An Introduction to the ColdBox Platform

www.coldbox.org

Covers up to version 3.7.0

Contents

ColdBox is an event-driven conventions-based MVC framework for ColdFusion. It is fast, scalable, and runs on CFML engines such as Adobe ColdFusion and the open source CFML engine Railo, for a completely free and open source development stack.

ColdBox itself is Professional Open Source Software, backed by Ortus Solutions which provides support, training, architecture, consulting and commercial additions. ColdBox was the first ColdFusion framework to embrace Conventions Over Configuration and comes with many out-of-the-box tools and plugins to enhance developer and team productivity. With integrated debugging, IDE integration, RESTFul services and an online code-sharing community called ForgeBox, ColdBox is more than just a framework, it's a development platform.

Here's a look at some of the core components of the ColdBox platform. Each of these come bundled with the framework, but are also available as separate stand-alone libraries that can be used in ANY ColdFusion application or framework:

LOGBOX By ColdBox

This is a highly-configurable logging library, inspired by Log4J, which can be set up to relay messages of different types from any portion of your application to any number of predefined logging appenders.

CACHEBOX By ColdBox

A highly-versatile caching aggregator and enterprise cache that allows for multiple named caching stores as well as granular control over cache behaviors and eviction policies. CacheBox can interface out of the box with Ehcache, Couchbase, Adobe ColdFusion cache and any Railo cache.

WIREBOX By ColdBox

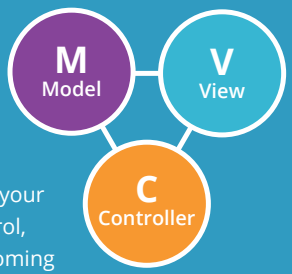
Managing your domain objects has never been easier with this Dependency Injection framework. WireBox supports all forms of injection as well as maintaining scope for your domain objects. WireBox can also interface and provide Java classes, web services, CacheBox integration and aspects of the ColdBox framework itself. WireBox also has built-in Aspect Oriented Programming (AOP) support, you'll never need another Dependency Injection engine again.

MOCKBOX By ColdBox

ColdBox encourages unit and integration testing as part of your development. That's why we provide MockBox which is a mocking and stubbing library designed to get your tests up and running fast without tediously creating mocks by hand.

WHAT IS MVC?

MVC is a popular design pattern called Model View Controller which seeks to promote good maintainable software design by separating your code into 3 main tiers.



The Model is the heart of your application. Your business logic should mostly live here in the form of services, entities, and DAOs.

Controllers are the traffic cops of your application. They direct flow control, and interface directly without incoming parameters from form and URL scopes. It is the controller's job to communicate with the appropriate models for processing, and set up either a view to display results or return marshalled data like JSON, XML, PDF, etc.

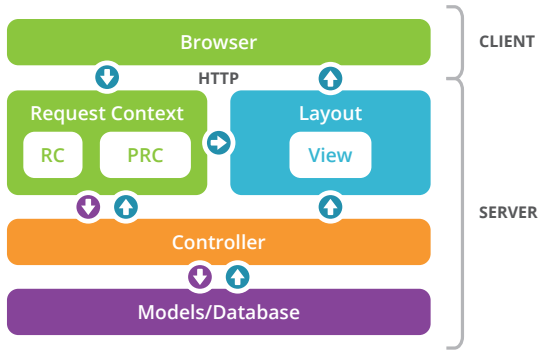
The Views are what the users see and interact with. They are the templates used to render your application out for the web browser. Typically this means HTML.

ColdBox embraces this standard and provides customizable conventions for how your app will be organized in a way that makes sense to programmers who have to maintain your code in the future.

THE REQUEST LIFECYCLE

A typical basic request to a ColdBox application looks like this:

1. HTTP(S) request is sent from browser to server `http://www.example.com/index.cfm?event=home.about`
2. Request context is created for this request and FORM/URL scopes are merged into a single request collection structure
3. Event (handler and action) are determined (handler is "home" and action is "about")
4. ColdBox event is run (`about()` method in `/handlers/home.cfc`)
5. The view set in the event is rendered (`/views/home/about.cfm`)
6. The view's HTML is wrapped in the rendered layout (`/layouts/main.cfm`)
7. Page is returned to the browser



INSTALLATION

Code Download links

To download the ColdBox Platform, visit `http://www.coldbox.org/download`. If you already have a ColdFusion engine installed, download the ColdBox Bundle and unzip it into a folder called "**coldbox**" in your web root. You should be able to immediately navigate to the "**samples**" directory to see some example apps in action. Once you're ready to start your own app, copy the contents of one of the application templates into your web root to provide you with an app skeleton containing the config file, `Application.cfc`, and a handler and view to get you started.

If you're brand new and starting from scratch, download DevBox from `http://www.coldbox.org/download/devbox` which is a complete, open source CFML stack based on Railo that will get you up and running in minutes with no installation. You can unzip the DevBox folder anywhere you like and click *start.exe*. For *nix/Mac, run "`chmod -R 777 *`" to make files executable, and run *start.sh*. The DevBox home page will be available at `http://localhost:8081/index.cfm` and will help you generate your own ColdBox applications in no time.

Common configuration settings

The first thing you'll want to check out in your new app is the config file. It's going to be located in `/config/ColdBox.cfc`. It is a simple CFC with a method called configure() that sets up several structures of settings for your application. The only required setting is coldbox.appName, but the docs outline all the possible configuration points that exist. Here are a handful of useful settings you're likely to want up front:

`coldbox.defaultEvent` - The default event to fire when you hit your application, defaults to *main.index*

`coldbox.debugMode` - Whether you want the ColdBox Debugger panel to be shown

`coldbox.handlerCaching` - Whether your event handlers (controllers) should be cached. Use false for development so you can see your development changes

`coldbox.handlersIndexAutoReload` - Whether to rescan your handlers directory or not. Use true for development so you can see your development changes

`settings` - Your custom application settings

More Info: `http://wiki.coldbox.org/wiki/Installation.cfm`

HANDLERS

The "C" in MVC stands for Controller, and ColdBox's controllers are called Event Handlers as they will be controlling events internally and externally. Handlers control the flow of your application. The code in your handlers interact with FORM, URL or REMOTE variables, interact with your model to get data and perform operations, and choose what views and layouts or data marshalling will need to happen to return a response to the client. Handlers are defined in ColdBox as regular CFC files (No XML!) and by convention, they will be located in the `/handlers` directory. The file `/handlers/main.cfc` represents a handler called "main".

ColdBox is an event-driven framework. That means that the framework's handlers define any number of named events which can be executed internally or externally. When a request comes in to the web server, it is mapped to an event which will be executed. Event names are in the form of `[package].handler.action`, where an action is just a method in that specific handler CFC. Examples would be `general.contact` or `inventory/product/details`.

Each action (or method) in a handler receives a reference to the current Request Context object, the request collection and the private request collection structures, which are passed for convenience. The Request Context object (commonly called "event") represents all the data for this client request including views to be rendered, etc. The request collection (commonly called "rc") is a struct containing the combined FORM, URL, or REMOTE arguments. The private request collection (or "prc") is a second collection of variables you can use to store additional data needed to process the request which has no ties to the outside world; thus being secure and private.

Part of the flow control available to a handler is the ability to redirect to another event. Handlers have a method called `setNextEvent()` available that can be used to redirect the user's browser to another part of your application.

```
setNextEvent('users.login');
```

Another way to reuse code is to run additional events from within an outer event, you can even pass a-la-carte arguments to that event call. There is no limit to the number of events you can run or how far you you can nest the calls. This makes it very easy to wrap up chunks of controller logic and run it anywhere you like.

```
runEvent(event='utility.attachmentUpload',
         eventArguments = { process=true });
```

You can also use the "private" argument to execute private methods that are not accessible from the outside world.

Sample Handler Code

In addition to rendering an HTML view, event handlers can also marshall data to return it as JSON, XML, TEXT, PDF etc. These data rendering capabilities are just one of the many features that are used for building RESTFul web services:

contacts.cfc

```
function index(event,rc,prc){
    prc.users = getModel('UserService').getUsers();
    event.setView('contacts/index');
}
function delete(event, rc, prc) {
    getModel('UserService').delete(event.getValue('id',0));
    setNextEvent('contacts.index');
}
function usersAsJSON(event, rc, prc) {
    var qUsers = getModel('UserService').getUsers();
    event.renderData(type='json', data=qUsers);
}
function sayHello(event,rc,prc){
    return '<h1>Hello From ColdBox Land</h1>';
}
```

More Info: http://wiki.coldbox.org/wiki/EventHandlers.cfm

REQUEST CONTEXT

We mentioned the Request Context object earlier as it is what gets passed into each action as the "event" argument. A new context is created for every client server request. It holds the name of the current event being handled (`contacts.aboutUs`), and a "request collection" of all FORM, URL, and REMOTE parameters combined. The handler interacts with the Request Context object to set views, set layouts, set data rendering options, interact with the request collection, and many other utility functions.

Here are some common uses of the requestContext object that you might see in an event h andler or view:

```
event.getValue('productID');
event.getValue('firstName','default value');
event.setValue('email','foo@bar.com');
event.getValue(name='productID', private=true);
event.setView('contacts/addNew');
event.setLayout('layout.popup');
event.getCurrentEvent();
<a href="#event.buildLink('contacts.index')#">link</a>
```

More Info: http://wiki.coldbox.org/wiki/RequestContext.cfm

VIEWS

Views are a large part of the trinity of MVC. They are templates for the HTML or data you'll be sending back to the client. In ColdBox we once again fall back on conventions. Views are cfm files stored in the `/views` folder and when you reference them, you leave off the extension. When the framework renders a view, it will make available to it our Request Context event object, as well as the request collections and a host of other utility methods such as `getSetting()`, `getfwLocale()`, etc.

Another handy convention that ColdBox allows for is implicit views. That means that if you don't explicitly set a view in your handler, ColdBox will look for a view with the same name as the action it just executed in a folder with the same name as the handler. For example, the event products.detail would automatically look for `/views/products/detail.cfm` if you didn't set a specific view. Setting a view is easy. Just use the `setView()` method on the Request Context object.

```
event.setView('products.soldOutMessage');
```

Views have automatic access to the same event, rc, prc objects and utility methods that the event handlers have as well. All of them are already available for you, you don't have to do anything to use them. We also give you an object reference called html, which is our HTML Helper plugin that can help you build HTML and HTML 5 elements with style and less verbosity.

```
#event.getValue('loggedInUser','Not Logged In')#
#HTMLEditFormat(rc.comment)#
#html.table(data=query)#
#html.emailField(name="Email", bind=prc.user)#
#html.img('includes/images/picture.png')#
#html.video('includes/assets/coldbox.avi')#
#dollarFormat(prc.totalPrice)#
#getModel('userService').getLoggedInUserName()#
#addAsset('includes/js/jquery.js, includes/css/awesome.css')#
```

More Info: http://wiki.coldbox.org/wiki/layouts-views.cfm

LAYOUTS

Layouts fall under the "View" part of MVC, but are reusable content wrappers or skins that can be used to enclose the specific HTML generated from your event. You can have an many layouts as you want defined in your application. A layout consists of a .cfm file located in your layout convention directory which is `/layouts` by default. The default layout by convention is called main.cfm.

Layouts can consume views and even other layouts to produce great rendering schemas or simualted layout inheritance.

Layouts usually include any menus, headers, and footers that you want on every page that uses that layout. Your layout code has full access to the ColdBox framework, can run events, and can render views or other layouts to conveniently organize and re-use code. Since a layout generally wraps the view declared in the handler, you need to tell the layout where to put the output from the view by calling `renderView()` with no arguments. You can also use the renderView(view) to generate a-la-carte view renderings as well.

You can configure a default layout to be used in your ColdBox config like so:

```
// Layout Settings
layoutSettings = {
    defaultLayout = "contentWrapper.cfm"
}
```

If you want to render a specific layout for an event, you can set one explicitly from the handler's action method. This will override any default layout that may be configured.

```
function home(event, rc, prc){
    // logic here
    // set view
    event.setView('general/home');
    // set layout
    event.setLayout('homePageLayout');
    // Set view and Layout
    event.setView(view='general/home',
                  layout='homePageLayout');
    // Set view with no layout
    event.setView(view='general/home', noLayout=true);
}
```

A simple layout could look like this:

```
#html.docType()#
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=UTF8" />
<title><cfoutput>#rc.title#</cfoutput></title>

<-- SES URLs base -->
<cfoutput>
<base href="#getSetting('htmlBaseURL')#">
</cfoutput>
</head>

<body>
<cfoutput>

<!--- Render a header view here and cache it for 30 minutes.
--->
#renderView(view='tags/header',cache=true,cacheTimeout='30')#

<div id="content">
    <!--- Use a plugin helper to render a UI messagebox --->
    #getPlugin('MessageBox').renderit()#
    <!--- Render the set view below: NO Arguments --->
    #renderView()#
</div>

<!--- Render the footer and also cache it --->
#renderView(view='tags/footer',cache=true,cacheTimeout='30')#
</cfoutput>
</body>
</html>
```

More Info: http://wiki.coldbox.org/wiki/Layouts-Views.cfm#Layouts

ENVIRONMENT CONTROL

One of the most common server configurations is to have a production server and then 1 or more development or testing servers. The trick with your "lower" environments is you typically want different settings for logging, error messages, data sources, or outgoing E-mails. Manually switching settings when you move code is sketchy at best and setting up deployment scripts can be more work than you're willing to take on.

Enter ColdBox Environment Control. ColdBox makes it easy to have different settings for each environment. In your configuration CFC, you have a `configure()` method that creates several structs of setting variables. Let's consider these our default production values. Next, all you do is create a method for each additional environment such as `development()`, `stage()`, etc. ColdBox will automatically call the appropriate environment override after the main `configure()` method and you can add, remove, or override settings for that environment as you see fit.

The default way ColdBox detects what environment it is running on is set up in the "environments" setting struct which declares a list of regular expressions to match against the URL to determine the environment. When not in production, the appropriate method, such as `development()` or `stage()`, will be called where it can override or add settings as it sees fit.

ColdBox.cfc

```
component {
    function configure() {
        coldbox = {
            setting1 = 'value1',
            setting2 = 'value2',
            setting3 = 'value3'
        };
        environments = {
            development = "^dev\.,^local\.",
            stage = "^stage\.,^test\."
        };
    }
    function development() {
        coldbox.setting1 = 'devValue';
        arrayAppend(interceptors, {
            class = 'coldbox.system.interceptors.
                    ColdboxSidebar' } );
    }
    function stage() {
        coldbox.setting1 = 'stageValue';
    }
}
```

If you don't want to use URL to determine your environment you can create a method called `detectEnvironment()` and simply have it return a string corresponding with the correct environment for that server. You can base off the machine name, IP address, or even the location of the code on the file system.

More Info: http://wiki.coldbox.org/wiki/ConfigurationCFC.cfm#environments

PLUGINS

Plugins are a ColdBox-centric way to easily encapsulate bits of handy logic for your own site organization or to share with other ColdBox devs. The ColdBox Platform ships with a number of core plugins which provide a rich tool set for developers to use. Only plugins that are used get loaded into memory. The framework also automatically handles persistence of all plugins as well. In addition to keeping the framework as modular as possible, core plugins can be extended to your pleasing to override default behavior

Tip: ForgeBox is a great place to upload your plugins to so the whole community can use them (www.coldbox.org/forgebox)

Here's are some of the core ColdBox plugins and what they do:

AntiSamy - Allows you to use the OWASP anti forgery and XSS cleanup library in your ColdBox applications

Cookie Storage - A facade to the cookie scope. It will also take care of JSON serializations for complex variables for you. It can also use encryption and other goodies

Session Storage - A session scope manager and facade with some extra functionality

File, Date, JVM Utils - A useful collection of file related utility methods

i18n, Resource Bundle - Allows you to talk to languages & resource bundles in your application and be able to switch and load according to user's locales

Message Box - A UI plugin to help render message boxes on your applications for errors, warnings, and informational messages.

Query Helper - A utility helper to sort, filter, append, union, join queries on-the-fly

ORM Service - A facade to our ColdBox enhanced ORM Services that will allow you to leverage a virtual service layer for ORM entity operations

HTML Helper - Allows you to build uniform & DRY HTML elements with ORM binding capabilities, data capabilities, HTML5 integrations, asset management and so much more.

Using a plugin is easy. If you're inside of a view, handler, layout, etc, just call `getPlugin("pluginName")`.

```
#getPlugin("messagebox").renderit()#
```

If you need your plugin in a model object that is being provided by WireBox, it can be autowired with the following DSL:

```
component {

    property name='myPlugin' inject='coldbox:plugin:name';

    function methodName() {
        // Use our injected plugin instance
        myPlugin.doSomething();
    }
}
```

Writing your own plugin is also easy. A ColdBox plugin is simply a CFC that is placed in the plugins convention directory which is /plugins by default. If you provide an init() method, you can set some meta properties in it such as name, author, or description. Plugins inherit from the framework supertype so they have access to framework services. Since plugins are created by WireBox they can also be autowired with dependencies.

Here is a sample plugin:

```
component singleton {

    function init() {
        setPluginName("My Awesome Plugin");
        setPluginAuthor("John Doe");
        setPluginVersion("1.0");
        setPluginDescription("A very cool plugin");
        return this;
    }

    function awesomeSauce(input) {
        // Do awesome stuff here
    }
}
```

More Info: http://wiki.coldbox.org/wiki/Plugins.cfm

ROUTES

By default, a ColdBox URL will look like this:

```
http://mysite.com/index.cfm?event=myHandlerName.myActionName
```

That URL would run the "myActionName" method in the "myHandlerName" handler. We can activte the SES interceptor which enables URL routing. Now, the following URL does the same thing:

```
http://mysite.com/index.cfm/myHandlerName/myActionName
```

Of course, a simple web server rewrite rule can simplify that even further to:

```
http://mysite.com/myHandlerName/myActionName
```

That URL pattern is called a "route" and not only can be changed, but you can add as many customized routes as your application desires. The magic happens in your `/config/routes.cfm` file where all the routes are declared in the order they should be processed. You'll find the default route in that file:

```
addRoute(pattern=":handler/:action?");
```

The syntax for adding a new custom route looks like this:

```
addRoute(pattern="/URLStringToMatch", handler="handlerName",
action="actionName");
```

That means, you could set up a route like so:

```
addRoute(pattern="/blog", handler="entryHandler",
action="listEntries");
```

GIven the above route, the following two URLs would point to the exact same thing (with the second option being obviously preferable):

```
http://mysite.com/entryHandler/listEntries
http://mysite.com/blog
```

You can even clean up URLs that have variables. Consider the following page on your site:

```
http://mysite.com/services/products/productDetail/productID/1
```

Just add a new route that looks like this:

```
addRoute(pattern="/product/:productID", handler="services.
products", action="productDetail");
```

And now the following URL will give you the same page:

```
http://mysite.com/product/1
```

As you can see, routes are a very powerful way to keep your URLs looking sharp. They can also be expanded to respond to RESTFul routing.

More Info: http://wiki.coldbox.org/wiki/URLMappings.cfm

MODULES

One of the great ways that ColdBox lets you organize your code is via modules. Modular MVC is far more that just reusable plugins. They allow you to take an entire slice of your application; models, views, controllers, and related config and drop them into a tidy self-contained folder. This lets developers avoid monolithic apps and organize related chunks of their applications in a modular way. In addition, modules can be shared with the community so other developers can add functionality to a ColdBox site by dropping in the related module.

Examples of ColdBox Modules are:

Amazon S3 Explorer - A cool ColdBox explorer for Amazon S3 accounts.

Filebrowser - An awesome file browser and manager.

ContentBox Modular CMS - A modern modular content management engine.

A module is contained inside a folder that you drop in your "modules" directory. It stores it's configuration in a file at its root called `ModuleConfig.cfc`. Also inside, are folders for views, handlers, models, etc. The complexity or simplicity of your modules are up to you.

More Info: http://wiki.coldbox.org/wiki/Modules.cfm

INTERCEPTORS

One of the most powerful parts of the ColdBox framework are interceptors. They allow you to follow an event-driven model of coding that helps decouple the pieces of your application. Interception points are events that happen over the life cycle of your application or a user request which are broadcast by the framework. Examples would be

`preProcess` - When a request first comes in

`onException` - When an error happens

`preViewRender` - before a view is rendered

If you have any code you want to execute at those points, or if you want to override/modify the default behavior of the framework, you can just register one or more interceptors to listen for that broadcasted event.

Interceptors, like most everything else in ColdBox, are implemented as simple CFCs which have a `configure()` method that is called on their creation. Then you create as many methods as you want named after the interception points you want to respond to.

/interceptors/GateKeeper.cfc

```
component {
    property name="securityService" inject="model";

    function configure() {}

    function preEvent(event, interceptData) {
        if(interceptData.processedEvent == 'secure.page' &&
                        !securityService.loggedIn()) {
            setNextEvent("login.page");
        }
    }
}
```

Then, all you have to do is register your interceptor in the config like so:

```
interceptors = [
  {class="interceptors.GateKeeper", properties={}}
];
```

Now, the code in our `preEvent()` method will get called upon to run before each event in your application. This keeps cross cutting concerns nicely encapsulated in a way that can easily be turned on and off without actually touching the parts of the app that they are listening to.

As if all the built-in ColdBox interception points weren't cool enough, you can create your own custom interceptions in your app like `orderCreated`, `userLoggedIn`, or `accountDeleted` and then write interceptors that listen for them and do special logic. You can even announce events asynchronously, how cool is that!

Once you declare your custom interception points in the config file, you can announce them like so:

```
var interceptData = {orderNumber=rc.orderNumber};
announceInterception('orderCreated', interceptData);
```

More Info: http://wiki.coldbox.org/wiki/Interceptors.cfm

ERROR HANDLING

ColdBox automatically wraps every request in a try/catch block to capture errors in either the framework or your application code. If you do nothing, ColdBox will render a configurable error template. The default error template is located in `\coldbox\system\includes\BugReport.cfm`

You can supply a custom error template in the ColdBox configuration file which is recommended on production so your users see a nice message and not the guts of your code.

```
customErrorTemplate = "includes/callCustomerService.cfm"
```

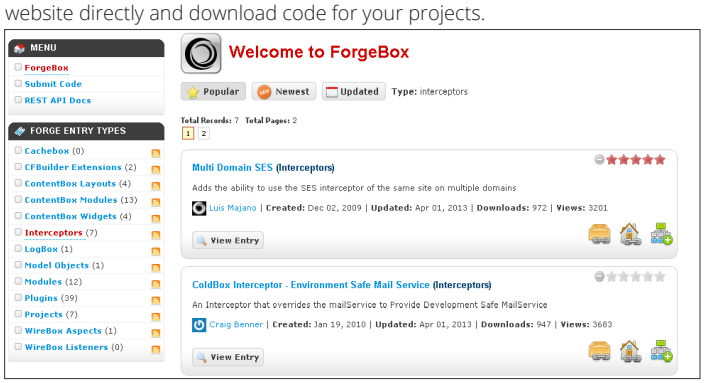
One of the several ways you can handle errors is with an onException interceptor. Just register a CFC in the "interceptors" array of your config that has an `onException()` method. Any errors captured in your application will be announced to your interceptor where you can do custom logging, redirects, or handling.

Sample interceptor

```
component {
    function onException(event, interceptData) {
        local.e = interceptData.exception;
        if (e.type == 'user_logged_out') {
            setNextEvent('main.login');
        } else {
            log.error(e.message, e);
            setNextEvent('sorry.callCustomerService');
        }
    }
}
```

FORGEBOX

ForgeBox is the online code-sharing community for any of our frameworks and products. It is located at http://www.coldbox.org/forgeBox and contains many user-submitted modules, plugins, interceptors, and more. ColdBox is an open source platform and we encourage everyone to share code for the benefit of the community. To use ForgeBox, you can browse the website directly and download code for your projects.



ForgeBox also has a public REST API. For example, here is the URL for the entries endpoint formatted as JSON:

```
http://www.coldbox.org/api/forgebox/json/entries
```

This powers the ForgeBox Module that you can drop right into your application to browse for entries and download them directly to your web root.

A small list of projects on ForgeBox include

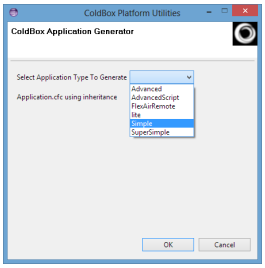
- **Amazon S3** Plugin
- **JSMin Compressor** Plugin
- **ContentBox Modular CMS** Modules
- **HTML Compressor** Interceptor
- **Bitly URL Shortener** Plugins

IDE EXTENSIONS

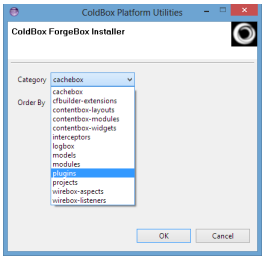
If you are using Adobe's ColdFusion Builder IDE to code in, we offer a free extension called the ColdBox Platform Utilities which adds context menus into the editor that helps you with common development tasks

More Info: http://wiki.coldbox.org/wiki/CFBuilderExtensions.cfm

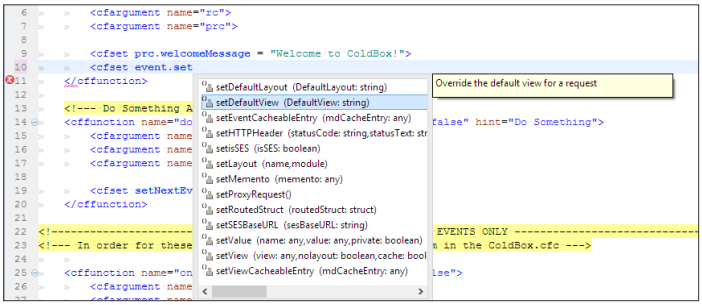
When starting a new project, right click on the project and choose "New Application". This wizard will copy a working app skeleton into your web root.



The platform utilities also tie into our ForgeBox API and will allow you to browse and download entries right into your app.



We also have custom syntax libraries for a number of IDEs including Builder, CFEclipse, DreamWeaver, and Sublime that give you code completion for the internal ColdBox classes



More Info: http://wiki.coldbox.org/wiki/SyntaxDictionaries.cfm

The ColdBox Framework is completely documented in our online wiki located at http://wiki.coldbox.org/

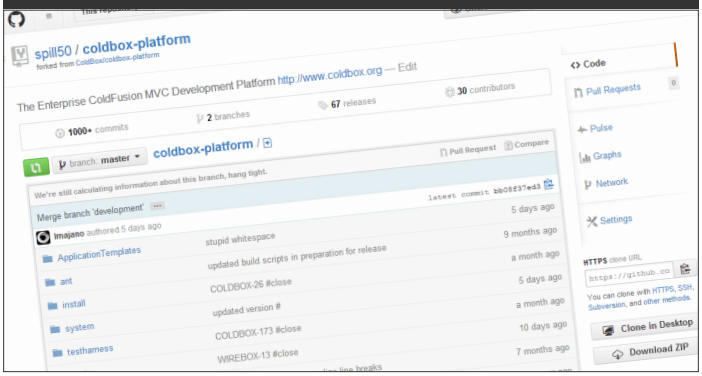
For API docs to see class definitions and method signatures, please visit the API docs located at http://www.coldbox.org/api

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

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

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

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



An Introduction to the ColdBox Platform

Ortus Solutions
A Software Revolution

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

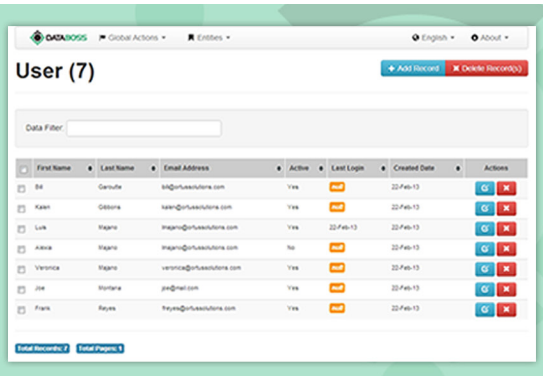
Support Program

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

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

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

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



"Build dynamic ORM administrators in seconds"

DATABOSS
Dynamic Administrator

www.data-boss.com

www.coldbox.org | www.ortussolutions.com

First Edition - July 2013