An Introduction to WireBox AOP

www.coldbox.org

Covers up to version 1.8

Contents

WireBox is a dependency injection (DI) framework that helps create objects and prepare them for use in CFML applications.

When you use WireBox to create objects, they are automatically injected with their dependencies and persisted for you. This allows you to simplify your code by giving control to the DI engine (WireBox) to do its job. If you haven't read it, check out our WireBox Ref Card here: http://wiki.coldbox.org/wiki/WireBox.cfm Once you funnel creation of objects through a central factory, there are extremely productive new design patterns that are available to you. One of those is AOP, short for Aspect Oriented Programming.

WHAT IS AOP

AOP stands for **Aspect Oriented Programming** and provides a powerful way to decouple redundant parts of your application out into separate CFCs or "aspects". This helps to simplify your code, reduces copied and pasted logic, and is easily configurable at run time. Aspects can be bound to any method you want and you will be able to add code before, after, or around the execution of that method. Think of ColdFusion's `onSessionStart()` and `onSessionEnd()` conventions, but way more powerful!

Let's look at a method that can be improved via AOP:

As you can see, this method has logging information at the start and end of its execution. It is also wrapping its

```
function save() {
    log.info("method save() called with arguments: #serializeJSON(arguments)#");
    transaction {
        // do some work here
    }
    log.info("Save completed successfully!");
}
```

logic in a transaction. The "do some work" comment is actually the only bit of real business code that is unique to this method, the rest is coding fluff we refer to as cross-cutting concerns; concerns that are not the real business logic.

With AOP we can create two generic aspects; one for logging method calls, and another for wrapping logic in a transaction and then bind those aspects to our save method or any other method in our code for that matter. Imagine taking a stamp tool and stamping that piece of code in multiple locations as you see fit.

ENABLE AOP IN WIREBOX

The AOP implementation for WireBox is completely modular and implemented entirely via external listeners. Enabling AOP is as simple as a line of config registering the AOP listener. In you WireBox config file, add the following listener:

```
listeners = [
   { class="coldbox.system.aop.Mixer" }
]
```

Now, any time WireBox creates an instance of a CFC, the AOP listener will wrap any eligible methods in that component with your aspect.

ASPECTS

An Aspect is a CFC that supplies some sort of code you want run before/after another method. The code in the Aspect is called an "advice". It advises the original method to add functionality at runtime. Let's take a look at a simple Aspect that wraps the target method in a transaction:

myAspect.cfc

```
component {
   function invokeMethod( invocation ) {
      // Let's do the around advice now:
      transaction {
         /* execute the method or other
           aspects in a transaction */
         return arguments.invocation.proceed();
      }
   }
}
```

Let's take a look at what's going on here. Our aspect has a single method called "`invokeMethod()`" that accepts an object called "invocation" which represents the original function being wrapped. The `invocation.proceed()` line is where we actually continue execution of the original method. Any logic you want can be added before and/or after the call to the original method. This allows you to specify "before" advice, "after" advice, or "around" advice.

Here's some of the methods that the "invocation" object gives you to work with:

- `getMethod()` : Get the name of the method that we are proxying
- `getArgs()` : Get the argument collection of the method call
- `setArgs()` : Override the argument collection of the method call
- `proceed()` : Tells WireBox AOP to continue to execute the target method. If you do not call this in your aspect, then the proxied method will NOT be executed.

Note: Multiple aspects can be applied to a single target method, and they will be chained together. This means invocation.proceed() may be pointing to the next aspect in line to add its advice.

The last thing we need to do with our aspect is to tell WireBox about it. To do this, we'll add a mapping to our WireBox config file that looks like this:

```
mapAspect("myAspect").to("path.to.MyAspect");
```

BINDING ASPECT

So we've created an aspect that can wrap any function in a transaction, but how do we specify when it gets run? This is done by binding our aspect with what we call component and method matchers. Here are some criteria that matchers can use to bind our aspect to a specific component and method:

- A specific component or method name
- An annotation on the target component or method
- Specific WireBox mappings ID
- A regular expression
- Any (wildcard)

Aspect bindings always have two parts; the matcher for components and the matcher for methods. If the aspect bindings never change, the easiest way to specify them is to just add the matcher information as annotations to the aspect itself. This is called an auto-binding aspect.

myAspect.cfc

```
/* Match any methods called "save" in the UserService and
   WidgetService */
component classMatcher'mappings:UserService,WidgetService'
methodMatcher='methods:save' {
   function invokeMethod() {...}
}

/* Any methods starting with "set" in any CFC whose name ends
   with "Service" */
component classMatcher'regex:Service$' methodMatcher='^set' {
   function invokeMethod() {...}
}

/* Match any method in any CFC annotated with the word
   "transactional" */
component classMatcher'any'
methodMatcher='annotatedWith:transactional' {
   function invokeMethod() {...}
}
```

You can also do these same bindings in your WireBox config. This can be a matter of preference or because you want the bindings to be dynamic based on application settings or environment. Here are the exact same aspect bindings as above, but expressed as mapping DSL in your WireBox configuration file.

```
mapAspect("myAspect").to("path.to.myAspect");

/* Match any methods called "save" in the UserService and
   WidgetService */
bindAspect(classes=match()
   .mappings('UserService,WidgetService'),
      methods=match().methods('save'),
      aspects="myAspect");

/* Any methods starting with "set" in any CFC whose name ends
   with "Service" */
bindAspect(classes=match().regex('Service$'),
   methods=match().regex('^set'),
   aspects="myAspect");
```

```
/* Match any method in any CFC annotated with the word
   "transactional" */
bindAspect(classes=match().any(),
      methods=match().annotatedWith('transactional'),
      aspects="myAspect");
```

So, let's say we went with the annotatedWith binding above. Here's an example of a method that would be automatically wrapped in a transaction:

```
function save() transactional {
   // do some work here
}
```

ADDITIONAL EXAMPLES

WireBox ships with a few simple aspects including a transactional aspect like the one we showed here. There is also a method logger aspect that captures the arguments and return data every time a method in your code gets executed.

Use your imagination for additional aspects you can create that can remove redundant code from your application:

- Threading
- Transform method results
- Method auditing
- Method security

Additional aspects can also be found on ForgeBox such as CacheBack which will automatically cache the results of a method-- all with just a little metadata.

More info: http://www.coldbox.org/forgebox/type/wirebox-aspects

The WireBox AOP framework is completely documented in our online wiki located at
http://wiki.coldbox.org/wiki/WireBox-AOP.cfm

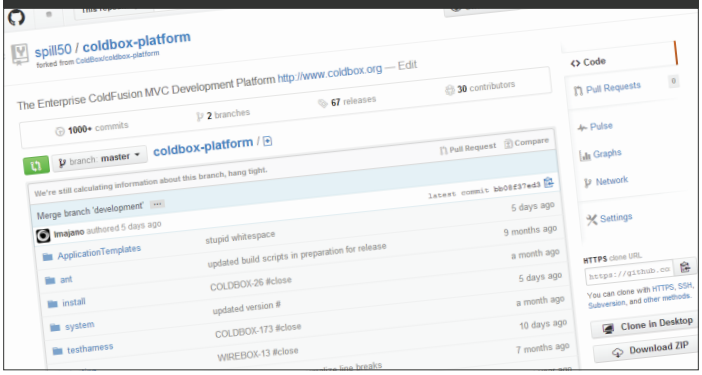
For API docs to see class definitions and method signatures, please visit the API docs located at http://www.coldbox.org/api

We have an active Google Group with hundreds of subscriber located at

http://groups.google.com/group/coldbox

Our official code repository is on GitHub. Please favorite us and feel free to fork and submit pull requests.

https://github.com/ColdBox/coldbox-platform



An Introduction to WireBox AOP

Ortus Solutions
A Software Revolution

WireBox is professional open source backed by Ortus Solutions, who provides training, support & mentoring, and custom development.

Support Program

The Ortus Support Program offers you a variety of Support Plans so that you can choose the one that best suit your needs. Subscribe to your plan now and join the Ortus Family!

With all of our plans you will profit not only from discounted rates but also receive an entire suite of support and development services. We can assist you with sanity checks, code analysis, architectural reviews, mentoring, professional support for all of our Ortus products, custom development and much more!

For more information visit
www.ortussolutions.com/services/support

Our Plans	M1	Entry	Standard	Premium	Enterprise
Price	**$199**	**$2099**	**$5699**	**$14099**	**$24599**
Support Hours	2 4 tickets	10 20 tickets	30 60 tickets	80 160 tickets	150 300 tickets
Discounted Rate	**$185/hr**	**$180/hr**	**$175/hr**	**$170/hr**	**$160/hr**
Renewal Price	**$199/month**	**$1800/year**	**$5250/year**	**$13600/year**	**$2400/year**
Phone/Online Appointments	✔	✔	✔	✔	✔
Web Ticketing System	✔	✔	✔	✔	✔
Architectural Reviews	✔	✔	✔	✔	✔
Hour Rollover		✔	✔	✔	✔
Custom Development		✔	✔	✔	✔
Custom Builds & Patches			✔	✔	✔
Priority Training Registration			✔	✔	✔
Development & Ticket Priority			✔	✔	✔
Response Times	1-5 B.D.	1-3 B.D.	1-2 B.D.	< 24 hr	< 12 hr
Books, Trainings, Product Discounts	0%	5%	10%	15%	20%
Free Books	0	0	1	3	5

COLDBOX METRICS & PROFILING

Leveraging the power of Integral's FusionReactor, fine tune, optimize and debug your ColdBox applications with absolute ease!

Find out more at www.ortussolutions.com/products/profilebox

Ortus Solutions
A Software Revolution

fusion reactor™

DESIGNED FOR

ColdBox ContentBox

www.coldbox.org | www.ortussolutions.com

First Edition - March 2014