<< Back to Dashboard

ColdBox's Event Handlers

Contents

- ColdBox's Event Handlers
  - Introduction
  - What are Event Handlers?
  - Locations
    - Event Handlers External Location
    - Handler Registration
  - How are events called?
    - Event Syntax
      - Examples:
  - Rules and Anatomy of an Event Handler
    - The Caching Parameters
      - Sample Handler Component Declaration
    - Composed Properties
    - Feature Properties
    - Constructors
      - Non-Inheritance Constructor
      - Inheritance Constructor
  - Anatomy of an Event Handler Action
    - Request Context
    - Action Samples
    - Event Default Action
  - Get/Set Request Values
  - Setting Views
    - Views To Handler-Action
    - No Rendering
  - Relocating
    - setNextEvent
  - Rendering Data
  - Handler Return Data
  - onMissingAction convention
  - onError convention
  - Flash Variables
  - Model Integration
    - Dependency Injection
    - Requesting Model Objects
    - A practical example
      - Injection
      - Requesting
  - Model Data Binding
  - Advanced Data Binding
  - Form/Object Validation
  - AOP Simple Interceptors
    - Pre Advice
      - Exception and Only Lists
    - Post Advice
      - Exception and Only Lists
    - Around Advice Interceptors
      - Around Handler Exception and Only Lists
  - Logging
  - Executing Events
  - Testing Controllers
  - HTTP Method Restrictions
    - A quick solution
    - AllowedMethods Property
  - Helper UDF's
  - Event Caching
    - Enabling Event Caching
    - Setting Up Actions For Caching
    - Storage
    - Purging
    - this.event_cache_suffix
    - Monitoring
  - Best Practices
    - Organization
    - Executing other events (Event Chaining)
    - Naming Conventions

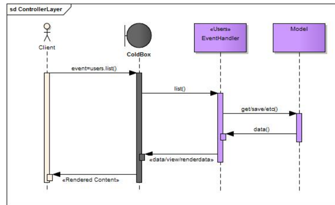
Covers up to version 3.5.0

Introduction

Event handlers are synonymous to the word **Controller** in the MVC design pattern. So every time you hear event handler, you are taking about a controller. Now, if you need a little refresher on what is MVC and how to apply it, read our MVC tutorial. If you are not familiar with ColdFusion components, you will have to get to speed in their usage in order to use ColdBox. Below are some resources on Coldfusion Components:

- Intro to CFC's by Ben Forta
- Coldfusion Articles
- Coldfusion Component Tips by O'reilly
- ColdFusion Components: A powerful tool for CF developers By: Jeffry Houser
- Our very own CFC and ColdFusion Development Best Practices

What are Event Handlers?



ColdBox event handlers are CFCs that are responsible for handling requests coming into the application from either a FORM/URL/REST or Remote sources (Flex/Air/SOAP). These event handlers carry the task of **controlling** your application flow, calling business logic, preparing a display to a user and pretty much **controlling** flow. Every method in this event handler CFC that has an access of **public** is automatically exposed as a runnable event in ColdBox and it will be auto-registered for you. That means there is no extra configuration or XML logic to define them. By convention they become alive once you create them and clients can request them. In ColdBox terms, each of these event handler methods are referred to as **actions**. As you can see from the diagram, ColdBox captures an incoming variable called **event** and uses it to execute the correct event handler CFC and action method.

```
component{
 function index(event,rc,prc){
  return "Hi from controller land!";
 }
}
```

Important Note: Event Handler's are not to be used to write business logic. They are used as controllers of your application, they make calls and redirect data.

Locations

Let's do a recap of our conventions handler locations (See Directory Structure):

```
+ application
  + handlers
    + Users.cfc
    + admin
   + Login.cfc
```

All your handlers will go in the **handlers** folder. Also notice that you can create packages or sub-folders inside of the handlers directory. This is encouraged on large applications so you can section off or package handlers logically and get better maintenance and URL experience. If you get to the point where your application needs even more decoupling and separation, please consider building ColdBox Modules instead.

Event Handlers External Location

You can also declare a **HandlersExternalLocation** setting in your Configuration CFC. This will be a dot notation path or instantiation path where more external event handlers can be found (You can use coldfusion mappings).

```
coldbox.handlersExternalLocation = "shared.myapp.handlers";
```

Note: If an external event handler has the same name as an internal conventions event, the internal conventions event will take precedence.

Handler Registration

At application startup, the framework registers all the valid event handler CFCs in these locations (plus handlers inside of modules). So for development it makes sense to activate the following setting in your Configuration CFC: **HandlersIndexAutoReload** so you can actively develop and see your changes. Once in production, change it to **false**. You also might want to disable handler caching as well so you can see your changes as you develop. Just make sure to turn it to **true** in production.

```
coldbox.handlersIndexAutoReload = true;
coldbox.handlerCaching = false;
```

How are events called?

As discussed in the introduction, events are determined via a special variable that can be sent in via the FORM or URL or REMOTELY. The default name for this variable is **event**. Of course, you can change this by updating the **EventName** setting in your configuration file. If no event name is detected as an incoming variable, the framework will look in the configuration settings for the **DefaultEvent** and use that instead (Also set in your configuration file). If you did not set a **DefaultEvent** setting then the framework will use the following convention for you:

```
DefaultEvent = "main.index";
```

Ok, so now that we know how we can determine what event to execute, how do we write the events since they are used by convention?

Event Syntax

So in order to call them you will use the following event syntax notation format:

- **no event** : Default event by convention is **main.index**
- event=**{handler}** : Default action method by convention is **index()**
- event=**{handler}.{method}** : Explicit handler + action method
- event=**{package}.{handler}.{method}** : Packaged notation
- event=**{module}:{package}.{handler}.{method}** : Module Notation (See ColdBox Modules)

This looks very similar to a java or CFC method call, example: *String.getLength()*, but without the parenthesis. Once the event variable is set and detected by the framework, the framework will tokenize the event string to retrieve the CFC and action call and validate it against the internal registry of registered events. It then continues to instantiate the event handler CFC or retrieve it from cache, and then finally executes the event handler's action method.

Important : All event handler objects are cached by default. So always always var scope and make sure they are thread safe. **Note**: Even if you use ColdBox's URL Mappings and SES URLs, you will ALWAYS end up with an event in the request collection.

Examples:

Incoming URIs

```
//No Packages
index.cfm?event=main.index

//With SES routing
index.cfm/main/index
```

Event Handler

```
component name="main"{
 function index(event,rc,prc){
  return "Hi from controller land!";
 }
}
```

This will tell the framework too look for the **main.cfc** and execute the **index()** action method. Once executed you will see a *Hi from controller land!* being rendered to the screen. Go ahead, try it, don't be shy! Later on you can learn about our powerful URL Mappings and abstract our URLs.

```
//With Package
index.cfm?event=blog.main.entry

//With SES Routing
index.cfm/blog.main/entry
```

This will tell the framework too look for the **blog** directory and then for the **main.cfc** and execute the **entry()** method.

Important: If you use the URL event syntax notation you will be **binded** to the location of your handler CFC and the name of the action methods. This is ok for small or administrative applications, but not for public facing or enterprise apps as if you refactor them, all your URLs change. Therefore, we highly encourage the use of ColdBox URL Mappings that will allow you to map custom URLs to executing events.

Rules and Anatomy of an Event Handler

- They can be simple CFCs or inherit from our base handler: **coldbox.system.EventHandler**.
- They must exist in the correct **handlers** directory under your application. (See Directory Structure)
- They must NOT contain any business logic, that is what the model or business layer is for.
- If the handlers are called via the ColdBox Proxy from Flex/Air/Remote applications, your event handlers will return data by having a return type and returning a value.
- They must have **public** methods (actions) that will respond to ColdBox events.
- Private events have an access type of **private** and can only be called from within the application by using the **runEvent()** method.
- Handlers are cached by **default**, unless the handler caching setting is off. You can configure persistence via metadata.
- You can easily wire up dependencies in your handler by using the WireBox injection DSL.

If you use the no inheritance approach, your CFCs will be mixed in a decorated at runtime to receive all the functionality of our core Base Event Handler. So it is a simulated inheritance.

The Caching Parameters

Since ColdBox is built with a solid cache foundation, CacheBox, your handlers can also be cached in the **default** cache provider. You will do this by adding meta data attributes to the **cfcomponent** tag. By default handlers **WILL BE** cached for performance, unless you specifically use the cache meta data attributes to tell the framework NOT to cache it. Caching of handlers simulates persistence, so remember this if you are planning handlers that can maintain their own persistence and ALWAYS ALWAYS var scope your function variables. That is the number one reason for illusive errors and bad best practices.

Attribute	Type	Description
cache	boolean	A true or false will let the framework know whether to cache this handler object or not. The default value is *true*.
cachetimeout	numeric	The timeout of the object in minutes. This is an optional attribute and if it is not used, the framework defaults to the default object timeout in the cache settings. You can place a 0 (Zero) in order to tell the framework to cache the handler for the entire application timeout controlled by coldfusion.
cacheLastAccesstimeout	numeric	The last access timeout of the object in minutes. This is an optional attribute and if it is not used, the framework defaults to the default last access object timeout in the cache settings. This tells the framework that if the object has not been accessed in X amount of minutes, then purge it.
singleton	none	If the component has this metadata argument then the handler will be cached forever.

The setting in your ConfigurationCFC that should be **false** in development and **true** in production is **HandlerCaching**. If you have this turned on in development you will never see your code changes:

```
coldbox.handlerCaching = false;
```

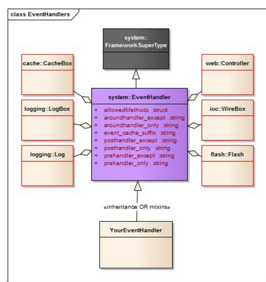
Sample Handler Component Declaration

Below is a sample handler component declaration which can exhibit some caching parameters discussed below and the default action method (index):

```
// with default caching
component{
 function index(event,rc,prc){}
}

// with singleton metadata
component singleton{
 function index(event,rc,prc){}
}

// with caching metadata
component cacheTimeout="20" cacheLastAccessTimeout="5"{
 function index(event,rc,prc){}
}

// do not cache this handler
component cache=false{
 function index(event,rc,prc){}
}
```

Composed Properties

It is imperative that you realize that there is a great object model behind every event handler that will enable you to do your work more efficiently. The following are the composed properties every event handler has in their *variables* scope, you do not need to do anything to retreive them, they are already there :)



Property	Description
cacheBox	A reference to the CacheBox framework factory (*coldbox.system.cache.CacheFactory*)
controller	A reference to the application's ColdBox Controller (*coldbox.system.web.Controller*)
flash	A reference to the current configured Flash Object Implementation that inherits from the *AbstractFlashScope* AbstractFlashScope (derived *coldbox.system.web.flash.AbstractFlashScope*)
logBox	The reference to the LogBox library (*coldbox.system.logging.LogBox*)
log	A pre-configured LogBox Logger object for this specific class object (*coldbox.system.logging.Logger*)
wirebox	A reference to the WireBox object factory (*coldbox.system.ioc.Injector*)
$super	A reference to the virtual super class (*coldbox.system.EventHandler*) **Only if using the non-inheritance approach**

Feature Properties

Each event handler can also exhibit several feature properties that can be tuned to alter the behavior of the local AOP interception points, event caching and HTTP method security. Most of these will be covered in this guide in their appropriate section, but here is a breakdown of them:

Property	Type	Description
allowedMethods	struct	A structure that determines HTTP method security for an event handler.
aroundHandler_only	string	One or a list of actions that should be intercepted for around advices **only**
aroundHandler_except	string	One or a list of actions that should NOT be intercepted for around advices
preHandler_only	string	One or a list of actions that should be intercepted for before advices **only**
preHandler_except	string	One or a list of actions that should NOT be intercepted for before advices
postHandler_only	string	One or a list of actions that should be intercepted for after advices **only**
postHandler_except	string	One or a list of actions that should NOT be intercepted for after advices

Constructors

Event handlers do not necessarily need constructors as they are already constructed by ColdBox. However, if you are in the habit of creating constructors you can in two approaches:

Non-Inheritance Constructor

Acts as a normal constructor:

```
component{

 function init(){
  // my stuf here
  return this;
 }

}
```

Inheritance Constructor

With inheritance, like any other object, you need to delegate to the super class for any overriden method:

```
component extends="coldbox.system.EventHandler"{

 function init(controller){
  // init super
  super.init(arguments.controller);
  // my stuf here
  return this;
 }

}
```

Anatomy of an Event Handler Action

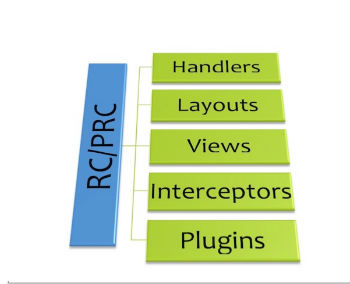
As we discussed before a handler CFC contains several methods or actions that can be executed by the framework by incoming requests or internally. You can name these methods in any way you like as long as they make sense, naming is important for readability. Each method action that you write receives some arguments:

- **event** : The request context object reference
- **rc** : A reference to the request collection inside of the request context object
- **prc** : A reference to the private request collection inside of the request context object

```
function sayHello(event,rc,prc){
 return "Hi, what's up!";
}
```

If you do not remember what the request context object is, here is a short recap or you can refer to the RequestContext guide.

Request Context



The incoming **URL, FORM and REMOTE variables** are merged into a single structure that we call the **request collection** and since we love objects that collection is stored in an object called **Request Context**. We also create a secondary collection called the **private request collection** that cannot be affected by the outside world as nothing is merged into it. You can use it for private request variables and the like.

The request context object has tons of methods to help you in setting and getting variables from one layer to another, to getting request metadata, rendering RESTful content, setting HTTP headers and more. It is your information super highway for specific requests. Remember that the API Docs are your best friend!

Also note that the **rc and prc** references each method receives are sent for convenience so you can interact with the structures instead of through the **event** object's methods. Interacting with structures over methods is much more performant.

Action Samples

```
component name="users"{

 function index(event,rc,prc){
  //get Entry
  rc.entry = getModel("EntryService").getEntry(event.getValue("entryID",0));

  //set view
  event.setView("blog/index");
 }

 function save(event,rc,prc){
  // Get a persisted entry and populated from the incoming FORM/URL variables
  var entry = populateModel( getModel("EntryService").get( rc.entryID ) );
  // Save it
  getEntryService().save( entry );
  // Relocate
  setNextEvent("blog.index");
 }

}
```

So if you needed to call the **index()** action in this handler you could do the following:

```
index.cfm?event=users.index
index.cfm?event=users
index.cfm/users/index
index.cfm/users
```

Event Default Action

The event default action setup by the framework is **index**. What this means, is that if the framework detects that an incoming event has no action attached to it, it should look for a method in that handler called **index**. If it exists, then it will execute it as the incoming event. You can change the name of this default action in your ConfigurationCFC. So if you have the following incoming URIs

```
index.cfm?event=users
or
index.cfm/users
```

The framework will look in the **users** handler for the **index** method. If it finds it, then it will treat the request as **users.index**.

```
component name="Users"{

 function index(event,rc,prc){

  prc.users = userService.list();

  event.setView("users/index");

 }

}
```

Get/Set Request Values

We all need values in our applications, and that is why we will interact with the request context in order to place data from our model layer so our views can display it or retreive data from a user's request. So you will either interact with the **event** object to get/set values or put/read values directly via the received **rc and prc** references. We recommend using the references as structures are much faster than method calls. However, the **event** object should not be discarded as it has some pretty cool and funky methods of its own. Below are some examples of its coolness!

We would recommend you use the private request collection for setting manual data and using the standard request collection for reading the user's request variables. This way a clear distinction can be made on what was sent from the user and what was set by your code.

```
<cfscript>
//set a value for views to use
event.setValue("name", "Luis");

// retrieve a value the user sent
event.getValue("name");

// retrieve a value the user sent or give me a default value
event.getValue("isChecked",false);

//param a value
event.paramValue("user_id","");

//remove a value
event.removeValue("name");

//check if value exists
if( event.valueExists("name") ){

}

// set a view for rendering
event.setView('blog/index');

// set a layout for rendering
event.setLayout('main');

// set a view and layout
event.setView(view="blog/userinfo",layout="ajax");

</cfscript>
```

Important : The most important paradigm shift from procedural to an MVC framework is that you NO LONGER will be talking to URL, FORM, REQUEST or any ColdFusion scope from within your handlers, layouts and views. The request collection already has URL, FORM and REQUEST scope capabilities, so leverage it.

Setting Views

The event object is the object that will let you set the views that you want to render, so please explore its API in the CFC Docs. To quickly set a view to render, do the following:

```
event.setView('view');
```

The view name is the name of the template in the **views** directory without appending the .cfm. So if the view is inside another directory you would do this:

```
event.setView('mydirectory/myView');
```

You can also do caching of views, layouts, and so much more. For that, we invite you to check out the Layouts-Views Layouts and Views guide after this one.

Here are some arguments to the **setView()** method:

- **view** - The name of the view to set. If a layout has been defined it will assign it, else if will assign the default layout. No extension please
- **nolayout** - Boolean flag, wether the view sent in will be using a layout or not. Default is false. Uses a pre set layout or the default layout.
- **cache** - True if you want to cache the view.
- **cacheTimeout** - The cache timeout
- **cacheLastAccessTimeout** - The last access timeout
- **cacheSuffix** - Add a cache suffix to the view cache entry. Great for multi-domain caching or i18n caching.
- **layout** - You can override the rendering layout of this setView() call if you want to. Else it defaults to implicit resolution or another override.
- **module** - Is the view from a module or not
- **args** - An optional set of arguments that will be available when the view is rendered

Views To Handler-Action

We recommend that you maintain a consistent naming and location schema between views and your handler and actions, often called *implicit views*. So if you have an incoming event called: **users.index** then make sure in your views folder you have:

```
+views
 +users
  +index.cfm
```

This way debugging is much easier and also Implicit Views can be used. Implicit views means that you won't use a **event.setView()** to specify what view to render. It is implied the view to render will be the same as the executing event.

No Rendering

Well, if you don't want to, then you don't have to. The framework gives you a method in the event object that you can use if maybe this specific request should just terminate gracefully and not render anything at all. All you need to do is use the event object to call on the **noRender()** method.

```
event.noRender();
```

This method tells the framework that this request will not produce any output, so just finalize the request. Most likely you will end up with a white page or if called from ajax, nothing.

Relocating

The framework provides you with a method that you can use to relocate to other events thanks to the framework super type object, the grand daddy of all things ColdBox.

```
public void setNextEvent([string event], [string queryString], [boolean addToken], [string persist], [struct persistStruct], [boolean ssl], [string baseURL], [boolean postProcessExempt], [string URL], [string URI], [numeric
```

It is **extremely** important that you use this method when relocating instead of the native ColdFusion methods as it allows you to gracefully relocate to other events or external URIs. By graceful, we mean it does a lot more behind the scenes like making sure the flash scope is persisted, loging, post processing interceptions can occurr and safe relocations. So always remember that you relocate via **setNextEvent** and if I asked you: "Where in the world does event handlers get this method from?", you need to answer: "From the super typed inheritance".

setNextEvent

The setNextEvent method can be used for both normal and ses urls, here are its parameters:

Argument	Required	Type	Description
event	false	string	The name of the event or SES pattern to relocate to, if not passed, then it will use the default event found in your configuration file. (Mutex with URI and URL)
URL	false	Absolute URL	The absolute URL to relocate to (Mutex with URI and event)
URI	false	Relative URL	The relative URI to relocate to (Mutex with event and URL)
queryString	false	query string	The query string to append to the relocation. It will be converted to SES if SES is used.
addToken	false	boolean	Whether to add the cf tokens or not. Default is false
persist	false	string(list)	A comma-delimited list of request collection key names that will be flash persisted in the framework's flash RAM and re-inflated in the next request.
persistStruct	false	struct	A structure of key-value pairs that will be flash persisted in the framework's flash RAM and re-inflated in the next request.
ssl	false	boolean(false)	Flag indicating if redirect should be done in ssl mode or not
baseURL	false	string	If used, then it is the base url for normal syntax redirection instead of just redirecting to the index.cfm
postProcessExempt	false	boolean(false)	Do not fire the postProcess interceptors
statusCode	false	numeric	The status code to relocate with

The ColdBox FlashRAM capabilities are discussed in this guide in much more detail, so we welcome you to go there and digest it after this guide.

Important: Please note that the **persist** argument refers to items ALREADY in the request collection.

Rendering Data

You can also use the **event.renderData()** method to render and marshal data directly from an event handler without the need to set a view for rendering. You can find an in depth guide for rendering data in our Layouts-Views. Out of the box ColdBox can marshall data (structs,queries,arrays,complex or even ORM entities) into the following output formats:

- XML
- JSON
- JSONP
- HTML
- TEXT
- WDDX
- PDF
- Custom

```
// xml marshalling
function getUsersXML(event,rc,prc){
 var qUsers = getUserService().getUsers();
 event.renderData(type="XML",data=qUsers)
}
//json marshalling
function getUsersJSON(event,rc,prc){
 var qUsers = getUserService().getUsers();
 event.renderData(type="json",data=qUsers)
}

// restful handler
function list(event,rc,prc){
 event.paramValue("format","html");

 rc.data = userService.list();

 switch(rc.format){
  case "json": "jsont": "xml"{
   event.renderData(type=rc.format,data=rc.data);
   break;
  }
  default {
   event.setView("users/list");
  }
 }

}

// simple tests
function data(event,rc,prc){
 var data = {
  name ="ColdBox", awesome = true, ratings = [5,5,4,3]
 };

 event.renderData(data=data,type="json");
}
```

As you can see, it is very easy to render data back to the browser or caller. You can even choose plain and send HTML back if you wanted too. You can also render out PDF's from ColdBox using the render data method. The **data** argument can be either the full binary of the PDF or simple values to be rendered out as a PDF, like views, layouts, strings, etc.

```
// from binary
function pdf(event,rc,prc){
  var binary = fileReadAsBinary( file.path );
  event.renderData(data=binary,type="PDF");
}

// from content
function pdf(event,rc,prc){
  event.renderData(data=renderView("views/page"), type="PDF");
}
```

There is also a **pdfArgs** argument in the render data method that can take in a structure of name-value pairs that will be used in the **cfdocument** (See docs) tag when generating the PDF. This is a great way to pass in arguments to really control the way PDF's are generated uniformly.

```
// from content and with pdfArgs
function pdf(event,rc,prc){
  var pdfArgs = { bookmark="yes", backgroundVisible="yes", orientation="landscape"};
  event.renderData(data=renderView("views/page"), type="PDF", pdfArgs=pdfArgs);
}
```

Handler Return Data

Handlers can also return either simple strings or complex objects. If they return simple strings then the strings will be rendered out to the user:

```
component name="general"{

 function index(event,rc,prc){
  return "<h1>Hello from my handler today at :#now()#</h1>";
 }

}
```

However, they can also return complex data in case you are integrating your application with Flex/Air/Ajax or SOAP Web services via the ColdBoxProxy ColdBox Proxy.

```
component name="users"{

 function list(event,rc,prc){
  rc.users = userService.list();
  if( event.isProxyRequest() ){
   return rc.users;
  }

  event.setView("users/list");
 }

}
```

onMissingAction convention

With this convention you can create virtual events that do not even need to be created or exist in a handler. Every time an event requests an action from an event handler and that action does not exists in the handler, the framework will check if an **onMissingAction()** method has been declared. If it has, it will execute it. This is very similar to ColdFusion's **onMissingMethod** but on an event-driven framework.

```
// On Missing Action
function onMissingAction(event,missingAction,eventArguments){
 // present a static page according to missing action
 event.setView("handler/#arguments.missingAction#");
}
```

Please note that the arguments for this method are different than normal action methods

This event has an extra argument: **missingAction** which is the missing action that was requested. You can then do any kind of logic against this missing action and decide to do internal processing, error handling or anything you like. The power of this convention method is extraordinary, you have tons of possibilities as you can create virtual events on specific event handlers.

onError convention

This is a localized error handler for your event handler. If any type of runtime error ocurrs in an event handler and this method exists, then the framework will call your **onError()** method so you can process the error. If the method does not exist, then normal error procedures ensue.

```
// On Error
function onError(event,faultAction,exception,eventArguments){
 // prepare a data packet
 var data = {
  error = true,
  messages = exception.message & exception.detail,
  data = ""
 }

 // log via the log variable already prepared by ColdBox
 log.error("Exception when executing #arguments.faultAction# #data.messages#", exception);

 // render out a json packet according to specs status codes and messages
 event.renderData(data=data,type="json",statusCode=500,statusMessage="Error ocurred");

}
```

Please note that the arguments for this method are different than normal action methods

As you can see from my sample code, I can use the **onError()** method to provide uniform error handling for event handlers. In this example, I am using it for a RESTful handler.

Flash Variables

As we shown before, you can use ColdBox's FlashRam persistence via the *setNextEvent* methods. However, you have a **flash** object reference available in your **variables** scope that is ready for usage for all your flashing needs (Ok, don't take that line out of context please). Read the FlashRAM guide for in-depth training and also visit the API Docs for the latest Flash RAM API.

```
function save(event,rc,prc){
 var user = populateModel( userService.new() );
 userservice.save( user );

 // put some message that user was saved
 flash.put("notice","User saved!");

 // relocate
 setNextEvent("users.index");
}

// Show notice in the index page, in two different approaches
<div class="notices">#flash.get(name="notice",default="")#</div>
// OR
<cfif flash.exists("notice")>
 <div class="notices">#flash.get("notice")#</div>
</div>
```

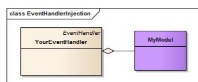
Model Integration

We have a complete guide dedicated to Model Integration but we wanted to review a little here since event handlers need to talk to the model layer all the time. By default you can interact with your models from your event handlers in two ways:

- Dependency Injection
- Request model objects

ColdBox offers its own dependency injection framework, WireBox, which allows you by convention to talk to your model objects. However, ColdBox also allows you to connect to ColdSpring, LightWire or any other custom object factory via our IOC plugin.

Dependency Injection



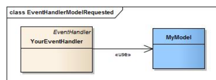
Your event handlers can be autowired with dependencies from either WireBox, ColdSpring, or any custom object factory by means of our injection DSL. By autowiring dependencies into event handlers, they will become part of the life span of the event handlers and thus gain on the performance that an event handler is wired with all necessary parts upon creation. This is a huge benefit and we encourage you to use injection whenever possible. Please note that injection aggregates model objects into your event handlers. The Injection DSL can be applied to:

- cfproperties
- constructor arguments
- setter methods

It will be your choice to pick an approach, but we mostly concentrate on property injection as you will see from our examples.

Aggregation differs from ordinary composition in that it does not imply ownership. In composition, when the owning object is destroyed, so are the contained objects. - 'wikipedia'

Requesting Model Objects



The other approach to integrating with model objects is to request them and use them as associations. From who? From either WireBox or the IOC Plugin. We would recommend requesting objects if they are transient objects or stored in some other volatile storage scope. Retreiving of objects is ok, but if you will be dealing with mostly singleton objects or objects that are created only once, you will gain much more performance by using injection.

Association defines a relationship between classes of objects that allows one object instance to cause another to perform an action on its behalf. - 'wikipedia'

A practical example

In this practical example we will see how to integrate with our model layer via WireBox injections and also requesting the objects. Let's say that we have a service object we have built called **FunkyService.cfc** and by convention we will place it in our applications **model** folder.

```
+ application
  + model
     + FunkyService.cfc
```

FunkyService.cfc

```
component singleton{

 function init(){
  return this;
 }

 function add(a,b){ return a+b; }

 function getFunkyData(){
  var data = [
   {name="Luis", age="33"},
   {name="Jim", age="99"},
   {name="Alex", age="1"},
   {name="Joe", age="23"}
  ];
  return data;
 }

}
```

Our funky service is not that funky after all, but it is simple. So how do we interact with it? Let's build a **Funky** event handler and work with it.

Injection

```
component{

 // Injection
 property name="funkyService" inject;

 function index(event,rc,prc){

  prc.data = funkyService.getFunkyData();

  event.renderData(data=prc.data,type="xml");
 }


}
```

So by convention, I can create a property and annotate it with a *inject* attribute and ColdBox will look for that model object by name in the **model** folder, create it, persist it, wire it and return it. If you execute it, you will get something like this:

```
<array>
 <item>
  <struct>
   <name>Luis</name>
   <age>33</age>
  </struct>
 </item>
 <item>
  <struct>
   <name>Jim</name>
   <age>99</age>
  </struct>
 </item>
 <item>
  <struct>
   <name>Alex</name>
   <age>1</age>
  </struct>
 </item>
 <item>
  <struct>
   <name>Joe</name>
   <age>23</age>
  </struct>
 </item>
```

```
</array>
```

Great! Just like that we can interact with our model layer without worrying about creating the objects, persisting them and even wiring them. That is exactly all the benefits that dependency injection and model integration bring to the table.

Alternative Wiring

```
// Injection using the DSL
property name="funkyService" inject="FunkyService";
// Injection using the DSL
property name="funkyService" inject="id:FunkyService";
// Injection using the DSL
property name="funkyService" inject="model:FunkyService";
```

Requesting

Let's look at the requesting approach. We can either use the following approaches:

Via Facade Method

```
component{

 function index(event,rc,prc){

  prc.data = getModel("FunkyService").getFunkyData();

  event.renderData(data=prc.data,type="xml");
 }


}
```

Directly to WireBox:

```
component{

 function index(event,rc,prc){

  prc.data = wirebox.getInstance("FunkyService").getFunkyData();

  event.renderData(data=prc.data,type="xml");
 }


}
```

Both approaches do exactly the same, in all reality **getModel()** does a **wirebox.getInstance()**, it is a facade method that is easier to remember. If you run this, you will also see that it works and everything is fine and dandy. However, the biggest difference can be seen with some practical math:

```
1000 Requests made
- Injection: 1000 handler calls + 1 model creation and wiring call = 1001 calls
- Requesting: 1000 handler calls + 1000 model retrieval + 1 model creation call = 2002 calls
```

As you can see, the best performance is due to injection as the handler object was wired and ready to roll, while the requested approach needed the dependency to be requested. Again, there are cases where you need to request objects such as transient or volatile stored objects.

Model Data Binding

WireBox also offers you the capability to bind incoming FORM/URL/REMOTE data into your model objects by convention. The easiest approach is to use our **populateModel()** function call:

```
populateModel(any model, [any scope=''], [any<Boolean> trustedSetter='false'], [any include=''], [any exclude=''])
```

This will try to match incoming variable names to setters or properties in your domain objects and then populate them for you.

Argument	Type	Required	Default	Description
model	any	true	---	The name of the model to get and populate or the actual model object reference.
scope	string	false		Use scope injection instead of setters population. Ex: scope=variables.instance.
trustedSetter	boolean	false	false	If set to true, the setter method will be called even if it does not exist in the model object
include	string	false		A list of keys to include in the population
exclude	string	false		A list of keys to exclude in the population

Let's do a quick sample:

Person.cfc

```
component accessors="true"{

 property name="name";
 property name="email";

 function init(){
  setName('');
  setEmail('');
 }
}
```

Then here is our form (using our awesome HTML helper):

editor.cfm

```
<cfoutput>
<h1>Funky Person Form</h1>
#html.startForm(action='person.save')#

 #html.textfield(label="Your Name:",name="name",wrapper="div")#
 #html.textfield(label="Your Email:",name="email",wrapper="div")#

 #html.submitButton(value="Save")#

#html.endForm()#
</cfoutput>
```

And our event handler to process it:

person.cfc

```
component{

 function editor(event,rc,prc){
  event.setView("person/editor");
 }

 function show(event,rc,prc){

  var person = populateModel("Person");

  writeDump(person);abort;
 }

}
```

In the dump you will see that the *name* and *email* properties have been binded, cool it works.

Advanced Data Binding

WireBox also sports several advanced data binding techniques:

- populateFromJSON()
- populateFromQuery()
- populateFromQueryWithPrefix()
- populateFromStruct()
- populateFromXML()

We suggest you look at the latest API Docs for the correct arguments and such. To use these beauties you can either go to the wirebox reference directly or use the *BeanFactory* plugin. Both accomplish the same:

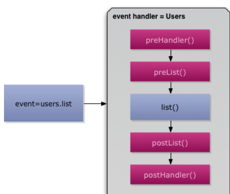
```
wirebox.getObjectPopulator().populateFromXXXX()
```

```
getPlugin("BeanFactory").populateFromXXXX()
```

Form/Object Validation

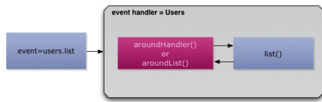
We also have the ability to validate forms and objects using our awesome Validation framework called ValidBox. So check it out to learn how to validate stuff.

AOP Simple Interceptors



Event Handlers Pre-Post Interceptors

There are also several simple implicit AOP interceptors that can be declared in your event handler that the framework will use in order to execute them anytime an event is fired from the current handler. This is great for intercepting calls, pre/post processing, localized security, logging, RESTful conventions and much more. Yes, you got that right, Aspect Oriented Programming just for you and without all the complicated setup involved! If you declared them, the framework will execute them. (See Request Life Cycle)



Event Handlers Around Interceptors

Method	Description
preHandler	Executes before any requested action (In the same handler CFC)
pre{Action}	Executes before the {action} requested ONLY
postHandler	Executes after any requested action (In the same handler CFC)
post{Action}	Executes after the {action} requested ONLY
aroundHandler	Executes around any request action (In the same handler CFC)
around{Action}	Executes around the {action} requested ONLY

Pre Advice

With this interceptor you can intercept local event actions and execute things before the requested action executes. You can do it globally by using the *preHandler()* method or targeted to a specific action *pre{actionName}()*.

```
// executes before any action
function preHandler(event,action,eventArguments){
}

// executes before the list() action ONLY
function preList(event,action,eventArguments){
}

// concrete example
function preHandler(event,action,eventArguments){
 if( !security.isLoggedIn() ){
  event.overrideEvent('security.login');
  log.info("Unauthorized accessed detected!", getHTTPRequestData());
 }
}
function preList(event,action,eventArguments){
 log.info("Starting executing the list action");
 getPlugin("Timer").start('list-profile');
}
```

The arguments received by these interceptors are:

Argument	Type	Description
event	*coldbox.system.web.context.RequestContext*	The request context object reference
action	string	The name of the action that got intercepted for *pre/post* interceptions only.
eventArguments	struct	The structure of name-value pairs the event was called with if called via **runEvent()**

Please note that the **rc,prc** references are not passed to the interceptors, you will manualy have to retrieve them (*event.getCollection(boolean:private)*) if you would like to use the structure references.

Exception and Only Lists

- **prehandler_only** : A list of actions that the **preHandler()** action will fire ONLY!
- **prehandler_except** : A list of actions that the **preHandler()** action will NOT fire on

```
// only fire for the actions: save(), delete()
this.prehandler_only = "save,delete";

// DO NOT fire for the actions: login(), doLogin(), logout()
this.prehandler_except = "login,doLogin,logout"
```

Post Advice

With this interceptor you can intercept local event actions and execute things after the requested action executes. You can do it globally by using the *postHandler()* method or targeted to a specific action *post{actionName}()*.

```
// executes after any action
function postHandler(event,action,eventArguments){
}

// executes after the list() action ONLY
function postList(event,action,eventArguments){
}

// concrete examples
function postHandler(event,action,eventArguments){
 log.info("Finalized executing #action#");
 getPlugin("Timer").stop(action);
}
```

The arguments received by these interceptors are:

Argument	Type	Description
event	*coldbox.system.web.context.RequestContext*	The request context object reference
action	string	The name of the action that got intercepted for *pre/post* interceptions only.
eventArguments	struct	The structure of name-value pairs the event was called with if called via **runEvent()**

Please note that the **rc,prc** references are not passed to the interceptors, you will manualy have to retrieve them (*event.getCollection(boolean:private)*) if you would like to use the structure references.

Exception and Only Lists

- **posthandler_only** : A list of actions that the **postHandler()** action will fire ONLY!
- **posthandler_except** : A list of actions that the **postHandler()** action will NOT fire on

```
// only fire for the actions: save(), delete()
this.posthandler_only = "save,delete";

// DO NOT fire for the actions: login(), doLogin(), logout()
this.posthandler_except = "login,doLogin,logout"
```

Around Advice Intercpetors

Around advices are the most powerful of all as you completely hijack the requested action with your own action that looks, smells and feels exactly as the requested action. This will allow you to run both before and after stuff but also surround the method call with whatever logic you want like transactions, try/caqtch blocks, or even decide to NOT execute the action at all. You can do it globally by using the *aroundHandler()* method or targeted to a specific action *around{actionName}()*.

```
// executes around any action
function aroundHandler(event,targetAction,eventArguments){
}

// executes around the list() action ONLY
function aroundList(event,targetAction,eventArguments){
}

// Around handler advice for transactions
function aroundHandler(event,targetAction,eventArguments){

 // log the call
 log.debug("Starting to execute #targetAction.toString()#");

 // start a transaction
 transaction{

  // prepare arguments for action call
  var args = {
   event = arguments.event,
   rc    = arguments.event.getCollection(),
   prc   = arguments.event.getCollection(private=true),

  };
  structAppend(args,eventArguments);
  // execute the action now
  return arguments.targetAction(argumentCollection=args);
 }

}

// Around handler advice for try/catches
function aroundHandler(event,targetAction,eventArguments){

 // log the call
 if( log.canDebug() ){
  log.debug("Starting to execute #targetAction.toString()#");
 }

 // try block
 try{

  // prepare arguments for action call
  var args = {
   event = arguments.event,
   rc    = arguments.event.getCollection(),
```

```
   prc   = arguments.event.getCollection(private=true),
  };
  structAppend(args,eventArguments);
  // execute the action now
  return arguments.targetAction(argumentCollection=args);
 }
 catch(Any e){
  // log it
  log.error("Error executing #targetAction.toString()#: #e.message# #e.detail#", e);
  // set exception in request collection and set view to render
  event.setValue("exception", e)
   .setView("errors/generic");

 }

}
```

The arguments received by these interceptors are:

Argument	Type	Description
event	*coldbox.system.web.context.RequestContext*	The request context object reference
targetAction	UDF Pointer	The UDF pointer to the action that got the around interception. It will be your job to execute it (Look at samples)
eventArguments	struct	The structure of name-value pairs the event was called with if called via **runEvent()**

Please note that the **rc,prc** references are not passed to the interceptors, you will manualy have to retrieve them (*event.getCollection(boolean:private)*) if you would like to use the structure references.

Around Handler Exception and Only Lists

- **aroundhandler_only** : A list of actions that the **aroundHandler()** action will fire ONLY!
- **aroundhandler_except** : A list of actions that the **aroundHandler()** action will NOT fire on

```
// only fire for the actions: save(), delete()
this.aroundhandler_only = "save,delete";

// DO NOT fire for the actions: login(), doLogin(), logout()
this.aroundhandler_except = "login,doLogin,logout"
```

Logging

Logging is an important aspect of any application as it allows you to report textual information about your application's state or environment. ColdBox has powerful logging capabilities via LogBox and your event handlers are already configured to use it, really, just use it. If you remember the composed properties section you will see that every event handler has two object references for logging: **logbox** and **log**. The **log** object is LogBox logger already configured for your event handler and you can use it for logging:

Logger Interface

```
function debug(message,extrainfo){}
function info(message,extrainfo){}
function warn(message,extrainfo){}
function error(message,extrainfo){}
function fatal(message,extrainfo){}
boolean function canDebug(){}
boolean function canInfo(){}
boolean function canWarn(){}
boolean function canError(){}
boolean function canFatal(){}
```

As you can see, ColdBox tackles all the complexities of logging and gives you so much more than plain 'ol *cflog*.

```
component{

 function index(event,rc,prc){
  log.info("Hey, I am here executing #event.getCurrentEvent()#", getHTTPRequestData() );
 }

}
```

Executing Events

Apart from executing events from the URL/FORM or Remote interfaces, you can also execute events internally, either public or private from within your event handlers or from plugins, interceptors, layouts or views. You do this by using the **runEvent()** method which is inherited from our *FrameworkSuperType* class.

```
runEvent([any event=''], [any<boolean> prepostExempt='false'], [any<boolean> private='false'], [any<struct> eventArguments='[runtime expression]'])
```

Arguments

Argument	Type	Required	Default	Description
event	string	true	---	The event to execute using event syntax.
prepostExempt	boolean	false	false	If set to true, it will bypass any pre or post handler implicit executions.
private	boolean	false	false	If set to true, it will try to execute a private event (private method).
eventArguments	struct	false	structnew()	A structure of name-value pairs to pass into the method call. Basically each key is passed an argument to the action method.

The interesting aspect of internal executions is that all the same rules apply, so your handlers can return content like widgets, views, or even data. Also, the **eventArguments** enables you to pass arguments to the method just like method calls:

```
//public event
runEvent('users.save');

//post exempt
runEvent(event='users.save',prepostExempt=true);

//Private event
runEvent(event='users.persist',private=true);

// Run event as a widget
<cfoutput>#runEvent(event='widgets.userInfo',prePostExempt=true,eventArguments={widget=true})#</cfoutput>

// handler responding to widget call
function userInfo(event,rc,prc,widget=false){

 prc.userInfo = userService.get( rc.id );

 // set or widget render
 if( arguments.widget ){
  return renderView("widgets/userInfo");
 }

 // else set view
 event.setView("widgets/userInfo");

}
```

Layouts-Views Viewlets or Widgets are a great way to create self-sustainable events, please check out our guide on it for more information.

Testing Controllers

ColdBox offers two approaches to testing your event handlers:

- **Integration Testing** : Tests everything top-down in your application
- **Handler Testing** : Like unit testing for handlers

The main difference is that integration testing will virtually create your application and execute the event you want. Thus, loading everything in a typical request and simulate it. This is a fantastic approach as it lets you test like you would from a browser. The handler testing just tests the event handler in isolation much like unit testing does. To get started with testing, please visit our Testing section.

HTTP Method Restrictions

More than often you will find that certain web operations needs to be restricted in terms of what HTTP verb is used to access a resource. For example, you do not want form submissions to be done via *GET* but via *POST* or *PUT* operations. HTTP Verb recognition is also essential when building strong RESTFul APIs and security is needed as well.

A quick solution

One way of dealing with these security concerns is to determine the incoming HTTP verb and respond accordingly:

```
function delete(event,rc,prc){
 // determine incoming http method
 if( event.getHTTPMethod() == "GET" ){
  flash.put("notice","invalid action");
  setNextEvent("users.list");
 }
 else{
  // do delete here.
 }
}

// or
function preDelete(event,action){
 // determine incoming http method
 if( event.getHTTPMethod() == "GET" ){
  flash.put("notice","invalid action");
  setNextEvent("users.list");
 }
}
```

This solution is great and works, but it is not THAT great. We can do better, how about using those nice before advices? You can but then you have to mess with the except and only lists. This approach is ok and you most likely will have great control on what to do with the invalid request, but there is no uniformity and you still have to write code for it.

AllowedMethods Property

Another feature property on an event handler is called **allowedMethods** and it is a declarative structure that you can use to determine what are the allowed HTTP methods for any action on the event handler. If the request action HTTP method is not found in the list then it throws a 403 exception that is uniform across requests.

```
component{

 this.allowedMethods = {
  delete = "POST,DELETE",
  list   = "GET"
 };

 function list(event,rc,prc){
  // list only
 }

 function delete(event,rc,prc){
  // do delete here.
 }
}
```

The key is the name of the action and the value is a list of allowed HTTP methods. If the action is not listed in the structure, then it means allow all. That's it! Just remember to either use the **onError()** convention or an exception handler to deal with the security exceptions.

Helper UDF's

ColdBox provides you with a way to actually inject your event handlers with custom UDF's, so they can act as helpers. This is called mixin methods and can be done via the **includeUDF()** method provided to every event handler or via the **UDFLibrary** setting in your configuration file. The method is a provided way for you to dynamically load UDF's at runtime and the **UDFLibrary** setting is a convention that acts globally on all layouts, views and event handlers.

```
coldbox.udfLibrary = 'includes/helpers/applicationHelper.cfm';
```

```
<cfset includeUDF('includes/helpers/handlerHelper.cfm')>
```

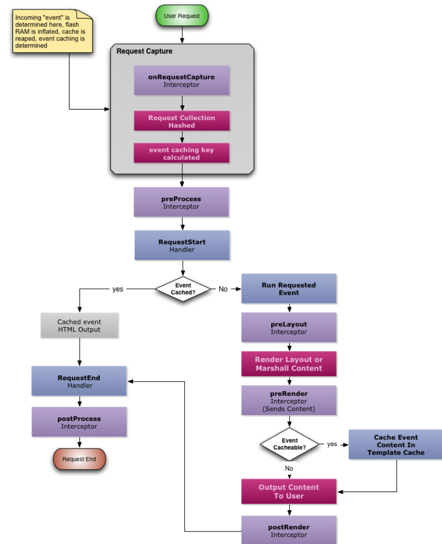
- The **includeUDF** method call will find the template and inject it to the event handler
- The **UDFLibrary** setting injects the UDF's in the template to the handlers/layouts and views.

Warning: If you try to inject a method that already exists, the call will fail and ColdFusion will throw an exception. Also, try not to abuse mixins, if you have too many consider refactoring into model objects or plugins.

Event Caching

Event caching is extremely useful and easy to use. All you need to do is add several metadata arguments to the action methods and the framework will cache the output of the event in the **template** cache provider in CacheBox. In other words, the event executes and produces output that the framework then caches. So the subsequent calls do not do any processing, but just output the content. For example, you have an event called **blog.showEntry**. This event executes, gets an entry from the database and sets a view to be rendered. The framework then renders the view and if event caching is turned on for this event, the framework will cache the HTML produced. So the next incoming show entry event will just spit out the cached html. Important to note also, that any combination of URL/FORM parameters on an event will produce a unique cacheable key. So **event=blog.showEntry&id=1 & event=blog.showEntry&id=2** are two different cacheable events.

Almost all of the entire life cycle is skipped, the content is just delivered. Below you can see the life cycle of both cached and normal events:



As you can tell from the diagram, event caching can really increase performance.

Enabling Event Caching

To enable event caching, you will need to set a setting in your ConfigurationCFC called **EventCaching** which is a Boolean variable.

```
coldbox.eventCaching = true;
```

If you do not enable this setting, the framework will not cache your events. It would be a good idea to have this set to *false* in development, so your changes can be reflected.

Enabling event caching does not mean that ALL events will be cached. It just means that you enable this feature.

Setting Up Actions For Caching

The way to set up an event for caching is on the function declaration with the following extra attributes (annotations):

Attribute	Type	Description
cache	boolean	A true or false will let the framework know whether to cache this event or not. The default is FALSE. So setting to false makes no sense
cachetimeout	numeric	The timeout of the event's output in minutes. This is an optional attribute and if it is not used, the framework defaults to the default object timeout in the cache settings. You can place a 0 in order to tell the framework to cache the event's output for the entire application timeout controlled by coldfusion, NOT GOOD. Always set a decent timeout for content.
cacheLastAccesstimeout	numeric	The last access timeout of the event's output in minutes. This is an optional attribute and if it is not used, the framework defaults to the default last access object timeout in the cache settings. This tells the framework that if the object has not been accessed in X amount of minutes, then purge it.

Important: Please be aware that you should not cache output with 0 timeouts (forever). Always use a timeout.

```
//Sample event caching
<cffunction name="showEntry" access="public" output="false" cache="true" cacheTimeout="30" cacheLastAccessTimeout="15">
 <cfargument name="event">
 <cfargument name="rc">
 <cfargument name="prc">
 <cfscript>
  //get Entry
  prc.entry = getEntryService().getEntry(event.getValue('entryID',0));

  //set view
  event.setView('blog/showEntry');
 </cfscript>
</cffunction>

// In Script
function showEntry(event,rc,prc) cache="true" cacheTimeout="30" cacheLastAccessTimeout="15"{
 //get Entry
 prc.entry = getEntryService().getEntry(event.getValue('entryID',0));

 //set view
 event.setView('blog/showEntry');
}
```

Note: DO NOT cache events as unlimited timeouts. Also, all events can have an unlimited amount of permutations, so make sure they expire and you purge them constantly. Every event + URL/FORM variable combination will produce a new cacheable entry.

Storage

All event and view caching are stored in a named cache called **template** which all ColdBox applications have by default. You can open or create a new CacheBox configuration object and decide where the storage is, timeouts, providers, etc. You have complete control of how event and view caching is stored.

Purging

We also have a great way to purge these events programmatically via our cache provider interface. You will have to either retrieve or inject a reference to the **template** cache provider and then call methods on it.

```
templateCache = getColdBoxOCM("template");
templateCache = cachebox.getCache("template");
```

Methods for event purging:

- **clearEvent**(string eventSnippet, string querystring=""): Clears all the event permutations from the cache according to snippet and querystring. Be careful when using incomplete event name with query strings as partial event names are not guaranteed to match with query string permutations
- **clearEventMulti**(eventsnippets,string querystring=""): Clears all the event permutations from the cache according to the list of snippets and querystrings. Be careful when using incomplete event name with query strings as partial event names are not guaranteed to match with query string permutations
- **clearAllEvents**([boolean async=true]) : Can clear ALL cached events in one shot and can be run asynchronously.

```
//Trigger to purge all Events
getColdBoxOCM("template").clearAllEvents();
```

```
//Trigger to purge all events synchronously
getColdBoxOCM("template").clearAllEvents(async=false);

//Purge all events from the blog handler
getColdBoxOCM("template").clearEvent('blog');

//Purge all permutations of the blog.dspBlog event
getColdBoxOCM("template").clearEvent('blog.dspBlog');

//Purge the blog.dspBlog event with entry of 12345
getColdBoxOCM("template").clearEvent('blog.dspBlog','id=12345')
```

this.event_cache_suffix

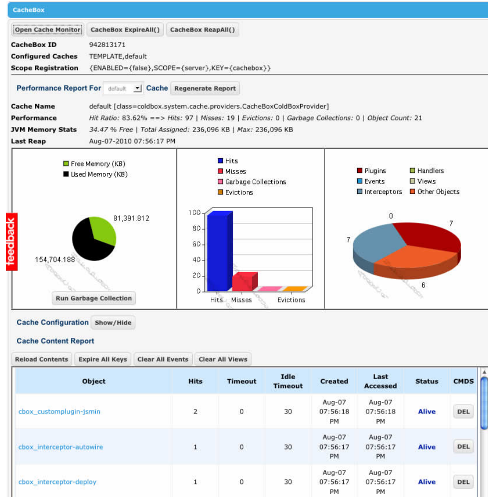
Do you remember this feature property? This property is great for adding your own dynamic suffixes when using event caching. All you need to do is create a public property called **EVENT_CACHE_SUFFIX** and populate it with something you want. Then the event caching mechanisms will automatically append the suffix and thus create event caching using this suffix for the entire handler.

```
this.EVENT_CACHE_SUFFIX = "My Suffix";
```

Note: This suffix will be appended to ALL events that are marked for caching within the handler in question ONLY.

Monitoring

CacheBox has an intuitive and powerful monitor that can be used when you are in debug mode in your ColdBox application. From the monitor you can purge, expire and view cache elements, etc.



Best Practices

Organization

Always try to have some kind of mapping between the different logical sections or modules of your application and their event handlers. For example, if the application has a section for user management, with master and detail views, create a single event handler CFC to hold all the methods related to that module. Now, large sections of your application that are more complex and have lots of actions and views, may require you to split the event handlers even more (like packages/directories), or ColdBox Modules.

The handler CFCs are also objects and therefore they should have their specific identity and function. So you need to map them in the best logical way according to their functions. So think of them as entities and how they would do tasks for you via the events. Once you do this, you can come up with a very good cohesive event model for your applications.

In conclusion, organize your handlers as you would a domain model, put in practice your Ontology skills and define what these handlers will do for your, what is their identity.

Executing other events (Event Chaining)

The best practice on event execution would be via the **runEvent()** method. However, please note that running events via this method does a complete life cycle execution. So do not abuse it. If you find that you need to chain events or are getting more complex, we suggest upgrading your code to use ColdBox Interceptors. This is a much more flexible and decoupled way of providing executing code chains.

Naming Conventions

Try always to add meaningful names to methods so other developers and users can understand what you are doing.