Spring Web Flow Plugin - Reference Documentation

Authors: Graeme Rocher

Version: 2.1.0

Table of Contents

1 Introduction

Overview

Grails supports the creation of web flows built on the Spring Web Flow project. A web flow is a conversation that spans multiple requests and retains state for the scope of the flow. A web flow also has a defined start and end state.

Web flows don't require an HTTP session, but instead store their state in a serialized form, which is then restored using a flow execution key that Grails passes around as a request parameter. This makes flows far more scalable than other forms of stateful application that use the HttpSession and its inherit memory and clustering concerns.

Web flow is essentially an advanced state machine that manages the "flow" of execution from one state to the next. Since the state is managed for you, you don't have to be concerned with ensuring that users enter an action in the middle of some multi step flow, as web flow manages that for you. This makes web flow perfect for use cases such as shopping carts, hotel booking and any application that has multi page work flows.

From Grails 1.2 onwards Webflow is no longer in Grails core, so you must install the Webflow plugin to use this feature.

Creating a Flow

To create a flow create a regular Grails controller and add an action that ends with the convention `Flow`. For example:

```
class BookController {

def index() {
      redirect(action: "shoppingCart")
   }

def shoppingCartFlow = {
        …
   }
}
```

Notice when redirecting or referring to the flow as an action we omit the `Flow` suffix. In other words the name of the action of the above flow is `shoppingCart`.

2 Start and End States

As mentioned before a flow has a defined start and end state. A start state is the state which is entered when a user first initiates a conversation (or flow). The start state of a Grails flow is the first method call that takes a block. For example:

```
class BookController {
   …
   def shoppingCartFlow ={
       showCart {
           on("checkout").to "enterPersonalDetails"
           on("continueShopping").to "displayCatalogue"
       }
       …
       displayCatalogue {
           redirect(controller: "catalogue", action: "show")
       }
       displayInvoice()
   }
}
```

Here the `showCart` node is the start state of the flow. Since the showCart state doesn't define an action or redirect it is assumed be a [view state](#) that, by convention, refers to the view `grails-app/views/book/shoppingCart/showCart.gsp`.

Notice that unlike regular controller actions, the views are stored within a directory that matches the name of the flow: `grails-app/views/book/shoppingCart`.

The `shoppingCart` flow also has two possible end states. The first is `displayCatalogue` which performs an external redirect to another controller and action, thus exiting the flow. The second is `displayInvoice` which is an end state as it has no events at all and will simply render a view called `grails-app/views/book/shoppingCart/displayInvoice.gsp` whilst ending the flow at the same time.

Once a flow has ended it can only be resumed from the start state, in this case `showCart`, and not from any other state.

3 Action States and View States

View states

A view state is a one that doesn't define an `action` or a `redirect`. So for example this is a view state:

```
enterPersonalDetails {
   on("submit").to "enterShipping"
   on("return").to "showCart"
}
```

It will look for a view called `grails-app/views/book/shoppingCart/enterPersonalDetails.gsp` by default. Note that the `enterPersonalDetails` state defines two events: `submit` and `return`. The view is responsible for triggering these events. Use the `render` method to change the view to be rendered:

```
enterPersonalDetails {
   render(view: "enterDetailsView")
   on("submit").to "enterShipping"
   on("return").to "showCart"
}
```

Now it will look for `grails-app/views/book/shoppingCart/enterDetailsView.gsp`. Start the `view` parameter with a / to use a shared view:

```
enterPersonalDetails {
   render(view: "/shared/enterDetailsView")
   on("submit").to "enterShipping"
   on("return").to "showCart"
}
```

Now it will look for `grails-app/views/shared/enterDetailsView.gsp`

Action States

An action state is a state that executes code but does not render a view. The result of the action is used to dictate flow transition. To create an action state you define an action to to be executed. This is done by calling the `action` method and passing it a block of code to be executed:

```
listBooks {
   action {
      [bookList: Book.list()]
   }
   on("success").to "showCatalogue"
   on(Exception).to "handleError"
}
```

As you can see an action looks very similar to a controller action and in fact you can reuse controller actions if you want. If the action successfully returns with no errors the `success` event will be triggered. In this case since we return a Map, which is regarded as the "model" and is automatically placed in flow scope.

In addition, in the above example we also use an exception handler to deal with errors on the line:

```
on(Exception).to "handleError"
```

This makes the flow transition to a state called `handleError` in the case of an exception.

You can write more complex actions that interact with the flow request context:

```
processPurchaseOrder {
    action {
        def a =  flow.address
        def p = flow.person
        def pd = flow.paymentDetails
        def cartItems = flow.cartItems
        flow.clear()

def o = new Order(person: p, shippingAddress: a, paymentDetails: pd)
        o.invoiceNumber = new Random().nextInt(9999999)
        for (item in cartItems) { o.addToItems item }
        o.save()
        [order: o]
    }
    on("error").to "confirmPurchase"
    on(Exception).to "confirmPurchase"
    on("success").to "displayInvoice"
}
```

Here is a more complex action that gathers all the information accumulated from the flow scope and creates an `Order` object. It then returns the order as the model. The important thing to note here is the interaction with the request context and "flow scope".

Transition Actions

Another form of action is what is known as a *transition* action. A transition action is executed directly prior to state transition once an event has been triggered. A simple example of a transition action can be seen below:

```
enterPersonalDetails {
   on("submit") {
       log.trace "Going to enter shipping"
   }.to "enterShipping"
   on("return").to "showCart"
}
```

Notice how we pass a block of the code to `submit` event that simply logs the transition. Transition states are very useful for data binding and validation, which is covered in a later section.

4 Flow Execution Events

In order to *transition* execution of a flow from one state to the next you need some way of trigger an *event* that indicates what the flow should do next. Events can be triggered from either view states or action states.

Triggering Events from a View State

As discussed previously the start state of the flow in a previous code listing deals with two possible events. A `checkout` event and a `continueShopping` event:

```
def shoppingCartFlow = {
    showCart {
        on("checkout").to "enterPersonalDetails"
        on("continueShopping").to "displayCatalogue"
    }
    …
}
```

Since the `showCart` event is a view state it will render the view `grails-app/book/shoppingCart/showCart.gsp`. Within this view you need to have components that trigger flow execution. On a form this can be done use the tags tag:

```
<g:form>
    <g:submitButton name="continueShopping" value="Continue Shopping" />
    <g:submitButton name="checkout" value="Checkout" />
</g:form>
```

The form automatically submits back to the `shoppingCart` flow. The name attribute of each tags tag signals which event will be triggered. If you don't have a form you can also trigger an event with the tags tag as follows:

```
<g:link event="checkout" />
```

Prior to 2.0.0, it was required to specify the controller and/or action in forms and links, which caused the url to change when entering a subflow state. When the controller and action are not specified, all url's are relative to the main flow execution url, which makes your flows reusable as subflows and prevents issues with the browser's back button.

Triggering Events from an Action

To trigger an event from an `action` you invoke a method. For example there is the built in `error()` and `success()` methods. The example below triggers the `error()` event on validation failure in a transition action:

```
enterPersonalDetails {
   on("submit") {
         def p = new Person(params)
         flow.person = p
         if (!p.validate()) return error()
   }.to "enterShipping"
   on("return").to "showCart"
}
```

In this case because of the error the transition action will make the flow go back to the `enterPersonalDetails` state.

With an action state you can also trigger events to redirect flow:

```
shippingNeeded {
   action {
       if (params.shippingRequired) yes()
       else no()
   }
   on("yes").to "enterShipping"
   on("no").to "enterPayment"
}
```

5 Flow Scopes

Scope Basics

You'll notice from previous examples that we used a special object called `flow` to store objects within "flow scope". Grails flows have five different scopes you can utilize:

- `request` - Stores an object for the scope of the current request
- `flash` - Stores the object for the current and next request only
- `flow` - Stores objects for the scope of the flow, removing them when the flow reaches an end state
- `conversation` - Stores objects for the scope of the conversation including the root flow and nested subflows
- `session` - Stores objects in the user's session

Grails service classes can be automatically scoped to a web flow scope. See the documentation on guide:services for more information.

Returning a model Map from an action will automatically result in the model being placed in flow scope. For example, using a transition action, you can place objects within `flow` scope as follows:

```
enterPersonalDetails {
    on("submit") {
        [person: new Person(params)]
    }.to "enterShipping"
   on("return").to "showCart"
}
```

Be aware that a new request is always created for each state, so an object placed in request scope in an action state (for example) will not be available in a subsequent view state. Use one of the other scopes to pass objects from one state to another. Also note that Web Flow:

1. Moves objects from flash scope to request scope upon transition between states;
2. Merges objects from the flow and conversation scopes into the view model before rendering (so you shouldn't include a scope prefix when referencing these objects within a view, e.g. GSP pages).

Flow Scopes and Serialization

When placing objects in `flash`, `flow` or `conversation` scope they must implement `java.io.Serializable` or an exception will be thrown. This has an impact on guide:GORM in that domain classes are typically placed within a scope so that they can be rendered in a view. For example consider the following domain class:

```
class Book {
    String title
}
```

To place an instance of the `Book` class in a flow scope you will need to modify it as follows:

```
class Book implements Serializable {
    String title
}
```

This also impacts associations and closures you declare within a domain class. For example consider this:

```
class Book implements Serializable {
    String title
    Author author
}
```

Here if the `Author` association is not `Serializable` you will also get an error. This also impacts closures used in guide:eventsAutoTimestamping such as `onLoad`, `onSave` and so on. The following domain class will cause an error if an instance is placed in a flow scope:

```
class Book implements Serializable {

String title

def onLoad = {
        println "I'm loading"
    }
}
```

The reason is that the assigned block on the `onLoad` event cannot be serialized. To get around this you should declare all events as `transient`:

```
class Book implements Serializable {

String title

transient onLoad = {
        println "I'm loading"
    }
}
```

or as methods:

```
class Book implements Serializable {

String title

def onLoad() {
        println "I'm loading"
    }
}
```

The flow scope contains a reference to the Hibernate session. As a result, any object loaded into the session through a GORM query will also be in the flow and will need to implement Serializable.

If you don't want your domain class to be Serializable or stored in the flow, then you will need to evict the entity manually before the end of the state:

```
flow.persistenceContext.evict(it)
```

6 Data Binding and Validation

In the section on start and end states, the start state in the first example triggered a transition to the `enterPersonalDetails` state. This state renders a view and waits for the user to enter the required information:

```
enterPersonalDetails {
   on("submit").to "enterShipping"
   on("return").to "showCart"
}
```

The view contains a form with two submit buttons that either trigger the submit event or the return event:

```
<g:form>
    <!-- Other fields -->
    <g:submitButton name="submit" value="Continue"></g:submitButton>
    <g:submitButton name="return" value="Back"></g:submitButton>
</g:form>
```

However, what about the capturing the information submitted by the form? To capture the form info we can use a flow transition action:

```
enterPersonalDetails {
   on("submit") {
      flow.person = new Person(params)
      !flow.person.validate() ? error() : success()
   }.to "enterShipping"
   on("return").to "showCart"
}
```

Notice how we perform data binding from request parameters and place the `Person` instance within `flow` scope. Also interesting is that we perform guide:validation and invoke the `error()` method if validation fails. This signals to the flow that the transition should halt and return to the `enterPersonalDetails` view so valid entries can be entered by the user, otherwise the transition should continue and go to the `enterShipping` state.

Like regular actions, flow actions also support the notion of guide:commandObjects by defining the first argument of the closure:

```
enterPersonalDetails {
   on("submit") { PersonDetailsCommand cmd ->
       flow.personDetails = cmd
      !flow.personDetails.validate() ? error() : success()
   }.to "enterShipping"
   on("return").to "showCart"
}
```

7 Subflows and Conversations

Calling subflows

Grails' Web Flow integration also supports subflows. A subflow is like a flow within a flow. For example take this search flow:

```
def searchFlow = {
    displaySearchForm {
        on("submit").to "executeSearch"
    }
    executeSearch {
        action {
            [results:searchService.executeSearch(params.q)]
        }
        on("success").to "displayResults"
        on("error").to "displaySearchForm"
    }
    displayResults {
        on("searchDeeper").to "extendedSearch"
        on("searchAgain").to "displaySearchForm"
    }
    extendedSearch {
        // Extended search subflow
        subflow(controller: "searchExtensions", action: "extendedSearch")
        on("moreResults").to "displayMoreResults"
        on("noResults").to "displayNoMoreResults"
    }
    displayMoreResults()
    displayNoMoreResults()
}
```

It references a subflow in the `extendedSearch` state. The controller parameter is optional if the subflow is defined in the same controller as the calling flow.

> ⚠ Prior to 1.3.5, the previous subflow call would look like `subflow(new SearchExtensionsController().extendedSearchFlow)`, with the requirement that the name of the subflow state be the same as the called subflow (minus `Flow`). This way of calling a subflow is deprecated and only supported for backward compatibility.

The subflow is another flow entirely:

```
def extendedSearchFlow = {
    startExtendedSearch {
        on("findMore").to "searchMore"
        on("searchAgain").to "noResults"
    }
    searchMore {
        action {
           def results = searchService.deepSearch(ctx.conversation.query)
           if (!results) return error()
           conversation.extendedResults = results
        }
        on("success").to "moreResults"
        on("error").to "noResults"
    }
    moreResults()
    noResults()
}
```

Notice how it places the `extendedResults` in conversation scope. This scope differs to flow scope as it lets you share state that spans the whole conversation, i.e. a flow execution including all subflows, not just the flow itself. Also notice that the end state (either `moreResults` or `noResults` of the subflow triggers the events in the main flow:

```
extendedSearch {
    // Extended search subflow
    subflow(controller: "searchExtensions", action: "extendedSearch")
    on("moreResults").to "displayMoreResults"
    on("noResults").to "displayNoMoreResults"
}
```

Subflow input and output

Using conversation scope for passing input and output between flows can be compared with using global variables to pass information between methods. While this is OK in certain situations, it is usually better to use method arguments and return values. In webflow speak, this means defining input and output arguments for flows.

Consider following flow for searching a person with a certain expertise:

```
def searchFlow = {
        input {
            expertise(required: true)
            title("Search person")
        }

search {
            onEntry {
                [personInstanceList:
Person.findAllByExpertise(flow.expertise)]
            }
            on("select") {
                flow.person = Person.get(params.id)
            }.to("selected")
            on("cancel").to("cancel")
        }

selected {
            output {
                person {flow.person}
            }
        }
        cancel()
    }

}
```

This flow accepts two input parameters:

- a required expertise argument
- an optional title argument with a default value

All input arguments are stored in flow scope and are, just like local variables, only visible within this flow.

A flow that contains required input will throw an exception when an execution is started without providing the input. The consequence is that these flows can only be started as subflows.

Notice how an end state can define one or more named output values. If the value is a closure, this closure will be evaluated at the end of each flow execution. If the value is not a closure, the value will be a constant that is only calculated once at flow definition time.

When a subflow is called, we can provide it a map with input values:

```
def newProjectWizardFlow = {
    ...

managerSearch {
        subflow(controller: "person", action: "search",
                input: [expertise : "management", title: "Search project
manager"])
        on("selected") {
            flow.projectInstance.manager = currentEvent.attributes.person
        }.to "techleadSearch"
    }

techleadSearch {
        subflow(controller: "person", action: "search",
                input: [expertise : { flow.technology }, title: "Search
technical lead"])
        on("selected") {
            flow.projectInstance.techlead = currentEvent.attributes.person
        }.to "projectDetails"
    }

...

}
```

Notice again the difference between constant values like `expertise : "management"` and dynamic values like `expertise : { flow.technology }`

The subflow output is available via `currentEvent.attributes`