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

Fusebox 5 Fundamentals

By Sean Corfield

Fusebox - What and Why

Overview

Fusebox 5 is the latest release of ColdFusion's oldest and most popular application framework. If you've never used Fusebox, this article will introduce you to its concepts and benefits. If you've been using Fusebox 4 or Fusebox 4.1, this article will explain what's new in Fusebox 5 and why you should upgrade.

Let's start by defining what we mean by an application framework in the context of Fusebox. An application framework is a set of files that provides two things:

1. Reusable code that is common across most web applications.
2. A standardized structure for your applications so that maintenance is easier.

Fusebox has been designed specifically to help you build more maintainable applications, regardless of your skill level with ColdFusion. Whether you write procedural code or object-oriented code, whether you are a novice or an expert — Fusebox can support your style of programming. This makes Fusebox particularly attractive to development teams where the skill level varies dramatically across the team or where the programming style varies, encompassing both procedural and object-oriented approaches.

Fusebox tries hard not to **enforce** a particular programming style but it strongly encourages best practices such as separating display code from database code from other business logic. Fusebox supports the widely used "Model-View-Controller" design pattern but does not force you to use it. Fusebox supports using ColdFusion Components in your application but, again, does not force you to do so.

Concepts

The basic metaphor behind Fusebox is a practical, everyday one: the electrical circuits in your home. In the same way that electricity flows into your house through a central fusebox and is then routed to individual circuits that control related appliances, the Fusebox application framework has a central control point – the fusebox – and organizes code into circuits containing related functionality. For example, a task manager application might have a circuit that handles user identity and a circuit that handles tasks. As the application grows, new circuits can be incorporated, each containing a group of related functionality. If you build a new application, you may well be able to reuse a circuit from a previous application, such as user identity.

All requests come through the central "fusebox" by way of a standard *URL* or *Form* variable, often called **fuseaction**. The value of this variable specifies which circuit to route the request to and which "fuseaction" to invoke within that circuit:

index.cfm?fuseaction=user.login

This specifies that the login fuseaction within the user circuit should be executed.

Within a fuseaction, you can specify that Fusebox execute multiple operations, using "verbs" in an XML file for each circuit. This could be as simple as including a ColdFusion source file using the include *verb* or as complex as conditional and looping control logic, using the *if* and *loop* verbs, for example. Fusebox has a very small set of verbs that allow you to perform basic control logic, at the same time making the Fusebox syntax easy to learn.

In keeping with the electrical metaphor, Fusebox refers to any included ColdFusion files as "fuses" to emphasize their intended small, atomic nature. In addition, Fusebox encourages the use of a file-naming convention to remind developers to separate their code appropriately. Files containing display logic (and HTML) are usually given the prefix **dsp** (for display) or **lay** (for layout), files containing database queries are usually given the prefix **qry** (for query) and files containing other business logic are usually given the prefix **act** (for action). In the first few releases of the Fusebox framework, all of this was very much just convention with only a few core files in the framework. With Fusebox 4, the structure became more formalized with the introduction of XML files to specify the "fusebox" itself – the set of circuits in the application and the various parameters of the framework – and to specify the fuseaction code in each circuit.

Returning to our task manager application, we might choose to organize our code like this:

- Application.cfm
- index.cfm – simply includes the main Fusebox core file.
- fusebox.xml – specifies where to find the user and task circuits as well as other application parameters.
- user/
  - circuit.xml – specifies the various fuseactions in the user circuit
  - dsp_login.cfm
  - ... other CFML fuse files ...
- task/
  - circuit.xml – specifies the various fuseactions in the task circuit
  - qry_task_list.cfm

A Fusebox Glossary

Circuit: A group of fuseactions, often self-contained, that implement a coherent subset of an application's functionality. Each circuit typically addresses a specific concept from the application's domain such as "user management" or "shopping cart" or "invoicing".

FLiP — Fusebox Lifecycle Process: A "project management approach to planning, architecting, coding, and testing a Fusebox application". FLiP focuses on building an HTML prototype of the website with the client and using that to drive the architecture of the Fusebox application that will power the site.

Fuse: A ColdFusion file that is executed by the *include* verb. Each fuse contains a simple piece of ColdFusion code that does just one "job": executes a database query or displays some HTML or performs some specific business operation. Fuse files are typically named with a prefix that indicates which type of job they do: act (action), dsp (display), qry (query), lay (layout), etc.

Fuseaction: A named handler for a specific request within a given circuit. A fuseaction defines the verbs that are executed in order to perform that request.

Lexicon: A collection of one or more related verbs. Each lexicon is stored in a separate directory underneath the lexicon/ directory within each application. A lexicon is made available within a circuit by declaring its location in the circuit tag, using an XML namespace, e.g., <circuit xmlns:prefix="location"/> which tells Fusebox that verbs with the specified prefix (<prefix:myverb/>) can be found in the lexicon/location/ directory.

Verb: An individual XML tag in the circuit.xml file that is compiled to CFML by executing the verb's implementation in the appropriate lexicon. A verb may be built-in (such

```
    qry_update_task.cfm
    ... other CFML fuse files ...
```

Here we have just two circuits, one containing all of the fuses related to user identity and the other containing all of the fuses related to task management. We can see a display fuse in the user circuit that would contain the login form. In the task circuit, we see query fuses that fetch tasks and update a task. There would be a number of other fuse files that contain the rest of the logic and display code for the application but the outline above should give you a sense of how a Fusebox application is structured.

Operations on tasks might be protected by a security check: we might require that users are logged in before they can use the task manager. We might want to execute the user.authenticate fuseaction prior to any fuseaction in the task circuit, for example. Fusebox provides an easy way to do this by specifying prefuseaction and postfuseaction operations on each circuit. In our task circuit, we might simply say:

as *set* or *include*), or it may be part of a user-defined lexicon.

XFA – eXit FuseAction: A fuseaction that represents an "exit point" from a given web page. Exit points are links and form actions that take you from one page to another. It is considered good practice not to hard code links into pages but rather use variables to create those links, with those variables usually set in the fuseaction that includes the display fuse. This allows display fuses to be reused when all that changes are the destinations of links.

```
<prefuseaction>
  <do action="user.authenticate"/>
</prefuseaction>
```

Fusebox would then automatically run this fuseaction for us before executing any other fuseaction in the task circuit. User.authenticate would check that the user is logged in and, if not, would cause a redirect to the user.login fuseaction. This modularity can allow developers to reuse whole circuits in other applications.

Justification

The simple structure and organization of a Fusebox application makes maintenance easier by allowing any developer who is familiar with the framework to pick up the code and get a sense of its purpose just by looking at the circuit layout and, within each circuit, the available fuseactions. Similarly, the file naming convention allows developers to quickly locate code and make changes in a systematic way.

Fusebox has been around long enough that there is a wealth of information out there about the framework, so it is easy to learn and get support from the community.

Books are available from Techspedition (http://techspedition.com/) and Proton Arts (http://protonarts.com/) as well as other technical bookstores.

The Fusebox website (http://fusebox.org/) has documentation about the framework and forums where you can engage with the large Fusebox community if you need assistance. There are also several mailing lists, three of which are linked from the home page of the website, as well as an extensive resources list on the website.

All of the foregoing should make you feel comfortable about choosing the Fusebox application framework for your projects. In the remainder of this article we're going to focus on what's new in Fusebox 5 and why existing Fusebox developers will want to upgrade.

What's New in Fusebox 5?

Compatible

The first and most important thing to note about Fusebox 5 is that it is designed to be completely backward compatible with Fusebox 4.1 (and Fusebox 4). If you have an existing Fusebox 4.1 application, it should run without any changes on this new release of the framework.

Multiple Applications

Previous releases of the Fusebox framework assumed that you built monolithic applications and that each individual Fusebox application was also a separate ColdFusion application. Fusebox stored information about your application in the application.fusebox data structure. This made it hard to write Fusebox applications that integrated well with each other because they couldn't share application data or session data, making it hard to have single sign-on across multiple applications, for example.

By default, Fusebox 5 uses application.fusebox for information about your application but you can now override this by setting the FUSEBOX_APPLICATION_KEY variable in index.cfm before you include the framework runtime file. This allows you to have multiple Fusebox applications sharing a single ColdFusion application name without overwriting each other.

The Fusebox Grammar

The following verbs are built into Fusebox 5:

do: execute another fuseaction as part of the current fuseaction. *do* may have optional parameter children (new in Fusebox 5).

if: conditionally execute other verbs. *if* may contain a true group and/or a false group.

include: include a fuse file. *include* may have optional parameter children (new in Fusebox 5).

instantiate: instantiate a ColdFusion Component that was previously declared in the <classes> section of the fusebox.xml file.

instantiate may have optional argument children (new in Fusebox 5).

invoke: invoke a component's method. *invoke* may have optional argument children (new in Fusebox 5).

loop: repeatedly execute a group of other verbs.

relocate: relocate to a new URL – similar to CFLOCATION in ColdFusion.

set: set a variable to a given value – similar to CFSET in ColdFusion.

xfa: set an eXit FuseAction variable to a given fuseaction.

Application Initialization

Just as ColdFusion MX 7 introduced Application.cfc and a way to write code that executes just once at application startup, through the onApplicationStart() method, so Fusebox 5 introduces a portable way to do this within the framework. Fusebox 4.1 introduced an optional file called fusebox.init.cfm that is executed automatically at the start of each request. Fusebox 5 introduces an optional file called fusebox.appinit.cfm that is executed automatically when the first request is made for the application. Like CFMX7's onApplicationStart() method, the code in fusebox.appinit.cfm is executed inside a lock to ensure it is thread safe. Fusebox 5 also provides a way to execute code at application startup, i.e., as fuseactions specified in the XML files, within the framework itself.

Custom Lexicons

In order to understand this major extension in Fusebox 5, we need to cover in a little more detail the machinery that Fusebox uses to handle requests. The first time a specific request comes into the framework, Fusebox locates the appropriate circuit and the fuseaction within it and then converts the XML for that fuseaction into ColdFusion code, which it writes to a parsed directory. Subsequent requests for that fuseaction are handled by simply including the generated file.

Each fuseaction is specified in the XML file as a sequence of verbs to be executed. For example, looking at our task manager application:

Listing 1: Showall Fuseaction in the Fusebox XML file

```
<fuseaction name="showall">
    <set name="max" value="20" />
    <include template="qry_task_list" />
    <xfa name="view" value="task.showtask" />
    <include template="dsp_task_list" contentvariable="body" />
    <include template="lay_main" />
</fuseaction>
```

Here we see three XML "verbs" in use: *set*, *include* and *xfa*. The set verb generates code that sets a variable (called max) to the specified value (20, in this case). The *include* verb generates code that includes the specified file (with a .cfm extension automatically appended). The *xfa* verb is a special case of the *set* verb that generates code to set an eXit FuseAction variable to the specified value (XFAs are kept in a special structure variable called xfa). The generated code from this fuseaction will be stored in parsed/task.showall.cfm (based on the name of the circuit/fuseaction).

The XFA variable is used in a display fuse to construct a link to another page:

```
<a href="#myself##xfa.view#&amp;taskId=#taskId#">View this task</a>
```

Myself is usually set in fusebox.init.cfm and refers to the base URL of the Fusebox index.cfm page:

```
<cfset self = "index.cfm">
<cfset myself = self &"?" & myFusebox.getApplication().fuseactionVariable & "=>
```

This technique allows the fuseaction to determine where the links inside the display fuse will actually go, allowing display fuses to be reused more easily.

In Fusebox 4, the built-in verbs were all handled magically inside the framework and there was no way to add new verbs. Fusebox 4.1 introduced an experimental mechanism called lexicons that allowed Fusebox developers to create their own verbs as ColdFusion files and specify their location (using the <lexicons> section in the fusebox.xml file). User-defined verbs could not have nested child verbs and were, overall, quite a pain to write.

Fusebox 5 takes this basic idea and integrates it into the core of the framework. Apart from *do*, which is more of a compiler directive, there are no magic built-in verbs. All of the standard verbs in Fusebox 5 exist as ColdFusion files that are executed by the framework. User-defined verbs are just like those standard verbs except that they must be introduced using an XML namespace in each circuit that needs them.

The basic skeleton application for Fusebox 5 shows an example of this as well as providing almost a dozen verbs that mimic tags within the ColdFusion language:

```
<circuit access="public" xmlns:cf="cf/">
```

This introduces the XML namespace cf and defines the path for it: cf/. The path is relative to the lexicon/ directory within your application root and doesn't need to match the namespace name. Once the XML namespace has been declared, whenever Fusebox sees a verb in cf namespace, it will look in lexicon/cf/ for the implementation file:

```
<cf:dump label="Attributes Scope" var="#attributes#">
```

Fusebox expects to find dump.cfm in the lexicon/cf/ directory and will execute it much like a custom tag is executed in ColdFusion. Each verb is run twice, first with an execution mode of "start" when the tag is opened and then again with an execution mode of "end" when the tag is closed. The ColdFusion file that implements the verb generates ColdFusion code and writes it to the parsed/ file using convenience methods such as fb_appendLine():

```
fb_appendLine(`<cfdump #fb_.label# var="#fb_.verbInfo.attributes.var#">');
```

The basic skeleton application for Fusebox 5 includes several Fusebox verbs that mimic ColdFusion tags, such as try / catch, switch / case / defaultcase, and others. These are good examples of how to extend the Fusebox XML language in any way you want. For a more complex example, I showed on my blog what it might look like to have a set of Model-Glue style verbs for use within Fusebox:

http://corfield.org/entry/Fusebox_5_in_a_ModelGlue_style
http://corfield.org/entry/Fusebox_5__The_Power_of_Custom_Lexicons

Note that this blog entry does not use Model-Glue code at all; it merely emulates it as a Fusebox 5 custom lexicon.

You might wonder why you would want to duplicate ColdFusion tags into the Fusebox grammar. The built-in Fusebox grammar is deliberately very simple so that you are not tempted to put too much logic into your fuseactions; the original guiding principle for Fusebox was that all logic belonged in fuses instead. However, with the increasing use of ColdFusion Components in Fusebox applications, there are a number of situations where the simple Fusebox grammar gets in the way of writing clean code. For example, although you can invoke a component's method directly within a fuseaction, if the method can throw an exception that you want to handle gracefully, Fusebox 4.1 forced you to create an action fuse and place the invocation in there, surrounded by your try / catch logic. This made the application flow harder to read since you were forced to open up a fuse just to see the method call, as well as being forced to put conditional checking into your fuseactions. Fusebox 5 allows you to keep simple control logic like this directly in the fuseaction, making it easier to follow the application flow. Compare the following two examples:

Listing 2: Fusebox 4.1 Code:
Showquote Fuseaction

```
1  <fuseaction name="showquote">
2    <include template="actGetQuote" />
3    <if condition="len(stockPrice) neq 0">
4      <true>
5        <include template="dspStockQuote" />
6      </true>
7      <false>
8        <include template="dspQuoteFailed" />
9      </false>
10   </if>
11 </fuseaction>
```

actGetQuote.cfm

```
1  <cftry>
2    <cfinvoke component="#application.stockTracker#" method="getQuote"
       returnvariable="stockPrice">
```

```
3       <cfinvokeargument name="symbol" value="#attributes.symbol#">
4     </cfinvoke>
5  <cfcatch type="any">
6     <cfset stockPrice = "">
7  </cfcatch>
8  </cftry>
```

Listing 3: Fusebox 5 Code

```
1  <fuseaction name="showquote">
2     <cf:try>
3        <invoke object="application.stockTracker" method="getQuote"
           returnvariable="stockPrice">
4           <argument name="symbol" value="#attributes.symbol#" />
5        </invoke>
6        <include template="dspStockQuote" />
7     <cf:catch type="any">
8        <include template="dspQuoteFailed" />
9     </cf:catch>
10    </cf:try>
11 </fuseaction>
```

By providing extensibility through optional lexicons in this way, Fusebox 5 retains the small, easy-to-learn grammar of Fusebox 4.1 while offering increased expressiveness to those developers who need it, especially those using ColdFusion Components extensively.

XML Grammar

In addition to the major new language features introduced by the custom lexicons, Fusebox 5 provides a number of smaller improvements to the language as defined in Fusebox 4.1. In Fusebox 5, there are no restrictions on nesting of *if* and *loop* verbs as there were in Fusebox 4.1 and Fusebox 5 now supports the same five styles of loop that ColdFusion itself supports (query, collection, condition, list, from/to).

There has been a lot of development in the area of CFC support in Fusebox 5. In addition to the "once-per-application" initialization hooks discussed above, a new *argument* verb has been added as a child to the *instantiate* and *invoke* verbs to provide a much cleaner syntax for working with CFCs that is more in line with ColdFusion's own syntax:

```
<invoke component="util.myCFC" method="doSomething">
   <argument value="firstArg" />
   <argument value="2" />
</invoke>
```

You can specify unnamed (positional) arguments as above or named arguments as below:

```
<invoke component="util.myCFC" method="doSomething">
   <argument name="arg1" value="firstArg" />
   <argument name="arg2" value="2" />
</invoke>
```

The *do* and *include* verbs have a similar extension so that parameters can be passed into fuseactions and fuses. In the security example on page 59, we might want to pass in the name of a fuseaction that the user login code can redirect to once a user has successfully authenticated:

```
<prefuseaction>
  <do action="user.authenticate">
    <parameter name="returnAction" value="task.#myFusebox.thisFuseaction#" />
  </do>
</prefuseaction>
```

This will set the variable returnAction to the value of the current circuit (task) and fuseaction for the duration of the user.authenticate fuseaction.

Runtime Control

In addition to the changes in the Fusebox language itself, there are a number of improvements in the control you have over the runtime behavior of the framework.

Fusebox 5 provides a new execution mode that only reloads individual circuits that have changed, without resetting application data on each request. You can also tell Fusebox 5 to remove all the previous parsed files or to regenerate all public fuseactions in the entire application – both of which can be useful when you are upgrading a production website to ensure that you have all the latest code running.

Stephen Judd has created a very useful extension for Firefox (http://cecf1.unh.edu/fusebox/) that lets you easily and automatically control the execution of your Fusebox 5 application. (He also has a version of the extension for Fusebox 4).

The new runtime change that will most likely appeal to the largest number of Fusebox developers is the addition of a debug mode for the framework. Just add this to your fusebox.xml file:

```
<parameter name="debug" value="true" />
```

and debugging information will be displayed at the end of each page you request. The debugging information shows every fuseaction executed and every fuse included, as well as the time taken to get to each entry and a count of any repeated fuseactions and fuses.

This can be very helpful in identifying bottlenecks in your application as well as debugging unusual behavior because you can see the execution path through your code.

Why Upgrade?

Since Fusebox 5 is backward compatible with Fusebox 4.1, we might as well ask, "Why wouldn't you upgrade?" Well, an upgrade is going to mean that you need to test your applications completely on the new framework release so there is always a cost associated with any upgrade.

The next question might be: "If it's backward compatible, what benefits will I get from simply running my Fusebox 4.1 application on it?" That's a good question. If you really aren't planning to change your codebase to take advantage of the new features, you won't get much benefit from an upgrade. However, being able to simply enable tracing / debugging in fusebox.xml and get timing information about your application with no code changes might be sufficient incentive to encourage you to upgrade even without changing your codebase.

Finally, you might just find the changes discussed appealing enough to make this an easy decision. Perhaps the ability to use some of the sample verbs from the new Fusebox 5 skeleton application, such as *cf:dump*, might be cause enough. Maybe the new CFC-related syntax or the parameters on *do* and *include* verbs will entice you? Or perhaps the attraction is the option to perform all your per-application initialization directly within the framework in a thread-safe manner.

Fusebox 5 is the first major release of Fusebox that is backward compatible with the previous release. It protects your investments – both intellectual (knowledge of Fusebox 4.x) and physical (your Fusebox 4.x codebase) – while offering increased functionality in a number of important areas. The completely rewritten core files provide a solid basis for future development, assuring you of continued growth and richness of the Fusebox framework. There has never been a better time to be a Fusebox developer!

Sean is currently Senior Computer Scientist and Team Lead in the Hosted Services group at Adobe Systems Incorporated. He has worked in the IT industry for nearly twenty-five years, first in database systems and compilers, then in mobile telecoms, and finally in web development. Sean is a staunch advocate of software standards and best practices, and is a well-known and respected speaker on these subjects. Sean has championed and contributed to a number of ColdFusion frameworks, and is a frequent publisher on his blog, http://corfield.org/.

ColdFusion OO Resources

Blogs

- Application Generation (Peter Bell) - http://www.pbell.com/
- An Architect's View (Sean Corfield) - http://corfield.org/blog/
- Clearsoftware.net (Joe Rinehart / Model-Glue) - http://www.clearsoftware.net/
- Nictunney.com (Nic Tunney) - http://www.nictunney.com/
- cf.Objective: n00b Entries (Jared Rypka-Hauer) - http://www.web-relevant.com/blogs/cfobjective/index.cfm?mode=cat&catid=06286005-103D-536E-CC1C11AB465935B7
- Brian's Blog: OOP CF Entries (Brian Kotek) - http://www.briankotek.com/blog/index.cfm?mode=cat&catid=E3B11FA8-3048-23C1-DDADAFDD4093690F

Frameworks

- Mach-ii
  http://www.mach-ii.com
  http://www.mach-ii.info
- Model-Glue
  http://www.model-glue.com
- Fusebox
  http://www.fusebox.org
- Reactor:
  http://trac.reactorframework.com
- ColdSpring:
  http://www.coldspringframework.org
- Transfer:
  http://www.compoundtheory.com/transfer
- ObjectBreeze:
  http://www.objectbreeze.com
- onTap:
  http://www.fusiontap.com