

Lesson 7 – Looking at Fusebox 4.1

At the time of writing we're currently up to Beta 2 of Fusebox 4.1 with the final gold release to be sometime soon. Fusebox 4.1 brings some really cool additions to the Fusebox developer's tool box; just a few are listed below;

- New <class> lexicon for defining classes in fusebox.xml
- New <instantiate> and <invoke> for instantiating CFCs and invoking CFC methods
- <loop> now supports looping over a query
- Fusebox.init for configuration settings
- Ability to use a single source of core files across all Fusebox4.1 applications on a single server
- The core files now throw errors of type fusebox.* that can be caught and graciously handled at runtime
- Custom lexicon – this is tabled for inclusion as a Fusebox4.2 feature but it is just so cool that it's been included in Fusebox4.1 in its experimental state.

Previously, Fusebox developers were able to use CFCs with Fusebox but as their popularity has increased it was decided to add lexicon to support so as to make the XML files cleaner to read. So, let's take a look at the new lexicon.

<class>

So, the <class> lexicon is used in the fusebox.xml file and looks something like this;

```
<classes>
  <class alias="blog" type="component"
classpath="beynon.cfcs.blog" constructor="init" />
</classes>
```

In this code snippet 'alias' is the name which will be used to refer to the class from this point on – i.e. when the class is instantiated. The 'type' attribute can be either component or webservice. The 'classpath' is a mapping using . (dot) notation to the CFC, and the constructor attribute is the name of the constructor which exists within the component.

NOTE: Beta1 and Beta2 of Fusebox4.1 would try and call a constructor called init even if the constructor attribute was not defined. As people moved to using <class> it was found that not every CFC is written with a constructor named init so the need to explicitly specify the constructor name was added. If a constructor attribute is not defined than a constructor will not be called when it's instantiated.

Once you've defined your classes to your application they will sit there doing absolutely nothing at all, that is – until you instantiate them. You do that with the `<instantiate>` grammar in your circuit.xml file.

<instantiate>

An example of `<instantiate>` would look like this;

```
<instantiate object="application.blog" class="blog"
arguments="#params#" />
```

What's this doing then? Firstly it's creating an instance of my class (blog) and it's placing it into application.blog, it's also passing parameters (in the form of a structure) into the constructor (as defined in the `<class>`). Of course you don't have to instantiate into a stateful scope as I've done here. There's also another attribute I've missed out here, 'overwrite' which takes a Boolean either true or false. If you set it to false and reinstate the component then it wouldn't be over written, instead just instantiated the once.

So now that we have defined our component via the `<class>` lexicon and we've instantiated the component we can invoke methods that exist within the component.

<invoke>

Invoke uses attributes similar to CFINVOKE that Cold Fusion uses;

```
<invoke object="application.blog"
methodcall="properties(attributes.blogID) "
returnvariable="properties" />
```

The attribute 'object' uses the same name as the object attribute in the `<instantiate>` lexicon. MethodCall as you'd expect is the name of the method to call, any attributes are passed into the method inside the brackets. Return variable is the name of the variable any results the method passes back are placed into

Other grammar changes

That concludes looking at the new Fusebox 4.1 grammar. There have been a number of changes to existing pieces of grammar. For instance you can now place the results of an `<include>` into a content variable for subsequent use. On a more minor note, the use of `<do>` inside a globalfuseaction has been deprecated, in preference of `<fuseaction>`. `<do>` will still work in Fusebox 4.1 to retain backwards compatibility but may be dropped in subsequent releases.

Changes to <loop>

Fusebox 4 did not support looping over a query, you had to use a condition and loop from 1 to the recordcount in the query, Fusebox 4.1 now introduces query looping by popular demand, simply used as;

```
<loop query="">
    <do action="foo.bar" />
</loop>
```

Fusebox.init

Let's continue by looking at the new fusebox.init file. With Fusebox 4 many developers used a plugin or a global preprocess fuseaction to load application wide variables – this was seen as common to the majority of Fusebox applications so it was decided to provide the same functionality within the Fusebox cores. If you place a fusebox.init file in the application root then it will be called just after the loader has completed. This means variables set in your fusebox.xml file are available to fusebox.init – commonly I place the following lines into fusebox.init by default;

```
<cfscript>
    // common fusebox variables
    self = "index.cfm";
    request.myself =
"#self#?#application.fusebox.fuseactionVariable#";
</cfscript>
```

So now anywhere where I would usually hardcode index.cfm?fuseaction= I would use #request.myself# just in case my boss comes to me and tells me to get rid of index.cfm and fuseaction= from URLs and instead use default.cfm and mymethod= - all I'd have to do is rename my index.cfm to default.cfm, nip into fusebox.init and change the value of self as well as changing the value of fuseactionvariable in my fusebox.xml file.

Error handling

With previous Fusebox releases it was pretty tough to gracefully catch Fusebox errors and handle them – Fusebox 4 needed a plugin but now with Fusebox 4.1 the core throws multiple types of errors which can be gracefully handled. The full list is shown below along with which of the core files throws the error;

Custom Error type

```
fusebox.missingCoreFile
fusebox.versionMismatchException

fusebox.malformedFuseaction
fusebox.undefinedCircuit
```

Core file that throws error

```
Runtime
loader,parser,transformer
Runtime
Runtime
```

| | |
|---|-------------|
| fusebox.undefinedFuseaction | Runtime |
| fusebox.invalidAccessModifier | Runtime |
| fusebox.errorWritingParsedFile | Runtime |
| fusebox.missingParsedFile | Runtime |
| fusebox.missingAppPath | Runtime |
| fusebox.missingFuseboxXML | Loader |
| fusebox.fuseboxXMLError | loader |
| fusebox.missingCircuitXML | loader |
| fusebox.circuitXMLError | Loader |
| fusebox.undefinedCircuit | transformer |
| fusebox.undefinedFuseaction | transformer |
| fusebox.overloadedFuseaction | Transformer |
| fusebox.invalidAccessModifier | Transformer |
| fusebox.missingFuse | Parser |
| fusebox.failedAssertion | Parser |
| fusebox.badGrammar | Parser |
| fusebox.badGrammar.noInvokeException | Parser |
| fusebox.badGrammar.unregisteredLexiconException | parser |
| fusebox.badGrammar.missingImplementationException | parser |

Error templates must live in a folder named `errortemplates` off the application root and are named according to the error type they are going to handle, eg a error type of `undefinedCircuit` would be handled by a file named `fusebox.undefinedCircuit.cfm` in the `errortemplates` folder.

Rather than create multiple files for each error you can create a single `fusebox.cfm` in the `errortemplates` folder which would be used as a 'best match' template. Some applications have `errortemplates` for each error type but then just `cinclude` a generic error handler file. How you handle your error handling is up to you as the developer.

Custom Lexicon

Let's end this lesson by looking at Custom Lexicon. Despite only appearing in Fusebox 4.1 in an experimental state, it really is too cool not to just say a few words about. Custom Lexicon allows developers to define their own verbs for use within `circuits.xml` files. Lexicon needs to live in a folder aptly named 'lexicon' and you need to register the lexicon folder in your `fusebox.xml` file,

```
<lexicons>
  <lexicon namespace="jb" path="jb/" />
</lexicons>
```

So now, any file placed in `lexicon/jb` will be available to me when the `circuit.xml` file is parsed, but it must be prefixed using the namespace attribute as defined in `<lexicon>`.

Why would you want to write your own lexicon? Well so far I've written two pieces of lexicon – one to allow me to wrap statements within a <cflock> block and another to wrap statements within a <cftransaction> block. Both of these pieces of lexicon are available for download on my site.

Provided you've downloaded them and placed them in a folder named jb and registered it in fusebox.xml you'd be able to write the following in your circuit.xml file;

```
<fuseaction name="initialize">
    <jb.lock mode="start" type="exclusive"
scope="application" />
    <instantiate object="application.blog" class="blog"
arguments="#params#" />
    <set name="application.initialized" value="true" />
    <jb.lock mode="end" />
</fuseaction>
```

Cool huh? Any attributes the custom lexicon takes are handled by the lexicon file itself – which then write to the parsed files, in this example writing <cflock> around the CFC instantiation being written into the application scope.

If you plan on writing your own lexicon then there are some handy UDFs in the Fusebox 4.1 cores to write output and format the code written into the parsed files, fb_appendline() , fb_increaseindent(), fb_decreaseindent(). It's always handy to look at the files that are written into the parsed folder to make sure your verb is working as you'd expect. The structure fb_.verbinfo contains useful information which your verb can make use of.