# An Introduction to the ColdBox LITE

www.coldbox.org

Covers up to version 1.1.0

**Contents**



**ColdBox LITE (CBL) is a lightweight 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 LITE itself is Professional Open Source Software, backed by Ortus Solutions which provides support, training, architecture, consulting and commercial additions.

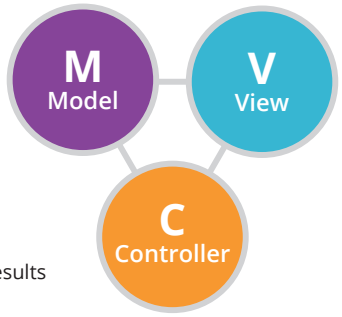
ColdBox LITE's big brother is the ColdBox Platform that contains all the functionality of the LITE core as well as a myriad of Plugins, and libraries for caching, logging, modularization, integration testing, AOP (aspect oriented programming), mocking/stubbing and more.

ColdBox LITE aims to provide a simple MVC framework to help get smaller apps up and running quickly.  ColdBox LITE is also handy if you already have a collection of libraries and utilities you want to use for your application and you're just looking for MVC conventions.  You can always seamlessly upgrade to the full ColdBox Platform if you want to leverage more than just MVC. You can see our complete list of features here

`http://wiki.coldbox.org/wiki/cbl.cfm#What's_not_included.3F`

## 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.

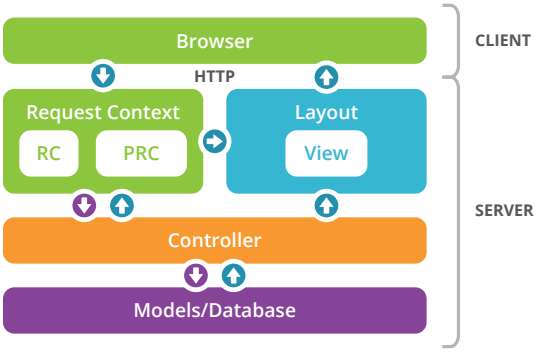
The Views are what the users see and interact with. They are the templates used to render your application out for the web browser. Typically this means HTML.

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 LITE 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 ColdBox LITE, visit `http://www.coldbox.org/download`

ColdBox LITE comes in two flavors:

**ColdBox LITE MVC** - The standard ColdBox LITE MVC package

**ColdBox LITE MVC + ORM** - The standard ColdBox LITE MVC package with all of our fancy ORM builders and helpers. Use this one if you're planning on using ORM in your app.

If you already have a ColdFusion engine installed, download the version of ColdBox LITE MVC you want and unzip the contents into your web root. You can also create a mapping "`/coldbox`" that points to the location of the downloaded package so you can place ColdBox LITE off the web root for added security. Also download the ColdBox LITE Samples and place the contents in your web root. You can then navigate directly to the sample apps to check them out like `http://localhost/samples/chuckNorris`

**Note**: that the ContactManager and ActiveContactManager sample apps require the ORM version of ColdBox LITE.

Once you're ready to start your own app, copy the contents the LITE application template or a sample app 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 create your own ColdBox LITE 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 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, by convention the default event is `main.index`

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

`settings` - Your custom application settings

After making configuration changes, you'll need to reinitialize the framework by typing `?fwreinit=1` on the end of the URL:

`http://www.mysite.com/index.cfm?fwreinit=1`

**Note**: adding fwreinit=1 to the end of any ColdBox LITE application URL will restart your application to first request state, recreate model objects, clear caches and much more. This is your handy best friend URL action.

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

# HANDLERS

The "C" in MVC stands for Controller, and ColdBox LITE'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 LITE 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 the method `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 });
```

**Sample Handler Code**

In addition to rendering an HTML view, event handlers can also marshall data to return it as JSON, XML, PDF etc. This 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){
     rc.qUsers = getModel('UserService').getUsers();
     event.setView('contactList');
}
function delete(event, rc, prc) {
     getModel('UserService')
          .deleteUser(event.getValue('userID'));
     setNextEvent('contacts.index');
}
function usersAsJSON(event, rc, prc) {
     event.renderData(type='json', data=getUserService()
          .getUsers());
}
function sayHello(event, rc, prc){
     return '<h1>Hello #event.getValue("name","cbl")#, how
are you today?</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 handler or view:

```
event.getValue('productID');
event.getValue('firstName','default value');
event.setValue('email','foo@bar.com');
event.setView('contacts/addNew');
event.setLayout('layout.popup');
event.renderData();
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 LITE 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 handful of utility methods such as `getSetting()`, and `getModel()`.

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 don't set a specific view.

Setting a view is easy. Just use the `setView()` method on the event object in your handler.

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

Views have automatic access to the same event, rc, prc objects and utility methods that the event handlers do. All of them are already available for you, you don't have to do anything to use them.

```
#event.getValue('loggedInUser','Not Logged In')#
#HTMLEditFormat(rc.comment)#
#dollarFormat(prc.totalPrice)#
#getModel('userService').getLoggedInUserName()#
// You can render a sub-view inside of another view
#renderView(view='pods.news', args={ title='News',
gadgets=true })#
```

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

## LAYOUTS

Layouts fall under the "View" part of MVC, but are reusable content wrappers 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 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 LITE framework, can run events, and can render views or other layouts to conveniently organize and reuse 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 configure a default layout to be used in your ColdBox LITE application 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:

```
<cfoutput>
<!DOCTYPE html >
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html;
charset=UTF8" />
    <title>#rc.title#</title>
</head>
<body>
    <!--- Render a header view here --->
    #renderView(view='tags/header')#
    <div id="content">
        <!--- Render the set view below: NO Arguments --->
        #renderView()#
    </div>
    <!--- Render the footer --->
    #renderView(view='tags/footer')#
</body>
</html>
</cfoutput>
```

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

## MODEL INTEGRATION

ColdBox LITE comes bundled with WireBox, our Dependency Injection library that helps you manage your models such as services,entities, beans, or DAOs. WireBox is capable of not only creating these objects at your beckon call, but injecting them with their dependencies and persisting them automatically. The model convention folder is called "model". You can place your CFCs in that folder, and WireBox will find them when you ask.

Consider the following CFCs:

/model/UserService.cfc

/model/UserBean.cfc

/model/util/Formatter.cfc

You can access then in a view, layout, or handler like this:

```
getModel("UserService").getUsers();
getModel("UserBean").doSomething();
getModel("util.Formatter").formatSSN()
```

Place the annotation "singleton" in any CFC (handlers or models) that should only exist once for your entire application. If you also want one of your CFCs to be created with a reference to another CFC, you can use properties at the top of the target component.

UserService.cfc

```
component singleton {}
```

UserBean.cfc

```
component scope="session" {
    property name='UserService' inject='model:UserService';
    property name='formatter' inject='util.formatter';
}
```

UserHandler.cfc

```
component {
    property name='UserService' inject='model:UserService';
}
```

For more information on all the amazing things you can do with WireBox, check out the WireBox docs, or our WireBox Reference Card.

**More Info:** http://wiki.coldbox.org/wiki/Wirebox.cfm

## ENVIRONMENT CONTROL

One of the most common server configurations is to have a production server and then one 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 LITE Environment Control. ColdBox LITE 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 LITE 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 LITE 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'
        };
                                                    // cont.
```

```
        environments = {
            development = "^dev\.,^local\.",
            stage = "^stage\.,^test\."
        };
    }
    function development() {
        coldbox.setting1 = 'devValue';
    }
    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

## ORM SERVICES

You can always use ColdFusion's ORM with ColdBox LITE MVC only, but if you downloaded the version of ColdBox LITE that comes with our ORM Services, you have some powerful tools to help you deal with your ORM entities. One of these tools is the Virtual Entity Service, which is a service layer that we have pre-built for you with tons of convenience methods for interacting with the binded entity. You can do listing, finding, getting, dynamic counters/finders, get metadata, transaction safe deletes, saves and updates and so much more. If you have a handler that needs to deal with a "user" ORM entity, inject an entity service for your user like this:

```
property name="UserService" inject="entityService:User";
```

Then you can call it's convenience methods like so:

```
UserService.list(
    sortBy="fname",
    offset=event.getValue("startrow", 1),
    max=20);
UserService.countWhere(isActive=true, married=false);
UserService.new(properties={
    firstName="John", lastName="Doe", age="25"});
UserService.delete(user);
```

Another powerful feature are Criteria Queries which let you dynamically build SQL statements using an expressive set of chainable methods.

```
c = UserService.newCriteria();
// Fetch users whose first name starts with "Jo" from Accounting
// who are between the age 20 and 30. Order by last name desc
// and show 50 records starting at the 20th record.


                                                    // cont.
```

```
var results = c.like("firstName", "Jo%")
    .order("lastName", "desc")
    .and(
        c.restrictions.between( "age", 20, 30),
        c.restrictions.eq("department", "Accounting")
    )
    .list(max=50, offset=20);
```

Please see our docs to learn more about the many other things you can do.

**More Info:** http://wiki.coldbox.org/wiki/Extras:BaseORMService.cfm

## ERROR HANDLING

ColdBox LITE automatically wraps every request in a try/catch block to capture errors in either the framework or your application code. You can register an exceptionHandler setting which points to the event you want executed in case of an error. That event can access the exception in the request collection and decide what to do with it.

You can also specify an onInvalidEvent setting which points to the event you want executed when someone hits a bad URL that doesn't map to a handler or action.

If you don't have either of the above settings, or they throw an error of their own, ColdBox LITE will look for a customErrorTemplate file to include. This is not a view, but just a stand-along cfm file that will be executed.

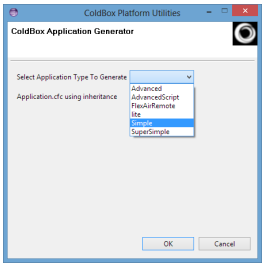
```
coldbox = {
    exceptionHandler = "main.onException",
    onInvalidEvent = "main.pageNotFound",
    customErrorTemplate = "includes/templates/generic_error.cfm"
};
```

## IDE EXTENSIONS

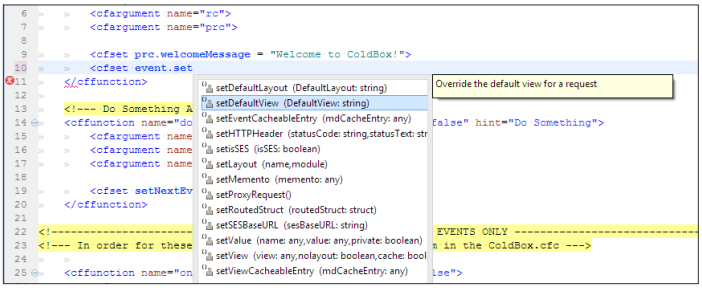
If you are using Adobe's ColdFusion Builder IDE to code in, we offer a free extension called the ColdBox Platform Utilities which adds context menus into the editor that helps you with common development tasks

**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.




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.



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

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

ColdBox

# An Introduction to ColdBox LITE

Ortus Solutions
A Software Revolution

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

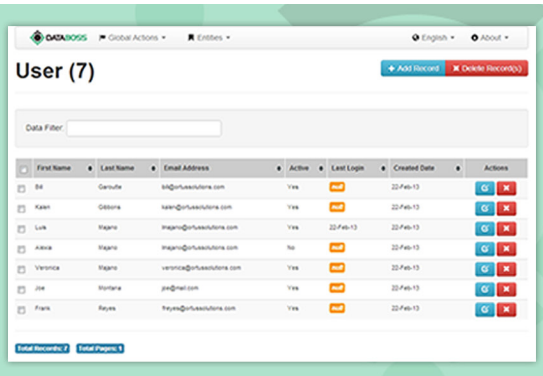
## Support Program

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<br>4 tickets | 10<br>20 tickets | 30<br>60 tickets | 80<br>160 tickets | 150<br>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 - August 2013