<< Back to Dashboard

ColdBox Proxy Guide

Contents

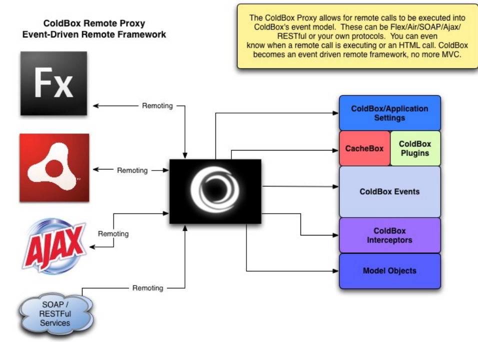
- ColdBox Proxy Guide
  - Introduction
  - Getting Started
    - AppMapping
    - Proxy Example
  - The Base Proxy Object
    - Expanding The Proxy
  - The Event Handlers
    - Distinguishing Request Types
    - RenderData()
  - Proxy Events
  - Ajax Data Binding & More
  - Flex Integration
  - Standard Return Types
  - Execution Profiler Monitor
  - Caveats & Gotchas
  - Conclusion

Covers up to version 3.5.0

Introduction

The ColdBox proxy enables remote applications or technologies like Flex, AIR, SOAP WebServices, RESTful Webservices and AJAX to communicate with ColdBox and provide an event driven model framework for those applications or for it to act as an enhanced service layer. Not only that, but you can reinitialize the entire application, get settings, announce custom or core interceptions, execute events, and so much more. You can create custom interceptor chains for your model that can be executed asynchronously when a user hits a save record button for example. You can create a Service Layer with built-in environmental settings, logging, error handling, event interception and chaining, you name it, and the possibilities are endless.

> The one key feature here, is that ColdBox morphs into a remote event-driven framework and no longer an MVC framework that produces HTML.



We not only give you the ability to do remote calls but also to monitor them. ColdBox has an execution monitor that can help you debug and analyze remote calls if you are in debug mode. This allows you to know what happens on asynchrounous calls from Flex/Air/SOAP, etc. The remote functionality of the framework enables you to actually create any amount of front ends using the same reusable ColdBox and model code. The code is the same, you create event handlers, you interact with a request collection, with core and custom plugins, but you don't set views or layouts because the framework is now a remote framework for your model. So what do you do, well, return data, arrays, xml, value objects. Anything, right from within the event handlers or setup a configuration setting that tells the framework to always return the request collection. You can also just create remote proxies to your service components (if using a service layers approach), so you can go directly to any object factory to request for services, interact with them and return results. The ColdBox proxy gives you flexibility in all aspects.

Getting Started

Most remote APIs are strongly typed so it makes sense to create as many ColdBox proxy objects as you see fit. Don't just create one proxy with 1000 methods on it. Try to apply identity to these objects as well. We also recommend you create a **remote** folder in your application where you can store all your remote proxy objects.

```
/Application
  /remote
    + MyProxy.cfc
```

The concept behind the ColdBox proxy is to create CFC's that extend our proxy class: **coldbox.system.remote.ColdboxProxy**. This will give you the ability to locate and talk to your running ColdBox application. However, since some of these requests won't be done via HTTP but other protocols like Flex/Air, your ColdBox application must know where in your server the application is located in. By default, when using HTTP calls, ColdBox can auto-locate your application with no issues at all, but with Flex/AIR or other protocols you must set this location in your *Application.cfc*.

AppMapping

In your *Application.cfc* you will find a directive called: **COLDBOX_APP_MAPPING**:

```
<cfset COLDBOX_APP_MAPPING = "">
```

This tells the framework where in the web server (sub-folder) your application is located in. By default, your application is the root of your website so this value is empty or / and you won't modify this. But if your application is in a sub-folder then add the full instantation path here. So if your application is under *apps/myApp* then the value would be:

```
<cfset COLDBOX_APP_MAPPING = "apps.myApp">
```

The AppMapping value is an instantiation path value

Proxy Example

The concept of a Proxy is to give access to another system. As Wikipedia mentions:

> "A proxy, in its most general form, is a class functioning as an interface to something else. The proxy could interface to anything: a network connection, a large object in memory, a file, or some other resource that is expensive or impossible to duplicate" by Wikipedia

The proxy will give you access to your entire ColdBox application assets but also allow you to proxy in request to the normal ColdBox event model. You will do this via our **process()** method. This method expects a **event** argument to be passed to it which is the event that will be executed for you and all the other arguments passed to this method will be converted and merged into the RequestContext's request collection. Then your event handlers can respond to these requests just like normal requests and even return data back to the caller. The advanced ColdBox template gives you a sample proxy object in your *remote/MyProxy.cfc* folder.

```
<cfcomponent name="MyProxy" output="false" extends="coldbox.system.remote.ColdboxProxy">

 <cffunction name="yourRemoteCall" output="false" access="remote" returntype="YourType" hint="Your Hint">
  <cfset var results = "">

  <!--- Set the event to execute --->
  <cfset arguments.event = "">

  <!--- Call to process a coldbox event cycle, always check the results as they might not exist. --->
  <cfset results = super.process(argumentCollection=arguments)>

  <cfreturn results>
 </cffunction>

</cfcomponent>
```

This simple proxy object extends the ColdBox Proxy class that gives you all the remote abilities. Then it is up to you to create methods that will respond to either Flex/Air/Soap or now with ColdFusion 10; Restful services.

The Base Proxy Object

As you can see from the code above, this proxy inherits from the **coldbox.system.remote.ColdboxProxy** class. This is the key to it all, this base class has all the necessary hooks for your proxy to work. Why is it inherited? Why not just use that one? Well, the answer is that every project is different and I believe in empowering the developer. Therefore, you have your own class in which you can expose any other remote methods as you need. You only need to know the methods that are available to you from the base class. The following are the most commonly used methods in the base proxy class, for an in-depth review of the methods, please visit the CFC API:

Method	Description
announceInterception(state, data)	Processes a remote interception.
getBean()	Get a bean from the ioc plugin
getCacheBox()	Get a reference to CacheBox
getColdboxOCM(cacheName='default')	Get a reference to a named cache provider
getController()	Returns the ColdBox controller instance
getInterceptor()	Get a named interceptor
getIoCFactory()	Get the IoC factory object
getLogBox()	Get a reference to LogBox
getModel(name,dsl,initArguments)	Get a Wirebox model object
getPlugin()	Get a core or custom plugin
getWireBox()	Get a reference to WireBox
process()	Processes a remote call that will execute a coldbox event and returns data/objects back.
loadColdBox()	Gives you the ability to load any external coldbox application in the application scope. Great for remotely loading any coldbox application, it can be located anywhere.
tracer()	Ability to send tracer messages to the debugger

You can use any of those methods in your own proxy or even override them. With this in mind, you can create a set of remote objects that inherit from the ColdBox proxy that will be your proxy layer to any remote communication if necessary or just have a main proxy component used to call in to your events.

Expanding The Proxy

Below is a custom method that I created in my own application proxy for retrieving application settings. **I DO NOT recommend this, since it will expose your settings. This is just a sample**

```
<---  Get a setting --->
<cffunction name="getSetting" hint="I get a setting from the FW Config structures. Use the FWSetting boolean argument to retrieve from the fwSettingsStruct." access="remote" returntype="struct" output="false">
<---************************************************************** --->
<cfargument name="name"     type="string"   hint="Name of the setting key to retrieve">
<cfargument name="FWSetting" type="boolean"   required="false" hint="Boolean Flag. If true, it will retrieve from the fwSettingsStruct else from the configStruct. default is false." default="false">
<---************************************************************** --->
<cfscript>
var cbController = "";
var setting = "";

cbController = getController();


//Get Setting else return ""
if( cbController.settingExists(argumentCollection=arguments) ){
 setting = cbController.getSetting(argumentCollection=arguments);
}

//Get settings
return setting;
</cfscript>
</cffunction>
```

This method basically retrieves a setting via the controller and returns it. You can basically create any amount of methods that correspond to your proxy object or just leave the main methods: **process() and announceInterception()**.

The Event Handlers

The event handlers that you will produce for remote interaction are exactly the same as your other handlers, with the exception that they have a return type and return data back to the caller; our proxies. Then our proxies can strong type the return data elements:

Handler:

```
function getCachekEys(event,rc,prc){
 return getColdBoxOCM( rc.cacheProvider ).getKeys();
}

function listUsers(event,rc,prc){
 prc.data = userService.list();

 if( event.isProxyRequest() ){
  return prc.data;
 }

 event.setView("users/listUsers");
}
```

Proxy:

```
array function getCachekEys(string cacheProvider="default"){
 arguments.event = "proxy.getCacheKeys";
 return process(argumentCollection=arguments);
}

string function getUsersJSON(){
 arguments.event = "proxy.listUsers";
 return serializeJSON( process(argumentCollection=arguments) );
}
```

Distinguishing Request Types

Now, what if you want to distinguish between a normal request and a proxy request? Well, the request context object, most commonly known as the **event** object has a method called:

- **isProxyRequest**: This boolean method determines what type of request is being executed.

This is extremely useful if you are doing path operations. Why? Well, when you call a coldfusion component via flash remoting or live data cycle services, the **expandPath, getCurrentTemplatePath, getBaseTemplatePath**, and other template path methods will always give you the path according to the file you are currently executing. This is due to how ColdFusion deals with remote calls. So you can use the method below to distinguish and load accordingly:

```
<---  Use the isProxyRequest() --->
<cfif event.isProxyRequest()>
 <cfset getPlugin("JavaLoader").setup( listToArray( ExpandPath("../includes/helloworld.jar") ) )>
<cfelse>
 <cfset getPlugin("JavaLoader").setup( listToArray( ExpandPath("includes/helloworld.jar") ) )>
</cfif>
```

The above code will be executed within the proxy as if it's in the handler's directory. So to expand the jar file path in the includes directory, you have to go back a level when using the proxy and just directly if not. This is very important to grasp, since its not !ColdBox doing this, but ColdFusion. Executions via http protocols might not exhibit the same behavior.

RenderData()

The ColdBox Proxy also has the ability to use the RequestContext's *renderData()* method. So you can build a system that just uses this functionality to transform data into multiple requests. Even have the ability for the same handler to respond to REST/SOAP and MVC all in one method:

Event Handler:

```
function list(event,rc,prc){
 event.paramValue("format","html");

 prc.data = userService.list();

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

Proxy:

```
string function getUsers(string format="json"){
 validateFormat( arguments.format );
 arguments.event = "users.list";
 return process(argumentCollection=arguments);
}
```

This handler can now respond to HTMl requests, SOAP requests, Flex/Air Requests and even RESTFul requests. How about that!

Proxy Events

The ColdBox Proxy also has a different life cycle that you can follow: See RequestLifecycles. All of a request's interception points fire with one addition: **preProxyResults**. This event fires right before the proxy returns results back to proxies. This is a great way to do transformations, logging, etc.

```
component{

 function preProxyResults(event, interceptData){
  log.debug("Proxy request finalized", interceptData.proxyResults );
 }

}
```

Ajax Data Binding & More

Please visit the ColdBox & Ajax Integration Guide to learn how to use advanced techniques with the proxy to do data binding, mixing in with the **cfajaxproxy** tag and much more. Below are just some simple examples:

```
<---  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>


//ColdBox Proxy Code
<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 = "" />
       <---  It's very interesting.. how I am interacting with service-layer, just bypassing controller layer --->
       <cfset ReturnValue = getModel("ArtService").getArtist(argumentCollection=arguments) />

       <cfreturn ReturnValue>
</cffunction>

<cffunction name="getNames" output="false" access="remote" returntype="Array" 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 = getModel("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>
```

Flex Integration

This is the last step of the guide, so let's review for a moment.

- We have configured the application to return data/objects from the event handlers
- We have customized our proxy and made sure it inherits from the base proxy.
- We have created some event handler methods
- We learned how to determine when a remote call was made via the event object

Now our last step is to actually show some remote calls. For this example, I will be showing some Flex code. I am no big Flex expert, but below are some of the flex sample codes on how to call the ColdBox proxy. I recommend encapsulating the calls to the ColdBox proxy into a class of its own as a delegate class or however you see it fit in your Flex application architecture. The sample below is very flat.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute">

<mx:Script>
<![CDATA[
 import mx.rpc.events.FaultEvent;
 import mx.events.ItemClickEvent;
 import mx.rpc.events.ResultEvent;
 import mx.controls.Alert;
 import mx.collections.ArrayCollection;

 //My Proxy Path
 public var cbProxyPath:String = "coldbox.samples.applications.ColdboxFlexTester.webroot.coldboxproxy"

 /* PUBLIC FAULT Handler*/
 public function faultHandler(event:FaultEvent):void{
  Alert.show(event.fault.toString());
 }
 /* UTILITY METHOD TO GET A CBPROXY OBJECT, You can separate all this to a delegate class*/
 public function getColdBoxProxy():RemoteObject{
  var cProxy:RemoteObject = new RemoteObject("ColdFusion");
  cProxy.source= cbProxyPath;
  cProxy.showBusyCursor = true;
  cProxy.addEventListener( FaultEvent.FAULT, faultHandler );
  return cProxy;
 }
 /* Read the Cache Objects*/
 public function handleCacheResults(event:ResultEvent):void{
  var cacheItems:Object = new Object();
  var key:String;
  var array:Array = new Array();

  cacheItems = event.result;
  for( key in cacheItems ){
   array.push( { item:key, total: cacheItems[key] });
  }
  var collection:ArrayCollection = new ArrayCollection(array);
  cachechart.dataProvider = collection;
 }
 /* Call the proxy for cache objects*/
 public function readCache():void{
  var cProxy:RemoteObject = getColdBoxProxy();
  cProxy.process.addEventListener("result", handleCacheResults );
  cProxy.process({event:"ehFlex.getCacheItemTypes"});
 }
 ]]>
</mx:Script>

 <mx:PieChart id="cachechart"
            height="190"
            width="205"
            showDataTips="true" x="10" y="450">
       <mx:series>
          <mx:PieSeries field="total" nameField="item"
              labelPosition="callout"/>
       </mx:series>
    </mx:PieChart>
    <mx:Button x="45" y="420" label="Get Cache Chart" click="readCache()"/>

</mx:Application>
```

So what does this application do. Well let's start from the top. The first part of the application just imports some classes for us to use. Then we declare the path to our coldbox proxy:

```
//My Proxy Path
public var cbProxyPath:String = "coldbox.samples.applications.ColdboxFlexTester.webroot.coldboxproxy"
```

We then setup a public default error handler:

```
/* PUBLIC FAULT Handler*/
public function faultHandler(event:FaultEvent):void{
 Alert.show(event.fault.toString());
}
```

We then declare our mini proxy delegate. Again, this is for sample purposes, you must encapsulate this into a class of its own and even expand on it to make it a true delegate.

```
/* UTILITY METHOD TO GET A CBPROXY OBJECT, You can separate all this to a delegate class*/
public function getColdBoxProxy():RemoteObject{
 var cProxy:RemoteObject = new RemoteObject("ColdFusion");
 cProxy.source= cbProxyPath;
 cProxy.showBusyCursor = true;
 cProxy.addEventListener( FaultEvent.FAULT, faultHandler );
 return cProxy;
}
```

Now, you might ask, why create a delegate and not a remote object and just call it. Well, the problem lies in that you will always be calling the same method: **process()** on the proxy, but need to bind the results to different result handlers. Therefore, you need to create a delegate to process your request and assign a results handler for you. There are tons of ways to achieve what I am doing, I am doing the poor man's delegate.

Once I have done this, then I can create some object for me to display the cache in:

```
<mx:PieChart id="cachechart"
        height="190"
        width="205"
        showDataTips="true" x="10" y="450">
    <mx:series>
       <mx:PieSeries field="total" nameField="item"
           labelPosition="callout"/>
    </mx:series>
</mx:PieChart>
<mx:Button x="45" y="420" label="Get Cache Chart" click="readCache()"/>
```

This declares a pie chart object and a push button. Once the button get's clicked on it will execute the *readCache* method we will cover below.

```
/* Read the Cache Objects*/
public function handleCacheResults(event:ResultEvent):void{
 var cacheItems:Object = new Object();
 var key:String;
 var array:Array = new Array();

 cacheItems = event.result;
 for( key in cacheItems ){
  array.push( { item:key, total: cacheItems[key] });
 }
 var collection:ArrayCollection = new ArrayCollection(array);
 cachechart.dataProvider = collection;
}
/* Call the proxy for cache objects*/
public function readCache():void{
 var cProxy:RemoteObject = getColdBoxProxy();
 cProxy.process.addEventListener("result", handleCacheResults );
 cProxy.process({event:"ehFlex.getCacheItemTypes"});
}
```

The *readCache* method basically calls the *getColdBoxProxy* ***delegate method to get a remote object for the proxy. It then creates an event listener for it and calls the proxy. The listener is called handleCacheResults which then manipulates the results and renders the cache.***

Important: Please note that this is a sample Flex application that does not adhere to any Flex MVC techniques or enterprise best practices. It is here to learn how to call the proxy from it and start your process into flex-ColdBox integration. I highly suggest creating a separate AS3 class that will act as a ColdBox delegate that can call your methods, set call back handlers and fault handlers.

Standard Return Types

The ConfigurationCFC has one setting that affects proxy operation:

- **ProxyReturnCollection** : This boolean setting determines if the proxy should return what the event handlers return or just return the request collection structure every time a **process()** method is called. This can be a very useful setting, if you don't even want to return any data from the handlers (via the **process()** method). The framework will just **always** return the request collection structure back to the proxy caller. By default, the framework has this setting turned to **false** so you can have more control and flexibility.

Execution Profiler Monitor

ColdBox Execution Profiler Report

Monitor Refresh Frequency (Seconds): No Polling

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)

Close Monitor

This monitor will keep a stack of the latest X requests and their execution profiles that is all configurable via the ConfigurationCFC. This monitor can be kept side by side with your remote application or any front-end and give you execution profiles about your event model and show you event tracers. By default the request profiler is turned off, so you will have to turn it on via the new Debugger Settings in your configuration file. You can also choose the max number of requests to persist execution profiles for, so you can keep track of any amount of requests.

Important: The execution tracers and profilers will ONLY be logged if the user making the calls is in Debug Mode. To learn how to enter into debug mode, please review the configuration guide and the URL actions guide.

Caveats & Gotchas

The most important gotchas in using the coldbox proxy for remoting or even event gateways is pathing. Paths are totally different if you are using **expandPath()** or per-application mappings. Per-Application mappings can sometimes be a hassle for onSessionEnd(). So always be careful when setting up your paths and configurations for remoting. Try to always have the correct paths assigned and tested.

Conclusion

As you can see, calling the ColdBox proxy from flex or any remote interface is very easy, but extremely powerful. Some applications for the proxy are :

- Remote Event driven Model Framework
- One configuration file for all application GUI's
- One common application language for all backend operations
- Multiple GUI's running on one single ColdBox application
- Enhanced service layers (caching, logging, AOP, interceptions, etc)
- Create beautiful RIA applications with a solid backend
- Create widgets and application monitors.
- What your imagination dictates.

Indeed, the ColdBox proxy is a great feature to complement your already great ColdBox applications. You can now use a standardized language for application development in all your front-ends.