CMD Internet Direct Marketing Engine

Application Architecture Guide
Version 1.10
9/21/2001

Table of Contents:

Document History

The following is a history of the life of this document.

Date	Reviser	Revision Description
09/17/2001	Robert Foley Jr.	Created Document
09/20/2001	Robert Foley Jr.	Added Tutorials, Section Headers
09/22/2001	Robert Foley Jr.	Added more tutorials and code samples
09/23/2001	Robert Foley Jr.	Updated tutorials and code samples to reflect new research
09/27/2001	Robert Foley Jr.	Added directory naming standards spec

Authors

The following individuals contributed to the creation of this document.

- Robert Foley Jr *(Interactive Architect)*

References and Resources

The following information is provided as additional information or referenced within this document.

Document Title	Document Name	Location
ColdFusion Fusebox Methodology	ebook_fusebox-11-02-00.pdf	\\IMPALA\projects\interactive\Jobs\Other_Jobs\1765_Internal_IDM\Info_Design\highlevel-design\ApplicationArchitecture\Support_docs http://www.secretagents.com/products/index.cfm?fuseaction=product&product_id=5&page_id=24
Mastering Fusebox : Fusebox 101	Online Course: it's free!	http://halhelms.com/OnlineCourses/BasicFusebox/index.cfm
Fusedocs Lesson	Online Course: it's free!	Http://halhelms.com/OnlineCourses/Fusedocs/index.cfm
Application Directory Structure Diagram	CMD-IDM-appdirmodel-ver1.0.vsd	\\IMPALA\projects\interactive\Jobs\Other_Jobs\1765_Internal_IDM\Info_Design\highlevel-design\ApplicationArchitecture
Functional System Module Diagram	CMD-IDM-systemmodules-Ver1.4.vsd	\\IMPALA\projects\interactive\Jobs\Other_Jobs\1765_Internal_IDM\Info_Design\highlevel-design\ApplicationArchitecture
System Use Case Diagrams	CMD-IDM-System-UseCases-ver1.2.vsd	\\IMPALA\projects\interactive\Jobs\Other_Jobs\1765_Internal_IDM\Info_Design\highlevel-design\ApplicationArchitecture
Primary Use Cases Document	CMD-IDM-Primary-UseCases-ver1.0.doc	\\IMPALA\projects\interactive\Jobs\Other_Jobs\1765_Internal_IDM\Info_Design\highlevel-design\Global_Design

Overview

The following document will provide a detailed explanation of the application framework used for the IDM (Internet Direct Marketing) engine. The application framework is loosely modeled after the Fusebox programming framework (http://www.fusebox.org). The architecture is a multi-programming language application framework that provides programming structure and good programming practices to the application as a whole. By using the framework it ensures the development staff utilizes modular, OOD (object oriented design), and OOP (object oriented programming) metaphors throughout the life of the project.

Please note: This document references terminology and techniques outlined within the Fusebox Methodologies ebook. This document can be referenced on the network at this location:

Location: \\IMPALA\projects\interactive\Jobs\Other_Jobs\1765_Internal_IDM\Info_Design\highlevel-design\ApplicationArchitecture\Support_docs

File Name: ebook_fusebox-11-02-00.pdf

Alternatively, it can be purchased for about $12 to $29 at the FUSEBOX URL provided above.

Before we can go into detail about the specific architecture of the IDM system it is important to grasp the basics of the fusebox programming framework as well as some of the processes involved in developing within the framework. The following sections will provide a detailed look at the fusebox system and documentation practices used. Please note, the information provided in the following sections are outlining the newest design of Fusebox, which is more advanced than the e-book mentioned above. After all the fusebox methodology and framework is an evolving effort and the book was written in 2000. The good thing is the newer version of the fusebox spec is an augmentation or addition to the spec and doesn't radically change the way you would create a fusebox application.

Once you have read the tutorials and ran through the exercises the next sections will outline rules and properties of the architecture that will be built for the IDM system. Directory and file naming standards will be defined and an detailed outline of the system will be discussed.

And now, the tutorials.

Fusebox 101

Written by: Hal Helms

http://halhelms.com/OnlineCourses/BasicFusebox/index.cfm

Who needs Fusebox? What's the big fuss about an application's *architecture*? After all, applications aren't essentially about architecture--they're about *doing* things--things like logging in, querying and displaying a result set from a database, or making a request for a credit card validation.

Suppose that a user wishes to log into an application. The user enters their username and password and clicks a submit button. That can be handled by writing a file called Login.cfm, whose code looks like this:

```
<form action="ValidateUser.cfm" method="post">
   User Name: <input type="Text" name="userName"><br>
   Password: <input type="Password" name="password"><br>
   <input type="Submit" value=" ok ">
</form>
```

Another file, ValidateUser.cfm, then would be written to accept the form variables passed to it, run a query against a database and then either show the user a menu or return them to try another login.

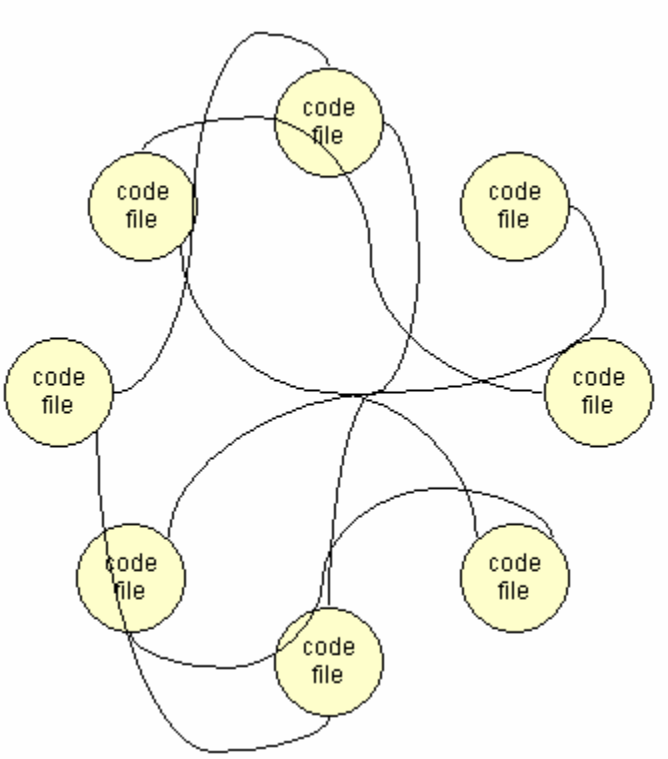
```
<cfquery datasource="myDSN" name="myQ">
   SELECT userID, firstName, lastName
   FROM Users
   WHERE userName = '#form.userName#'
   AND password = '#form.password#'
</cfquery>

<cfif myQ.recordCount>
   <cflocation url="UserMenu.cfm">
<cfelse>
   <cflocation url="Login.cfm">
</cfif>
```

What could be simpler?

Not much--and if your applications aren't any more complex than this, you won't need Fusebox--or any other methodology. Unfortunately, applications are never this simple. Even applications that start off simple--where the client *promises* that they will stay simple--almost never do. I've become convinced that the most dangerous words a client can utter are, "Wouldn't it be nice..."

In fact, studies show that between 70% and 90% of the entire cost of an application over its life is the ongoing maintenance of the application. It's a universal truism that developers would almost always rather rewrite than try to maintain someone else's code. Why is this? One excellent reason is that in a real-world application (unlike the simplistic one I've shown), the direct links from one code file to another--all those `<cflocation>`s and `<form action>`s--become enormously difficult to keep track of.

Whether maintaining an application or developing it fresh, another problem rears itself--if you make a change in one file, what others files will be effected? The problem is even worse if several developers are working on the same application. The bigger the application, the more there is to keep track of--until the complexity and the interconnections preclude any but the most experienced developers from working on it. If you think of a pyramid turned upside down--and you at the bottom like Atlas carrying the world atop his shoulders, you won't be far off. And heaven help you if you have a distributed development team. The problems multiply geometrically and we have tangled code that looks like a plate of spaghetti.



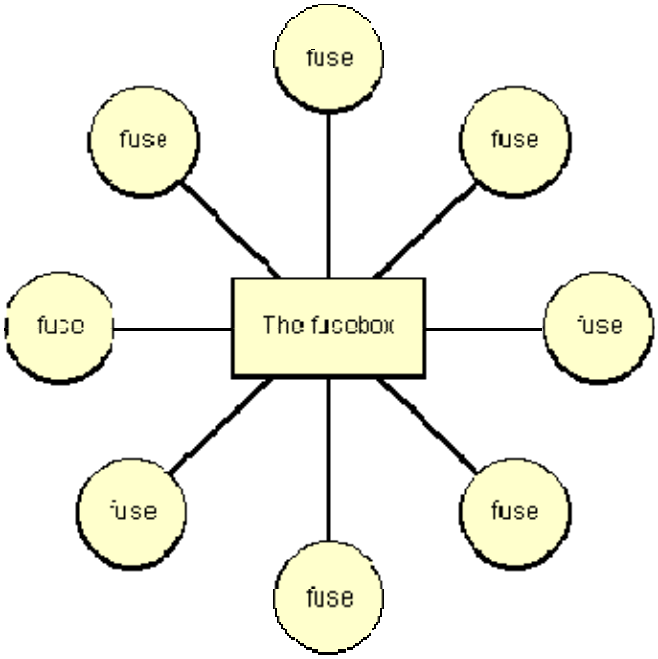
I won't belabor the difficulties of development as you probably wouldn't be looking at Fusebox if you hadn't experienced either these or other problems. The point I'd like to make is that the problems come about as a *natural result* of the normal development process. There's nothing mysterious about it; there aren't evil forces conspiring to make your life difficult, no matter how true that may seem at 2 am of the night before a deadline. If you want to change the results, *you must change the process*.

So how does Fusebox help--and how much is this help going to cost in terms of changing the way you write code?

Surprisingly little. Here is that same code you saw a minute ago written to the Fusebox methodology:

```
<form
   action="index.cfm?fuseaction="#attributes.XFA.validateUser#"
   method="post">
   User Name: <input type="Text" name="userName"><br>
   Password: <input type="Password" name="password"><br>
   <input type="Submit" value=" ok ">
</form>
```

Fusebox helps developers build robust and scalable web applications easily, surely, and quickly--and it's surprisingly simple. The tangle of spaghetti code is replaced with a simple, ordered, easy-to-maintain architecture:

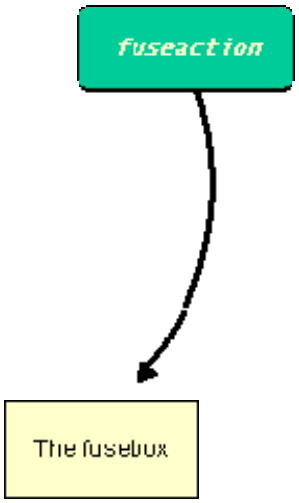


A Fusebox application works by responding to requests to do something--a *fuseaction*, in Fusebox parlance. This request may come about as a result of a user action--a user submitting a form, for example, or clicking a link--or it may occur as a result of a system request. Let's consider the first situation.

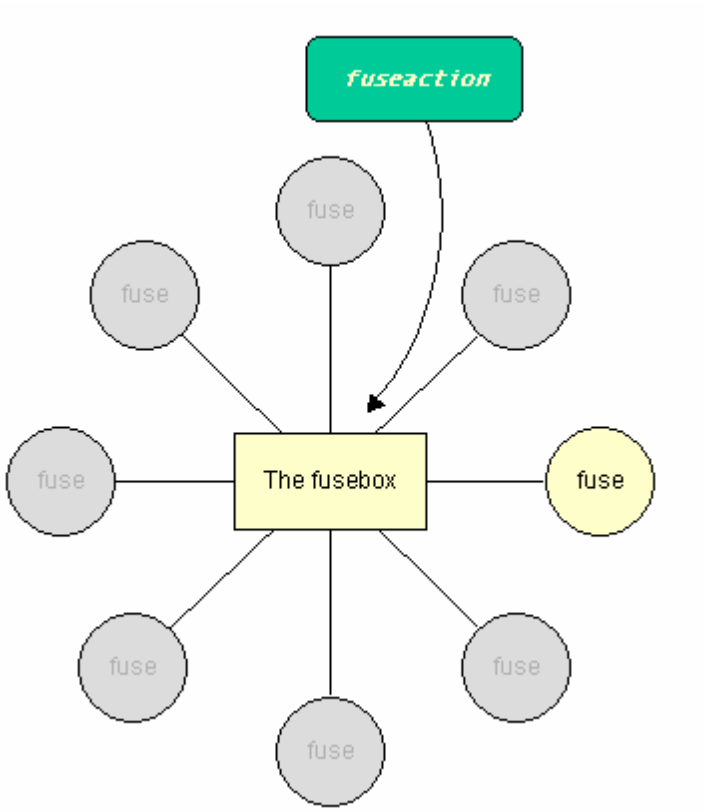
You type in the URL for `www.halhelms.com`, your favorite Fusebox site! You press the secret combination of keys while doing the prescribed breathing exercises that gets you to the secret portion of the site. There you are asked for your username and password. This is the code you saw on the last page:

```
<form
   action="index.cfm?fuseaction="#attributes.XFA.validateUser#"
   method="post">
   User Name: <input type="Text" name="userName"><br>
   Password: <input type="Password" name="password"><br>
   <input type="Submit" value=" ok ">
</form>
```

When you submit the form, index.cfm is called. In fact, *everything* an application can do is done by sending a fuseaction request to index.cfm. This file, so central to the methodology, is called the *fusebox*. When a fusebox is called, a variable called "fuseaction" is also sent.



The fusebox's main job is to route a fuseaction request to one or more code files called *fuses*. These files are typically small and have well-defined roles.



Here is the code for a really small fusebox--one that is only capable of handling a couple of different fuseactions.

```
<cf_formURL2attributes>

<!--- In case no fuseaction was provided, I'll set up one
to use by default. --->
<cfparam name="attributes.fuseaction" default="login">

<cfinclude template="myGlobals.cfm">


<cfswitch expression = "#attributes.fuseaction#">

<cfcase value="login">
   <cfset attributes.XFA.submitForm = "validateLogin">
   <cfinclude template="dspLogin.cfm">
</cfcase>

<cfcase value="validateLogin">
   <cfset attributes.XFA.successfulLogin = "showUserMenu">
   <cfset attributes.XFA.failedLogin = "login&badLogin=True">
   <cfinclude template="actValidateUser.cfm">
</cfcase>

</cfswitch>
```

The first part of the Fusebox takes care of some housekeeping chores that I'll discuss later. For now, I want to draw your attention to the navy color-coded section, where the routing is being done.

The routing begins with a `<cfswitch>` statement that examines the value of `attributes.fuseaction`.

Why *`attributes.`*`fuseaction` you might ask?

The technical reason is that the custom tag called above, `<cf_formURL2attributes>`, converts all form and URL variables into attributes-style variables. The larger reason is--well, I'd like to defer the larger reason for a while. So far, you know that a fusebox is looking for an attributes-scoped variable called `fuseaction`.

Once it finds a matching value in one of the `<cfcase>` statements, the code between those particular `<cfcase>` tags is executed.

```
<cfcase value="login">
   <cfset attributes.XFA.submitForm = "validateLogin">
   <cfinclude template="dspLogin.cfm">
</cfcase>
```

In the Fusebox methodology, the `<cfcase>` code is used to set up an environment in which one or more fuses can be called that will perform whatever actions are needed to do the work requested by the variable `fuseaction`.

What sort of things might a fuse do? Things like display a form, see if a user's password and username match those found in a database, show a menu of user options--in short, anything a web application can do will be done through the use of fuses. Earlier, I mentioned that well-written fuses are very short, restricting themselves to doing only one or two things. Why? Well, short fuses are easier to write (and maintain), are less buggy and easier to debug, and facilitate code reuse. After all, the odds of you needing one "do-it-all" fuse in several different places are much less than those of you reusing code that only handles a single problem.

Fuses have one or more *exit points*--areas where the action will return to the fusebox. For example, clicking the submit button on a form is an exit point--when the user clicks "Submit", the action goes back to the fusebox (as you saw on a previous page). Every link on a user menu is an exit point (as each click will cause the browser to return to the fusebox), each drilldown to get more information is an exit point.

Some exit points are visible and require user interaction, such as submitting a form or clicking a link; other exit points are not visible. For example, look at this fuse that validates a user's login:

```
<cfquery datasource="#request.dsn#" name="FindUser">
   SELECT userID, firstName, lastName, userGroups
   FROM Users
   WHERE userName = '#attributes.userName#'
   AND password = '#attributes.password#'
</cfquery>

<cfif FindUser.recordcount>
   <cflocation url="index.cfm?fuseaction=#attributes.XFA.successfulValidation#">
<cfelse>
   <cflocation url="index.cfm?fuseaction=#attributes.XFA.failedValidation#">
</cfif>
```

Here, we have two different exit points, neither of which require user interaction. If the results of a query return any rows--that is, if the query found a user--the action returns to the fusebox with a fuseaction having the value of `attributes.XFA.successfulValidation`, while failure to find a user will result in returning to the fusebox with a fuseaction having the value of `attributes.XFA.failedValidation`.

Whether generated by user interaction or by the application itself, exit points in fuses are always returned to the fusebox with a fuseaction.

Can you identify the exit points in this fuse?

```
<cftry>

<cflock scope="session" timeout="10" throwontimeout="Yes">
   <cfset myCurrentUser = session.currentUser>
</cflock>

<cfcatch>
   <cflocation url="index.cfm?fuseaction=#attributes.XFA.retryUser#">
</cfcatch>

</cftry>

<cfif myCurrentUser.booksRated LT 100>
   I'm sorry, but this area is only for Premier Book Partners.
   Click <a href="index.cfm?fuseaction=#attributes.XFA.mainMenu#">
   here</a> to go back to the main menu.
<cfelse>
   Welcome to the Premier Book Partners site. Are you interested in taking a
   quick tour?
   <form
      action="index.cfm?fuseaction=#attributes.XFA.submitForm#"
      method="post">
      <input
         type="Submit"
         name="yes"
         value="Yes">
      <input
         type="Submit"
         name="no"
         value="No">
   </form>
</cfif>
```

Here's that same code--only this time I've color-coded each exit point in green.

```
<cftry>

<cflock scope="session" timeout="10" throwontimeout="Yes">
   <cfset myCurrentUser = session.currentUser>
</cflock>

<cfcatch>
   <cflocation url="index.cfm?fuseaction=#attributes.XFA.retryUser#">
</cfcatch>

</cftry>

<cfif myCurrentUser.booksRated LT 100>
   I'm sorry, but this area is only for Premier Book Partners.
   Click <a href="index.cfm?fuseaction=#attributes.XFA.mainMenu#">
   here</a> to go back to the main menu.
<cfelse>
   Welcome to the Premier Book Partners site. Are you interested in taking a
   quick tour?
   <form
      action="index.cfm?fuseaction=#attributes.XFA.submitForm#"
      method="post">
      <input
         type="Submit"
         name="yes"
         value="Yes">
      <input
         type="Submit"
         name="no"
         value="No">
   </form>
</cfif>
```

Notice that each exit point has a fuseaction associated with it that begins with an "XFA." This is shorthand for "eXit FuseAction". You could just hardcode a fuseaction at every exit point--something like this:

```
Click <a href="index.cfm?fuseaction=mainMenu">
   here</a> to go back to the main menu.
```

In fact, many Fusebox writers and coders do just that. But this impairs code reusability greatly. If you want to reuse a fuse--whether in another application or in a different point in the same one--you now must deal with the fact that your exit fuseactions may not be the same, varying according to the context in which they are used. This means that you must begin introducing conditional statements in your code:

```
<form action="index.cfm?fuseaction=<cfif attributes.fuseaction
is "newUser">insertUser<cfelseif attributes.fuseaction is
"editUser">updateUser</cfif>" method="post">
```

Now, each time you wish to reuse the fuse, you must open up the file, alter the existing code and save it, introducing the very real possibility that you will introduce a bug into the code. Further, it reduces readability (and maintainability) of the code, and makes it just plain ugly.

To me, though, the worst thing about this practice is that it requires the coder of one fuse know too much about the entire application. One of the great things about Fusebox is that it can free us developers from the "inverted pyramid" syndrome I spoke of earlier. Using XFAs lets us create fuses that will be used anywhere--without

worrying about where a form should be submitted to or a link pointed to. You create variables for these exit fuseactions and then let the application's fusebox set the values for these at run time:

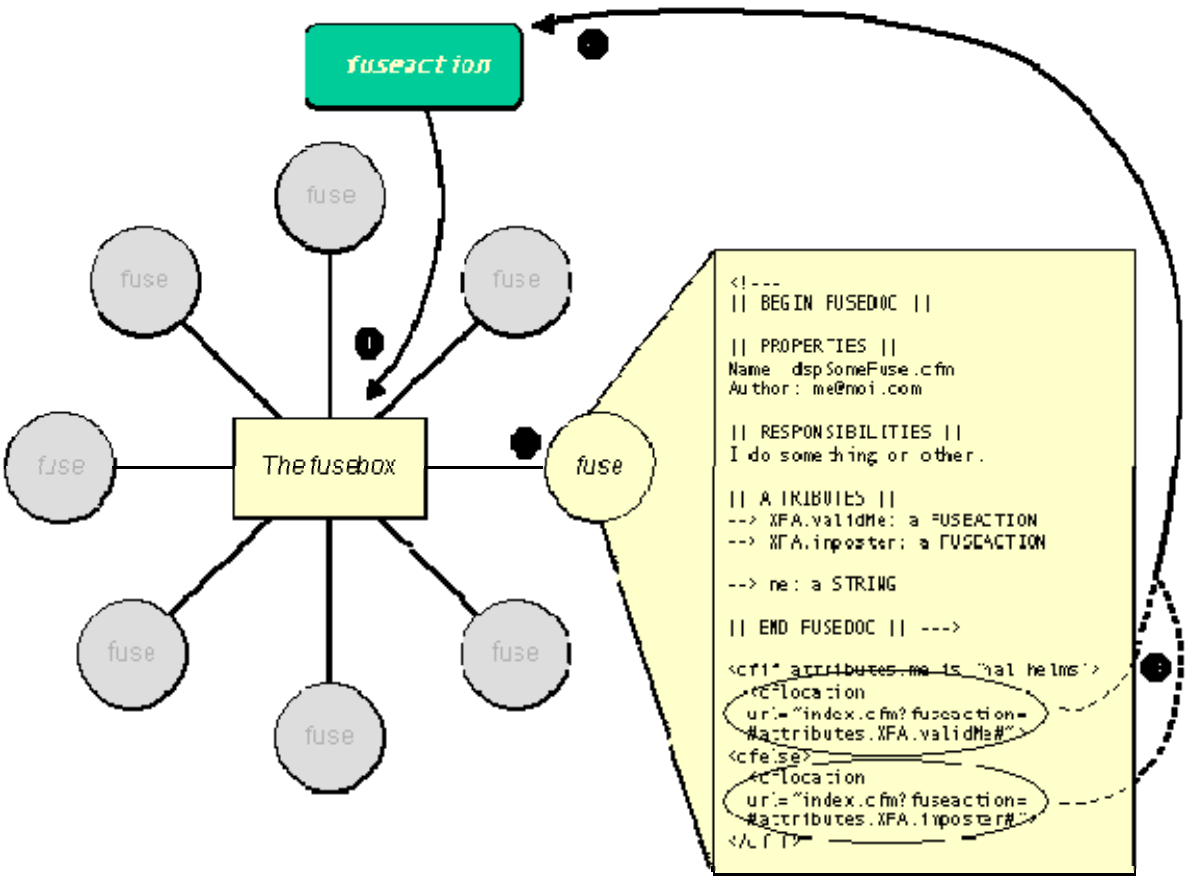
```
<cfcase value="validateLogin">
   <cfset attributes.XFA.successfulValidation = "mainMenu">
   <cfset attributes.XFA.failedValidation = "login&badLogin=True">
   <cfinclude template="actValidateUser.cfm">
</cfcase>
```

A recap of this section:

The program flow of a Fusebox application:

1. A fuseaction request is sent to the fusebox.
2. The fusebox calls one or more fuses to perform some actions related to this request.
3. The fuse(s) themselves contain exit fuseactions,
4. One of which will be sent back to the fusebox

and the process repeats itself.



It's helpful to categorize fuses into different types. For example, some fuses display information, forms, etc. to users; others work behind the scenes to do things like process credit cards; still others are responsible for querying databases. Many Fusebox developers find it helpful to use a prefix when naming a fuse that conveys the fuse type. Examples of these are:

dspShowProductInfo.cfm	a display type fuse used to show or request information from a user
act_saveUserInfo.cfm	an action type fuse used to perform an action without displaying information to a user
qryGetUserInfo.cfm	a query type fuse used to interact with data sources without displaying information to a user
url_processOrder.cfm	an action type fuse used to redirect an action???
app_Locals.cfm	an application type fuse used to set up the environment in which the application runs

These naming conventions are best thought of as suggestions: You can use the naming conventions outlined here--or not, depending on what suits you best.

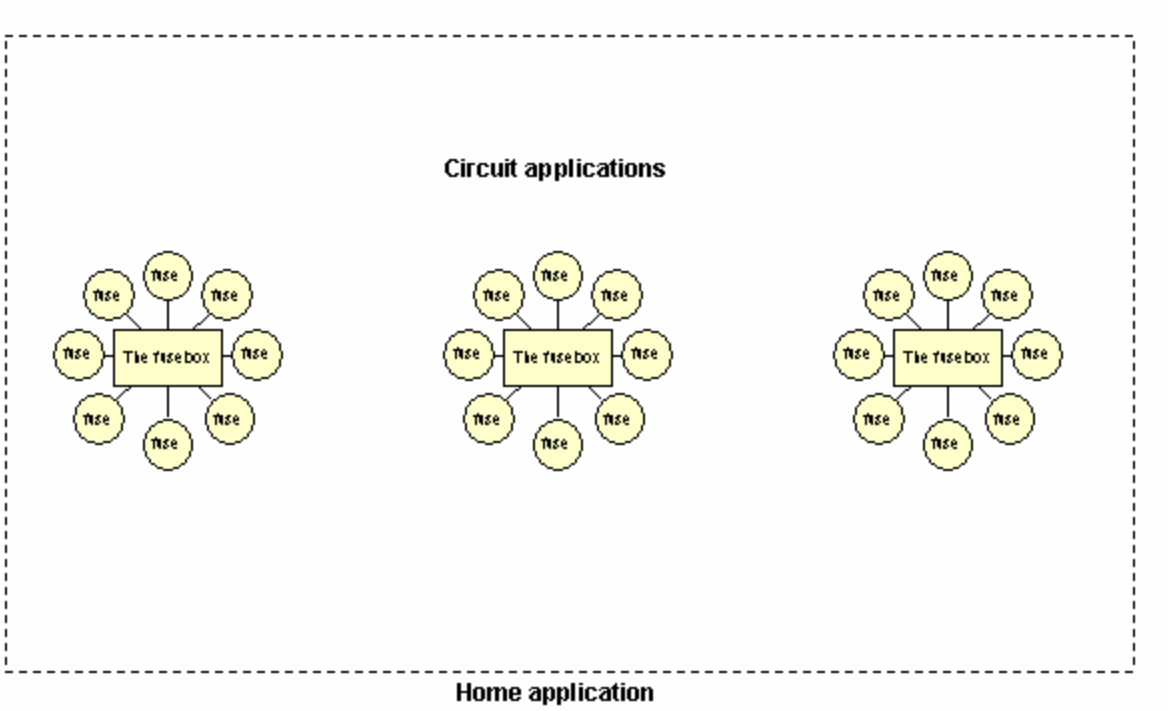
If you do use them, will you use an underscore to separate prefix from fusename (`dsp_fusename.cfm`)--or will you use the mixed case spelling that many developers prefer (`actAnotherFuseName.cfm`)?

I like the mixed case usage, but you may prefer something else. Whatever you choose, don't get bogged down in disputes over naming conventions; Fusebox is primarily about developing successful applications, not about naming schemes.

You have seen how fuseactions are returned to the fusebox, and you have had a glimpse of the mechanism the fusebox uses to call helper fuses (a `<cfswitch>` statement). Fuses really do all the work in a Fusebox application. The fusebox itself acts like a manager, delegating work to one or more of these fuses.

One of the principles of modern programming is **encapsulation**, which states that, as much as is possible and reasonable, applications should be divided into areas of related functionality. We do this constantly in many aspects of our lives. Animal bodies are made up of various systems: circulatory, respiratory, skeletal, etc.. Companies are composed of various functions: human resources, sales, production, etc..

Applying this principle to the Fusebox methodology, we get the concept of **circuit applications**. Instead one single, enormous Fusebox application (with a monolithic fusebox), we can now break a complex application into several (hopefully simpler) parts--each with their own fusebox. Together, these constitute the entire application, which is called the **home application**.



Circuit applications are groupings of fuses that, at least in the Fusebox architect's mind, makes logical and coherent sense. You commonly see circuit applications for things like a shopping cart, or an admin module, or a user module.

Each one of these circuit applications is a little self-contained Fusebox application. In order to make them work together, the original Fusebox specification directs that an `app_Locals.cfm` file should be created for each circuit application and an `app_Globals.cfm` file for the home application.

The job of these "app" files is to set up the environment in which the application will run. So, things that are application-global--an application datasource name, for example--will be set in `app_Globals.cfm` and variables specific to a circuit application are set in the `app_Locals.cfm` file.

There is only one `app_Globals` file, set in the home application's directory. Here is an example `app_Globals.cfm` file:

```
<cfapplication
   name="MyAppName"
   clientmanagement="Yes"
   clientstorage="cookie">
<cfset request.mainDSN = "MyDSN">
<cflock type="EXCLUSIVE" scope="APPLICATION" timeout="5">
   <cfif NOT IsDefined( 'application.EmployeeList' )>
      <cfinclude template="queries/qryEmployeeList.cfm">
   </cfif>
</cflock>
```

Some "environment variables" pertain only to the particular circuit application; these should be stored in the `app_Locals.cfm` file. Each circuit application can define its own `app_Locals.cfm` file.

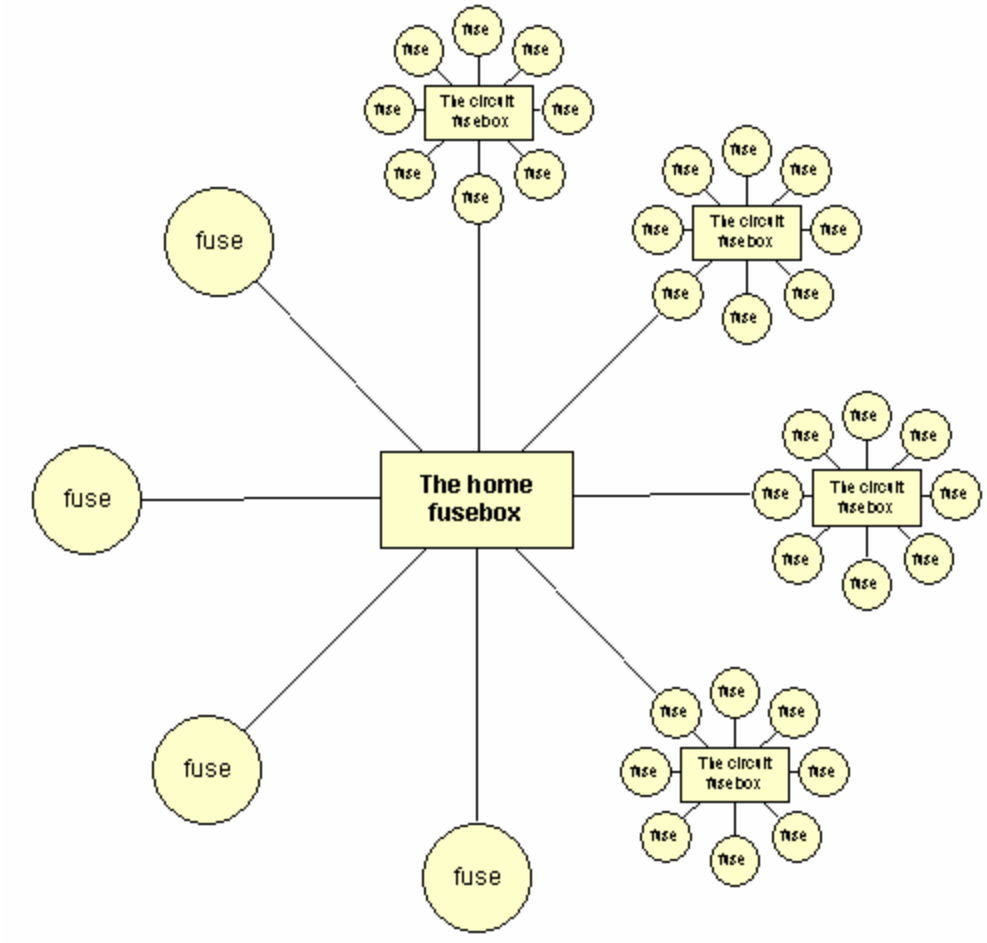
```
<cfset userDSN = "Users">
<cfset userStyleSheet = "UserStyle.css">
```

One of the great things about Fusebox is the flexibility it gives developers. This not only lets you decide on things like naming conventions, but also allows Fusebox developers to experiment with new ideas without fear of running afoul of any Fusebox police. This, in turn, lets Fusebox evolve, whether to stay abreast of technology developments or simply to incorporate good ideas that weren't originally envisioned. Different developers have different goals and bring with them different techniques. Such creative diversity can only be good for the methodology.

One idea that has proven very successful in the object-oriented world is that of **inheritance**, a mechanism for reusing code (surely the Holy Grail of many programmers). Having written a perfectly good circuit application--say a user module--for one application, we surely want to use it in others, and would like to do so without having to make wholesale changes to the code.

The original Fusebox specification doesn't make this particularly easy, so you'll have to muck about, changing the `app_Locals.cfm` and the `app_Globals.cfm`. I think this idea of inheritance can make this process both easier and safer.

Instead of having separate fuseboxes independent of each other, I use the concept of **nested fuseboxes**.



The home application itself is a fusebox that may delegate work to a fuse--or to a nested fusebox. Under this scheme, the information in `app_Globals` and `app_Locals` is combined into a single `myGlobals.cfm` file that works whether operating in stand-alone mode (as a home application) or if nested in the context of a larger application.

Code for `myGlobals.cfm` must recognize that the application may be operating as either a home application or it may have been incorporated as a circuit application. If I'm a home app, I will want to set up state management

with `<cfapplication>`, but I don't want to do so if it's already been set up. I use the presence of application.applicationName to indicate which mode `myGlobals.cfm` is in.

The logic goes like this: I am a `myGlobals` file, but I don't know if I'm the home app or if someone is using me as a circuit application. However, I do know that if `application.applicationName` already exists, it will have been set by my a preceding fusebox using the `<cfapplication name="App_Name">` tag. That means that I'm *not* the home application, and I won't set up state management with `<cfapplication>`.

Here's an example of myGlobals.cfm:

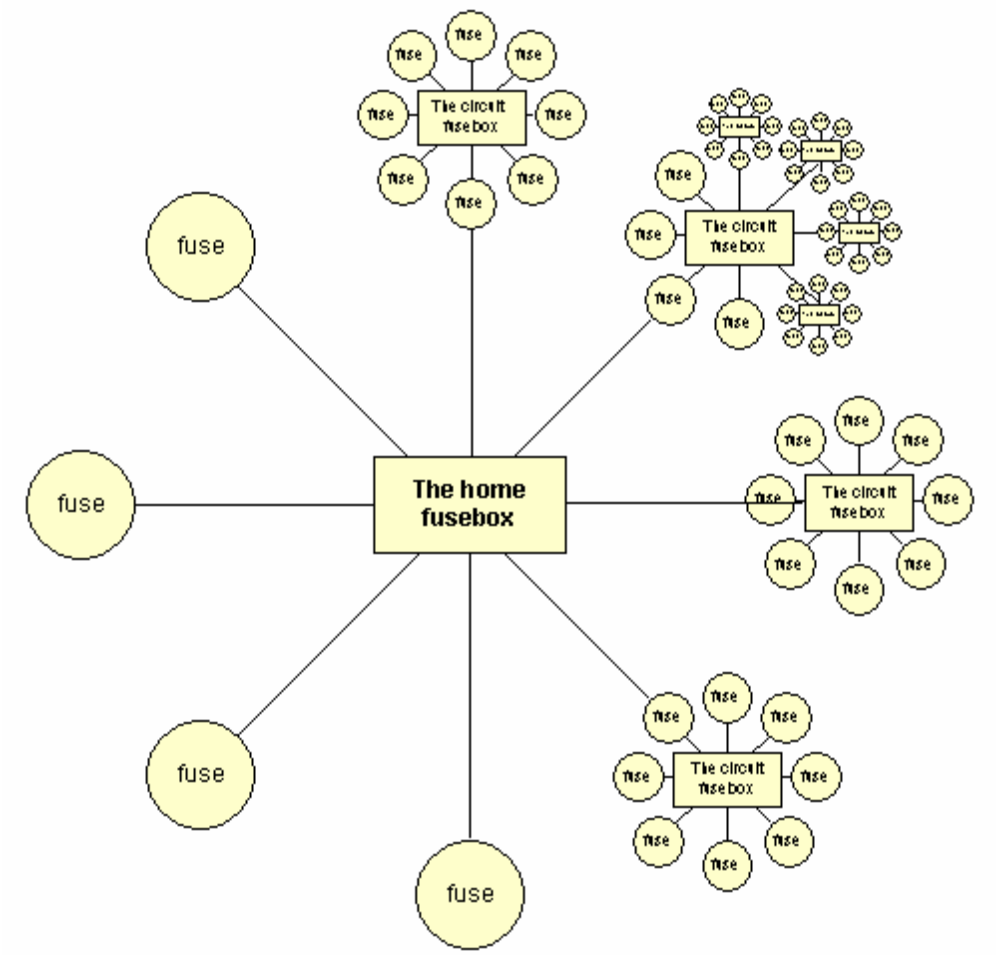
```
<cfif NOT IsDefined( 'application.applicationName' )>
   <cfapplication
      clientmanagement="Yes"
      clientstorage="cookie"
      name="MyAppName">
</cfif>
```

Other variables can be set using `<cfparam>` to make sure that variables previously set aren't unintentionally overwritten.

```
<cfparam name="request.mainDSN" default="MyDSN">
<cfparam name="request.mainStyleSheet" default="StyleA.css">
```

Each nested fusebox inherits the environment from its "parent", overwriting particular varibles if need be. That seems to me a much simpler mechanism and simplicity, for me at least, is the true Holy Grail of programming.

Having created the idea of nesting fuseboxes two levels deep, can we go deeper? Yes--and though I don't often find a need for this, I have done it in some particularly complex applications--and was glad I had the ability to do so.



There are a couple of other excellent reasons for nesting fuseboxes. One concerns security and authentication issues, in which the ability to inherit permissions aids greatly in creating a coherent security scheme. The other is in exception and error handling; there applications should have a common method for dealing with global errors and exceptions. Inheritance through nested fuseboxes lets exceptions "bubble up" to parent applications, letting circuit applications handle only those type of exceptions that are specific to the circuit. [See the lessons on Security/Authentication and Error/Exception Handling for more on these.]

With all that theory behind us, let's take a closer look at the fusebox itself--one that works with the nested fusebox scheme. Here's a skeleton I use when beginning to create the fusebox:

```
<!-- index.cfm -->

❶<!--- Part of the Fusebox method is to convert incoming form
or URL variables into attributes variables.
This custom tag does the trick. --->
<cf_formURL2attributes>

❷<!--- In case no fuseaction was given, I'll set up one s
to use by default. --->
<cfparam name="attributes.fuseaction" default="bas.1a">

❸<!--- Am I the home application? --->
<cfif NOT IsDefined( 'application.applicationName' )>
   <cfapplication clientmanagement="Yes" name="Application_Name">
</cfif>

❹<!--- I can't count on application.cfm firing in case I get
called as a custom tag, so I have to have my own way of a
"universal include". --->
<cfinclude template="myGlobals.cfm">

❺<!--- Include Circuits if I'm the home app--->
<cftry>
   <cfif NOT IsDefined( 'request.Circuits' )>
      <cfinclude template="Circuits.cfm">
   </cfif>
   <cfcatch type="MissingInclude">
      I couldn't find the file Circuits.cfm.
      <cfabort>
   </cfcatch>
</cftry>

❻<!--- Translate the fuseaction sent to me into a fully qualified
fuseaction --->
<cfparam
   name="faTranslated"
   default="FALSE"
   type="boolean">
<cftry>
   <cfif NOT faTranslated AND ListLen( attributes.fuseaction, '.' ) GT 1>
      <cfset attributes.fuseaction =
      request.circuits[ListFirst( attributes.fuseaction, '.' )] & '.'
      & ListLast( attributes.fuseaction, '.' )>
      <cfset faTranslated = TRUE>
   </cfif>
   <cfcatch>
      I could not resolve the circuit application prefix sent to me.
      <cfabort>
   </cfcatch>
</cftry>
<cfif ListLen( attributes.fuseaction, '.' ) GT 1>
   <cfset attributes.fuseaction =
   ListDeleteAt( attributes.fuseaction, 1, '.' )>
```

```
</cfif>

<!--- Peel off the first layer and determine whether to delegate
the fuseaction. --->
<cfif ListLen( attributes.fuseaction, '.' ) GT 1>
   <cfinclude template =
   "#ListFirst( attributes.fuseaction, '.' )#/index.cfm">
<cfelse>

<!--- I'm going to get one (and only one) fuseaction at a time.
Fuseactions correspond to methods or messages in OO languages. --->
   <cfswitch expression = "#attributes.fuseaction#">

   <cfdefaultcase>
   I received a fuseaction called <B><FONT COLOR="000066">
"#attributes.fuseaction#"</FONT></B> that I don't have
a handler for.
      </cfdefaultcase>

   </cfswitch>
</cfif>
```

Let's break the fusebox down...

Let's walk through the fusebox, so you can see what each part does.

```
<!--- Part of the Fusebox method is to convert incoming form
or URL variables into attributes variables.
This custom tag does the trick. --->
<cf_formURL2attributes>
```

I said earlier I wanted to defer the question about the use of attributes variables. Well, it's time to come clean. Here, we're face to face with a call to a custom tag that converts form and URL variables sent into the fusebox into attributes type variables. The question is, Why bother?

The "conventional wisdom" (if such a thing can be said to exist with so new a methdology) is that the attributes scope is needed because of the two ways fuses--and circuit apps themselves--can be called. As you know, when a fuseaction is sent to the fusebox, the fusebox uses a `<cfswitch>` statement to determine which fuses to call on. The examples I've shown you so far use `<cfinclude>s` to call the fuse:

```
<cfcase value="showSpecialPromotion">
   <cfset attributes.XFA.addItemToCart = "crt.addToCart">
   <cfset attributes.XFA.mainMenu = "bbh.home">
   <cfset attributes.productName = "Megadeveloper 2000">
   <cfinclude template="dspProductDetail.cfm">
</cfcase>
```

Although I suggest you use this method, some people may wish to call the fuse as a custom tag:

```
<cfcase value="showProductDetail">
   <cfmodule
      template="dspProductDetail.cfm"
      XFA.addItemToCart = "crt.addToCart"
      XFA.mainMenu = "home"
      productName = "Megadeveloper 2000">
</cfcase>
```

If you've worked with custom tags at all, you know that they exist in a separate memory space from the page that called them. If `PageA.cfm` calls `PageB.cfm` as a custom tag and each one sets a variable called `myVar`, both variables can exist independently. Changes to one will not affect the other. This is done deliberately to prevent a custom tag from "stepping on" the calling page's variables. If you wish to make variables available to the custom tag, you should pass them explicitly (as the example above does).

Within the custom tag, you refer to variables that have been explicitly passed in with the `attributes.` prefix. For example, if the custom tag needed to refer to the variable productName, it would do so like this:

```
Well, this month we're featuring a special on #attributes.productName#.
```

This means that a fuse (and, indeed, an entire circuit application) can be called in two modes: as an included page and as a custom tag. Now the conventional wisdom had it that if code had to work both as an include AND a custom tag, it would need to reference variables differently. I say this is conventional wisdom because it turns out not to be accurate. Custom tags can see form and URL variables just fine.

Still, I think a good argument can be made for keeping `FormURL2Attributes` for an entirely different reason, one that addresses the idea of distributed development. If you have taken the online class on Fusedocs, you're aware of how much stress I lay on the ability to divide an application into small pieces that can be distributed to a wide range of developers. Fusedoc makes this work by telling each developer what s/he is responsible for.

For example this part of Fusedoc tells the developer to pass a variable called `myVar` (which will be a Boolean value) back to the fusebox:

```
<-- myVar: a BOOLEAN
```

It does *not* tell the developer how to pass that--whether as a form or URL variable. This level of detail should be left to the coder to determine. But if we don't have a way of "homogenizing" variables into a common form, the

coder of another page that will receive these variables will need to deal with the uncertainty of what kind of variable s/he will get. Sure, there are ways of doing this, but arguably the best is letting FormURL2Attributes handle this for you.

You know that upon exiting a fuse, a fuseaction is normally sent back to the fusebox via code such as...

```
<cflocation url="index.cfm?fuseaction=#attributes.XFA.continue#">
```

or

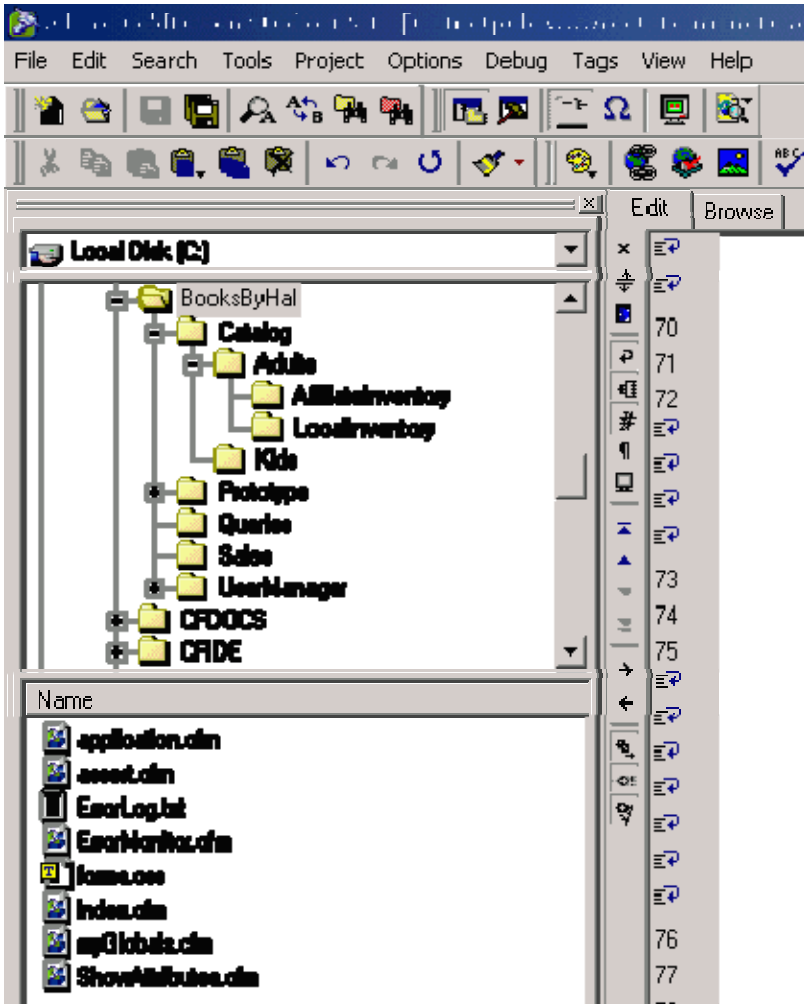
```
<form action="index.cfm" method="post">
   <input type="Hidden" name="fuseaction" value="#attributes.XFA.submitForm#">
   ...
</form>
```

This works fine while the calls to the fusebox are being set by the program itself (as these two examples are). But for users to get to your application, they need to enter a URL, and they prefer simple URLs like `www.halhelms.com`. Since you don't want the code to break, you need a create a default fuseaction for when none is specified. You could use the `<cfdefaultcase>` for this, but `<cfdefaultcase>` is so helpful when used as a debugging tool (as I'll explain at section ❾) that I instead create a default fuseaction explicitly:

```
❷<!--- In case no fuseaction was given, I'll set up one
to use by default. --->
<cfparam name="attributes.fuseaction" default="home">
```

The next several sections (❸through ❼) are used to accomodate the nested fusebox architecture I discussed earlier. This code will also work fine if you choose not to nest fuseboxes. I won't bother with what each line is doing, but I do need to explain some essentials for using nested fuseboxes.

If you choose to use nested fuseboxes, each "child" circuit should be physically located under their parent directory. Here is an example:



The `BooksByHal` home application has a child called `Catalog` that has a child called `Adults` that, in turn, has a child called `LocalInventory`.

One of the reasons I stated for preferring an inheritance scheme with Fusebox is that it allows you to start with a home application used for one purpose and then incorporate this (as a circuit application) into a larger application. In keeping with this goal, I did not want to force the name of each fuseaction for an application to be unique, as how could I guarantee that the coder of one circuit application would not use a fuseaction name that turned out to be the same as that used in another circuit?

Obviously, though, we need some way of uniquely identifying fuseactions; else, how will the home application's fusebox know that the link to `home` means the home fuseaction of the shopping cart circuit and not the home fuseaction of the user circuit? And, in keeping with the stated goal, we want to let applications transition from stand-alone to circuit status with a minimum of fuss.

We can accomplish this by defining uniqueness as the combination of circuit application *and* fuseaction. There may be many "home" (or whatever) fuseactions in a large application, but there can be only one in each circuit application, and those circuit applications must themselves have unique names.

To identify a particular fuseaction, you can reference the circuit application name and the fuseaction name. To cut down on typing, I allow for an abbreviation of circuit application names, so that Shopping Cart may become `crt` or user module becomes `usr`. These "compound" fuseaction names are then set by the fusebox as exit fuseactions. Here are the abbreviations I decided on for the BooksByHal application

Directory Name	Abbreviation
BooksByHal	bbh
Catalog	cat
Adults	adu
AffiliateInventory	afl
LocalInventory	loc
...	

Here's an example of setting exit fuseactions in the fusebox:

```
<cfcase value="displayPage">
   <cfset attributes.XFA.moreInfo = "cat.displayItem">
   <cfset attributes.XFA.checkAvailability = "loc.checkInventory">
   <cfinclude template = "dspDisplayPage.cfm">
</cfcase>
```

Using nested fuseboxes, all action returns to the *home* application--it is this fusebox that needs to know how to resolve fuseactions like `cat.displayPage`. In theory, the home app could "walk" the directory tree, trying each of its children (and its children's children and so forth) for the appropriate circuit application. This would certainly allow us the greatest flexibility, but would impose a severe performance penalty.

A good solution is to "register" each circuit application with the home application. Then, the home application can use this information directly without walking a directory tree.

Begin by creating a `Circuits.cfm` file in the directory of the home application that creates a request structure called `Circuits`. This structure should have a key for each application--both home and circuit--defining its "path":

```
<cfset request.Circuits = StructNew()>
<cfset request.Circuits.bbh = "BooksByHal">
<cfset request.Circuits.cat = "BooksByHal.Catalog">
<cfset request.Circuits.kid = "BooksByHal.Catalog.Kids">
<cfset request.Circuits.adu = "BooksByHal.Catalog.Adults">
<cfset request.Circuits.afl = "BooksByHal.Catalog.Adults.AffiliateInventory">
<cfset request.Circuits.loc = "BooksByHal.Catalog.Adults.LocalInventory">
```

The code in section ❺ is put in every fusebox. If the variable `request.Circuits` doesn't exist, it will include the `Circuits.cfm` file.

```
❺<!--- Include Circuits if I'm the home app--->
<cftry>
   <cfif NOT IsDefined( 'request.Circuits' )>
      <cfinclude template="Circuits.cfm">
   </cfif>
   <cfcatch type="MissingInclude">
      I couldn't find the file Circuits.cfm.
      <cfabort>
   </cfcatch>
</cftry>
```

The effect of all this code is somewhat like peeling an onion. The onion, in this case, is the "fully qualified" fuseaction that's been processed, replacing the circuit abbreviation with the path found in `request.Circuits`.

Section ❼ determines whether the fuseaction belongs to this fusebox ("this" being whatever fusebox we're in at the moment).

```
❼<!--- Peel off the first layer and determine whether to delegate
the fuseaction. --->
```

```
<cfif ListLen( attributes.fuseaction, '.' ) GT 1>
   <cfinclude template =
   "#ListFirst( attributes.fuseaction, '.' )#/index.cfm">
<cfelse>
```

If the fuseaction is meant for a further nested fusebox, it includes this fusebox; otherwise, it uses a `<cfswitch>` statement to call the fuse(s) needed to perform the action.

```
<!--- I'm going to get one (and only one) fuseaction at a time.
Fuseactions correspond to methods or messages in OO languages. --->
   <cfswitch expression = "#attributes.fuseaction#">
```

Finally, if no `<cfcase>` value was found for attributes.fuseaction, the fusebox tells the user the name of the fuseaction that couldn't be handled.

```
 <cfdefaultcase>
   I received a fuseaction called <B><FONT COLOR="000066">
"#attributes.fuseaction#"</FONT></B> that I don't have
a handler for.
      </cfdefaultcase>
```

I mentioned earlier that `<cfdefaultcase>` is a very useful tool for debugging; it will alert you if your fusebox is getting a fuseaction that you weren't expecting. This helps eliminate those little errors where you typed `showProductInfo` instead of `displayProductInfo`, or `editUsrInfo` when you meant `editUserInfo`. At production, you may want to eliminate this, as all errors should be handled by that point.

Just a few last remarks and then I offer a complete (albeit small) Fusebox program that you can download and examine.

I try to avoid hardcoding values that can change--most programmers would agree that's just good practice. One of the things that can change is the name of the fusebox file itself. In all of the examples, I've been hardcoding `index.cfm`. But you may be writing in an environment where some other page is served up by default--perhaps `default.cfm`.

Rather than going back and doing global search and replace (the very thought of which chills my cowardly soul), I use the variable, `self`, as an alias for the fusebox. This value is set in `myGlobals.cfm`, making for links that look like this:

```
<a href="#self#?fuseaction=#attributes.XFA.someFuseaction#">Click me</a>
```

On another topic, I recommend that you take a look at the specification for Fusedoc. This is really a hybrid "documentation-program definition language" that has really revolutionized the way I write code. There's a free lesson on Fusedocs, so whatcha waitin' for?

The code for this program, SimpleContactManager, is meant to help you understand Fusebox architecture. It's certainly not production-ready code--for one thing, there's no exception handling--but I wanted to keep it as lean as possible so that you can quickly work your way through the code to the underlying architecture.

You can set this code to run on your own system by creating an ODBC datasource called "SimpleContactManager" and pointing it at the Access 2000 DB included in the zip file.

Click here to download the zipped Fusebox application.

Mastering FuseDoc

Written by: Hal Helms
http://halhelms.com/OnlineCourses/Fusedocs/index.cfm?fuseaction=doc.pedagogy

Fusedoc is an emerging standard for documenting ColdFusion code pages. Unlike much documentation that is written while coding, Fusedoc is meant to be written prior to coding. If done correctly, it creates a blueprint for a fuse. (If you are familiar with the concept of PDLs (program definition languages), there are similarities between Fusedoc and PDLs.)

One of the big questions in documentation is sometimes known as the "banana" problem, as in the little girl who said, "I know how to spell banana; I just don't know when to stop." When do we reach "enough" documentation? Exactly what should it include?

Mark Twain once remarked that the most frightening thing he knew of was a blank piece of paper. Many of us can relate to this with regards to documentation. When I wrote the specifications for Fusedoc, I had this in mind. I wanted to offer help in what documentation should be included as well as provide a format that would make it easier.

So, how do we know when enough is enough? A properly written Fusedoc should tell a competent ColdFusion programmer everything s/he needs to write the fuse without any knowledge of either the application the fuse belongs to or the underlying database.

That said, let me offer my own experiences with Fusedoc. I spend a good deal of time architecting ColdFusion applications using Fusebox. I work with developers in creating a framework for the application and deal with issues such as scalability and reuse of code. I help companies determine the circuit applications and the individual fuseactions that make up an application as well as working with them on the underlying data schema.

I felt it was critical that I offer clients a way to move from the architecting phase to the development phase without needing me there to manage the project. I have worked with some of the big consulting firms whose business model I sometimes refer to as the "Barnacle Plan". Under this plan, a consultant (or more likely, several consultants) come in to a company and begin billing at very high rates. Their stated goal is to help the project succeed, but after working with them for a while, I began to suspect that their real goal was to affix themselves to the project for as long as possible, like...well, barnacles!

That's a very old model and one I didn't have much interest in replicating. Besides, companies didn't need that; They usually have an abundance of talent who only need some help and direction. I needed a way to communicate fully to these developers so that they could take the documentation I handed them and start coding without having to attempt divination of pigeon entrails as a means of understanding what the client wanted.

I tried several things. I wrote lots of words. I got lots of questions. I drew diagrams. I got lots more questions. One of the big problems was that each developer needed to get up to speed on the entire project. That was really inefficient and pretty unsuccessful. That encylopedic knowledge of the project is usually the hardest part, and a system that meant that people couldn't contribute without such an understanding seemed pretty flawed to me.

What if coders were responsible for just one thing--coding? Not having to make decisions about what the client probably wanted. Not having to figure out how their code would impact code someone else was writing. Not having to re-write code when many of these guesses inevitably turned out to be wrong. That seemed like a good idea. What if I thought of the coders as working on the other side of an impenetrable wall? I could slip paper to them and they could slip paper back to me, but that was it for communication. That seemed to me like a pretty great goal. If I could figure out a way, it would mean they could do their job and be successful at it instead of wasting time.

I asked myself what a ColdFusion coder--someone good, but not an expert--would need to write a fuse. From that, I developed the basic skeleton of a Fusedoc. I tried it out. I made improvements. I added, deleted, changed it in response to what real coders told me they needed and what was superfluous. I then implemented the information as a ColdFusion comment and I placed it as the first thing that appears on the code page.

Here's what I came up with as the Fusedoc skeleton:

```
<!---
|| BEGIN FUSEDOC ||

|| Properties ||
Name: fusename.cfm
Author: hal.helms@TeamAllaire.com

|| Responsibilities ||


|| Attributes ||


|| END FUSEDOC ||--->
```

It looks simple--and it is, but here's where my story turns ugly. I tried actually writing Fusedocs for a project. Instead of things going without a hitch and making me the hero of my own story, the coders sent them back almost immediately. "You left out some stuff," they said. It turned out they were being very gentle in their appraisal. These clearly were not--could not be--the Fusedocs I sent to the coders. What I sent them were elegant, beautifully-crafted gems of lucidity; What I got back were the hideous, malformed, incoherent products of an obviously deeply-confused mind.

I'm joking now, but I didn't feel very funny then. It was deeply disturbing. How could I have left out so much stuff? Was my thinking really that incoherent? And I was trying really hard to be clear! I tell this story on myself so that if, as you begin, you run into the same problems, you'll at least know that this is ground that others have trod. And there's good news: the second version was much better than the first, though still far from perfect. The third version was better yet. I actually started getting code back--and it worked! What a thrill!

As I have taught others about Fusedocs, I've learned that this turns out to be a common experience. The one ingredient needed is perseverance to get through this shadowy land and out into the sun. The developers who have persevered discover that learning to write well-crafted Fusedocs has a staggering impact on the quality, scalability, and reusability of their code.

What if you're the only developer--you don't have any coders you're slipping Fusedocs to? You will still benefit enormously from writing Fusedocs for your code. After all, you're going to have to make these decisions and figure out the issues Fusedocs bring up. The real question is: When do you want to deal with them? During the architecture phase or while you're in the midst of coding? I can absolutely tell you that, just as with a building, changes made during the architectural phase are far less expensive and easier than those done during construction.

Convinced? Then, place one hand on the monitor and vow that you won't give up until your Fusedocs sparkle. Now, let's get on with it!

The Fusedoc has at least three sections: **Properties**, **Responsibilities**, and **Attributes**. In order to provide flexibility and to fit in with coding standards that you may already have in place, other sections, may be added if needed. This could be handy if, for instance, you wanted a section detailing a fuse's history.

```
<!---
|| BEGIN FUSEDOC ||

|| Properties ||
Name: fusename.cfm
Author: hal.helms@TeamAllaire.com

|| Responsibilities ||


|| Attributes ||


|| END FUSEDOC ||--->
```

Only one of these Fusedocs adheres to the standard. Which one?

FUSEDOC 1

```
<cf_FUSEDOC>

|| Properties ||
Name: dspLogin.cfm
Author: hal.helms@TeamAllaire.com

|| Responsibilities ||
I provide a login for a user to enter their username and password and
I pass it on to the fusebox with an XFA of submitForm. But if I got a
variable called "badLogin", I let the user know their first login failed.


|| Attributes ||
--> XFA.submitForm: a FUSEACTION
--> XFA.cancelForm: a FUSEACTION
--> [badLogin]: a BOOLEAN

<-- userName: a STRING
<-- password: a STRING

</cf_FUSEDOC>
```

FUSEDOC 2

```
<!---
|| BEGIN FUSEDOC ||

|| Properties ||
Name: dspLogin.cfm
Author: hal.helms@TeamAllaire.com

|| Responsibilities ||
I provide a login for a user to enter their username and password and
I pass it on to the fusebox with an XFA of submitForm. But if I got a
variable called "badLogin", I let the user know their first login failed.

|| History ||
2.15.99 -- created
6.11.99 -- added in a badLogin notification

|| Attributes ||
--> XFA.submitForm: a FUSEACTION
--> XFA.cancelForm: a FUSEACTION
--> [badLogin]: a BOOLEAN

<-- userName: a STRING
<-- password: a STRING


|| END FUSEDOC ||--->
```

If you picked the second sample code the you were right:

```
<!---
|| BEGIN FUSEDOC ||

|| Properties ||
Name: dspLogin.cfm
Author: hal.helms@TeamAllaire.com

|| Responsibilities ||
I provide a login for a user to enter their username and password and
I pass it on to the fusebox with an XFA of submitForm. But if I got a
variable called "badLogin", I let the user know their first login failed.

|| History ||
2.15.99 -- created
6.11.99 -- added in a badLogin notification

|| Attributes ||
--> XFA.submitForm: a FUSEACTION
--> XFA.cancelForm: a FUSEACTION
--> [badLogin]: a BOOLEAN

<-- userName: a STRING
<-- password: a STRING


|| END FUSEDOC ||--->
```

The problem with the first sample is that there is no custom tag involved in Fusedoc.

Remember that you can add in sections if you need to. In this Fusedoc, the section "History" was added.

Responsibilities are written in the first person, as if the fuse were alive and explaining itself. In a large-scale environment, writing the Fusedoc is done by the application architect, while writing the fuse itself is done by a ColdFusion programmer. Fusedocs are written prior to writing the fuse. Consequently, Responsibilities tell **what** a fuse should do, but usually not **how** it will do it.

Which of these Responsibilities best express the job of a fuse that will receive a comma-delimited list of productIDs in a variable called "selectedProducts" and turn these into a session 1-D array called "selectedProducts"?

Option 1

|| Responsibilities ||
I receive a comma-delimited list of values in a variable called "selectedProducts". I then <cfloop> over the list and read each, placing each one into a new array (using ArrayNew()) called "session.selectedProducts". I use <cflock> to single-thread the write and prevent possible data corruption.

Option 2

```
|| Responsibilities ||
I get a comma-delimited list called "selectedProducts". I turn this list
into a 1-D array called "selectedProducts", which is a session variable.
```

Option 3

```
|| Responsibilities ||
I get a normal list and turn this into a new array, using <cfloop>.
```

If You picked the second sample code then:

|| Responsibilities ||
I get a comma-delimited list called "selectedProducts". I turn this list
into a 1-D array called "selectedProducts", which is a session variable.

That is correct. Here's the first sample code:

|| Responsibilities ||
I receive a comma-delimited list of values in a variable called
"selectedProducts". I then <cfloop> over the list and read each,
placing each one into a new array (using ArrayNew()) called
"session.selectedProducts". I use <cflock> to single-thread the
write and prevent possible data corruption.

It's perfectly proper for a Fusedoc to say "turn this list into a 1-D array", but, how that's done (the <cfloop> and the <cflock>) should be left to the coder, as they will have a much better grasp on the implementation details than the architect. Even in environments where the architect and the coder are the same, it's best to leave such details out of the Fusedoc. This makes it easier to read the Responsibilities section of a fuse and quickly grasp what the fuse does.

At the same time, the third sample code:

|| Responsibilities ||
I get a normal list and turn this into a new array, using <cfloop>.

fails because it doesn't contain enough information (what's the name of the list; what's the name of the new array) while it provides unneeded instruction (use the <cfloop>). The best code sample is the second one:

|| Responsibilities ||
I get a comma-delimited list called "selectedProducts". I turn this list
into a 1-D array called "selectedProducts", which is a session variable.

Try another one: Study this code and the two candidates for Fusedoc for it. Which is the best?

```
<cfmodule
        template="/BooksByHal/Queries/qryValidateLogin.cfm"
        password="#Encrypt( attributes.password, Hash( LCase( attributes.userName ) ) )#">

<cfif ValidateLogin.recordCount>
        <cfset aStruct = StructNew()>
        <cfset aStruct.userID = ValidateLogin.userID>
        <cfset aStruct.firstName = ValidateLogin.firstName>
        <cfset aStruct.lastName = ValidateLogin.lastName>
        <cfset aStruct.userGroups = ValidateLogin.userGroups>
        <cfwddx action="CFML2WDDX" input="#aStruct#" output="client.currentUser">
        <cflocation url="#self#?fuseaction=#attributes.XFA.successfulValidation#">
<cfelse>
        <cflocation url="#self#?fuseaction=#attributes.XFA.failedValidation#">
</cfif>
```

```
<!---
|| BEGIN FUSEDOC ||

|| Properties ||
Name: actValidateLogin.cfm
Author: hal.helms@TeamAllaire.com

|| Responsibilities ||
I get a userName and password sent to me. I then hash the lower-cased userName
and use this as a key to encrypt the password. I send this to the query
qryValidateLogin. If I get a row back, I'm good to go and I create a WDDX'd
client var called "currentUser" from a structure consisting of userID,
firstName, lastName, userGroups from the query.


|| Attributes ||
--> XFA.successfulValidation: a FUSEACTION
--> XFA.failedValidation: a FUSEACTION
--> userName: a STRING
--> password: a STRING

<++ [currentUser]: a WDDX CLIENT structure:
        < userID: a PRIMARY KEY from Users table
        < firstName: a STRING
        < lastName: a STRING
        < userGroups: an INTEGER

+++ qryValidateLogin.cfm
        > password
        < userID: a PRIMARY KEY from Users table
        < firstName: a STRING
        < lastName: a STRING
        < userGroups: an INTEGER

|| END FUSEDOC ||--->
```

Second Fusedoc on next page.

```
<!---
|| BEGIN FUSEDOC ||

|| Properties ||
Name: actValidateLogin.cfm
Author: hal.helms@TeamAllaire.com

|| Responsibilities ||
I get a userName and password sent to me. I send this to the query
qryValidateLogin. If I get a row back, I'm good to go and I create
a client var called "currentUser" from a structure consisting of
userID, firstName, lastName, userGroups from the query.


|| Attributes ||
--> XFA.successfulValidation: a FUSEACTION
--> XFA.failedValidation: a FUSEACTION
--> userName: a STRING
--> password: a STRING

<++ [currentUser]: a WDDX CLIENT structure:
        < userID: a PRIMARY KEY from Users table
        < firstName: a STRING
        < lastName: a STRING
        < userGroups: an INTEGER

+++ qryValidateLogin.cfm
        > password
        < userID: a PRIMARY KEY from Users table
        < firstName: a STRING
        < lastName: a STRING
        < userGroups: an INTEGER

|| END FUSEDOC ||--->
```

Which FuseDoc is the best? Find out on the next page.

If Your choice was the first sample the you were right, but understanding why is important. Development occurs in several phases. One of these is the architectural phase, where we're deciding on the structures that will let the application work well, expand gracefully, and be maintained easily.

Another phase is coding--and in that phase, you want all the energy focused on tight, clean implementation of decisions already made in the architecting phase. This is undermined if the coder has to stop to figure out how various parts should work together.

In this example, there's no way for the coder to know that s/he needs to hash the lowercase userName and use this as a key to encrypt the password, so the Responsibilities section should convey this as clearly and simply as possible.

Similarly, the coder will not know to take the userID, firstName, lastName, and userGroups from the query and turn these into a structure and then to turn this structure into a WDDX packet that should be saved as a client variable.

These Responsibilities (though wordier than I prefer) are needed to give the coder necessary information.

```
<!---
|| BEGIN FUSEDOC ||

|| Properties ||
Name: actValidateLogin.cfm
Author: hal.helms@TeamAllaire.com

|| Responsibilities ||
I get a userName and password sent to me. I then hash the lower-cased userName
and use this as a key to encrypt the password. I send this to the query
qryValidateLogin. If I get a row back, I'm good to go and I create a WDDX'd
client var called "currentUser" from a structure consisting of userID,
firstName, lastName, userGroups from the query.


|| Attributes ||
--> XFA.successfulValidation: a FUSEACTION
--> XFA.failedValidation: a FUSEACTION
--> userName: a STRING
--> password: a STRING

<++ [currentUser]: a WDDX CLIENT structure:
            < userID: a PRIMARY KEY from Users table
            < firstName: a STRING
            < lastName: a STRING
            < userGroups: an INTEGER

+++ qryValidateLogin.cfm
            > password
            < userID: a PRIMARY KEY from Users table
            < firstName: a STRING
            < lastName: a STRING
            < userGroups: an INTEGER

|| END FUSEDOC ||--->
```

Although the properties section of the fusedoc has a set format, the actual properties being set depend on your preferences and needs. The format for properties should be

property_name:*property_contents*

|| Properties ||
Name: actValidateLogin.cfm
Author: hal.helms@TeamAllaire.com
Created On: 2.25.99
Last Edited: 7.25.00
Which of these are valid Properties?

|| Properties ||

```
Name: #attributes.myName#
Last updated on 5.15.99
xxxx: yyyzzzz
level: 2
turn-ons: long walks on the beach, piles of cash
```

Answers to "Which of these are valid Properties?"

Name: #attributes.myName#

Verdict: Iffy. Technically valid, but since Fusedocs are meant to give guidance at design time, having variables that only are resolved at run time violates the idea behind Fusedoc.

Last updated on 5.15.99

Verdict: Not valid. A valid property has a name and a value, separated by a colon. Making this valid would be easy:

Last update: 5.15.99

xxxx: yyyzzzz

Verdict: Valid. A little weird, admittedly, but it conforms to the specification.

level: 2

Verdict: Valid

turn-ons: long walks on the beach, piles of cash

Verdict: Valid

A recap of this section:

- Fusedoc is a method for providing comments in Fusebox code files.
- Fusedocs have at least three sections: Properties, Responsibilites, and Attributes, which can be added to if you wish.
- Fusedocs are written prior to writing the fuse , and create a blueprint for coding the fuse.
- Fusedocs should tell the coder *what* the fuse should do, rather than *how* it should be done.
- The properties section is very flexible, specifying only that entries in this section should appear in the form *property_name: property_contents*.

Learning FuseDoc Examples

The final section, Attributes, is the most complex one, though it looks worse than it is.

Rather than give you more stuff for you to remember, let me see if I can make it clear by example.

|| Attributes ||
--> userName: a STRING
--> password: a STRING

indicates that two variables are being explictly passed into the fuse as "attributes" type variables and it gives a little information about what kind of data the variable will hold--strings in this case.

Wherever the type is not specified, you can count on it being an attributes-scoped variable.

Learning Fusedoc

More examples:

|| Attributes ||
--> [badLogin]: a BOOLEAN

indicates that the variable "badLogin" *may* be passed into the fuse. The square brackets indicate that this variable is optional. If it is present, though, it will be a BOOLEAN value.

Which of these are valid?
--> myVar: a variable
--> variable1: an INTEGER
--> selectedProducts: a LIST
`--> substitute SKU: an INTEGER`**Verdict: Not valid.** But almost; the problem is that "a variable" isn't information about the data type. Admittedly, I'm being a little picky...

--> variable1: an INTEGER

Verdict: Valid. Sure, "variable1" is a pretty poor choice for a name, as it doesn't convey any information about what the variable stands for, but syntactically, it's OK.

--> selectedProducts: a LIST

Verdict: Valid.

--> substitute SKU: an INTEGER

Verdict: Not valid. It's a trick question; you can't have a variable named "substitute SKU" -- no spaces are allowed in variable names.

More examples:

|| Attributes ||
++> userID: a SESSION INTEGER PRIMARY KEY from Users table
++> currentUser: a CLIENT WDDX STRUCTURE
  < lastName: a STRING
  < firstName: a STRING
  < userGroups: an INTEGER

The ++> indicates that the variable following is a global variable. This would include session, client, application, server, and request variables.

When an "arrow" appears in a symbol (such as --**>** or **<**++), the direction of the arrow tells us which "direction" the variables are being passed; whether *to* or *from* the particular structure, array, query, or template that is involved.

Arrows pointing to the left (<) indicate information that is coming from;
arrows pointing to the right (>) indicate information being sent to.
Examples:

<-- myVar: a STRING

indicates a variable passed FROM or out of the fuse. The fuse is responsible for setting this as a form or URL variable or, in the case of a custom tag, a "caller"-type variable.

--> myVar: a STRING

indicates a variable being sent TO the fuse.

++> myStructure
  < name: a STRING
  < address: a STRING

indicates that name and address are coming FROM the structure--in other words, they are part of the structure.

<++ myQuery
  > userID: an INTEGER
  < firstName: a STRING
  < lastName: a STRING

indicates that userID is sent TO the query and firstName and lastName are returned FROM the query.

There is also a client variable called "currentUser" that is shown. The small indented symbols
< lastName: a STRING
< firstName: a STRING
< userGroups: an INTEGER

indicate that the structure will have three key/value pairs.

Which one of these conveys that I may be receiving an application variable that is a query and that returns courseID, courseName, location?

Option 1

|| Attributes ||
++> courses: a WDDX QUERY (courseID, courseName, location)

Option 2

```
|| Attributes ||
++> [courses]: an APPLICATION QUERY
       < courseID
       < courseName
       < location
```

Option 3

```
|| Attributes ||
++> courses?: an APPLICATION QUERY
       < courseID
       < courseName
       < location
```

Well, the first one is out.

|| Attributes ||
++> courses: a WDDX QUERY (courseID, courseName, location)

There was nothing said about the query being WDDXd. Also, the directions said you MAY be receiving a variable, indicating that this variable is optional. In the first sample Fusedoc, there's nothing about courses being optional.

The second one is right. That's the one you picked.

|| Attributes ||
++> [courses]: an APPLICATION QUERY
< courseID
< courseName
< location

The square brackets indicate the variable is optional and everything else is fine.

Surprisingly, the third code sample is OK, too.

|| Attributes ||
++> courses?: an APPLICATION QUERY
< courseID
< courseName
< location

Fusedoc also allows for the use of these symbols indicating quantities:

? means 0 or 1 (optional)
+ means 1 or more
* means 0 or more

This means that these two code snippets mean the same thing:

--> [myVar]: a STRING
--> myVar?: a STRING

Consider the following snippet from a valid Fusedoc. What can be said about it? (Note: this Fusedoc uses some syntax not yet introduced. Feel free to try your hand at discerning the meaning of these new symbols.)

```
|| Attributes ||
--> XFA.submitForm: a FUSEACTION
--> searchCriteria: a STRING
--> ISBN: a STRING

++> request.DSN: an ODBC DATASOURCE
++> userSettings?: a SESSION ARRAY where column 1 is colors,
column 2 is backgrounds, column 3 is fonts

<-- [bookFound]: a BOOLEAN ( TRUE )

<++ book: a WDDX CLIENT STRUCTURE of
        < ISBN: a PRIMARY KEY from Books table
        < title: a STRING
        < author: a STRING

+++ qryGetBookInfo.cfm: a QUERY file to be included
        > ISBN: a STRING
        < title: a STRING
        < author: a STRING
```

- There are five variables being explictly passed into the fuse.
- The variable "userSettings" is an optional variable.
- The variable "DSN" is an APPLICATION-scope variable.
- The variable "bookFound" is an optional variable that may be passed explictly OUT OF the fuse as a form or URL variable.
- If the variable "bookFound" is passed explictly from the fuse, it will have a BOOLEAN value of TRUE.
- The variable "book" will either be passed into the fuse explicitly or it will be available to the fuse as a client variable.
- When the variable "book" is written, it will be a structure with three key/value pairs.
- qryGetBookInfo.cfm is a custom tag called by the fuse.
- The file qryGetBookInfo.cfm expects an attributes-style variable called "ISBN".
- The file qryGetBookInfo.cfm will return the following fields: ISBN, title, author

This one was a hard one. There was lots of stuff in there that hadn't been discussed at all. Still, let's see how you did...

Here are the answers:

There are five variables being explictly passed into the fuse.
Verdict: False. There are three variables being explictly passed in (XFA.submitForm, searchCriteria, and ISBN) and two variables that are global (request.DSN and session.userSettings).

The variable "userSettings" is an optional variable.
Verdict: True. ? means 0 or 1 (optional), + means 1 or more and * means 0 or more.

The variable "DSN" is an APPLICATION-scope variable.
Verdict: False. It's a request-scope variable, not an application-scope one.

The variable "bookFound" is an optional variable that may be passed explictly OUT OF the fuse as a form or URL variable.
Verdict: True. The square brackets indicate it's optional and the only way for a variable to be explicitly passed out of a fuse is by a URL or form variable.

If the variable "bookFound" is passed explictly from the variable, it will have a BOOLEAN value of TRUE.
Verdict: True. When parentheses are used in attributes, it usually indicates a default value.

The variable "book" will either be passed into the fuse explicitly or it will be available to the fuse as a client variable.
Verdict: False. The <++ symbol indicates that the fuse will set a global variable. In this case, the global variable will be a WDDXd client variable that is a structure.

When the variable "book" is written, it will be a structure with three key/value pairs.
Verdict: True.

---

qryGetBookInfo.cfm is a custom tag called by the fuse.
Verdict: False. qryGetBookInfo.cfm is a file that will be <cfinclude>d. The +++ symbol means that this file is required for the fuse to operate, but if it were to be called as a custom tag, the Fusedoc should have indicated so.

The file qryGetBookInfo.cfm expects an attributes-style variable called "ISBN".
Verdict: True. Unless otherwise noted, all variables are presumed to be attributes-style variables.

The file qryGetBookInfo.cfm will return the following fields: ISBN, title, author
Verdict: False. This formulation...

> ISBN: a STRING
< title: a STRING
< author: a STRING

indicates that ISBN will be available to the query file and that the query will return two fields: title and author.

A recap of this section:

- --> indicates a variable passed explicitly into the fuse
- <-- indicates a variable passed explicitly out of the fuse
- <-> indicates a variable that will be passed explicitly into and out of the fuse without change
- ++> indicates a global variable available to the fuse
- <++ indicates a global variable that is to be set by the fuse
- +++ indicates a file required by the fuse
- [] indicates an optional element
- ? indicates an element will appear 0 or 1 times
- + indicates an element will appear 1 or more times
- * indicates an element will appear 0 or more times
- Small arrows (< or >) are used to indicate direction of "internal parts" of a variable, such as a structure's keys or an array's columns.

Learning FuseDoc Questions

Ok now lets go through a couple exercises to test your knowledge.

Select all options for each question that are true.

Question 1

The example:

```
<-- myVar: a LIST
```

indicates...
A. a request variable globally present
B. a variable explicitly passed out of this fuse
C. a variable explicitly passed into this fuse

Question 2

The example :

```
<++ application.myVar: a QUERY result set
        < userID
        < firstName
        < lastName
        < userGroups
```

indicates...
A. an application variable globally available to this fuse
B. an application structure with four key/value pairs
C. an application query explictly passed out of this fuse
D. an application query set by this fuse

Question 3:

```
+++ CreditCardAuthorization.cfm: a custom tag
        > saleID
        > saleAmount
        > userZipcode
        < approvalStatus
        < approvalCode
```

indicates...
A. a global variable available to this fuse
B. a ColdFusion file that can be called as cf_CreditCardAuthorization or with <cfmodule>
C. a variable explicitly passed into the fuse
D. a required file for this fuse

Answers to Multiple Choice Questions

Question 1

The example was:

```
<-- myVar: a LIST
```

indicates...
A. a request variable globally present
B. a variable explicitly passed out of this fuse
C. a variable explicitly passed into this fuse
D. a required file for this fuse **The correct answer(s) is B**

Question 2

The example was:

```
<++ application.myVar: a QUERY result set
        < userID
        < firstName
        < lastName
        < userGroups
```

indicates...

You chose...
A. an application variable globally available to this fuse
B. an application structure with four key/value pairs
C. an application query explictly passed out of this fuse
D. an application query set by this fuse

The correct answer is D

Also a little picky; application variables can't be explicitly passed out of a fuse as a form or URL variable are; they are instead **set** by the fuse.

Question 3

The example was:

```
+++ CreditCardAuthorization.cfm: a custom tag
        > saleID
        > saleAmount
        > userZipcode
        < approvalStatus
        < approvalCode
```

indicates...

You chose...
A. a global variable available to this fuse
B. a ColdFusion file that can be called as cf_CreditCardAuthorization or with <cfmodule>
C. a variable explicitly passed into the fuse
D. a required file for this fuse

The correct answer is B & D

Is this a valid Fusedoc?

```
<!---
|| BEGIN FUSEDOC ||

|| Properties ||
Name: actDecrementBook.cfm
Author: hal.helms@TeamAllaire.com

|| Responsibilities ||
I receive an ISBN, strip out any weird characters and check that
it's 10 characters long. If not, I throw an error of custom type "BadISBN".
Then I call qryDecrementBook.cfm and go on my way to XFA.continue.


|| Attributes ||
--> XFA.continue: a FUSEACTION
--> ISBN: a STRING

+++ qryDecrementBook.cfm: a custom tag
        > ISBN: a STRING

|| END FUSEDOC ||--->
```

Verdict: Valid

"XFA.continue" and "ISBN" are variables explictly passed into the fuse, and "qryDecrementBook.cfm" is a required file that will be called as a custom tag. When it is called, it expects the variable "ISBN" to be passed to it.

Is this a valid Fusedoc?

```
<!---
|| BEGIN FUSEDOC ||

|| Properties ||
Name: fncSortMemberSearchResults.cfm
Author: hal.helms@TeamAllaire.com

|| Responsibilities ||
I receive an array of structures where each structure has a key/value
pair called "price". I pick the three structures with the lowest price
and return these in their own array of structures.

|| Attributes ||
--> resultsArray: an ARRAY of STRUCTURES

<-- SortMemberSearchResults: a CALLER ARRAY of STRUCTURES

|| END FUSEDOC ||--->
```

Verdict: Valid

This is a custom tag's Fusedoc. "resultsArray" will be passed explicitly into it and it will pass out "SortMemberSearchResults". Note the use of the word "CALLER" indicating that this is a custom tag that will be setting a variable on the calling page. This conforms to the rules for creating Fusefunctions.

Is this a valid Fusedoc?

```
<!---
|| BEGIN FUSEDOC ||

|| Properties ||
Name: qryAllPermissions.cfm
Author: hal.helms@TeamAllaire.com

|| Responsibilities ||
I return all the permission names and values


|| Attributes ||
<-- permissionName: a STRING
<-- permissionValue: an INTEGER

|| END FUSEDOC ||--->
```

Verdict: Valid

This is the Fusedoc for a query file. You would normally use a query sim initially here and later replace it with an actual <cfquery> or <cfstoredproc> tag.

Now, it's time for you to try your hand at writing some Fusedocs.

Please reference the complete list of attribute symbols within the appendix to assist in writing your fusedocs.

How would you indicate the following descriptions as attributes within a fusedoc?

Responsibility 1:

..a variable, an integer called "myVar" being passed explicitly into the fuse.

Responsibility 2:

...a variable, productID which is a primary key in the Products table that is a cookie.

Responsibility 3:

...setting a client variable called "chosenBook" that is a WDDX structure consisting of these keys: "ISBN", an integer; "title", a string; "author", a string; and "price", a number.

Responsibility 4:

...a global request.scope query called "myQ" consisting of "userID", a "lastName", "fistName" that is available to the fuse

Responsibility 5:

...a custom tag called "MyTag.cfm" needed by the fuse that should have "userID" passed into it, and that will return "lastName", "firstName" and "userGroups". "userID" is an integer, as is "userGroups"; the other two variables are strings.

Responsibility 6:

...a variable, "myVar" that is to be passed out of the fuse and that consists of a list.

Responsibility 7:

...a SESSION variable, "shoppingCart", an array that is available and that includes the following columns: 1 is product ID, 2 is description, 3 is price, 4 is quantity

Responsibility 8:

...a variable, "shoppingCart", an array that is passed into a custom tag and that includes the following columns: 1 is product ID, 2 is description, 3 is price, 4 is quantity

Responsibility 9:

...an exit fuseaction, "XFA.submitForm" passed into a fuse.

Responsibility 10:

...0 or more variables to be set as cookies, in the form "product_skuNumber", where "sku_Number" is an actual products SKU.

Responsibility 11:

...a variable, "badLogin" that will be sent out of the fuse only in some cases. If it is sent out, it will be a BOOLEAN value.

Responsibility 12:

...a variable, "myStruct" that may be available to the fuse. If it is, it will be a client variable, a WDDXd structure with the following keys: "myName", "myAddress", "myCity", "myState".

Ok now lets see how you stacked up.

Responsibility 1:

..a variable, an integer called "myVar" being passed explicitly into the fuse.

Answer:

```
--> myVar: an INTEGER
```

Responsibility 2:

...a variable, productID which is a primary key in the Products table that is a cookie.

Answer:

```
++> productID: a COOKIE PRIMARY KEY from Products table
or
++> cookie.productID: a PRIMARY KEY from Products table
```

Responsibility 3:

...setting a client variable called "chosenBook" that is a WDDX structure consisting of these keys: "ISBN", an integer; "title", a string; "author", a string; and "price", a number.

Answer:

```
<++ chosenBook: a CLIENT WDDX STRUCTURE
        < ISBN: an INTEGER
        < title: a STRING
        < author: a STRING
        < price: a NUMBER
or
<++ client.chosenBook: a WDDX STRUCTURE
        < ISBN: an INTEGER
        < title: a STRING
        < author: a STRING
        < price: a NUMBER
```

Responsibility 4:

...a global request.scope query called "myQ" consisting of "userID", a "lastName", "fistName" that is available to the fuse

Answer:

```
++> myQ: a REQUEST QUERY
        < userID
        < lastName
        < firstName
or
++> request.myQ: a QUERY
        < userID
        < lastName
        < firstName
```

Responsibility 5:

...a custom tag called "MyTag.cfm" needed by the fuse that should have "userID" passed into it, and that will return "lastName", "firstName" and "userGroups". "userID" is an integer, as is "userGroups"; the other two variables are strings.

Answer:

```
+++ MyTag.cfm
        > userID: an INTEGER
        < lastName: a STRING
        < firstName: a STRING
        < userGroups: an INTEGER
```

Responsibility 6:

...a variable, "myVar" that is to be passed out of the fuse and that consists of a list.

Answer:

```
<-- myVar: a LIST
```

Responsibility 7:

...a SESSION variable, "shoppingCart", an array that is available and that includes the following columns: 1 is product ID, 2 is description, 3 is price, 4 is quantity

Answer:

```
++> shoppingCart: a SESSION ARRAY where column 1 is product ID,
    column 2 is product description, column 3 is the price and
    col 4 the quantity.
or
++> session.shoppingCart: an ARRAY where column 1 is product ID,
    column 2 is product description, column 3 is the price and
    col 4 the quantity.
or
++> session.shoppingCart: an ARRAY
   < [1] as product ID
   < [2] as product description
   < [3] as price
   < [4] as quantity
```

Responsibility 8:

...a variable, "shoppingCart", an array that is passed into a custom tag and that includes the following columns: 1 is product ID, 2 is description, 3 is price, 4 is quantity

Answer:

```
--> shoppingCart: an ARRAY where column 1 is product ID, 2 is
    description, 3 is the price, and 4 the quantity
or
--> shoppingCart: an ARRAY
   < [1] as product id
   < [2] as description
   < [3] as price
   < [4] as quantity
```

Responsibility 9:

...an exit fuseaction, "XFA.submitForm" passed into a fuse.

Answer:

```
--> XFA.submitForm: a FUSEACTION
```

Responsibility 10:

...0 or more variables to be set as cookies, in the form "product_skuNumber", where "sku_Number" is an actual products SKU.

Answer:

```
<++ product_&skuNumber&*: a COOKIE STRING
        Developer Note: "&skuNumber& will be replaced by an actual product SKU at
run time
or
<++ cookie.product_&skuNumber&*: a STRING
        Developer Note: "&skuNumber& will be replaced by an actual product SKU at
run time
```

Responsibility 11:

...a variable, "badLogin" that will be sent out of the fuse only in some cases. If it is sent out, it will be a BOOLEAN value.

Answer:

```
<-- [badLogin]: a BOOLEAN
or
<-- badLogin?: a BOOLEAN
```

Responsibility 12:

...a variable, "myStruct" that may be available to the fuse. If it is, it will be a client variable, a WDDXd structure with the following keys: "myName", "myAddress", "myCity", "myState".

Answer:

```
++> [client.myStruct]: a WDDXd STRUCTURE
        < myName: a STRING
        < myAddress: a STRING
        < myCity: a STRING
        < myState: a STRING
or
++> client.myStruct?: a WDDXd STRUCTURE
        < myName: a STRING
        < myAddress: a STRING
        < myCity: a STRING
        < myState: a STRING
or
++> [myStruct]: a CLIENT WDDXd STRUCTURE
        < myName: a STRING
        < myAddress: a STRING
        < myCity: a STRING
        < myState: a STRING
or
++> myStruct?: a CLIENT WDDXd STRUCTURE
        < myName: a STRING
        < myAddress: a STRING
        < myCity: a STRING
        < myState: a STRING
```

A recap of this section:

This section asked you to write Fusedoc elements and then offered a valid Fusedoc element for you to compare yours with.

Fore an updated version of this tutorial and many more great tutorials and whitepapers please visit:

http://halhelms.com/index.cfm?fuseaction=tutorials.detail

Xfusebox development strategies

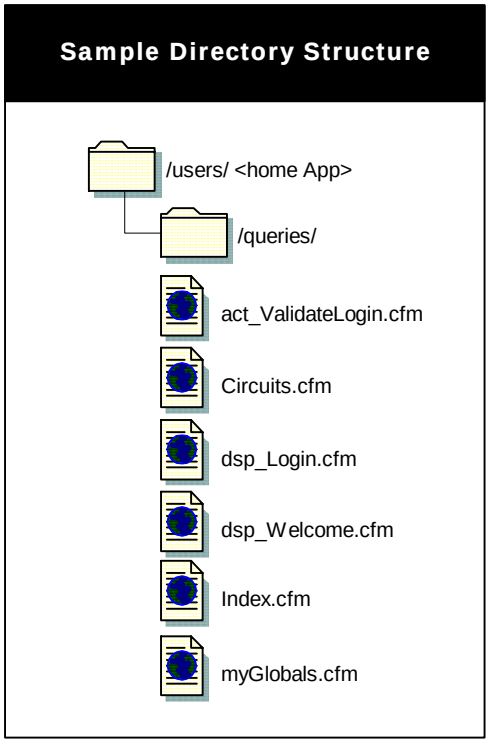
This section will provide additional information and tutorials on extended fusebox development methodologies and development strategies.

Nested Fuseboxes

Written by Hal Helms
Repurposed by Robert Foley Jr.

Full tutorial can be seen at: http://www.halhelms.com/index.cfm?fuseaction=tutorials.detail

Let's start by looking at a simple circuit application whose job is to provide user logins and validate these logins against a database. You've probably built such functionality many times, but here I'm going to show you how to reuse the entire circuit application with a minimum of changes to code.



First off, we will begin by evaluating the index.cfm file (also called the "fusebox"). Nested fuseboxes require a slightly altered index.cfm. One thing I do with all my files is place some code at the top of the file. At runtime, this will place the current file path in a JavaScript comment. Which can be viewed with "View Source". This is very helpful when debugging, as you can quickly see what files have executed.

1 – Setting debugging code

```
<cfoutput>
        <!-- #GetCurrentTemplatePath()# -->
</cfoutput>
```

Note: For security pruposes, you will probably NOT want to use this exact code, as it will revieal the underlying structure of you directories. Instead, use #GetFileFromPath(GetCurrentTemplatePath())# .

The next section of code for the index.cfm is a call to the custom tag FormURL2Attributes. This can be called from an included file or from the application.cfm if you prefer. I would suggest running it from the index.cfm file.

Here is the code:

2 – Running custom tag

```
<!--- Part of the Fusebox method is to convert incoming form or URL variables into attributes variables. This custom tag does that trick -->
<cf_formURL2attributes>
```

3 – Setting the default fuseaction

```
<cfparam name="attributes.fuseaction" default="users.home">
```

The next step is to setup the default fuseaction in the event that a fuseaction is not passed to the fusebox a default fuseaction is declared. You will note the dot notation used for defining the default fuseaction. When using nested fuseboxes, you need to provide both the circuit application name and the fuseaction within that circuit application. This method allows the same fuseaction name to be used in multiple circuits, requiring only that circuits have unique names.

3 – Check the usage status of the fusebox

Since any circuit could potentially be included into a larger application, this code checks to see if the ColdFusion application framework has already been set up. If it has NOT, this code will do so.. Please note that you could place this code in the global include file if preferred.

```
<!--- Am I the home application? -→
<cfif NOT IsDefined('application.applicationName')>
        <cfappliation clientmanagement="Yes" name="TEST">
</cfif>
```

4 – Include global variables

Now we call the global include file. Note that Extended Fusebox does not use appGlobals and appLocals. One file, myGlobals.cfm will work in both home and circuit application modes.

```
<cfinclude template="myGlobals.cfm">
```

5 – check usage and include circuits.cfm

Here's some code new to Extended Fusebox. Each circuit has a Circuits.cfm file that may be included and which creates a require structure called "Circuits". We'll see more about this in a moment. For now, note that it is called only if require Circuits doesn't already exist. If you think about this, you'll see that if an application was operating as a home circuit (and not as a nested circuit), no such structure would exist and this code would execute. If however, this circuit was nested into a larger application, the home application would a have already established requires Circuits and this code would not fire.

```
<!–Include Circuites if I'm the Home app --->
<cfif NOT IsDefined('request.Circuites')>
   <cfinclude template="Circuits.cfm">
</cfif>
```

6 – call nesting custom tag

The call to the custom tag Nesting.cfm is also new to Extended Fusebox. If you're curious about how the nesting mechanism works, you can examine this code. However,it isn't necessary to understand the internal workings of it any more than you need understand about internal combustion engines in order to drive a car.

```
<!--- translate the fuseaction sent to me into a fully qualified fuseaction --->
<cf_Nesting>
```

7 – Running fusebox switches

This last bit of Extended Fusebox specific code decides whether or not the fuseaction called belongs to this circuit or to one nested beneath it. If the fuseaction called is identified as a nested fusebox then the nested circuit's fusebox code is included into this fusebox's switching code and executed.

```
<!--- Peel off the first layer and etermine whether to delegate the fuseaction. --->
<cfif ListLen(attributes.fuseaction,'.') GT 1>
  <cfinclude template="#ListFirst(attributes.fuseaction,'.')#/index.cfm">
<cfelse>

<!--- I'm going to get one (and only one) fuseaction at a time. Fuseactions
correspond to methods or messages in OO languages. --->
```

Then run a cfswtch of fuseactions, the rest of the fusebox is identical to what you've been using all along.

Index.cfm

```
<cfoutput>
        <!-- #GetCurrentTemplatePath()# -->
</cfoutput>

<!--- Part of the Fusebox method is to convert incoming form or URL variables into
attributes variables. This custom tag does that trick -→
<cf_formURL2attributes>

<cfparam name="attributes.fuseaction" default="users.home">

<!--- Am I the home application? -→
<cfif NOT IsDefined('application.applicationName')>
        <cfappliation clientmanagement="Yes" name="TEST">
</cfif>

<cfinclude template="myGlobals.cfm">

<!—Include Circuites if I'm the Home app --->
<cfif NOT IsDefined('request.Circuites')>
   <cfinclude template="Circuits.cfm">
</cfif>

<!--- translate the fuseaction sent to me into a fully qualified fuseaction --->
<cf_Nesting>

<!--- Peel off the first layer and etermine whether to delegate the fuseaction. ---
>
<cfif ListLen(attributes.fuseaction,'.') GT 1>
  <cfinclude template="#ListFirst(attributes.fuseaction,'.')#/index.cfm">
<cfelse>

<!--- I'm going to get one (and only one) fuseaction at a time. Fuseactions
correspond to methods or messages in OO languages. --->
Then run a cfswtch of fuseactions, the rest of the fusebox is identical to what you've been using all along.

</cfif>
```

END Index.cfm

Now let's look at the Circuits.cfm.

The Circuits.cfm file create a request scope structure called "Circuits". We use this mechanism to "register" each circuit application and its sub-circuits. This lets the home application "know" what sub-circuits are operating within its sphere. Here, there is only one application, user. When there are additional, nested circuits, the parent's Circuits.cfm file is responsible for including them.

You'll see more of this in a bit.

```
<cfoutput>
<!--- #GetCurrentTemplatePath()# --->
</cfoutput>

<CF_AddCircuits
      MyName="users"
      Children=""
      LOCATION="#GetCurrentTemplatePath()#">
```

Now let's look at the myGlobals.cfm file

This file sets up the "environment" variables that circuit needs in order to run. With Extended Fusebox, you don't need an app_locals and an app_globals. Everything can be done in one fle.

```
<cfoutput>
<!--- #GetCurrentTemplatePath()# --->
</cfoutput>

<cfif NOT IsDefined('application.applicationName')>
     <cfapplication name="Users" sessionmanagement="Yes" clientmanagement="Yes">
</cfif>

<cfparam name="self" default="index.cfm">
<cfparam name="request.self" default="index.cfm">
<cfparam name="request.AppRoot" default="http:/:locathost/bpcf/compete/home">
```

Note the conditional logic throughout the file. This is because the circuit may be running in either stand-alone or nested circuit configuration. This will be determined at run time. If the circuit is in nested mode, the <cfparam>d variables will have already been set. Otherwise, we want them set now.

That's all there is to setting up for using Extended fuseboxes. Let's take a look at the fuses that make up the "Users" application.

Dsp_login.cfm

```
<cfoutput>
<!--- #GetCurrentTemplatePath()# --->
</cfoutput>

<!---
|| BEGIN FUSEDOC ||

|| Properties ||
Name: dsp_login.cfm
Author: Hal.helms@teamallaire.com

|| Responsibilities ||

I give the user a form to enter their username and password. If I get a var called
“badlogin”, the user has failed a login and I let them know.

|| Attributes ||

XFA.submitForm: a FUSEACTION
[badLogin]: a BOOLEAN ( TRUE )

← username: a STRING
← password: a PASSWORD

|| END FUSEDOC || --->

<cfoutput>

<cfparam name=”attributes.badLogin” default=”FALSE”>

<form action=”#self#?fuseaction=”#attributes.XFA.submitForm#” method=”post”>

<cfif attributes.badLogin>
       <font color=”red”> Sorry, we couldn’t log you in. Try again?<BR></font>
</cfif>

<table>
     <tr>
          <td align=”right”>User name:</td>
          <td><input type=”text” name=”username”></td>
     </tr>
     <tr>
          <td align=”right”>Password:</td>
          <td><input type=”password” name=”password”></td>
      </tr>
      <tr>
          <td align=”center” colspan=”2”><input type=”submit” value=”   ok
“></td>
     </tr>
</table>
```

END Dsp_login.cfm

As the Fusedoc directed, all this file does is provide a form to let the user enter their usename and password. Notice, though that the fuse action is NOT hard-coed, but uses a variable. A fuse normally has one or more ways of making a return trip to the fusebox. I call these “exit points” – and each exit point needs to have an eXit FuseAction (XFA) declared. Then the various XFAs are given values by the fusebox that uses them. If you think of the fuses as many thruly reusable pieces, there is no way of determining in what application they will be used – and the name of the fuseaction they should call. Using XFAs creates a layer between the application and the fuse. This layer of interface makes for smoother development. If instead of using XFAs , we hardcored the various exit point fuseactions, reusing the circuit application would mean having to open up individual fuses and making changes. XFAs let us “wire” fuses together in the fusebox without making any changes to fuses themselves. This is beneficial, not only for code reuse, but for code maintenance.

Act_ValidateLogin.cfm

The next fuse, act_ValidateLogin.cfm, runs a query that looks for the username and password in a talbe. If it finds, it, the use returns to the fusebox with an XFA of “attributes.XFA.success”. If not, it uses, “attributes.XFA.failure”.

```
<cfoutput>
<!--- #GetCurrentTemplatePath()# --->
</cfoutput>

<!—
|| BEGIN FUSEDOC ||

|| Properties ||
Name: act_ValidateLogin.cfm
Author: hal.Helms@TeamAllaire.com

|| Responsibilities ||

I call qry_ValidateLogin.cfm. If I get a recordcount, then the Login was successful
and I return to the fusebox with XFA.success; otherwise, the login failed and I
return to the fusebox with XFA.failure and an additional variable of “badLogin” set
to “TRUE”.

|| Attributes ||

→ XFA.success: a FUSEACTION
→ XFA.failure: a FUSEACTION
→ username: a STRING
→ password: a STRING

← [badlogin]: a BOOLEAN ( TRUE ) on XFA.failure

+++ qry_ValidateLogin.cfm
username: a STRING
password: a STRING
<    variables.ValidateLogin: a QUERY result set
      < userID: a STRING
      < userRoles: an INTEGER

|| END FUSEDOC || -→

<cfinclude template=”queries/qry_ValidateLogin.cfm”>

<cfif ValidateLogin.recordcount>
          <cfset currentUser = StructNew()>
          <cfset currentUser.userID = ValidateLogin.userID>
          <cfset currentUser.userRoles = ValidateLogin.userRoles>

          <cfwddx action=”CFML2WDDX” input=”currentUser#”
output=”#self#?fuseaction=#attributes.XFA.success# addtoken=”yes”>

<cfelse>
          <cflocation url=”#self#?fuseaction=#attributes.XFA.failure#&badLogin=TRUE” addtoken=”YES”>
</cfif>
```

END Act_ValidateLogin.cfm

Fuses work best when they do a small, descrete bit of work. If you find your fuses are hundreds of lines long, look to see if you really are combining what are logically separate fuseactions. Remember that the greatest code reuse occurs when the code is small and well-defined.

The last code base is the display page.

dsp_Welcome.cfm

```
<cfoutput>
        <!-- #GetCurrentTemplatePath()# -->
</cfoutput>

<!–
|| BEGIN FUSEDOC ||

|| Properties ||
Name: dsp_Welcome.cfm
Author: hal.helms@TeamAllaire.com

|| Responsiblities ||

I just say Hi

|| Attributes ||

|| END FUSEDOC || -→
```

Hi you are logged in.

END dsp_Welcome.cfm

Not much to it.

Now let's look in more detail and see how the functions are managed and called within the fusebox. Let's go back to index.cfm and look in more detail into the cfswitch cases.

Index.cfm -- continued

```
<cfswitch value="#attributes.fuseaction#">

<cfcase value="home, login.badlogin">
        <cfset attributes.XFA.submitForm = "users.validate">
        <cfinclude template="dsp_Login.cfm">
</cfcase>

<cfcase value="validate">
        <cfset attributes.XFA.success = "users.welcome">
        <cfset attributes.XFA.failure = "users.badLogin">
        <cfinclude template="act_ValidateLogin.cfm">
</cfcase>

<cfcase value="welcome">
        <cfinclude template="dsp_Welcome.cfm">
</cfcase>

<cfdefaultcase>
        <CF_UnknownFuseAction>
</cfdefaultcase>

</cfswtch>
```

End Index.cfm -- continued

Within the <cfswitch> of the fusebox. The value property of <cfcase> can be a comma-delimited list, letting you use several fuseaction names to accomplish the same action. Would you want to use more than one fuseaction? Well, I find that having a fuseaction “badLogin”. After a failed login is clearer than a fuseaction of “home” or a simple “login”.

Also note:

That the dsp_login.cfm that the fuaseaction name in the form.query string was not hard-coded, but used a variables, “attributes.XFA.submitForm”. Here is where that variables value is set.

```
<cfcase value="home, login.badlogin">
        <cfset attributes.XFA.submitForm = "users.validate">
        <cfinclude template="dsp_Login.cfm">
</cfcase>
```

And here we set XFAs for both success and failure.

```
<cfcase value="validate">
        <cfset attributes.XFA.success = "users.welcome">
        <cfset attributes.XFA.failure = "users.badLogin">
        <cfinclude template="act_ValidateLogin.cfm">
</cfcase>
```

Now, user login/validation is so common a thing, I’d like to be able to write it once and use it several places. And that’s just where Extended Fusebox with its nested fusedox scheme will come in very useful.

Let’s say you have a very small application called “MyApp”

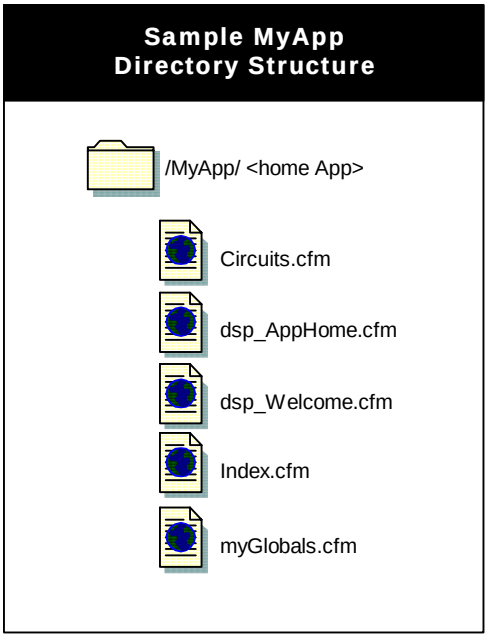
As you can see the application is very small. Let’s look at the Circuits file.

Circuits.cfm

```
<cfoutput>
        <!-- #GetCurrentTemplatePath()# -->
</cfoutput>
<CF_AddCircuits
        MyName="MyApp"
        Children=""
        LOCATION="#GetCurrentTemplatePath()#">
```

END Circuits.cfm

This app has no sub-circults and so it only declares its own circuit.



Let's look at the fusebox in the index.cfm

Index.cfm

```
<cfoutput>
        <!-- #GetCurrentTemplatePath()# -->
</cfoutput>

<!--- Part of the Fusebox method is to convert incoming form or URL variables into
attributes variables. This custom tag does that trick -→
<cf_formURL2attributes>

<!—In case no fuseaction was given, I'll set up one to use by default --->
<cfparam name="attributes.fuseaction" default="MyApp.home">

<!--- Am I the home application? -→
<cfif NOT IsDefined('application.applicationName')>
        <cfappliation clientmanagement="Yes" name="TEST">
</cfif>

<cfinclude template="myGlobals.cfm">

<!—Include Circuites if I'm the Home app --->
<cfif NOT IsDefined('request.Circuites')>
   <cfinclude template="Circuits.cfm">
</cfif>

<!--- translate the fuseaction sent to me into a fully qualified fuseaction --->
<cf_Nesting>

<!--- Peel off the first layer and etermine whether to delegate the fuseaction. ---
>
<cfif ListLen(attributes.fuseaction,'.') GT 1>
  <cfinclude template="#ListFirst(attributes.fuseaction,'.')#/index.cfm">
<cfelse>

<!--- I'm going to get one (and only one) fuseaction at a time. Fuseactions
correspond to methods or messages in OO languages. --->

<cfswitch expression="#attributes.fuseaction#">
        <cfcase value="home">
                <cfinclude template="dsp_MyAppHome.cfm">
        </cfcase>
        <cfdefaultcase>
                <CF_UnknownFuseAction>
        </cfdefaultcase>
</cfswitch>
</cfif>
```

End Index.cfm

Well this is a simple app. It has only one fuseaction, no XFAs, and a single fuse.

The file, myGlobals.cfm, looks very similar to the one used for the “Users” application.

myGlobals.cfm

```
<cfoutput>
<!--- #GetCurrentTemplatePath()# --->
</cfoutput>

<cfif NOT IsDefined('application.applicationName')>
     <cfapplication name="MyApp" sessionmanagement="Yes" clientmanagement="Yes">
</cfif>

<cfparam name="self" default="index.cfm">
<cfparam name="request.self" default="index.cfm">
<cfparam name="request.AppRoot" default="http:/:locathost/bpcf/compete/home">
```

END myGlobals.cfm

Now let’s look at the soul fuse of this app (dsp_MyAppHome.cfm)

dsp_MyAppHome.cfm

```
<cfoutput>
        <!-- #GetCurrentTemplatePath()# -->
</cfoutput>

<!---
|| BEGIN FUSE DOC ||

|| Properties ||
Name: dsp_MyAppHome.cfm
Author: hal.helms@TeamAllaire.com

|| Responsibilities ||
I check to see if client.currentUser exists. If it does, I deserialize it from its
WDDX format and say Hi and thell the user their userID. Otherwise, I just say Hi.

|| Attributes ||

[client.currentUser]: a WDDX STRUCTURE
   < userID. A STRING
   < userRoles: an INTEGER

|| END FUSEDOC || --->

<cfoutput>
Hi There!
<cfif IsDefined('client.currentUser')>
        <cfwddx action="WDDX2CFML" input="#client.currentUser#" output="temp">
        Your userID is #temp.userID#
</cfif>

</cfoutput>
```

End dsp_MyAppHome.cfm

The fusedoc specifies what this fuse should do. In this case, since we have no client variaible called “currentUser” (that was in the separate “Users” app), the entire application should consist of nothing more than “Hi there!” At least, it’s a friendly app.

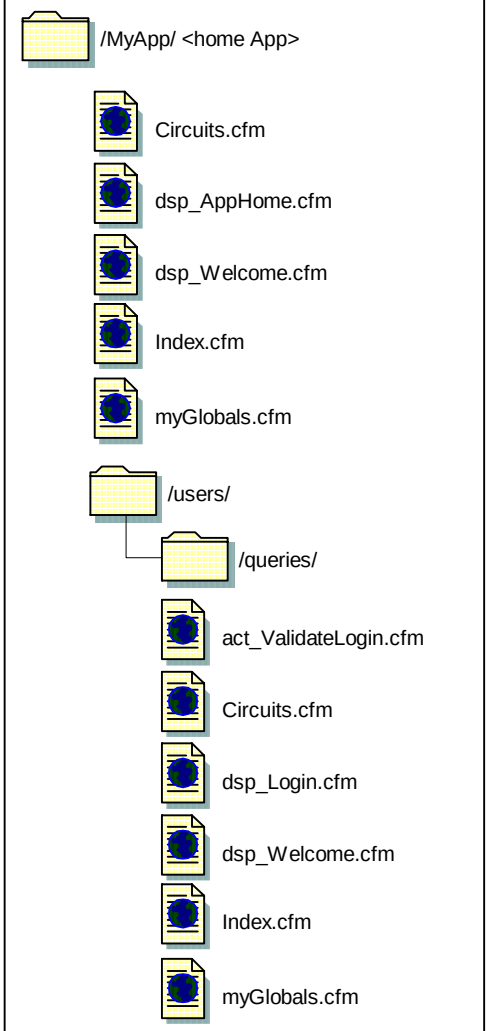
Now lets take the hello there! App and add the need for a user to log-in to view the content.

Using Extended Fusebox's nesting fusebox methodology, adding this in is a snap. You first copy the circuit (and any sub-circuits) into the direcotyr of it's new home.

Now we need to integrate the new circuit application into our main application "MyApp". The first change we will have to do is to the Circuits.cfm of the home application.

Since the parent application is responsible for knowing about any children under it, I add "users" into my Circuits structure.

Sample MyApp Directory Structure



Circuits.cfm

```
<cfoutput>
<!--- #GetCurrentTemplatePath()# --->
</cfoutput>

<CF_AddCircuits
        MyName="MyApp"
        Children="users"
        LOCATION="#GetCurrentTemplatePath()#">
```

End Circuits.cfm

Note that I use a dot notion to separate the parent ciruit name from the child circuit name. Here, I have only one sub-ciruit under MyApp, called users. If users had a sub-circuit called Employees, I would need to include that as

<cfset request Circuits.Employees = "MyApp.users.Employees">

Now, we need to rewire the fusebox so that the default fuseaction points to the login. We do that by opening up index.cfm and changing the default fuseaction to the circuit app instead of the home app.

Index.cfm

```
<cfoutput>
        <!-- #GetCurrentTemplatePath()# -->
</cfoutput>

<!--- Part of the Fusebox method is to convert
incoming form or URL variables into attributes
variables. This custom tag does that trick -→
<cf_formURL2attributes>

<!—In case no fuseaction was given, I'll set up one to use by default --->
<cfparam name="attributes.fuseaction" default="users.home">
```

Now, when a user comes to the page, the application will call the "home" fuseaction of the "Users" directory to make them log in. Remember that nested fuseboxes require that both the circuit name AND the fuseaction name be provided.

We now go to the index.cfm of the “users” circuit application.

users/Index.cfm

```
<cfswitch value="#attributes.fuseaction#">

<cfcase value="home, login.badlogin">
        <cfset attributes.XFA.submitForm = "users.validate">
        <cfinclude template="dsp_Login.cfm">
</cfcase>

<cfcase value="validate">
        <cfset attributes.XFA.success = "MyApp.home">
        <cfset attributes.XFA.failure = "users.badLogin">
        <cfinclude template="act_ValidateLogin.cfm">
</cfcase>

<cfcase value="welcome">
        <cfinclude template="dsp_Welcome.cfm">
</cfcase>

<cfdefaultcase>
        <CF_UnknownFuseAction>
</cfdefaultcase>

</cfswtch>
```

END users/index.cfm

Instead of sending validated users to the welcome fuse in users, we now want to send validated users to the home fuseaction of MyApp. It’s as simple as re-writing the fusebox. Now, in the validate cfcase, attributes.XFA.success points to MyApp.home. Because we’re using XFAs, we did not have to make changes to the fuses themselves.

Nested circuits have other advantages as well, as each circuit “inherits” from the circuit above it. (more on this in another tutorial.) Also, runtime exceptions can either be handled in the circuit they occurred, or exceptions can “bubble up” to parent circuits – all the way to the home circuit. Now, if you use nested circuits to add a shopping car to the friendly App, all you need to do is spend all that money!

Fusebox Directory and Naming Standards

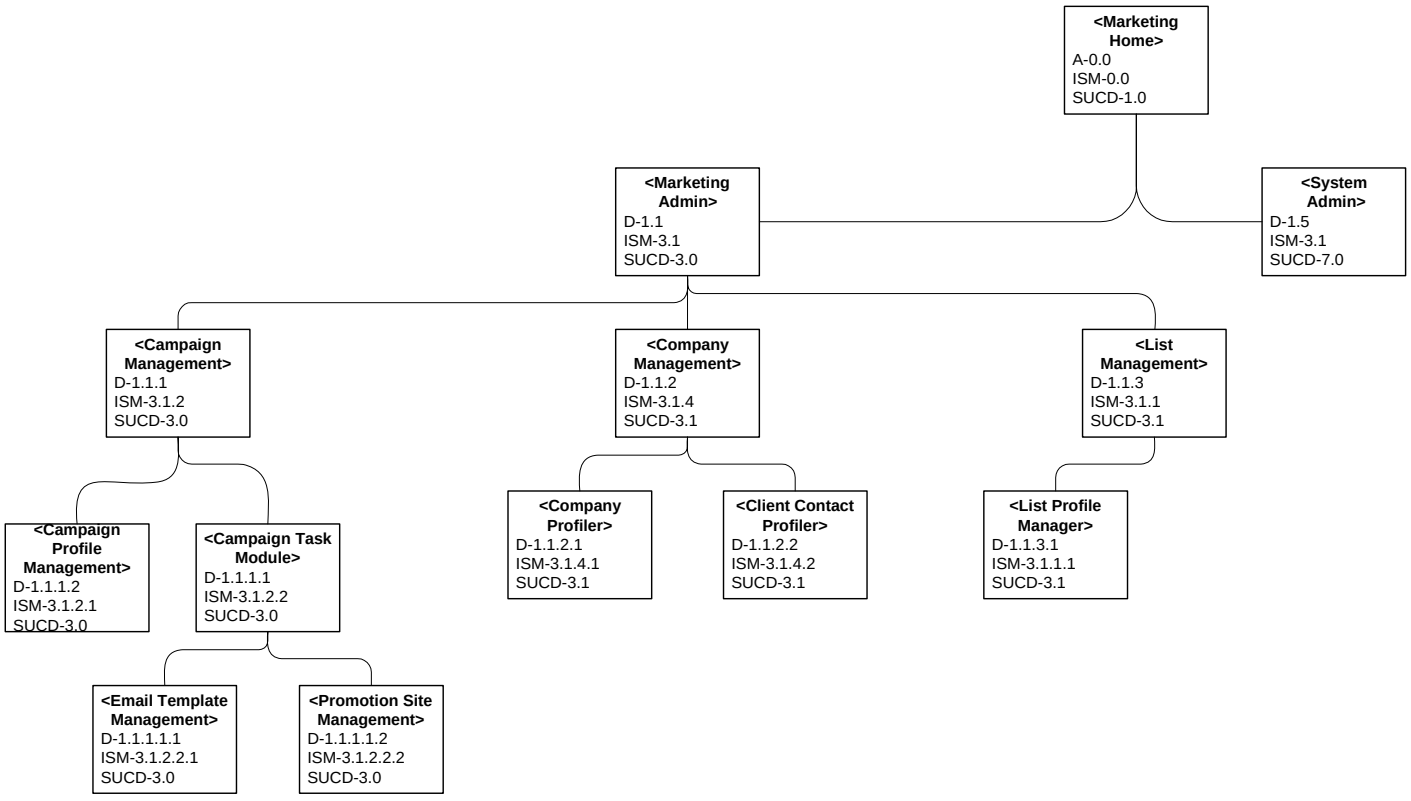
Ok, so you made it through the tutorials, good job!

The next step is learning the naming standards for programming Fusebox circuits in the IDM. The fusebox application is designed to be created in small circuit applications. The directory structure of each circuit is exactly the same in structure from circuit application to circuit application. In fact, even the home application has the same structure as the circuit applications. This design is the most efficient and promotes object oriented programming patterns, also it ensures that your circuits are compliant with the xFusebox methodology.

Application Architecture Naming Standards

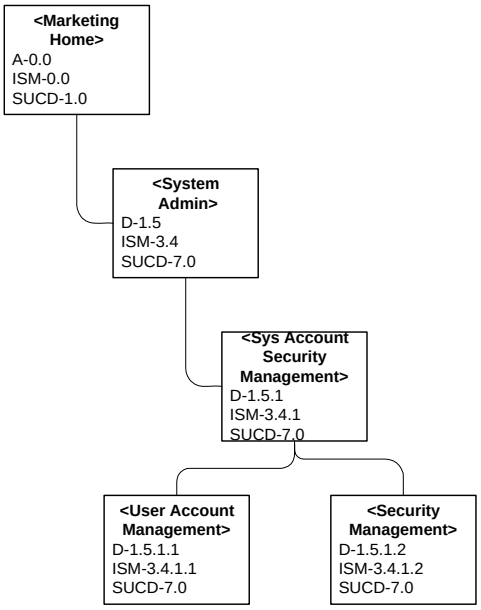
The application directory structure, which is the organization of the circuit applications under the home application. Are separated into directories for each of the system functions defined in the Functional system module diagrams, System Use Case Diagrams, and System Use Case Diagrams.

The following diagram represents a top down flow of Phase 1.0 directory structure of the IDM system.



SDSD-1.0 - Marketing Tool Directory Structure Use Case Version 1.0

The rest of the directory structure is on the next page.

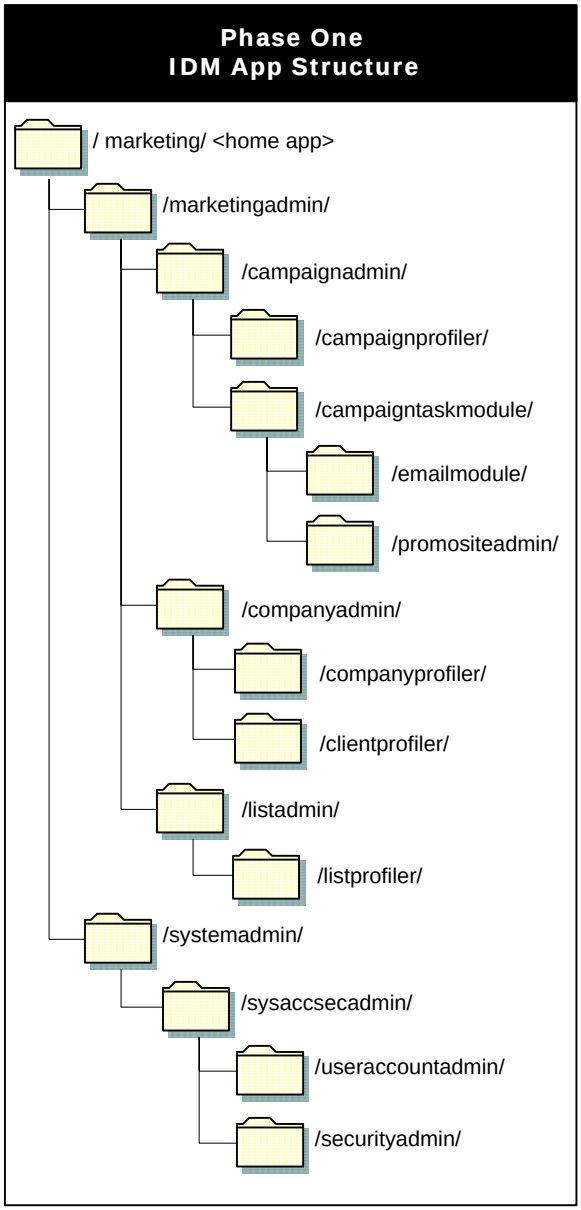


SDSD-1.1 - Marketing Tool Directory Structure

The diagrams are based on a top down flow of each system component and the circuit applications bellow them. If you are have having a hard time visualizing the application structure, a traditional directory structure diagram is provided to the right.

The home application, which is the marketing system is the root directory, followed by the main sections and the circuit applications and so forth. Until all of the primary and support applications have been outlined.

After the application directories have been defined, each application directory has a series of support directories and required files. The directory structure is organized into “bins” where similar functions within the fusebox methodology are gathered for easy referencing and reuse.



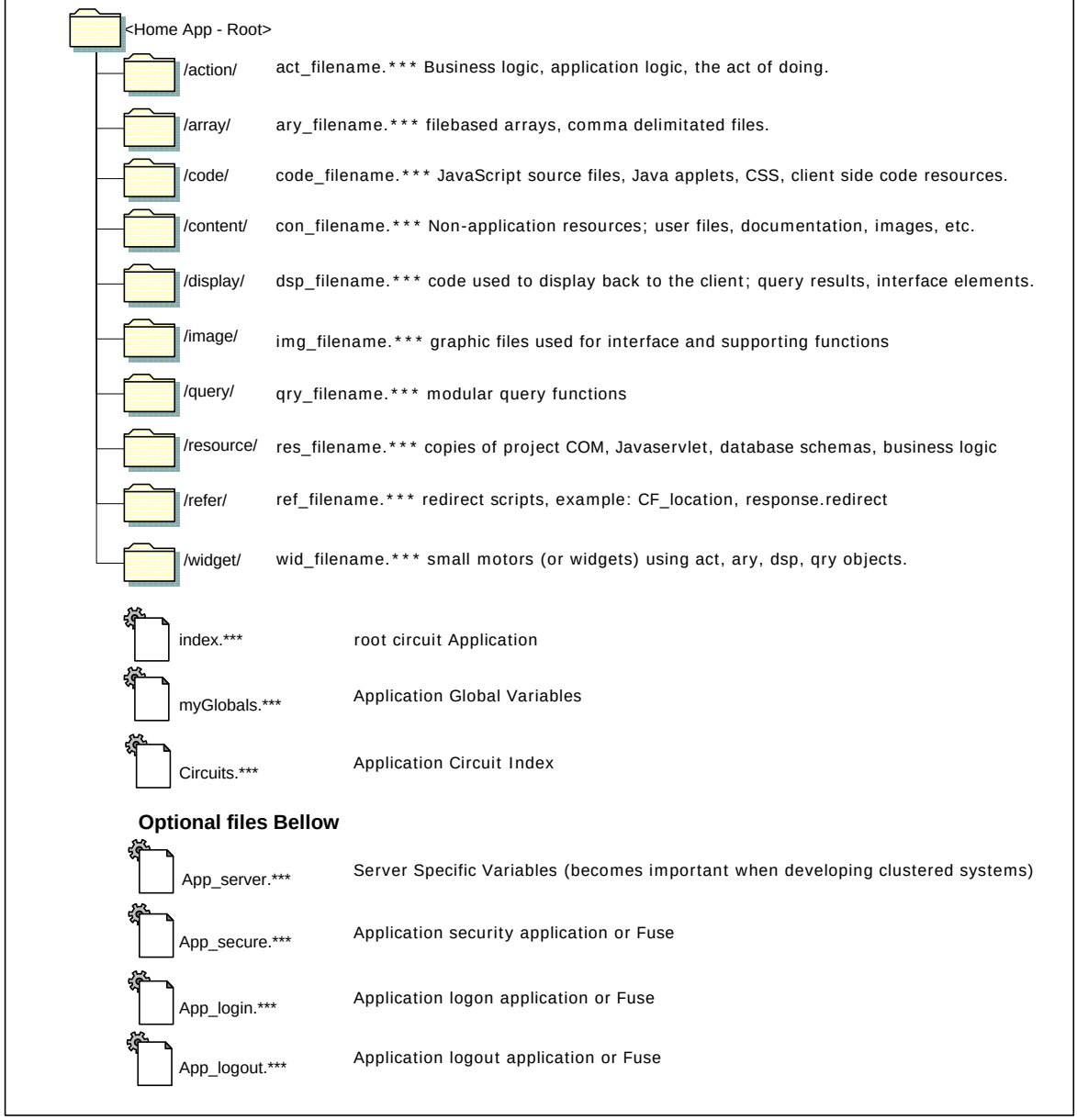
Resource Directory Structure Naming Standards

The resource directory structure is organized into the following directories. There is a total ten resource folders the first is the “action” folder. As the name implies all action-based files for this circuit are put in this folder. The second folder is “array”, this folder holds file based arrays or code that manipulates arrays.

The third folder is “code”, this folder holds all Javascript, CCS, Java applets, and client side code resources for the circuit application.

The fourth folder is “content”, this folder holds content specific to the topics and subjects discussed in the circuit application. For example, word docs, project images, or section headers. Things that are unique to a specific section or topic are put into this folder.

Application Resource Definitions



The fifth folder is “display”, this folder holds application files that display the interface and content of the circuit application.

The sixth folder is “image”, this folder contains all image resources that support and makeup the interface and application functions of the circuit application.

The seventh folder is “query”, this folder holds all application code used to query the database. This is useful if you have multiple display templates outputting the same query recordset. Or have a query function you are going to use in multiple locations or in compound functions.

The eighth folder is “resource”, This folder holds setup or read me documentation, copies of COM or Java servlet objects that are used within the circuit, database schemas, or business logic documentation. Basically any component or information required for the manipulation or setup of the circuit either within a new application or for maintenance of the circuit.

The ninth folder is “refer”, this folder holds application logic for redirection scripts that could be reused in multiple location throughout the application. This is commonly done using CF_location in CFML or response.redirect in ASP.

The last folder is “widget”, widgets are small collections of act, ary, dsp, and qry objects that perform a simple function. But are not big enough to be made into a sub circuit application. An example would be a small widget that displays the top ten projects for a specific user. The widget has two setting, one is to list the first five current project and the second is to list all current projects. This little widget could be included in the main section of an application and also referenced in a sub application. The architectural challenge is to determine if the widget should be put within the parent application or the child circuit application.

Within the resource directory folder there are three primary fusebox application files that must exist for the xFusebox application to function. The files are titled index.cfm, myGlobals.cfm, and Circuits.cfm. For more information about the structure and development methodology employed with the files please reference: Fusebox 101 on page 8.

Programming File Naming Standards

Now that we have defined the application and resource naming standards the next topic to define is the standards for naming application files and resources within the fusebox architecture. If you reference the Application Resource Definitions Diagram on Page 78. Each resource directory contains application files. When creating application logic for the fusebox methodology there are distinct names given based on how the application file is used.

Action files are named	act_filename.***
Array files are named	ary_filename.***
Display files are named	dsp_filename.***
Query files are named	qry_filename.***
Refer files are named	ref_filename.***
Widget files are named	wid_filename.***

All other files are named at the discretion of the authors.

The use of *** signifies that what ever lanaguage being used should be the prefix for the file.

CFM, ASP, PL, PHP, JSP, etc.

Fusebox File Structure Standards

App file type	Unique function title	Prefex
Act	_ Filename	.***

Example:

act_sortuserlist.cfm

dsp_primarymenu.cfm

ary_internetwebsiteslist.cfm

ref_mainhome.home.cfm

Summary

This document has provided a detailed tutorial on fusebox and fusedocs. This knowledge is required to conduct development and maintenance of the applications. Also, detailed rules and guidelines have been written for the application, resource, and file naming standards that are required when development modules and or maintaining the application.

If you have any questions or comments please contact Robert Foley Jr.

Please reference code, and sample applications on the network here:
\\IMPALA\projects\interactive\Jobs\Other_Jobs\1765_Internal_IDM\Programming\Working\Source_Samples

Appendix

The following is a collection of useful diagrams, statistics, and tables.

Attributes symbols for Fusedoc:

```
--> Indicates an "attributes" style variable explicitly passed
    into the fuse

<-- indicates an variable explicitly passed out of the fuse
    (as a URL or form variable)

++> indicates a global variable (request, session, client,
    application, server) available to the fuse. Also used to
    indicate a cookie available to the fuse.

<++ indicates a global variable set by the fuse. Also used to
    indicate a cookie set by the fuse.

<-> indicates a pass-thru variable, whose value will not be
    changed by the fuse

+++ indicates a file required by the fuse


[]  indicates the element specified is optional

?   indicates the quantity of the element specified is 0 or 1 (same as optional)

+   indicates the quantity of the element specified is 1 or more

*   indicates the quantity of the element specified is 0 or more
```

Glossary

This glossary is a collection of terms and definitions used within this document.