

[← Back to Dashboard](#)

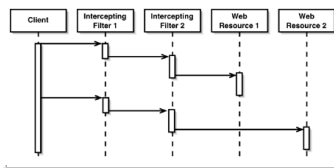
ColdBox Interceptor's Guide

Contents

Covers up to version 3.5.0

Introduction

The main purpose of creating ColdBox interceptors is to increase functionality for applications and framework alike, without touching the core functionality or events, and thus encapsulating logic into separate decoupled objects. This pattern wraps itself around a request in specific execution points in which it can process, pre-process, post-process and redirect requests. These interceptors can also be stacked to form interceptor chains that can be executed implicitly for you. These stacked interceptor chains form a chain of separate, declarative-deployable services to an existing web application or framework without incurring any changes to the main application or framework source code. This is a powerful feature that can help developers and framework contributors share and interact with their work. (Read more on [Intercepting Filters](#))

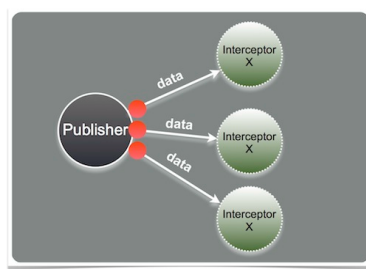


ForgeBox Interceptors

Please remember that in our ColdBox developer community site, [ForgeBox](#), you can find a big collection of [Interceptors](#) and so much more. Anybody can contribute and automatically all contributions will be accessible via our ColdBox application templates and via ColdFusion Builder.

Event-Driven Programming

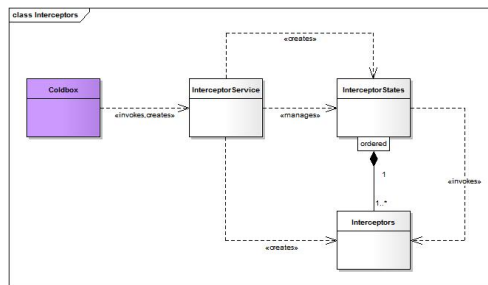
However, we went a step further with ColdBox interceptors and created the hooks necessary in order to implement an event-driven programming pattern into the entire interceptor service. Ok ok, what does this mean? It means, that you are not restricted to the pre-defined interception points that ColdBox provides, **you can create your own WOW!** Really? Yes, you can very easily declare execution points via the configuration file or register at runtime, create your interceptors with the execution point you declared (Conventions baby!?) and then just announce interceptions in your code via the interception API.



If you are familiar with design patterns, custom interceptors can give you an implementation of observer/observable objects, much like any event-driven system can provide you. If you are not familiar with event-driven systems or what an observer is, please read the following resources. In a nutshell, an observer is an object that is registered to listen for certain types of events, let's say as an example `onError` is a custom interception point and we create a CFC that has this `onError` method. Whenever in your application you announce or broadcast that an event of type `onError` occurred, this CFC will be called by the ColdBox interceptor service.

Resources

- https://en.wikipedia.org/wiki/Observer_pattern
- http://www.cmcrossroads.com/design_patterns/observer/
- <http://java.sun.com/blueprints/control2patterns/Patterns/InterceptingFilter.html>



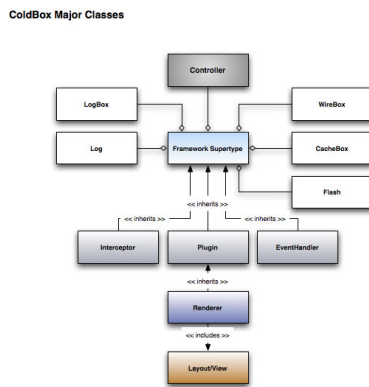
For what can I use them

Below are just a few applications of ColdBox Interceptors:

- Security
- Event based Security
- Method Tracing
- AOP Interceptions
- Publisher/Consumer operations
- Implicit chain of events
- Content Appending or Pre-Pending
- View Manipulations
- Custom SES support
- Cache advices on insert and remove
- Much much more...

How do they work?

Interceptors are CFC's that can either extend or not our ColdBox base class: `coldbox.system.Interceptor`, implement a configuration method called `configure` and then create as many methods as events it will listen to. All interceptors are treated as `singletons` in your application, so make sure they are thread safe and var scoped.



The interceptor has one important method that you can use for configuration, called `configure()`. This method will be called right after the interceptor gets created and injected with your configuration file properties.

These properties are local to the interceptor only!

As you can see on the diagram, the interceptor class is part of the ColdBox framework super type family, and thus inheriting the functionality of the framework. (See [ADP](#)).

```
/**
 * My Interceptor
 */
component {
    function configure() {}
    function preProcess(event, interceptData) {}
}
```

Common Property Methods

- `getProperty(name)` : Get a single property
- `setProperty(name, value)` : Set a property
- `propertyExists()` : Verify that a property exists
- `getProperties()` : Get the entire properties struct
- `setProperties()` : Set the entire properties struct

Conventions

What about the interception points that you keep mentioning. Well, below we will cover all of the interception points that ColdBox provides for you. However, the main convention on interceptors is that you will have to create a method with the same name as the interception point you would like to listen to. If you want it to listen to the `preProcess` point, you will create a `preProcess(event, interceptData)` method. What else? Well, this method has a return type of boolean and two arguments. So let's explore their rules.

- The interceptor method must have the same name as the interception point it is trying to listen to
- This method must accept two arguments
 - `event` which is the request context object
 - `interceptData` which is a structure of data that the broadcaster sends
- This method must return a boolean value or none at all.
 - `True` means break the chain of execution
 - `False` or `void` continue execution

The return boolean variable is very important because it tells the interceptor service whether to continue executing interceptors in the chain or not. Ok, what chain, well you can have as many interceptor objects listening to the same interception point. So if interceptors are declared like so:

- `interceptor X - preProcess`
- `interceptor Y - preProcess`
- `interceptor Z - preProcess`

Let's say that `interceptor Y` returns a `true` from the `preProcess` method, then `interceptor Z` will never get executed, the chain is broken now.

```
component extends=coldbox.system.interceptor {
    function configure() {}
    boolean function afterConfigurationLoad(event, interceptData) {
        if !getProperty('interceptorComplete') eq false {
            parameterize();
            setProperty('interceptorComplete', true);
        }
        return false;
    }
}
```

Also remember that all interceptors are created by [WireBox](#), so you can use dependency injection, configuration binder's, and even [ADP](#) on interceptor objects. Here is a more complex sample:

HTTP Security Example

```
/**
 * Intercepts with HTTP Basic Authentication
 */
component {
    // Security Service
    property name=securityService inject=id:SecurityService

    void function configure() {
        if !propertyExists('enabled') {
            setProperty('enabled', true);
        }
    }

    void function preProcess(event, struct interceptData) {
        // verify turned on
        if !getProperty('enabled') { return }

        // Verify Incoming Headers to see if we are authorizing already or we are already Authorized
        if !securityService.isLoggedin() OR len( event.getHeader('http') ) {

            // Verify incoming authorization
            var credentials = event.getHTTPBasicCredentials();
            if securityService.authorize(credentials.username, credentials.password) {
                // we are secured woot woot!
                return;
            };

            // Not secure!
            event.setHTTPHeader('WWW-Authenticate', 'Basic realm=Please enter your username and password for our Cold App!');

            // secured content data and skip event execution
            event.renderData(data='Unauthorized Access<br>Content Requires Authentication', statusCode=401, statusText='Unauthorized', noExecution());
        }
    }
}
```

eventpattern annotation

We have one annotation you can use on any interceptor event function called `eventPattern`. This annotation will be placed in the function and the value is a regular expression that the interceptor service uses to match against the incoming event. If the regex matches, the interception function executes, else it skips it. This is a great way for you to create interceptors that only fire not only for the specific interception point you want, but also on the specific incoming event.

```
// only execute for the admin package
void function preProcess(event, struct interceptData) eventPattern={}

// only execute for the blog module
void function preProcess(event, struct interceptData) eventPattern={blog}

// only execute for the blog module and the home handler
void function preProcess(event, struct interceptData) eventPattern={blog|home}

// only execute for the blog, forums, and shop modules
void function preProcess(event, struct interceptData) eventPattern={blog|forum|shop};
```

Interceptor Declaration

Interceptors can be declared in the [Configuration CFC](#) or programmatically at runtime. If you declare them in the configuration file, then you have control in the order in which they fire. If you register interceptors programmatically, you won't have control of order of execution. Interceptors are declared as an array of structures in an element called `interceptors`. The elements of the structure are:

- `class` - The instantiation path of the CFC.
- `name` - An optional unique name for the interceptor, if not passed then the name of the CFC will be used. We highly encourage the use of a name to avoid collisions.
- `properties` - A structure of configuration properties for the interceptor

Remember that order is important!

Configuration File

```
interceptors = [
    {class=coldbox.system.interceptors.interceptor, properties={}},
    {class=interceptors.Request, properties={trim=true}},
    {class=coldbox.system.interceptors.Security, properties={
        rulesSource 'model';
        rulesModel 'securityRuleService.cfc';
        rulesModelMethod 'getSecurityRules';
        validatorModel 'securityService.cfc';
    }}
];
```

Programmatically

You can also register CFCs as interceptors programmatically by talking to the application's `Interceptor Service`

```
// via controller
controller.getInterceptorService()

// via injection
property name=eventManager inject=coldbox:InterceptorService
```

Once you have a handle on the interceptor service you can use the following methods to register interceptors:

- `registerInterceptor()` - Register an instance or CFC path and discover all events it listens to by conventions
- `registerInterceptionPoint()` - Register an instance in a specific event only

Here are the method signatures:

```
public any registerInterceptor([any interceptorClass], [any interceptorClass], [any interceptorProperties], [runtime expression], [any customPoint], [any interceptorName])
public any registerInterceptionPoint(any interceptorKey, any state, any oInterceptor)
```

Here are a few samples of objects that can register themselves:

```
// register yourself to listen to all events declared
controller.getInterceptorService()
    .registerInterceptor(interceptorObject);

// register yourself to listen to all events declared and register new events: onError, onLogin
controller.getInterceptorService()
    .registerInterceptor(interceptorObject, customPoints, 'onError, onLogin');

// Register yourself to listen to ONLY the afterInstanceAutowire event
controller.getInterceptorService()
    .registerInterceptionPoint(interceptorObject, 'state:afterInstanceAutowire', 'onInterceptOnly');
```

Custom Events

As we discussed before, you can also define your own events for your application. You can do this via the [Configuring CFC](#) or programmatically as well.

Instructional Video

Configuration Registration

In the [Configuration CFC](#) file, there is a structure called `interceptorSettings` with two keys:

- **throwOnInvalidStates** - This tells the interceptor service to throw an exception if the state announced for interception is not valid or does not exist. By default it does not throw an exception but ignore the announcement.
- **customInterceptionPoints** - This key is a comma delimited list of custom interception points you will be registering for execution.

```
//Interceptor Settings
interceptorSettings = {
    throwOnInvalidStates: false,
    customInterceptionPoints: 'onLogin, onWikiTranslation, onAppClose'
};
```

Programmatic Registration

You can use the interceptor service to register new events or interception points via the `appendInterceptionPoints()` method:

```
public any appendInterceptionPoints(any customPoints)
```

The value of the `customPoints` argument can be a list or an array of interception points to register so the interceptor service can manage them for you:

```
controller.getInterceptorService()
    .appendInterceptionPoints('log', 'onError', 'onRecordInsertId');
```

Usage

Once your custom interception or event points are registered and CFC are registered then you can write the methods for listening to those events:

```
component {
    function onLog(event, interceptData) {
        // your code here
    }
    function onRecordInserted(event, interceptData) {
        // your code here
    }
}
```

Announcing Interceptions

That is awesome, I declared them and then I code them, but how do they get executed? Well, there is a special method called `announceInterception()` that all your handlers, plugins and even the interceptors themselves receive via the framework super type. You can call this method with some data and all events listening will be dispatched:

```
<!-- prepare interception structure -->
<cfset var data = structNew()
<cfset data.timeIntercepted = now()
<cfset data.user = event.getValue('user')
<cfset data.event = event.getCurrentEvent()

<!-- Broadcast Interception -->
<cfset announceInterception(log, data)

<!-- Alternate Broadcast Interception Syntax -->
<cfset getController().getInterceptorService().processState(state, interceptData)
```

As you can see from the sample below, you can alternatively announce via the interceptor service's `processState()` method as well. You can easily inject the interceptor service in any model object and even produce announcements from your domain objects.

- `announceInterception(state, interceptData)` inherited by plugins, handlers and interceptors.
- `getController().getInterceptorService().processState(state, interceptData)` via the controller.

Core Interception Points

There are many interception points that occur during an MVC and Remote life cycle in a ColdBox application. We highly encourage you to follow our flow diagrams in our [Roamster lifecycle](#) guide so you can see where in the life cycle these events fire. Also remember that each event can pass a data structure that can be used in your application. So always look at the `interceptData` structure for elements you can use.

Application Life Cycle

Interception Point	Intercept Structure	Description
afterConfigurationLoad	---	This occurs after the framework loads and your applications' configuration file is read and loaded. An important note here is that your application aspects have not been configured yet: bug reports, ioc plugin, validation, logging, and internationalization.
afterAspectsLoad	---	This occurs after the configuration loads and the aspects have been configured. This is a great way to intercept on application start.
preReinit	---	This occurs every time the framework is re-initialized
onException	exception - The <i>cfcatch</i> exception structure	This occurs whenever an exception occurs in the system. This event also produces data you can use to respond to it.
onRequestCapture	---	This occurs once the FORM/URL variables are merged into the request collection but before any event caching or processing is done in your application. This is a great event to use for incorporating variables into the request collections, altering event caching with custom variables, etc.
onInvalidEvent	<ul style="list-style-type: none"> ● invalidEvent - The invalid event ● ehBean - The event handler bean object you can use to override the event ● override - A flag that allows you to override the invalid event into an event of your choice 	This occurs once an invalid event is detected in the application. This can be a missing handler or action. This is a great interceptor to use to alter behavior when events do not exist or to produce 404 pages.
applicationEnd	---	This occurs whenever the Application ends
sessionStart	<i>session</i> structure	This occurs when a user's session starts
sessionEnd	<ul style="list-style-type: none"> ● sessionReference - A reference to the session structure ● applicationReference - A reference to the application structure 	This occurs when a user's session ends
preProcess	---	This occurs after a request is received and made ready for processing. This simulates an on request start interception point. Please note that this interception point occurs before the request start handler.
preEvent	<ul style="list-style-type: none"> ● processedEvent - The event that is about to be executed ● eventArguments - A structure of arguments (if any) the event got executed with 	This occurs before ANY event execution, whether it is the current event or called via the run event method. It is important to note that this interception point occurs before the preHandler convention. (See Event Handler Guide)
postEvent	<ul style="list-style-type: none"> ● processedEvent - The event that is about to be executed ● eventArguments - A structure of arguments (if any) the event got executed with 	This occurs after ANY event execution, whether it is the current event or called via the run event method. It is important to note that this interception point occurs after the postHandler convention (See Event Handler Guide)
postProcess	---	This occurs after rendering, usually the last step in an execution. This simulates an on request end interception point.
preProxyResults	{proxyResults}	This occurs right before any ColdBox proxy calls are returned. This is the last chance to modify results before they are returned.

Object Creation Events

Interception Point	Intercept Structure	Description
afterHandlerCreation	<ul style="list-style-type: none"> ● handlerPath - The path of the handler ● handler - The handler instance 	This occurs whenever a handler is created
afterInstanceCreation	<ul style="list-style-type: none"> ● mapping - The object mapping ● target - The target object ● injector - The Warehouse injector that produced the object 	This occurs whenever a warehouse object is created

afterPluginCreation	<ul style="list-style-type: none"> ● pluginPath - The path to the plugin ● custom - A custom or core plugin ● module - A module plugin ● plugin - The plugin object 	This occurs whenever a plugin object is created
----------------------------	---	---

[WireBox](#) also announces several other events in the object creation life cycles, so please see [the WireBox events](#).

Layout-View Events

Interception Point	Intercept Structure	Description
preLayout	---	This occurs before any rendering or layout is executed
preRender	{renderedContent}	This occurs after the layout+view is rendered and this event receives the produced content
postRender	---	This occurs after the content has been rendered to the buffer output
preViewRender	<ul style="list-style-type: none"> ● view - The name of the view to render ● cache - If the view will be cached ● cacheTimeout - The cache timeout ● cacheLastAccessTimeout - The idle timeout of the view ● cacheSuffix - A suffix to append to the cacheable key name ● module - The module name of the view if any ● args - The arguments to pass into the view ● collection - The collection this view will iterate on ● collectionAs - The alias of the collection name ● collectionStartRow - The start row of the collection iteration ● collectionMaxRows - The max rows of the collection iteration ● collectionDelim - The delimiter used for the collection iteration 	This occurs before any view is rendered
postViewRender	All of the data above plus: <ul style="list-style-type: none"> ● renderedView - The view contents that was rendered 	This occurs after any view is rendered and passed the produced content
preLayoutRender	<ul style="list-style-type: none"> ● layout - The name of the layout to render ● view - The name of the view to render if any ● module - The module name of the layout if any ● args - The arguments to pass into the layout + view (if any) ● viewModule - The name of the module the view will be rendered from 	This occurs before any layout is rendered
postLayoutRender	Everything above plus: <ul style="list-style-type: none"> ● renderedLayout - The layout + view contents that was rendered 	This occurs after any layout is rendered and passed the produced content

Module Events

Interception Point	Intercept Structure	Description
preModuleLoad	{moduleLocation, moduleName}	This occurs before any module is loaded in the system
postModuleLoad	{moduleLocation, moduleName, moduleConfig}	This occurs after a module has been loaded in the system
preModuleUnload	{moduleName}	This occurs before a module is unloaded from the system
postModuleUnload	{moduleName}	This occurs after a module has been unloaded from the system

Debugger Events

Interception Point	Intercept Structure	Description
beforeDebuggerPanel	---	This occurs before the ColdBox debugger panel
afterDebuggerPanel	---	This occurs after the ColdBox debugger panel

ORM Events

These events are re-broadcasted from hibernate and the ColdBox ORM services

Interception Point	Intercept Structure	Description
ORMPostNew	{entity}	Called via the postNew() event
ORMPreLoad	{entity}	Called via the preLoad() event
ORMPostLoad	{entity}	Called via the postLoad() event
ORMPostDelete	{entity}	Called via the postDelete() event
ORMPreDelete	{entity}	Called via the preDelete() event
ORMPreUpdate	{entity,oldData}	Called via the preUpdate() event
ORMPostUpdate	{entity}	Called via the postUpdate() event
ORMPreInsert	{entity}	Called via the preInsert() event
ORMPostInsert	{entity}	Called via the postInsert() event

CacheBox Events

Our caching engine, [CacheBox](#), also announces several events during its life cycle, so please see [The CacheBox Events section](#).

Interceptor Output Buffer

Every interceptor has the following methods that enable you to add content to an output buffer that will gracefully be outputted for you at the end of the event:

```

● clearBuffer():void
● appendToBuffer(string):void
● getBufferString():string
● getBufferObject():coldbox.system.core.util.RequestBuffer

```

The buffer is unique per interception point but available to the entire chain of execution within an interception point. Once the interception point is executed, the interceptor service will check to see if the output buffer has content, if it does it will advance to write the output to the output stream. This way, you can produce output very cleanly from your interception points, without adding any messy-encapsulation breaking `output=true` tags to your interceptors. (BAD PRACTICE). This is an elegant solution that can work for both core and custom interception points.

```

function preRender(event, interceptData){
//Clear all of it first, just in case.
clearBuffer();
//Append to buffer
appendToBuffer("<h1>This software is copyright by Funky J&#246;#!</h1>");
}

```

Important: Each execution point will have its own clean buffer to work with. As each interception point has finalized execution, the output buffer is flushed, only if content exists.

Unregistering Interceptors

Each interceptor can unregister itself from any event by using the `unregister(state)` method. This method is found in all Interceptors or can be accessed via the interceptor service as well.

```

// Inside an interceptor
unregister

```
process
```

;

// From the interceptor service
controller.getInterceptorService()
unregister(interceptor

```
unregister
```

, state

```
process
```

);

```

By using the code above in my interceptor, I would have unregistered it from the `preProcess` execution point or any point I pass.

Reporting Methods

There are several reporting and utility methods in the interceptor service that I recommend you explore. Below are some sample methods:

```

//get the entire state container for preProcess For metadata or reporting
getController().getInterceptorService().getState

```
container
```

;

//Get all the interception state containers for metadata or reporting
getController().getInterceptorService().getInterceptionStates();

//Get all the interception points registered in the application
getController().getInterceptorService().getInterceptionPoints();

```

Conclusion

As you can see from this guide, constructing ColdBox Interceptors are very easy and powerful. You have a pre-defined structure to use and a very big list of core interception points, but not only that, you can also create your own execution points. This is true power to the developer. The framework does not limit you, it actually creates a very rock solid observer/observable framework for you. So now that you understand what an interceptor is, what are the core interception points, what kind of data they receive, how they are called, how they are stored, and how to use them, you are ready to start coding your own. Godspeed my friend, Happy Intercepting!!