state (see the *Inheritance for states* subsection for more information about this feature).

Attribute	Description

`view`
(optional)
	The name of the view. If you do not specify the name, the `id` of the `view-state` is used as the basename for the view file. That file has to be in the same directory as the flow definition file. If you define a view, the view has to be in the same directory as the flow definition file. A simple example is shown in the line below:
	`<view-state id="add" view="addIssue.jsp" />`
	If you want to show a view outside your application, there is the `externalRedirect` prefix.
	It is possible to use expressions with `${}` format. The supported formats for `externalRedirect` are:
	`externalRedirect:`<servlet relative path>
	`externalRedirect:contextRelative:`<context relative path>
	`externalRedirect:serverRelative:`<server relative path>
	`externalRedirect:`<fully qualified http:// or https:// URL>
	For example:
	• `externalRedirect:contextRelative:/flowtrac/issue/all`
	• `externalRedirect:contextRelative:/flowtrac/issue/all?id=${issue.id}`
	The `flowRedirect`: prefix may be used to redirect to another flow:
	`flowRedirect:myOtherFlow?someData=${flowScope.data}`
	When this attribute is not specified, the view to render will be determined by convention. The default convention is to treat the `id` of this `view-state` as the view identifier. For exotic usages, you may plug in a custom `ViewFactory` bean.

`redirect` **(optional)**	With this attribute (the value is of type Boolean), the `view-state` sends a flow execution redirect before the view is rendered. This feature is essential if you need a refresh after event processing. Internally, it is done from `FlowHandlerAdapter` (package `org.springframework.webflow.mvc.servlet`) in the `sendFlowExecutionRedirect` method. The steps to do that redirection are first to elaborate the URL for the redirection and then to redirect to the flow in the actual state.
`popup` **(optional)**	With this attribute (the value is of type Boolean), the view is displayed in a pop-up dialog. For example: `<view-state id="test" popup="true" />`
`model` **(optional)**	With this attribute, it is possible to specify a model that can be bound to the properties of that view. `<view-state id="add" model="issue" >` The model can be in any accessible scope, for example in the `flow` scope. In our sample application, we add to the `flow` scope (`<evaluate expression="issueService.findById(id)" result="flowScope.issue">`)

The current release of Spring Web Flow 2 (2.0.4 at the time of writing) supports two technologies for the view. On the one hand, there is the possibility to use plain JSP, and on the other, JSF facelets (check `https://facelets.dev.java.net/`). The technology for the view does not relate to the controller you use. This means you do not have to use Spring MVC as a controller framework. But the integration with Spring MVC is highly recommended because it is very seamless.

Validation inside the view-state

We have shown that the `view-state` has the optional `model` attribute, which is responsible for binding the data of a form to an internal model class. If we talk about the input of user data, there is often the need for a validation. Spring Web Flow offers us two ways for server-side validation. These two are:

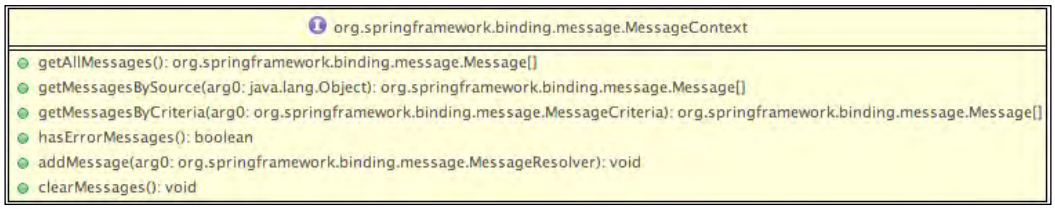
- Implement a validate method inside the `model` class
- Implement an encapsulated Validator

The first way of validation is the mentioned implementation of a validate method inside the `model` class. The name of the method begins with the name `validate` appended with the state which should be validated (`validate${state}`). For example, if we have the following state:

```
<view-state id="add" model="issue">
```

we have to implement a method `validateAdd`. Additionally, we have to provide that method with a `MessageContext` (package `org.springframework.binding.message`).

The mentioned `MessageContext` is the central class for recording and retrieving messages for the case of display. See the class diagram below to get an overview about the methods.



An example implementation for this method (in our sample, we added it to the used model class `Issue`) is shown below.

```
public void validateAdd(MessageContext context)  {
   if (this.name == null || this.name.trim().length() == 0) {
      context.addMessage(new MessageBuilder().error().source("name").
defaultText("the name is required").build());
   }
}
```

To add messages to `MessageContext`, you have to provide an instance of a `MessageResolver` (package `org.springframework.binding.message`). To create such an instance, there is a class named `MessageBuilder` in the same package. This class has a fluent interface (the return values of the methods are the `MessageBuilder` instance). See the example above.

The second way for programmatic validation on the server side is to implement an own Validator class. The name of the method is the same as before. See the example implementation below.

```
@Component
public class IssueValidator {
   public void validateAdd(Issue issue, MessageContext context)  {
      if (issue.getName() == null || issue.getName().trim().length()
== 0) {
         context.addMessage(new MessageBuilder().error().
source("name").defaultText("validator: the name is required").
build());
      }
   }
}
```

If you use the `@Component` annotation (available in Spring 2.5 or above), the classpath-scanning registers the bean with the name `issueValidator`.

Which way should I take?

We have looked at the two ways which offer you the same functionality. The problem with the first way is that you mix the concerns. But the `model` class should only encapsulate the data and nothing else. Additionally, if you implement the method in your value object class, you get a dependency to the Spring Binding (library `org.springframework.binding-x.x.x.RELEASE.jar`). The implemention of an own validator class should be prefered to follow the concept of loose coupling.

More details about validation

To complete the validation, we want to provide a complete overview as to how the validation methods are called from the framework. The method which does the validation is a private method called `validate` (internally, the method calls the `validate` method of the `ValidationHelper` helper class inside the `org.springframework.webflow.validation` package) inside the `AbstractMvcView` class (package `org.springframework.webflow.mvc.view`).

Step 1: Create the name of the `validate` method. Append the name of the state to `validate` (capitalized, that means `add` becomes `Add`). The method name for the state `add` is `validateAdd`.

Step 2: Search for the method with the `MessageContext` parameter (package `org.springframework.binding.message`). If the method is found, it is called.

Step 3a: If a bean factory is available, a validator class is searched. This is done by appending the value of the model to `Validator`, for example `issueValidator`. If the bean is found, the next steps are executed.

Step 3b: A method with the same naming principle as in the above Step 3a is searched (additionally there is a parameter for the model first). If the method is found, it is executed.

Step 3c: If no method is found, there is a search for a method with a parameter called `Errors` (package `org.springframework.validation`). If the method is found, it is executed.

As you can see, it is possible to have two validator classes. First, the validate method inside the model class is executed (if available). Next, the method inside the validator (if available) is executed.

It is not possible to have more than one Validator class. If you have more than one, a run-time exception from the `java.lang.IllegalStateException` type on starting the application context is thrown.

Externalize your messages

Inside the validator, we have used `MessageBuilder` to create an instance of `MessageResolver`. We have used the `defaultText` method to express your error message. Additionally, there is a method callled `code` to provide an error key. This key is resolved by looking into the `messages.properties` file inside the directory of the flow. The `messages.properties` file is available inside the flow and also inside the views for that flow. Therefore, you can use it for internationalization too. By the way, it is possible to mix the `defaultText` and `code` methods (fluent interface). If no error key is found, the value of `defaultText` is used.

The decision-state

There are often cases when you want to know whether a view is rendered or not. For that case, there is a concept of `decision-state`. In the following example, we select a specific state inside the flow depending on whether the `simpleValue` variable (it comes from a request) is set or not.

```
<input name="simpleValue" value="flashScope.simpleValue" />
<decision-state id="check">
  <if test=" simpleValue != null" then="${simpleValue}" else="init" />
</decision-state>
```

The subflow-state

A subflow is a flow inside a flow. With this concept, it is possible to have a hierarchy of flows. You have one main (parent) flow and can execute some subflows.

A small example of a subflow:

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

More details about subflows are provided in Chapter 5 of this book. Besides subflows, there is a concept of flow hierarchy which is explained at the end of this chapter.

The end-state

To close a flow, you have to define an `end-state`. Just a small example from our sample application:

```
<end-state id="issueStore" commit="true" />
```

Additionally, it is possible to define an output for this flow.

```
<end-state id="issueStore" commit="true">
  <output name="name" value="issue.name" />
</end-state>
```

One important question is what to show after the flow ends. For that, you have the `view` attribute. With that, you can define the view which should be shown after the flow exits. One example from our example application:

```
<end-state id="issueStore" commit="true" view="externalRedirect:
contextRelative:/flowtrac/issue/all" />
```

The exit point

This is the page to go when the flow ends. Normally, it is defined in the `end-state`.

Section footer

The footer of a flow definition file can have the following elements:

- `global-transitions`
- `on-end`
- `output`
- `exception-handler`
- `bean-import`

The `on-end` event handler is described earlier in this chapter (in the *The action-state and execution of the business logic* section). The other elements are described below.

global-transitions: global handling of events

With the `global-transitions` element, you have the opportunity to define state transfers. Often applications have a global menu and the events can be handled globally. The child element of `global-transitions` is the `transition` element. Additionally, it is possible to use `global-transitions` in conjunction with the transitions as mechanism for global exception handling (see the *exception-handler: Exceptions between the execution of a flow* section below). An example for declaring `global-transitions` is shown below.

```
<global-transitions>
    <transition on="login" to="login">
    <transition on="logout" to="logout">
</global-transitions>
```

on-end: execution of actions at the end of the flow

Before the exit from a flow, it is possible to execute an action inside the `on-end` element. The following lines show an example for the usage of the `on-end` element.

```
<on-end>
  <evaluate expression="itemInformationService.enrich(issue)" />
</on-end>
```

output: output of the flow

Besides the option to declare the output for each `end-state`, it is possible to declare an output (the `output` element) for the flow. For example:

```
<output name="id" />
```

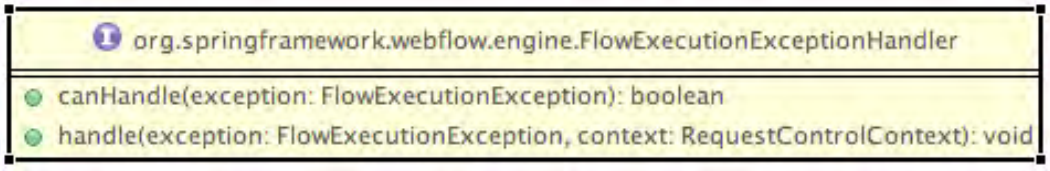
exception-handler: exceptions between the execution of a flow

Between the execution of a flow (or the states inside the flow), exceptions can occur. The default implementation throws the exception and the user is confronted with that exception. To prevent the situation of an unhandled exception which occurs inside a flow (or state), it is possible to register an own handler for exceptions.

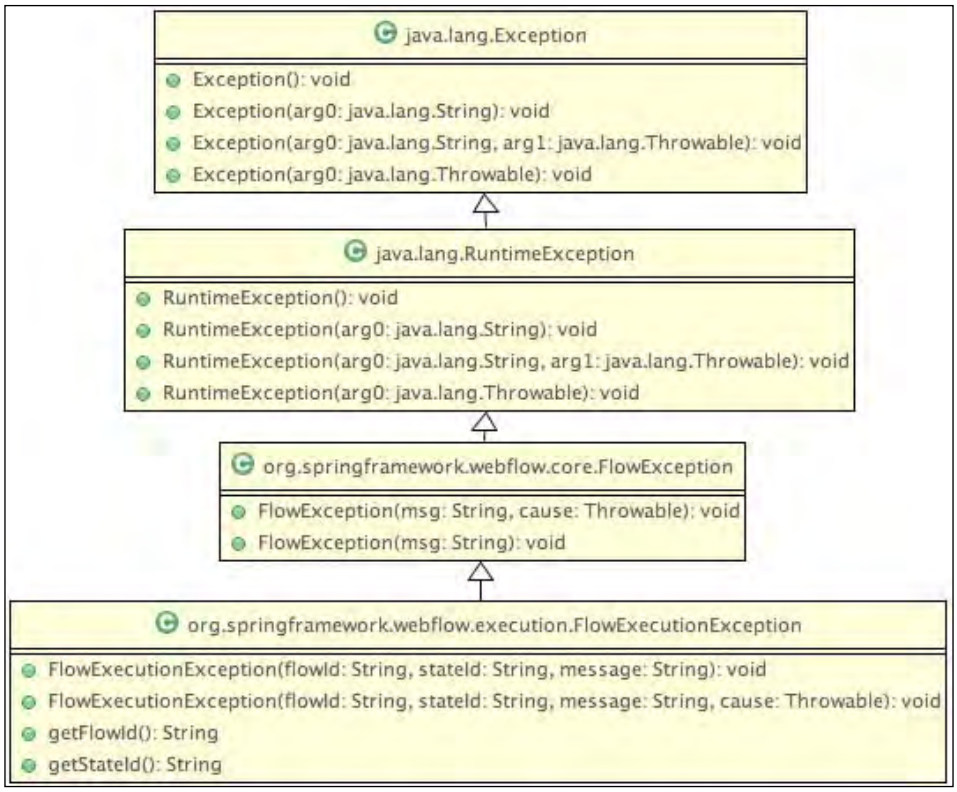
The following elements can have an own exception handler:

- `action-state`
- `decision-state`
- `end-state`
- `view-state`
- `flow`

The exception handler has to be from the `FlowExecutionExceptionHandler` type. The package of that interface is `org.springframework.webflow.engine`. The methods for that handler are shown in the following class diagram:



As you can see in the interface of `FlowExecutionExceptionHandler`, the general exception is `FlowExcecutionException` in the `org.springframework.webflow.execution` package. The class hierarchy of that exception is shown in the diagram below:

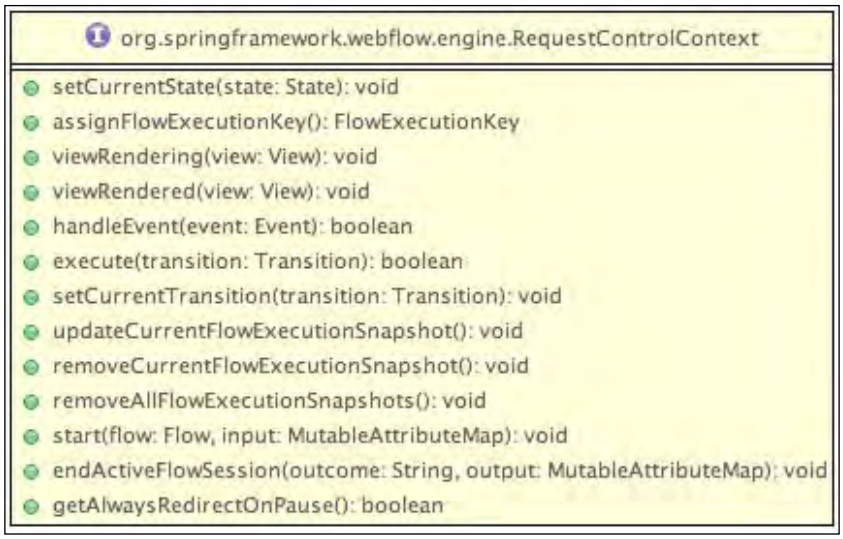


As you can see, the exception is from the `java.lang.RuntimeException` type and therefore it is an unchecked exception (that means you don't have to catch the exception). If an exception occurs, you can use the `getFlowId` method to get the `id` of the flow which has thrown the exception. Additionally, the `getStateId` method can be used for retrieving the `id` of the state in which the exception occurred. For handling a specific exception, you have to implement two methods of the interface.

First, there is a method called `canHandle`. If this method returns `true`, the handler tells the system that it is ready to handle the exception. For example, the following method returns `true` if `java.lang.NumberFormatException` occurs.

```
public boolean canHandle(FlowExecutionException exception) {
   return (exception.getCause() instanceof NumberFormatException);
}
```

The second method that needs to be implemented is the `handle` method with a parameter of the occurred exception (from the `FlowExecutionException` type) and an instance of `RequestControlContext`. The interface is shown in the diagram below:



The instance of `RequestControlContext` is the central interface to manipulate an ongoing flow execution. It is used from various artifacts inside the context of a running flow. One important point to mention if we talk about handling exceptions is how to continue the flow with a specific transition. The method we need for this is the `execute(Transition transition)` method in `RequestControlContext`. As a parameter, we need an instance of the `Transition` class in the `org.springframework.webflow.engine` package. To create this class, we have to provide an instance of `TargetStateResolver` in the same package. There is a default implementation which should fit in most cases. The name of that implementation class is `DefaultTargetStateResolver` in the `org.springframework.webflow.engine.support` package. It has a constructor to provide only the `id` of the state to transfer. The following line shows an example of the usage:

```
context.execute(new Transition(new DefaultTargetStateResolver("target
StateId")));
```

A `TargetStateResolver` has only the `resolveTargetState` method with a state as the return value. It is interesting to know that the return value `null` indicates that there is no state transfer. The task of that class is to resolve the target state of the transition from the source state (it comes as a parameter, but it could be `null`) in the current request context.

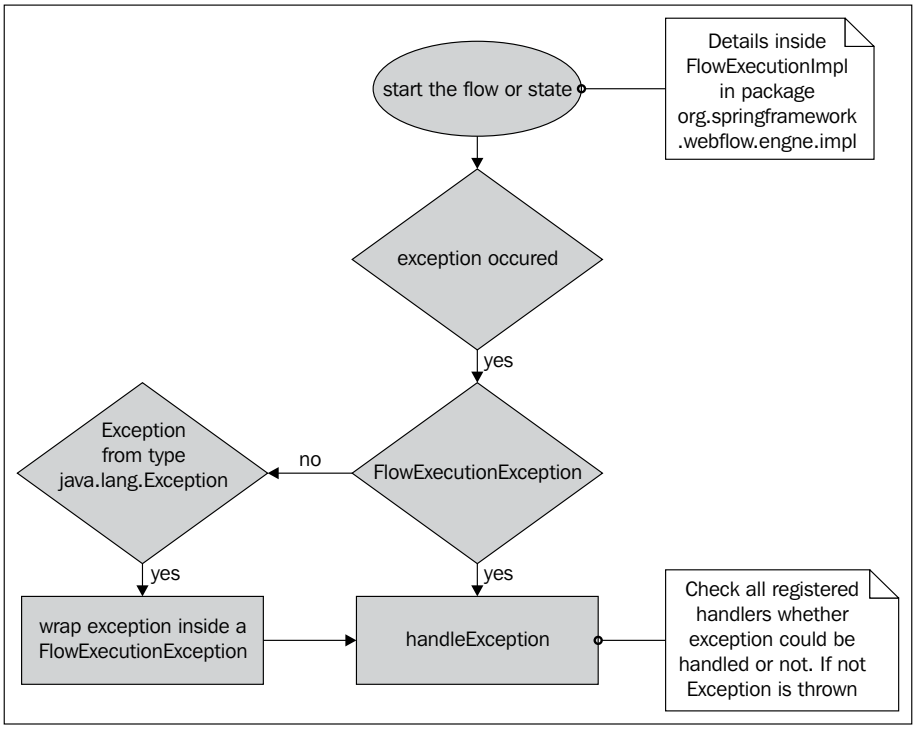
To use an own exception handler, you have to register (inside the flow definition file) the handler with the `exception-handler` tag. An example for a `view-state` is shown in the lines below:

```
<view-state id="add" model="issue" >
   <exception-handler bean="myExceptionHandler" />
</view-state>
```

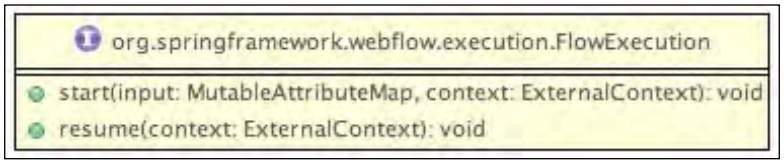
The `exception-handler` tag has only the `bean` attribute, which is a required attribute. The value of the attribute is the name of a bean which has to be registered inside an application context that is reachable from the flow. The bean has to implement the mentioned handler interface (`FlowExecutionExceptionHandler`). The declaration of that bean inside an application context file is not different from the other definition. For completeness, a sample definition is shown below:

```
<bean id="myExceptionHandler" class="flowtrac.service.
MyExceptionHandler" />
```

The algorithm for handling exceptions is implemented inside the `FlowExecutionImpl` internal class (package `org.springframework.webflow.engine.impl`). The following diagram visualizes how exceptions are handled if they occur while the flow executes:



For starting and resuming a flow, there is a central interface called `FlowExecution` (inside the `org.springframework.webflow.execution` package). The methods of this interface are shown in the diagram below:



Additionally, there is one more place where you can deal with exceptions. It is on a `transition`. Here, you have an additional attribute called `on-exception`. The value of this attribute is the fully qualified exception (e.g. `java.lang.NumberFormatException`) which you want to handle. If the mentioned exception occurs, the transition is triggered and the flow is transferred to the specified state in the `to` attribute. An example for that is shown below:

```
<transition on-exception="java.lang.NumberFormatException" to="add" />
```

Take care that you do not use the `on` and `on-exception` attributes together. If you mix these attributes in one transition, an exception of the `NoMatchingTransitionException` type (package `org.springframework.webflow.engine`) is thrown, because the transition cannot be found. The `on-exception` attribute cannot be used in conjunction with a secured element (see more about security in Chapter 7).

The internal implementation of the `on-exception` attribute is done with an exception handler from type of the already mentioned interface `FlowExecutionExceptionHandler`. The name of the class is `TransitionExecutingFlowExecutionExceptionHandler`. This handler is automatically registered from `FlowModelFlowBuilder` if an `on-exception` attribute occurred. If the handler handles an exception, the exception and the root cause are stored inside the `flash` scope (remember: the `el` variable name is `flashScope`). See the following table for the names:

Attribute	Description
`flowExecutionException`	The attribute name where the handled exception is stored inside the `flash` scope.
`rootCauseException`	The attribute name where the root cause of the handled exception is stored inside the `flash` scope.

With the combination of the `on-exception` and the `global-transitions` attributes, you have the opportunity to define global exception handlers. An idea is to create an abstract flow with just such global exception handlers and define the flow as abstract. With the possiblity of the inheritance of the flows, you can isolate the exception handling from your flows (seperation of concerns).

bean-import: declaring beans for a flow

With the `bean-import` element, it is possible to import beans which become part of the bean factory for the flow that imports the beans. The beans must be defined inside a Spring configuration file. To define that file, there is a mandatory attribute called `resource`. Here, you specify the file relative to the location of the flow definition file. In the example below, the `addIssue-beans.xml` resource file is imported and the beans inside are added to the bean factory of the flow. The mentioned resource file in our example must be in the same directory as the flow definition file:

```
<bean-import resource="addIssue-beans.xml" />
```

If you want to import more than one file, you can add just another `bean-import` tag to the flow definition file.

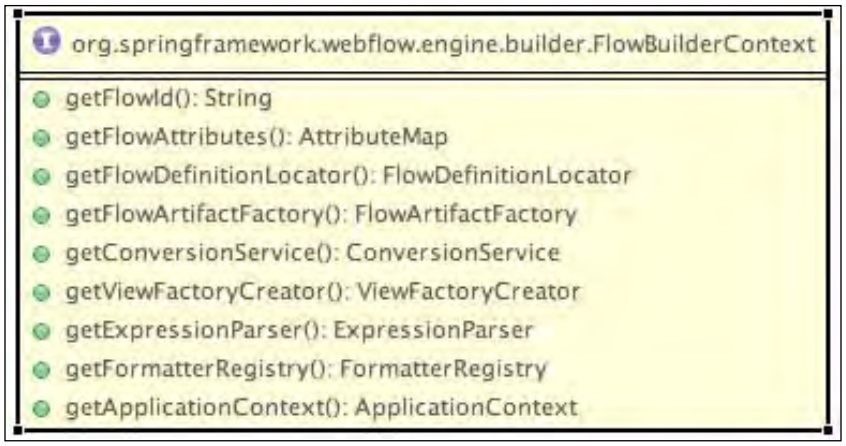
Internals of building a flow

To have a complete overview of building a flow, we describe the building also from the Java view.

Building a flow is a synonym for the transformation of the DSL (Domain Specific Language) which is used to describe the flow in the flow definition file and is needed for the internal representation with instances of Java classes.

Step 1: The initialization

The first step is the initialization of `FlowBuilder` (implementation of `org.springframework.webflow.engine.builder.FlowBuilder`). Create the initial flow definition by calling the `init` method with an instance of `org.springframework.webflow.engine.builder.FlowBuilderContext`. This class is an internal helper to access the elements of a flow definition. For a complete overview, see the class diagram below.



The flow itself is created by an internal call of the `create` method on the `org.springframework.webflow.engine.Flow` class.

Step 2: The variables

Now the `buildVariables` method of `FlowBuilder` is called in order to build any variables initialized by the flow when it starts.

Step 3: The inputs

The input mapper (from type `org.springframework.binding.mapping.Mapper`) is created and set for the flow. The method for that is `buildInputMapper`.

Step 4: The start actions

The `buildStartAction` method creates and adds each of the start actions to the flow.

Step 5: The states

Now the states of the flow in development are created and added to the internal representation of the flow. The method is called `buildStates`.

Step 6: The global transitions

The `buildGlobalTransitions` method creates the transitions which are shared between the states.

Step 7: The end actions

The end actions are created and added through the `buildEndActions` method.

Step 8: The outputs

Now the output mapper is created and set inside the called `buildOutputManager` method. The output mapper is from the `org.springframework.binding.mapping.Mapper` type. It is from the same type as the input mapper.

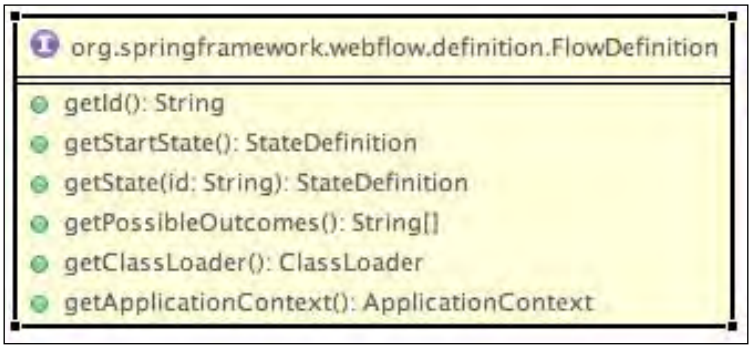
Step 9: The exceptions

Inside the `buildExceptionHandlers` method, the handlers of the flow are created and added to the internal flow representation.

Step 10: Get the full flow definition

From `org.springframework.webflow.engine.builder.FlowAssembler`, the `getFlow` method is called (inside the `assembleFlow` method). It is done to get a reference to the fully built flow definition (directly called from `org.springframework.webflow.config.FlowRegistryFactoryBean`).

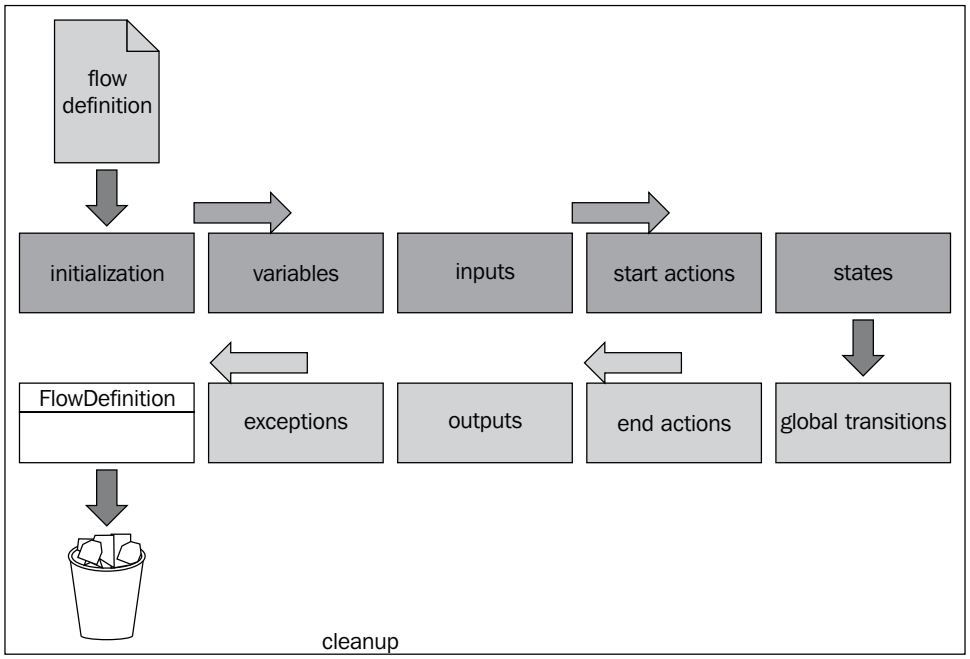
The internal representation of the flow is from the `FlowDefinition` type in the `org.springframework.webflow.definition` package. See the class diagram below:



Step 11: Cleanup

The last step is a cleanup of resources which are allocated during the creation process. This is done with the help of the `dispose` method.

For completion, the eleven steps for creating an instance of the `FlowDefiniton` class (package `org.springframework.webflow.definition`) are shown below.



Configuration

Besides the flow definition file, there is a main configuration file for configuring the flow engine. That configuration file is an XML file, too. The grammar for the file is the XSD `http://www.springframework.org/schema/webflow-config/spring-webflow-config-2.0.xsd`. The XSD file is located inside the `org.springframework.webflow-2.X.X.RELEASE.jar` library in the `org/springframework/webflow/config` `folder`. The configuration is not only a Spring Web Flow configuration but a Spring configuration too. Therefore, you have to add an XSD for that, too. An example header for the configuration file is shown below.

This configuration file has to be loaded in order to bootstrap the Spring Web Flow subsystem. The process of loading this file is often done in the deployment descriptor file of a web application, which is named `web.xml`. For an example, see the quickstart example in Chapter 2 of this book.

```
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:webflow="http://www.springframework.org/schema/webflow-
config"
```

```
   xsi:schemaLocation="
            http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans
/spring-beans-2.5.xsd
            http://www.springframework.org/schema/webflow-config
            http://www.springframework.org/schema/webflow-config/
spring-webflow-config-2.0.xsd">
```

There are two important elements in the configuration which you have to configure. These two elements are:

- FlowRegistry
- FlowExecutor

To use these elements in the XML configuration, you have to use the `webflow` namespace (e.g. `webflow:flow-registry`).

FlowRegistry

You have to provide the webflow system with the location of your flow definition files. These files are registered in the FlowRegistry. An example from the bug tracker application is shown below.

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

As mentioned before, you have to use elements from the defined namespace for the webflow configuration. The name of the element is `flow-registry`. Additionally, it is essential to provide an `id` for `flow-registry`, which is done with the help of the `id` attribute. The line below shows an empty flow registry:

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

The flow registry can be filled with the flow definition file. For that, there is the `flow-location` element. You can define more than one `flow-location` element inside the flow registry. With the `path` attribute, you can specify one single flow definition file (see the line below).

```
<webflow:flow-location path="/WEB-INF/issue/add/add.xml"/>
```

Additionally, it is possible to provide an `id` for the flow location.

```
<webflow:flow-location path="/WEB-INF/issue/add/add.xml" id="addIssue"
/>
```

The mentioned `id` is the unique identifier for the flow. With the `id`, you can call the flow. In the case of our example, we don't define an `id`. Therefore, the flow has the `id` added (see *Internals of the WebFlow Configuration* for more information about this). We call the flow with the `/flowtrac-web/flowtrac/issue/add` URL. If we set the `id` to `addIssue` (the last example), the URL changes to `/flowtrac-web/flowtrac/issue/addIssue`.

The shown possibilities are useful if you have just a few definition files. If you have more configuration files, it could be painful to insert each single flow definition file. For that case, it is possible to provide a directory (with subdirectory) as the location for the configuration files.

```
<webflow:flow-location-pattern value="/WEB-INF/issue/**/*-flow.xml" />
```

In the sample above, all files inside `issue` and below are searched for files which end with `flow.xml`. The notation is similar to the patterns which are known from the build system Apache ANT (see `http://ant.apache.org`).

The flow location can be enriched with meta attributes. For that case, you can specify these attributes inside the `flow-definition-attributes` element.

```
<webflow:flow-location path="/WEB-INF/flows/booking/booking.xml">
    <flow-definition-attributes>
        <attribute name="caption" value="Books a hotel" />
        <attribute name="persistence-context" value="true"
type="boolean" />
    </flow-definition-attributes>
</webflow:flow-location>
```

You can use the attributes in the flow definition file inside the context of the request.

FlowExecutor

A **FlowExecutor** acts as the central point of entrance into the Spring Web Flow system. The FlowExecutor is responsible for executing the flow definition. Internally, it is represented as an interface from the `FlowExecutor` type (package: `org.springframework.webflow.executor`). The implementation of this interface is done in the `FlowExecutorImpl` internal class (package: `org.springframework.webflow.executor`). The FlowExecutor's task is to launch new flow executions and resume flow executions which are paused. The paused flow executions are stored in a `FlowExecutionRepostory` between requests. It is worth mentioning that there is no coupling with the HTTP environment. This means that the implementation is just an execution engine for the flow definition file. The coupling to the HTTP protocol is done in the used controllers.

FlowExecutor Listeners

With the `flow-execution-listeners` element, it is possible to define one or more listeners which observe the lifecycle of flow executions.

```
<flow-execution-listeners>
    <listener ref="listener1"/>
    <listener ref="listener2"/>
</flow-execution-listeners>
```

Two use cases where listeners can be useful are:

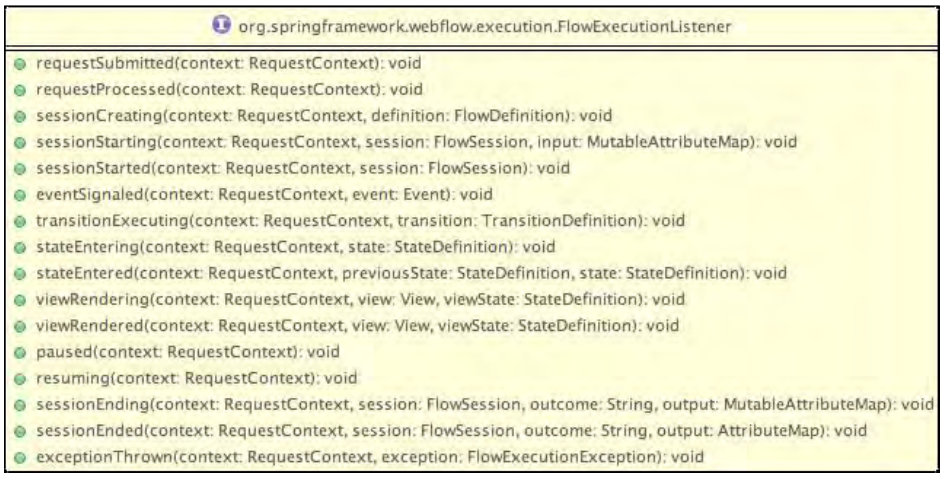
- Security
- Persistence

In our example application, we use such a listener for the persistence.

```
   <webflow:flow-executor id="flowExecutor">
      <webflow:flow-execution-listeners>
         <webflow:listener ref="jpaFlowExecutionListener" />
      </webflow:flow-execution-listeners>
   </webflow:flow-executor>

   <bean id="jpaFlowExecutionListener" class="org.springframework.
webflow.persistence.JpaFlowExecutionListener">
      <constructor-arg ref="entityManagerFactory" />
      <constructor-arg ref="transactionManager" />
   </bean>
```

The listener class has to be from the `org.springframework.webflow.execution.FlowExecutionListener` type. Just to see what you have to implement, we show the class diagram for the listener:



Implement your own FlowExecutionListener

The interface for `FlowExecutionListener` has many methods, but often you only need just a few of them. For this scenario, there is an abstract class which implements the methods with an empty body (the template method pattern which is known from GoF). The mentioned abstract class with the name `FlowExecutionListenerAdapter` is located in the same package as the `FlowExecutionListener` interface. The Spring Web Flow framework comes with many classes which work as listeners for a flow. But only the mentioned adapter class implements the `FlowExecutionListener` interface. All other classes are just subclasses from the adapter (for example, the `org.springframework.webflow.persistence.JpaFlowExecutionListener` class).

Internals of the Webflow Configuration

For the creation of the internal configuration of a class, there is a definition inside the `spring.handlers` properties file (inside the `META-INF` directory of the webflow JAR). Here is the class: `org.springframework.webflow.config.WebFlowConfigNamespaceHandler` (a mapping from XSD to handler class). It is worth mentioning that there is an additional `spring.schemas` file in the `META-INF` directory which sets the location of the XSD files. The namespace handler is defined for four elements (`flow-executor`, `flow-execution-listeners`, `flow-registry`, and `flow-builder-services`), which are the parsers for internal representation of the configuration. All parsers are located in the `org.springframework.webflow.config` package.

Name	Class
`flow-executor`	`FlowExecutionBeanDefintionParser`
`flow-execution-listeners`	`FlowExecutionListenerLoaderBeanDefintionParser`
`flow-registry`	`FlowRegistryBeanDefinitionParser`
`flow-builder-services`	`FlowBuilderServicesBeanDefintionParser`

The flow registry is an internal implementation of `org.springframework.webflow.engine.model.registry.FlowModelRegistry`. The name of the implementation class is `FlowModelRegistryImpl`. One interesting detail is the `getFlowId` method of the `FlowDefintionResourceFactory` class. This method creates the `id` of the flow if not specified.

```
protected String getFlowId(Resource flowResource) {
String fileName = flowResource.getFi lename();
   int extensionIndex = fileName.lastIndexOf('.');
```

```
        if (extensionIndex != -1) {
           return fileName.substring(0, extensionIndex);
        } else {
           return fileName;
        }
     }
```

As you can see above, the method just uses the filename of the flow definition as the `id` (without extension). If you want to change the algorithm, you can do so with the help of the `id` attribute. The method is only called if the `id` is specified. See the following extract from `createResource` of the `FlowDefinitionResourceFactory` class.

```
if (flowId == null || flowId.length() == 0) {
         flowId = getFlowId(resource);
}
```

By the way, the mentioned factory is responsible for the creation of an instance of `FlowDefintionResource`.

A helpful configuration is the parameter development of the mentioned element— `flowBuilderServices`. With this parameter, the system goes into development mode and each change results in a reload. See the following line for an example:

```
<webflow:flow-builder-services id="flowBuilderServices"
development="true" />
```

Inheritance inside a flow definition

With Spring Web Flow 2, the principle of inheritance for a flow definition is introduced. That means one flow can inherit the configuration of another flow. It is possible to use inheritance for states and flows. If you want to use inheritance, it is important that the parent flow is an element of the `flow-registry`.

If we compare the Java inheritance model and the flow model, there are some important differences. The key elements of the flow (state) inheritance are:

- it is not possible for a child flow to override an element from the parent flow
- elements which are similar between the child and parent are merged
- elements from the parent flow which are unique will be added to the child flow

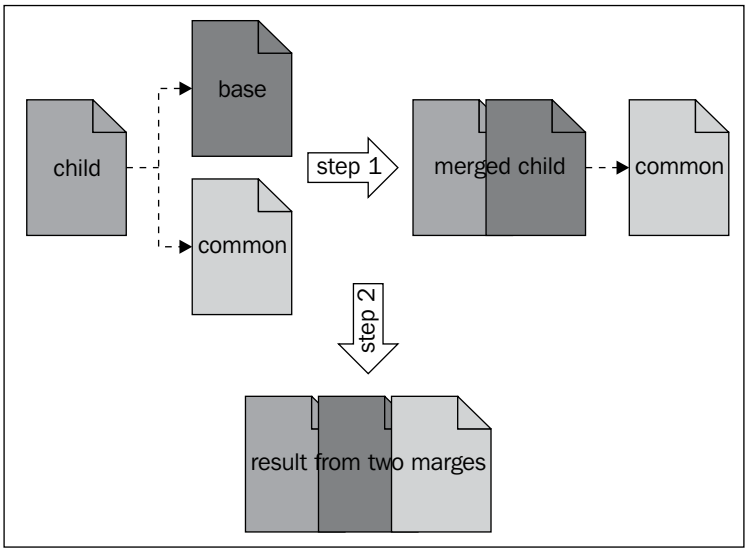
From Java, we know that we can only extend from one class. In terms of a flow, it is possible to extend from **more** than one flow. The attribute on the flow (and the state) for declaring inheritance is `parent`. It is important to say that a state can only extend from **one** other state and not more. Now, we just show two examples for a better understanding.

Inheritance for flows

The attribute for the inheritance for flows is `parent`. For a flow, it is possible to use more than one definition for `parent`. For that case, you have to use a comma as a delimiter. The merge process starts from the left to the right. See the line below for a simple definition of a flow inheritance with two parents:

```
<flow parent="base, common">
```

The diagram below helps to understand the direction of the merge process:



There is one additional feature in the case of inheritance. There is a possibility to declare a flow as abstract. This means that the flow cannot be instantiated. You can do so with the help of the `abstract` attribute for the `flow` element. An example is shown below:

```
<flow abstract="true" />
```

If you try to instantiate an abstract flow, a run-time exception from the `FlowBuilderException` type (package `org.springframework.webflow.engine.builder`) is thrown.

Inheritance for states

A state can only extend one state. For that, there is the `parent` attribute. Besides the name of the state (remember: you can only inherit from **one** state), you have to mention the flow. The name of the flow and the state are separated with the help of the # character.

Example:

```
<view-state id="child-state" parent="parent-flow#parent-view-state">
```

Merge or no merge

The inheritance of a flow or state is done by merging the elements. It depends on the element or attribute as to whether the merge process is started or not.

Name of element	Name of the attribute	M	NM
action-state	Id	x	
Attribute	Name	x	
bean-import	-		x
decision-state	Id	x	
Evaluate	-		x
end-state	Id	x	
exception-handler			x
end-state	Id	x	
Flow	-	x	
If	Test	x	
on-end	-	x	
on-entry	-	x	
on-exit	-	x	
on-render	-	x	
on-start	-	x	
Input	Name	x	
Output	Name	x	
persistence-context	-		x

Name of element	Name of the attribute	M	NM
Render	-		x
Secured	Attributes	x	
Set	-		x
subflow-state	-	x	
transition	On	x	
Var	-		x
view-state	Id	x	

The complete flow for the example

For your overview, we want to show you a complete flow file (`add.xml` from the `webapp/WEB-INF/issue/add` directory) from our example application:

```
<?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" >

    <persistence-context/>

    <!-- The id of the issue which should be edited -->
    <input name="id" />

    <on-start>
      <evaluate expression="issueService.findById(id)"
result="flowScope.issue">
         <attribute name="create" value="true" type="boolean"></
attribute>
      </evaluate>
   </on-start>

       <!--
       If no view is specified a page with the same name
       as the id is searched in the directory of the flow definition.
       Additionally its possible to define a view with the attribute
view.
     -->
```

```
      <view-state id="add" model="issue">
         <transition on="store" to="issueStore" >
            <evaluate expression="persistenceContext.persist(issue)"/>
         </transition>
      </view-state>
      <end-state id="issueStore" commit="true" view="externalRedirect:
contextRelative:/flowtrac/issue/all" />
      <on-end>
         <evaluate expression="itemInformationService.enrich(issue)" />
      </on-end>
   </flow>
```

Summary

In this chapter, we have shown you all the basics around the creation of a flow with the Spring Web Flow framework in version 2. With that knowledge, you should be ready to implement your own flows.

In the following chapters we are going to show you more concepts of Spring Web Flow 2. Especially, the integration with other frameworks will be a topic in the upcoming chapters.