PROFESSIONAL COLDFUSION CONTENT FOR THE COLDFUSION PROFESSIONAL

The Fusion Authority Quarterly Update

FALL2006

THE BUILDING BLOCKS OF OBJECT-ORIENTED PROGRAMMING IN CF

FRAMEWORKS DEMYSTIFIED

ALL THIS AND MORE BY YOUR FAVORITE COLDFUSION GURUS!

FALL 2006 $14.95
ISSN 1932-0264
00
02>
9 771932 026000

WWW.FUSIONAUTHORITY.COM

CONTENTS

Fusion Authority Quarterly ■ Vol. 2 ■ Issue 2 ■ Fall 2006

Editorial

Columns

Features

Concepts

Tools

Lessons I Learned from My First Model-Glue Application

by Jeffry Houser

I tend to do a lot of web site maintenance in my daily routine. When I'm working on an application that has a lot of history, it often does not make sense to change the approach used in that development. Recently, I had the rare chance to build an application from scratch. It seemed like the ideal time to jump on the framework bandwagon that has been gaining momentum in the ColdFusion blogsphere. I decided to build the application in Model-Glue, version 1.1. The public beta of Model-Glue:Unity came out after I did this, so when I talk about Model-Glue in this article I'm referring to the 1.1 release, unless otherwise noted.

Model-Glue 1.1 comes with a quick start guide. This guide can help you get up and running, but I was left with a lot of questions, especially when trying to do something in depth. Unity documentation has been fleshed out a bit, but some of the issues I was having still apply. So, here are seven problems that I couldn't solve by reading the Model-Glue Quick-Start Guide and how I solved them.

Problem 1: Triggering Events

Problem: The Model-Glue documentation talks about how to respond to events in the config file, how to broadcast messages to call controller methods, and how to pass values back and forth from the controller and view. It never tells you how to trigger an event. So, how does Model-Glue handle requests?

Solution: Using the default application template as an example, all requests should be sent to the home page, presumably index.cfm. You'll never have to write code for the index.cfm page and it is probably best to consider it a part of the Model-Glue framework. A query string variable, event, should be appended to your URL. This value tells which event to fire. Your URL will probably look like this:

http://localhost/index.cfm?event=MyEvent

Events are defined in the ModelGlue.xml config file. Here is a sample of an event taken right from the Model-Glue sample application template:

```
<event-handler name="Home">
  <broadcasts />
  <views>
    <include name="body" template="dspBody.cfm" />
    <include name="main" template="dspTemplate.cfm" />
  </views>
  <results />
</event-handler>
```

The *event-handler* tag defines the name of the event we want to run, with the event details being located between the open tag and close tag. The details will include broadcast messages, views, and results. Results will execute a different event, and return the results to this event. The broadcast

section tells Model Glue to execute certain methods in your application's controllers. The above example doesn't contain any results or broadcasts instructions. It does, however, contain views. The views refer to templates in the view directory that will be rendered and the final view displayed. If no event is specified, a default event, also defined in the config file, will execute.

Problem 2: View Rendering

Problem: When a Model-Glue Event processes an event, it may render (a.k.a create) multiple views. The last view that is rendered is the one that is displayed, while the others just hang out in memory. I wanted to use a 'header' and 'footer' template at the beginning and end of each page, as is a common in a lot of web sites. How can you get all views to render instead of just the final one?

Solution: When a view is rendered, you can give it a name. All names can be accessed inside a view using the Model-Glue viewCollection object. You can access the contents of a rendered view using the getView function, like this:

```
viewCollection.getView("ViewName")
```

Or you can check for the existence of the view, using the exists function, like this:

```
viewCollection.exists("ViewName")
```

The QuickStart teaches you how to use these two things to put one view inside of another. While this approach works, I wasn't completely comfortable with it. You may end up with a lot of conditions on what to display, and I didn't want to mix those types of conditionals in with other display code. I decided to create a view template that does nothing but display other views. That way each view is self-contained, and this master template puts it all together. You will need to render the master template view last, of course. Here is a sample Master View template:

```
<cfif viewCollection.exists("header")>
  <cfoutput>#viewCollection.getView("header")#</cfoutput>
</cfif>
<cfif viewCollection.exists("body")>
  <cfoutput>#viewCollection.getView("body")#</cfoutput>
</cfif>
<cfif viewCollection.exists("footer")>
  <cfoutput>#viewCollection.getView("footer")#</cfoutput>
</cfif>
```

The code checks to make sure that the view exists, and if it does it is displayed. Otherwise it will not be displayed.

Problem 3: How Do You Perform Form Validation?

Problem: In the old days of ColdFusion development, you'd create an HTML form on one page, and then submit it to another page for processing. The processing template would perform any relevant server-side validation of the form data and if valid, store the information in a database or process it in some manner. On the processing page, you would refer to *Form* variables using the *Form* scope. How do you perform this action in Model-Glue? Where should your form submit the data? How do you access the form values for validation? Where do you put your validation code?

Solution: Your form should submit to index.cfm, specifying an event variable. I used a *URL* variable; I suspect that an event *Form* variable would work as well, but didn't test it. The form submission code would look like this:

```
<form action="index.cfm?event=ValidateForm" method="post">
```

ValidateForm is the name of the event that processes your form variable. That event broadcasts a "ValidateForm" message, which will run a method in the controller. Theoretically, the controller is part of the same request. Therefore, you could access *Form* variables inside the controller just by using the *Form* variable scope. However, this violates encapsulation and would be considered bad practice. The controller shouldn't know about *Form* variables. Instead, you can access *Form* (and *URL*) variables using the Model-Glue API. Each method in the controller receives an instance of the Model-Glue event object as an argument. The event object contains all *Form* and *URL* variables internally, and you access them using a GetValue method, like this:

```
arguments.event.GetValue("MyFormVariable")
```

You can also write values (such as a validation error message) to the event object using the SetValue method. These values become available in the view using the ViewState method.

```
arguments.event.SetValue("MyFormVariable",myValue)
```

While I'm on the subject of form validation, it took me a while to figure out how to determine if a checkbox was checked or not. An unchecked checkbox will not exist on the form action page. Normally you would default the values with CFPARAM or use an IsDefined to check for the existence of a value. In Model-Glue, you should use the event object's ValueExists function to tell whether a checkbox exists or not:

```
arguments.event.ValueExists("MyFormVariable")
```

If the value is not defined, then you can assume that the checkbox was not checked.

Here's a potential "gotcha": If you use the GetValue function to access a non-existent value, an error is not thrown. Instead, an empty string is returned. This can be counter-intuitive and misleading. At first I mistakenly believed that the checkbox was checked because no error was thrown when the application tried to access the nonexistent value.

In the end, I put all form validation inside the controller method. If it validates, the value is passed into the model for further processing; otherwise an error or list of errors is displayed to the user. If I had to do this all over again, I'd move the validation business logic out of the controller and into the model. I think that would be the best approach. But hindsight is 20/20.

Problem 4: How do I redirect?

Problem: If form validation fails, I need to display the error messages to the user and allow him to correct his information. If form validation is successful, I need to process the data (most likely saving it in a database) and thank the user for the input. This is a very common approach, but is not covered in the quick start.

In a traditional ColdFusion application, I would use the CFLOCATION tag to redirect to the Thank You page if there was no error. If there were an error, I'd CFINCLUDE a page that displays the initial form, along with some error messages. Redirecting with a CFLOCATION from inside the controller seemed to work, but it felt like an odd approach. Including a file from the controller just seemed wrong. The config file should dictate what views are displayed, right? Would an include in the controller

display at all? I'm not sure. We could render both the 'wrong input' and 'success' messages and then display the appropriate one based on whether or not there is an error. But, that seemed like extra, unnecessary work. What is the best way to do this with Model-Glue?

Solution: My first approach to this problem was to use an event object method called forward. It stops processing the current event and starts processing another. The state of the request is retained, so the form values or error variables are still accessible via views in the new event. Inside a controller method, you would do something like this:

```
arguments.event.forward("NewEvent")
```

Normally, you'd have to write some sort of code to retain state between the initial page and the page that CFLOCATION redirects to. You might store the information as cookies, as a WDDX packet, or as part of the *Session* scope. Model-Glue handles this behind the scenes.

Despite my initial excitement about the forward method, it is not considered a best practice. Model Glue offers a much more elegant way to implement the same functionality. Inside the controller, you can announce results using the addResult method of the event object. When you do this, the config files knows to execute another event based on that result. Announcing a result is conceptually similar to how Model-Glue broadcasts events. When broadcasting events, the event makes the broadcast, which is heard by a listener, and the listener executes a method in the controller. Announcing a result just works the opposite way. The controller tells Model Glue to execute a result. Results are defined in the config file as part of the event declaration. I used the result event earlier to create a master view template, as an example.

This is how you might announce a successful validation result:

```
arguments.event.addResult("valid")
```

A sample validation event handler might look like this:

```
<event-handler name="handleForm">
  <broadcasts>
    <message name="handleForm" />
  </broadcasts>
  <views />
  <results>
    <result name="valid" do="showThankYou" redirect="true" />
    <result name="invalid" do="showForm" />
  </results>
</event-handler>
```

A form is submitted and the handleForm event is fired. The handleForm message is broadcast, and some controller method is fired. If the valid result is announced, then the showThankYou event is executed. If the invalid result is announced, then the showForm event is executed. If no results are specified then the specified views are processed normally.

The redirect attribute of the result tag will force the result to be executed immediately, ignoring unexecuted code in the current event. One gotcha I discovered is that if a result is not named it will be always executed as part of the request. If no redirects are specified, then this unnamed result will trump any results that you had announced. As such, I would not recommend mixing named results with unnamed results because the named result will never execute.

Problem 5: How To Handle Shared Scope Variables?

Problem: I'm building an e-commerce system. I need to know how to handle sessions and keep track of important data across multiple browser requests. What facilities does Model-Glue offer to help accomplish this?

Solution: The answer is either "the same facilities that ColdFusion offers" or "none". You can take your pick. Model-Glue is designed to facilitate communication between your backend CFCs and your front-end interface. When dealing with an Application.cfc, there is no user interface, so there is no need to code this using Model-Glue techniques. You can set up your *Application* and *Session* variables just as you would in an application that uses a different framework, or even no framework at all.

The model part of your application shouldn't be accessing the shared scope variables because it breaks encapsulation. The view shouldn't be accessing them for the same reason. I decided to access the shared scopes, as necessary, inside the controller. This seemed like the logical place. If the model needs values stored in the session, I can pass them in as parameters. If the view needs *Session* variables, I can set them in the event object and they'll be accessible in the view via Model-Glue's ViewState object.

Problem 6: Why won't my JavaScript work?

Problem: In one of the views, I have an HTML form with some JavaScript validation. When I load the page in Model-Glue, I cannot get the JavaScript to run successfully. What gives?

Solution: This one had me tearing my hair out for a bit. If you have debugging turned on, and the "Report Execution Times" feature checked, then each rendered view will be displayed as part of the debugging output. In my experience, each view was displayed four times. That means there were four identically-named JavaScript functions on the page, along with four identically-named form elements. Which of these four functions should be called to process your results? The browser doesn't know. It just gets confused. You can either disable "Report Execution Times" from the debugging settings in the ColdFusion Administrator or turn off debugging using the CFSETTING tag.

Problem 7: Building the Model

Problem: Model-Glue is a Model View Controller framework, but it did not help me build a model. Using Model-Glue did not enhance (or limit) my model-building skills. This should come as no surprise, as the purpose of Model-Glue is to facilitate communication between the model (backend code) and the view (display code); it does not build the model or view for you.

Solution: There are many books, blogs, articles, and other resources that talk about building applications. I suggest searching out some of these resources. I recommend *Head First Design Patterns* as a good read for beginners. You'll often find a wealth of information on such topics. I also recommend searching the ColdFusion-related blogs. Try reading some of the aggregators such as MXNA (http://weblogs.macromedia.com) or Feed Squirrel (http://www.feed-squirrel.com/).

So, there you have it: seven things I learned about Model Glue. I think the problems that I experienced are going to be common problems that others will experience, and I hope you're able to use my solutions to bring your next project to a successful completion.

Jeffry Houser is owner of DotComIt, a web-consulting firm based out of Connecticut. He has written three ColdFusion books, has written numerous articles, has presented at User Groups and conferences all over the US, and co-manages the Connecticut CFUG. You can find out the latest and greatest about him at his blog at http://www.jeffryhouser.com.