Fusebox 5.5

Release Notes

Saturday, December 1, 2007

Table of Contents

Overview

This document describes the new features in Fusebox 5.5 and will ultimately be integrated into the main Fusebox documentation.

There are two primary themes to Fusebox 5.5:

1. Simplify. Remove barriers. Make it easier for newbies. Make building applications faster by favoring convention over configuration.
2. Extend through extensibility. Provide new functionality outside the core. Add plugins, lexicons, even standardized circuits.

In addition, backward compatibility is of paramount importance so all Fusebox 4.x and Fusebox 5.x applications should run, unchanged, on Fusebox 5.5. In theory, you can just unzip the new core files over the top of your existing core files and everything should just work.

Theme: Simplification

According to the survey conducted early in 2007 by TeraTech, Inc., the main complaint about Fusebox 4.x through Fusebox 5.1 is the use of XML to configure the application and to specify the flow of control. Whether real or merely perceived, the use of XML appears to be the single largest barrier to adoption of Fusebox. Accordingly, the primary focus of this theme for Fusebox 5.5 is to make the XML files optional for most use cases by adopting a set of reasonable defaults and conventions. The Alpha allows you to build Fusebox applications without using XML, as long as you follow a number of conventions in terms of file system layout and naming.

Theme: Extensibility

Fusebox 4.0 introduced a number of extension points into the framework that allow users to extend the behavior of the core files without needing to change the core files. Subsequent releases have continued on this path with Fusebox 5.0 introducing lexicons as a way to extend the language of Fusebox. Fusebox 5.5 now offers a standard set of extensions as a separate download that includes: lexicons for ColdFusion-like verbs, ColdSpring, ORMs (Object Relational Mappings) for both Transfer and Reactor, and a more comprehensive lexicon for Reactor; plugins for assertions; scaffolding for quickly generating basic data management applications.

Making XML Optional

Fusebox 4.x through Fusebox 5.x use XML for two purposes:

1. `fusebox.xml` – for configuring your application: setting application parameters and declaring classes, plugins and global fuseactions;
2. `circuit.xml` – for specifying the logical flow of control for a given fuseaction.

Within these XML files, many of the settings and/or attributes are already optional, with documented default values. In order to make the XML files themselves optional, another mechanism must be provided to either override the defaults or specify values that cannot be defaulted. For example, the default value for the `fuseactionVariable` is `"fuseaction"` but the default value for the `defaultFuseaction` is `""`. Clearly the latter has to be overridden for all applications.

The simplest Fusebox application

Here is the classic Hello World! application in Fusebox 5.5 in the simplest possible form:

`index.cfm`:

```
	<cfapplication name="helloworld" />
	<cfset FUSEBOX_PARAMETERS.defaultFuseaction = "main.welcome" />
	<cfset FUSEBOX_PARAMETERS.allowImplicitFusebox = true />
	<cfinclude template= "/fusebox5/fusebox5.cfm" />
```

`main/welcome.cfm`:

```
	Hello World!
```

This simple, two file application displays `Hello World!` with no XML. The circuit `main` is resolved to the directory `main/` and the fuseaction `welcome` is resolved to the fuse `welcome.cfm`.

The following sections explain in more detail how you can build Fusebox applications without XML and what options you have for structuring your code.

FUSEBOX_PARAMETERS and omitting fusebox.xml

Fusebox 5.5 introduces a new top-level variable called `FUSEBOX_PARAMETERS` which is a structure that you can use to specify parameters that would normally be set in the `<parameters>` block inside `fusebox.xml`. You can use this in your `index.cfm` file (or `Application.cfc` - see below) to specify parameters, prior to including the core file entry point:

```
<cfset FUSEBOX_PARAMETERS.defaultFuseaction = "controller.welcome" />
<cfset FUSEBOX_PARAMETERS.debug = true />
```

These values will override any matching `<parameter>` settings in `fusebox.xml` whenever the Fusebox application is reloaded.

Fusebox 5.5 also introduces a new parameter setting, `allowImplicitFusebox`, which allows the `fusebox.xml` file to be omitted. Naturally, this has to be set in `index.cfm` (or `Application.cfc`):

```
<cfset FUSEBOX_PARAMETERS.allowImplicitFusebox = true />
```

When this is specified as `true`, the `fusebox.xml` file may be omitted and Fusebox uses default values for all settings that are not explicitly specified in the `FUSEBOX_PARAMETERS` structure in `index.cfm` (or `Application.cfc`).

If you omit the `fusebox.xml` file, the only settings that will be active will be those specified in the `FUSEBOX_PARAMETERS` structure and any standard Fusebox defaults for settings not specified in that structure.

Omitting circuit.xml

Fusebox 5.1 introduced the parameter setting, `allowImplicitCircuits`, so that you could omit the `circuit.xml` file for a circuit that is used only as a container for fuses that are included by other circuits (via the `<include>` verb with the `circuit=` attribute). In such a situation, the `circuit.xml` file, if present, would be "empty", containing only `<circuit/>`, possibly with an `access=` attribute.

Fusebox 5.5 extends the meaning of `allowImplicitCircuits` to also indicate that circuit declarations can be omitted completely. If you omit `fusebox.xml`, you have no circuit declarations and so `allowImplicitCircuits` is automatically set to true. If `fusebox.xml` is present but you wish to omit the circuit declarations, you must set `allowImplicitCircuits` to true.

Implicitly locating a circuit

If you attempt to execute a fuseaction in a circuit that is not declared, Fusebox 5.5 attempts to deduce the location and form of the circuit using a series of conventions.

Fusebox 5.5 first attempts to find a regular `circuit.xml` (or `circuit.xml.cfm`) file by looking in the following locations relative to the application root (assuming a fuseaction of *alias.action*):

- controller/*alias*/
- model/*alias*/
- view/*alias*/
- *alias*/controller/
- *alias*/model/
- *alias*/view/
- *alias*/

If a circuit file is found, it is treated as a regular Fusebox circuit that simply was not declared in `fusebox.xml`. Otherwise, Fusebox attempts to find a CFC that represents the circuit by looking for:

- controller/*alias*.cfc [access="public"]
- model/*alias*.cfc [access="internal"]
- view/*alias*.cfc [access="internal"]

If such a CFC is found, it is treated as a circuit with the methods representing fuseactions – see below. Otherwise, Fusebox looks for matching directories:

- controller/*alias*/ [access="public"]
- model/*alias*/ [access="internal"]
- view/*alias*/ [access="internal"]
- *alias*/ [access="public"]

If such a directory is found, it is treated as a circuit with the files within that directory representing fuseactions – see below. Otherwise, Fusebox throws an exception that the requested circuit is undefined.

Whilst this may seem like a complicated set of rules, it is intended to support the most common conventions used today in Fusebox applications (and, in fact, in several other frameworks). The intent is that Fusebox should be intuitive to use without having to think about the rules: the common conventions should "just work".

The first searches – for `circuit.xml` files – are intended to support common Fusebox conventions in the absence of a `fusebox.xml` file. The remaining searches are intended to support a more object-oriented approach (CFCs-as-circuits) or a simple procedural approach (directories-as-circuits). This is in line with Fusebox's goals of supporting a wide range of programming styles while not getting in the way of the programmer.

CFCs as Circuits

If Fusebox identifies a ColdFusion Component as a circuit, it expects to be able to call methods on that CFC corresponding to fuseactions within the specified circuit. If a request is made for *alias.action* and Fusebox determines that *`alias`*`.cfc` is the specified circuit, then the following method should exist in that CFC:

```
<cffunction name="action">
      <cfargument name="myFusebox" />
      <cfargument name="event" />
      ... perform the fuseaction ...
</cffunction>
```

The method may execute other fuseactions – using the dynamic "do" operation on `myFusebox` (see below) – and may store result data into the `event` object (which is an encapsulation of the Fusebox `attributes` scope).

If a method named `prefuseaction` is present, it is called before each *`action`* method. If a method named `postfuseaction` is present, it is called after each *`action`* method. `prefuseaction()` and `postfuseaction()` methods share the same `variables` scope as the *`action`* method.

Results may be passed back via two mechanisms:

- Using `myFusebox.variables().key = value` to assign directly into Fusebox's common `variables` scope – consider this to be the "old-school", procedural approach.

- Using `event.setValue("key",value)` to assign into Fusebox's `attributes` scope, which is wrapped in the `event` object – consider this to be the more modern, object-oriented approach.

Directories as Circuits

If Fusebox identifies a directory as a circuit, it expects to find individual files within that directory corresponding to fuseactions within the specified circuit. If a request is made for *alias.action* and Fusebox determines that `alias/` is the specified circuit, then one of the following files should exist in that directory:

- *`action`*`.xml`
- *`action`*`.cfm`
- *`action`*`.cfc`

The behavior in each of these scenarios is described in the following sections.

CFML Templates as Fuseactions

A request for *alias.action* is resolved to the file `action.cfm` in the directory `alias/`. The behavior is as-if the following code existed in a `circuit.xml` file:

```
<fuseaction name="action">
     <include template="action" />
</fuseaction>
```

In other words, the `action.cfm` file is included as a regular fuse file.

If a file named `prefuseaction.cfm` is present in the same directory, it is included before the `action.cfm` file. If a file named `postfuseaction.cfm` is present in the same directory, it is included after the `action.cfm` file.

CFCs as Fuseactions

A request for *alias.action* is resolved to the file `action.cfc` in the directory `alias/`. Fusebox treats the ColdFusion Component as a “command” object and the following method should exist in that CFC:

```
<cffunction name="do">
     <cfargument name="myFusebox" />
     <cfargument name="event" />
     ... perform the fuseaction ...
</cffunction>
```

The method may execute other fuseactions – using the dynamic “do” operation on `myFusebox` (see below) – and may store result data into the `event` object (which is an encapsulation of the Fusebox `attributes` scope).

If a method named `prefuseaction` is present, it is called before the `do()` method. If a method named `postfuseaction` is present, it is called after the `do` method. `prefuseaction()` and `postfuseaction()` methods share the same `variables` scope as the `do()` method.

Dynamic Do

Fusebox 5.5 adds a new method to the myFusebox object:

```
string do( action : string,
           contentvariable : string = "",
           append : boolean = false,
           returnOutput : boolean = false );
```

This allows code to execute an arbitrary fuseaction at runtime. The specified fuseaction may either be fully qualified (*alias.action*) or just an action within the currently executing circuit. For example:

```
<invoke object="myFusebox" methodcall="do('main.welcome')" />
```

This is equivalent to:

```
<do action="main.welcome" />
```

except that you could specify a dynamic value instead of `'main.welcome'`. If you are already executing a fuseaction in the circuit `main`, this is also equivalent to either of the following:

```
<invoke object="myFusebox" methodcall="do('welcome')" />
```

```
<do action="welcome" />
```

A dynamic "do" operation is still compiled down to straight-line CFML in the `parsed/` directory, just like a regular fuseaction request. Whereas regular fuseaction requests are compiled down to files called `*alias*.action.cfm` which implement complete requests, a dynamic "do" is compiled down to a file called `do.*alias*.*action*.cfm` which implements only part of a request.

Handling Content Variables

If you specify `contentvariable=` and specify a variable name, `do()` will capture any generated output and store it into that named variable in the top-level variables scope. For example:

```
<set value="#myFusebox.do(action='welcome',
                          contentvariable="body")#" />
```

is equivalent to:

```
<do action="welcome" contentvariable="body" />
```

Only simple variable names may be used in this context. If you use a dotted name (i.e., qualified with a scope or struct name), it will behave like:

```
variables["dotted.name"] = content
```

rather than:

```
variables.dotted.name = content
```

If you specify a content variable, you may optionally specify that the content be appended to the variable, using `append=true`, e.g.,

```
<set value="#myFusebox.do(action='welcome',
                          contentvariable="body",
                          append=true)#" />
```

is equivalent to:

```
<do action="welcome" contentvariable="body" append="true" />
```

The `append=` attribute has been added since the Public Beta.

If you need to store content in a struct variable or scoped variable, you need to use the `returnOutput` argument instead.

If you specify `returnOutput=true`, `do()` will return any generated output as a string (and will not output anything). This is the equivalent of using a content variable. For example:

```
<set name="content.body"
     value="#myFusebox.do(action='welcome',
                          returnOutput=true)#" />
```

is equivalent to:

```
<do action="welcome" contentvariable="content.body" />
```

Model-View-Controller w/out XML

Let's look at a simple MVC Fusebox 5.5 application that uses a CFC for the controller:

`index.cfm`:

```
<cfapplication name="helloworld" />
<cfset FUSEBOX_PARAMETERS.defaultFuseaction = "main.welcome" />
<cfset FUSEBOX_PARAMETERS.allowImplicitFusebox = true />
<cfinclude template= "/fusebox5/fusebox5.cfm" />
```

`controller/main.cfc`:

```
<cfcomponent>
     <cffunction name="welcome">
          <cfargument name="myFusebox" />
          <cfset myFusebox.do("dsp.welcome") />
     </cffunction>
</cfcomponent>
```

`view/dsp/welcome.cfm:`

```
Hello World!
```

In this example, we are taking advantage of the Fusebox 5.5 convention that searches for circuits within `controller/`, `model/` and `view/` directories, as well as the convention that allows a CFC to be treated as a circuit. The `main.welcome` fuseaction is resolved to the `welcome()` method in the `main.cfc` component (circuit) in the `controller/` directory. That fuseaction (method) uses the new dynamic “do” operation to execute the `dsp.welcome` fuseaction. The `dsp.welcome` fuseaction is resolved to the `welcome.cfm` fuse in the `dsp` circuit in the `view/` directory.

prefusaction / postfuseaction

A circuit CFC may contain methods called `prefuseaction()` and/or `postfuseaction()` which are called automatically by the framework, just as `<prefuseaction>` and `<postfuseaction>` works in a `circuit.xml` file. This can used to handle automatic layouts, security etc, e.g.,

`controller/main.cfc`:

```
<cfcomponent>
     <cffunction name="postfuseaction">
          <cfset myFusebox.do("lay.main") />
     </cffunction>
     <cffunction name="welcome">
          <cfargument name="myFusebox" />
          <cfset myFusebox.do("dsp.welcome","body") />
     </cffunction>
</cfcomponent>
```

In this example, the `controller.welcome` fuseaction executes the `dsp.welcome` fuseaction and captures its output into the (top-level) variable called `body`. The `postfuseaction()` method will be executed automatically which executes the `lay.main` fuseaction (which will wrap `#body#` in a layout).

Thread safety and CFC instantiation

Fusebox 5.5 instantiates a circuit CFC on each request so you do not have to worry about thread safety and this also discourages you from putting complex logic in your circuit CFC (since it cannot have state). Your circuit CFCs should be very simple, just providing simple flow of control logic that interacts with the underlying business model and selecting display fuseactions. Your business model service layer objects can be stored in the Fusebox application data and accessed via the `getApplicationData()` method on the `myFusebox` object (a new method in Fusebox 5.5).

To clarify, Fusebox 5.5 instantiates a single instance of each requested circuit CFC per-request and reuses it during that request. This means that the `prefuseaction()` and `postfuseaction()` and regular fuseaction methods can all communicate using the CFC's `variables` scope. In particular, if a single request causes multiple methods of the same circuit CFC to be executed, those methods will also all share a single instance.

The Event Object

Fusebox 5.1 introduced the `event` object as an encapsulation for the `attributes` scope, making it cleaner to refer to within object-oriented Fusebox code. Fusebox 5.5 adds the XFA pseudo-scope structure to the `event` object so that exit fuseactions can be manipulated more easily in object-oriented Fusebox applications. This adds one new method with the following behaviors:

```
string xfa( name : string );
string xfa( name : string, value : string );
```

The first form simply returns the specified XFA value. The second form sets the named XFA to the specified value (but see also **XFA Parameters** below). If the value is not a fully-qualified fuseaction, the current circuit name is used to create a fully-qualified fuseaction. For example:

`controller/main.cfc`:

```
<cfcomponent>
	<cffunction name="welcome">
		<cfargument name="myFusebox" />
		<cfargument name="event" />
		<cfset event.xfa("home","welcome") />
		<cfset myFusebox.do("dsp.welcome") />
	</cffunction>
</cfcomponent>
```

`view/dsp/welcome.cfm:`

```
<cfoutput>
<p>Hello World!</p>
Go <a href="#myFusebox.getMyself()##xfa.home#">home</a>!
</cfoutput>
```

In this example, the `main.welcome` fuseaction (method) sets the home XFA to `welcome` (which is actually `main.welcome` since the currently executing fuseaction is in the `main` circuit). The `dsp.welcome` fuseaction references that XFA (and the built-in “myself” value). Note that XFAs set inside the `event` object are still directly accessible in the `variables` scope in a fuse.

XFA Parameters

The `<xfa>` verb supports nested `<parameter>` tags to allow you to specify URL parameters that are handled by the SES URL generation code (if you set the SES URL parameters `queryS-tringStart`, `queryStringSeparator` and `queryStringEqual` in `fusebox.xml` or the `FUSEBOX_PARAMETERS` structure). Support for this is provided through the `xfa()` method by allowing an arbitrary number of additional argument pairs when you set an XFA value. For example:

```
<cfset event.xfa("next","main.home","message","Thank you!") />
```

This sets the XFA next to `main.home` and also adds the message URL parameter with the value Thank you! according to the settings of the SES URL parameters. With the standard settings, you would get:

```
main.home&message=Thank%20you!
```

With SES URL settings using forward slashes, you would get:

```
main.home/message/Thank%20you!
```

myFusebox.relocate()

In keeping with the changes to the `event` object to support XFAs, `myFusebox` now has a `relocate()` method that is identical to the `<relocate>` verb except that JavaScript redirection is not supported. The method has the following signature:

```
void relocate( url : string = "", xfa : string = "",
                addtoken : boolean = false,
                type : string = "client" );
```

You can specify either a full URL using the `url` argument or an XFA name using the `xfa` argument, just like the `<relocate>` verb, therefore you must use named arguments to specify `xfa`, `addtoken` or `type`. A common usage would be something like this:

```
event.xfa("next","task.show","taskId",id);
myFusebox.relocate(xfa="next");
```

This sets an XFA, `next`, with a URL parameter, `taskId`, and then relocates to it.

Application.cfc Support

Fusebox has always supported `Application.cfm` and `index.cfm` as the standard entry points for the framework:

`Application.cfm` (provided by the skeleton application):

```
<cfsilent>
<cfif right(cgi.script_name, len("index.cfm")) neq "index.cfm"
          and right(cgi.script_name, 3) neq "cfc">
     <cflocation url="index.cfm" addtoken="no" />
</cfif>
<!--- there must be no newline after the closing cfsilent tag
      if you want all leading whitespace suppressed --->
</cfsilent>
```

`index.cfm`:

```
<cfapplication name="someApplicationName" />
<!--- optionally set FUSEBOX_* variables --->
<cfinclude template="/fusebox5/fusebox5.cfm" />
```

Fusebox 5.5 introduces support for `Application.cfc`:

`Application.cfc`:

```
<cfcomponent extends="fusebox5.Application" output="false">
     <cfset this.name = "someApplicationName" />
     <!--- optionally set FUSEBOX_* variables --->
</cfcomponent>
```

`index.cfm`:

```
<!--- empty file --->
```

The `index.cfm` file has to be present but is **not actually used** so it should be empty.

Event Handler Methods

If you override any of the event handler methods in your `Application.cfc`, you must invoke the `super` method at the beginning of your method code, otherwise the base Fusebox `Application.cfc` methods will not be executed and Fusebox will not work correctly.

For example, `onRequestStart()`:

```
<cffunction name="onRequestStart">
     <cfargument name="targetPage" />

     <cfset super.onRequestStart(arguments.targetPage) />

     <cfset self = myFusebox.getSelf() />
     <cfset myself = myFusebox.getMyself() />

</cffunction>
```

This example shows how to do something in `Application.cfc` that would previously have been done in `fusebox.init.cfm` or the `<preprocess>` global fuseaction (and which can still be done that way in Fusebox 5.5, unless you omit `fusebox.xml`). The `self` and `myself` variables are set into the `Application` component's `variables` scope, which is also the top-level Fusebox `variables` scope (because `onRequest()` is used to process the actual request). Note that `myFusebox` is also available in the `Application` component's `variables` scope.

onApplicationStart() and onFuseboxApplicationStart()

You can use `onApplicationStart()` to set ColdFusion `application` scope variables but remember that method is only called (by ColdFusion) when the ColdFusion application starts up – which is not the same as the Fusebox application since Fusebox applications can be restarted programmatically at any time, independent of the ColdFusion application. To execute code when the Fusebox application is loaded, you can use the `onFuseboxApplicationStart()` method (or the previously introduced `fusebox.appinit.cfm` file or the `<appinit>` global fuseaction).

onRequest() and remote CFC calls

Note also that Fusebox's `Application.cfc` uses `onRequest()` to handle the underlying request. This means that the old code to trap requests to files other than `index.cfm` is no longer necessary. A request to any `.cfm` file is automatically routed through the framework. Also note that requests for CFCs (web services, Flex Remoting, AJAX etc) are routed through the framework up to, but not including, `onRequest()`.

This means that such calls can rely on `application`, `session` and `request` variables set in `Application.cfc` handlers, as well as any processing done in `fusebox.init.cfm` if it is present, thus sharing context between Fusebox applications and remote CFC invocations.

Additional Changes

This section of the Release Notes provides an outline of other minor enhancements in Fusebox 5.5 as well as bugs fixed since Fusebox 5.1.

Enhancements

This section lists minor enhancements in Fusebox 5.5 with ticket numbers where appropriate.

Assertions

Ticket #25. The `assertions` plugin is the official solution to this problem - and it is more powerful than the original Fusebox 4.1 approach.

See the comments in `extensions/plugins/assertions.cfm` for usage details.

conditionalParse

Ticket #75. Fusebox 4.1 supported a `conditionalParse` parameter in `fusebox.xml` that prevented parsed files being recreated if they would not have changed. This parameter was ignored in Fusebox 5.0 and 5.1. Fusebox 5.5 recognizes this parameter but treats it slightly differently to how Fusebox 4.1 treated it. In Fusebox 5.5, the parse still occurs (except in production mode) but the parsed file is only written to disk if it has changed since the last parse, thus avoiding the recompilation overhead of parsed files.

Since parsing is actually fairly fast compared to the ColdFusion compilation overhead, this should substantially improve performance of `development-circuit-load` mode and may also improve the performance of `development-full-load` mode as well.

contentVariable append

Ticket #290. Dynamic "do" now allows you to specify that the output of a fuseaction should be appended to a content variable. This is a change from the Public Beta.

Debug Trace

The debug trace output now shows a stack trace for any exceptions that are caught by the core files (rather than user-defined exception handlers). Additionally, if the core files have to rethrow the exception because it is not handled by an error template, the debug trace is rendered before the exception is thrown (previously the debug trace was suppressed). This makes it much easier to debug exceptions that are not caught within the application.

Ticket #297. Debug trace output is now generated as CSS-based output instead of using inline styles (so you can override the formatting). Thanks go to Nathan Strutz for this update!

errortemplates

Ticket #259. Historically, the `errortemplates` directory has always been part of the skeleton application. This has led to everyone having a copy of this directory in every single Fusebox application.

Fusebox 5.1 introduced the ability to override the Fusebox parameter `errortemplatesPath` and use a mapped (or webroot-relative) path, there was no longer any reason to duplicate this directory unless you wanted to modify the templates. Consequently, in Fusebox 5.5, the core files contain a master copy of the `errortemplates` directory and the skeleton applications override the `errortemplatesPath` Fusebox parameter, using `/fusebox5/errortemplates/` instead of the default local version.

If you want to customize the error templates, you should copy the core file directory into your local application and modify the templates as you see fit. Then you can just change the Fusebox parameter `errortemplatesPath` (or remove it to default to the local copy of the directory).

myFusebox.showDebug

Ticket #250. You can now suppress debug trace output on a per-request basis by setting `myFusebox.showDebug` to false at any point during the request. If debugging is not enabled in `fusebox.xml` (or via `FUSEBOX_PARAMETERS` in `index.cfm` or `Application.cfc`), this setting has no effect.

myFusebox.getApplicationData()

Ticket #190. In Fusebox 5.1, to retrieve the Fusebox application's data structure you had to say:

```
myFusebox.getApplication().getApplicationData()
```

In Fusebox 5.5, you can use:

```
myFusebox.getApplicationData()
```

as a shortcut to access that structure.

myFusebox.getOriginalCircuit() / myFusebox.getOriginalFuseaction()

Ticket #227. Convenience methods added to return the original circuit object and the original fuseaction object, mirroring the existing methods on `myFusebox`: `getCurrentCircuit()` and `getCurrentFuseaction()`.

myFusebox.variables()

Tickets #263 and #292. This is a synonym for the top-level `variables` scope in the Fusebox application. Since in the Public Beta you could no longer `<cfdump>` either `myFusebox` or the top-level `variables` scope, this has been changed to a method in the final release. You can now `<cfdump>` `myFusebox`! Code based on the Public Beta that uses `myFusebox.variables` will need to be updated.

This was previously accessible in Alpha builds through the `getTopLevelVariablesScope()` method.

self parameter

Ticket #258. The default value for the Fusebox parameter `self` has changed from `index.cfm` to the value of `CGI.SCRIPT_NAME`. This should not cause any issues but it is something to be aware of if you are relying on the default value but access Fusebox through a URL other than `index.cfm`.

SES URL processing

Ticket #252. Fusebox 5.1 introduced the ability for the `<xfa>` verb and the "myself" value to be formatted for Search Engine Safe URLs, e.g., `index.cfm/fuseaction/app.welcome` but Fusebox 5.1 did not process these URLs coming into the framework so you needed to add your own home-brewed SES URL parser. Fusebox 5.5 will attempt to parse SES URLs coming into the framework if the `queryStringStart` parameter is set to something other than the default ("?") in `fusebox.xml` or `FUSEBOX_PARAMETERS`.

Skeleton application reorganized

Ticket #268 (primarily). In Fusebox 5.1, the skeleton application had three circuits: controller (app), model (m) and views (v). With the introduction of the no-XML variant, this was confusing due to the implied circuit search order being controller, model and view (without the 's'). In order to reduce that confusion and to provide a better example of how MVC applications work, the skeleton application now has four circuits: controller (app), model/time (time), view/display (display) and view/layout (layout). Note the renaming of the views directory to view. This means that the no-XML variant can follow the structure exactly, using the same directory paths and the same circuit aliases. This should make comparisons between the two techniques much easier.

Bugs Fixed

This section lists Fusebox 5.1 and Fusebox 5.5 Alpha/Beta bugs that are fixed in Fusebox 5.5, ordered by ticket number.

194 - `circuit=` attribute on `<include>` verb now works correctly with implicit circuits.

197 - cleaned up skeleton application and moved layout prefuseaction into the controller.

198 - Element fusebox is undefined in a Java object - If your `fusebox.xml` or `circuit.xml` file contained illegal XML, you now get the correct exception.

204 - `circuit.dtd` missing step attribute on loop verb - added.

205 - clear up inconsistencies in `fusebox.dtd` - DTD now correctly says everything is optional (which it is, technically, even tho' a *useful* Fusebox application requires at least some elements be present).

212 - double-hashed CSS colors in Fusebox debug report - removed spurious # symbols.

213 - circuit aliases can include dots (.) so `fuseaction=foo.bar.action` is legal (and the circuit is `foo.bar` with a fuseaction of `action`). This is a first step to supporting drop-in modules in a future release.

232 - request timeout is automatically increased to ten minutes on framework load.

233/242 - ColdSpring get bean definition typo - custom lexicon fixed.

240 - sandbox security exceptions give a better error message.

243 - renamed `fuseboxImplicitFuse.cfc` to `fuseboxImplicitFuseaction.cfc`.

245 - plugin syntax errors give better error messages.

246 - fuseaction in URL should be trimmed - long-standing bug fixed so leading / trailing whitespace in a fuseaction value is now ignored.

247 - added `cf:savecontent` lexicon.

257 - `FUSEBOX_PARAMETERS` did not all correctly override `fusebox.xml` parameter values. This was a bug introduced in Fusebox 5.5 Alpha.

264 - `FUSEBOX_APPLICATION_PATH` did not work correctly with implicit circuits.

265 - Exceptions refer to `Application.cfc` location, not original source location.

268 - Reorganized skeleton applications' circuit to reduce confusion when comparing traditional and noxml versions.

269 - Fixed double application load after ColdFusion server is restarted.

270 - Worked around sandbox security disabling Java access by falling back to ColdFusion code.

286 - `getCanonicalPath()` now correctly handles Windows file paths.

291 - `this` scope in a traditional `Application.cfc` (that does not extend `fusebox5.Application`) is now correctly passed into context of the application.

292 - `myFusebox.variables` is now a method - `myFusebox.variables()` - so you can dump `myFusebox` and `variables` scope again.

293 - added guard code to dynamic `do()` to ensure it can't be called until its environment is setup correctly. It throws an exception if it is called inappropriately.

295 - Key Fusebox variables are now correctly exposed to a traditional `Application.cfc` (that does not extend `fusebox5.Application`).

Known Issues

This section will contain any known issues with the current Fusebox 5.5 core files.

- If a Fusebox application is running and you change any of the SES URL parameters (the ones whose names begin with `queryString`), the updated values are not reflected in the generated `myself` parameter (or `myFusebox.getMyself()`) unless you force a full reload of the framework (with `fusebox.load=true`) - even with `development-full-load mode`. This is a "fact of life" because users are allowed to override the `myself` parameter directly and the framework cannot tell whether it's calculated default should take precedence over the current `myself` parameter value.
- There may still be some situations where an exception that occurs during startup is silently "swallowed" resulting in a white screen of death. Enabling the Fusebox `debug` parameter will show the "missing" exception.