

[← Back to Dashboard](#)

Layouts & Views Guide

Covers up to version 3.5.0

Introduction

ColdBox provides you with a very simple but flexible and powerful layout manager and content renderer. You no longer need to create module tags or convoluted broken up HTML anymore. You can concentrate on the big picture and create as many layouts as your application needs. Then you can programmatically change rendering schemas (or skinning) and also create composite views. In this guide we will explore the different rendering mechanisms that ColdBox offers and also how to utilize them. As you know, [event handlers](#) are our controller layer in ColdBox and we will explore how these objects can interact with the user in order to render content, whether HTML, JSON, XML or any type of rendering data.

Response Types

The event handlers will be the objects in question that will be responding to user requests and they have a few choices when it comes down to rendering content back:

- Set a view to be rendered back to the user (with or without a layout)
- Render data in multiple view formats: JSON, XML, WDDX, HTML, TEXT or CUSTOM
- Render nothing
- Do an HTTP redirect to another event (part of our [event handler](#) guide via `setNextEvent()`)

Conventions

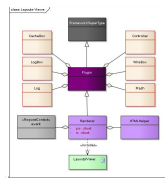
Let's do a recap of our conventions for layouts and view locations:

- + application
- + layouts
- + views

All your layouts will go in the **layouts** folder and all your views will go in the **views** folder.

Also remember that ColdBox allows you to change your conventions via the [application configuration CFC](#).

ColdBox Renderer



It is imperative to know **who** does the rendering in ColdBox and that is the **Renderer** core plugin that you can see from our diagram above. As you can tell from the diagram, it includes your layouts and/or views inside itself in order to render out content. So by this association and inheritance **all** layouts and views have some variables and methods at their disposal since they get absorbed into the plugin. You can visit the [API docs](#) to learn about all the Renderer Plugin methods. All of the following property members exist in all layouts and views rendered by the *Renderer*:

Property	Description
event	A reference to the Request Context
rc	A reference to the request collection inside of the request context (For convenience)
prc	A reference to the private request collection inside of the request context (For convenience)
html	A reference to the HTML Helper plugin (<i>coldbox.system.plugins.HTMLHelper</i>) that can help you build interactive HTML
cacheBox	A reference to the CacheBox framework factory (<i>coldbox.system.cache.CacheFactory</i>)
controller	A reference to the application's ColdBox Controller (<i>coldbox.system.web.Controller</i>)
flash	A reference to the current configured Flash Object Implementation that inherits from the AbstractFlashScope <i>AbstractFlashScope</i> (derived <i>coldbox.system.web.flash.AbstractFlashScope</i>)
logBox	The reference to the LogBox library (<i>coldbox.system.logging.LogBox</i>)
log	A pre-configured LogBox Logger object for this specific class object (<i>coldbox.system.logging.Logger</i>)
wirebox	A reference to the WireBox object factory (<i>coldbox.system.ioc.Injector</i>)

As you can see, all views and layouts have direct reference to the request collections so it makes it incredibly easy to get and put data into it. Also, remember that the *Renderer* inherits from the base ColdBox *Plugin* class which in turn inherits from the *FrameworkSuperType* class. So all methods are at your disposal if needed.

Do not put any type of business logic in layouts and views, it does not belong there!

Rendering Views

Views are HTML/XML content that can be rendered inside of a layout or by themselves. They can be either rendered on demand or by being set by an event handler. Views can also produce any type of content apart from HTML like JSON/XML/WDDX via our view renderer that we will discover also. So get ready for some rendering goodness!

Setting Views For Rendering

Usually, event handlers are the objects in charge of setting views for rendering. However, ANY object that has access to the request context object can do this also. This is done by using the `setView()` method in the request context object. Let's see the signature of this magnificent method:

```
setView([string view], [boolean noLayout='false'], [boolean cache='false'], [string cacheTimeout=''], [string cacheLastAccessTimeout=''], [string cacheSuffix=''], [string layout], [string module])
```

Setting a view does not mean that it gets rendered immediately. It means that it is deposited in the request context. The framework will later on in the execution process pick those variables up and do the actual rendering. To do immediate rendering you will use the inline rendering methods describe later on.

Handler Code:

```
component name="general" {
    function index(event, rc, prc) {
        // call some model for data and put into the request collection
        prc.myQuery = getModuleService().getData();
        // set the view for rendering
        event.setView("general/index")
    }
}
```

View Location (view="general/index"):

```
/application
  /views
    /general
      +index.cfm
```

That's it, we use the `setView()` method to set the view `general/index.cfm` to be rendered. Now the cool thing about this, is that we can override the view to be rendered anytime during the flow of the request. So the last event to execute the `setView()` method is the one that counts. Also notice a few things:

- No `.cfm` extension is needed.
- You can traverse directories by using `/` like normal `cinclude` notation.
- The view can exist in the conventions directory `views` or in your [configured external locations](#).
- You did not specify a layout for the view, so the application's **default layout** will be used, we will cover this later on.

As best practice view locations should simulate handler location + action name. So if the event is `general.index`, there should be a `general` folder in the root views folder with a view called `index.cfm`.

Let's look at the view code:

```
<<output>
<h3> Cool Data! </h3>
#html.table(data=prc.myQuery)#
</output>
```

I am using our cool [HTML Helper](#) plugin that is smart enough to render tables, data, HTML 5 elements etc and even bind to ColdFusion ORM entities. All layouts/views have access to the HTML helper.

Implicit Views

You can also omit the explicit `event.setView()` if you want. ColdBox will then look for the view according to the executing event's syntax. So if the incoming event is called `general.index` and no view is explicitly defined in your handler, ColdBox will look for a view in the `general` folder called `index.cfm`. That is why we recommend trying to match event resolution to view resolution even if you use or not implicit views. This feature is more for conventions purists than anything else. However, we do recommend as best practice to use explicitly declare the view to be rendered when working with team environments. Below is the change in code to use implicit views, basically remove the `setView()` method call.

```
component name="general" {
    function index(event, rc, prc) {
        // call some model for data and put into the request collection
        prc.myQuery = getModuleService().getData();
    }
}
```

Contents

- [Layouts & Views Guide](#)
- [Introduction](#)
- [Response Types](#)
- [Conventions](#)
- [ColdBox Renderer](#)
- [Rendering Views](#)
- [Setting Views For Rendering](#)
- [Implicit Views](#)
- [Event Sensitivity](#)
- [Views With No Layout](#)
- [Views With Specific Layout](#)
- [Views From A Module](#)
- [View Caching](#)
- [Dynamic Views](#)
- [Handler Return Rendering](#)
- [Render Nothing](#)
- [Inline Rendering](#)
- [RenderView\(\)](#)
- [RenderExternalView\(\)](#)
- [RenderLayouts\(\)](#)
- [Control Variable Views](#)
- [Rendering External Views](#)
- [Rendering With Local Variables](#)
- [Rendering Collections](#)
- [View Helpers](#)
- [View Events](#)
- [Sample Interceptor](#)
- [Layouts](#)
- [Basic Layout](#)
- [Nested Layouts](#)
- [Default Layout](#)
- [Default View](#)
- [Layout Helpers](#)
- [Overriding Layouts](#)
- [Implicit Layout/View Declarations](#)
- [Layouts From A Module](#)
- [Layout Events](#)
- [Rendering Data](#)
- [Global Parameters](#)
- [JSON Parameters](#)
- [XML Parameters](#)
- [PDF Parameters](#)
- [Custom Data Conversion](#)
- [Helper Libraries](#)
- [Viewlets \(Portable Events\)](#)
- [Empty Viewlets](#)
- [Tips And Tricks](#)
- [I don't want to render anything](#)
- [I don't want a coldbox debugging panel attached to this view](#)
- [Is this a ColdBox proxy request](#)
- [Am I in an ses application](#)
- [Am I in an sub-app](#)
- [Render to event ses viewlet](#)
- [What is the current layout?](#)
- [What is the current view?](#)

Important: If using implicit views, please note that the name of the view will ALWAYS be in lower case. So please be aware of this limitation. I would suggest creating [SES URL Mappings](#) with explicit event declarations so case and location can be controlled. When using implicit views you will also loose fine rendering control.

Case Sensitivity

The ColdBox rendering engine can also be tweaked to use case-insensitive or sensitive implicit views by creating a setting called: **caseSensitiveImplicitViews** in your configuration file. The default is to turn all implicit views to **lower case**, so the value is always false.

```
settings = {
  caseSensitiveImplicitViews: true
};
```

Views With No Layout

So what happens if I DO NOT want the view to be rendered within a layout? Am I doomed? Of course not, just use the same method with some extra parameters or *event.noLayout()*:

```
component name="general" {
  function index(event, rc, prc) {
    // call some model for data and put into the request collection
    prc.myQuery = getModuleService().getData();
    // set the view for rendering
    event.setView(view="general/index@noLayout=true");
  }

  function index(event, rc, prc) {
    // call some model for data and put into the request collection
    prc.myQuery = getModuleService().getData();
    // set the view for rendering
    event.setView(view="general/index@notLayout()");
  }
}
```

That's it folks! You use the **noLayout=true** argument or the **noLayout()** method.

Views With Specific Layout

If you need the view to be rendered in a specific layout, then use the **layout** argument:

```
component name="general" {
  function index(event, rc, prc) {
    // call some model for data and put into the request collection
    prc.myQuery = getModuleService().getData();
    // set the view for rendering
    event.setView(view="general/index@layout=ajax");
  }
}
```

The view will now be rendered into the **Ajax** layout instead of the default one.

Views From A Module

If you need the set view to be rendered from a specific [ColdBox Module](#) then use the **module** argument alongside any other argument combination:

```
component name="general" {
  function index(event, rc, prc) {
    // call some model for data and put into the request collection
    prc.myQuery = getModuleService().getData();
    // set the view for rendering
    event.setView(view="general/index@module=shared-view@pr");
  }
}
```

By using the **module** argument, you are explicitly telling the ColdBox renderer to look for the view inside of the passed module name.

View Caching

You can also pass in the caching arguments below and your view will be rendered once and then cached for further renderings. Every ColdBox application has two active cache regions by default: **default** and **template**. All view and event caching renderings go into the **template** cache.

Argument	Type	Required	Default	Description
cache	boolean	false	false	Cache the view to be rendered
cacheTimeout	numeric	false	(provider default)	The timeout in minutes or whatever the cache provider defines
cacheLastAccessTimeout	numeric	false	(provider default)	The idle timeout in minutes or whatever the cache provider defines
cacheSuffix	string	false	---	Adds a suffix key to the cached key. Used for providing uniqueness to the cacheable entry

```
component name="general" {
  function index(event, rc, prc) {
    // call some model for data and put into the request collection
    prc.myQuery = getModuleService().getData();
    // view with caching parameters
    event.setView(view="general/index@cache=true@cacheTimeout=60@cacheLastAccessTimeout=15@cacheSuffix=getfwLocale());
  }
}
```

Purging Views

So now that our views are cached, how do I purge them programmatically? Well, you need to talk to the **template** cache provider.

```
// get a reference to the template cache
cache = cachebox.getCache@platform;
// or
cache = getColdBoxOC@platform;
```

Then we can perform several operations on views:

- **clearView**(string viewSnippet): Used to clear a view from the cache by using a snippet matched according to name + cache suffix.
- **clearMultiView**(any viewSnippets): Clear using a list or array of view snippets.
- **clearAllViews**(boolean async=true): Can clear ALL cached views in one shot and can be run asynchronously.

```
cachebox.getCache@platform.clearView@general/index;
cachebox.getCache@platform.clearAllViews@async=true;
cachebox.getCache@platform.clearMultiView@general/index@index:'home');
```

Handler Return Rendering

Handlers can also return content by simply returning a simple string from the action method. How that string is built is up to you, it can be a simple string, a call from a model CFC, or inline rendering that we will see below.

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

That's it, just return the simple content and it will sent to the output buffer. This technique is great for producing viewlets/widgets or self-sustainable events, where you can call and render events from within anywhere in ColdBox via the **runEvent()** method. Please refer to the **viewlets** section in this guide.

Render Nothing

You can also tell the renderer to not render back anything to the user by using the **event.noRender()** method.. Maybe you just took some input and need to gracefully shutdown the request.

```
component name="general" {
  function saveData(event, rc, prc) {
    // do your work here ....

    // set for no render
    event.noRender();
  }
}
```

Inline Rendering

We have now seen how to set views to be rendered from our handlers. However, we can use three cool methods to render views and layouts on-demand, much how *cfinclude* is used. These methods exist in the **Renderer** plugin and several facade methods exist in the super type so you can call it from any plugin, handler, interceptor, view or layout:

renderView()

Render inline views from the parent application or from specific modules according to arguments:

Argument	Type	Required	Default	Description
view	string	false	look in request collection	The name of the view to render
cache	boolean	false	false	Cache the view to be rendered

cacheTimeout	numeric	false	(provider default)	The timeout in minutes or whatever the cache provider defines
cacheLastAccessTimeout	numeric	false	(provider default)	The idle timeout in minutes or whatever the cache provider defines
cacheSuffix	string	false	---	Adds a suffix key to the cached key. Used for providing uniqueness to the cacheable entry
module	string	false	---	The name of the module to render the view from
args	struct	false	{}	Local arguments to pass into the view rendering
collection	any	false	---	A collection to use by this Renderer to render the view as many times as the items in the collection (Array or Query)
collectionAs	any	false	---	The name of the collection variable in the partial rendering. If not passed, we will use the name of the view by convention
prepostExempt	boolean	false	false	If true, pre/post view interceptors will not be fired. By default they do fire

renderExternalView()

Render inline views from anywhere in the server according to arguments:

Argument	Type	Required	Default	Description
view	string	true	---	The name of the view to render
cache	boolean	false	false	Cache the view to be rendered
cacheTimeout	numeric	false	(provider default)	The timeout in minutes or whatever the cache provider defines
cacheLastAccessTimeout	numeric	false	(provider default)	The idle timeout in minutes or whatever the cache provider defines
cacheSuffix	string	false	---	Adds a suffix key to the cached key. Used for providing uniqueness to the cacheable entry
module	string	false	---	The name of the module to render the view from
args	struct	false	{}	Local arguments to pass into the view rendering

renderLayout()

Render nested layouts or view/layout combinations:

Argument	Type	Required	Default	Description
layout	string	true	---	The name of the explicit layout to render
view	string	false	---	The name of the view to render this layout with
module	string	false	---	The name of the module to render the view from
args	struct	false	{}	Local arguments to pass into the view rendering

Some Examples

```
<<foutput>
// render inline
#renderView(tags/metadata)#
// render from a module
#renderView(view=security/user,module=security)#
// render a view from the handler action
function showData(event,rc,prc){
// data here
return renderView(view=general/showData*
}
// render an email body content in an email template layout
body = renderLayout(layout=sl/view+templates/email_generic'
</cfoutput>
```

Inline renderings are a great asset for reusing views and doing layout compositions

Let's explore all the things we can do with inline renderings.

Content Variable Views

A content variable is a variable that contains HTML/XML or any kind of visual content that can easily be rendered anywhere. You can easily do this by leveraging inline rendering and placing the content into a variable, usually in an event handler:

```
function home(event,rc,prc){
// render some content variables with funky arguments
prc.sideColumn = renderView(view=sideColumn,cache=true,cacheTimeout=10);
// set view
event.setView(general/home#
}
So how do I render it?
```

```
<div id=#content#
<div id=#leftColumn#
<cfoutput=#prc.sideColumn#>cfoutput>
</div>
<div id=#mainView#
<cfoutput=#renderView(#)>cfoutput>
</div>
</div>
```

Another example, is what if we do not know if the content variable will actually exist? How can we do this? Well, we use the event object for this and its magic *getValue()* method.

```
<div id=#content#
<div id=#leftColumn#
<cfoutput=#event.getValue(name='sideColumn',defaultValue='')>cfoutput#
</div>
<div id=#mainView#
<cfoutput=#renderView(#)>cfoutput>
</div>
</div>
```

So now, if no content variable exists, an empty string will be rendered.

Important: String manipulation in Java relies on immutable structures, so performance penalties might ensue. If you will be doing a lot of string manipulation, concatenation or rendering, try to leverage native java objects: *StringBuilder* or *StringBuffer*

Rendering External Views

So what if I want to render a view outside of my application without using the setting explained above? Well, you use the **renderExternalView()** method.

```
renderExternalView([string view, [boolean cache='false'], [string cacheTimeout=''], [string cacheLastAccessTimeout=''], [string cacheSuffix=''], [any args]
```

```
<<foutput>#renderExternalView(view=views/mapping/tags/footer)>cfoutput>
```

Rendering With Local Variables

Passing local variables to layouts and views: *renderView(args)*, *renderLayout(args)* now get the **args** argument which can be a structure of data that will be specifically passed to views/layouts for rendering ONLY there. This gives you great DRYness (yes that is a word) when building new and edit forms or views as you can pass distinct arguments to distinguish them and keep structure intact.

Universal Form

```
<h1 #args.type# User#1>
<form method="post" action=#args.action#>
..
</form>
```

New Form

```
#renderView(view=erms/universa#args={type=new;action=user.create})#
```

Edit Form

```
#renderView(view=erms/universa#args={type=edit;action=user.update})#
```

Rendering Collections

You have a few arguments in the **renderView()** method that deal with collection rendering:

- **collection**: A data collection that can be a query or an array of objects, structs or whatever
- **collectionAs**: The name of the variable in the **variables** scope that will hold the collection pivot.
- **collectionStartRow**: Defaults to 1 or your offset row for the collection rendering
- **collectionMaxRows**: Defaults to show all rows or you can cap the rendering display
- **collectionDelim**: An optional delimiter to use to separate the collection renderings. By default it is empty.

Once you call *renderView()* with a collection, the renderer will render the view once for each member in the collection. The views have access to the collection via *arguments.collection* or the member currently iterating. The name of the member being iterated as *it* by convention the same name as the view. So if we do this in any layout or simple view:

```
#renderView(view=#args.comment, collection=rc.comments)#
```

Then the *tags/comment* will be rendered as many times as the collection *rc.comments* has members on it and by convention the name of the variable is *comment* the same as the view name. If you don't like that, then use the **collectionAs** argument:

```
#renderView(view=#args/comment, collection=rc.comments, collectionAs=#args)#
```

So let's see the collection view now:

```
<h1 #title: #comment.getTitle()# (#_counter# of #j# items#>
<p>Author: #comment.getAuthor()#
#comment.getComment()#
```

```
</div>
```

You can see that I just call methods on the member as if I was looping (which we are for you). But you will also see two little variables here:

- **counter**: A variable created for you that tells you in which record we are currently looping on
- **items**: A variable created for you that tells you how many records exist in the collection

This will then render that specific dynamic HTML view as many times as their are records in the `rc.comments` array and concatenate them all for you. In my case, I separate each iteration with a simple `
` but you can get fancy and creative.

```
#renderView(viewName='news' collection=prc.news, collectionStartRow=11, collectionMaxRows=20)#
```

View Helpers

This is a nifty little feature that enables you to create nice helper templates on a per-view or per-folder basis. If the framework detects the helper, it will inject it into the rendering view so you can use methods, properties or whatever. All you need to do is follow a set of conventions. Let's say we have a view in the following location:

```
/views
  /general
    home.cfm
```

Then we can create the following templates

- `homeHelper.cfm`: Helper for the `home.cfm` view.
- `generalHelper.cfm`: Helper for any view in the general folder.

```
/views
  /general
    +home.cfm
    +homeHelper.cfm
    +generalHelper.cfm
```

homeHelper.cfm:

```
<script>
// @todo: get lib resource
function() {return getResource(argumentCollection=arguments); }
// cool formatted date function
function today() {return dateFormat(now,{full}); }
</script>
```

That's it. Just append **Helper** to the view or folder name and there you go, the framework will use it as a helper for that view specifically. What can you put in these helper templates:

- NO BUSINESS CODE
- UI logic functions or properties
- Helper functions or properties
- Dynamic JavaScript or CSS

External views can also use our helper conventions

View Events

All rendered views have associated events that are announced whenever the view is rendered. These are great ways for you to be able to intercept when views are rendered and transform them, append to them, or even remove content from them in a completely decoupled approach. The way to listen for events in ColdBox is to write **interceptors**, which are essential simple CFC's that by convention have a method that is the same name as the event they would like to listen to. Each event has the capability to receive a structure of information which you can alter, append or remove from. Once you write these **interceptors** you can either register them in your **Configuration File** or **programmatically**.

Event	Data	Description
	<ul style="list-style-type: none"> • view - The name of the view to render • cache - If the view will be cached • cacheTimeout - The cache timeout • cacheLastAccessTimeout - The idle timeout of the view • cacheSuffix - A suffix to append to the cachable key name • module - The module name of the view if any 	
preViewRender	<ul style="list-style-type: none"> • args - The arguments to pass into the view • collection - The collection this view will iterate on • collectionAs - The alias of the collection name • collectionStartRow - The start row of the collection iteration • collectionMaxRows - The max rows of the collection iteration • collectionDelim - The delimiter used for the collection iteration 	Executed before a view is about to be rendered
postViewRender	All of the data above plus: <ul style="list-style-type: none"> • renderedView - The view contents that was rendered 	Executed after a view was rendered

You can disable the view events on a per-rendering basis by passing the **prePostExempt** argument as true when calling **renderView()** methods.

Sample Interceptor

Here is a sample interceptor that trims any content before it is rendered:

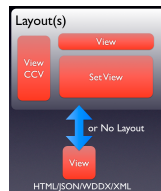
```
component {
  function postViewRender(event, interceptData) {
    interceptData.renderedView = trim( interceptData.renderedView );
  }
}
```

Of course, I am pretty sure you will be more creative than that!

Layouts

A layout is simply an HTML file that acts as your shell where views can be rendered in and exists in the **layouts** folder of your application. The layouts can be composed of multiple views and one **main** view. The **main** view is the view that gets set in the request collection during a request via `event.setView()` usually in your handlers or interceptors. You can have as many layouts as you need in your application and they are super easy to override or assign to different parts of your application. Imagine switching content from a normal HTML layout to a PDF or mobile layout with one line of code. How cool would that be? Well, it's that cool with ColdBox. Another big benefit of layouts, is that you can see the whole picture of the layout, not the usual cmodule calls where tables are broken, or divs are left open as the module wraps itself around content. The layout is a complete html document and you basically describe where views will be rendered. Much easier to build and simpler to maintain.

Another important aspect of layouts is that they can also consume other layouts as well. So you can create nested layouts very easily via the `renderLayout()` method.



Basic Layout

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"
[>
<html xmlns="http://www.w3.org/1999/xhtml"
<head>
<meta http-equiv="content-type" content="text/html; charset=ISO-8859-1"
<title>cfoutput#cc.title</cfoutput#title>
</head>
<body>
<cfoutput>
<!-- SES base -->
</cfoutput>
<base href="#getSetting('htmlBaseURL')#"
</cfoutput>
</head>
<body>
<cfoutput>
<!-- Render a header view here and cache it for 30 minutes.-->
#renderView(view='V@header', cache=true, cacheTimeout='30')#
</cfoutput>
<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='V@footer', cache=true, cacheTimeout='30')#
</cfoutput>
</body>
</html>
```

Nested Layouts

You can also wrap layouts within other layouts and get incredible reusability. This is accomplished by using a method from our glorious Renderer plugin: `renderLayout()`. As always, refer to the [CFC API](#) for the latest method arguments and capabilities.

`renderLayout([any layout], [any view], [any module], [any args])`

So if I wanted to wrap my basic layout, we just showed you (basic.cfm), in a PDF wrapper layout (pdf.cfm) I could do the following:

```
<cfdocument pageType="letter" format="pdf">
<!-- Header -->
<cfdocument item="header">
</cfdocument>
<div>
#dateFormat(now,{MM DD, YYYY} at #timeFormat(now,{full})#
</div>
</cfdocument>
</cfdocumentitem>
<!-- Footer -->
```

```

</cfdocumentitem type="footer">
</cfoutput>
<div>
Page #cfdocument.currentpagenumber# of #cfdocument.totalpagecount#
</div>
</cfoutput>
</cfdocumentitem>
<!-- Main Content via nested layout -->
#renderLayout(layout="basic")#
</cfdocument>

```

That's it! The `renderLayout()` method is extremely power as it can allow you to not only nest layouts but actually render a-la-carte layout/view combinations also.

Default Layout

All ColdBox applications give you the capability of choosing a default layout that can be used to render all your views in. This is done by choosing that default layout in your [Configuration.CFC](#) in the `layoutSettings` structure:

```

//Layout Settings
layoutSettings = {
defaultLayout "basic.cfm"
}

```

With this little configuration snippet, all set views or main views that do not define a layout will use the default one by convention.

Default View

The **Default View** element is a very cool feature of the layout manager. This element allows you to declare the name of the view to render if the event handler does not set a view for rendering.

```

//Layout Settings
layoutSettings = {
defaultLayout "basic.cfm"
defaultView "movie.cfm"
}

```

Layout Helpers

This is a nifty little feature that enables you to create nice helper templates on a per-layout or per-layout-folder basis. If the framework detects the helper, it will inject it into the rendering layout so you can use methods, properties or whatever. All you need to do is follow a set of conventions. Let's say we have a layout in the following location:

```

/layouts
/general
+main.cfm

```

Then we can create the following templates

- `mainHelper.cfm` : Helper for the `main.cfm` layout.
- `generalHelper.cfm` : Helper for any layout in the general folder.

```

/layouts
/general
+main.cfm
+mainHelper.cfm
+generalHelper.cfm

```

That's it. Just append **Helper** to the layout or folder name and there you go, the framework will use it as a helper for that layout specifically.

Important : Please note that layout helpers will be inherently available to any view rendered inside of the layout.

Overriding Layouts

Ok, now that we have started to get funky, let's keep going. How can I change the layout on the fly for a specific view? Very easily, using yet another new method from the `event` object, called `setLayout()`

```
event.setLayout( name )
```

This is great, so we can change the way views are rendered on the fly programmatically. We can switch the content to a PDF in an instant. So let's do that

```

function home(event, rc, prc) {
if( event.valueExists("print") ){
event.setLayout("layout.PDF");
}

// logic here

// set view
event.setView("general/home");
}

```

WOW!! That easy! We check if a variable `print` is sent in and if it is, we override the layout with the `event.setLayout()` method. Another interesting technique is using some of the implicit execution points of a handler: `preHandler()` & `postHandler()`. You can use these implicit events to determine a layout for an entire event handler.

```

function preHandler(event, action, eventArguments) {
event.setLayout("layout.PDF");
}
function postHandler(event, action, eventArguments) {
event.setLayout("layout.PDF");
}

```

But you can even get more creative and decide to either create a `preProcessInterceptor` or a request start handler and listen for certain environment changes and switch layouts. You can then completely take control of how views will be rendered according to maybe browser type, mobile usage, etc.

```

function onRequestStartHandler(event, rc, prc) {
if( mobileLayout ){
event.setLayout("layout.Mobile");
}
}

```

Implicit Layout/View Declarations

Now that we have seen what layouts and views are, where they are located and some samples, let's dig deeper. There is a special section in the configuration file of an application just for layouts and views. This section is the **layouts** section and you can find much more in detail information in the [Configuration.CFC](#) guide. Let's look at a sample declaration below:

```

//Register Layouts
layouts = {
name "login";
file "Layout.tester.cfm";
views "#vLogin.test";
folders "#tags.pdf/single";
};

```

This setting allows you to implicitly define layout to view/folder assignments without the need of programmatically doing it. This is a time saver and a nice way to pre-define how certain views will be rendered. Let's see more examples:

```

//Register Layouts
layouts = {
name "popup";
file "Layout.Popup.cfm";
views "#login.test";
folders "#tags.pdf/single";
},
{
name "help";
file "layout.help.cfm";
folders "#help";
};
};

```

In the sample, we declare the layout named `popup` and points to the file `Layout.Popup.cfm`. We can then assign it to views or folders:

- **Views** : The views to assign to this layout (no `cfm` extension)
- **Folders** : The folders and its children to assign to this layout. (regex ok)

This is cool, we can tell the framework that some views and some folders should be rendered within a specific layout. Wow, this opens the possibility of creating nested applications that need different rendering schemas! Is it that easy? Yes it is!

Layouts From A Module

If you need the set layout to be rendered from a specific [ColdBox Module](#) then use the `module` argument alongside any other argument combination:

```

component name="general" {
function index(event, rc, prc) {
// call some model for data and put into the request collection
prc.myQuery = getModuleService().getData();
// set the view for rendering
event.setLayout(layout="login,module="contentbox");
}
}

```

By using the `module` argument, you are explicitly telling the ColdBox renderer to look for the layout inside of the passed module layout's convention.

Layout Events

All rendered layouts have associated events that are announced whenever the layout is rendered. These are great ways for you to be able to intercept when layouts are rendered and transform them, append to them, or even remove content from them in a completely decoupled approach. The way to listen for events in ColdBox is to write [Interceptors](#), which are essential simple CFC's that by convention have a method that is the same name as the event they would like to listen to. Each event has the capability to receive a structure of information which you can alter, append or remove from. Once you write these [Interceptors](#) you can either register them in your [Configuration File](#) or [programmatically](#).

Event	Data	Description
<code>preLayoutRender</code>	<ul style="list-style-type: none"> • layout - The name of the layout to render • view - The name of the view to render if any • module - The module name of the layout if any • args - The arguments to pass into the layout + view (if any) • viewModule - The name of the module the view will be rendered from 	Executed before any layout is rendered

postLayoutRender All of the data above plus:
 ● **renderedLayout** - The layout + view contents that was rendered Executed after a layout was rendered

You can disable the layout events on a per-rendering basis by passing the **prePostExempt** argument as true when calling **renderLayout()** methods.

Rendering Data

ColdBox provides you with a utility method called **renderData()** located in the event object (request context) that can be used from your event handlers. This method will provide you with the ability to render data back to the browser or to a remote caller without rendering or creating views. The framework will take care of marshalling (converting) the data for you and return it back. This is an incredible tool to use when doing AJAX or remote interactions from Flex or air or when building RESTful web services. You can also do your own conversions if you like, but it would be your responsibility to format the data correctly, set the correct headers, encoding and content type. We call these view transformers and can be done very easily!

```
public any renderData(string type='HTML', any data, string contentType='', [string encoding='utf-8'], [numeric statusCode='200'], [string statusText=''], [string location=''], [string jsonCallback=''], [string jsonQueryFormat])
```

Out of the box ColdBox can marshal data (structs, queries, arrays, complex or even ORM entities) into the following output formats:

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

Global Parameters

Argument	Type	Required	Default	Description
type	string	true	HTML	The type of data to render. Valid types are JSON, JSONP, JSOINT, XML, WDDX, PLAIN/HTML, TEXT. The default is HTML or PLAIN. If an invalid type is sent in, this method will throw an error
data	any	false	---	The data you would like to marshal and return by the framework
contentType	string	false	---	The content type of the data. This will be used in the cContent tag: text/html, text/plain, text/xml, text/json, etc. The default value is text/html. However, if you choose JSON this method will choose application/json, if you choose WDDX or XML this method will choose text/xml for you. The default encoding is utf-8
encoding	string	false	utf-8	The default character encoding to use
statusCode	numeric	false	200	The HTTP status code to send to the browser.
statusText	string	false	---	Explains the HTTP status code sent to the browser
location	string	false	---	Optional argument used to set the HTTP Location header

JSON Parameters

Argument	Type	Required	Default	Description
jsonQueryFormat	string	false	query	JSON Only: query or array
jsonAsText	boolean	false	false	If set to false, defaults content mime-type to application/json, else will change encoding to plain/text
jsonCallback	string	false	---	If you want to learn more about JSONP please visit the Wikipedia entry . In a nutshell, JSONP denotes that the JSON packet will be wrapped in a JavaScript callback function of your choice. This argument is the name of such function name.

XML Parameters

Argument	Type	Required	Default	Description
xmlColumnList	string	false	---	XML Only: Choose which columns to inspect, by default it uses all the columns in the query, if using a query
xmlUseCDATA	boolean	false	false	XML Only: Use CDATA content for ALL values. The default is false
xmlListDelimiter	string	false	---	XML Only: The delimiter in the list. Comma by default
xmlRootName	string	false	---	XML Only: The name of the initial root element of the XML packet

PDF Parameters

Argument	Type	Required	Default	Description
pdfArgs	structure	false	---	All the PDF arguments to pass along to the CFDocument tag.

The purpose of this method is for you to set the data to return, how to marshal or convert it and what type it is. The framework will then take care of everything for you, no need to abort the request or set a view or layout. The framework morphs into a remote framework:

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

// json marshalling
function getUsersJSON(event, rc, prc) {
    var qUsers = getUserService().getUsers();
    event.renderData(type='json', data=qUsers);
}

// jsonp marshalling
function getUsersJSONP(event, rc, prc) {
    var qUsers = getUserService().getUsers();
    event.renderData(type='jsonp', data=qUsers, jsonCallback='callback');
}

// restful handler
function list(event, rc, prc) {
    event.params['idFormat'] = 'html';
    rc.data = userService.list();

    switch rc.format {
        case 'json':
        case 'jsonp':
            event.renderData(type=rc.format, data=rc.data);
            break;
        default {
            event.setViewUsers/list;
        }
    }
}

// simple tests
function data(event, rc, prc) {
    var data = {
        name='ColdBox', awesome=true, ratings=[5,5,4,3]
    };
    event.renderData(data=data, type='text');
}
```

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

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

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

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

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

Custom Data Conversion

You can do custom data conversion by convention when marshalling CFCs. If you pass in a CFC as the **data** argument and that CFC has a method called **\$renderData()**, then the marshalling utility will call that function for you instead of using the internal marshalling utilities. You can pass in the custom content type for encoding as well:

The call to **renderData** from your handlers:

```
// get an instance of your custom converter
myConverter = getModule('converter');
// put some data in it
myConverter.setData( data );
// marshal it out according to your conversions and the content type it supports
event.renderData(data=myConverter, contentType=myConverter.getContentType());
```

The CFC converter:

```
component access="public" {
    property name="data" type="myType"
    property name="contentType"

    function init() {
        setContentType('text');
        return this;
    }
}
```

```

}
// The magical rendering
function renderdata(){
var d = {
    n = data.getName(),
    a = data.getAge(),
    c = data.getCo(),
    today = now()
};
return d.toString();
}
}

```

In this approach your `$renderdata()` function can be much more customizable than our internal serializers. Just remember to use the right `contentType` argument so the browser knows what to do with it.

Helper UDF's

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

```
coldbox.udflibrary = 'includeHelper:applicationhelper.cfm';
```

```
<<cfsetincludeUDF(includes/helpers/viewHelper)></cfm>
```

- The `includeUDF` method call will find the template and inject it to the layout/view combination
- The `UDFLibrary` setting injects the UDF's in the template to the handlers/layouts and views.

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

Viewlets (Portable Events)

The final chapter in this guide is to explain what we consider as a ColdBox viewlet or portable events. A viewlet is a self sufficient view or a widget that can live on its own, its data is pre-fetched and can just be rendered anywhere in your system. What in the world is this? Well, imagine a portal, in which each section of the portal is self-sufficient, with controls and data. You don't want to call all the handlers for this data for every single piece of content. It's not efficient, you need to create a separation. Well, a viewlet is such a separation that provides you with the ability to create sustainable events or viewlets. So how do we achieve this?

1. You will use the method `runEvent()` anywhere you want a viewlet to be displayed, usually in a layout or another view. This calls an internal event that will be in charge to prepare and render the viewlet.
2. Create the portable event for the viewlet like any other normal event with one exception.
3. The event will return a rendered view or content

Layout Code (Where I place my viewlet call)

```
<div id="leftbar">
<runEvent(event="ewlets.userinfo"prepostExempt=true)
</div>
```

This code just renders out the results of a `runEvent()` method call. I would suggest you look at the API docs to discover all parameters to the `runEvent()` method call. The `prePostExempt` argument makes the execution of the event faster as it skips any `preEvent/postEvent` interception calls.

To learn more about the `runEvent()` method, please refer to the [online API Docs](#)

Event Code (viewlets.userinfo)

```
function userinfo(event,rc,prc){
// place data in prc and prefix it to avoid collisions
prc.userinfo_qData = userService.getUserInfo();
// render out content
return renderView(viewlets/userinfo)
}

```

This handler code is where the magic happens. I talk to a service layer and place some data on the private request collection my viewlet will use. I then `return` the results of a `renderView()` call that will render out the exact viewlet I want. You can be more creative and do things like:

- render a layout + view combo
- render data
- return your own custom strings
- etc.

Important: We would suggest you namespace or prefix your private request collection variables for viewlets in order to avoid collisions from multiple viewlet events in the same execution thread.

Viewlet Code (viewlets/userinfo.cfm)

```
<<cfoutput>
<div#user-Info-Pane#div>
<div#username: #prc.userinfo_qData.username#
<div#last-Login: #prc.userinfo_qData.lastLogin#
</cfoutput>
```

My view is a normal standard view, it doesn't even know it is a viewlet, remember, views are DUMB!

Funky Viewlets

ColdBox also allows you to have **Funky Viewlets** and yes that is a real technical term. Funky viewlets allows you to pass arguments to the `runEvent()` method calls so you can simulate normal method calls and pass extra arguments to the events. So let's make our previous viewlet into a funky viewlet:

```
function userinfo(event,rc,prc,boolean widget=false){
// place data in prc and prefix it to avoid collisions
prc.userinfo_qData = userService.getUserInfo();
// render out content as widget or normal procedures
if( arguments.widget ){
return renderView(viewlets/userinfo)
}
event.setView(viewlets/userinfo)
}

```

What have I done? I have added a custom argument to my action signature: `boolean widget=false`. Then I split the rendering depending on this incoming argument. WOW! So how do I call my funky viewlet?

```
<div id="leftbar">
<runEvent(event="ewlets.userinfo"prepostExempt=true eventArguments=(widget=false))#
</div>
```

That easy! Just add another argument called `eventArguments` and now you can pass extra arguments to your event actions and cook up some **funky viewlets!**

Tips And Tricks

Some of the methods below will help you when dealing with views and their renderings.

I don't want to render anything

What if I don't want to render anything, just process. Then use the `noRender()` method, no need to abort or anything. This tells the framework to stop gracefully.

```
<<cfsetevent.noRender()
```

I don't want a coldbox debugging panel attached to this view

```
<<cfsetevent.showDebugPanel=false>
```

Is this a ColdBox proxy request

```
<<cfsetevent.isProxyRequest()
```

Am I in an ses application

```
<<cfsetevent.isSES()
```

Am I in an ajax call

```
<<cfsetevent.isAjax()
```

Render an event as a viewlet

```
<<cfoutput><runEvent(event="mv:main.leftnav"prepostExempt=true)</cfoutput>
```

What is the current layout?

```
<<cfsetevent.getCurrentLayout()
```

What is the current view

```
<<cfsetevent.getCurrentView()
```