

[<< Back to Dashboard](#) | [<< Back to WireBox](#)

WireBox AOP Engine

Covers up to version 1.3.0

Introduction

WireBox fully supports aspect-oriented programming (AOP) for ColdFusion and any ColdFusion framework. Just note the different namespaces if using within the ColdBox Platform and standalone WireBox.

Namespaces

```
// ColdBox
coldbox.system.aop

// WireBox Standalone
wirebox.system.aop
```

Requirements

- ColdFusion 8 and above
- Railo 3.1 and above
- Disk/Memory Generation

Overview

In computing, aspect-oriented programming (AOP) is a programming paradigm which aims to increase modularity by allowing the separation of cross-cutting concerns. AOP forms a basis for aspect-oriented software development.

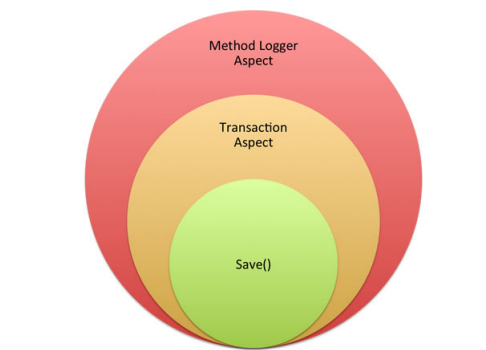
I won't go into incredible software theory but pragmatic examples. What is the value of AOP? What does it solve? Well, how many times have we needed to do things like this:

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

As you can see from the example above, my real business logic is in the `//do some work here` comment, but our code is littered with logging and transactions. What if I have 10 methods that are very familiarly the same? Do I repeat this same littered code ([cross-cutting concerns](#))? The answer for most of us has always been "YES! Of Course! How else do I do this?". Well, you don't have to, AOP to the rescue. What AOP will let you do is abstract all that logging and transaction code to another object usually called an Aspect. Then we need to apply this aspect to something right? Well, in our case it has to apply to a target object, `out/serService`, and to a specific method, `save()`, which is usually referred to as the **join point**.

What does applying an aspect mean? It means that we will take that aspect you write and execute it at different points in time during the execution of the method you want to apply it to. This is usually refer to as an advice, "Hey Buddy! Run it here!!!". There are multiple types of AOP advices like before, around, after, etc. We have seen in ColdBox event handlers that you can do `preHandler`, `postHandler` and `aroundHandler` methods. These are AOP advices localized to event handlers that let you execute code before a handler event, after a handler event, or completely around the handler event. The most powerful form of advice is around, as it allows you to completely surround a method call with your own custom code. Does it ring a bell now? The transaction code for Pete's Sake! You need it to completely surround the method call! Voila! The around advice will allow you to completely take over the execution and you can even determine if you want to continue the execution or not.

So how does this magic happen? Well, our WireBox AOP engine will hijack your method (join point) and replace it with a new one, usually called an AOP proxy. This new method has all the plumbing already to allow you to apply as many aspects you like to that specific method. So as you can see from our diagram below, the `save` method is now decorated with our two aspects, but for all intent and purposes the outside world does not care about it, they just see `save()` method.



AOP Vocabulary

- **Aspect**: A modularization of a concern that cuts across multiple objects.
- **Target Object**: The object that will be applied with Aspects across certain methods or join points.
- **Join Point**: A point of execution in a target object that will be applied a specific aspect to it. This is usually the execution of a method.
- **Advice**: An action taken at a particular join point. Usually, before, after or around it.
- **AOP Proxy**: An object or method representation for the original join point or method.

Activate the AOP Listener

WireBox has an amazing [event driven architecture](#) that can help you modify, listen and do all kinds of magic during object creation, wiring, etc. Our AOP implementation is just a listener that will transform objects once they are finalized with dependency injection. This means, our AOP engine is completely decoupled from the internals of the DI engine and is incredibly fast and light weight. So let's activate it in our WireBox binder configuration:

```
wirebox.listeners = [
    { class="coldbox.system.aop.Mixer", properties={} }
];
```

That's it! That tells WireBox to register the AOP engine once it loads. This listener also has some properties that you can tweak:

Property	Type	Required	Default Value	Description
<code>generationPath</code>	cf include path	false	<code>/coldbox/system/aop/tmp</code>	The location where UDF stubs will be generated to. This can be to disk or memory.
<code>classMatchReload</code>	boolean	false	false	A cool flag to allow you to reload the class matching dictionary for development purposes only.

Create Your Aspect

Now that we have activated the AOP engine, let's build a simple method logger aspect that will intercept before our method is called and after our method is called. So if you remember your AOP dictionary terms, we will create an aspect that does a before and after advice on the method. Phew! To do this we must implement a CFC that WireBox AOP gives you as a `template:coldbox.system.aop.MethodInterceptor`. This CFC interface looks like this:

```
<cfinterface hint="Our AOP Method Interceptor Interface">
    <--- invokeMethod --->
    <cffunction name="invokeMethod" output="false" access="public" returnType="any" hint="Invoke an AOP method invocation">
        <cfargument name="invocation" required="true" hint="The method invocation object: coldbox.system.ioc.aop.MethodInvocation">
    </cffunction>
</cfinterface>
```

This means, that we must create a CFC that implements the `invokeMethod` method with our own custom code. It also receives 1 argument called `invocation` that maps to a CFC called `coldbox.system.aop.MethodInvocation` that you can learn from our cool [AOP](#). Our approach to AOP is simplicity, therefore this `invokeMethod` implements the most powerful advice called around advice, so you

Contents

- [WireBox AOP Engine](#)
 - [Introduction](#)
 - [Namespaces](#)
 - [Requirements](#)
 - [Overview](#)
 - [AOP Vocabulary](#)
 - [Activate the AOP Listener](#)
 - [Create Your Aspect](#)
 - [MethodInvocation Useful Methods](#)
 - [MethodLogger Aspect](#)
 - [Aspect Registration](#)
 - [Aspect Binding](#)
 - [Auto Aspect Binding](#)
 - [ClassMatcher Annotation DSL](#)
 - [MethodMatcher Annotation DSL](#)
 - [Included Aspects](#)
 - [CFTransaction](#)
 - [HibernateTransaction](#)
 - [MethodLogger](#)
 - [Summary](#)

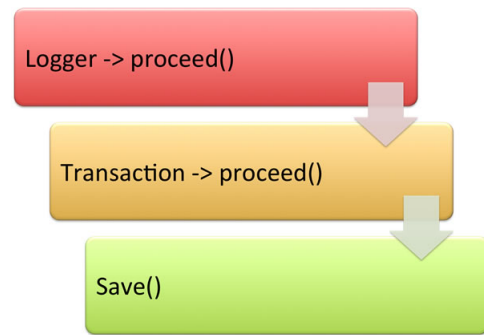
can always will do an around advice, but it will be up to your custom code to decide what it does before (beforeAdvice), around (aroundAdvice) and after (afterAdvice) the method call. The other advantage of WireBox AOP aspects is that once they are registered with WireBox they act just like normal DI objects in WireBox, therefore you can apply any type of dependency injection to them.

MethodInvocation Useful Methods

Here are a list of the most useful methods in this CFC

- `getMethod()`: Get the name of the method (join point) that we are proxying and is being executed
- `getMethodMetadata()`: Get the metadata structure of the current executing method. A great way to check for annotations on the method (join point)
- `getArgs()`: Get the argument collection of the method call
- `setArgs()`: Override the argument collection of the method call
- `getTarget()`: Get the object reference to the target object
- `getTargetName()`: Get the name of the target object
- `getTargetMapping()`: Get the object mapping of the proxied object. Great for getting metadata information about the object, extra attributes, etc.
- `proceed()`: This is where the magic happens, this method 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.

The last method is the most important one as it tells WireBox AOP to continue executing either more aspects, the proxied method or nothing at all.



MethodLogger Aspect

Here is my `MethodLogger` aspect that I will create:

```

<cfcomponent output="false" implements="coldbox.system.aop.MethodInterceptor" hint="A simple interceptor that logs method calls and their results">
  <!-- Dependencies -->
  <cfproperty name="log" inject="logbox:logger:{this}">
  <!-- init -->
  <cffunction name="init" output="false" access="public" returntype="any" hint="Constructor">
    <cfargument name="logResults" type="boolean" required="false" default="true" hint="Do we log results or not?"/>
    <cfscript>
      instance = {
        logResults = arguments.logResults
      };
      return this;
    </cfscript>
  </cffunction>
  <!-- invokeMethod -->
  <cffunction name="invokeMethod" output="false" access="public" returntype="any" hint="Invoke an AOP method invocation">
    <cfargument name="invocation" required="true" hint="The method invocation object: coldbox.system.aop.MethodInvocation">
    <cfscript>
      var refLocal = {};
      var debugString = "target: #arguments.invocation.getTargetName()#,method: #arguments.invocation.getMethod()#,arguments:#serializeJSON(arguments.invocation.getArgs())#";
      // log incoming call
      log.debug(debugString);
      // proceed execution
      refLocal.results = arguments.invocation.proceed();
      // result logging and returns
      if( structKeyExists(refLocal,"results") ){
        if( instance.logResults ){
          log.debug("#debugString#, results:", refLocal.results);
        }
        return refLocal.results;
      }
    </cfscript>
  </cffunction>
</cfcomponent>
  
```

You can see that I do some DI via annotations:

```

<!-- Dependencies -->
<cfproperty name="log" inject="logbox:logger:{this}">
  
```

A normal constructor with one optional argument for logging results:

```

<!-- init -->
<cffunction name="init" output="false" access="public" returntype="any" hint="Constructor">
  <cfargument name="logResults" type="boolean" required="false" default="true" hint="Do we log results or not?"/>
  <cfscript>
    instance = {
      logResults = arguments.logResults
    };
    return this;
  </cfscript>
</cffunction>
  
```

And our `invokeMethod` implementation:

```

<!-- invokeMethod -->
<cffunction name="invokeMethod" output="false" access="public" returntype="any" hint="Invoke an AOP method invocation">
  <cfargument name="invocation" required="true" hint="The method invocation object: coldbox.system.aop.MethodInvocation">
  <cfscript>
    var refLocal = {};
    var debugString = "target: #arguments.invocation.getTargetName()#,method: #arguments.invocation.getMethod()#,arguments:#serializeJSON(arguments.invocation.getArgs())#";
    // log incoming call
    log.debug(debugString);
    // proceed execution
    refLocal.results = arguments.invocation.proceed();
    // result logging and returns
    if( structKeyExists(refLocal,"results") ){
      if( instance.logResults ){
        log.debug("#debugString#, results:", refLocal.results);
      }
      return refLocal.results;
    }
  </cfscript>
</cffunction>
  
```

As you can see, the before advice part is what happens before the execution of the real method (or more aspects) occurs. So everything before the call to `arguments.invocation.proceed()`:

```

var refLocal = {};
var debugString = "target: #arguments.invocation.getTargetName()#,method: #arguments.invocation.getMethod()#,arguments:#serializeJSON(arguments.invocation.getArgs())#";
// log incoming call
log.debug(debugString);
  
```

Then we execute the real method or more aspects (we do not do anything around the method call):

```
// proceed execution
refLocal.results = arguments.invocation.proceed();
```

Finally, we do the after advice part which happens after the method or other aspects fire and results are returned:

```
// result logging and returns
if( structKeyExists(refLocal,"results") ){
    if( instance(logResults) ){
        log.debug("#debugString#, results:", refLocal.results);
    }
    return refLocal.results;
}
```

That's it. I have successfully created an aspect. What's next!

Aspect Registration

Now that we have built our aspect we need to register it with WireBox so it knows about it and all DI can be performed in it. Let's open our WireBox binder and use the following DSL method:

```
•mapAspect(aspect,autoBinding={true})
mapAspect("MethodLogger").to("model.aspects.MethodLogger");
```

This tells WireBox to register a new aspect called **MethodLogger** that points to the CFC **model.aspects.MethodLogger** that I have just built. WireBox will then mark that object as an aspect, create it once the injector loads and have it ready for building.

Aspect Binding

Since we have now mapped our aspect in WireBox, we now need to tell it the most important things:

- 1.. To what classes or CFCs should we apply this aspect to?
- 2.. To what methods or join points should we apply this aspect to?

We do this with another binder DSL method:

```
•bindAspect(classes,methods,aspects)
bindAspect(classes=match().mappings('UserService'),methods=match().methods('save'),aspects="MethodLogger");
```

What is up with that funky *match()*.... stuff? Well, the *classes* and *methods* arguments of the *bindAspect()* call uses our AOP funky matcher methods that exist in an object called: **coldbox.system.aop.Matcher**. This matcher object is used to match classes and methods to whatever criteria you like. The binder has a convenience method called **match()** that creates a new matcher object for you and returns it so you can configure it for classes and method matching. Here are the most common matching methods below:

Method	Class Matching	Method Matching	Description
any()	true	true	Matches against any class path or method name
returns(type)	false	true	Matches against the return type of methods
annotatedWith(annotation,[value])	true	true	Matches against the finding of an annotation in a cfcomponent or cffunction or matches against the finding AND value of the annotation.
mappings(mappings)	true	false	Matches to ONLY the named mapping(s) you pass to this method as a list or array.
instanceOf(classPath)	true	false	Matches if the target object is an instance of the <i>classPath</i> . This internally uses the ColdFusion <i>isInstanceOf()</i> method
regex(regex)	true	true	Matches against a CFC instantiation path or function name using regular expressions
methods(methods)	false	true	Matches against a list of explicit method names as a list or array
andMatch(matcher)	true	true	Does an AND evaluation with the current matcher and the one you pass in the method.
orMatch(matcher)	true	true	Does an OR evaluation with the current matcher and the one you pass in the method.

WOW! We can really pin point anything in our system! So now that we have binded our aspect to our *UserService* I can rewrite it to this:

```
component name="UserService"{
    function save(){
        transaction {
            // do some work here
        }
    }
}
```

Now isn't that pretty? Much nicer and compact huh! Plus I can reuse the method logger for ANY method in ANY class I desire (Muuaaahaaaa), that's an evil laugh by the way! But we are not done, let's keep going and do the Transactional aspect.

Auto Aspect Binding

WireBox AOP supports the concept of self aspect bindings. This means that we can make things even simpler by letting the Aspect you build control what class and methods it will match against. This is great, one less thing to remember when developing the code. So let's start with the code first:

```
/**
 * @classMatcher any
 * @methodMatcher annotatedWith:transactional
 */
component name="TransactionAspect" implements="coldbox.system.aop.MethodInterceptor"{
    function init(){ return this; }

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

That's it! Our transaction aspect is done and it will also bind itself to ANY class and ANY method that has the transactional annotation. Then you just map it:

```
mapAspect("TransactionAspect").to("model.aspects.MyTransactionAspect");
```

We are done now, by mapping the aspect WireBox detects the two annotations **classMatcher** and **methodMatcher** and binds it for you. WOW, but where did you get that from? Well, the component has two annotations:

```
/**
 * @classMatcher any
 * @methodMatcher annotatedWith:transactional
 */
OR
component classMatcher="any" methodMatcher="annotatedWith:transactional"{}
```

How cool is that! My aspect can determine the matching for me already. So what can I use for these matchers:

ClassMatcher Annotation DSL

Create a **classMatcher** annotation on the component with the following DSL values:

DSL	Description
any	Matches against any class path or method name
annotatedWith:{annotation}	Matches against the finding of an annotation in a cfcomponent
annotatedWith:{annotation}:{value}	Matches against the finding of an annotation value in a cfcomponent
mappings:{mappings}	Matches to ONLY the named mapping(s) you pass to this method as a list or array.

instanceOf:{classPath}	Matches if the target object is an instance of the <i>classPath</i> . This internally uses the ColdFusion <i>isInstanceOf()</i> method.
regex:{regex}	Matches against a CFC instantiation path or function name using regular expressions

MethodMatcher Annotation DSL

Create a **methodMatcher** annotation on the component with the following DSL values:

DSL	Description
any	Matches against any class path or method name
annotatedWith:{annotation}	Matches against the finding of an annotation in a ccomponent
annotatedWith:{annotation}:{value}	Matches against the finding of an annotation value in a ccomponent
returns:{type}	Matches to ONLY the methods that return the {type}
methods:{methods}	Matches to ONLY the named methods(s) you pass to this method as a list or array.
regex:{regex}	Matches against a CFC instantiation path or function name using regular expressions

One thing to note about self binding aspects is that you can also override their matching by using the **autoBind** argument in the *mapAspect()* method call. So if you wanted to override the class and method matching on this aspect you would do this:

```
mapAspect(aspect="TransactionAspect", autoBind=false).to("model.aspects.MyTransactionAspect");
// match only methods that start with the regex ^save
bindAspect(classes=match().any(), methods=match().regex("^save"), aspects="TransactionAspect");
```

Included Aspects

WireBox comes bundled with three aspects that you can use in your applications and can be found in *coldbox.system.aop.aspects*

Aspect	Description
CFTransaction	A simple ColdFusion transaction Aspect for WireBox
HibernateTransaction	A cool annotation based Transaction Aspect for WireBox
MethodLogger	A simple interceptor that logs method calls and their results

CFTransaction

This aspect is a self-binding aspect that will surround methods with simple ColdFusion transaction blocks.

ClassMatcher	MethodMatcher
<i>any</i>	<i>annotatedWith:transactional</i>

To use, just declare in your binder, overriding the self-binding is totally optional.

```
mapAspect("CFTransaction").to("coldbox.system.aop.aspects.CFTransaction");
```

HibernateTransaction

This aspect is a self-binding aspect that will surround methods with native Hibernate transaction blocks.

ClassMatcher	MethodMatcher
<i>any</i>	<i>annotatedWith:transactional</i>

To use, just declare in your binder, overriding the self-binding is totally optional.

```
mapAspect("CFTransaction").to("coldbox.system.aop.aspects.HibernateTransaction");
```

MethodLogger

This aspect enables you to log method calls and their results. You will have to bind it to the methods you want it to listen to. It has one constructor argument:

Name	Type	Required	Default	Description
logResults	boolean	false	true	Whether to log the results of method calls or not.

```
// Map the Aspect
mapAspect("MethodLogger")
.to("coldbox.system.aop.aspects.MethodLogger")
.initArg(name="logResults", value="true or false");

// Bind the Aspect to all service class methods
bindAspect(classes=matcher().regex(".*Service"), methods=matcher().any(), aspects="MethodLogger");
```

Summary

So now that we mapped our self binding aspect, let's clean up our code further:

```
component name="UserService"{
    function save() transactional{
        // do some work here
    }
}
```

Now isn't this amazing! We are back to our original business logic and code with no extra fluff around logging, transactions and pretty much just code noise. Now we are talking and cooking with AOP. So hopefully by now you have a good grasp of what AOP does for you and how to implement it in WireBox. So here are the steps to remind you:

1. Create an aspect that implements: **coldbox.system.aop.MethodInterceptor**
2. Map the aspect in your WireBox binder
3. Bind the aspect to classes and methods

So, easy as 1-2-3! AOP is powerful and extremely flexible. Hopefully, your imagination is now going into overdrive and coming up with ways to not only clean up your code from cross cutting concerns, but also help you do things like:

- threaded methods
- transform method results
- method auditing
- method security
- cache method results
- etc.