

[<< Back to Dashboard](#)

# ColdBox-Ajax Integration

Covers up to version 3.5.0

## Introduction

There are several techniques and ways you can integrate ColdBox with any Ajax library, either to return xml/json/wddx or html snippets (views). In our [ColdBox Bundle](#) download we have over 20 different ColdBox sample applications and most of them use all kinds of Ajax integrations. There are no limitations on what JavaScript libraries you use, they all work as long as Ajax requests are made, we can even tell you via the [RequestContext](#) if a request is a normal request or an Ajax request (`event.isAjax()`).

As the developer you will have to decide what technique best suites your problem at hand, whether to return any type of data back to the caller via rendering JSON/WDDX/XML/CUSTOM or actual HTML blocks. ColdBox can do all of these return types natively and you don't even need to know how to marshall (convert) the data; ColdBox will do it for you if so needed. This guide focuses on Ajax integrations but most of these topics are covered already in our [Event Handlers](#) and [Layouts & Views](#) guides.

Most JavaScript examples are based on JQuery

## Returning HTML

HTML renderings is easy because that is what ColdBox does in its MVC core, render HTML. So to do a simple ajax call to the framework, you just call it as if you were doing it from the browser and requesting an event. The framework receives the request, process it, a view gets rendered and the ajax call receives the HTML snippet and you can div replace it anywhere you want.

```
// From JavaScript, we just execute normal ColdBox events:

// Div loading of data
$("#myDiv").load( 'index.cfm/Security/sayHello' );
$("#login").load( 'index.cfm/Security/loginForm' );

// Get
$.get( 'index.cfm/Security/sayHello' , function(data){
    $('#myDiv').html( data )
});

// Post
$.post( 'index.cfm/Security/sayHello' , {name: 'My Name' , age:40}, function(data){
    $('#myDiv').html( data )
});
```

## Event Handler Return

Any event handler can return HTML data by just returning the HTML from its action function. Then you can just call the event from any AJAX enabled library as a GET or POST.

```
function sayHello(){
    return "<h1>Say Hello</h1>" ;
}

function loginForm(){
    return renderView( "security/loginForm" );
}
```

## Render HTML

You can use the request context's [renderdata](#) method to return HTML as well.

```
function sayHello(){
    event.renderData(data= "<h1>Say Hello</h1>" );
}

function loginForm(){
    event.renderData(data=renderView( "security/loginForm" ));
}
```

## Layout+View

With a normal Ajax GET or POST you can also just return a normal ColdBox layout + view combination, which is a normal ColdBox Event.

```
function loginForm(){
    // setup here.
    event.setView(view= "security/loginForm" );
}
```

This event would render out the `security/loginForm.cfm` template in the default layout back to the calling user, whether Ajax or not. However, sometimes we might just want the view to be returned with no layout at all:

```
function loginForm(){
    // setup here.
    event.setView(view= "security/loginForm" , noLayout= true );
}
```

This would just render out the set view back to the caller. Sometimes it is also best practice to create an Ajax layout where you can control how HTML ajax requests are rendered back:

```
function loginForm(){
    // setup here.
    event.setView(view= "security/loginForm" , layout= "Ajax" );
}
```

## Layout Code:

```
<--- Remove CF Output --->
<cfsetting showdebugoutput= "false" >
<--- Remove the ColdBox Debugger --->
<cfset event.showdebugpanel( "false" )>
<--- Render a messagebox, just for kicks --->
<cfset WriteOutput(getPlugin( "messageBox" ).renderit()) >
<--- Render the set View --->
<cfset WriteOutput(renderView()) >
```

## Contents

- [ColdBox-Ajax Integration](#)
  - [Introduction](#)
  - [Returning HTML](#)
    - [Event Handler Return](#)
    - [Render HTML](#)
    - [Layout+View](#)
  - [Returning DATA \(XML/JSON/WDDX/ANY\)](#)
  - [The ColdBox Proxy](#)
    - [What is the proxy?](#)
    - [Techniques](#)
    - [How ?](#)
  - [ColdBox Debugger](#)

The first thing we do is remove the ColdFusion debuggin, if available. Then we use the **special magic** function in the event object to disable the coldbox debugger: `event.showdebugpanel( boolean )` Then we just output a messagebox if available and then we output the view to be rendered out which is set by any event handler. That's it. The interesting methods here are that I tell ColdBox to remove the debugger panel if I am in debugging mode. This layout is then used to render any view that is brought via Ajax. This opens a world of ideas on how it can be so flexible for your view renderings.

## Returning DATA (XML/JSON/WDDX/ANY)

The ColdBox [RequestContext](#) has an awesome method: `event.renderData()` that was built specifically for you to interact remotely with the framework and return data either via Ajax or RESTful requests. Our [Layouts and Views](#) guide goes in depth about `renderdata()`. This method can produce for you:

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

```
// 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 getUsersJSON(event,rc,prc){
    var qUsers = getUserService().getUsers();
    event.renderData(type= "jsonp" ,data=qUsers,jsonCallback= "myCallback" );
}

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

    rc.data = userService.list();

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

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

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

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

## The ColdBox Proxy

One of the great tools you will find in the ColdBox Platform is the [ColdBox proxy](#). This proxy element converts this MVC framework into a **remote-driven framework**. You can use it for integrating Flex/Air/AJAX and Remote calls. In this guide we will only touch AJAX, but the concepts are the same. For an in depth guide, please read the [ColdBox Proxy Guide](#). So let's start with the basics.

### What is the proxy?

The proxy is just a simple CFC that you will use and interact with from remote systems like AJAX. You call it or use data binding with it or even use the great ColdFusion Ajax tags: `cfajaxproxy` with it. It has several utility methods that you can use when dealing with remote calls. Some are shown below, for a full list, please visit the [latest API](#). I would also recommend you check out the sample application called **CF8Ajax** in the bundle download. It contains all kinds of AJAX-Coldbox goodness!!

**Note :** Using the ColdBox Proxy for Ajax calls is not our preferred approach as direct calls to render out HTML or data is preferred. However, there are certain ColdFusion data binding tags or features that require a CFC to connect to. Then the ColdBox proxy will help you.

**Useful Methods** (There are more)

Method	Description
<code>process</code>	execute an event remotely. You call this method and pass in an event and any arguments you like. The framework will then simulate an event request and return you data or the entire request collection. This is determined by you by a setting in the configuration file. Your event handlers can now return data instead of setting views to render. The framework morphs into a remote framework.
<code>announceInterception</code>	The ability to announce an interception and send in a structure of data.
<code>getPlugin</code>	Ability to get a coldbox plugin
<code>getBean</code>	Ability to get a bean from an IoC container
<code>getModel</code>	Ability to get a model object from <a href="#">WireBox</a>
<code>tracer</code>	Ability to trace messages to the debugger
<code>getColdboxOCM</code>	Get a reference to ColdBox caches

## Techniques

There are two techniques when interfacing with the proxy:

1. Calling the `process()` method or calling a method that uses the `process()` method to execute an event within your application.
2. Calling a method in a proxy object that gets its data from a service layer directly via the object retrieval methods.

Which one you use, well depends on your requirements and interactivity. If you have a service layer that can provide you with data directly with no request flow or security, then use method 2, else maybe method 1 is your cup of tea.

## How ?

The greatest question of all. Well, in the distro application template you will find a basic `coldboxproxy.cfc` that can be used for remote operations. The inherent basics here is that you have a component that extends the base `coldbox proxy class: extends=coldbox.system.extras.ColdBoxProxy`. Below is this simple cfc

```
<cfcomponent name="coldboxproxy" output="false" extends="coldbox.system.extras.ColdboxProxy" >
<!-- You can override this method if you want to intercept before and after. -->
<cffunction name="process" output="false" access="remote" returntype="any" hint="Process a remote call and return data/objects back." >
<cfset var results = "">

<!-- Anything before -->

<!-- Call the actual proxy -->
<cfset results = super.process(argumentCollection=arguments) >

<!-- Anything after -->

<cfreturn results >
</cffunction>
</cfcomponent>
```

You can use this simple proxy to call events via the process method. How? Here is a javascript sample using `cfajaxproxy`:

```
<!-- Declare the CF Ajax Proxy HERE -->
<cfajaxproxy cfc="coldboxproxy.cfc" jsclassname="cbProxy" >

<script type="text/javascript" >
var getArtists = function () {
var cbox = new cbProxy();
// Setting a callback handler for the proxy automatically makes
// the proxy's calls asynchronous.
cbox.setCallbackHandler(populateArtists);
cbox.setErrorHandler(myErrorHandler);
// The proxy getArtists function represents the CFC
// getArtists function.
cbox.process(event: 'artists.list' );
}
</script>
```

This calls the process method with an argument of `event = artists.list`. You can extrapolate how to do more stuff from here. So let's do another sample, let's add a method to our proxy so we can do data binding.

```
<cffunction name="getArtists" output="false" access="remote" returntype="Any" hint="Process a remote call and return data/objects back."
<cfargument name="ARTISTID" type="numeric" required="false" default="0">
<cfset var ReturnValue = "" />
<!-- Its very interesting.. how I am interacting with service-layer, just bypassing controller layer -->
<cfset ReturnValue = getBean( "ArtService" ).getArtist(argumentCollection=arguments) />

<cfreturn ReturnValue >
</cffunction>

<cffunction name="getNames" output="false" access="remote" returntype="Any" hint="Process a remote call and return data/objects back."
<cfset var qry = "" />
<!-- CFSELECT (bind) -->
<cfset var TwoDimensionalArray = ArrayNew(2) />

<!-- Get Qry Directly from ArtService.cfc -->
<cfset qry = getBean( "ArtService" ).getArtist() />

<cfset TwoDimensionalArray[1][1] = '0' />
<cfset TwoDimensionalArray[1][2] = 'Please select' />

<cfloop query="qry" >
<cfset TwoDimensionalArray[qry.CurrentRow + 1][1] = trim(qry.ARTISTID) >
<cfset TwoDimensionalArray[qry.CurrentRow + 1][2] = trim(qry.FIRSTNAME & chr(32) & qry.LASTNAME) >
</cfloop>

<!-- Anything after -->
<cfreturn TwoDimensionalArray >
</cffunction>
```

The two methods above go directly to the service layer by using the convenience method of `getBean()`. Then some js calls using `cfajaxproxy`:

```
<script language="javascript" >
var getArtistDetails = function (id){
document.getElementById( 'empData' ).innerHTML = 'Please wait! loading data...<br><br>' ;
var e = new cbProxy();
e.setCallbackHandler(populateArtistDetails);
e.setErrorHandler(myErrorHandler);
// This time, pass the employee name to the getArtists CFC
// function.
e.getArtists(id);
}
</script>
```

The last sample is another method added to our proxy object that gets some content:

```
<cffunction name="dspTab2" output="false" access="remote" returntype="any" hint="Process a remote call and return data/objects back." >
<cfset var results = "" />
<!-- call even handler to get query data etc -->
<cfset arguments[ "event" ] = "ehAjax.dspTab2" >

<!-- Call the actual proxy -->
<cfset results = super.process(argumentCollection=arguments) >

<cfreturn results >
</cffunction>
```

This simple method executes the `process()` method on its base class, our `coldbox proxy`, and then returns the results in whatever format they come in. So our `ehAjax.dspTab2` handler can look something like this:

```
<!-- tab2 content -->
<cffunction name="dspTab2" access="public" returntype="any" output="false" >
<cfargument name="Event" type="coldbox.system.beans.requestContext" >
<!-- send back to proxy -->
<cfset event.renderData(type="plain",data=renderView( 'ajax/vwTab2' )) />
```

```
</cffunction>
```

How cool is that, we just used the renderData method to render a view remotely. How COOL is that? You can see from this example, that ColdBox truly offers flexibility and great interaction with any remote caller.

Some more samples below:

```
//CFGRID data binding
<cfform>
  <cfgrid name = "FirstGrid"
    format = "html"
    font = "Tahoma"
    fontsize = "12"
    pageSize = "10"
    width = "100%"
    preservePageOnSort = "yes"
    bind = "cfc:coldboxproxy.getAllArtist({cfgridpage},{cfgridpagesize},{cfgridsortcolumn},{cfgridsortdirection})"
  >
    <cfgridcolumn name = "ARTISTID" display = "true" header = "ARTIST ID" />
    <cfgridcolumn name = "ARTNAME" display = "true" header = "Name" />
    <cfgridcolumn name = "FIRSTNAME" display = "true" header = "First Name" />
    <cfgridcolumn name = "LASTNAME" display = "true" header = "Last Name" />
    <cfgridcolumn name = "EMAIL" display = "true" header = "Email" />
  </cfgrid>
</cfform>

//AUTO SUGGEST
<cfform>
<h2> CFINPUT Auto-Suggest: </h2>
<p> Type <strong> Ma</strong> or <strong> Ch</strong> </p>
<cfinput type = "text"
  name = "employeeName"
  autosuggestminlength = "2"
  autosuggest = "cfc:coldboxproxy.SearchName({cfautosuggestvalue})"
>
</cfform>

<!-- CF8 cfgrid using ajax cfform is mandatory for using ajax stuff -->
<h1> cfselect bind to remote cfc: </h1>
<cfoutput>
<cfform name = "mycfform" >
  <!-- The States selector. The bindonload attribute is required to fill the selector. -->
  <cfselect name = "ArtistID" bind = "cfc:coldboxproxyS.getNames()" bindonload = "true" >
    <option name = "0">--select-- </option>
  </cfselect>
</cfform>
</cfoutput>
```

## ColdBox Debugger

As you know, ColdBox provides you (The Developer) with many great tools and visual representations of your application. The ones that are invaluable for AJAX/Remote development are the **cache panel monitor** and the **execution profiler monitor**. You can read more about these panels by [Clicking Here](#).

Here is a screenshot of the Cache Monitor:

**CacheBox**

Open Cache Monitor: CacheBox ExpireAll() CacheBox ReapAll()

CacheBox ID: 942813171

Configured Caches: TEMPLATE,default

Scope Registration: {ENABLED={false},SCOPE={server},KEY={cachebox}}

Performance Report For: default Cache Regenerate Report

Cache Name: default [class=coldbox.system.cache.providers.CacheBoxColdBoxProvider]

Performance: Hit Ratio: 83.62% ==> Hits: 97 | Misses: 19 | Evictions: 0 | Garbage Collections: 0 | Object Count: 21

JVM Memory Stats: 34.47 % Free | Total Assigned: 236,096 KB | Max: 236,096 KB

Last Reap: Aug-07-2010 07:56:17 PM

Cache Configuration: Show/Hide

Cache Content Report

Reload Contents Expire All Keys Clear All Events Clear All Views

Object	Hits	Timeout	Idle Timeout	Created	Last Accessed	Status	CMDS
cbbox_customplugin-jsmin	2	0	30	Aug-07 07:56:18 PM	Aug-07 07:56:18 PM	Alive	DEL
cbbox_interceptor-autowire	1	0	30	Aug-07 07:56:17 PM	Aug-07 07:56:17 PM	Alive	DEL
cbbox_interceptor-deploy	1	0	30	Aug-07 07:56:17 PM	Aug-07 07:56:17 PM	Alive	DEL

Here is screenshot of the execution profiler:

**ColdBox Execution Profiler Report**

Monitor Refresh Frequency (Seconds):

Profilers in stack 4 / 10

Below you can see the incoming request profilers. Click on the desired profiler to view its execution report.

**05/01/2012 02:07:00.143 AM (127.0.0.1)**

Timestamp	Execution Time	Framework Method
02:06:59.351 AM	0 ms	invoking runEvent [Main.onRequestStart]
02:06:59.361 AM	0 ms	invoking runEvent [preHandler] for forgebox:manager.index
02:07:00.99 AM	738 ms	invoking runEvent [forgebox:manager.index]
02:07:00.142 AM	29 ms	rendering View [manager/index.cfm]
02:07:00.143 AM	38 ms	rendering Layout [forgebox.main.cfm]

**05/01/2012 02:06:54.943 AM (127.0.0.1)**

**05/01/2012 02:06:51.848 AM (127.0.0.1)**

Timestamp	Execution Time	Framework Method
02:06:51.836 AM	0 ms	invoking runEvent [Main.onRequestStart]
02:06:51.841 AM	1 ms	invoking runEvent [General.index]
02:06:51.848 AM	1 ms	rendering View [home.cfm]
02:06:51.848 AM	4 ms	rendering Layout [Layout.Main.cfm]

**05/01/2012 02:06:36.189 AM (127.0.0.1)**

What you will find in the execution profiler that is invaluable, is that if you are doing ajax calls via the coldbox proxy, then you can actually see that data that comes in a request, then see the execution of the request, and then finally see the outcome of the request. No need to figure out what in the world happened during that call. The monitor is an essential tool in today's web application development.