

[<< Back to Dashboard](#)

Unit Testing & Integration Testing

Covers up to version 3.5.0

Introduction

One of the best things you can do when you develop software applications is TEST! I know nobody likes it, but hey, you need to do it right? With the advocacy of frameworks today, you get all these great tools to build your software applications, but how do you test your framework code. ColdBox has revolutionized testing MVC and framework code, since you can unit test your event handlers, plugins, interceptors, model objects and even do integration testing and test your entire application with no browser at all. We have lots of testing enhancements via [MXUnit](#) and our very own Mocking and Stubbing library [MockBox](#) which will be essential to testing more Object Oriented applications.

Benefits

It might be that testing is tedious and takes time to get into the flow of Test Driven Development. However, there are incredible benefits to testing:

- Can improve code quality
- Quick error discovery
- Code confidence via immediate verification
- Can expose high coupling
- Encourages refactoring to produce testable code
- **Testing is all about behavior and expectations**

Resources

- [Wikipedia](#)
- [MXUnit](#)
- [ColdBox CheatSheet](#)

Testing Concepts

In computer programming, unit testing is a procedure used to **validate** that **individual** units of source code are working properly. A unit is the **smallest testable** part of an application. In procedural programming a unit may be an individual program, **function**, procedure etc, while in object-oriented programming, the smallest unit is always a **Class**; which may be a base/super class, abstract class or derived/child class. [Wikipedia](#)

The testing process will take us through a journey of exploration of our software in order to determine its quality, if it met the gathered requirements, work as expected, and provide the required functionality. However, all the testing that we do is mostly controlled and under certain conditions. It is extremely difficult to determine all factors and all software conditions. Therefore attaining a testing nirvana is unrealistic.

Functional Testing

[Functional testing](#) relies on the action of verification of specific requirements in our software. This can usually be tracked via use cases or Agile stories. "Can a user log in?", "Does the logout work", "Can I retrieve my password", "Can I purchase a book?". Our main focus of functional testing is that all these use cases need to be met.

Non-Functional Testing

[Non Functional Testing](#) are testing aspects that are not related to our specific requirements but more of the quality of our software.

- Can our software scale?
- Is our software performant?
- Can it sustain load? What is the maximum load it can sustain?
- What do we do to extend our security? Are our servers secure?

As developers, we sometimes forget the importance of non-functional testing, but what good is an application that meets the requirements but cannot be online for more than an hour? Both are extremely important and we must dedicate time to each area of testing so our software not only meets the requirements, but has great quality!

Bugs Cost Money

Cost To Fix		Time detected				
		Requirements	Design	Building	Testing	Post-Release
Time Introduced	Requirements	1x	3x	5-10x	10x	10-100x
	Design	---	1x	10x	15x	25-100x
	Building	--	--	1x	10x	10-25x

Kaner, Cem; James Bach, Bret Pettichord (2001). Lessons Learned in Software Testing: A Context-Driven Approach. Wiley. p. 4. ISBN 0-471-08112-4.

It is believed that the earlier a bug is found the cheaper it is to fix it. This is something to remember as when things affect our income is when we become babies and really want to fix things then. So always take into consideration that bugs do cost money and we can be proactive about it. So what are some things apart from unit testing that we can do to improve the quality of our software? Well, a part from applying our functional and non-functional testing approaches we can also do some static and dynamic testing of our software.

Static Testing

Static testing relies on human interaction not only with our code but with our developers and stakeholders. Some aspects can be code reviews, code/ui walkthroughs, design and modeling sessions, etc. Through experience, we can also attest that static testing is sometimes omitted or considered to be a waste of time. Especially when you need to hold up on production releases to review somebody else's work. Not only that, but sometimes code reviews can spark animosity between developers and architects as most of the time the idea is to find holes in code, so careful tact must be in place and also maturity across development environments. In true fun we can say: "Come on, take the criticism, it is for the common good!"

Dynamic Testing

On the other side of the spectrum, dynamic testing relies on the fact of executing test cases to verify software. Most of the time it can be the developers themselves or a QA team than can direct testing cases on executed code.

Developer Focus

This guide is written for developers, so what is our focus? Our focus is to get awareness about the different aspects of testing and what are some of the testing focuses for the development phase:

- **Unit Testing** - We should be testing all the CFCs we create in our applications. Especially when dealing with Object Oriented systems, we need to have mocking and stubbing capabilities (See [MockBox](#)).
- **Integration Testing** - Unit tests can only expose issues at the CFC level, but what happens when we put everything together? Integration testing in ColdBox allows you to test requests just like if they were executed from the browser. So all your application loads, interceptions, caching, dependency injection, ORM, database, etc.
- **UI Integration Testing** - Testing verification via the UI using tools like Selenium is another great idea!
- **Load Testing** - There are tons of free load testing tools and they are easy to test how our code behaves under load. Don't open a browser with 5 tabs on it and start hitting enter! We have been there and it don't work that great :)

Testing Vocabulary

Here are some terms that we need to start getting familiar with:

- **Test Plan**: A test plan documents the strategy that will be used to verify and ensures that a product or system meets its design specifications and other requirements.
- **Test Case**: A set of conditions or variables under which a tester will determine whether an application or software system is working correctly or not. In our ColdFusion space, it might be represented by an MXUnit test case CFC.
- **Test Script**: A set of instructions that will be performed on the system under test to test that the system functions as expected. This can be manual or automated. Sometimes they can be called runners.

Contents

- [Unit Testing & Integration Testing](#)
 - [Introduction](#)
 - [Benefits](#)
 - [Resources](#)
 - [Testing Concepts](#)
 - [Functional Testing](#)
 - [Non-Functional Testing](#)
 - [Bugs Cost Money](#)
 - [Static Testing](#)
 - [Dynamic Testing](#)
 - [Developer Focus](#)
 - [Testing Vocabulary](#)
 - [Testing Tools](#)
 - [MXUnit Installation](#)
 - [Testing Templates](#)
 - [Eclipse MXUnit Plugin Caveats](#)
 - [ColdBox Testing Classes](#)
 - [Integration Testing](#)
 - [Test Annotations](#)
 - [Common Methods](#)
 - [Common Mocking Methods](#)
 - [Test Setup](#)
 - [The Handler To Test](#)
 - [The Integration Test](#)
 - [Handler Returning Results](#)
 - [Plugin Testing](#)
 - [Handler Isolation Testing](#)
 - [Interceptor Testing](#)
 - [Model Object Testing](#)
 - [Tips & Tricks](#)
 - [Mocking HTTP Methods](#)
 - [Conclusion](#)

- **Test Suite:** A collection of test cases that are intended to be used to test a software program to show that it has some specified set of behaviors or met expectations.
- **Test Data:** A collection of pre-defined data or mocked data used to run against our test cases
- **Test Harness:** A collection of software and test data configured to test a program unit by running it under varying conditions and monitoring its behavior and outputs.
- **TDD:** Test Driven Development - A software development process that relies on the repetition of a very short development cycle: first the developer writes a failing automated test case that defines a desired improvement or new function, then produces code to pass that test and finally refactors the new code to acceptable standards.
- **Mock Object:** Simulated objects that mimic the behavior of real objects in controlled ways
- **Stub Object:** Simulated objects that have no behavior or implementations and can be controlled and injected with functionality
- **Assertions:** The function to test if a predicate is true or false: $X > 1, 4 = 5$

Testing Tools

We all need a great arsenal of tools when it comes to software development and testing. Here are some of our recommendations:

- **MXUnit** - The ColdFusion testing framework
- **ColdBox-MockBox** - MVC development and Mocking & Stubbing Framework
- **Adobe ColdFusion Builder** - Our preferred IDE - <http://www.adobe.com/products/coldfusion-builder.html>
- **ColdBox Platform Utilities Extension for CFBuilder** - A great collection of tools for working with ColdBox applications within ColdFusion Builder. - <http://coldbox.org/forgobox/view/ColdBox-Platform-Utilities>
- **CFEclipse** - Alternative ColdFusion IDE - <http://www.cfeclipse.org>
- **Apache ANT** - Automation - <http://ant.apache.org>
- **Jenkins CI Server** - Continuous Integration - <http://jenkins-ci.org>
- **Selenium** - UI Testing - <http://seleniumhq.org>
- **ColdBox Relax** - ColdBox RESTful modeling, testing and documentation - <http://www.coldbox.org/forgobox>
- **JMeter** - Load Testing - <http://jakarta.apache.org/jmeter>
- **Webstress Tool** - Load Testing - <http://www.paessler.com>

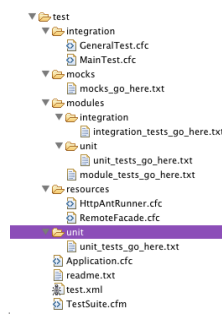
MXUnit Installation

The first step in starting with testing in your ColdBox applications is to get the testing framework, MXUnit installed and configured. You can follow the instructions at <http://www.mxunit.org> or you can also download our MXUnit installation documentation from our [CROX204 - Testing, Mocking, Stubbing Oh My!](http://www.coldbox.org/forgobox/view/ColdBox-Platform-Utilities) training course.

- **Download MXUnit 204 Chapter** - <https://forums-public.s3.amazonaws.com/CROX204-IntroToMXUnit.pdf>

This guide will show you how to install MXUnit, its basics and then how to install the Eclipse Plugin so you can do TDD via Adobe ColdFusion Builder or CFEclipse.

Testing Templates



The ColdBox download comes with an application templates folder or the CF Builder tools also generate ColdBox applications with a **test** directory that includes all you need to begin your testing adventure. So let's start by how we have partitioned these templates so we can start testing.

```
Advanced
|--test
|   |-- integration ( integration tests go here )
|   |-- mocks ( Manual mock objects or data go here )
|   |-- modules ( Testing for modules goes here )
|   |-- resources
|       |-- RemoteFacade.cfc ( To connect Eclipse to your Application )
|       |-- HttpAntRunner.cfc ( To connect ANT to your Application )
|-- unit ( Unit tests for model objects go here )
+ Application.cfc (Your testing application file)
+ test.xml (ANT script to automate your tests)
+ TestSuite.cfm (A test suite to run all your tests)
```

This test template or test harness includes folders for major test classes you will be creating, Eclipse & ANT connection resources, ANT scripts and so much more. This is a great starting point for your application testing adventure. The most important piece of your test harness is the *Application.cfc* file. Please note that your testing harness must have a different Application space than your normal ColdBox application. You do not want your tests to alter the real application and vice versa. So in this *Application.cfc* you will try to mimic all the real application's session, client, scopes, ORM, S3, settings, etc. The trickiest part to setup is ORM integration. You have to make sure that your paths and entities can be located by your tests which run in a separate application space.

Basic Application.cfc:

```
<cfcomponent output="false">
<!-- APPLICATION CFC PROPERTIES -->
<cfset this.name = "ColdBoxTestingSuite" & hash(getCurrentTemplatePath())>
<cfset this.sessionManagement = true>
<cfset this.sessionTimeout = createTimeSpan(0,0,30,0)>
<cfset this.setClientCookies = true>
</cfcomponent>
```

More Complex Application.cfc

```
<cfcomponent output="false">
<!-- APPLICATION CFC PROPERTIES -->
<cfset this.name = "ContentBoxTestingSuite" & hash(getCurrentTemplatePath())>
<cfset this.sessionManagement = true>
<cfset this.sessionTimeout = createTimeSpan(0,0,0,30)>
<cfset this.setClientCookies = true>

<cfscript>
// ORM Settings
this.ormEnabled = true;
// FILL OUT: THE DATASOURCE OF CONTENTBOX
this.datasource = "contentbox";
// FILL OUT: THE LOCATION OF THE CONTENTBOX MODULE
rootPath = replacenocase(replacenocase(getDirectoryFromPath(getCurrentTemplatePath()), "test/", "**"), "test/", "**");

this.mappings["/root"] = rootPath;
this.mappings["/contentbox-test"] = getDirectoryFromPath(getCurrentTemplatePath());
this.mappings["/contentbox"] = rootPath & "/modules/contentbox";
this.mappings["/contentbox-ui"] = rootPath & "modules/contentbox-ui";
this.mappings["/contentbox-admin"] = rootPath & "modules/contentbox-admin";
this.mappings["/contentbox-modules"] = rootPath & "modules/contentbox-modules";
this.mappings["/coldbox"] = rootPath & "/coldbox";

this.ormSettings = {
cflocation["./modules/contentbox"],
logSQL = true,
flushAtRequestEnd = false,
autoManageSession = false,
eventHandling = true,
eventHandler = "contentbox.model.system.EventHandler",
skipCFCWithError = true,
secondarycacheenabled = true,
cacheprovider = "ehCache"
};

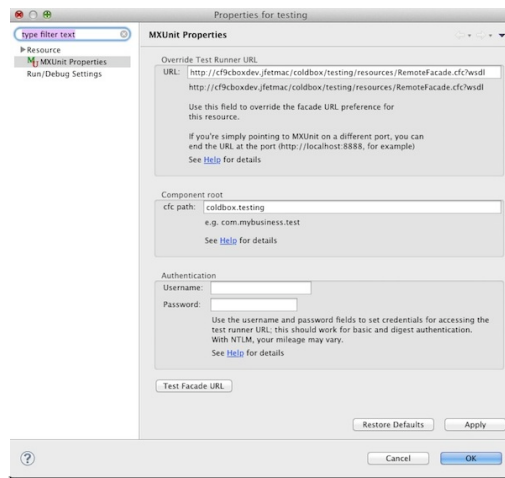
public boolean function onRequestStart(String targetPage){
// ORM Reload
ormReload();

return true;
}
</cfscript>
</cfcomponent>
```

Eclipse MXUnit Plugin Caveats

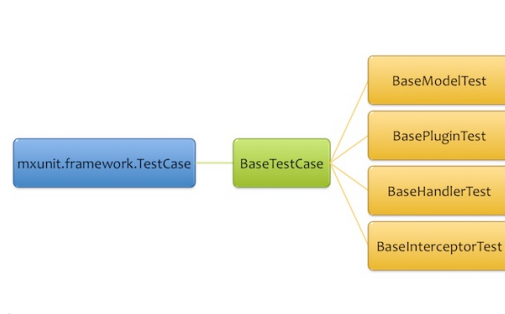
If you will be doing your testing from Eclipse then you must alter the location of the MXUnit Remote Facade via the MXUnit Eclipse Plugin. You can do this by reading our [Sample Chapter](#) from our 204 training course, visiting the [MXUnit documentation](#) or by doing the following:

- Right Click on your project
- Click on **MXUnit Properties**



Here you can place the location of the **RemoteFacade.cfc** according to where your application's Remote Facade is located in your `/test/resources/RemoteFacade.cfc` folder. Also make sure to fill out the `cfc.path` which is the dot notation path to the root of your project. If the project is the root, then this value is empty. This links Eclipse to **your application** and not the MXUnit application, which is by default. This is an extremely important step as you need for your tests to be executing within your testing harness.

ColdBox Testing Classes

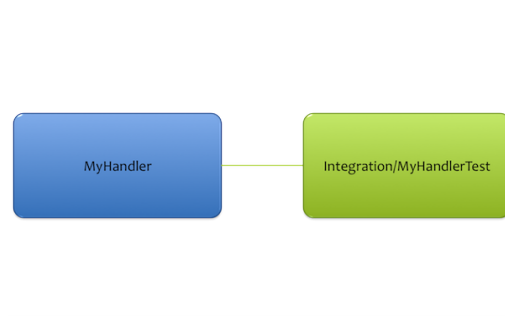


Before we begin our adventures in testing, let's review what classes does ColdBox give you for testing and where you can find them. From the diagram you can see that our pivot class for testing is the MXUnit **TestCase** class, which is typically used for any Unit Test. From that super class we have our own ColdBox **BaseTestCase** which is our base class for any testing in ColdBox and the class used for Integration Testing. We then spawn several child classes for targeted testing of different objects in your ColdBox applications:

- **BaseTestCase** - Used for Integration Testing
 - **BaseModelTest** - Used for model object testing
 - **BasePluginTest** - Used for plugin testing
 - **BaseInterceptorTest** - Used for interceptor testing
 - **BaseHandlerTest** - Used for isolated handler testing

Integration Testing

We will begin our adventure with integration testing. Integration testing allows you to test a real life request to your application without using a browser. Your test will load a new virtual application for you for each test; all aspects of your application are loaded: caching, dependency injection, AOP, etc. Then you can target an event to test and verify its behavior accordingly. First of all, these type of tests go in `integration` folder of your test harness or in the specific `module` folder if you are testing modules.



Here are the basics to follow for integration testing:

- Create one test CFC for each event handler you would like to test
- Test CFC inherits from `coldbox.system.testing.BaseTestCase`
- The test CFC can have some annotations that tell the testing framework to what ColdBox application to connect to and test
- For each action in your event handler, create a `testAction()` method
- Execution of the event is done via the `execute()` method, which returns a request context object
- Most verifications and assertions are done via the contents of the request context object (request collections)

The creation of your test case based off the inherited **BaseTestCase** class will give you the ability to test your applications without the need of a browser to trigger the application startup routines, configuration and finally execution of your event. A key point here is that your test CFC has the machinery to talk to **ANY** ColdBox application in the same server only. By default it connects to the `/` or root of your server to look for the application. We have several annotations you can add to the test CFC component tag that alters this behavior:

Test Annotations

Annotation	Type	Required	Default	Description
appMapping	string	false	/	The application mapping of the ColdBox application to test. By defaults it maps to the root. Extremely important this mapping is a slash notation that points to the root of the ColdBox application to test.
configMapping	string	false	{appMapping}/config/Coldbox.cfc	The configuration file to load for this test, which by convention uses the same configuration as the application uses. This is a dot notation path to a configuration CFC.
coldboxAppKey	string	false	<i>cbController</i>	The named key of the ColdBox controller that will be placed in application scope for you to simulate the ColdBox application. Used mostly on advanced testing cases where you have altered the default application key.

```
loadColdBox Boolean false true
```

If you call `super.init()` on the test case, this flag tells the base test case to load up the virtual testing application or not. This flag is mostly used for advanced testing cases, by default it always load ColdBox in Base Test Cases.

```
<cfcomponent extends="coldbox.system.testing.BaseTestCase" appMapping="/apps/MyApp">
</cfcomponent>
<cfcomponent extends="coldbox.system.testing.BaseTestCase" appMapping="/apps/MyApp" configMapping="apps.MyApp.test.resources.Config">
</cfcomponent>
```

Important : The `AppMapping` setting is the most important one. This is how your test connects to a location of a ColdBox application to test.

Common Methods

The inheritance gives you access not only to MXUnit's assertions but also some great ColdBox methods. Please see the [API Docs](#) for all the methods, or our nifty [Cheat Sheet](#).

Key	Description
<code>Sabort()</code> , <code>\$dump()</code> , <code>\$include()</code> , <code>\$rethrow()</code> , <code>\$throw()</code>	CF Facades
<code>announceInterception()</code>	Announce an interception state with an intercept data
<code>execute()</code>	Execute an event to do integration test with
<code>getAppMapping()</code>	Getter for the application's mapping
<code>getConfigMapping()</code>	Getter for the config.xml location
<code>getCacheBox()</code>	Get a reference to CacheBox
<code>getColdBoxOCM()</code>	Get a named cache provider
<code>getController()</code>	Getter for the ColdBox controller.
<code>getFlashScope()</code>	Getter for the application's flash scope, great for asserting
<code>getInterceptor()</code>	Get a loaded interceptor
<code>getLogBox()</code>	Get the application's loaded LogBox
<code>getModel()</code>	Retrieve a model object
<code>getRequestContext()</code>	Getter for the current request collection object (event)
<code>getWireBox()</code>	Get a reference to WireBox
<code>reset()</code>	Resets everything in application scope and internally in a test case. Great when things go wrong
<code>setup()</code>	The setup method where the coldbox controller and virtual application gets created and configured.
<code>teardown()</code>	The teardown method where the virtual coldbox controller gets destroyed.

Common Mocking Methods

Here are some methods that will help you when mocking:

Key	Description
<code>getMockBox()</code>	Get a reference to MockBox for mocking and stubbing
<code>getMockConfigBean()</code>	Get a config bean object with mocking capabilities
<code>getMockDatasource()</code>	Build a mock datasource object with your own mocked data and mocking capabilities
<code>getMockModel()</code>	Get a mock model object by convention
<code>getMockPlugin()</code>	Get a mock plugin object by convention
<code>getMockRequestContext()</code>	Get a mock request context object
<code>querySim()</code>	Simulate any query easily

Test Setup

Now that we know how to start, here is the code for a test. Please note that if you override the `setup()` method, you **must** call its parent. If not, the application will not load.

```
component extends="coldbox.system.testing.BaseTestCase" appMapping="/apps/MyApp"{
    function setup(){
        super.setup();
    }
}
```

The Handler To Test

Let's use a sample event handler so we can test it:

```
component{
    function index(event,rc,prc){
        rc.welcomeMessage = "Welcome to ColdBox!";
        event.setView("general/index");
    }

    function doSomething(event,rc,prc){
        setNextEvent("general.index");
    }

    function dspLogin(event,rc,prc){
        event.setView("general/dspLogin");
    }

    function doLogin(event,rc,prc){
        event.paramValue("username","");
        event.paramValue("password","");

        if( !len(rc.username) or !len(rc.password) ){
            flash.put("notice","Username and password must be filled out!");
            setNextEvent("general.dspLogin");
        }

        if( rc.username == "luis" and rc.password = "luis" ){
            setNextEvent("general.hello");
        }
        else{
            flash.put("notice","Invalid Credentials! Try Again!");
            setNextEvent("general.dspLogin");
        }
    }

    function hello(event,rc,prc){
        return "Howdy user!";
    }
}
```

I can test this entire handler without me building any views yet. I can even test the relocations that happen via `setNextEvent()`. ColdBox will wire itself up with some mocking classes to intercept those relocations for you and place those values in the request collection for you so you can assert them. It creates a key called `setnextevent` in the request collection and any arguments passed to the method are also saved as keys with the following pattern:

```
setnextevent_{argumentName}
```

Important: Any relocation produced by the framework via the `setNextEvent` method will produce some variables in the request collection for you to verify relocations.

The Integration Test

Now that we have seen the handler code, let's see the testing code as well. One important thing to note is that I use **URL** variables in my tests instead of **FORM** variables. I use **URL** because if the tests are executed via the MXUnit Eclipse Plugin they execute via SOAP and ColdFusion does not translate the **FORM** structure in SOAP web services. If you only test via the browser then you are ok, but if you use the Eclipse Plugin you must use **URL** scope instead in your tests. The great thing about ColdBox is that it abstracts **FORM** and **URL** scope into the request collections!

```
component extends="coldbox.system.testing.BaseTestCase" appmapping="/apps/MyApp"{
    function setup(){
        super.setup();
    }

    function testindex(){
        var event = execute("general.index");
        assertEquals("Welcome to ColdBox!", event.getValue("welcomeMessage"));
    }

    function testdoSomething(){
        var event = execute("general.doSomething");
        assertEquals("general.index", event.getValue("setnextevent"));
    }

    function testdspLogin(){
        var event = execute("general.dspLogin");
        var prc = event.getCollection(private=true);
        assertEquals("general.dspLogin", prc.currentView );
    }

    function testdoLoginWithNoData(){
        var event = execute("general.doLogin");
        assertTrue( getFlashScope().exists("notice") );
        assertEquals("general.dspLogin", event.getValue("setnextevent") );
    }

    function testDoLoginWithInvalidData(){
        URL.username = "joe";
        URL.password = "hacker";
        var event = execute("general.doLogin");
        assertTrue( getFlashScope().exists("notice") );
        assertEquals("general.dspLogin", event.getValue("setnextevent") );
    }

    function testdoLoginWithGoodData(){
        URL.username = "luis";
        URL.password = "luis";
        var event = execute("general.doLogin");
        assertFalse( getFlashScope().exists("notice") );
        assertEquals("general.hello", event.getValue("setnextevent") );
    }

    function testhello(){
        var event = execute("general.hello");
        assertEquals("Howdy user!", event.getValue("cbox_handler_results") );
    }
}
```

Handler Returning Results

If you have written event handlers that actually return data, then you will have to get the values from the request collection. This is done in order to assert returned results from handlers when most likely this handler is called from a coldbox proxy or just returning HTML. The following are the keys created for you in the request collection.

Key	Description
<code>cbox_handler_results</code>	The value returned from the event handler method. This key will NOT be created if the handler does not return any data.

Plugin Testing

You can test ColdBox plugins directly with no need of doing integration testing. This way you can unit test plugins directly very very easily with mocking integration built-in. All you need to do is the following:

1. Create a test class that inherits from `coldbox.system.testing.BasePluginTest`
2. Create a component annotation called `plugin` that equals the full path of the plugin to target for testing

This testing support class will create your plugin, and decorate with mocking capabilities via [MockBox](#) and mock all the necessary companion objects around plugins. The following are the objects that are placed in the `variables` scope for you to use:

- **plugin**: The target plugin to test
- **mockController**: A mock ColdBox controller in use by the target plugin
- **mockRequestService**: A mock request service object
- **mockLogger**: A mock logger class in use by the target plugin
- **mockLogBox**: A mock LogBox class in use by the target plugin
- **mockFlash**: A mock flash scope in use by the target plugin

All of the mock objects are essentially the dependencies of plugin objects. You have complete control over them as they are already mocked for you. We actually use this approach to test all shipped ColdBox plugins.

Important : We do not initialize your plugins for you. This is your job as you might need some mocking first.

Basic Setup

```
component extends="coldbox.system.testing.BasePluginTest" plugin="myapp.plugins.CoolPlugin"{
    // Just create test methods, no need to use the setup() method unless you want to:
    function setup(){
        super.setup();
        // test custom constructor
        plugin.init();
    }
}
```

Real Sample

```
<cfcomponent extends="coldbox.system.testing.BasePluginTest" plugin="coldbox.system.plugins.HTMLHelper">
</cfscript>
function testaddAssetJS(){
    var mockEvent = getMockRequestContext();
    mockRequestService.$("getContext", mockEvent);

    // mock the plugin's htmlhead method
    plugin.$("${htmlhead}");

    // Call method to test
    plugin.addAsset('test.js,luis.js');

    debug( plugin.$callLog().$htmlhead );

    // test duplicate call
    assertEquals('<script src="test.js" type="text/javascript"></script><script src="luis.js" type="text/javascript"></script>', plugin.$callLog().$htmlhead[1][1] );
    plugin.addAsset('test.js');
    assertEquals(1, arrayLen(plugin.$callLog().$htmlhead) );
}

function testTableORM(){
    data = entityLoad("User");

    str = plugin.table(data=data,includes="firstName");
    debug(str);
    assertEquals('<table><thead><tr><th>firstName</th></tr></thead><tbody><tr><td>Joe</td></tr><tr><td>Luis</td></tr></tbody></table>',str);
}
</cfscript>
</cfcomponent>
```

Handler Isolation Testing

You can test handlers directly with no need of doing integration testing. This way you can unit test handlers directly very very easily. All you need to do is the following:

1. Create a test class that inherits from `coldbox.system.testing.BaseHandlerTest`
2. Create a component annotation called `handler` that equals the full path of the handler CFC to target for unit testing
3. Create an optional annotation called `UDFLibraryFile` that will be the path of the UDF library file that is declared in your ColdBox configuration or any other UDF file you load dynamically in your handlers.

This testing support class will create your handler, and decorate with mocking capabilities via [MockBox](#) and mock all the necessary companion objects around handlers. The following are the objects that are placed in the `variables` scope for you to use:

- `handler`: The target handler to test
- `mockController`: A mock ColdBox controller in use by the target handler
- `mockRequestService`: A mock request service object
- `mockLogger`: A mock logger class in use by the target handler
- `mockLogBox`: A mock LogBox class in use by the target handler
- `mockFlash`: A mock flash scope in use by the target handler

All of the mock objects are essentially the dependencies of handler objects. You have complete control over them as they are already mocked for you.

Important: We do not initialize your handlers for you. So if you have a `init()` method, you need to call it manually. Also note that this CFC is in isolation, you will have to mock all of its dependencies if needed.

Basic Setup

```
/**
 * @handler myApp.handler.User
 * @UDFLibraryFile /myApp/includes/helpers/AppHelper.cfm
 */
component extends="coldbox.system.testing.BaseHandlerTest" {
    // Just create test methods, no need to use the setup() method unless you want to:
    function setup() {
        super.setup();
        mockUserService = getMockBox().createEmptyMock("myapp.model.user.UserService");
        // wire in my mock dependencies as this handler already has mocking capabilities
        handler.$property("userService", "variables", mockUserService);
    }
}
```

Interceptor Testing

You can now test interceptors directly with no need of doing integration testing. This way you can unit test interceptors directly very very easily. All you need to do is the following:

1. Create a test class that inherits from `coldbox.system.testing.BaseInterceptorTest`
2. Create a component annotation called `interceptor` that equals the full path of the handler CFC to target for unit testing
3. Create an optional structure variable in the `variables` scope called `configProperties` in the `setup()` method before your `super.setup()` method, if you would like to test the interceptor with your configuration properties structure.

This testing support class will create your interceptor, and decorate with mocking capabilities via [MockBox](#) and mock all the necessary companion objects around interceptors. The following are the objects that are placed in the `variables` scope for you to use:

- `interceptor`: The target interceptor to test
- `mockController`: A mock ColdBox controller in use by the target interceptor
- `mockRequestService`: A mock request service object
- `mockLogger`: A mock logger class in use by the target interceptor
- `mockLogBox`: A mock LogBox class in use by the target interceptor
- `mockFlash`: A mock flash scope in use by the target interceptor

All of the mock objects are essentially the dependencies of interceptor objects. You have complete control over them as they are already mocked for you.

Basic Setup

```
component extends="coldbox.system.testing.BaseInterceptorTest" interceptor="myApp.interceptors.Security" {
    // Just create test methods, no need to use the setup() method unless you want to:
    function setup() {
        configProperties = {
            roles = "admin,user,moderator",
            security = "active"
        };
        super.setup();
        mockSecurityService = getMockBox().createEmptyMock("myapp.model.SecurityService");
        // wire in my mock dependencies as this interceptor already has mocking capabilities
        interceptor.$property("securityService", "variables", mockSecurityService);
    }
    // we are now ready to test this interceptor
}
```

Real Example

```
<cfcomponent extends="coldbox.system.testing.BaseInterceptorTest" interceptor="coldbox.system.interceptors.Deploy">
</cfscript>

function setup() {
    super.setup();

    // mocks
    mockController.$("getAppRootPath", expandPath("../coldbox/testharness"));
    interceptor.setProperty("tagFile", "config/.deploy_tag");
    interceptor.$("locateFilePath", "config/.deploy_tag").$("setSetting");
}

function testConfigure() {
    interceptor.configure();
}

function testAfterAspectsLoad() {
    mockLogger.$("info");
    interceptor.afterAspectsLoad(getMockRequestContext());
}

function testPostProcess() {
    //mocks
    mockController.$("getColdboxInitiated", true).$("setColdboxInitiated").$("setAspectsInitiated");
    testDate = now();
    mockLogger.$("info").$("error");
    interceptor.$property("tagFilePath", "instance", "config/.deploy_tag");
    interceptor.$property("deployCommandObject", "instance", '');

    // Test no setting
    interceptor.$("settingExists", false).$("configure");

    interceptor.postProcess(getMockRequestContext());
    assertEquals( 1, arrayLen(interceptor.$callLog().configure) );

    // Test setting exists but same date
    interceptor.$("getSetting", testDate).$("fileLastModified", testDate).$("settingExists", true);
    interceptor.postProcess(getMockRequestContext());
    assertEquals( 0, arrayLen(mockController.$callLog().setColdboxInitiated) );

    // Test it works
    interceptor.$("getSetting", testDate).$("fileLastModified", testDate+10).$("settingExists", true);
    interceptor.postProcess(getMockRequestContext());
    assertEquals( 1, arrayLen(mockController.$callLog().setColdboxInitiated) );
}
</cfscript>
</cfcomponent>
```

Model Object Testing

You can now test model objects directly with no need of doing integration testing. This way you can unit test model objects very very easily using great mocking capabilities. All you need to do is the following:

1. Create a test class that inherits from `coldbox.system.testing.BaseModelTest`
2. Create a component annotation called `model` that equals the full path of the model object CFC to target for unit testing

This testing support class will create your model object, and decorate with mocking capabilities via [MockBox](#) and create some mocking classes you might find useful in your model object unit testing. The

following are the objects that are placed in the *variables* scope for you to use:

- **model**: The target model object to test
- **mockLogger**: A mock logger class
- **mockLogBox**: A mock LogBox class

Important : We do not initialize your model objects for you. This is your job as you might need some mocking first.

Basic Setup

```
component extends="coldbox.system.testing.BaseModelTest" model="myApp.model.User"{
    function setup(){
        super.setup();
        user = model;
    }

    function testIsActive(){
        assertEquals( false, user.isActive() );
    }
}
```

Tips & Tricks

Here are some useful tips for you when doing testing with ColdBox Applications:

- If you are using relative paths, this might be a problem as the running application is different from the test application. Try to always use paths based on the application's **AppMapping**
- Always use **setNextEvent** for relocations so they can be mocked
- Use **URL** variables instead of **FORM** variables if you will be testing from within Eclipse
- Leverage **querySim()** for query mocking
- Leverage **MockBox** for mocking and stubbing
- If you use **ColdBox Security** or other security mechanisms in your application, you will have to mock a login for integration tests
- Integration tests are **NOT** the same as handler tests. Handler tests will just test the handler CFC in isolation, so it will be your job to mock everything around it.
- You can use the **BaseModelTest** to test any domain object
- The ColdBox source **testing** folder has over 5000 tests, mocking scripts, etc. where you can learn from, so check it out -> <https://github.com/ColdBox/coldbox-platform/tree/master/testing>.

Mocking HTTP Methods

There might be a case when you are building RESTful applications that you need to test how your handlers respond via multiple HTTP verbs. If you use the MXUnit Eclipse Plugin, this uses only a **POST** approach, so even your simple **GET** operations could fail. So here is a little snippet of how to mock the HTTP verb before an execution in a test case:

```
function testList(){
    // mock HTTP verb to GET
    getMockBox().prepareMock( getRequestContext() ).$("getHTTPMethod","GET");
    // execute now as a GET
    var event = execute("users.list");
}
```

Conclusion

As you can see, ColdBox makes it really easy for you to test your handlers and your application. So no more excuses, get in to the best practice of unit testing and now unit test your application code. Enjoy.