TestBox - xUnit Primer

www.coldbox.org

Covers up to version 1.0.0 Alpha

Contents

TestBox is a xUnit style and behavior-driven development (BDD) testing framework for CFML (ColdFusion). It does not depend on any other framework for operation and it has a clean and descriptive syntax that will make you giggle when writing your tests. It ships with not only the xUnit and BDD testing capabilities, but also an assertions and expectations library, several different runners, reporters and MockBox, for mocking and stubbing capabilities.

NOTE

This article is a primer to get you started with the xUnit capabilities of TestBox, if you would like to explore the BDD style capabilities of TestBox we recommend you visit our BDD primer here

`http://wiki.coldbox.org/wiki/TestBox-BDD-Primer.cfm`

Download

TestBox (Includes API Docs) - `http://www.coldbox.org/download`

API Docs - `http://www.coldbox.org/api`

Requirements

The xUnit functionalities of TestBox will require Railo 4.1+ or ColdFusion 9.01+.

INSTALLING TESTBOX

TestBox can be downloaded as a separate framework or it is included with the latest Coldbox Platform release. The main difference between both versions is the instantiation and usage namespace, the rest is the same.

Standalone

You can place the TestBox package in your webroot or create a mapping called `/testbox` that points to it.

```
testbox.system.testing
```

ColdBox

You can just follow the normal Installation procedures and you will use the following instantiation namespace:

```
coldbox.system.testing
```

Important : Please note that some reporters require UI assets, therefore the `/testbox` or `/coldbox` installation must be web-accessible.

BUNDLES: GROUP YOUR TESTS

TestBox relies on the fact of creating testing bundles which are basically CFCs. A bundle CFC will hold all the tests the TestBox runner will execute and produce reports on. Thus, sometimes this test bundle is referred to as a test suite in xUnit terms.

```
component displayName="My test suite" {
    // executes before all tests
    function beforeTests() {}

    // executes after all tests
    function afterTests() {}
}
```

As you can see, it is a simple CFC, no inheritance required. It can contain 2 life-cycle methods, the `beforeTests()` and `afterTests()` methods will execute once before ALL your tests run and then after ALL your tests run. This is a great way to do any kind of global setup or teardown in your tests. It also contains a `displayName` argument in the component declaration which gives you a way to name your testing suite. We will see later the other annotations you can add to the component declaration.

TEST METHODS

TestBox will try to test any public method which contains the word test in it. It will NOT execute any private methods or methods that do not include the word test in them. Remember that, that is our testing convention. Each test methods will test the state of the SUT (software under test) or sometimes referred to as code under test. It will do so by asserting that actual values from an execution match an expected value or condition. TestBox offers an assertion library that you have available in your bundle via the injected variable $assert. You can also use our expectations library if you so desire, but that is mostly used in our BDD approach.

```
function testIncludes() {
       $assert.includes( "hello", "HE" );
       $assert.includes( [ "Monday", "Tuesday" ] , "monday" );
}
```

Each test function can also have some cool annotations attached to it.

labels - The list of labels this test belongs to

skip - A boolean flag that makes the runners skip the test for execution. It can also be the name of a UDF in the same CFC that will be executed and MUST return a boolean value.

An expectation or an assertion is a nice DSL that TestBox exposes so you can pretty much read what should happen in the testing scenario. A test will pass if **all** assertions pass. A spec with one or more assertions that fail will fail the entire test.

ASSERTIONS

Assertions are self-concatenated strings that evaluate an actual value to an expected value or condition. These are initiated by the global TestBox variable called `$assert` which contains tons of included assertion methods so you can evaluate your tests.

Evaluators

Each assertion evaluator will compare the actual value and an expected value or condition. It is responsible for either passing or failing this evaluation and reporting it to TestBox. Each evaluator also has a negative counterpart assertion by just prefixing the call to the method with a **not** expression.

```
function testIncludes() {
    $assert.includes( "hello", "HE" );
    $assert.includes( [ "Monday", "Tuesday" ] , "monday" );
}

function testNotIncludes() {
    $assert.notIncludes( "hello", "what" );
    $assert.notIncludes( [ "Monday", "Tuesday" ] , "Friday" );
}
```

Included Evaluators

TestBox has a plethora (That's Right! I said Plethora) of evaluators that are included in the release. The best way to see all the latest evaluator methods is to visit our API and digest the coldbox.system.testing.Assertion class. There is also the ability to register and write custom assertion evaluators in TestBox via our `addAssertions()` function, so we recommend you visit the TestBox documentation for full disclosure on these features.

LIFE CYCLE Methods

```
function beforeTests() {
    application.salvador = 1;
}
function afterTests() {
    structClear( application );
}
function setup() {
    request.foo = 1;
}
function teardown() {
    structClear( request );
}
```

TEST Methods

```
function testIncludes() {
    $assert.includes( "hello", "HE" );
    $assert.includes( [ "Monday", "Tuesday" ] , "monday" );
}
function testIncludesWithCase() {
    $assert.includesWithCase( "hello", "he" );
    $assert.includesWithCase( [ "Monday", "Tuesday" ] ,
"Monday" );
}
function testnotIncludesWithCase() {
    $assert.notincludesWithCase( "hello", "aa" );
    $assert.notincludesWithCase( [ "Monday", "Tuesday" ] ,
                                 "monday" );
}
function testNotIncludes() {
    $assert.notIncludes( "hello", "what" );
    $assert.notIncludes( [ "Monday", "Tuesday" ] , "Friday" );
}
function testIsEmpty() {
    $assert.isEmpty( [] );
    $assert.isEmpty( {} );
    $assert.isEmpty( "" );
    $assert.isEmpty( queryNew("") );
}
function testIsNotEmpty() {
    $assert.isNotEmpty( [1,2] );
    $assert.isNotEmpty( {name="luis"} );
    $assert.isNotEmpty( "HelloLuis" );
    $assert.isNotEmpty( querySim( "id, name 1 | luis" ) );
}
function testSkipped() skip {
    $assert.fail( "This Test should fail" );
}
```

There are many more Test Methods available, to many to list here. You can find the full list here

http://wiki.coldbox.org/wiki/TestBox-Unit-Primer.cfm#Included_Evaluators

NON-RUNNABLE Methods

```
function nonStandardNamesWillNotRun() {
    fail( "Non-test methods should not run" );
}
private function privateMethodsDontRun() {
    fail( "Private method don't run" );
}
```

SETUP AND TEARDOWN

TestBox not only provides you with global life-cycle methods but also with localized test methods. This is a great way to keep your tests DRY (Do not repeat yourself)! TestBox provides the `setup()` and `teardown()` methods. which as their names indicate, they execute before a test and after a test in a test bundle.

```
component displayName="TestBox xUnit suite" labels="railo,cf"
{
    function setup() {
        application.wirebox =
            new coldbox.system.ioc.Injector();
        structClear( request );
    }
    function teardown() {
        structDelete( application, "wirebox" );
        structClear( request );
    }
    function testThrows() {
        $assert.throws(function() {
            var hello = application.wirebox.getInstance(
                        "myINvalidService" ).run();
        });
    }
    function testNotThrows() {
        $assert.notThrows(function() {
            var hello = application.wirebox.getInstance(
                        "MyValidService" ).run();
        });
    }
}
```

TEST AND SUITE LABELS

Tests and suites can be tagged with TestBox labels. Labels allows you to further categorize different tests or suites so that when a runner executes with labels attached, only those tests and suites will be executed, the rest will be skipped. Labels can be applied globally to the component declaration of the test bundle suite or granularly at the test method declaration.

```
component displayName="TestBox xUnit suite"
          labels="railo,stg,dev" {
    function setup() {
        application.wirebox = new coldbox.system
                                  .ioc.Injector();
        structClear( request );
    }
    function teardown() {
        structDelete( application, "wirebox" );
        structClear( request );
    }
    function testThrows() {
        $assert.throws(function() {
            var hello = application.wirebox.getInstance(
                        "myINvalidService" ).run();
        });
    }
    function testNotThrows() {
        $assert.notThrows(function() {
            var hello = application.wirebox.getInstance(
                        "MyValidService" ).run();;
        });
    }
    function testFailsShortcut() labels="dev" {
        fail( "This Test should fail when executed
                  with labels" );
    }
}
```

SKIPPING TESTS AND SUITES

Tests and suites can be skipped from execution by using the skip annotation in the component or function declaration. The reporters will show that these suites or tests where skipped from execution. The value of the skip annotation can be a simple true or false or it can be the name of a UDF that exists in the same bundle CFC. This UDF must return a boolean value and it is evaluated at runtime.

```
component displayName="TestBox xUnit suite" skip="testEnv" {
    function setup() {
        application.wirebox = new coldbox.system
                                  .ioc.Injector();
        structClear( request );
    }
    function teardown() {
        structDelete( application, "wirebox" );
        structClear( request );
    }
    function testThrows() skip="true" {
        $assert.throws(function() {
            var hello = application.wirebox.getInstance(
                        "myINvalidService" ).run();
        });
    }
    function testNotThrows() {
        $assert.notThrows(function() {
            var hello = application.wirebox.getInstance(
                        "MyValidService" ).run();
        });
    }
    private boolean function testEnv() {
        return ( structKeyExists( request, "env") && request.
                  env == "stg" ? true : false );
    }
}
```

SPIES & MOCKING

Please refer to our MockBox guide to take advantage of all the mocking and stubbing you can do. However, every test bundle CFC has the following functions available to you for mocking and stubbing purposes:

`makePublic( target, method, newName )` - Exposes private methods from objects as public methods

`querySim( queryData )` - Simulate a query

`getMockBox( [generationPath] )` - Get a reference to MockBox

`createEmptyMock( [className], [object], [callLogging=true])` - Create an empty mock from a class or object

`createMock( [className], [object], [clearMethods=false], [callLogging=true])` - Create a spy from an instance or class with call logging

`prepareMock( object, [callLogging=true])` - Prepare an instance of an object for method spies with call logging

`createStub( [callLogging=true], [extends], [implements])` - Create stub objects with call logging and optional inheritance trees and implementation methods

ASYNCHRONOUS TESTING

You can tag a bundle component declaration with the boolean `asyncAll` annotation and TestBox will execute all specs in separate threads for you concurrently.

```
component displayName="TestBox xUnit suite" skip="testEnv"
          asyncAll=true {
    function setup() {
        application.wirebox = new coldbox.system
                                    .ioc.Injector();
        structClear( request );
    }
    function teardown() {
        structDelete( application, "wirebox" );
        structClear( request );
    }
    function testThrows() skip="true" {
        $assert.throws(function() {
            var hello = application.wirebox.getInstance(
                        "myINvalidService" ).run();
        });
    }
    function testNotThrows() {
        $assert.notThrows(function() {
            var hello = application.wirebox.getInstance(
                        "MyValidService" ).run();
        });
    }
    private boolean function testEnv() {
        return ( structKeyExists( request, "env") && request.
                  env == "stg" ? true : false );
    }
}
```

RUNNERS AND REPORTERS

TestBox ships with several test runners internally but we have tried to simplify and abstract it with our TestBox object which can be found in the coldbox.system.testing package. The TestBox object allows you to execute tests from a CFC, CFM, HTTP, SOAP or REST. The main execution methods are:

```
// Run tests and produce reporter results
run()
// Run tests and get raw TestResults object
runRaw()
// Run tests and produce reporter results from SOAP, REST, HTTP
runRemote()
```

We encourage you to read the API docs included in the distribution for the complete parameters for each method.

We encourage you to read the API docs included in the distribution for the complete parameters for each method.

Here are the arguments you can use for initializing TestBox or executing the `run()` method

`bundles` - The path, list of paths or array of paths of the spec bundle CFCs to run and test

`directory` - The directory information struct to test: [mapping = the path to the directory using dot notation (`myapp.testing.specs`), recurse = boolean, filter = closure that receives the path of the CFC found, it must return true to process or false to continue process]

`reporter` - The type of reporter to use for the results, by default is uses our 'simple' report. You can pass in a core reporter string type or an instance of a `coldbox.system.testing.reports.IReporter`. You can also pass a struct with `[type="string or classpath", options={}]` if a reporter expects options.

`labels` - The string or array of labels the runner will use to execute suites and specs with.

`options` - A structure of property name-value pairs that each runner can implement and use at its discretion.

`testSuites` - A list or array of suite names that are the ones that will be executed ONLY!

`testSpecs` - A list or array of test names that are the ones that will be executed ONLY

Here are the arguments you can use for executing the `runRemote()` method of the TestBox object:

`bundles` - The path, list of paths or array of paths of the spec bundle CFCs to run and test

`directory` - The directory mapping to test: directory = the path to the directory using dot notation (`myapp.testing.specs`)

`recurse` - Recurse the directory mapping or not, by default it does

`reporter` - The type of reporter to use for the results, by default is uses our 'simple' report. You can pass in a core reporter string type or a class path to the reporter to use.

`reporterOptions` - A JSON struct literal of options to pass into the reporter

`labels` - The string array of labels the runner will use to execute suites and specs with.

`options` - A JSON struct literal of configuration options that are optionally used to configure a runner.

`testSuites` - A list of suite names that are the ones that will be executed ONLY!

`testSpecs` - A list of test names that are the ones that will be executed ONLY

The `bundles` argument which can be a single CFC path or an array of CFC paths or a `directory` argument so it can go and discover the test bundles from that directory. The `reporter` argument can be a core reporter name like: json, xml, junit, raw, simple, dots, tap, min, etc. or it can be an instance of a reporter CFC. You can execute the runners from any cfm template or any CFC or any URL, that is up to you.

Bundle(s) Runner

```
<cfset r = new coldbox.system.testing.TestBox(
        "coldbox.testing.cases.testing.specs.BDDTest" ) >
<cfoutput>#r.run()#</cfoutput>

<cfset r = new coldbox.system.testing.TestBox(
      [ "coldbox.testing.cases.testing.specs.BDDTest",
        "coldbox.testing.cases.testing.specs.BDD2Test" ] ) >
<cfoutput>#r.run()#</cfoutput>

<cfset r = new coldbox.system.testing.TestBox(
      bundles="coldbox.testing.cases.testing.specs.BDDTest",
      labels="railo" ) >
<cfoutput>#r.run(reporter="json")#</cfoutput>
```

Directory Runner

```
<cfset r = new coldbox.system.testing.TestBox(
          directory="coldbox.testing.cases.testing.specs" ) >
<cfoutput>#r.run()#</cfoutput>

<cfset r = new coldbox.system.testing.TestBox(
          directory={
              mapping="coldbox.testing.cases.testing.specs",
              recurse=false }) >
<cfoutput>#r.run()#</cfoutput>

<cfset r = new coldbox.system.testing.TestBox(
          directory={
              mapping="coldbox.testing.cases.testing.specs",
              recurse=true,
              filter=function(path) {
                  return ( findNoCase( "test", arguments.path )
                      ? true : false ); }}) >
<cfoutput>#r.run()#</cfoutput>

<cfset r = new coldbox.system.testing.TestBox(
          directory={
              mapping="coldbox.testing.cases.testing.specs",
              recurse=true,
              filter=function(path) {
                  return ( findNoCase( "test", arguments.path )
                      ? true : false ); }}) >
<cfset fileWrite( 'testreports.json', r.run() )>
```

Reporters

TestBox comes also with a nice plethora of reporters:

Console : Sends report to console

Dot : Builds an awesome dot report

JSON : Builds a report into JSON

JUnit : Builds a JUnit compliant report

Raw : Returns the raw structure representation of the testing results

Simple : A basic HTML reporter

Text : Back to the 80's with an awesome text report

XML : Builds yet another XML testing report

Tap : A test anything protocol reporter

Min : A minimalistic view of your test reports

TestBox is completely documented in our online wiki located at http://wiki.coldbox.org/wiki/TestBox.cfm

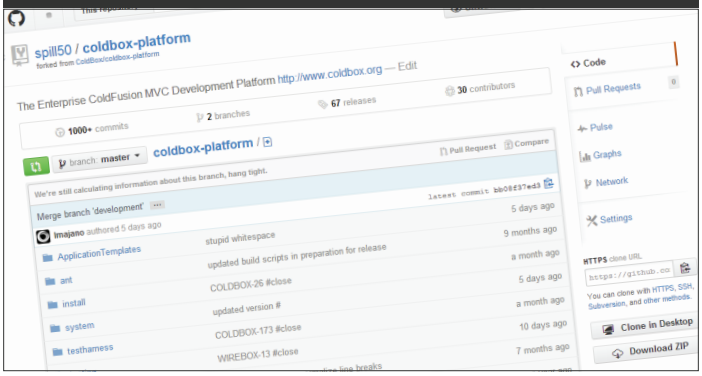
For API docs to see class definitions and method signatures, please visit the API docs located at http://www.coldbox.org/api

We have an active Google Group with hundreds of subscriber located at

http://groups.google.com/group/coldbox

Our official code repository is on GitHub. Please favorite us and feel free to fork and submit pull requests.

https://github.com/ColdBox/coldbox-platform



TestBox - xUnit Primer

Ortus Solutions
A Software Revolution

CacheBox is professional open source backed by Ortus Solutions, who provides training, support & mentoring, and custom development.

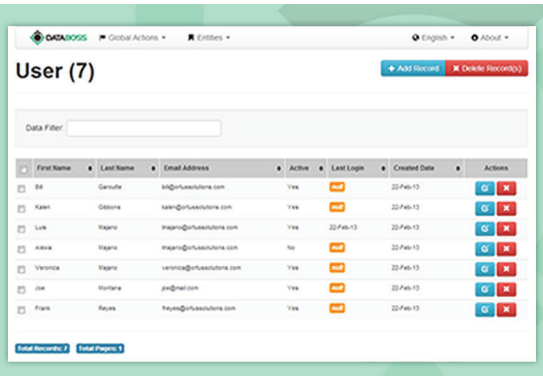
Support Program

The Ortus Support Program offers you a variety of Support Plans so that you can choose the one that best suit your needs. Subscribe to your plan now and join the Ortus Family!

With all of our plans you will profit not only from discounted rates but also receive an entire suite of support and development services. We can assist you with sanity checks, code analysis, architectural reviews, mentoring, professional support for all of our Ortus products, custom development and much more!

For more information visit
www.ortussolutions.com/services/support

Our Plans	M1	Entry	Standard	Premium	Enterprise
Price	**$199**	**$2099**	**$5699**	**$14099**	**$24599**
Support Hours	2 4 tickets	10 20 tickets	30 60 tickets	80 160 tickets	150 300 tickets
Discounted Rate	**$185/hr**	**$180/hr**	**$175/hr**	**$170/hr**	**$160/hr**
Renewal Price	**$199/month**	**$1800/year**	**$5250/year**	**$13600/year**	**$2400/year**
Phone/Online Appointments	✔	✔	✔	✔	✔
Web Ticketing System	✔	✔	✔	✔	✔
Architectural Reviews	✔	✔	✔	✔	✔
Hour Rollover		✔	✔	✔	✔
Custom Development		✔	✔	✔	✔
Custom Builds & Patches			✔	✔	✔
Priority Training Registration			✔	✔	✔
Development & Ticket Priority			✔	✔	✔
Response Times	1-5 B.D.	1-3 B.D.	1-2 B.D.	< 24 hr	< 12 hr
Books, Trainings, Product Discounts	0%	5%	10%	15%	20%
Free Books	0	0	1	3	5



"Build dynamic ORM administrators in seconds"

DATABOSS
Dynamic Administrator

www.data-boss.com

www.coldbox.org | www.ortussolutions.com

First Edition - November 2013