



Reference Documentation

Version 1.0.1

January 2007

Copyright © 2004-2007 Keith Donald, Erwin Vervaet, Ross Stoyanchev

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

Sponsors

Spring Web Flow would not be possible without the investment of its sponsors: [Interface21](#) and [Eryacon](#).



Table of Contents

Preface	
1. Introduction	
1.1. Overview	6
1.2. Architecture overview	6
1.3. Architectural layers	6
1.4. Layer descriptions	7
1.4.1. The Execution Core Layer (Bottom Layer)	7
1.4.2. The Execution Engine Layer	9
1.4.3. The Test Layer	9
1.4.4. The Executor Layer	10
1.4.5. The System Configuration Layer (Top Layer)	10
1.5. Support	11
2. Flow definition	
2.1. Introduction	12
2.2. FlowDefinition	12
2.2.1. XML-based Flow template	13
2.2.2. Java Flow API example	14
2.3. StateDefinition	14
2.4. Transitionable State	15
2.4.1. XML-based state template	15
2.4.2. Java state API example	16
2.5. TransitionDefinition	16
2.5.1. Transition XML template	17
2.5.2. Transition Java API example	17
2.5.3. Action transition execution criteria	17
2.5.4. Dynamic transitions	17
2.5.5. Global transitions	18
2.5.6. Transition executing state exception handlers	19
2.6. Concrete state types	19
2.6.1. ViewState	20
2.6.2. ActionState	26
2.6.3. DecisionState	35
2.6.4. SubflowState	36
2.6.5. EndState	39
3. Flow execution	
3.1. Introduction	43
3.2. FlowExecution	43
3.2.1. Flow execution creation	43
3.2.2. Flow execution startup	43
3.2.3. Flow execution resume	44
3.2.4. Flow execution lifecycle	44
3.2.5. Flow execution properties	45
3.2.6. Flow execution impl creation	45
3.3. Flow execution context	46
3.4. Flow execution scopes	48
3.5. Flow execution testing	49
3.5.1. Flow execution test example	49
3.5.2. Execution unit testing vs. full-blown system testing	52

4. Flow execution repositories	
4.1. Introduction	54
4.2. Repository architecture overview	54
4.3. Flow execution identity	55
4.3.1. Conversation identifier	55
4.3.2. Continuation identifier	56
4.3.3. Flow execution key	56
4.4. Conversation ending	56
4.5. Flow execution repository implementations	56
4.5.1. Simple flow execution repository	56
4.5.2. Continuation flow execution repository	57
4.5.3. Client continuation flow execution repository	57
5. Flow executors	
5.1. Introduction	59
5.2. FlowExecutor	59
5.2.1. FlowExecutorImpl	60
5.2.2. A typical flow executor configuration with Spring 2.0	60
5.2.3. A flow executor using a simple execution repository	61
5.2.4. A flow executor using a client-side continuation-based execution repository	61
5.2.5. A flow executor using a single key execution repository	61
5.2.6. A flow executor setting system execution attributes	61
5.2.7. A flow executor setting custom execution listeners	62
5.2.8. A Spring 1.2 compatible flow executor configuration	62
5.3. Spring MVC integration	63
5.3.1. A single flow controller executing all flows in a Servlet MVC environment	63
5.3.2. A single portlet flow controller executing a flow within a Portlet	63
5.4. Flow executor parameterization	63
5.4.1. Request parameter-based flow executor argument extraction	64
5.4.2. Request path based flow executor argument extraction	65
5.5. Struts integration	66
5.5.1. A single flow action executing all flows	66
5.6. Java Server Faces (JSF) integration	66
5.6.1. A typical faces-config.xml file	66
5.6.2. Launching a flow execution - command link	67
5.6.3. Resuming a flow execution - form	67
6. Practical Use of Spring Web Flow	
6.1. Sample applications	68
6.2. Running the Web Flow sample applications	68
6.2.1. Building from the Command Line	68
6.2.2. Importing Projects into Eclipse	69
6.2.3. Deploying projects inside Eclipse using Eclipse Web Tools (WTP)	69
6.2.4. Other IDE's	69
6.3. Fileupload Example	69
6.3.1. Overview	69
6.3.2. Web.xml	70
6.3.3. Spring MVC Context	70
6.3.4. Fileupload Web Flow	71
6.4. Birthdate Example	71
6.4.1. Overview	71
6.4.2. Web.xml	72
6.4.3. Struts Configuration	72
6.4.4. Birthdate Web Flow	73

6.4.5. Birthdate-alternate Web Flow 75

Preface

Many web applications consist of a mix of free browsing, where the user is allowed to navigate a web site as they please, and controlled navigations where the user is guided through a series of steps towards completion of a business goal.

Consider the typical shopping cart application. While a user is shopping, she is freely browsing available products, adding her favorites to her cart while skipping over others. This is a good "free browsing" use case. However, when the user decides to checkout, a controlled workflow begins--the checkout process. Such a process represents a single user conversation that takes place over a series of steps, and navigation from step-to-step is controlled. The entire process represents an discrete application transaction that must complete exactly once or not at all.

Consider some other good examples of "controlled navigations": applying for a loan, paying your taxes on-line, booking a trip reservation, registering an account, or updating a warehouse inventory.

Traditional approaches to modeling and enforcing such controlled navigations or "flows" fall flat, and fail to express the Flow as a first class concept. Spring Web Flow (SWF) is a component of the Spring Framework's web stack focused on solving this problem in a productive and powerful manner.

Chapter 1. Introduction

1.1. Overview

Spring Web Flow (SWF) is a component of the Spring Framework's web stack focused on the definition and execution of UI flow within a web application.

The system allows you to capture a logical flow of your web application as a self-contained module that can be reused in different situations. Such a flow guides a single user through the implementation of a business task, and represents a single user *conversation*. Flows often execute across HTTP requests, have state, exhibit transactional characteristics, and may be dynamic and/or long-running in nature.

Spring Web Flow exists at a higher level of abstraction, integrating as a self-contained *flow engine* within base frameworks such as Struts, Spring MVC, Portlet MVC, and JSF. SWF provides you the capability to capture your application's UI flow explicitly in a declarative, portable, and manageable fashion. SWF is a powerful controller framework based on a finite-state machine, fully addressing the "C" in MVC.

1.2. Architecture overview

Spring Web Flow has been architected as a self-contained *flow engine* with few required dependencies on third-party APIs. All dependencies are carefully managed.

At a minimum, to use Spring Web Flow you need:

- spring-webflow (the framework)
- spring-core (miscellaneous utility classes used internally by the framework)
- spring-binding (the Spring data binding framework, used internally)
- commons-logging (a simple logging facade, used internally)
- OGNL (the default expression language)

Most users will embed SWF as a component within a larger web application development framework, as SWF is a focused *controller technology* that expects a calling system to care for request mapping and response rendering. In this case, those users will depend on a thin integration piece for their environment. For example, those executing flows within a Servlet environment might use the Spring MVC integration to care for dispatching requests to SWF and rendering responses for SWF view selections. Spring Web Flow ships convenient Spring MVC, Struts Classic, and JSF integration out of the box.



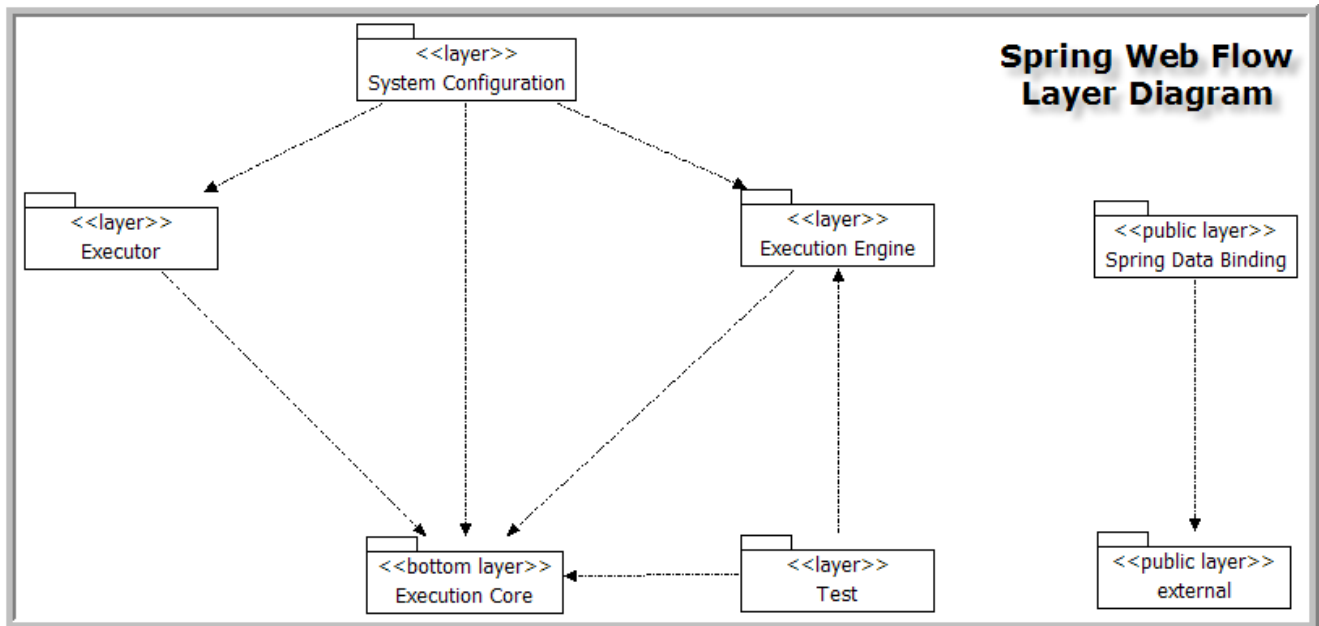
Note

Spring Web Flow, like Spring, is a *layered* framework, packaged in a manner that allows teams to use the parts they need and nothing else. For example, one team might use Spring Web Flow in a Servlet environment with Spring MVC and thus require the Spring MVC integration. Another team might use SWF in a Portlet environment, and thus require the Portlet MVC integration. Another team might mix and match. A major benefit of SWF is that it allows you to define reusable, self-contained controller modules that can execute in *any* environment.

1.3. Architectural layers

Spring Web Flow is a layered framework. A diagram of Spring Web Flow's layered architecture is shown

below:



Spring Web Flow layer diagram

1.4. Layer descriptions

Each layer is partitioned into one or more subsystems that together carry out the layer's role within the overall system. This section notes the purpose of each layer and describes each subsystem in the following format:

- *Subsystem name* - The name of a layer subsystem.
- *Description* - The purpose of the subsystem.
- *Packages* - The Java packages that contain the source code for the subsystem. The packages are rooted at the `org.springframework.webflow` root package in the package hierarchy.
- *Subsystem interfaces* - Central API elements exposed by the subsystem, typically through Java interfaces.
- *Internal dependencies* - Dependencies of the subsystem. These could be other subsystems of the layer or external libraries.

1.4.1. The Execution Core Layer (Bottom Layer)

Defines core flow definition and execution public APIs. As the "bottom layer", this layer is highly stable with no dependencies on any other layer.

Table 1.1. Execution Core Subsystems

Subsystem name	Description	Packages	Subsystem interfaces	Internal dependencies
Core	Foundational, generic types usable by all other subsystems. Contains the default expression parser (OGNL-based) and core collection types (AttributeMap and company).	core, core.collection	None	None
Util	Low level utilities used by all other	util	None	None

Subsystem name	Description	Packages	Subsystem interfaces	Internal dependencies
	parts of the system.			
Flow Definition	Central abstractions for modeling flow definitions. These abstractions include <code>FlowDefinition</code> , <code>StateDefinition</code> , and <code>TransitionDefinition</code> that form the domain language for describing flows.	<code>definition</code>	<code>FlowDefinition</code>	Core
Flow Definition Registry	Support for working with registries of flow definitions. Flow definitions eligible for execution are typically stored in a registry providing lookup services.	<code>definition.registry</code>	<code>FlowDefinitionRegistry</code> , <code>FlowDefinitionLookup</code>	Core, Flow Definition
External Context	Provides normalized access to a client environment that has called into Spring Web Flow.	<code>context</code> , <code>context.servlet</code> , <code>context.portlet</code>	<code>ExternalContext</code>	Core, <code>context.servlet</code> requires Servlet API 2.3, <code>context.portlet</code> requires Portlet API 1.0 in addition to Servlet API 2.3
Conversation	Manages the creation and cleanup of conversational state. Used by the execution repository system to begin new user conversations and track execution state.	<code>conversation</code> , <code>conversation.impl</code>	<code>ConversationManager</code>	Core, Util, External Context
Flow Execution	Stable runtime abstractions that define the flow definition execution model. For executing flow definitions and representing execution state.	<code>execution</code> , <code>execution.support</code> , <code>execution.factory</code>	<code>FlowExecution</code>	Core, External Context, Flow Definition
Flow Execution Repository	For persisting paused flow executions beyond a single request into the server.	<code>execution.repository</code> , <code>execution.repository.support</code> , <code>execution.repository.continuation</code>	<code>FlowExecutionRepository</code>	Core, Util, Flow Definition, Conversation, Flow Execution, <code>repository.continuation</code> requires commons-codec 1.0 if using client continuations
Action	Reusable action implementations.	<code>action</code> , <code>action.portlet</code>	None	Core, Util, Flow Definition, External

Subsystem name	Description	Packages	Subsystem interfaces	Internal dependencies
				Context, Flow Execution

1.4.2. The Execution Engine Layer

Defines an implementation of the flow execution core API, forming the basis of the state machine or "engine" implementation. More volatile, as it contains specific implementations of stable execution abstractions.

Depends On: Execution Core

Table 1.2. Execution Engine Subsystems

Subsystem name	Description	Packages	Subsystem interfaces	Internal dependencies
Engine Implementation	The implementation of the flow execution engine based on a finite state machine.	engine, engine.support, engine.impl	None	None
Flow Definition Builder	Abstractions used at configuration-time for building and assembling Flow definitions executable by this engine implementation. Flows are typically defined in externalized resources such as XML files.	engine.builder, engine.builder.xml	FlowBuilder	Engine Implementation, Spring Beans 1.2.7, Spring Context 1.2.7, builder.xml requires JDK 1.5 or Xerces for XSD support

1.4.3. The Test Layer

Support for unit testing flow artifacts and system testing flow executions.

Depends On: Execution Engine, Execution Core

Table 1.3. Test Subsystems

Subsystem name	Description	Packages	Subsystem interfaces	Internal dependencies
Engine Artifact Unit Test Support	Support for unit testing implementations such as Actions in isolation.	test	None	JUnit 3.8.1
Flow Execution Test Support	Support for testing Flow Executions out-of-container.	test.execution	None	Spring Beans 1.2.7, JUnit 3.8.1

1.4.4. The Executor Layer

Stable higher-layer for driving and coordinating the execution of flow definitions. This layer is decoupled from the more-volatile engine implementation.

Depends On: Execution Core

Table 1.4. Executor Subsystems

Subsystem name	Description	Packages	Subsystem interfaces	Internal dependencies
Core	Stable, generic flow executor abstractions and support.	executor, executor.support	FlowExecutor	None
Spring MVC	The integration between Spring Web Flow and the Spring MVC framework.	executor.mvc	None	Core, Spring Web MVC 1.2.7, Portlet MVC requires Spring 2.0
Struts	The integration between Spring Web Flow and the Struts Classic framework.	executor.struts	None	Core, Struts 1.1
Java Server Faces (JSF)	The integration between Spring Web Flow and the Java Server Faces framework.	executor.jsf	None	Core, JSF 1.0

1.4.5. The System Configuration Layer (Top Layer)

The top-most layer for configuring the overall Spring Web Flow system for use within an application. As the top layer, this layer depends on the most.

Depends On: Executor, Execution Engine, Execution Core

Table 1.5. System Configuration Subsystems

Subsystem name	Description	Packages	Subsystem interfaces	Internal dependencies
Spring Configuration Support	For configuring Spring Web Flow using Spring 1.x and 2.x.	config	None	Spring Beans 1.2.7, spring-webflow-config-1.0 XSD support requires Spring 2.0



Note

As described above, some subsystem packages are optional depending on your use of the subsystem. For example, use of Spring Web Flow in a Servlet environment entails use of the

`ExternalContext` `context.servlet` package which requires the Servlet API to be in the classpath. In this case the `context.portlet` package is not used and the Portlet API is not required.

For the exact list of dependencies, as well as supported product usage configurations, see the Ivy dependency manager descriptor located within the SWF distribution.

1.5. Support

Spring Web Flow 1.x is supported on Spring Framework 1.2.7 or > for the 1.x series and supported on 2.0 or > for the 2.x series.

XML-based flow building requires Xerces 2 or JDK 5.0 (for XSD support).

The Spring Web Flow Portlet integration requires Spring Portlet MVC 2.0.

Our active community support forum is located at <http://forum.springframework.org>.

Chapter 2. Flow definition

2.1. Introduction

Spring Web Flow allows developers to build reusable, self-contained controller modules called flows. A flow defines a user dialog that responds to user events to drive the execution of application code to complete a business goal.

Flows are defined declaratively using a rich domain-specific language (DSL) tailored to the problem domain of UI flow. Currently, XML and Java-based forms of this language are provided.

This chapter documents Spring Web Flow's core flow definition language. You will learn the core domain constructs of the system and how those constructs are representable in an externalized XML form.

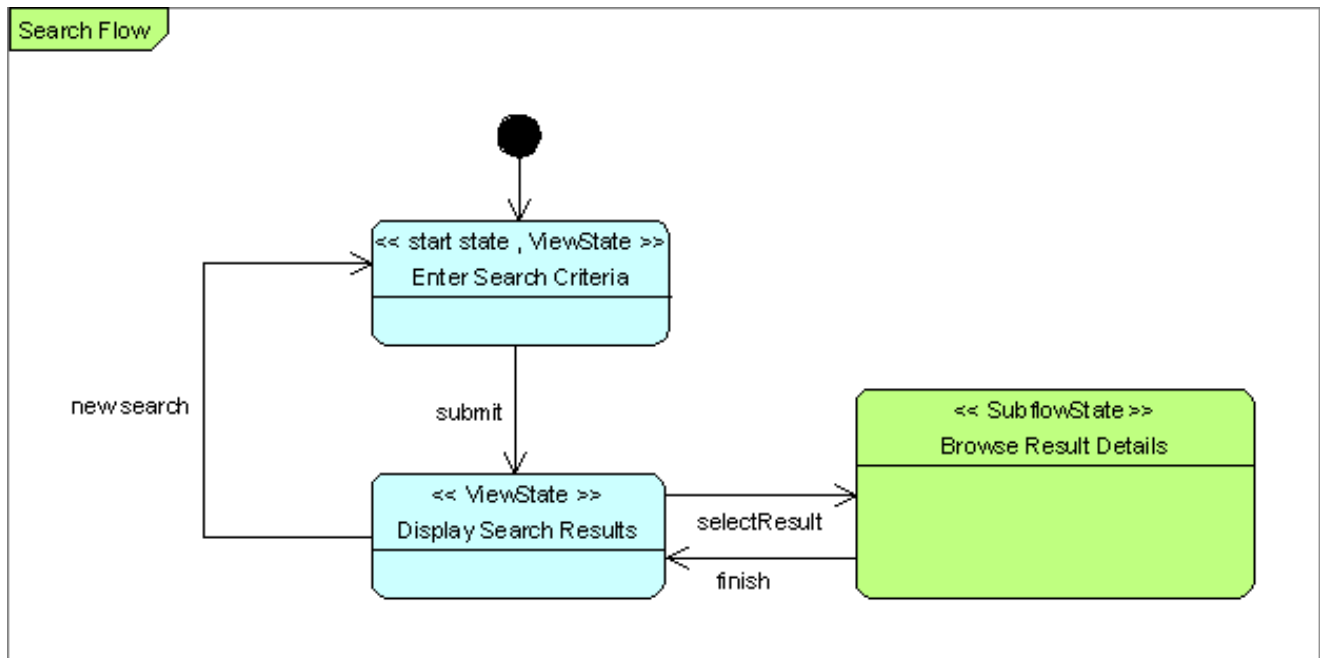
2.2. FlowDefinition

A flow definition is an instance of `org.springframework.webflow.definition.FlowDefinition`. This is the central domain artifact representing the definition of a user dialog or task.

A flow definition consists of a set of one or more states, where each state defines a step in the flow that when entered executes a behavior. What behavior is executed is a function of the state's type and configuration. The outcome of a state's execution, called an event, is used by the flow to drive a state transition.

Exactly one of a flow's states is the `startState` that defines the starting point of the flow. Optionally, a flow can have one or more end states defining the ending points of the flow.

An example definition of a simple flow to carry out a search process is shown graphically below:



Search Flow

The default `FlowDefinition` implementation in Spring Web Flow is `org.springframework.webflow.engine.Flow`. Its configurable properties are summarized below:

Table 2.1. Flow properties

Property name	Description	Cardinality	Default value
id	The identifier of the flow definition, typically unique to all other flows of the application.	1	
attributes	Additional custom attributes about the flow.	0..*	None
states	The steps of the flow.	1..*	
startState	The starting point of the flow.	1	
variables	The set of flow variables to create each time an execution of the flow is started.	0..*	Empty
inputMapper	The service responsible for mapping flow input provided by a caller each time an execution of the flow is started.	0..1	Null
startActions	The list of actions to execute each time an execution of the flow is started.	0..*	Empty
endActions	The list of actions to execute each time an execution of the flow ends.	0..*	Empty
outputMapper	The service responsible for mapping flow output to expose to the caller each time an execution of the flow ends.	0..1	Null
globalTransitions	The set of transitions shared by all states of the flow.	0..*	Empty
exceptionHandlers	An ordered set of handlers to be applied when an exception is thrown within a state of the flow.	0..*	Empty
inlineFlows	A set of inner flows that will be called as subflows; these flows are locally scoped to the outer flow.	0..*	Empty

Below is a high level example of how these properties can be configured in XML form or directly in Java code.

2.2.1. XML-based Flow template

```
<?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-1.0.xsd">

  <attribute .../>

  <var .../>

  <input-mapper .../>
```

```

<start-actions>
  ...
</start-actions>

<start-state idref="yourStartingStateId"/>

<-- your state definitions go here -->

<global-transitions>
  ...
</global-transitions>

<end-actions>
  ...
</end-actions>

<output-mapper .../>

<exception-handler .../>

<inline-flow>
  ...
</inline-flow>

</flow>

```

2.2.2. Java Flow API example

```

Flow flow = new Flow("id");
flow.getAttributeMap().put("name", "value");
flow.addState(...);
flow.setStartState("startingPoint");
flow.addVariable(...);
flow.setInputMapper(...);
flow.getStartActionList().add(...);
flow.getEndActionList().add(...);
flow.setOutputMapper(...);
flow.getGlobalTransitionSet().add(...);
flow.getExceptionHandlerSet().add(...);
flow.addInlineFlow(...);

```

A Flow is typically built by a FlowBuilder rather than assembled by hand. The flow building subsystem is contained within the `org.springframework.webflow.engine.builder` package. The XML Flow Builder and `spring-webflow.xsd` schema are located within the `org.springframework.webflow.engine.builder.xml` package. The XML-based format is the most popular way to define flows.

2.3. StateDefinition

A StateDefinition defines the behavior for a step of a FlowDefinition. The base implementation class for all Flow state types is `org.springframework.webflow.engine.State`. This abstract class defines common properties applicable to all state types, which include:

Table 2.2. State properties

Property name	Description	Cardinality	Default value
id	The id of the state, unique to its containing flow definition.	1	

Property name	Description	Cardinality	Default value
owner	The owning flow definition.	1	
attributes	Additional custom attributes about the state.	0..*	None
entryActions	The list of actions to execute each time the state is entered.	0..*	Empty
exceptionHandlers	An ordered set of handlers to be invoked when an exception is thrown within the state.	0..*	Empty

2.4. Transitionable State

A central subclass of `State` is `org.springframework.webflow.transitionablestate`. This abstract class defines common properties applicable to all state types that execute transitions to other states in response to events. These properties include:

Table 2.3. TransitionableState properties

Property name	Description	Cardinality	Default value
transitions	The eligible paths out of this state.	1..*	
exitActions	The list of actions to execute each time this state is exited.	0..*	Empty

Below is a mock flow definition snippet showing how properties may be configured for a `TransitionableState` in XML and in Java code:

2.4.1. XML-based state template

```
<?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-1.0.xsd">

  <start-state idref="myStateId"/>

  <xxx-state id="myStateId">
    <attribute name="..." value="..."/>

    <entry-actions>
      ...
    </entry-actions>

    <transition on="..." to="..."/>
    <transition on-exception="..." to="..."/>

    <exit-actions>
      ...
    </exit-actions>

    <exception-handler .../>
  </xxx-state>
</flow>
```

```
</flow>
```

2.4.2. Java state API example

```
Flow flow = new Flow("id");
TransitionableState state = new XXXState(flow, "stateId");
state.getAttributeMap().put("name", "value");
state.getEntryActionList().add(...);
state.getTransitionSet().add(...);
state.getExitActionList().add(...);
```

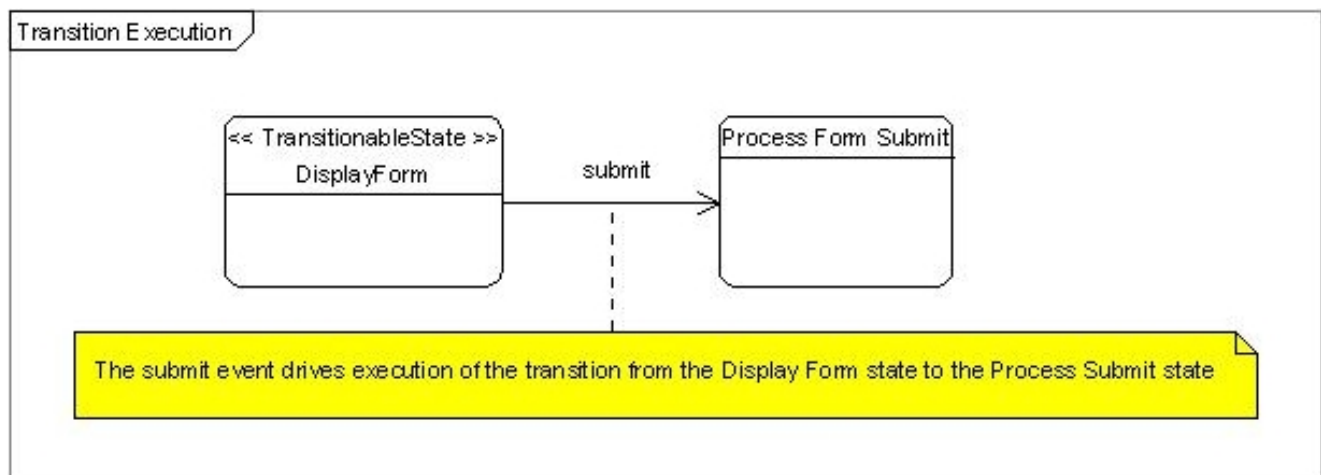
A State is typically constructed by a `FlowArtifactFactory`, used by a `FlowBuilder` during flow assembly. The flow building subsystem is contained within the `org.springframework.webflow.engine.builder` package.

2.5. TransitionDefinition

A transition takes a flow from one state to another, defining a *path* through the flow. This is modeled using a `TransitionDefinition`.

Recall that all `TransitionableStates` have a set of one or more transitions, each defining a path to another state in the flow (or a recursive path back to the same state). When a transitionable state is entered it executes a behavior. For example, a transitionable state called "Display Form" may display a form to the user and wait for user input. The outcome of the state's execution, called an event, is used to drive execution of one of the state's transitions. For example, the user may press the form submit button which signals a *submit* event that matches the transition to the "Process Submit" state.

This event-driven transition execution process is shown graphically below:



Transition execution

The transition definition implementation is defined by an instance of `org.springframework.webflow.engine.Transition`. Its properties are summarized below:

Table 2.4. Transition properties

Property name	Description	Cardinality	Default value
attributes	Additional attributes describing the transition.	0..*	None
matchingCriteria	The strategy that determines if the transition matches on an event occurrence.	1	Always matches
executionCriteria	The strategy that determines if the transition, once matched, is allowed to execute.	1	Always allowed
targetStateResolver	The strategy that resolves the target state of the transition. Most transitions always resolve to the same target state. This strategy allows for dynamic resolution.	1	

Below is a high-level example of how a Transition can be configured in XML form or directly in Java code.

2.5.1. Transition XML template

```
<transition on="event" to="targetState">
  <attribute ... />
  <action ... />
</transition>
```

2.5.2. Transition Java API example

```
Transition transition = new Transition("targetState");
transition.getAttributeMap().put("name", "value");
transition.setMatchingCriteria(new EventIdTransitionCriteria("event"));
transition.setExecutionCriteria(...);
```

2.5.3. Action transition execution criteria

In the XML transition template above note the support for the `action` element within the `transition` element.

A transition may be configured with one or more actions that execute *before* the transition itself executes as `executionCriteria`. If one or more of these actions do not complete successfully the transition will *not* be allowed. This *action transition criteria* makes it possible to execute arbitrary logic after a transition is matched but before it is executed. This is useful when you want to execute event post-processing logic. A good example is executing form data binding and validation behavior after a form submit event.

2.5.4. Dynamic transitions

A transition's target state resolver can be configured to dynamically calculate the target state. For example:

```
<transition on="back" to="{flowScope.lastStateId}" />
```

This will transition the flow to the state resolved by evaluating the `flowScope.lastStateId` expression.

2.5.5. Global transitions

As outlined, one or more transitions are added to all `TransitionableState` types, attached at the state-level. Optionally, transitions may also be added at the *flow-level* where they are shared by all states. These shared transitions are called *global transitions*.

When an event is signaled in a transitionable state the state will first try and match one of its own transitions. If there is no match at the state level the set of global transitions will be tested. If there still is no match a `NoMatchingTransitionException` will be thrown.

Global transitions are useful in situations where many states of the flow share the same transitional criteria. For example, consider a navigation menu that displays alongside each view of a flow. Logic to process navigation menu events is needed by all view states. This is the problem global transitions are designed to solve.

2.5.5.1. Global transitions - XML example

The following example shows transitions defined at the state level, as well as global transitions defined at the flow level.

```
<?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-1.0.xsd">

  <start-state idref="state1"/>

  <xxx-state id="state1">
    <transition on="localEvent1" to="state2"/>
  </xxx-state>

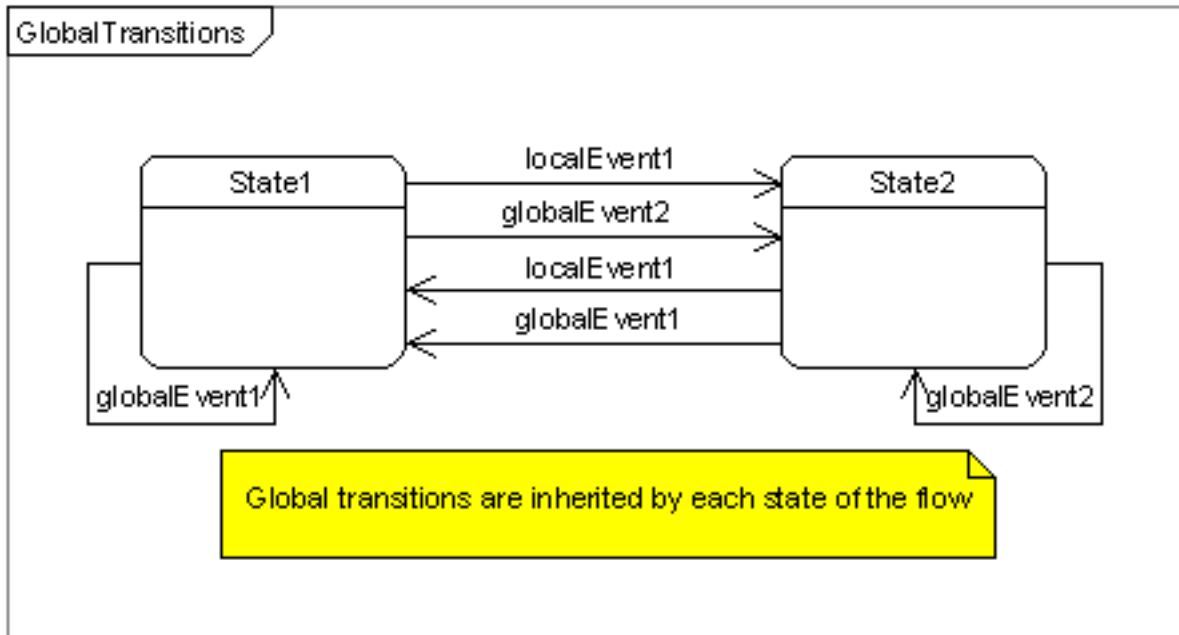
  <xxx-state id="state2">
    <transition on="localEvent1" to="state1"/>
  </xxx-state>

  <global-transitions>
    <transition on="globalEvent1" to="state1"/>
    <transition on="globalEvent2" to="state2"/>
  </global-transitions>

</flow>
```

In this mock example `state1` defines one transition and also inherits the two others defined within the `global-transitions` element. Any other states defined within this flow would also inherit those global transitions.

This example is shown graphically below:



Global transitions

2.5.6. Transition executing state exception handlers

The `<transition/>` element contains an exclusive `on-exception` attribute used to specify an exception-based criteria for transition execution. This allows you to transition the flow to another state on the occurrence of an exception.

2.5.6.1. Exception handling - XML example

The following example shows a transition that is applied as a state exception handler:

```
<?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-1.0.xsd">

  <start-state idref="state1"/>

  <xxx-state id="state1">
    <transition on="event1" to="state2"/>
    <transition on-exception="example.MyBusinessException" to="state3"/>
  </xxx-state>

  ...

</flow>
```

In this example `state1` defines one transition and an exception handler which executes a transition to `state3` if a `MyBusinessException` is thrown within the state.

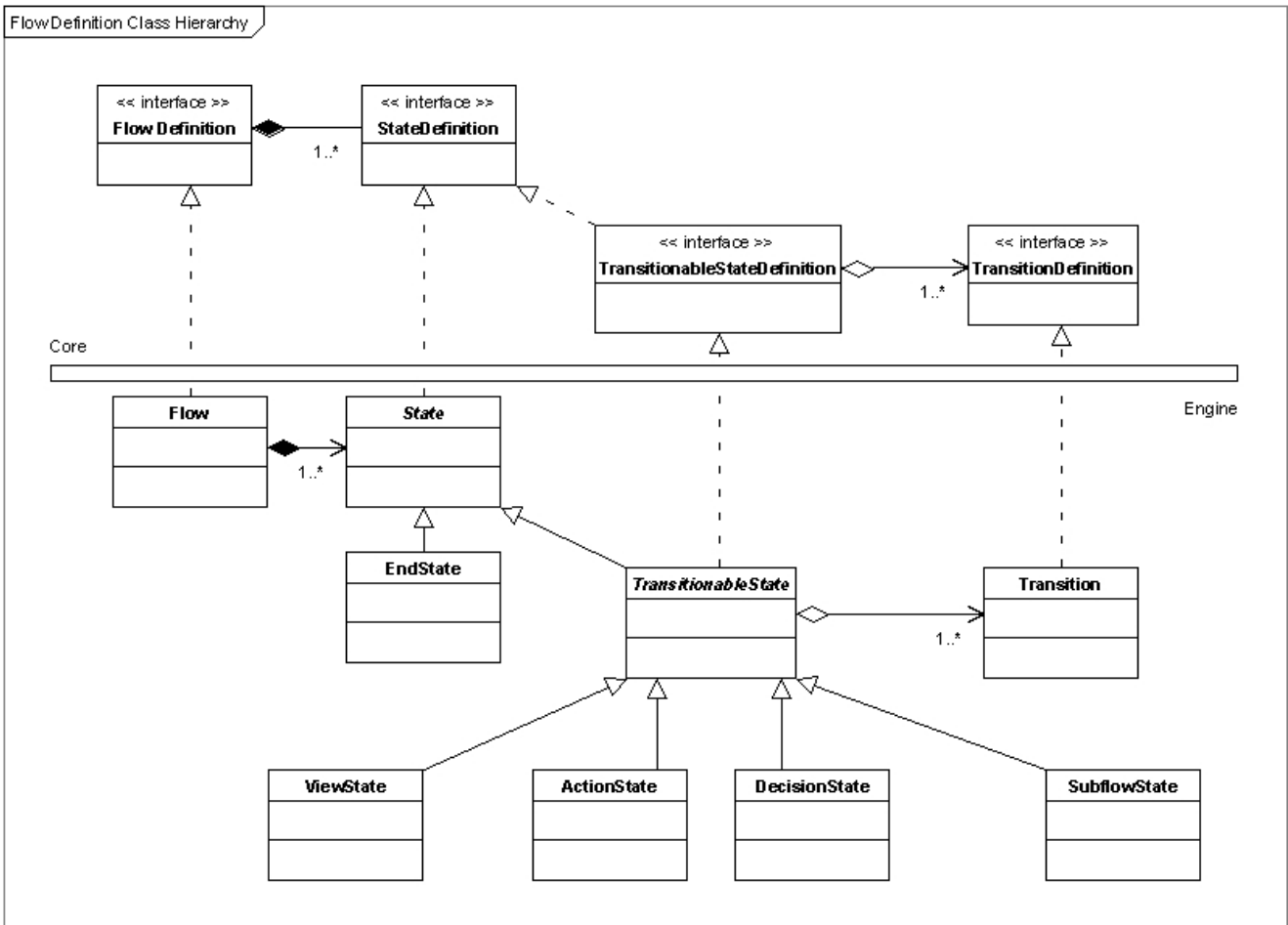
2.6. Concrete state types

Spring Web Flow has five (5) built-in concrete state types, all contained within the `org.springframework.webflow.engine` package. These states execute common controller behaviors

including:

1. allowing the user to participate in a flow (ViewState)
2. executing business application code (ActionState)
3. making a flow routing decision (DecisionState)
4. spawning another flow as a subflow (SubflowState)
5. terminating a flow (EndState)

Each of these state types, with the exception of EndState, is transitionable. This hierarchy is illustrated below:



FlowDefinition class diagram

As you will see, with these five basic state types you can develop rich controller modules.

2.6.1. ViewState

When entered a view state allows the user (or other external client) to participate in a flow. This participation process goes as follows:

1. The entered view state makes a `org.springframework.webflow.execution.ViewSelection` that represents a *logical* response to issue to the caller.
2. The flow execution 'pauses' in this state, and control is returned to the calling system.

3. The calling system uses the returned `ViewSelection` to present a suitable interface (or other response) to the user.
4. After some 'think time' the user signals an input event to resume the flow execution from the 'paused' point.

Spring Web Flow gives you full control over the view selection process and, on resume, how a view state responds to a user input event. It's important to understand that Spring Web Flow is *not* responsible for response rendering--as a controller, a flow makes a *logical* view selection when user input is required, where a view selection serves as a response instruction. It is up to the calling system to interpret that instruction to issue a response suitable for the environment in which the flow is executing.


The properties of a `org.springframework.webflow.engine.ViewState` are summarized below:

Table 2.5. ViewState properties

Property name	Description	Cardinality	Default value
<code>viewSelector</code>	The strategy that makes the view selection when this state is entered.	<i>0..1</i>	Null
<code>renderActions</code>	The list of actions to execute each time a renderable view selection is made. Allows for execution of pre-render logic.	<i>0..*</i>	Empty

The `org.springframework.webflow.execution.ViewSelection` base class is abstract, acting as a marker indicating a response should be issued to the client interacting with the flow. Concrete subtypes exist for each of the supported response types. These response types are summarized below:

Table 2.6. Concrete ViewSelection types

Type	Description
<code>ApplicationView</code>	Requests the rendering of a local, internal application view resource such as a JSP, Velocity, or Freemarker template.
<code>FlowExecutionRedirect</code>	Requests a redirect back to the <code>ViewState</code> at a unique <i>flow execution URL</i> . When this URL is accessed on subsequent requests an <code>ApplicationView</code> will be reconstituted and rendered. The URL is refreshable while the flow execution remains active.  <p>Note</p> <p>Multiple flow execution URLs may be generated for a single logical user conversation. In that case each flow execution URL provides access to the conversation from a previous point (<code>ViewState</code>). Accessing the URL refreshes the execution from that point.</p>
<code>FlowDefinitionRedirect</code>	Requests a redirect that launches an entirely new flow execution. Used to support <i>redirect to flow</i> (flow chaining) and <i>restart flow</i> use cases.
<code>ExternalRedirect</code>	Requests a redirect to an arbitrary external URL, typically used to interface with an external system.

Type	Description
NullView	Requests that no response be issued; for use in corner cases where the flow itself has already issued the response.

2.6.1.1. ViewSelector

The creational strategy responsible for making a `ViewSelection` when an `ViewState` is entered is `org.springframework.webflow.engine.ViewSelector`. This provides a plugin-point for customizing *how* a response instruction is constructed.

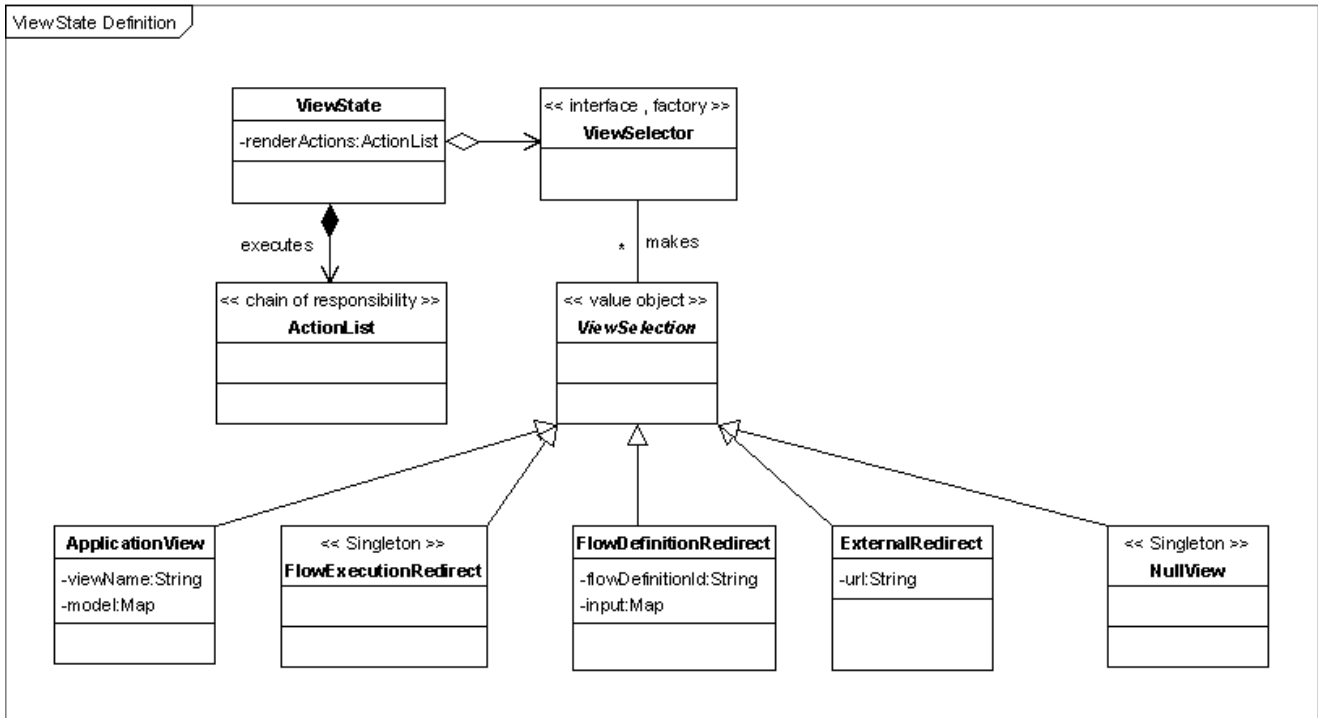
Four `ViewSelector` implementations are provided with Spring Web Flow:

Table 2.7. ViewSelector implementations

Implementation	Description
<code>ApplicationViewSelector</code>	Returns an <code>ApplicationView</code> referencing a logical <code>viewName</code> to render and containing a <code>modelMap</code> with the application data needed by the rendering process (by default, this map contains the union of the data scopes such flow, flash, and request scope). Supports setting a <code>redirect</code> flag that triggers a browser redirect to the selected view using a <code>FlowExecutionRedirect</code> . The default implementation.
<code>FlowDefinitionRedirectSelector</code>	Returns a <code>FlowDefinitionRedirect</code> with a <code>flowId</code> and <code>executionInput</code> map requesting the launch of an entirely new flow execution (an instance of the <code>FlowDefinition</code> identified by the <code>flowId</code>). Useful for <i>redirect after flow completion</i> , where one flow ending should trigger the start of another flow independently.
<code>ExternalRedirectSelector</code>	Returns an <code>ExternalRedirect</code> that triggers a browser redirect to an arbitrary external URL. Mainly used by end states to redirect to external systems after flow completion, but can also be used by view states to interface with an external system that may call back into the flow execution at a later point.
<code>NullViewSelector</code>	Returns an <code>NullView</code> indicating that no response should be issued.

2.6.1.2. ViewState class diagram

The class diagram below shows the `ViewState` and the associated types used to carry out the view selection process:



ViewState class diagram

2.6.1.3. ViewState XML - application view selection

The following example shows a `view-state` definition in XML that makes an application view selection when entered, selecting the `searchForm` view for display and, on resume, responding to two possible user input events (submit and cancel) in different ways:

```

<?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-1.0.xsd">

  <start-state idref="displaySearchForm"/>

  <view-state id="displaySearchForm" view="searchForm">
    <transition on="submit" to="processFormSubmission"/>
    <transition on="cancel" to="processCancellation"/>
  </view-state>

  ...

</flow>

```

View name expressions may also be specified for the `view` attribute to achieve runtime view name calculation. For example, `view="${requestScope.calculatedViewName}"`.

2.6.1.4. ViewState API - application view selection

The following example shows the equivalent view state definition using the FlowBuilder API:

```

public class SearchFlowBuilder extends AbstractFlowBuilder {
    public void buildStates() {
        addViewState("displaySearchForm", "searchForm",
            new Transition[] {
                transition(on("submit"), to("processFormSubmission")),
            }
        );
    }
}

```

```

        transition(on("cancel"), to("processFormCancellation"))
    };
    ...
}
}

```

2.6.1.5. ViewState XML - flow execution redirect

The following example illustrates a `view-state` definition in XML that makes an flow execution redirect selection when entered, redirecting to the `yourList` view for display.

```

<?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-1.0.xsd">

  <start-state idref="displayList"/>

  <view-state id="displayList" view="redirect:yourList">
    <transition on="add" to="addListItem"/>
  </view-state>

  ...

</flow>

```

This example is called a *flow execution redirect* because the application view selected is rendered only after a redirect to the flow execution. The redirect request is sent to a URL that *refreshes* the flow execution paused in the `displayList` view state. Refresh then triggers the rendering of the `yourList` application view on the next request into the server.

2.6.1.5.1. POST+REDIRECT+GET in Spring Web Flow

The above example is one way to achieve the `POST+REDIRECT+GET` pattern in Spring Web Flow. When the redirect is performed, the GET request issued hits a stable *flow execution URL* which remains active for the duration of the conversation. This URL may be freely refreshed. Browser navigational buttons may be used freely without browser warnings.

Later in this document the execution attribute `alwaysRedirectOnPause` is discussed, which enforces this pattern by default. In that case each time a view state is entered a redirect is always performed--*automatically*.

2.6.1.6. ViewState API - flow execution redirect

The following example shows the equivalent view state definition using the `FlowBuilder` API:

```

public class SearchFlowBuilder extends AbstractFlowBuilder {
    public void buildStates() {
        addViewState("displayList", viewSelector("redirect:yourView"),
                    transition(on("add"), to("addListItem")));
    };
    ...
}
}

```

2.6.1.7. ViewState XML - null view

The following example illustrates a `view-state` definition in XML that makes a null view selection when entered, which causes no additional response to be issued.

```
<?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-1.0.xsd">

  <start-state idref="displayPdf" />

  <view-state id="displayPdf">
    <render-actions>
      <action bean="pdfWriter" method="write" />
    </render-actions>
  </view-state>

  ...

</flow>
```

2.6.1.8. FlowDefinitionRedirect and ExternalRedirect

The `FlowDefinitionRedirect` and `ExternalRedirect` are not normally used with a view state. Instead they're used in an end state to continue with another, independent flow or to redirect to an external URL. Examples are provided in the discussion of the end state.

2.6.1.9. ViewState XML - form state behavior

The following example illustrates a `view-state` definition in XML that encapsulates typical "form state" behavior.

Consider the requirements of typical input forms. Most forms require *pre-render* or *setup* logic to execute before the form is displayed. For example, such logic might load the *backing form object* from the database, install formatters for formatting form field values, and pull in supporting form data needed to populate drop-down menus.

In addition, most forms require *post-back* or *submission* logic to execute when the form is submitted. This logic typically involves binding form input to the *backing form object* and performing type conversion and data validation.

This "form state" behavior of form setup, display, and post-back is handled elegantly in Spring Web Flow by the capabilities of the `view-state` construct. See below:

```
<?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-1.0.xsd">

  <start-state idref="displayForm" />

  <view-state id="displayForm" view="form">
    <render-actions>
      <action bean="formAction" method="setupForm" />
      <action bean="formAction" method="loadFormReferenceData" />
    </render-actions>
    <transition on="submit" to="saveForm">
```

```

        <action bean="formAction" method="bindAndValidate"/>
    </transition>
</view-state>

    ...

</flow>

```

This reads "when this flow starts enter the `displayForm` state to execute the `setupForm` and `loadFormReferenceData` methods before rendering the `form` view. On submit, transition to the `saveForm` state if the `bindAndValidate` method executes successfully."

2.6.2. ActionState

When entered, an action state executes business application code, then responds to the result of that execution by deciding what state in the flow to enter next. Specifically:

1. The entered action state executes an ordered list of one or more `org.springframework.webflow.execution.Action` instances. This `Action` interface is the central abstraction that encapsulates the execution of a logical unit of application code.
2. The state determines if the outcome of the first action's execution matches a transition. If there is a match, the transition is executed. If there is no match, the next action in the list is executed. This process continues until a transition is matched or the list of actions is exhausted.

Spring Web Flow gives you full control over implementing your own actions and configuring when they should be invoked within the lifecycle of a flow. The system can also *automatically* adapt methods on your existing application objects (POJOs) to the `Action` interface in a non-invasive manner. This means in many cases you can implement your flows without needing to develop custom glue code to bind SWF to your service layer operations.

The properties of a `org.springframework.webflow.engine.ActionState` are summarized below:

Table 2.8. ActionState properties

Property name	Description	Cardinality	Default value
actions	The ordered list of actions to execute when the state is entered.	1..*	

2.6.2.1. Action execution points

As outlined, the `ActionState` is the dedicated state type for invoking one or more actions and responding to their result to drive a state transition. There are also other points within the lifecycle of a flow where a chain of actions can be executed. At all of these points the only requirement is that these actions implement the central `org.springframework.webflow.execution.Action` interface.

Table 2.9. Other points in a Flow where an Action can be executed and how those points can be defined in a XML-based Flow definition.

Point	Description	XML Configuration Element
on flow start	Each time a new flow session starts.	A flow's <code><start-actions/></code>
on state entry	Each time a state enters.	A state's <code><entry-actions/></code>
on transition	Each time a state transition is matched but before it is executed.	A transition <code><action/></code>
on state exit	Each time a transitionable state exits.	A transitionable state's <code><exit-actions/></code>
before view rendering	Each time a renderable view selection is made.	A view state's <code><render-actions/></code>
on flow end	Each time a flow session terminates.	A flow's <code><end-actions/></code>



Note

The above other points in a flow where actions may be executed do not allow you to execute a state transition in response to the action result event. If you need such flow control you must execute the action from within an action state.

2.6.2.2. Action attributes

An `Action` may be annotated with attributes by wrapping the `Action` in a decorator, an instance of `org.springframework.webflow.engine.AnnotatedAction`. These attributes may provide descriptive characteristics, or may be used to affect the action's execution in a specific usage context.

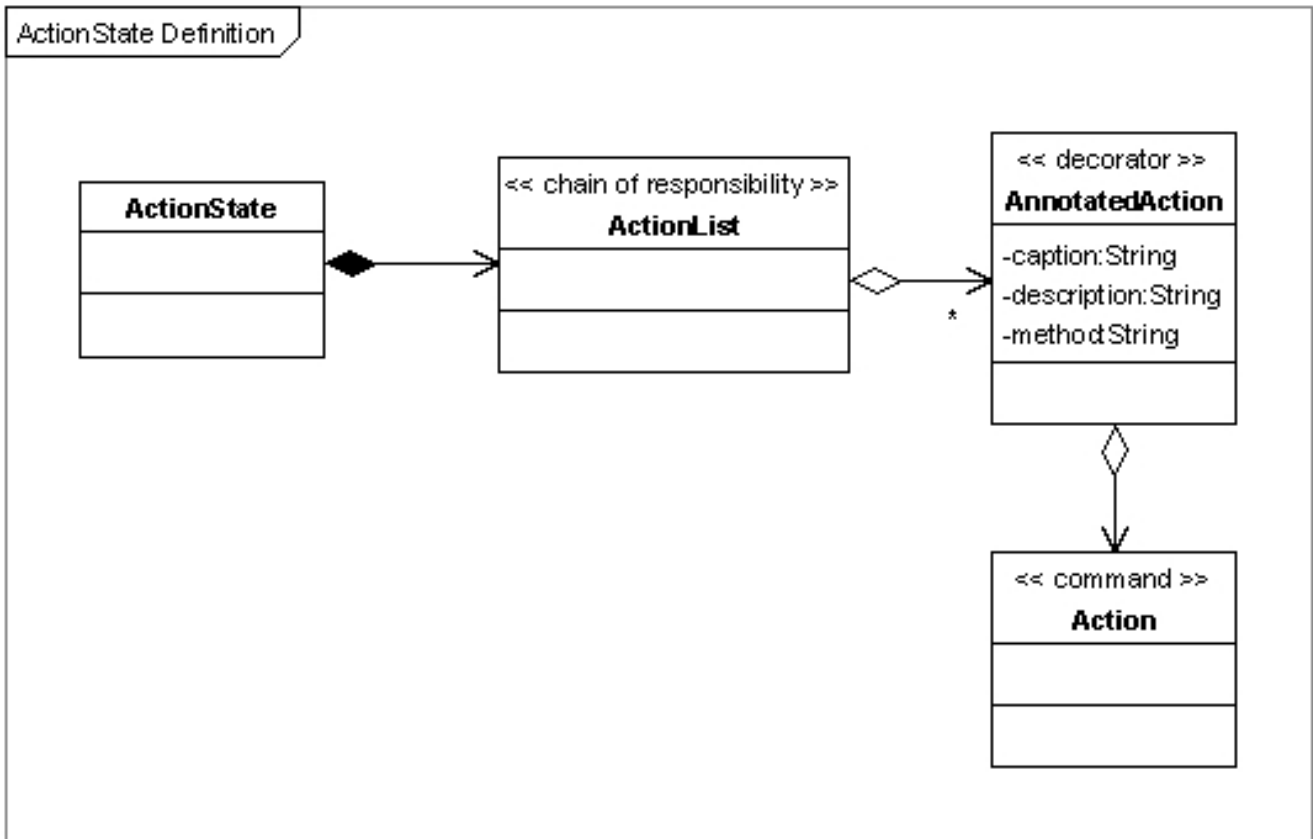
Support for setting several common attributes is provided for convenience. These include:

Table 2.10. Common Action attributes

Attribute name	Description
caption	A short description about the action, suitable for display as a tooltip.
description	A long description about the action, suitable for display in a text box.
name	The name of the action, used to qualify the action's result event. For example, an <code>Action</code> named <code>placeOrder</code> that returns <code>success</code> would be assigned a result event identified by <code>placeOrder.success</code> . This allows you to distinguish logical execution outcomes by action, useful when invoking multiple actions as part of a chain.
method	The name of the target method on the <code>Action</code> instance to invoke to carry out execution. This facilitates multiple <i>action methods</i> per <code>Action</code> instance, supported by the <code>org.springframework.webflow.action.MultiAction</code> .

2.6.2.3. ActionState class diagram

The class diagram below shows the `ActionState` and the associated types used to carry out the action execution process:



ActionState class diagram

2.6.2.4. ActionState XML - simple action execution

The following example constructs an `ActionState` definition from XML that executes a single action when entered and then responds to its result:

```

<?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-1.0.xsd">

  <start-state idref="executeSearch"/>

  <action-state id="executeSearch">
    <action bean="searchAction"/>
    <transition on="success" to="displayResults"/>
  </action-state>

  ...

</flow>

```

This state definition reads *"when the executeSearch state is entered, execute the searchAction. On successful execution, transition to the displayResults state."*

The binding between the `searchAction` id and an `Action` implementation is made at Flow build time by querying a service locator, typically a Spring BeanFactory. For example:

```

<beans>
  <bean id="searchAction" class="example.webflow.SearchAction"/>
</beans>

```

... binds the `searchAction` action identifier to a singleton instance of the `example.webflow.SearchAction` class.

A simple `SearchAction` implementation might look like this:

```
public class SearchAction implements Action {
    private SearchService searchService;

    public SearchAction(SearchService searchService) {
        this.searchService = searchService;
    }

    public Event execute(RequestContext context) {
        // lookup the search criteria in "flow scope"
        SearchCriteria criteria =
            (SearchCriteria)context.getFlowScope().get("criteria");

        // execute the search
        Collection results = searchService.executeSearch(criteria);

        // set the results in "request scope"
        context.getRequestScope().put("results", results);

        // return "success"
        return new Event(this, "success");
    }
}
```

2.6.2.5. ActionState API - standard action

The following example constructs the equivalent action state definition using the `FlowBuilder` API:

```
public class SearchFlowBuilder extends AbstractFlowBuilder {
    public void buildStates() {
        ...
        addActionState("executeSearch", action("searchAction"),
            transition(on("success"), to("displayResults")));
        ...
    }
}
```

2.6.2.6. ActionState XML - multi action

The next example constructs an `ActionState` definition from XML that executes a single *action method* on a `org.springframework.webflow.action.MultiAction` and then responds to its result:

```
<?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-1.0.xsd">

    <start-state idref="executeSearch"/>

    <action-state id="executeSearch">
        <action bean="searchAction" method="executeSearch"/>
        <transition on="success" to="displayResults"/>
    </action-state>

    ...

</flow>
```

This state definition reads *"when the executeSearch state is entered, call the executeSearch method on the searchFlowAction. On successful execution, transition to the displayResults state."*

A SearchAction implementation containing multiple action methods might look like this:

```
public class SearchAction extends MultiAction {
    private SearchService searchService;

    public SearchAction(SearchService searchService) {
        this.searchService = searchService;
    }

    public Event executeSearch(RequestContext context) {
        // lookup the search criteria in "flow scope"
        SearchCriteria criteria =
            (SearchCriteria)context.getFlowScope().get("criteria");

        // execute the search
        Collection results = searchService.executeSearch(criteria);

        // set the results in "request scope"
        context.getRequestScope().put("results", results);

        // return "success"
        return success();
    }

    public Event someOtherRelatedActionMethod(RequestContext context) {
        ...
        return success();
    }

    public Event yetAnotherRelatedActionMethod(RequestContext context) {
        ...
        return success();
    }
}
```

As you can see, this allows you to define one to many action methods per Action class. With this approach, there are two requirements:

1. Your Action class must extend from `org.springframework.webflow.MultiAction`, or another class that extends from `MultiAction`. The multi action cares for the action method dispatch that is based on the value of the `method` property.
2. Each action method must conform to the signature illustrated above: `public Event ${method}(RequestContext) { ... }`

MultiActions are useful for centralizing command logic on a per-flow definition basis, as a flow definition typically carries out execution of a single application use case.

2.6.2.7. ActionState API - multi action

The following example constructs the equivalent action state definition using the FlowBuilder API:

```
public class SearchFlowBuilder extends AbstractFlowBuilder {
    public void buildStates() {
        ...
        addActionState("executeSearch", invoke("executeSearch", action("searchAction")),
            transition(on("success"), to("displayResults")));
        ...
    }
}
```

2.6.2.8. ActionState XML - bean action

The next example constructs an `ActionState` definition from XML that executes a single method on a Plain Old Java Object (POJO) and then responds to the result:

```
<?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-1.0.xsd">

  <start-state idref="executeSearch"/>

  <action-state id="executeSearch">
    <bean-action bean="searchService" method="executeSearch">
      <method-arguments>
        <argument expression="${flowScope.criteria}"/>
      </method-arguments>
      <method-result name="results"/>
    </bean-action>
    <transition on="success" to="displayResults"/>
  </action-state>

  ...

</flow>
```

This state definition reads *"when the `executeSearch` state is entered, call the `executeSearch` method on the `searchService` passing it the object indexed by name `criteria` in `flowScope`. On successful execution, expose the method return value in the default scope (request) under the name `results` and transition to the `displayResults` state."*

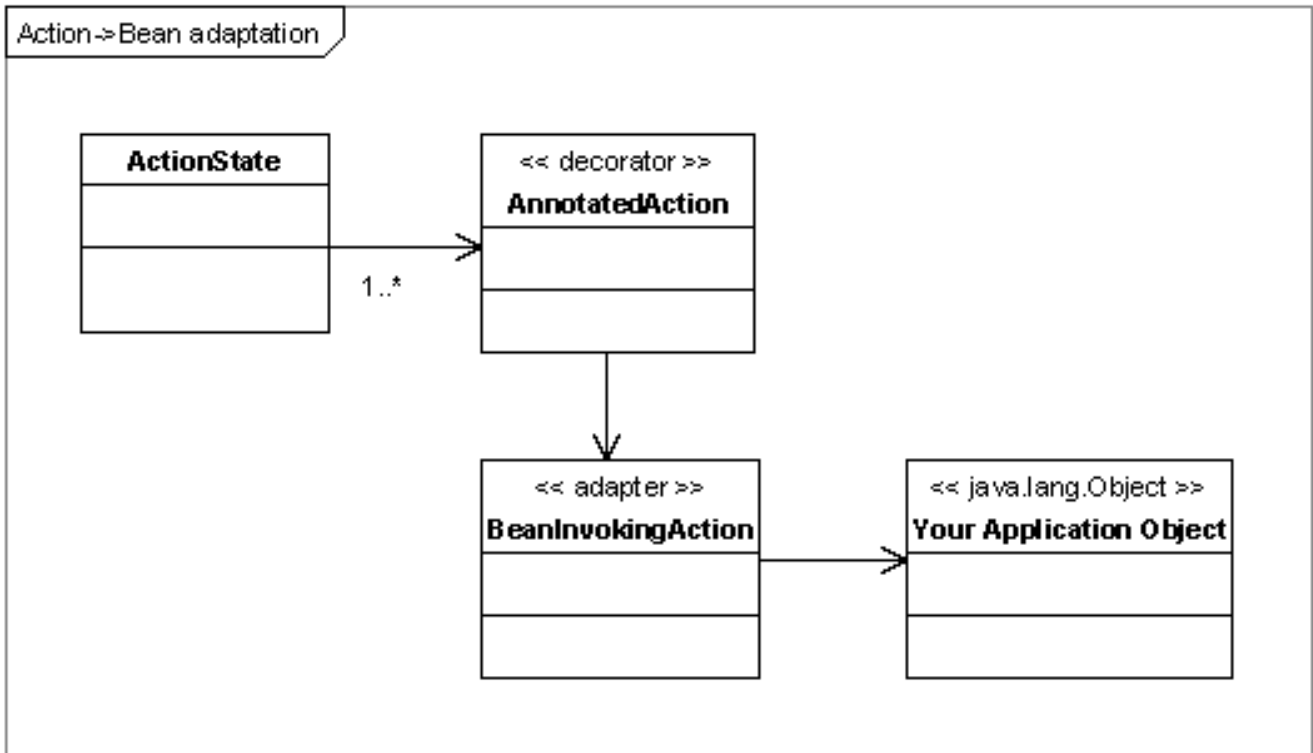
In this example the referenced bean `searchService` would be *your application object*, typically a transactional business service. Such a service implementation must have defined the the `Collection executeSearch(SearchCriteria)` method, typically by implementing a service interface:

```
public interface SearchService {
    public Collection executeSearch(SearchCriteria criteria);
}
```

With this approach there are no requirements on the signature of the methods that carry out action execution, nor is there any requirement to extend from a Web Flow specific base class. Basically, you are not required to write a custom `Action` implementation at all--you simply instruct Spring Web Flow to call your business methods directly. The need for custom "glue code" to bind your web-tier to your middle-tier is eliminated.

Spring Web Flow achieves this by automatically adapting the method on your existing application object to the `Action` interface and caring for exposing any return value in the correct scope.

This adaption process is shown graphically below:



Bean->Action adapter

2.6.2.9. ActionState XML - decision bean action

The following example constructs an `ActionState` from XML that executes an action whose execution result forms the basis for the transition decision:

```

<?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-1.0.xsd">

  ...

  <action-state id="shippingRequired">
    <bean-action bean="shippingService" method="isShippingRequired">
      <method-arguments>
        <argument expression="${flowScope.purchase}"/>
      </method-arguments>
    </bean-action>
    <transition on="yes" to="enterShippingDetails"/>
    <transition on="no" to="placeOrder"/>
  </action-state>

  ...

</flow>

```

This state definition reads *"if the `isShippingRequired` method on the `shippingService` returns true, transition to the `enterShippingDetails` state, otherwise transition to the `placeOrder` state."*



Note

Note how the boolean return value of the `isShippingRequired` method is converted to the event identifiers `yes` or `no`.

This conversion process is handled by the action adapter responsible for adapting the method on your application object to the `org.springframework.webflow.execution.Action` interface. By default, this adapter applies a number of rules for creating a result event from a method return value.

These conversion rules are:

Table 2.11. Default method return value to Event conversion rules

Return type	Event identifier
boolean	yes or no
java.lang.Enum	this.name()
org.springframework.core.enum.LabeledEnum	this.getLabel()
org.springframework.webflow.execution.Event	this.getId()
java.lang.String	the string
any other type	success

You may customize these default conversion policies by setting a custom `ResultEventFactory` instance on the bean invoking action performing the adaption. Consult the JavaDoc documentation for more details on how to do this.

2.6.2.10. ActionState XML - decision bean action with enum return value

The following example constructs an `ActionState` from XML that executes a action that invokes a method on an application object that returns a `java.lang.Enum`:

```
<?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-1.0.xsd">
  ...
  <action-state id="shippingRequired">
    <bean-action bean="shippingService" method="calculateShippingMethod"/>
    <method-arguments>
      <argument expression="\${flowScope.order}"/>
    </method-arguments>
  </bean-action>
  <transition on="BASIC" to="enterBasicShippingDetails"/>
  <transition on="EXPRESS" to="enterExpressShippingDetails"/>
  <transition on="NONE" to="placeOrder"/>
</action-state>
  ...
</flow>
```

This state definition reads *"if the `calculateShippingMethod` method on the `shippingService` returns `BASIC` for the current order, transition to the `enterBasicShippingDetails` state. If the return value is `EXPRESS` transition to the `enterExpressShippingDetails` state. If the return value is `NONE` transition to the `placeOrder` state."*

2.6.2.11. ActionState XML - evaluate action

The following example constructs an `ActionState` from XML that executes a action that evaluates an expression against the flow request context and exposes the evaluation result:

```
<?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-1.0.xsd">

  <action-state id="getNextInterviewQuestion">
    <evaluate-action expression="flowScope.interview.nextQuestion()"/>
    <evaluation-result name="question"/>
  </evaluate-action>
  <transition on="success" to="displayQuestion"/>
</action-state>

</flow>
```

This state definition reads *"evaluate the `flowScope.interview.nextQuestion()` expression and expose the result under name `question` in the default scope."*

The expression can evaluate any object traversable from the flow's `org.springframework.webflow.execution.RequestContext`. This example expression evaluates the `nextQuestion` method on the `interview` business object in flow scope.

2.6.2.12. ActionState XML - set action

The next example constructs an `ActionState` from XML that executes an action on a success transition that sets an attribute in "flash scope":

```
<?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-1.0.xsd">

  <view-state id="selectFile" view="fileUploadForm">
    <transition on="submit" to="uploadFile"/>
  </view-state>

  <action-state id="uploadFile">
    <action bean="uploadAction" method="uploadFile"/>
    <transition on="success" to="selectFile">
      <set attribute="fileUploaded" scope="flash" value="true"/>
    </transition>
  </action-state>

</flow>
```

This flow definition reads *"display the `fileUploadForm`. On form submit invoke the `uploadFile` method on the `uploadAction`. On success allow the user to select another file to upload. Report that the last file was uploaded successfully by setting the `fileUploaded` attribute in flash scope to `true`."*



Note

Flash scoped attributes are preserved until the next user event is signaled into the flow execution.

In this example this means the `fileUploaded` attribute is preserved across a redirect to the `selectFile` view state and any subsequent browser refreshes. Only when the `submit` event is signaled will the flash scope be cleared.

2.6.2.13. When to use which kind of action?

Simple action, Multi action, bean action, evaluate action, set? When to use one or the other?

Table 2.12. Action implementation usage guidelines

Action type	Usage scenario
Simple (extends <code>AbstractAction</code>)	You have a specialized behavior that stands on its own; for creating lightweight stubs or mocks for testing purposes.
<code>MultiAction</code>	To group related command logic together. Particularly useful for when there are multiple related behaviors called by a flow.
Bean action	When the logical behavior maps well to a method call on a service layer bean. When there is no "special" or exotic glue code required.
<code>EvaluateAction</code>	When you need to invoke a bean in flow scope or evaluate any other flow expression.
<code>SetAction</code>	When you need to set an attribute in flow or other scope during the course of flow execution.

2.6.3. DecisionState

When entered, a decision state makes a flow routing decision. This process consists of:

1. Evaluating one or more boolean expressions against the executing flow to decide what state to transition to next.

The properties of a `org.springframework.webflow.engine.DecisionState` are summarized below:

Table 2.13. DecisionState properties

Property name	Description	Cardinality	Default value
transitions (inherited from <code>TransitionableState</code>)	The transitions that are evaluated on an event occurrence that forms the basis for the decision.	1..*	

2.6.3.1. DecisionState XML - expression evaluation

The following example constructs a `DecisionState` from XML that evaluates a boolean expression to determine what transition to execute:

```
<?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-1.0.xsd">
    ...

    <decision-state id="shippingRequired">
      <if test="{flowScope.order.needsShipping}" then="enterShippingDetails" else="placeOrder" />
    </decision-state>

    ...

  </flow>

```

This state definition reads "if the `needsShipping` property on the `order` object in flow scope is true, transition to the `enterShippingDetails` state, otherwise transition to the `placeOrder` state."



Note

Caution: flow definitions should *not* be vehicles for business logic. In this case the decision made was controller logic, reasoning on a pre-calculated value to decide what step of the flow to transition to next. That is the kind of logic that should be in a flow definition. In contrast, having the state *itself* embed the business rule defining how shipping status is calculated is a misuse. Instead, push such a calculation into application code where it belongs and *instruct* the flow to invoke that code using an action.

2.6.4. SubflowState

When entered, a subflow state spawns another flow as a subflow.

Recall that a flow is a reusable, self-contained controller module. The ability for one flow to *call* another flow gives you the ability to compose independent modules together to create complex controller workflows. Any flow can be used as subflow by any other flow, and there is a well-defined contract in play. Specifically:

1. A Flow is an instance of `org.springframework.webflow.engine.Flow`.
2. A newly launched flow can be passed input attributes which it may choose to map into its own local scope.
3. An ending flow can return output attributes. If the ended flow was launched as a subflow, the resuming parent flow may choose to map these output attributes into its own scope.

It is helpful to think of the process of calling a flow like calling a Java method. Flows can be passed input arguments and can produce return values just like methods can. Flows are more powerful because they are potentially long-running, as they can span more than one request into the server.

The properties of a `org.springframework.webflow.engine.SubflowState` are summarized below:

Table 2.14. SubflowState properties

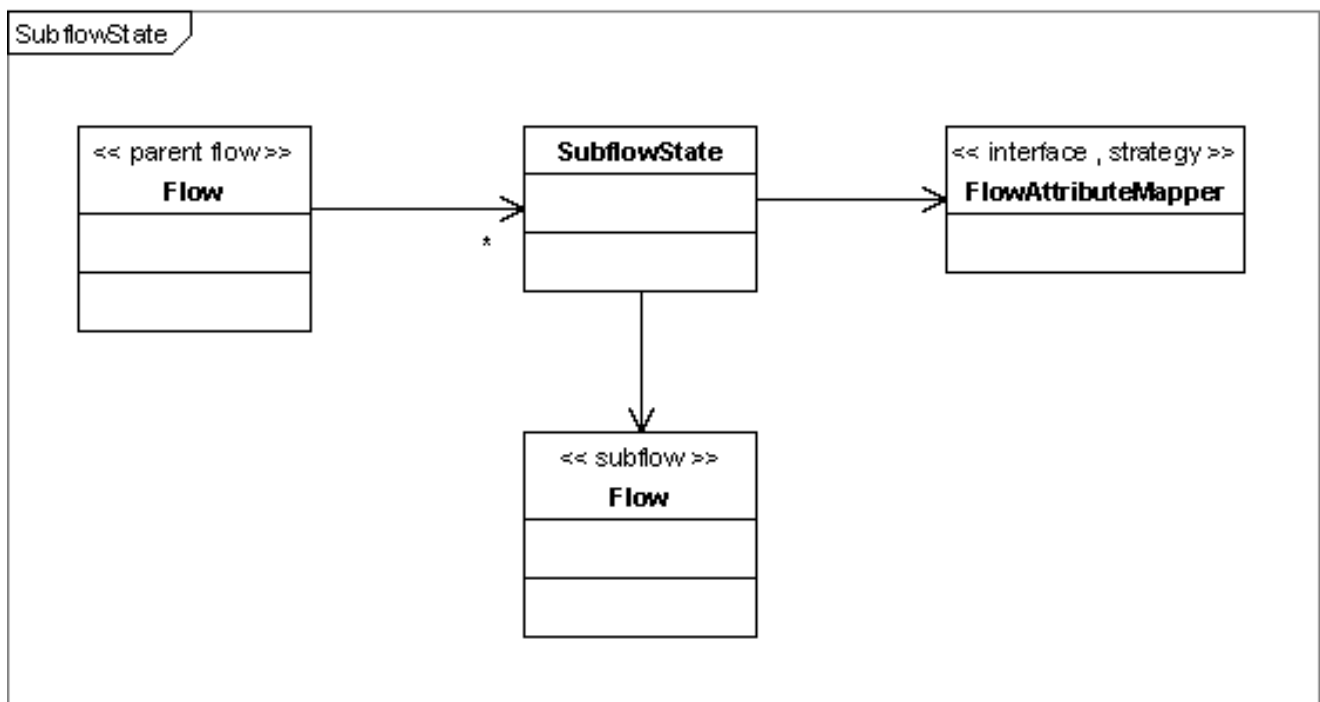
Property name	Description	Cardinality	Default value
subflow	The definition of the flow to be spawned as a subflow.	1	
attributeMapper	The strategy responsible for mapping input attributes to the subflow and	0..*	Null

Property name	Description	Cardinality	Default value
	mapping output attributes from the subflow.		

When a SubflowState is entered, the following behavior occurs:

1. The state first messages its `attributeMapper`, an instance of `org.springframework.webflow.engine.FlowAttributeMapper`, to prepare a `Map` of input attributes to pass to the subflow.
2. The subflow is spawned, passing the input attributes. When this happens, the parent flow *suspends* itself in the subflow state until the subflow ends.
3. When the subflow ends, a *result event* is returned describing the flow outcome that occurred. The parent flow *resumes* back in the subflow state.
4. The resumed subflow state messages its `attributeMapper` to map any output attributes returned by the subflow into flow scope, if necessary.
5. Finally, the resumed subflow state responds to the result event returned by the ended subflow by matching and executing a state transition.

The constructs used in spawning a flow as a subflow are shown graphically below:



SubflowState class diagram

2.6.4.1. SubflowState XML - with input attribute

The following example constructs an `SubflowState` from XML that spawns a shipping subflow:

```

<?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-1.0.xsd">

...

<subflow-state id="enterShippingDetails" flow="shipping">
  <attribute-mapper>
    <input-mapper>
      <mapping source="flowScope.order.shipping" target="shipping"/>
    </input-mapper>
  </attribute-mapper>
  <transition on="finish" to="placeOrder"/>
</subflow-state>

...

</flow>

```

This subflow state definition reads *"spawn the shipping flow and pass it the value of the shipping property on the order object in flow scope as an input attribute with the name shipping. When the shipping flow ends, respond to the finish result event by transitioning to the placeOrder state."*



Note

The inner structure and behavior of the `shipping` flow is fully encapsulated within its own flow definition. A flow calling another flow as a subflow can pass that flow input and capture its output, but it cannot see inside it. Flows are *black boxes*. Because any flow can be used as a subflow, it can be reused in other contexts without change.

2.6.4.2. SubflowState API - input attributes

The following illustrates the equivalent example using the `FlowBuilder` API:

```

public class OrderFlowBuilder extends AbstractFlowBuilder {
    public void buildStates() {
        ...
        addSubflowState("enterShippingDetails", flow("shipping"), shippingMapper(),
            transition(on("finish"), to("placeOrder")));
        ...
    }

    protected FlowAttributeMapper shippingMapper() {
        DefaultFlowAttributeMapper mapper = new DefaultFlowAttributeMapper();
        mapper.addInputMapping(mapping().source("flowScope.order.shipping").target("shipping").value());
        return mapper;
    }
}

```

2.6.4.3. Flow input mapping - input contract

Internally within the definition of the `shipping` flow referenced above, the flow may choose to map the `shipping` input attribute into its own scope using its input mapper when it starts. Any input attributes must be explicitly mapped, defining the input contract for the flow:

```

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

```

```
<input-mapper>
  <input-attribute name="shipping" />
</input-mapper>

...

</flow>
```

This short-form input mapper declaration reads "*when a new execution of this flow starts map the shipping input attribute into flowScope under the name shipping.*"



Note

Had this input mapping not been defined the shipping attribute made available as input to this flow by a calling parent flow or external client would have been ignored.

2.6.5. EndState

When entered, an end state terminates a flow. A `EndState` represents exactly one logical flow outcome; for example, "finish", or "cancel".

If the ended flow was acting as a top-level or *root flow* the entire flow execution ends and cannot be resumed. In this case the end state is responsible for making a `ViewSelection` that is the basis for the ending response (for example, a confirmation page, or a redirect request to another flow or an external URL).

If the ended flow was acting as a subflow, the spawned subflow session ends and the calling parent flow *resumes* by responding to the end result returned. In this case the responsibility for any `ViewSelection` falls on the parent flow.

Once a flow ends any attributes in flow scope go out of scope immediately and become eligible for garbage collection.

As outlined, an end state entered as part of a root flow messages its `ViewSelector` to make a ending view selection. Typically this is a redirect-based `ViewSelector`, allowing for *redirect after flow completion*. An end state entered as part of a subflow is not responsible for a view selection; this responsibility falls on the calling flow.

2.6.5.1. EndState result events

When a `EndState` is entered it terminates a flow and, if used as subflow, returns a result event the parent flow uses to drive a state transition from the calling subflow state. It is the end state's responsibility to create this result event which is the basis for communicating the *logical flow outcome* to callers.

By default, an `EndState` creates a result event with an identifier that matches the identifier of the end-state itself. For example, an end state with id `finish` returns a result event with id `finish`. Also, any attributes in flow scope that have been explicitly mapped as *output attributes* are returned as result event parameters. This allows you to return data along with the logical flow outcome.

Spring Web Flow gives you full control over the ending view selection strategy, as well as what flow attributes should be exposed as output on a per `EndState` basis. These configurable properties are summarized below:

2.6.5.2. EndState Properties

Table 2.15. EndState properties

Property name	Description	Cardinality	Default value
viewSelector	The strategy that makes the ending view selection when this state is entered and the flow is a root flow.	0..1	Null
outputMapper	The service responsible for exposing flow output attributes, making those attributes eligible for output mapping by a calling flow.	0..1	None

2.6.5.3. EndState XML - redirect to flow after completion

The following example constructs an `EndState` from XML that terminates a shipping subflow and requests a redirect response to another flow:

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="
        http://www.springframework.org/schema/webflow
        http://www.springframework.org/schema/webflow/spring-webflow-1.0.xsd">
  ...
  <end-state id="finish" view="flowRedirect:searchFlow"/>
</flow>
```

This end state definition reads *"terminate the `order` flow and redirect to a new execution of the `searchFlow`".*

2.6.5.4. EndState XML - redirect after flow completion

The following example constructs an `EndState` from XML that terminates a shipping subflow and requests a redirect response to an external URL:

```
<?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-1.0.xsd">
  ...
  <end-state id="finish" view="externalRedirect:/orders/${flowScope.order.id}"/>
</flow>
```

This end state definition reads *"terminate the `order` flow and redirect to the URL returned by evaluating the `/orders/${flowScope.order.id}` expression."*

This is an example of the familiar *redirect after post* pattern where after transaction completion a redirect is issued allowing the result of the transaction to be viewed (in this case using REST-style URLs).

2.6.5.5. EndState XML - flow output attribute

The following example constructs an `EndState` from XML that terminates a shipping subflow:

```
<?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-1.0.xsd">

  ...

  <end-state id="finish">
    <output-mapper>
      <output-attribute name="shipping"/>
    </output-mapper>
  </end-state>

</flow>
```

This end state definition reads *"terminate the shipping flow and expose the shipping property in flow scope as an output attribute with name shipping."*

2.6.5.6. EndState API - flow output attribute

The following illustrates the equivalent example using the `FlowBuilder` API:

```
public class ShippingFlowBuilder extends AbstractFlowBuilder {
    public void buildStates() {
        ...
        addEndState("finish",
            new DefaultAttributeMapper().add(
                mapping().source("flowScope.shipping").target("shipping").value()
            );
    }
}
```

Since this end-state does not make a view selection it is expected this flow will be always used as a subflow. When this flow ends, the calling parent flow is expected to respond to the `finish` result, and may choose to map the `shipping` output attribute into its own scope.

2.6.5.7. SubflowState XML - mapping an output attribute

The next example shows how a `subflow-state` can respond to the ending result of a subflow and map output attributes into its own scope:

```
<?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-1.0.xsd">

  ...

  <subflow-state id="enterShippingDetails" flow="shipping">
    <attribute-mapper>
      <output-mapper>
        <output-attribute name="shipping"/>
      </output-mapper>
    </attribute-mapper>

  </subflow-state>

</flow>
```

```
    <transition on="finish" to="placeOrder"/>
  </subflow-state>

  ...

</flow>
```

This subflow state definition reads *"spawn the shipping flow as a subflow. When the shipping flow ends map the shipping output attribute into flow scope under the name shipping, then respond to the finish result event by transitioning to the placeOrder state."*



Note

Had this output mapping not been defined the shipping attribute made available as output to this flow by the ending subflow would have been ignored.

Chapter 3. Flow execution

3.1. Introduction

Once a flow has been defined any number of executions of it can be launched in parallel at runtime. Execution of a flow is carried out by a dedicated system that is based internally on a state machine that runs atop the Java VM. As the life of a flow execution can span more than one request into the server, this system is also responsible for persisting execution state across requests.

This chapter documents Spring Web Flow's flow execution system. You'll learn the core constructs of the system and how to execute flows out-of-container within a JUnit test environment.

3.2. FlowExecution

A `org.springframework.webflow.execution.FlowExecution` is a runtime instantiation of a flow definition. Given a single `FlowDefinition` any number of independent flow executions may be created, typically by a `FlowExecutionFactory`.

A flow execution carries out the execution of program instructions defined within its definition in response to user events.

It may be helpful to think of a flow definition as analagous to a Java `Class` and a flow execution as analagous to an object instance of that `Class`. Signaling an execution event can be considered analagous to sending an object a message.

3.2.1. Flow execution creation

```
FlowDefinition definition = ...
FlowExecutionFactory factory = ...
FlowExecution execution = factory.createFlowExecution(definition);
```

Once created, a new flow execution is initially inactive, waiting to be started. Once started a flow execution becomes active by entering its `startState`. From there it continues executing until it enters a state where user input is required to continue or it terminates.

3.2.2. Flow execution startup

```
MutableAttributeMap input = ...
ExternalContext context = ...
ViewSelection startingView = execution.start(input, context);
```

When a flow execution reaches a state where input is required to continue it is said to have *paused*, where it waits in that state for the input to be provided. After pausing the `ViewSelection` returned is typically used to issue a response to the user that provides a vehicle for collecting the required input.

User input is provided by *signaling an event* that *resumes* the flow execution by communicating what user

action was taken. Attributes of the signal event request form the basis for user input. The flow execution resumes by consuming the event.

Once a flow execution has resumed it continues executing until it again enters a state where more input is needed or it terminates. Once a flow execution has terminated it becomes inactive and cannot be resumed.

3.2.3. Flow execution resume

```

ExternalContext context = ...
ViewSelection nextView = execution.signalEvent("submit", context);
if (execution.isActive()) {
    // still active but paused
} else {
    // has ended
}

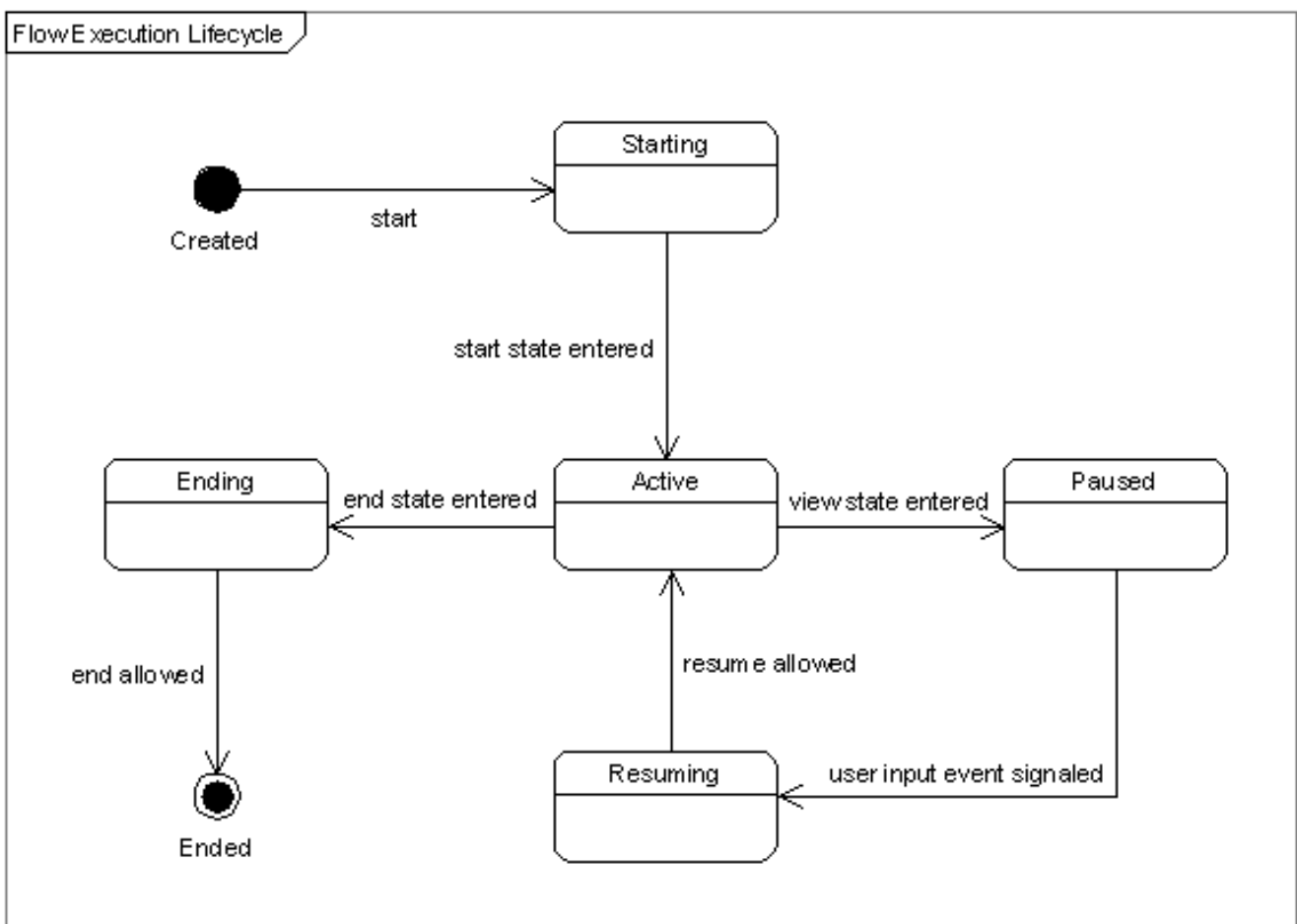
```

3.2.4. Flow execution lifecycle

As outlined, a flow execution can go through a number of phases throughout its lifecycle; for example, *created*, *active*, *paused*, *ended*.

Spring Web Flow gives you the ability to observe the lifecycle of an executing flow by implementing a `FlowExecutionListener`.

The different phases of a flow execution are shown graphically below:



Flow execution lifecycle

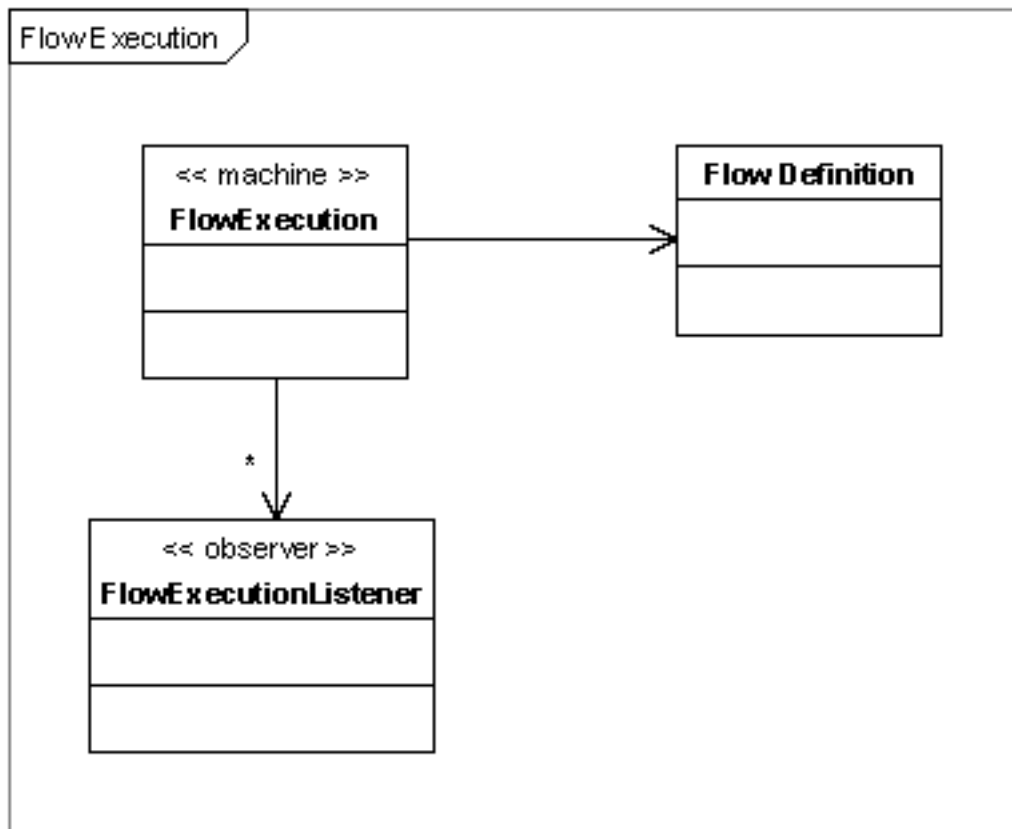
3.2.5. Flow execution properties

The Spring Web Flow flow execution implementation is `org.springframework.webflow.engine.impl.FlowExecutionImpl`, typically created by a `FlowExecutionImplFactory` (a `FlowExecutionFactory` implementation). The configurable properties of this flow execution implementation are summarized below:

Table 3.1. Flow Execution properties

Property name	Description	Cardinality	Default value
definition	The flow definition to be executed.	1	
listeners	The set of observers observing the lifecycle of this flow execution.	0..*	Empty
attributes	Global system attributes that can be used to affect flow execution behavior	0..*	Empty

The configurable constructs related to flow execution are shown graphically below:



Flow execution

3.2.6. Flow execution impl creation

```

FlowExecutionFactory factory = new FlowExecutionImplFactory();
factory.setExecutionListeners(...);
factory.setExecutionAttributes(...);
FlowExecution execution = factory.createFlowExecution(definition);
  
```

3.3. Flow execution context

Once created, a flow execution, representing the state of a flow at a point in time, maintains contextual state about itself that can be reasoned upon by clients. In addition, a flow execution exposes several data structures, called scopes, that allow clients to set arbitrary attributes that are managed by the execution.

The contextual properties associated with a flow execution are summarized below:

Table 3.2. Flow Execution Context properties

Property name	Description	Cardinality	Default value
active	A flag indicating if the flow execution is active. An inactive flow execution has either ended or has never been started.	1	
definition	The definition of the flow execution. The flow definition serves as the blueprint for the program. <i>It may be helpful to think of a flow definition as like a <code>Class</code> and a flow execution as like an instance of that <code>Class</code>.</i> This method may always be safely called.	1	
activeSession	The active flow session, tracking the flow that is currently executing and what state it is in. The active session can change over the life of the flow execution because a flow can spawn another flow as a subflow. This property can only be queried while the flow execution is active.	1	
conversationScope	A data map that forms the basis for "conversation scope". Arbitrary attributes placed in this map will be retained for the life of the flow execution and correspond to the length of the logical conversation. This map is <i>shared</i> by all flow sessions.	1	

As a flow execution is manipulated by clients its contextual state changes. Consider how contextual state is effected when the following events occur:

Table 3.3. An ordered set of events and their effects on flow execution context

Flow Execution Event	Active?	Value of the <code>activeSession</code> property
created	false	Throws an <code>IllegalStateException</code>
started	true	A <code>FlowSession</code> whose definition is the top-level flow definition and whose state is the definition's start state.
state entered	true	A <code>FlowSession</code> whose definition is the top-level

Flow Execution Event	Active?	Value of the <code>activeSession</code> property
		flow definition and whose <code>state</code> is the newly entered state.
subflow spawned	true	A <code>FlowSession</code> whose definition is the subflow definition and whose <code>state</code> is the subflow's start state.
subflow ended	true	A <code>FlowSession</code> whose definition is back to the top-level flow definition and whose <code>state</code> is the resuming state.
ended	false	Throws an <code>IllegalStateException</code>

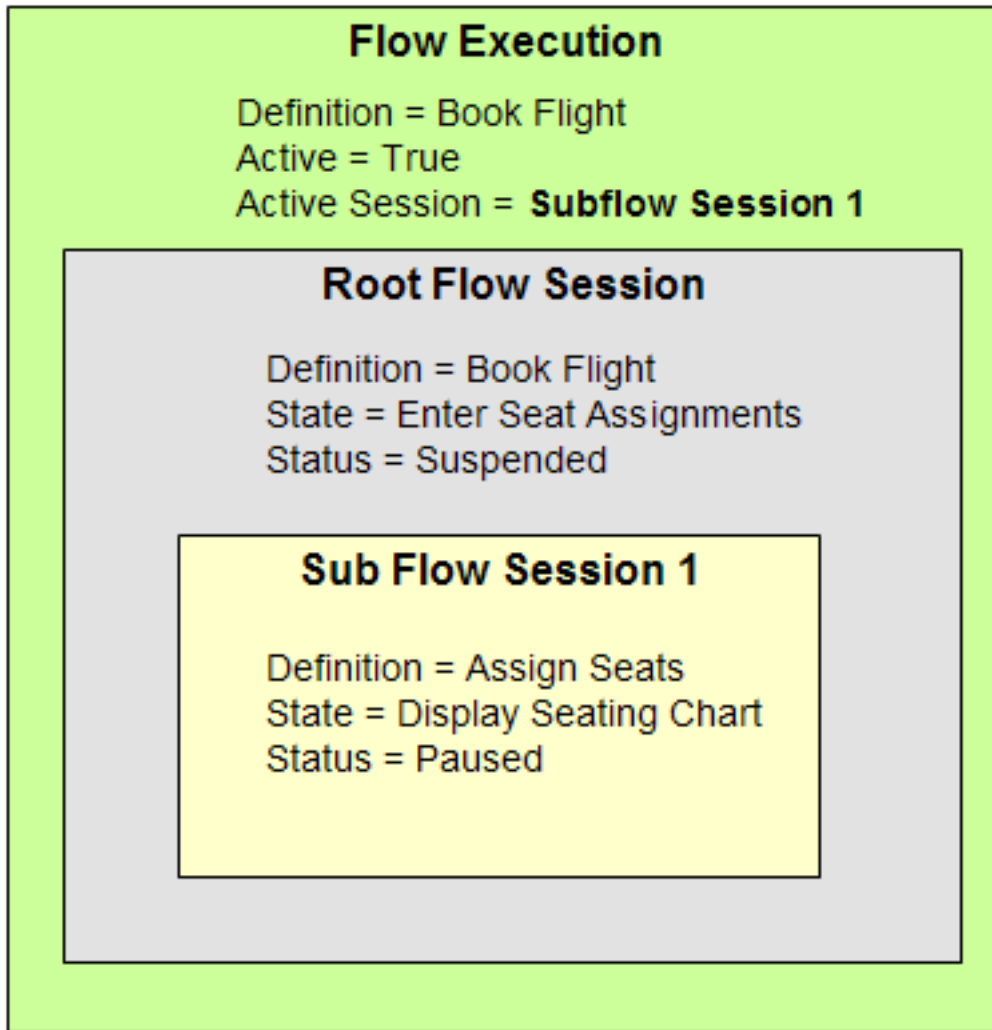
As you can see, the `activeSession` of a flow execution changes when a subflow is spawned. Each flow execution maintains a stack of flow sessions, where each flow session represents a spawned instance of a flow definition. When a flow execution starts, the session stack initially consists of one (1) entry, an instance dubbed the *root session*. When a subflow is spawned, the stack increases to two (2) entries. When the subflow ends, the stack decreases back to one (1) entry. The active session is always the session at the top of the stack.

The contextual properties associated with a `FlowSession` are summarized below:

Table 3.4. Flow Session properties

Property name	Description	Cardinality	Default value
definition	The definition of the flow the session is an instance of.	1	
state	The current state of the session.	1	
status	A status indicator describing what the session is currently doing.	1	
scope	A data map that forms the basis for <i>flow scope</i> . Arbitrary attributes placed in this map will be retained for the scope of the flow session. This map is <i>local</i> to the session.	1	
flashMap	A data map that forms the basis for <i>flash scope</i> . Attributes placed in this map will be retained until the next external user event is signaled in the session.	1	

The following graphic illustrates an example flow execution context and flow session stack:



Flow execution context

In this illustration a flow execution has been created for the `Book Flight` flow. The execution is currently active and the `activeSession` indicates it is in the `Display Seating Chart` state of the `Assign Seats` flow, which was spawned as a subflow from the `Enter Seat Assignments` state.



Note

Note how the active session status is `paused`, indicating the flow execution is currently waiting for user input to be provided to continue. In this case, it is expected the user will choose a seat for their flight.

3.4. Flow execution scopes

As alluded to, a flow execution manages several containers called *scopes*, which allow arbitrary attributes to be stored for a period of time. There are four scope types, each with different storage management semantics:

Table 3.5. Flow execution scope types

Scope type name	Management Semantics
request	Eligible for garbage collection when a single call into the flow execution completes.
flash	Cleared when the next user event is signaled into the flow session;

Scope type name	Management Semantics
	eligible for garbage collection when the flow session ends.
flow	Eligible for garbage collection when the flow session ends.
conversation	Eligible for garbage collection when the root session of the governing flow execution (logical conversation) ends.

3.5. Flow execution testing

Spring Web Flow provides support within the `org.springframework.webflow.test` package for testing flow executions with JUnit. This support is provided as convenience but is entirely optional, as a flow execution is instantiable in any environment with the standard `new` operator.

The general strategy for testing flows follows:

1. Your own implementations of definitional artifacts used by a flow such as actions, attribute mappers, and exception handlers should be unit tested in isolation. Spring Web Flow ships convenient stubs to assist with this, for instance `MockRequestContext`.
2. The execution of a flow should be tested as part of a system integration test. Such a test should exercise all possible paths of the flow, asserting that the flow responds to events as expected.

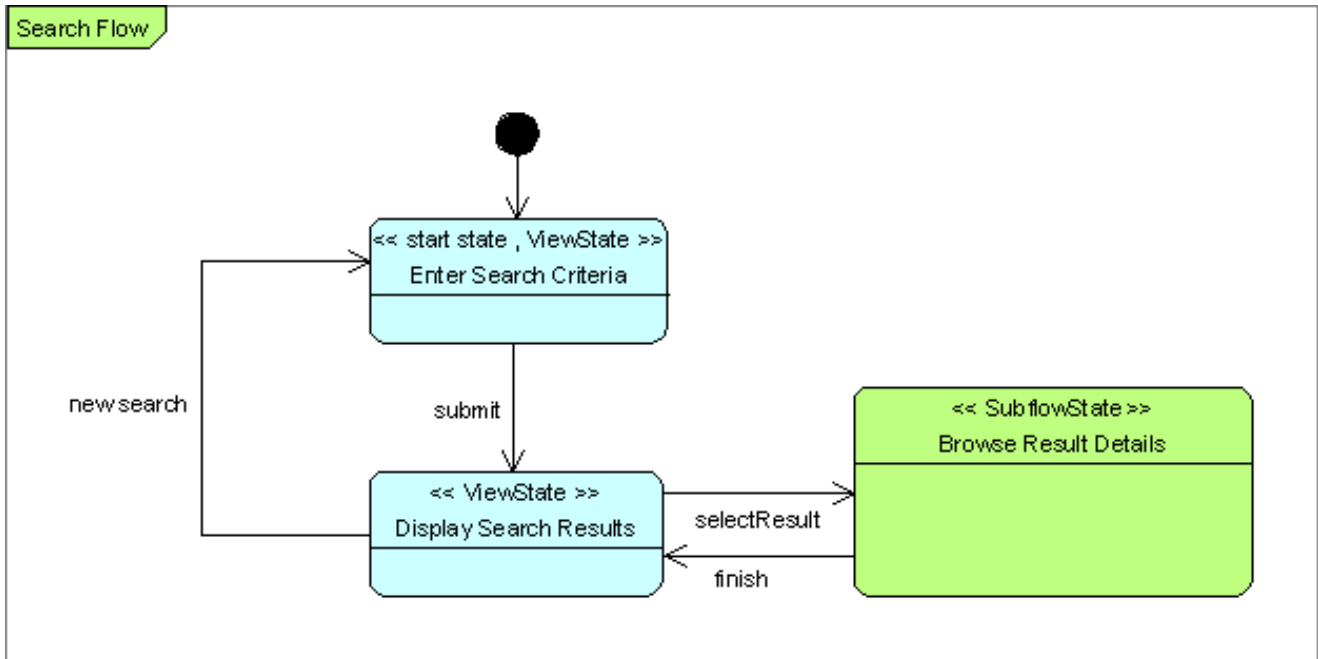


Note

A flow execution integration test typically selects mock or stub implementations of application services called by the flow, though it may also exercise production implementations. Both are useful, supported system test configurations.

3.5.1. Flow execution test example

To help illustrate testing a flow execution, first consider the following flow definition to search a phonebook for contacts:



Phonebook Search Flow - State Diagram

The corresponding XML-based flow definition implementation:

```

<?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-1.0.xsd">

  <start-state idref="enterCriteria"/>

  <view-state id="enterCriteria" view="searchCriteria">
    <render-actions>
      <action bean="formAction" method="setupForm"/>
    </render-actions>
    <transition on="search" to="displayResults">
      <action bean="formAction" method="bindAndValidate"/>
    </transition>
  </view-state>

  <view-state id="displayResults" view="searchResults">
    <render-actions>
      <bean-action bean="phonebook" method="search">
        <method-arguments>
          <argument expression="flowScope.searchCriteria"/>
        </method-arguments>
        <method-result name="results"/>
      </bean-action>
    </render-actions>
    <transition on="newSearch" to="enterCriteria"/>
    <transition on="select" to="browseDetails"/>
  </view-state>

  <subflow-state id="browseDetails" flow="detail-flow">
    <attribute-mapper>
      <input-mapper>
        <mapping source="requestParameters.id" target="id" from="string" to="long"/>
      </input-mapper>
    </attribute-mapper>
    <transition on="finish" to="displayResults"/>
  </subflow-state>

</flow>
  
```

Above you see a flow with three (3) states that execute these behaviors, respectively:

1. The first state `enterCriteria` displays a search criteria form so the user can enter who he or she wishes to search for.
2. On form submit and successful data binding and validation the search is executed. After search execution a results view is displayed.
3. From the results view the user may select a result they wish to browse additional details on or they may request a new search. On select, the "detail" flow is spawned and when it finishes the search is re-executed and it's results redisplayed.

From this behavior narrative the following assertable test scenarios can be extracted:

1. That when a flow execution starts, it enters the `enterCriteria` state and makes a `searchCriteria` view selection containing a *form object* to be used as the basis for form field population.
2. That on submit with valid input, the search is executed and a `searchResults` view selection is made.
3. That on submit with invalid input, the `searchCriteria` view is reselected.
4. That on `newSearch`, the `searchCriteria` view is selected.
5. That on select, the `detail` flow is spawned and passed the `id` of the selected result as expected.

To assist with writing these assertions Spring Web Flow ships with JUnit-based flow execution test support within the `org.springframework.webflow.test` package. These base test classes are indicated below:

Table 3.6. Flow execution test support hierarchy

Class name	Description
<code>AbstractFlowExecutionTests</code>	The most generic base class for flow execution tests.
<code>AbstractExternalizedFlowExecutionTests</code>	The base class for flow execution tests whose flow is defined within an externalized resource, such as a file.
<code>AbstractXmlFlowExecutionTests</code>	The base class for flow execution tests whose flow is defined within an externalized XML resource.

The completed test for this example extending `AbstractXmlFlowExecutionTests` is shown below:

```
public class SearchFlowExecutionTests extends AbstractXmlFlowExecutionTests {

    public void testStartFlow() {
        ApplicationView view = applicationView(startFlow());
        assertCurrentStateEquals("enterCriteria");
        assertViewNameEquals("searchCriteria", view);
        assertModelAttributeNotNull("searchCriteria", view);
    }

    public void testCriteriaSubmitSuccess() {
        startFlow();
        MockParameterMap parameters = new MockParameterMap();
        parameters.put("firstName", "Keith");
        parameters.put("lastName", "Donald");
        ApplicationView view = applicationView(signalEvent("search", parameters));
    }
}
```

```

    assertCurrentStateEquals("displayResults");
    assertViewNameEquals("searchResults", view);
    assertModelAttributeCollectionSize(1, "results", view);
}

public void testCriteriaSubmitError() {
    startFlow();
    signalEvent("search");
    assertCurrentStateEquals("enterCriteria");
}

public void testNewSearch() {
    testCriteriaSubmitSuccess();
    ApplicationView view = applicationView(signalEvent("newSearch"));
    assertCurrentStateEquals("enterCriteria");
    assertViewNameEquals("searchCriteria", view);
}

public void testSelectValidResult() {
    testCriteriaSubmitSuccess();
    MockParameterMap parameters = new MockParameterMap();
    parameters.put("id", "1");
    ApplicationView view = applicationView(signalEvent("select", parameters));
    assertCurrentStateEquals("displayResults");
    assertViewNameEquals("searchResults", view);
    assertModelAttributeCollectionSize(1, "results", view);
}

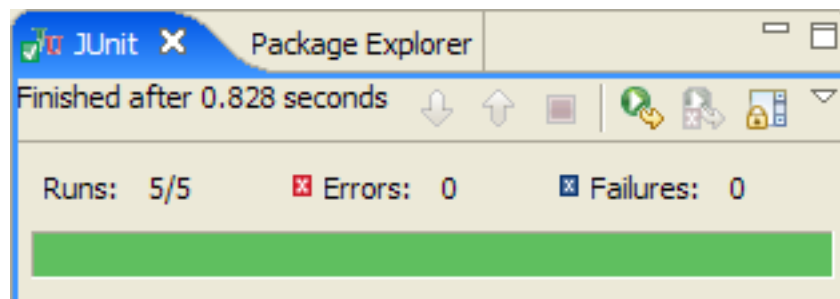
@Override
protected FlowDefinitionResource getFlowDefinitionResource() {
    return createFlowDefinitionResource("src/main/webapp/WEB-INF/flows/search-flow.xml");
}

@Override
protected void registerMockServices(MockFlowServiceLocator serviceRegistry) {
    Flow mockDetailFlow = new Flow("detail-flow");
    mockDetailFlow.setInputMapper(new AttributeMapper() {
        public void map(Object source, Object target, Map context) {
            assertEquals("id of value 1 not provided as input by calling search flow", new Long(1), ((Attribute) source).getValue());
        }
    });
    // test responding to finish result
    new EndState(mockDetailFlow, "finish");

    serviceRegistry.registerSubflow(mockDetailFlow);
    serviceRegistry.registerBean("phonebook", new ArrayListPhoneBook());
}
}

```

With a well-written flow execution test passing that covers the controller behavior scenarios possible for your flow you have concrete evidence the flow will execute as expected when deployed in a container.



Go for Green

3.5.2. Execution unit testing vs. full-blown system testing

The previous example shows how to test a flow execution in relative isolation with a mock service layer and mock subflows. Flow execution testing against a real service-layer and real subflows is also supported.

The next example shows how the `createFlowServiceLocator` method can be overridden to create the service-layer using a Spring application context:

```
public class SearchFlowExecutionTests extends AbstractXmlFlowExecutionTests {

    ...

    @Override
    protected FlowDefinitionResource getFlowDefinitionResource() {
        return createFlowDefinitionResource("src/main/webapp/WEB-INF/flows/search-flow.xml");
    }

    @Override
    protected FlowServiceLocator createFlowServiceLocator() {

        // create a context to host our middle tier services
        ApplicationContext context =
            new ClassPathXmlApplicationContext(new String[] {
                "classpath:service-layer-config.xml",
                "classpath:data-access-layer-config.xml"
            });

        // create a registry for our flow definitions being tested
        FlowDefinitionRegistry registry = new FlowDefinitionRegistryImpl();

        // initialize the service locator
        DefaultFlowServiceLocator locator = new DefaultFlowServiceLocator(registry, context);

        // perform subflow definition registration with the help of a registrar
        XmlFlowRegistrar registrar = new XmlFlowRegistrar(locator);
        registrar.addResource(createFlowDefinitionResource("/WEB-INF/flows/search-flow.xml"));
        registrar.addResource(createFlowDefinitionResource("/WEB-INF/flows/detail-flow.xml"));
        registrar.registerFlowDefinitions(registry);

        return locator;
    }
}
```

Chapter 4. Flow execution repositories

4.1. Introduction

A flow execution represents an executing flow *at a point in time*. At runtime there can be any number of flow executions active in parallel. A single user can even have multiple executions active at the same time (for example, when a user is operating multiple windows or tabs within their browser).

Many of these flow executions span multiple requests into the server and therefore must be saved so they can be resumed on subsequent requests. This presents technical challenges, as there must exist a stable mechanism for a new request to be associated with an existing execution in the view state that matches what the user expects. This problem is more difficult when you consider that many applications require use of browser navigational buttons and their use involves updating local history without notifying the server.

The problem of flow execution persistence is addressed by Spring Web Flow's *flow execution repository subsystem*. In this chapter you will learn how to use the system to manage the storage of active web conversations in a stable manner.

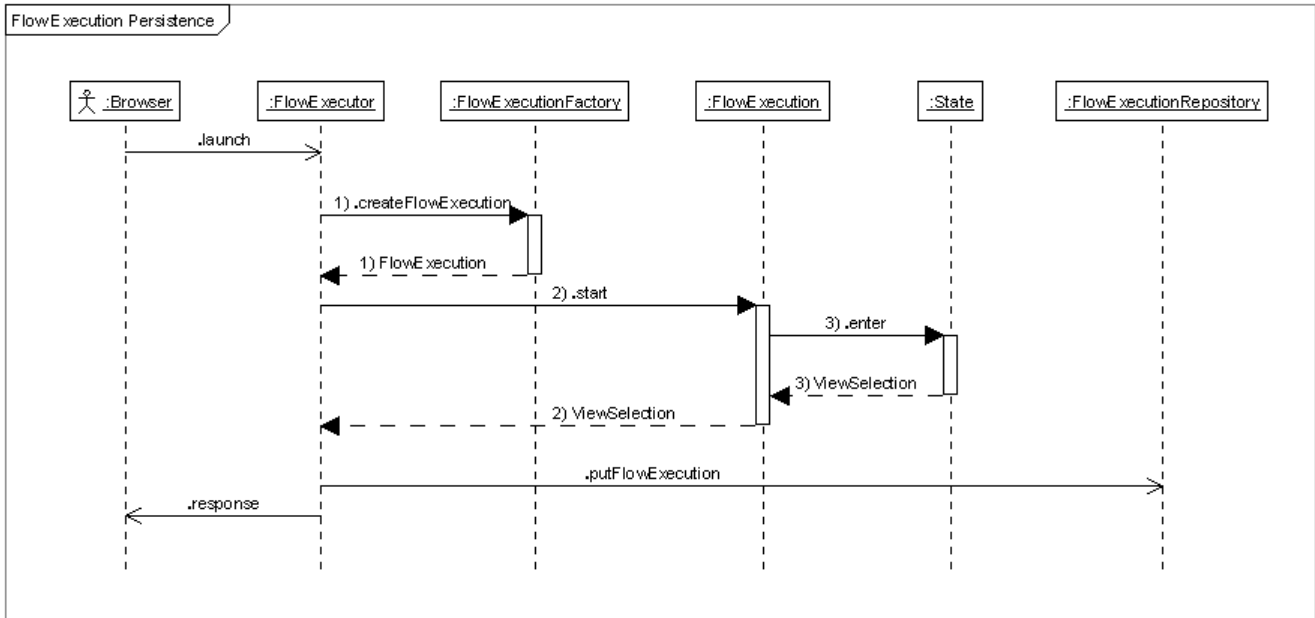
4.2. Repository architecture overview

Recall the following bullet points noting what happens when a flow execution enters a ViewState:

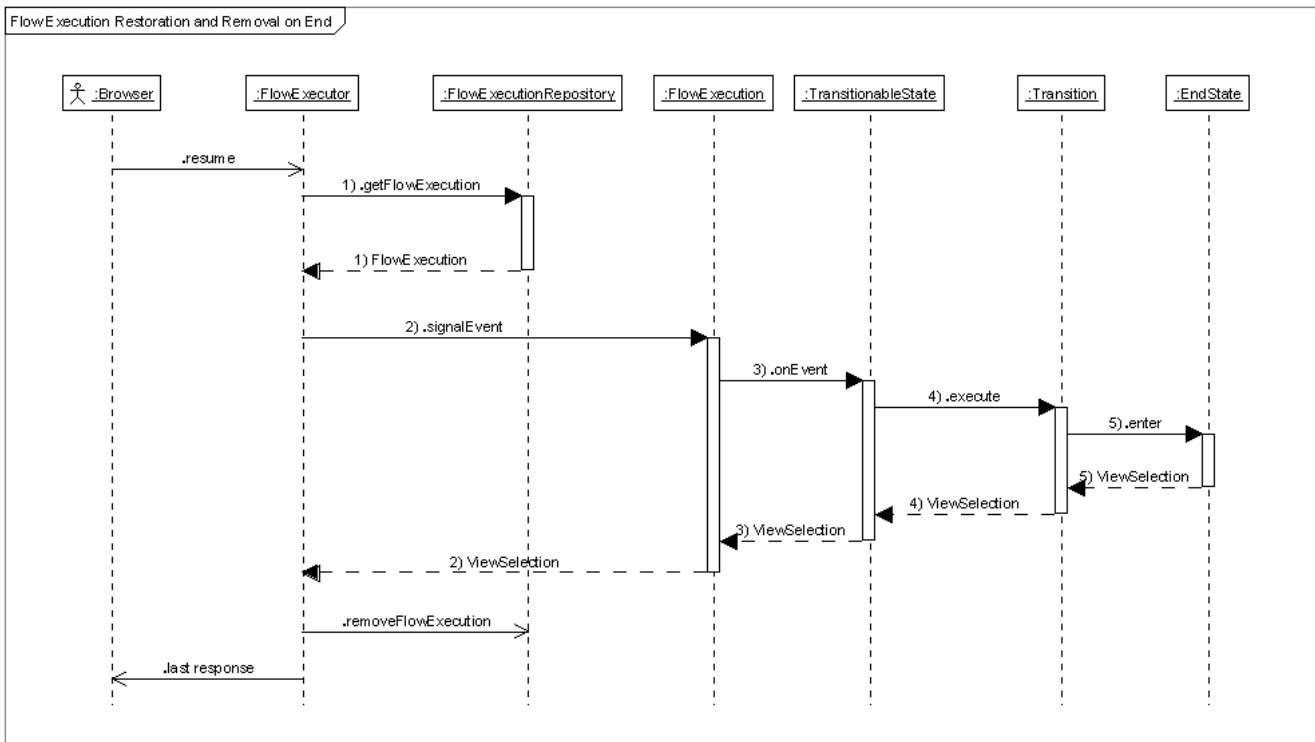
1. When a flow execution reaches a `viewState` it is said to have *paused*, where it waits in that state for user input to be provided so it can continue. After pausing the `viewSelection` returned is used to issue a response to the user that provides a vehicle for collecting the required input.
2. User input is provided by *signaling an event* that *resumes* the flow execution in the paused view state. The input event communicates what user action was taken.

Each time an active flow execution is *paused* it is saved out to a repository. When the next request comes in for that flow execution, it is restored from the repository, *resumed*, and continued. This process continues until the flow execution reaches an end state, at which time it is removed from the repository.

This process is demonstrated over the next two graphics:



Request one (1) - Paused flow execution persistence



Request two (2) - Paused flow execution restoration, removal on end

4.3. Flow execution identity

When a flow execution is created it marks the start of a new conversation between a browser and the server. As outlined, a new flow execution that is still active after startup processing indicates the start of a conversation that will span more than one request and needs to be persisted. When this is the case, that flow execution is assigned an *persistent identifier* by the repository. By default the structure of this identifier consists of a two-part composite key. This key is used by clients to restore the flow execution on subsequent requests.

4.3.1. Conversation identifier

The first part of a flow execution's persistent identity is a unique *conversation identifier*. This serves as an index into the *logical* conversation between the browser and the server that has just started.

4.3.2. Continuation identifier

The second part of a flow execution's persistent identity is a *continuation identifier*. This identifier serves as an index into a flow execution representing the state of the conversation *at this point in time*.

4.3.3. Flow execution key

Together the conversation id plus the continuation id make up the unique two-part *flow execution key* that identifies a state of a conversation *at a point in time*. By submitting this key in a subsequent request a browser can restore the conversation at that point and *continue* from there.

So on a subsequent request the conversation is resumed by restoring a flow execution from the repository using the two-part key. After event processing if the flow execution is still active it is saved back out to the repository. At this time a new flow execution key is generated. By default that key retains the same *conversation identifier*, as the same logical conversation is in progress; however the *continuation identifier* changes to provide an index into the state of the flow execution *at this new point in time*.

By submitting this new key in a subsequent request a browser can restore the conversation at that point and *continue* from there. This process continues until a flow execution reaches an end state during event processing signaling the end of the conversation.

4.4. Conversation ending

When a flow execution reaches an end state it terminates. If the flow execution was associated with a logical conversation that spanned more than one request, it is removed from the repository. More specifically, the entire conversation is *ended*, resulting in any flow execution continuations associated with the conversation being purged.

Once a conversation has been ended the conversation identifier is no longer valid and cannot ever be used again.

4.5. Flow execution repository implementations

The next section looks at the repository implementations that are available for use with Spring Web Flow out-of-the-box.

4.5.1. Simple flow execution repository

The simplest possible repository (`SimpleFlowExecutionRepository`). This repository stores *exactly one* flow execution instance per conversation in the user's session, invalidating it when its end state is reached. This repository implementation has been designed with minimal storage overhead in mind.



Note

It is important to understand that use of this repository consistently prevents duplicate submission when using the back button. If you attempt to go back and resubmit, the continuation id stored in

your browser history will not match the current continuation id needed to access the flow execution and access will be disallowed.



Note

This repository implementation should generally be used when you do not have to support browser navigational button use; for example, when you lock down the browser and require that all navigation events to be routed through Spring Web Flow.

4.5.2. Continuation flow execution repository

This repository (`ContinuationFlowExecutionRepository`) stores *one to many* flow execution instances per conversation in the user's session, where each flow execution represents a paused and restorable state of the conversation at a point in time. This repository implementation is considerably more flexible than the simple one, but incurs more storage overhead.



Note

It is important to understand that use of this repository allows resubmission when using the back button. If you attempt to go back and resubmit while the conversation is active, the continuation id stored in your browser history will match the continuation id of a previous flow execution in the repository. Access to that flow execution representing the state of the conversation at that point in time will be granted.

Like the simple implementation, this repository implementation provides support for *conversation invalidation after completion* where once a logical conversation completes (by one of its `FlowExecutions` reaching an end state), the entire conversation is invalidated. This prevents the possibility of resubmission after completion.

This repository is more elaborate than the default repository, offering more power (by enabling multiple continuations to exist per conversation), but incurring more storage overhead. This repository implementation should be considered when you do have to support browser navigational button use. This implementation is the default.

4.5.3. Client continuation flow execution repository

This repository is entirely stateless and its use entails no server-side state (`ClientContinuationFlowExecutionRepository`).

This is achieved by encoding a serialized flow execution directly into the flow execution continuation key that is sent in the response.

When asked to load a flow execution by its key on a subsequent request, this repository decodes and deserializes the flow execution, restoring it to the state it was in when it was serialized.



Note

This repository implementation does not currently support *conversation invalidation after completion*, as this capability requires tracking active conversations using some form of centralized storage, like a database table.



Note

Storing state (a flow execution continuation) on the client entails a certain security risk that should

be evaluated. Furthermore, it puts practical constraints on the size of the flow execution.

Chapter 5. Flow executors

5.1. Introduction

Flow executors are the highest-level entry points into the Spring Web Flow system, responsible for driving the execution of flows across a variety of environments.

In this chapter you'll learn how to execute flows within Spring MVC, Struts, and Java Server Faces (JSF) based applications.

5.2. FlowExecutor

`org.springframework.webflow.executor.FlowExecutor` is the central facade interface external systems use to drive the execution of flows. This facade acts as a simple, convenient service entry-point into the Spring Web Flow system that is reusable across environments.

The `FlowExecutor` interface is shown below:

```
public interface FlowExecutor {
    ResponseInstruction launch(String flowDefinitionId, ExternalContext context);
    ResponseInstruction resume(String flowExecutionKey, String eventId, ExternalContext context);
    ResponseInstruction refresh(String flowExecutionKey, ExternalContext context);
}
```

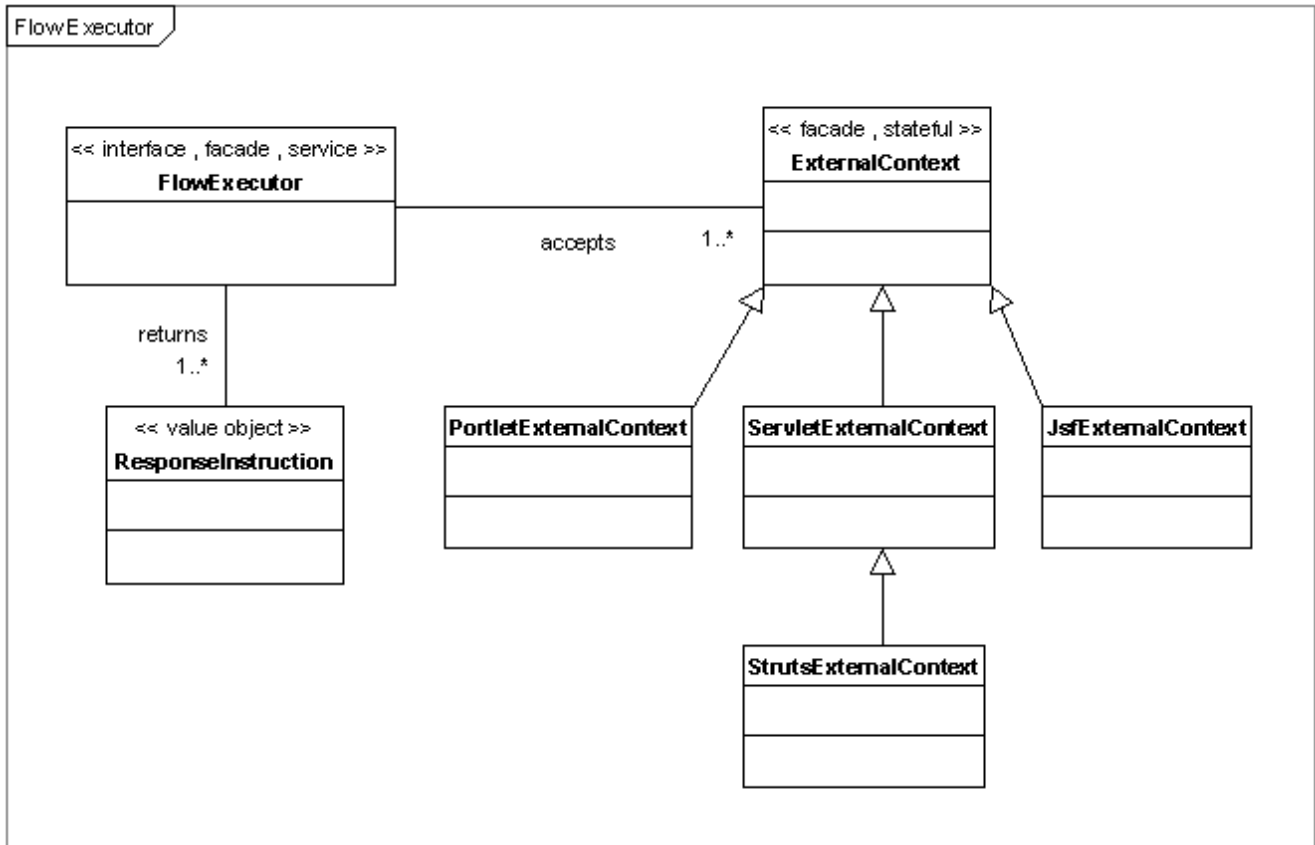
As you can see there are three central use-cases fulfilled by this interface:

1. Launch (start) a new execution of a flow definition.
2. Resume a paused flow execution by signaling an event against its current state.
3. Request that the last response issued by a flow execution be re-issued. Unlike start and signalEvent, the refresh operation is an idempotent operation that does not affect the state of a flow execution.

Each operation accepts an `ExternalContext` that provides normalized access to properties of an external system that has called into Spring Web Flow, allowing access to environment-specific request parameters as well as request, session, and application-level attributes.

Each operation returns a `ResponseInstruction` which the calling system is expected to use to issue a suitable response.

These relationships are shown graphically below:



Flow executor

As you can see, an `ExternalContext` implementation exists for each of the environments Spring Web Flow supports. If a flow artifact such as an Action needs to access native constructs of the calling environment it can downcast a context to its specific implementation. The need for such downcasting is considered a special case.

5.2.1. FlowExecutorImpl

The default executor implementation is `org.springframework.webflow.executor.FlowExecutorImpl`. It allows for configuration of a `FlowDefinitionLocator` responsible for loading the flow definitions to execute, as well as the `FlowExecutionRepository` strategy responsible for persisting flow executions that remain active beyond a single request into the server.

The configurable `FlowExecutorImpl` properties are shown below:

Table 5.1. FlowExecutorImpl properties

Property name	Description	Cardinality
<code>definitionLocator</code>	The service for loading flow definitions to be executed, typically a <code>FlowDefinitionRegistry</code>	<i>1</i>
<code>executionFactory</code>	The factory for creating new flow executions.	<i>1</i>
<code>executionRepository</code>	The repository for saving and loading persistent (paused) flow executions	<i>1</i>

5.2.2. A typical flow executor configuration with Spring 2.0

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:flow="http://www.springframework.org/schema/webflow-config"
       xsi:schemaLocation="
         http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
         http://www.springframework.org/schema/webflow-config
         http://www.springframework.org/schema/webflow-config/spring-webflow-config-1.0.xsd">

  <!-- Launches new flow executions and resumes existing executions. -->
  <flow:executor id="flowExecutor" registry-ref="flowRegistry"/>

  <!-- Creates the registry of flow definitions for this application -->
  <flow:registry id="flowRegistry">
    <flow:location path="/WEB-INF/flows/**/*-flow.xml"/>
  </flow:registry>

</beans>
```

This instructs Spring to create a flow executor that can execute all XML-based flow definitions contained within the `/WEB-INF/flows` directory. The default flow execution repository, `continuation`, is used.

5.2.3. A flow executor using a simple execution repository

```
<flow:executor id="flowExecutor" registry-ref="flowRegistry" repository-type="simple"/>
```

This executor is configured with a simple repository that manages execution state in the user session.

5.2.4. A flow executor using a client-side continuation-based execution repository

```
<flow:executor id="flowExecutor" registry-ref="flowRegistry" repository-type="client"/>
```

This executor is configured with a continuation-based repository that serializes continuation state to the client using no server-side state.

5.2.5. A flow executor using a single key execution repository

```
<flow:executor id="flowExecutor" registry-ref="flowRegistry" repository-type="singleKey"/>
```

This executor is configured with a simple repository that assigns a single flow execution key per conversation. The key, once assigned, never changes for the duration of the conversation.

5.2.6. A flow executor setting system execution attributes

```
<flow:executor id="flowExecutor" registry-ref="flowRegistry" repository-type="continuation">
  <flow:execution-attributes>
    <flow:alwaysRedirectOnPause value="false"/>
    <flow:attribute name="foo" value="bar"/>
  </flow:execution-attributes>
</flow:executor>
```

```
</flow:execution-attributes>
</flow-executor>
```

This executor is configured to set two flow execution system attributes `alwaysRedirectOnPause=false` and `foo=bar`.



Note

The `alwaysRedirectOnPause` attribute determines if a flow execution redirect occurs automatically each time an execution pauses (automated `POST+REDIRECT+GET` behavior). Setting this attribute to `false` will disable the *default 'true' behavior* where a flow execution redirect always occurs on pause.

5.2.7. A flow executor setting custom execution listeners

```
<flow:executor id="flowExecutor" registry-ref="flowRegistry" repository-type="continuation">
  <flow:execution-listeners>
    <flow:listener ref="listener" criteria="order-flow"/>
  </flow:execution-listeners>
</flow-executor>

<!-- A FlowExecutionListener to observe the lifecycle of order-flow executions -->
<bean id="listener" class="org.springframework.webflow.samples.sellitem.SellItemFlowExecutionListener"/>
```

This executor is configured to apply the execution listener to the "order-flow".

5.2.8. A Spring 1.2 compatible flow executor configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
  "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>

  <!-- Launches new flow executions and resumes existing executions: Spring 1.2 config version -->
  <bean id="flowExecutor" class="org.springframework.webflow.config.FlowExecutorFactoryBean">
    <property name="definitionLocator" ref="flowRegistry"/>
    <property name="executionAttributes">
      <map>
        <entry key="alwaysRedirectOnPause">
          <value type="java.lang.Boolean">false</value>
        </entry>
      </map>
    </property>
    <property name="repositoryType" value="CONTINUATION"/>
  </bean>

  <!-- Creates the registry of flow definitions for this application: Spring 1.2 config version -->
  <bean id="flowRegistry"
    class="org.springframework.webflow.engine.builder.xml.XmlFlowRegistryFactoryBean">
    <property name="flowLocations">
      <list>
        <value>/WEB-INF/flows/**/*-flow.xml</value>
      </list>
    </property>
  </bean>

</beans>
```

This achieves similar semantics as the Spring 2.0 version above. The 2.0 version is more concise, provides

stronger validation, and encapsulates internal details such as `FactoryBean` class names. The 1.2 version is Spring 1.2 or > compatible and digestable by Spring IDE 1.3.

5.3. Spring MVC integration

Spring Web Flow integrates with both Servlet and Portlet MVC which ship with the core Spring Framework. Use of Portlet MVC requires Spring 2.0.

For both Servlet and Portlet MVC a `FlowController` acts as an adapter between Spring MVC and Spring Web Flow. As an adapter, this controller has knowledge of both systems and delegates to a flow executor for driving the execution of flows. One controller typically executes all flows of an application, relying on parameterization to determine what flow to launch or what flow execution to resume.

5.3.1. A single flow controller executing all flows in a Servlet MVC environment

```
<bean name="/flowController.htm" class="org.springframework.webflow.executor.mvc.FlowController">
  <property name="flowExecutor" ref="flowExecutor"/>
</bean>
```

This controller, exported at the context-relative `/flowController.htm` URL, delegates to the configured flow executor for driving flow executions in a Spring Servlet MVC environment.

5.3.2. A single portlet flow controller executing a flow within a Portlet

```
<bean id="portletModeControllerMapping"
  class="org.springframework.web.portlet.handler.PortletModeHandlerMapping">
  <property name="portletModeMap">
    <map>
      <entry key="view" value-ref="flowController"/>
    </map>
  </property>
</bean>

<bean id="flowController" class="org.springframework.webflow.executor.mvc.PortletFlowController">
  <property name="flowExecutor" ref="flowExecutor"/>
  <property name="defaultFlowId" ref="search-flow"/>
</bean>
```

This controller, exported for access with the configured portlet mode, delegates to the configured flow executor for driving flow executions in a Spring Portlet MVC environment (by default, an execution of the `search-flow` will be launched).

5.4. Flow executor parameterization

Spring Web Flow allows for full control over how flow executor method arguments such as the `flowDefinitionId`, `flowExecutionKey`, and `eventId` are extracted from an incoming controller request with the `org.springframework.webflow.executor.support.FlowExecutorArgumentExtractor` strategy.



Note

The various flow controllers typically do not use this strategy directly but instead use a convenient `FlowExecutorArgumentHandler` implementation that takes care of argument extraction as well as exposing responsibilities (in callback URLs).

The next several examples illustrate strategies for parameterizing flow controllers from the browser to launch and resume flow executions:

5.4.1. Request parameter-based flow executor argument extraction

The default executor argument extractor strategy is request-parameter based. The default request parameters are:

Table 5.2. Extractor request parameter names

Parameter name	Description
<code>_flowId</code>	The flow definition id, needed to launch a new flow execution.
<code>_flowExecutionKey</code>	The flow execution key, needed to resume and refresh an existing flow execution.
<code>_eventId</code>	The id of an event that occurred, needed to resume an existing flow execution.

5.4.1.1. Launching a flow execution - parameter-style anchor

```
<a href="flowController.htm?_flowId=myflow">Launch My Flow</a>
```

5.4.1.2. Launching a flow execution - form

```
<form action="flowController.htm" method="post">
  <input type="submit" value="Launch My Flow"/>
  <input type="hidden" name="_flowId" value="myflow">
</form>
```

5.4.1.3. Resuming a flow execution - anchor

```
<a href="flowController.htm?_flowExecutionKey=${flowExecutionKey}&_eventId=submit">
  Submit
</a>
```

5.4.1.4. Resuming a flow execution - form

```
<form action="flowController.htm" method="post">
  ...
  <input type="hidden" name="_flowExecutionKey" value="${flowExecutionKey}">
  <input type="hidden" name="_eventId" value="submit"/>
  <input type="submit" class="button" value="Submit">
```

```
</form>
```

5.4.1.5. Resuming a flow execution - multiple form buttons

```
<form action="flowController.htm" method="post">
  ...
  <input type="hidden" name="_flowExecutionKey" value="{flowExecutionKey}">
  <input type="submit" class="button" name="_eventId_submit" value="Submit">
  <input type="submit" class="button" name="_eventId_cancel" value="Cancel">
</form>
```



Note

In this case the `eventId` is determined by parsing the name of the button that was pressed.

5.4.1.6. Refreshing a flow execution

```
<a href="flowController.htm?_flowExecutionKey={flowExecutionKey}">Refresh</a>
```

5.4.2. Request path based flow executor argument extraction

The request-path based argument extractor strategy relies on executor arguments being path elements as much as possible. This results in friendlier REST-style URLs such as `http://host/app/myflow` instead of `http://host/app?_flowId=myflow`.

5.4.2.1. A flow controller with a request-path based argument extractor

```
<bean name="/flowController.htm" class="org.springframework.webflow.executor.mvc.FlowController">
  <property name="flowExecutor" ref="flowExecutor"/>
  <property name="argumentHandler">
    <bean class="org.springframework.webflow.executor.support.RequestPathFlowExecutorArgumentHandler"/>
  </property>
</bean>
```

5.4.2.2. Launching a flow execution - REST-style anchor

```
<a href="flowController/myflow"/>Launch My Flow</a>
```

5.4.2.3. Resuming a flow execution - multiple form buttons

```
<form action="{flowExecutionKey}" method="post">
  ...
  <input type="submit" class="button" name="_eventId_submit" value="Submit">
  <input type="submit" class="button" name="_eventId_cancel" value="Cancel">
</form>
```

5.4.2.4. Refreshing a flow execution

```
<a href="flowController/k/${flowExecutionKey}">Refresh</a>
```

5.5. Struts integration

Spring Web Flow integrates with Struts 1.x or >. The integration is very similar to Spring MVC where a single front controller (FlowAction) drives the execution of all flows for the application by delegating to a configured flow executor.

5.5.1. A single flow action executing all flows

```
<form-beans>
  <form-bean name="actionForm" type="org.springframework.web.struts.SpringBindingActionForm"/>
</form-beans>

<action-mappings>
  <action path="/flowAction" name="actionForm" scope="request"
    type="org.springframework.webflow.executor.struts.FlowAction"/>
</action-mappings>
```

5.6. Java Server Faces (JSF) integration

Spring Web Flow integrates with JSF. The JSF integration relies on custom implementations of core JSF artifacts such as navigation handler and phase listener to drive the execution of flows.

5.6.1. A typical faces-config.xml file

```
<faces-config>
  <application>
    <navigation-handler>
      org.springframework.webflow.executor.jsf.FlowNavigationHandler
    </navigation-handler>
    <property-resolver>
      org.springframework.webflow.executor.jsf.FlowPropertyResolver
    </property-resolver>
    <variable-resolver>
      org.springframework.webflow.executor.jsf.FlowVariableResolver
    </variable-resolver>
    <variable-resolver>
      org.springframework.web.jsf.DelegatingVariableResolver
    </variable-resolver>
    <variable-resolver>
      org.springframework.web.jsf.WebApplicationContextVariableResolver
    </variable-resolver>
  </application>

  <lifecycle>
    <phase-listener>org.springframework.webflow.executor.jsf.FlowPhaseListener</phase-listener>
  </lifecycle>
</faces-config>
```

5.6.2. Launching a flow execution - command link

```
<h:commandLink value="Go" action="flowId:myflow"/>
```

5.6.3. Resuming a flow execution - form

```
<h:form id="form">
  ...
  <h:inputText id="propertyName" value="#{flowScope.managedBeanName.propertyName}"/>
  ...
  <input type="hidden" name="_flowExecutionKey" value="{flowExecutionKey}">
  <h:commandButton type="submit" value="Next" action="submit"/>
</h:form>
```

Chapter 6. Practical Use of Spring Web Flow

6.1. Sample applications

It is recommended that you review the Spring Web Flow sample applications included in the release distribution for best-practice illustrations of the features of this framework. A description of each sample is provided below:

1. Phonebook - the original sample demonstrating most features (including subflows).
2. Sellitem - demonstrates a wizard with conditional transitions, flow scope, flow execution redirects, and continuations.
3. Flowlauncher - demonstrates all the possible ways to launch and resume flows.
4. Itemlist - demonstrates REST-style URLs and inline flows.
5. Shippingrate - demonstrates Spring Web Flow together with Ajax technology.
6. NumberGuess - demonstrates use of stateful middle-tier components to carry out business logic.
7. Birthdate - demonstrates Struts integration and the MultiAction.
8. Fileupload - demonstrates multipart file upload.
9. Phonebook-Portlet - the phonebook sample in a Portlet environment (notice how the flow definitions do not change).
10. Sellitem-JSF - the sellitem sample in a JSF environment (notice how the flow definition is more concise because JSF takes care of data binding and validation).

6.2. Running the Web Flow sample applications

The samples can be built from the command line and imported as Eclipse projects - all samples come with Eclipse project settings. It is also possible to start by importing the samples into Eclipse first and then build with Ant within Eclipse.

6.2.1. Building from the Command Line

Java 1.5 (or greater) and Ant 1.6 (or greater) are prerequisites for building the sample applications. Ensure those are present in the system path or are passed on the command line. To build Web Flow samples from the command line, open a prompt, cd to the directory where Spring Web Flow was unzipped and run the following:

```
cd projects/spring-webflow/build-spring-webflow
ant dist
```

This builds all samples preparing "target" areas within each sample project subdirectory containing webapp structures in both exploded and WAR archive forms. The build also provides basic helper targets for deploying to Tomcat from Ant; however these webapp structures can be copied to any servlet container, and each project

is also a Eclipse Dynamic Web Project (DWP) for easy deployment inside Eclipse with the Eclipse Webtools Project (WTP).

6.2.2. Importing Projects into Eclipse

Importing the sample projects into Eclipse is easy. With a new or an existing workspace select: *File > Import > Existing Projects into Workspace*. In the resulting dialog browse to the project subdirectory where Spring Web Flow was unzipped and choose it as the root directory to import from. Select OK. Here Eclipse will list all projects it found including the sample application projects. Select the projects you're interested in, and select Finish.

If you previously built each project from the command line Eclipse will compile with no errors. If not you will need to run the Ant build *once* for these errors to clear and you can do that within Eclipse.

To build all projects inside Eclipse, import and expand the `build-spring-webflow` project, right-click on `build.xml` and select *Run As > Ant Build*. Doing this will run the default Ant target and will build all sample projects.

To build a single project inside Eclipse, simply select the project, right-click, and select *Run As > Ant Build*. You can also use the convenient shortcut ALT + SHIFT + X (Execute menu), then Q (Run Ant Build).

After Ant runs and the libraries needed to compile each project are downloaded, all errors in the Eclipse problems view should go away. Try refreshing a project (F5) if you still have errors. In general, from this point on you no longer need Ant: you can rely on Eclipse's incremental compile and Eclipse's web tools (WTP) built-in JEE support for deployment. (Ant is only needed in the system for command-line usage or when the list of jar dependencies for a project changes and new jars need to be downloaded).

6.2.3. Deploying projects inside Eclipse using Eclipse Web Tools (WTP)

Each Spring Web Flow sample application project is a Eclipse Dynamic Web Project (DWP), for easy deployment to a server running inside the Eclipse IDE. To take advantage of this, you must be running Eclipse 3.2 with Web Tools 1.5.

To run a sample application as a webapp inside Eclipse, simply select the project, right-click, and select *Run -> Run On Server*. A convenient shortcut for this action is ALT + SHIFT + X (Execute menu), R (Run on Server). The first time you do this you will be asked to setup a Server, where you are expected to point Eclipse to a location where you have a Servlet Container such as Apache Tomcat installed. Once your container has been setup and you finish the deployment wizard, Eclipse will start the container and automatically publish your webapp to it. In addition, it will launch a embedded web browser allowing you to run the webapp fully inside the IDE.

6.2.4. Other IDE's

Importing samples into other IDE's should be fairly straight-forward. If using another IDE running the Ant build from the command line first may help as it will populate the lib subdirectories of each sample project. Follow steps similar as those outlined for Eclipse above.

6.3. Fileupload Example

6.3.1. Overview

Fileupload is a simple one page web application for uploading files to a server. It is based on Spring MVC, uses a Web Flow controller and one web flow with two states: a view state for displaying the initial JSP page and an action state for processing the submit.

6.3.2. Web.xml

The web.xml configuration maps requests for "*.htm" to the fileupload servlet - a regular Spring MVC DispatcherServlet:

```
<servlet>
    <servlet-name>fileupload</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>fileupload</servlet-name>
    <url-pattern>*.htm</url-pattern>
</servlet-mapping>
```

6.3.3. Spring MVC Context

The Spring MVC servlet context for the fileupload servlet (WEB-INF/fileupload-servlet.xml) defines one controller bean:

```
<bean name="/admin.htm" class="org.springframework.webflow.executor.mvc.FlowController">
    <property name="flowExecutor" ref="flowExecutor" />
</bean>
```

FlowController is a Web Flow controller. It is the main point of integration between Spring MVC and Spring Web Flow routing requests to one or more managed web flow executions. The FlowController is injected with flowExecutor and flowRegistry beans containing one web flow definition:

```
<!-- Launches new flow executions and resumes existing executions. -->
<flow:executor id="flowExecutor" registry-ref="flowRegistry" repository-type="singlekey"/>

<!-- Creates the registry of flow definitions for this application -->
<flow:registry id="flowRegistry">
    <flow:location path="/WEB-INF/fileupload.xml" />
</flow:registry>
```

Given the above definitions the following URI can be used to invoke the "fileupload" flow:

```
/swf-fileupload/admin.htm?_flowId=fileupload
```

Both flowExecutor and flowRegistry beans are defined with Spring custom tags schema available in Spring 2.0. The custom tags make configuration less verbose and more readable. Regular Spring bean definitions can be used as well with earlier versions of Spring.

The Spring MVC context also defines a view resolver bean for resolving logical view names and a multipartResolver bean for the upload component. In general Web Flow does not aim to replace the flexibility of Spring MVC for view resolution. It focuses on the C in MVC.

6.3.4. Fileupload Web Flow

The start state for the fileupload flow (WEB-INF/fileupload.xml) is a view state:

```
<start-state idref="selectFile"/>

<view-state id="selectFile" view="fileForm">
    <transition on="submit" to="uploadFile"/>
</view-state>
```

View states allow a user to participate in a flow by presenting a suitable interface. The view attribute "fileForm" is a logical view name, which the Spring MVC view resolver bean will resolve to /WEB-INF/jsp/fileForm.jsp.

The fileForm.jsp has an html form that submits back to the same controller (/swf-fileupload/admin.htm) and passes a "_flowExecutionKey" parameter. The value for _flowExecutionKey is provided by the FlowController - it identifies the current instance of the flow and allows Web Flow to resume flow execution, which is paused each time a view is displayed.

The name of the form submit button "_eventId_submit" indicates the event id to use for deciding where to transition to next. Given an event with id of "submit" the "selectFile" view transitions to the "uploadFile" state:

```
<action-state id="uploadFile">
    <action bean="uploadAction" method="uploadFile"/>
    <transition on="success" to="selectFile">
        <set attribute="fileUploaded" scope="flash" value="true"/>
    </transition>
    <transition on="error" to="selectFile"/>
</action-state>
```

The "uploadFile" state is an action state. Action states integrate with business application code and respond to the execution of that code by deciding what state of the flow to enter next. The code for the uploadFile state is in the "uploadAction" bean declared in the Spring web context (/WEB-INF/fileupload-servlet.xml):

```
<bean id="uploadAction" class="org.springframework.webflow.samples.fileupload.FileUploadAction" />
```

FileUploadAction has simple logic. It picks one of two Web Flow defined events - success or error, depending on whether the uploaded file size is greater than 0 or not. Both success and error transition back to the "selectFile" view state. However, a success event causes an attribute named "fileUploaded" to be set in flash scope

A flash-scoped attribute called "file" is also set programmatically in the FileUploadAction bean:

```
context.getFlashScope().put("file", new String(file.getBytes()));
return success();
```

This illustrates the choice to save attributes in one of several scopes either programmatically or declaratively.

6.4. Birthdate Example

6.4.1. Overview

Birthdate is a web application with 3 consecutive screens. The first two collect user input to populate a form object. The third presents the results of business calculations based on input provided in the first two screens.

Birthdate demonstrates Spring Web Flow's Struts integration as well as the use of FormAction, a multi-action used to do the processing required for all three screens. The sample also uses JSTL taglibs in conjunction with flows.

6.4.2. Web.xml

The web.xml configuration maps requests for "*.do" to a regular Struts ActionServlet:

```
<servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

The web.xml also sets up the loading of a Spring context at web application startup:

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        /WEB-INF/webflow-config.xml
    </param-value>
</context-param>

<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

The Spring web context contains beans to set up the Web Flow runtime environment. As will be shown in the next section Struts is configured with a Web Flow action that relies on the presence of a flowExecutor and a flowRegistry beans in this context.

6.4.3. Struts Configuration

The Struts configuration (WEB-INF/struts-config.xml) defines the following action mapping:

```
<action-mappings>
    <action path="/flowAction" name="actionForm" scope="request"
        type="org.springframework.webflow.executor.struts.FlowAction"/>
</action-mappings>
```

FlowAction is a Struts action acting as a front controller to the Web Flow system routing Struts requests to one or more managed web flow executions. To fully configure the FlowAction a Spring web context is required to define flowExecutor and flowRegistry beans (named exactly so). This is an excerpt from the Spring web context (/WEB-INF/webflow-config.xml) defining these beans:

```
<!-- Launches new flow executions and resumes existing executions. -->
<flow:executor id="flowExecutor" registry-ref="flowRegistry"/>

<!-- Creates the registry of flow definitions for this application -->
<flow:registry id="flowRegistry">
```

```
<flow:location path="/WEB-INF/birthdate.xml"/>
  <flow:location path="/WEB-INF/birthdate-alternate.xml"/>
</flow:registry>
```

Based on the above, Web Flow is configured with two flows - birthdate and birthdate-alternate, which can be invoked as follows:

```
/swf-birthdate/flowAction.do?_flowId=birthdate
/swf-birthdate/flowAction.do?_flowId=birthdate-alternate
```

The Struts configuration file also defines several global forwards: birthdateForm, cardForm, and yourAge, which will be referenced from Web Flow definitions as logical view names (and left to Struts to resolve to actual JSP pages). In general Web Flow does not aim to replace view resolution capabilities of web frameworks such as Struts or Spring MVC. It focuses on the C in MVC.

6.4.4. Birthdate Web Flow

The birthdate web flow (WEB-INF/birthdate.xml) defines the following start state:

```
<view-state id="enterBirthdate" view="birthdateForm">
  <render-actions>
    <action bean="formAction" method="setupForm" />
  </render-actions>
  <transition on="submit" to="processBirthdateFormSubmit" />
</view-state>
```

The setupForm action is called to perform initializations for the enterBirthdate view state. Its action bean is defined in the Spring web context WEB-INF/webflow-config.xml:

```
<bean id="formAction" class="org.springframework.webflow.samples.birthdate.BirthDateFormAction" />
```

BirthDateFormAction is a FormAction - it extends Web Flow's FormAction class, which serves a purpose similar to that of Spring MVC's SimpleFormController providing common form functionality for data binding and validation.

When the BirthDateFormAction bean is instantiated it sets the name, class and scope of the form object to use for loading form data upon display and collecting form data upon submit:

```
public BirthDateFormAction() {
  // tell the superclass about the form object and validator we want to
  // use you could also do this in the application context XML ofcourse
  setFormObjectName("birthDate");
  setFormObjectClass(BirthDate.class);
  setFormObjectScope(ScopeType.FLOW);
  setValidator(new BirthDateValidator());
}
```

The form object "birthDate" is placed in flow scope, which means it will not be re-created with each request but will be obtained from flow scope instead as long as the request remains within the same flow.

Once setupForm is done, the "birthdateForm" view will be rendered. The logical view name "birthdateForm" is a global-forward in struts-config.xml resolving to /WEB-INF/jsp/birthdateForm.jsp. This JSP collects data for the fields "name" and "date" bound to the birthDate form object and posts back to FlowAction with a submit

image named "_eventId_submit". An event with the id of "submit" causes a transition to the processBirthdateFormSubmit action state defined as follows:

```
<action-state id="processBirthdateFormSubmit">
  <action bean="formAction" method="bindAndValidate">
    <attribute name="validatorMethod" value="validateBirthdateForm" />
  </action>
  <transition on="success" to="enterCardInformation" />
  <transition on="error" to="enterBirthdate" />
</action-state>
```

The processBirthDateFormSubmit action state uses the same formAction bean as the one already used to setup the form. This time its bindAndValidate method is used to populate and validate the html form values. Also, note the "validateMethod" attribute used to specify the name of the method to invoke on the Validator object setup in the constructor of the BirthDateFormAction. The use of this attribute allows partial validation of complex objects populated over several consecutive screens.

On error the action returns to the view state it came from. On success it transitions to the enterCardInformation view state:

```
<view-state id="enterCardInformation" view="cardForm">
  <transition on="submit" to="processCardFormSubmit" />
</view-state>
```

The logical view name "cardForm" is a global-forward in struts-config.xml resolving to /WEB-INF/jsp/cardForm.jsp. This JSP collects data for the remaining fields of the birthDate form object - "sendCard" and "emailAddress", and posts back to FlowAction with a submit image named "_eventId_submit". An event with the id of "submit" causes a transition to the processCardFormSubmit action state defined as follows:

```
<action-state id="processCardFormSubmit">
  <action bean="formAction" method="bindAndValidate">
    <attribute name="validatorMethod" value="validateCardForm" />
  </action>
  <transition on="success" to="calculateAge" />
  <transition on="error" to="enterCardInformation" />
</action-state>
```

For this action state the bindAndValidate method of the formAction bean is used to populate and validate the remaining html form values. The "validateMethod" attribute specifies the name of the method to invoke on the Validator object specific to the fields loaded on the current screen.

On error the action returns to the view state it came from. On success it transitions to another action state called calculateAge:

```
<action-state id="calculateAge">
  <action bean="formAction" method="calculateAge" />
  <transition on="success" to="displayAge" />
</action-state>
```

The logic for the calculateAge action state is in the calculateAge method of the same formAction bean used for data binding and validation. This demonstrates the flexibility Web Flow allows in properly structuring control and business logic according to function.

The caculateAge method performs business calculations and adds a string in request scope with the calculated

age. Upon successful completion the calculateAge action state transitions to the end view state:

```
<end-state id="displayAge" view="yourAge" />
```

Once again the logical view name "yourAge" is a global-forward in struts-config.xml resolving to /WEB-INF/jsp/yourAge.jsp. This JSP page retrieves the calculated age from request scope and displays the results for the user.

The transition to the end state indicates the end of the web flow. The flow execution is cleaned up. If the web flow is entered again a new flow execution will start, creating a new form object named "birthDate" and placing it in flow scope.

6.4.5. Birthdate-alternate Web Flow

The birthdate-alternate web flow (/WEB-INF/birthdate-alternate.xml) offers an alternative way and more compact way of defining the same web flow. For example the birthdate web flow defines two independent states for the first screen - a view state (enterBirthdate) and an action state (processBirthdateFormSubmit). In birthdate-alternate those are encapsulated in the view state enterBirthdate as follows:

```
<view-state id="enterBirthdate" view="birthdateForm">
  <render-actions>
    <action bean="formAction" method="setupForm" />
  </render-actions>
  <transition on="submit" to="enterCardInformation">
    <action bean="formAction" method="bindAndValidate">
      <attribute name="validatorMethod" value="validateBirthdateForm" />
    </action>
  </transition>
</view-state>
```

Here the setupForm action state is defined as a render-action of the enterBirthdate view state while the transition to the next screen uses a nested action bean invoked before the transition occurs. Notice that success is implicitly required for the transition to occur. Similarly on error the transition does not occur and the same view state is displayed again.

The second screen is also defined with a nested transition and action bean:

```
<view-state id="enterCardInformation" view="cardForm">
  <transition on="submit" to="calculateAge">
    <action bean="formAction" method="bindAndValidate">
      <attribute name="validatorMethod" value="validateCardForm" />
    </action>
  </transition>
</view-state>
```

The remaining two states - calculateAge and displayAge are identical.