

[<< Back to Dashboard](#)

ColdBox Configuration CFC

Covers up to version 3.5.0

Overview

The ColdBox configuration CFC is the heart of your ColdBox application. It contains the initialization variables for your application and extra information used by the ColdBox software aspects and ultimately how your application boots up.

Note: You can find various sample configuration objects in the bundle download and viewing the *ApplicationTemplates* folder.

Introduction

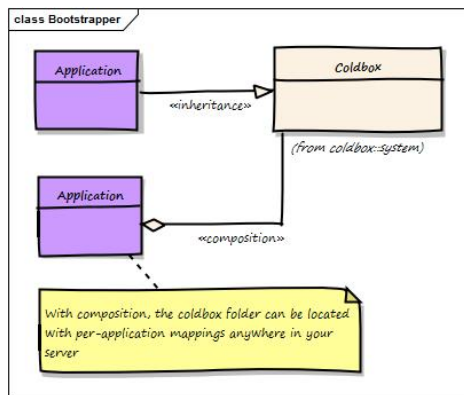
The configuration CFC is a simple CFC that lives in your *config* folder by convention called *Coldbox.cfc*. This is a convention that is setup by default, but of course you can alter the location by tweaking some variables in the *Application.cfc* provided in the templates. So before we start digesting what we can do in our configuration object, let's explore these directives as they are extremely important.

```
+ Application.cfc
+ /config
+ Coldbox.cfc
```

Application.cfc Directives

The provided *Application.cfc* object in the application templates can come in two flavors:

1. Using inheritance
2. Using Composition (**Our Preference**)



The inheritance approach is the easiest way to start but it is bound to extending the *Application.cfc* object to the ColdBox bootstrapper. With the no-inheritance or composition approach it is much more flexible and we can use per-application mappings for choosing the location of the **coldbox** distribution package. Anyways, once you make this decision there are some directives or variables you can set that will alter some behavior.

Variable	Type	Required	Default	Description
COLDBOX_APP_ROOT_PATH	string	true	<code>getDirectoryFromPath(getCurrentTemplatePath())</code>	Automatically set for you. This path tells the framework what is the base root location of your application and where it should start looking for all the agreed upon conventions. You usually will never change this, but you can.
COLDBOX_APP_MAPPING	string	DEPENDS	---	The application mapping is ESSENTIAL when dealing with Flex or Remote applications. This is the location of the application from the root of the web root. So if your app is at the root, leave this setting blank. If your application is embedded in a sub-folder like <i>MyApp</i> , then this setting will be <i>/MyApp</i> .
COLDBOX_CONFIG_FILE	string	false	<code>config/Coldbox.cfc</code>	The absolute or relative path to the configuration CFC file to load. This bypasses the conventions and uses the configuration file of your choice.
COLDBOX_APP_KEY	string	false	<code>cbController</code>	The name of the key the framework will store the application controller under in the <i>application</i> scope.

Contents

- [ColdBox Configuration CFC](#)
 - [Overview](#)
 - [Introduction](#)
 - [Application.cfc Directives](#)
 - [Creating the ColdBox.cfc](#)
 - [The Decorated Variables](#)
 - [The Configure\(\) Method](#)
 - [The Simplest Configuration](#)
 - [Configuration Storage](#)
 - [Configuration CFC As An Interceptor](#)
 - [cacheBox](#)
 - [coldbox](#)
 - [Application Setup](#)
 - [Development Settings](#)
 - [Implicit Event Settings](#)
 - [Extension Points Settings](#)
 - [Exception Handling](#)
 - [Application Aspects](#)
 - [conventions](#)
 - [datasources](#)
 - [debugger](#)
 - [environments](#)
 - [Custom Environment Detection](#)
 - [flash](#)
 - [i18n](#)
 - [Good i18n Resources](#)
 - [Sample resource bundle](#)
 - [interceptorSettings](#)
 - [interceptors](#)
 - [ioc](#)
 - [layoutSettings](#)
 - [layouts](#)
 - [logBox](#)
 - [Appender Definition](#)
 - [Root Logger](#)
 - [Categories](#)
 - [Implicit Categories](#)
 - [mailSettings](#)
 - [Available Protocols](#)
 - [FileProtocol Properties](#)
 - [PostmarkProtocol Properties](#)
 - [modules](#)
 - [orm](#)
 - [settings](#)
 - [validation](#)
 - [webservice](#)
 - [wirebox](#)
 - [Interacting With The Loaded Settings](#)

Most of the time you will never alter these settings if your application is the root application and it will look like the following:

```
<!-- COLDBOX STATIC PROPERTY, DO NOT CHANGE UNLESS THIS IS NOT THE ROOT OF YOUR COLDBOX APP -->
<cfset COLDBOX_APP_ROOT_PATH = getDirectoryFromPath(getCurrentTemplatePath())
<!-- The web server mapping to this application. Used for remote purposes or static purposes -->
<cfset COLDBOX_APP_MAPPING = "">
<!-- COLDBOX PROPERTIES -->
<cfset COLDBOX_CONFIG_FILE = "">
<!-- COLDBOX APPLICATION KEY OVERRIDE -->
<cfset COLDBOX_APP_KEY = "">
```

Or you can modify it, like the following example in order to load a custom configuration file and a custom application mapping.

```
<!-- COLDBOX STATIC PROPERTY, DO NOT CHANGE UNLESS THIS IS NOT THE ROOT OF YOUR COLDBOX APP -->
<cfset COLDBOX_APP_ROOT_PATH = getDirectoryFromPath(getCurrentTemplatePath())
<!-- The web server mapping to this application. Used for remote purposes or static purposes -->
<cfset COLDBOX_APP_MAPPING = "apps.myApp">
<!-- COLDBOX PROPERTIES -->
<cfset COLDBOX_CONFIG_FILE = "shared.myApp.coldbox">
<!-- COLDBOX APPLICATION KEY OVERRIDE -->
<cfset COLDBOX_APP_KEY = "">
```

All of the directives have their appropriate getter and setter methods that you can use by calling it from the ColdBox bootstrapper object or via your *Application.cfc* using inheritance.

Creating the ColdBox.cfc

```
/**
 * A simple CFC that configures a ColdBox application. You can even extend, compose, strategize and do your OO goodness.
 */
component {

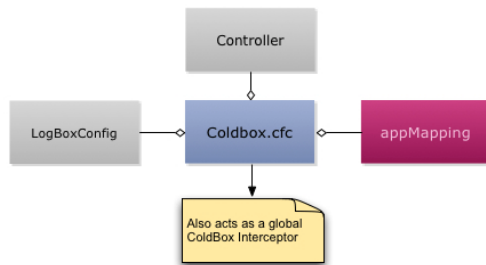
    // Mandatory configuration method
    void function configure(){

    }
}
```

This component is a simple CFC, no inheritance needed, because it is decorated with methods and variables by ColdBox at runtime. Once created, we must create a mandatory *configure()* method that will be executed by ColdBox in order to configure your application for production use. We will discover the optional methods later on, so don't worry. Another important facet of the configuration object is that it will be decorated at runtime with the following variables that will be placed in the component's *variables* scope:

The Decorated Variables

ColdBox.cfc



Property	Type	Description
controller	coldbox.system.web.Controller	A reference to the running ColdBox Application. You can call methods on this object, go crazy!
appMapping	string	The current running application mapping. You can use it to concatenate it with other settings so your application becomes really portable. Use it for prefixing interceptor declarations, paths, expand it, etc.

Note: Remember that the default scope in ColdFusion is the *variables* scope. So you can just refer to the decorated variables by their given property name.

The Configure() Method

As discussed previously we must implement one mandatory method called *configure()*. In this method is where we can set all the necessary ColdBox directives, settings, tier control, etc. If you are already familiar with configuring ColdBox applications via XML the programmatic approach will be very similar in terms of the settings and names. We really wanted to maintain some balance, so you will not see any weird new names. Basically, you have to create several structures or arrays in the *variables* scopes that must match the following names:

Variable	Type	Required	Description
cacheBox	struct	false	An optional structure used to configure CacheBox. If not setup the framework will use its default configuration found in <i>/coldbox/system/web/config/CacheBox</i>
coldbox	struct	true	The main coldbox directives structure that holds all the coldbox settings.
conventions	struct	false	A structure where you will configure the application convention names
datasources	struct	false	An optional metadata structure for datasource definitions. THEY ARE NOT COLD FUSION REGISTRATIONS

debugger	struct	false	An optional structure to configure the way the ColdBox debugger works
environments	struct	false	A structure where you will configure environment detection patterns
flash	struct	false	A structure where you will configure the FlashRAM
i18n	struct	false	An optional structure where you can configure your application for internationalization and localization
interceptorSettings	struct	false	An optional structure to configure application wide interceptor behavior
interceptors	array	false	An optional array of interceptor declarations for your application
ioc	struct	false	A structure where you can configure external dependency injection frameworks
layoutSettings	struct	false	A structure where you define how the layout manager behaves in your application
layouts	array	false	An array of layout declarations for implicit layout-view-folder pairings in your application
logBox	struct	false	An optional structure to configure the logging and messaging in your application via LogBox
mailSettings	struct	false	An optional metadata structure of mail settings
modules	struct	false	An optional structure to configure application wide module behavior
orm	struct	false	A structure where you can optionally configure ORM related features
settings	struct	false	A structure where you can put your own application settings
validation	struct	false	A structure where you can choose your form/object validation engine and store shared validation rules
webservice	struct	false	An optional metadata structure for webservice WSDL lookups
wirebox	struct	false	An optional structure used to define how WireBox is loaded

The Simplest Configuration

Before we start digesting each of these directive variables, let's look at the simplest of all configurations. Below is the simplest approach by only filling out the mandatory requirements:

```
function configure(){
  coldbox = {
    appName      =My Simple App"
    // defaultEvent = "main.index"
  };
}
```

There you go, that is the simplest that you can go with ColdBox, just the name of your application. The default event by convention is **main.index**. Just remember that ColdBox can be as simple as you like, or as complex as you like. ColdBox has lots of tools and software concerns that can address most typical ColdFusion applications, but its core is plain old simple conventions MVC. So go ahead, and build your own recipe!

Configuration Storage

Once the application starts up, a reference to the instantiated configuration CFC will be stored in the configuration settings with the key **coldboxConfig**. You can then retrieve it like so:

```
config = getSetting(coldboxConfig);
```

Having access to the reference can be useful.

Configuration CFC As An Interceptor

Another cool concept for the Configuration CFC is that it is also registered as a ColdBox [Interceptor](#) once the application starts up automatically for you. This means that you can create interception points in this CFC that will be registered upon application startup. This can be a great way for you to define startup procedures, listen to events, etc.

cacheBox

The **cacheBox** structure is based on the CacheBox declaration DSL, see [CacheBox](#), and it allows you to customize the caches in your application. Below are the main keys you can fill out, but we recommend you review the [CacheBox](#) documentation for further detail:

Key	Type	Required	Default	Description
configFile	file path	false	<i>config/CacheBox.cfc</i>	An absolute or relative path to the CacheBox configuration CFC or XML file to use instead of declaring the rest of the keys in this structure. So if you do not define a cacheBox structure, the framework will look for the default value: <i>config/CacheBox.cfc</i> and it will load it if found. If not found, it will use the default CacheBox configuration found in <i>/coldbox/system/web/config/CacheBox.cfc</i>
logBoxConfig	string	false	<i>coldbox.system.cache.config.LogBox</i>	The instantiation or location of a LogBox configuration file. This is only for standalone operation. Do not use when in use in a ColdBox application.
scopeRegistration	struct	false	{enabled=true,scope=server,key=cacheBox}	A structure that enables scope registration of the CacheBox factory in either <i>server</i> , <i>cluster</i> , <i>application</i> or <i>session</i> scope.

defaultCache	struct	true	---	The configuration of the default cache which will have an implicit name of default which is a reserved cache name. It also has a default provider of CacheBox which cannot be changed.
caches	struct	false	{ }	A structure where you can create more named caches for usage in your CacheBox factory.
listeners	array	false	[]	An array that will hold all the listeners you want to configure at startup time for your CacheBox instance. If you are running CacheBox within a ColdBox application, this item is not necessary as you can register them via the main ColdBox interceptors section.

Please note that there are certain flows for CacheBox configuration discovery if you do not use the inline CacheBox DSL. Please refer to the [CacheBox Application Configuration](#) section in order to learn it. Below is a snapshot of that documentation so you can get a feel of the inline CacheBox DSL. However, we recommend using an external portal file for configuring CacheBox, whether XML or programmatic (our choice!).

coldbox

The *coldbox* structure must be declared with the mandatory settings we saw in the previous section. Let's explore now the rest of the directives in our nice little charts below that are grouped by functionality:

Application Setup

Key	Type	Required	Default	Description
appName	string	true	---	The internal name of this application, used for internal locking, logging, etc. Make sure it does not have weird characters
eventName	string	false	<i>event</i>	The name of the incoming URL/FORM/REMOTE variable that tells the framework what event to execute. Ex: <i>index.cfm?event=users.list</i>

```
coldbox = {
  appName = "My App",
  eventName = "event"
};
```

Development Settings

Key	Type	Required	Default	Description
debugMode	boolean	false	false	Decides if debug mode will be set for ALL users at startup or not. Recommend to false in production. Once you go into debug mode an encrypted session only cookie is set for the user.
debugPassword	string	false	---	Protect the debug mode URL actions . We recommend setting one up for production.
reinitPassword	string	false	---	Protect the reinitialization of the framework URL actions . We recommend setting one up for production.
handlersIndexAutoReload	boolean	false	false	Will scan the conventions directory for new handler CFCs on each request if activated. Turn off for production.

```
coldbox = {
  debugMode = false,
  debugPassword = "whatUpDude",
  reinitPassword = "hlcker",
  handlersIndexAutoReload = true
};
```

Implicit Event Settings

Key	Type	Required	Default	Description
defaultEvent	event syntax	true	---	The second required setting. This points to a handler/method combination that the framework will execute when no incoming event is detected via URL/FORM or REMOTE executions.
requestStartHandler	event syntax	false	---	Fires at the beginning of every request.
requestEndHandler	event syntax	false	---	Fires at the end of every request.
applicationStartHandler	event syntax	false	---	Fires at the beginning of the 1st request.
applicationEndHandler	event syntax	false	---	Fires whenever the ColdFusion application expires and <i>onApplicationEnd()</i> is fired
sessionStartHandler	event syntax	false	---	Fires whenever a new ColdFusion session starts via <i>onSessionStart()</i>

sessionEndHandler	event syntax	false	---	Fires whenever a ColdFusion session ends via <i>onSessionEnd()</i>
missingTemplateHandler	event syntax	false	---	Executes whenever a CFML template is requested and not found in the application via <i>onMissingTemplate()</i>

Note: We recommend reviewing the MVC and Remote request lifecycles documentation: See [RequestLifecycles](#)

```
coldbox={
  //Implicit Events
  defaultEvent = "General.index",
  requestStartHandler = "Main.onRequestStart",
  requestEndHandler = "Main.onRequestEnd",
  applicationStartHandler = "Main.onAppInit",
  applicationEndHandler = "Main.onAppEnd",
  sessionStartHandler = "Main.onSessionEnd",
  sessionEndHandler = "Main.onSessionStart",
  missingTemplateHandler = "Main.onMissingTemplate"
}
```

Note: We recommend create a single handler called *Main.cfc* that can act as your main implicit event handler. This makes declaring and maintenance for the implicit events much nicer, but yet again, this is just a suggestion.

Extension Points Settings

Key	Type	Required	Default	Description
UDFLibraryFile	path	false	---	The absolute or relative path to a UDF helper file. The framework will load all the methods found in this helper file globally. Meaning it will be injected in ALL handlers, layouts and views.
coldboxExtensionsLocation	dot notation path	false	---	The dot notation path of where the framework extensions root is located. Ex: 'shared.coldbox.extensions'
modulesExternalLocation	include path	false	array	An array of locations of where ColdBox should look for modules to load into your application. The path can be a cf mapping or <i>cfinclude</i> compatible location. Modules are searched and loaded in the order of the declared locations. The first location ColdBox will search for modules is the conventions folder <i>modules</i>
pluginsExternalLocation	dot notation path	false	---	The dot notation path of where to look for custom plugins for your application. This path is a secondary location, meaning that the conventions folder takes precedence.
viewsExternalLocation	include path	false	---	The CF include path of where to look for secondary views for your application. Secondary views look just like normal views except the framework looks in the conventions folder first and if not found then searches this location.
layoutsExternalLocation	include path	false	---	The CF include path of where to look for secondary layouts for your application. Secondary layouts look just like normal layouts except the framework looks in the conventions folder first and if not found then searches this location.
handlersExternalLocation	dot notation path	false	---	The CF dot notation path of where to look for secondary events for your application. Secondary events look just like normal events except the framework looks in the conventions folder first and if not found then searches this location.
requestContextDecorator	CFC path	false	---	The dot notation CFC path of the request context decorator to use for this application. See RequestContextDecorator

```
coldbox={
  //Extension Points
  UDFLibraryFile = "includes/helpers/ApplicationHelper.cfm"
  coldboxExtensionsLocation = "sharedstuff.coldbox.extensions"
  modulesExternalLocation = "{/customer1/appmodules; /shared/app/modules}",
  pluginsExternalLocation = "sharedstuff.coldbox.plugins",
  viewsExternalLocation = "/sharedstuff/appViews",
  layoutsExternalLocation = "/sharedstuff/appLayouts",
  handlersExternalLocation = "sharedstuff.appHandlers",
  requestContextDecorator = "#appMapping#.model.coldbox.MyRequestContext"
}
```

Please note that some extension points require dot notation and some required a '/' notation which mimics what *cfinclude* uses.

Exception Handling

Key	Type	Required	Default	Description
exceptionHandler	event syntax	false	---	The event handler to call whenever ANY non-catched exception occurs anywhere in the request lifecycle execution. Before this event is fired, the framework will log the error via the <i>Logger</i> plugin object and set an <i>exceptionBean</i> variable in the request collection that models the thrown exception. Most likely you will use this setting along side the <i>CustomErrorTemplate</i> setting. See Recipes:Mv First Custom Exception Handler

onInvalidEvent	event syntax	false	---	This is the event handler that will fire masking a non-existent event that gets requested. This is a great place to place 302 or 404 redirects whenever non-existent events are being requested. See Recipes:Creating a 404 template via onInvalidEvent
customErrorTemplate	relative path	false	---	The relative path from the application's root level of where the custom error template exists. This template receives a key in the request collection called exceptionBean that contains the exception. Look at <i>coldbox.system.beans.ExceptionBean</i> for complete methods and API signature. See Recipes:Custom Exception Template

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

Application Aspects

Key	Type	Required	Default	Description
handlerCaching	boolean	false	true	This is useful to be set to false in development and true in production. This tells the framework to cache your event handler objects. The default persistence timeouts are set in your cache object settings or you can override it at the CFC level by adding caching metadata to the event handler. See EventHandlers
eventCaching	boolean	false	true	This directive tells ColdBox that when events are executed they will be inspected for caching metadata. This does not mean that ALL events WILL be cached if this setting is turned on. It just activates the inspection mechanisms for whenever you annotate events for caching. See EventHandlers
proxyReturnCollection	boolean	false	false	This is a boolean setting used when calling the ColdBox proxy's <i>process()</i> method. If this setting is set to true, the proxy will return back to the remote call the entire request collection structure ALWAYS! If set to false, it will return, whatever the event handler returned back. Our best practice is to always have this false and return appropriate data back.
ImplicitViews	boolean	false	true	Allows you to use implicit views in your application and view dispatching
caseSensitiveImplicitViews	boolean	false	false	By default implicit views are all in lower case, so you would turn this setting on to allow case sensitivity.

```
coldbox={
  // Application Aspects
  handlerCaching =false,
  eventCaching =true,
  proxyReturnCollection =false,
  implicitViews =true,
  caseSensitiveImplicitViews =true
}
```

conventions

This element defines custom conventions for your application. By default, the framework has a default set of conventions that you need to adhere too. However, if you would like to implement your own conventions for a specific application, you can use this setting, otherwise do not declare it:

Key	Type	Required	Default	Description
handlersLocation	relative path	false	handlers	This element is to declare where your handlers are stored within the application root. In my sample its the directory controllers
pluginsLocation	relative path	false	plugins	This element is to declare where your custom plugins are stored within the application root. In my sample it's the directory plugins
viewsLocation	relative path	false	views	This element is to declare where your views are stored within the application root. In my sample it's the directory views
layoutsLocation	relative path	false	layouts	This element is to declare where your layouts are stored within the application root. In my sample it's the directory views
modelsLocation	relative path	false	models	This element is to declare where your model objects are stored within the application root. In my sample it's the directory model
modulesLocation	relative path	false	modules	This element is to declare where your modules are stored within the application root. In my sample it's the directory modules
eventAction	string	false	index	This element defines what is the default event action an event handler will use if an incoming event has no defined action. If there is no action then the framework will look for this action in the handler and execute it.

```
//Conventions
conventions = {
  handlersLocation = "controllers",
  pluginsLocation  = "plugins",
  viewsLocation    = "views",
  layoutsLocation  = "views",
  modelsLocation   = "model",
  modulesLocation  = "modules",
  eventAction      = "index"
};
```

Note: The default conventions are inherited from the platform's configuration file found at `/coldbox/system/config/settings.xml`. What is shown is the shipped convention names.

datasources

This element is used to define metadata about datasources used in your application. This in no way means that whatever datasource you define here, ColdBox will register in the ColdFusion engine. No, this setting is for documentation and easy usage of datasources in your application. ColdBox even enables you to create datasource bean objects that represent these defined datasources and use them pretty much anywhere.

To get started you need to create internal structures that represent a datasource and the key that you use in the *datasources* structure is the alias of the datasource that you can use later on to autowire datasources or request datasource beans.

```
//Datasources
datasources = {
  alias1 = { name=" ", dbType=" ", username=" ", password=" " },
  alias2 = { name=" ", dbType=" ", username=" ", password=" " }
};
```

As you can see from the previous example, each internal key of the main *datasources* structure has another structure defined with the following elements:

Key	Type	Required	Default	Description
name	string	true	---	The name of the datasource registered in the CFML engine
dbType	string	false		A nice metadata item that denotes the database type
username	string	false		The metadata username of the datasource
password	string	false		The metadata password of the datasource

```
//Datasources
datasources = {
  blog = { name="blog2009", dbType="MSSQL", username=" ", password=" " },
  customers = { name="customers2010", dbType="MySQL", username="cn44", password="root" }
};
```

debugger

This element defines how the ColdBox debugger behaves and looks. If you do not need to alter this behavior then do not declare this structure.

Key	Type	Required	Default	Description
enableDumpVar	boolean	false	false	Enables the dumping of variables via the URL into the debugger. See URLActions
persistentRequestProfilers	boolean	false	true	Enables the tracking of request profilers in your application when in debug mode
maxPersistentRequestProfilers	numeric	false	10	The maximum number of profilers to keep in memory when profiling. This is a stack of profilers that renews itself on each request.
maxRCPanelQueryRows	numeric	false	50	The maximum number of rows to dump in the request collection panel
showTracerPanel	boolean	false	true	Whether the tracer panel is enabled in the debugger rendering
expandedTracerPanel	boolean	false	true	Whether the tracer panel will be expanded by default or collapsed
showInfoPanel	boolean	false	true	Whether the info panel is enabled in the debugger rendering
expandedInfoPanel	boolean	false	true	Whether the info panel will be expanded by default or collapsed
showCachePanel	boolean	false	true	Whether the cache panel is enabled in the debugger rendering
expandedCachePanel	boolean	false	true	Whether the cache panel will be expanded by default or collapsed
showRCPanel	boolean	false	true	Whether the request collection panel is enabled in the debugger rendering
showRCSnapshots	boolean	false	false	Turns on RC and PRC request profiling
expandedRCPanel	boolean	false	true	Whether the request collection panel will be expanded by default or collapsed
showModulesPanel	boolean	false	true	Whether the modules panel is enabled in the debugger rendering
expandedModulesPanel	boolean	false	true	Whether the modules panel will be expanded by default or collapsed

```
// ColdBox Debugger Settings
debugger = {
  enableDumpVar =false,
  persistentRequestProfilers =true,
  maxPersistentRequestProfilers = 20,
  maxRCPanelQueryRows = 10,
  //Panels
  showTracerPanel =true,
  expandedTracerPanel =true,
  showInfoPanel =true,
  expandedInfoPanel =true,
  showCachePanel =true,
  expandedCachePanel =true,
  showRCPanel =true,
  showRCSnapshots =false,
  expandedRCPanel =true,
  showModulesPanel =true,
  expandedModulesPanel =false
};
```

environments

The configuration CFC has embedded environment control and detection built in and it is much more extensive than using the Environment Interceptor and the XML approach. In this structure you will setup the environments and their associated regular expressions for its `cgi.http_host` names to match. If the framework matches the regex with the associated `cgi.http_host` of the startup request then it will set a setting called *Environment* in your configuration settings and will then go ahead and look for that environment setting name in your CFC as a method. That's right, it will check if your CFC has a method with the same name as the environment and if it exists it will call it for you. Here is where you basically override, remove or add any settings according to your environment.

Important : The environment detection occurs AFTER the `configure()` method is called. Therefore, whatever settings or configurations you have on the `configure()` method will be stored first.

```
environments = {
  development = "^cf9.^railo."
};
```

In the above example, I declare a *development* key with a value list of a regular expressions. So if I am in a host that starts with *cf9* this will match and set the *environment* setting equal to *development* and then look for a *development* method in this CFC and execute it:

```
/**
 * Executed whenever the development environment is detected
 */
function development(){
  // Override coldbox directives
  coldbox.handlerCaching =false;
  coldbox.eventCaching =false;
  coldbox.debugPassword ="";
  coldbox.reinitPassword ="";

  // Add dev only interceptors
  arrayAppend(interceptors, {class=coldbox.system.interceptors.ColdboxSidebar} );
}
```

Isn't this cool? Just create a method with the same name of the environment and go for it, sure makes life simpler! But what if I have my own strategy on detecting my environment that does not involve the `cgi.http_host`?

Custom Environment Detection

You then will do your own custom environment detection. You will NOT fill out an environments structure but actually create a method with the following signature:

```
string public detectEnvironment(){
}
```

Create a public method called `detectEnvironment()` that returns a string. If the framework detects that you have created this method, it will call it for you. You will then do your own custom environment detection and return the **name** of the environment you are on as a string. The framework will then save that string as the current *environment* setting and look for a method with that same name and execute it if it exists. So there you go, if you don't like our basic approach, then override it and extend it.

flash

The **flash** structure is used to configure the application's [FlashRAM](#) capabilities:

Key	Type	Required	Default	Description
scope	string or instantiation path	false	<i>session</i>	Determines what scope to use for Flash RAM . The available aliases are: session, client, cluster, ColdboxCache or a custom instantiation path
properties	struct	false	{ }	Properties that can be used inside the constructor of any Flash RAM implementation
inflateToRC	boolean	false	true	Whatever variables you put into the Flash RAM , they will also be inflated or copied into the request collection for you automatically.
inflateToPRC	boolean	false	false	Whatever variables you put into the Flash RAM , they will also be inflated or copied into the private request collection for you automatically
autoPurge	boolean	false	true	This is what makes the Flash RAM work, it cleans itself for you. Be careful when setting this to false as it then becomes your job to do the cleaning

autoSave boolean false true The [Flash RAM](#) saves itself at the end of requests and on relocations via `setNextEvent()`. If you do not want auto-saving, then turn it off and make sure you save manually

```
// flash scope configuration
flash = {
scope = "session,client,cluster,ColdboxCache,or full path"
properties = {}, // constructor properties for the flash scope implementation
inflateToRC = true, // automatically inflate flash data into the RC scope
inflateToPRC = false, // automatically inflate flash data into the PRC scope
autoPurge = true, // automatically purge flash data for you
autoSave = true // automatically save flash scopes at end of a request and on relocations.
};
```

i18n

The i18n structure is used to activate the localization and internationalization features of ColdBox. All you have to do is fill out the following keys:

Key	Type	Required	Default	Description
defaultResourceBundle	relative path	false	---	The base path of where resource bundles will be stored and the initial name of the default resource bundle to load. Ex: <i>includes/i18n/main</i> means that the name of the resource bundles will start with <i>main_language_countrycode</i> .
defaultLocale	string	false	---	The default locale to give users when hitting your application. Please remember that this is using standard java locale syntax. Some examples are <i>es_SV</i> , <i>es_DO</i> . This value gets appended to the default resource bundle setting to define the default language property file to load: <i>includes/main_en_US.properties</i> . So make sure your files are in the following format: {name}_{java standard locale}.properties
localeStorage	string	false	session	This is where the framework will store the client's locale in order to track them and you have three possibilities: <ul style="list-style-type: none"> • session • client • cookie • request
unknownTranslation	numeric	false	---	A setting used by the resource bundle plugin whenever a translation cannot be found. Instead of returning the default bogus <code>_UNKNOWN_</code> , you can choose your own string to return.

```
//i18n & Localization
i18n = {
defaultResourceBundle = "includes/i18n/main";
defaultLocale = "en_US",
localeStorage = "session",
unknownTranslation = "***NOT FOUND**"
};
```

Good i18n Resources

- [ColdBox i18n Guide](#)
- Paul Hastings cfc: <http://www.sustainablegis.com/unicode/resourceBundle/rb.cfm>
- For ISO Language Code information look at: <http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt>
- For ISO Country Code information look at: http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html
- For locale information look at <http://java.sun.com/j2se/1.4.2/docs/guide/intl/locale.doc.html>
- A free open source java translation editor <http://attesoro.org/>

Sample resource bundle

```
help_button=Help
closeLink=<a href=' javascript:window.close()'>Close</a>
relocate_button=Relocate Now!
intro_message=Welcome to my website.
search_button=Search It!
search_test=<i>Search for postings now.</i>
```

interceptorSettings

This structures configures the interceptor service in your application. Below are the main configuration keys:

Key	Type	Required	Default	Description
throwOnInvalidStates	boolean	false	false	This tells the interceptor service to throw an exception if the state announced for interception is not valid or does not exist.
customInterceptionPoints	list	false		This key is a comma delimited list of custom interception points you will be registering for execution. This is the way to provide an observer-observable pattern to your applications. Again, please see the [Interceptors Interceptor's Guide] for more information. Just note that here is where you register the custom interception points separated by a comma if more than one. This is needed so when interceptors are registered for execution points, these points will also be searched and registered for.

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

interceptors

This is an array of [interceptor](#) definitions that you will use to register them in your application. The key about this array is that **ORDER** matters. The interceptors will fire in the order that you register them whenever their interception points are announce, so please watch out for this caveat. Each array element is a structure that describes what interceptor to register. Below are the keys needed to register an interceptor:

Key	Type	Required	Default	Description
name	string	false	listLast(class, ".")	The unique name of the registered interceptor. If none is set, the framework will use the name of the interceptor file without the <i>.cfc</i> . Ex: <i>class="coldbox.system.interceptors.Deploy</i> , the name will be <i>Deploy</i>
class	CFC path	true	---	The instantiation path for the interceptor to register.
properties	struct	false		A structure of name-value pairs that will be used to construct the interceptor with. You can then use all the interceptor's properties' functions to interact with them.

```
//Register interceptors as an array, we need order
interceptors = [
  //Autowire
  {class="coldbox.system.interceptors.Autowire",
  properties={useSetterInjection=false}
  },
  //SES
  {class="coldbox.system.interceptors.SES" name="MySES"}
];
```

Important: Order of declaration matters! Also, when declaring multiple instances of the same CFC (interceptor), make sure you use the **name** attribute in order to distinguish them. If not, only one will be registered (the last one declared).

ioc

This structure is used to declare the integration of third party dependency injection - inversion of control frameworks such as WireBox or ColdSpring or LightWire or your own. Below are the configuration keys for this structure:

Key	Type	Required	Default	Description
framework	string or instantiation path	true	---	The registered name for the ioc framework to use. Possible values are: <ul style="list-style-type: none"> • wirebox • coldspring • coldspring2 • lightwire Or a custom CFC instantiation path to your very own IOC adapter.
reload	boolean	false	false	This value is used for development purposes. If enabled, the framework will reload the entire ioc factory on each request in order to allow for continuous development and changes to be reflected immediately.
objectCaching	boolean	false	false	This setting tells the IoC plugin to actually cache the objects created by the IoC framework in the ColdBox cache. By using the IoC plugin's <i>getBean()</i> method, you will be retrieving from the object cache and not creating an object again. Not only do you need to set this setting to true for the objects to get cached, but you need to add coldbox cache metadata to their <i>cfcomponent</i> tags in order for them to be cached. By default, the ioc plugin will NOT cache objects, unless their metadata specifies it. See Plugins:ColdspringIntegration , Plugins:LightwireIntegration
definitionFile	path	true	---	This is the location of your IoC configuration file. You can use either a relative or absolute path. If you use Lightwire then this setting is the instantiation path to your config bean or a valid coldspring configuration file that the framework will try to adapt for you. See Plugins:LightwireIntegration
parentFactory	struct	false	{ framework="", definitionFile="" }	Allows you to define a parent IoC factory.

```
//IOC Integration
ioc = {
  framework = "coldspring",
  reload =true,
  objectCaching =false,
  definitionFile = "config/coldspring.xml.cfm"
  parentFactory = {
    framework = "coldspring", definitionFile = "config/parent.xml.cfm"
  }
};
```

Please note that you can build your own IOC adapters for your own custom object factories or other IOC engines. You do this by implementing the following class: 'coldbox.system.ioc.AbstractIOCApapter'

layoutSettings

This structure is used in order to control the directives of the ColdBox layout manager. Below are the keys you can configure:

Key	Type	Required	Default	Description
defaultLayout	file path	false	---	The name of the file that will act as the default layout for all renderings the framework does that does not explicitly provide a layout for rendering. This file needs to exist in either the conventions <i>layouts</i> folder or the layouts external location.
defaultView	file path	false	---	The name of the file that will be rendered whenever a handler does not set a view for rendering. This file needs to exist in either the conventions <i>views</i> folder or the views external location.

```
//Layout Settings
layoutSettings = {
  defaultLayout = "Main.cfm",
  defaultView   = "youForgot.cfm"
};
```

You will get more in depth information about the layout manager in the [Layouts & Views](#) Guide.

layouts

The layouts array element is used to define implicit associations between layouts and views/folders, this does not mean that you need to register ALL your layouts. This is a convenience for pairing them, we are in a conventions framework remember. Before any renderings occur or lookups, the framework will check this array of associations to try and match in what layout a view should be rendered in. It is also used to create aliases for layouts so you can use aliases in your code instead of the real file name and locations. Each array element is a structure that contains the following keys:

Key	Type	Required	Default	Description
name	string	true	---	The name or alias of the layout you are about to register or associate.
file	file	true	---	The name of the file that represents this layout name. This file needs to exist in either the conventions <i>layouts</i> folder or the layouts external location.
views	view name list	false	---	The name of the file that will be rendered whenever a handler does not set a view for rendering. This file needs to exist in either the conventions <i>views</i> folder or the views external location. Do not append a <i>.cfm</i> extension, just the full name of the view.
folders	regex list	false	---	A list of regular expression folder names that you want to associate with the layout. This means that if you are rendering a view and that view exists in a folder

Important: All view declarations must NOT include the *.cfm* extension. ColdBox automatically appends the *.cfm* extension.

```
//Register Layouts
layouts = [
  { name="tester",file="Layout.tester.cfm",views="vwLogin,test",folders="tags,pdf/single" },
  { name="login",file="Login.cfm",folders="^admin/security" }
];
```

logBox

The logBox structure is based on the LogBox declaration DSL, see [LogBox](#). Below are the main keys you can fill out:

Key	Type	Required	Default	Description
configFile	file path	false	config/LogBox.cfc	An absolute or relative path to the LogBox configuration CFC or XML file to use instead of declaring the rest of the keys in this structure. So if you do not define a logBox structure, the framework will look for the default value: <i>config/LogBox.cfc</i> and it will load it if found. If not found, it will use the default logBox configuration found in <i>/coldbox/system/web/config/LogBox.cfc</i>
appenders	struct	true	empty	A structure where you will define appenders to use in your application's LogBox instance. Each appender structure has different keys you can define within it: class, layout, properties, levelMin, and levelMax.
root	struct	true	empty	The definition of the root logger object. The keys to configure will be: levelMin, levelMax and appenders.
categories	struct	false	empty	A structure where you can define granular categories with their optional logging levels and appenders.
DEBUG	array	false	empty	An array that will hold all the category names to place under the DEBUG logging level
INFO	array	false	empty	An array that will hold all the category names to place under the INFO logging level
WARN	array	false	empty	An array that will hold all the category names to place under the WARN logging level
ERROR	array	false	empty	An array that will hold all the category names to place under the ERROR logging level
FATAL	array	false	empty	An array that will hold all the category names to place under the FATAL logging level
OFF	array	false	empty	An array that will hold all the category names to not log at all

Important: If you do not define a `logBox DSL` structure, the framework will look for the default configuration file `config/LogBox.cfc`. If it does not find it, then it will use the framework's default logging settings. However, if you **DO** define the structure DSL, please make sure you define the required keys.

Appender Definition

So to define an appender you must define a key value which is the internal name of the appender with the following keys:

Key	Type	Required	Default	Description
class	CFC path	true	---	The CFC path of the appender.
layout	CFC path	false	---	The CFC that this appender will use to format its entry strings or logging mechanisms.
properties	struct	false	empty struct	A structure of name-value pairs with appender properties you can use within the appenders or to configure an appender with.
levelMin	numeric or string	false	0 = FATAL	The minimal logging level this appender will log to.
levelMax	numeric or string	false	4 = DEBUG	The maximum logging level this appender will log to.

```
//LogBox DSL
logBox = {
  // Define Appenders
  appenders = {
    coldboxTracer = {
      class="coldbox.system.logging.appenders.ColdboxTracerAppender"
      layout="coldbox.testing.cases.logging.MockLayout"
      properties = {
        name = "awesome"
      }
    },
    rollingFile = {
      class="coldbox.system.logging.appenders.AsyncRollingFileAppender"
      levelMax="WARN",
      levelMin="FATAL",
      properties={
        filePath="#appMapping#/logs",
        autoExpand=true",
        fileMaxSize=3000",
        fileMaxArchives=5"
      }
    }
  }
};
```

Root Logger

To define the root logger you can use the following keys:

Key	Type	Required	Default	Description
levelMin	numeric or string	false	0 = FATAL	The minimal logging level this appender will log to.
levelMax	numeric or string	false	4 = DEBUG	The maximum logging level this appender will log to.
appenders	list	true	---	A list of appender names to register with the root logger or use * to register all appenders.

```
//LogBox DSL
logBox = {
  root = { levelMin="FATAL", levelMax="INFO", appenders="*" }
};
```

Categories

Categories are defined by creating a set of structures within it with the key name being the name of the category you would like to declare and the following internal structure keys:

Key	Type	Required	Default	Description
levelMin	numeric or string	false	0 = FATAL	The minimal logging level this appender will log to.
levelMax	numeric or string	false	4 = DEBUG	The maximum logging level this appender will log to.
appenders	list	false	*	A list of appender names to register with this category, use * to register all appenders.

As you might notice the name of the keys on all the structures match 100% to the programmatic methods you can also use to configure logBox. So when in doubt, refer back to the argument names.

```
logBox = {
  // Categories
  categories = {
    "coldbox.system" = { levelMax=INFO },
    "coldbox.system.interceptors" = { levelMin=0, levelMax=DEBUG, appenders="*" },
    "hello.model" = { levelMax=4, appenders="*" }
  }
};
```

Implicit Categories

You can then define the implicit logging categories by using the severity name and assigning an array of categories to them:

```
logBox = {
  debug = [ "coldbox.system", "model.system" ],
  info = [ "hello.model", "yes.wow.wow" ],
  warn = [ "hello.model", "yes.wow.wow" ],
  error = [ "hello.model", "yes.wow1.wow" ],
  fatal = [ "hello.model", "yes.wow.wow" ],
  OFF = [ "hello.model", "yes.wow.wo2w" ]
};
```

mailSettings

Used to register a mail protocol to use when sending mails via our [Plugins:MailService](#). It also allows you to setup any key that corresponds to the `cfmail` tag to use as defaults.

Key	Type	Required	Default	Description
protocol	structure	false	{class="coldbox.system.core.mail.protocols.CFMailProtocol",properties={}}	Allows you to define the path to the mail protocol to use when sending email. Great for switching in tiers or just using an abstraction to send emails out. Valid keys are <i>class</i> and <i>properties</i>
server	numeric or string	false	---	The mail server address
username	numeric or string	false	---	The username
password	list	false	---	The password
port	numeric	false	25	The port of the mail server

```
//Mailsettings
mailSettings = {
  server = "mail.mydomain.com";
  username = "lui@mydomain.com";
  password = "YouRock";
  port = 25
};
```

Important: You can have default values for ANY of the `cfmail` tag attributes via the configuration element keys and they will be added to the mail payload object for you. This is a great way to define default mailing attributes or custom ones.

Available Protocols

The following are the core mail protocols, or you can build your own by extending our `coldbox.system.core.mail.AbstractProtocol` class.

- **CFMailProtocol**: Uses `cfmail`
- **FileProtocol**: Sends mail to files
- **PostmarkProtocol**: Sends mail via [PostMark](#)

FileProtocol Properties

Key	Type	Required	Default	Description
filePath	path	true	---	The relative or absolute path where the files will be stored
autoExpand	boolean	false	true	Use <code>expandPath()</code> on the file path property.

```
mailSettings = {
  protocol = {
    class = "coldbox.system.core.mail.protocols.FileProtocol",
    properties = {
      filePath = "mailspool"
    }
  }
};
```

PostmarkProtocol Properties

Key	Type	Required	Default	Description
APIKey	string	true	---	The postmark API key needed to send email out

```
mailSettings = {
protocol = {
  class = "coldbox.system.core.mail.protocols.PostmarkProtocol",
  properties = {
    APIKey = "YouWishYouHadIt"
  }
}
};
```

modules

The modules structure is used to configure the behavior of the ColdBox [Modules](#).

Let's explore these settings:

Key	Type	Required	Default	Description
AutoReload	boolean	false	false	Will auto reload the modules in each request. Great for development
Include	array	false	empty array	An array or list of module names that should be loaded when the application starts up. If this setting is empty, it means that the framework will load ALL modules
Exclude	array	false	empty array	An array or list of module names that should be EXCLUDED when the application starts up. If this setting is empty, it means that the framework will load ALL modules

```
modules = {
  autoReload = true,
  include = [],
  exclude = ["paidModule1", "paidModule2"]
};
```

orm

The **orm** structure is used to configure the application for entity injection and ORM services integration:

Key	Type	Required	Default	Description
injection	struct	false	{}	Used to configure the application for entity injection. Valid keys are: <ul style="list-style-type: none"> ● enabled: Turn on entity injection ● include: A list of entity names to include ONLY in the injections ● exclude: A list of entity names to exclude ONLY

```
orm = {
  injection = {
    // enable entity injection
    enabled = true,
    // a list of entity names to include in the injections
    include = "",
    // a list of entity names to exclude from injection
    exclude = ""
  }
};
```

Important: Please note that you must enable the [ORM Event Handler](#) in order for the entity injection to work.

settings

This element is used by the developer to set any values he/she would like to use in the application. This can be configuration settings, etc. Remember that you can use any of the already created variables in this CFC or the injected variables to concatenate or append your own settings. Also, since you are in a programmatic CFC you can pretty much do whatever you like here.

```
// Custom Settings
settings = {
  useSkins = true,
  myCoolArray = [1,2,3,4],
  skinsPath = "views/skins",
  myUtil = createObject("component", "#appmapping#.model.util.MyUtility")
};
```

We will cover all the methods to interact with settings later on, but the simplest way to get settings is well, to use:

```
getSetting(name, [FWSettingfalse])
```

validation

Here is where you can tune the ColdBox [Validation](#) engine, **ValidBox**, or connect to a third party validation engine. You can also store validation rules by a cool alias name, so you can refer to them later on in code; or as we call them, shared constraints.

Key	Type	Required	Default	Description
-----	------	----------	---------	-------------

manager	instantiation path or WireBox ID	false	coldbox.system.validation.ValidationManager	You can override the default ColdBox validation manager with your own implementation. Just use an instantiation path or a valid WireBox object id.
sharedConstraints	struct	false	{ }	This structure will hold all of your shared constraints for forms or/and objects.

```
validation = {
  sharedConstraints = {
    user = {
      fName = {required:true},
      lName = {required:true},
      age = {required:true, max=18 }
      metadata = {required:false, type="json"}
    },
    loginForm = {
      username = {required:true}, password = {required:true}
    },
    changePasswordForm = {
      password = {required:true, min=6}, password2 = {required:true, sameAs="password", min=6}
    }
  }
}
```

ColdBox will also create a mapping to your validation manager with the id: **WireBoxValidationManager**, so you can easily retrieve it.

webservices

Since we are now in a service oriented web architecture, this was a great way to declare all the web services my application that will be used. ColdBox reads these elements and places them in the configuration structure. You can then use the webservice plugin to get the webservice's URL, get a webservice object already instantiated or refresh the web services's stubs or use WireBox to autowire webservice objects according to their registered names. So the key is the alias you want to use in your application and the value is the WSDL URL.

```
//webservices
webservice = {
  testWS = "http://www.test.com/test.cfc?wsdl"
  AnotherTestWS = "http://www.coldbox.org/distribution/updatews.cfc?wsdl"
};
```

wirebox

This configuration structure is used to configure the [WireBox](#) dependency injection framework embedded in ColdBox. For version 3.0.0 you **must** enable it in order to be used as the primary engine. If not, we will fall back to a compatibility mode engine until ColdBox 3.1 is released. Why do we do this? Well, ColdBox 3 is still valid for ColdFusion 7 and WireBox is for ColdFusion 8 and above. Therefore, you must explicitly enable it for 3.0.0. However, please note that the **enabled** key will be dropped by the next release:

Key	Type	Required	Default	Description
binder	instantiation path	false	<i>config.WireBox</i>	The location of the WireBox configuration binder to use for the application. If empty, we will use the binder in the <i>config</i> folder called by conventions: <i>WireBox.cfc</i>
singletonReload	boolean	false	false	A great flag for development. If enabled, on every request WireBox will flush its singleton objects so you can develop without any headaches of reloading.

```
// wirebox integration
wirebox = {
  singletonReload = true,
  binder = 'config.WireBox'
};
```

Interacting With The Loaded Settings

Now that you know how to configure your application, you can now check out how to [interact with those loaded settings](#).

If you would like to reload your settings, you will need to reinitialize the framework by using the **fwreinit=1 URL action** (See [URL Actions Guide](#)). This is a very important URL action, as most of the time your settings are cached and you need to reload them.