TestBox - BDD Primer

www.coldbox.org

Covers up to version 1.0.0

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 BDD capabilities of TestBox, if you would like to explore the xUnit style capabilities of TestBox we recommend you visit our xUnit primer here

`http://wiki.coldbox.org/wiki/TestBox-unit-primer.cfm`

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.

Download

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

Requirements

The BDD functionalities of TestBox will require Railo 4.1+ or ColdFusion 10+.

BUNDLES: GROUP YOUR TESTS

TestBox relies on the fact of creating testing bundles which are basically CFCs. A bundle CFC will hold all the suites and specs a TestBox runner will execute and produce reports on.

```
component {
    // executes before all suites
    function beforeAll() {}

    // executes after all suites
    function afterAll() {}

    // All suites go in here
    function run() {}
}
```

As you can see, it is a simple CFC, no inheritance required. It can contain 2 life-cycle functions and a single `run()` function where you will be writing your test suites. The `beforeAll()` and `afterAll()` methods are called life-cycle methods. They will execute once before the `run()` function and once after the `run()` function. This is a great way to do any kind of global setup or teardown in your tests.

SUITES: DESCRIBE YOUR TESTS

A test suite begins with a call to our TestBox `describe()` function with at least two arguments: a **title** and a **function**. The title is the name of the suite to register and the function is the block of code that implements the suite. There are more arguments which you can see below:

title - The title of the suite to register

body - The closure that represents the test suite

labels - The list or array of labels this suite group belongs to

asyncAll - If you want to parallelize the execution of the defined specs in this suite group

skip - A flag or a closure that tells TestBox to skip this suite group from testing if true. If this is a closure it must return boolean.

```
function run() {
    describe("A suite", function() {
        it("contains spec with an awesome expectation",
           function() {
            expect( true ).toBeTrue();
        });
    });
}
```

SPECS

Specs are defined by calling the TestBox `it()` global function, which takes in a title and a function. The title is the title of this spec or test you will write and the function is a block of code that represents the test/spec. A spec will contain most likely one or more expectations that will test the state of the SUT (software under test) or sometimes referred to as code under test.

title - The title of the spec

body - The closure that represents the spec

labels - The list or array of labels this suite group belongs to

skip - A flag or a closure that tells TestBox to skip this suite group from testing if true. If this is a closure it must return boolean.

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

```
function run() {
    describe("A suite", function() {
        it("contains spec with an awesome expectation",
           function() {
            expect( true ).toBeTrue();
        });
        it("contains a spec with more than 1 expectation",
           function() {
            expect( [1,2,3] ).toBeArray();
            expect( [1,2,3] ).toHaveLength( 3 );
        });
    });
}
```

They are closures Ma!

Since the implementations of the `describe()` and `it()` functions are closures, they can contain executable code that is necessary to implement the test. All ColdFusion rules of scoping apply to closures, so please remember them. We recommend always using the variables scope for easy access and distinction.

```
function run() {
    describe("A suite is a closure", function() {
        c = new Calculator();
        it("and so is a spec", function() {
            expect( c ).toBeTypeOf( 'component' );
        });
    });
}
```

EXPECTATIONS

Expectations are self-concatenated strings that evaluate an actual value to an expected value or condition. These are initiated by the global TestBox method called `expect()` which takes in a value called the actual value. It is concatenated in our expectations DSL with a matcher function that will most likely take in an expected value or condition to test.

Matchers

Each matcher implements a comparison or evaluation of 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 matcher also has a negative counterpart assertion by just prefixing the call to the matcher with a **not** expression.

```
function run() {
    describe("The 'toBe' matcher evaluates equality",
             function() {
        it("and has a positive case", function() {
            expect( true ).toBe( true );
        });
        it("and has a negative case", function() {
            expect( false ).notToBe( true );
        });
    });
}
```

Included Matchers

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

```
describe("Some Included TestBox Matchers:", function() {
    describe("The 'toBe' matcher", function() {
        it("works for simple values", function() {
            coldbox = 1;
            expect( coldbox ).toBe( 1 );
```

Cont.

Included Matchers Cont.

```
            expect( coldbox ).notToBe( 5 );
        });
        it("works for complex values too
(arrays,structs,queries,objects)", function() {
            expect( [1,2] ).toBe( [1,2] );
            expect( { name="luis", awesome=true} ).toBe( {
awesome=true, name="luis" } );
            expect( this ).toBe( this );
            expect( queryNew("") ).toBe( queryNew("") );
        });
    });
    it("The 'toBeWithCase' matches strings with case
        equality", function() {
        expect( 'hello' ).toBeWithCase( 'hello' );
        expect( 'hello' ).notToBeWithCase( 'HELLO' );
    });
    it("The 'toBeTrue' and 'toBeFalse' matchers are used for
        boolean operations", function() {
        coldbox_rocks = true;
        expect( coldbox_rocks ).toBeTrue();
        expect( !coldbox_rocks ).toBeFalse();
    });
    it("The 'toBeNull' expects null values", function() {
        foo = "bar";
        expect( javaCast("null", "") ).toBeNull();
        expect( foo ).notToBeNull();
    });
});
```

There are many more included Matchers available, to many to list here. You can find the full list here

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

SUITE GROUPS

As we have seen before, the `describe()` function describes a test suite of related specs in a test bundle CFC. The title of the suite is concatenated with the title of a spec to create a full spec's name which is very descriptive. If you name them well, they will read out as full sentences as defined by **BDD** style.

```
describe("A spec", function() {
    it("is just a closure, so it can contain any code",
        function() {
        coldbox = 0;
        coldbox++;
        expect( coldbox ).toBe( 1 );
    });
    it("can have more than one expectation", function() {
        coldbox = 0;
        coldbox++;
        expect( coldbox ).toBe( 1 );
        expect( coldbox ).toBeTrue();
    });
});
```

Setup and Teardown

If you are familiar with xUnit style frameworks, the majority of them provide a way to execute functions before and after every single test case or spec in BDD. This is a great way to keep your tests DRY (Do not repeat yourself)! TestBox provides the `beforeEach()` and `afterEach()` methods that each take in a closure as their argument. As their names indicate, they execute before a spec and after a spec in a related describe block.

```
describe("A spec (with setup and tear-down)", function() {
    beforeEach(function( currentSpec ) {
        coldbox = 22;
        application.wirebox = new coldbox.system
                                        .ioc.Injector();
    });
    afterEach(function( currentSpec ) {
        coldbox = 0;
        structDelete( application, "wirebox" );
    });
    it("is just a function, so it can contain any code",
        function() {
        expect( coldbox ).toBe( 22 );
    });
    it("can have more than one expectation and talk to
       scopes", function() {
        expect( coldbox ).toBe( 22 );
        expect( application.wirebox.getInstance( 'MyService' ) )
                                    .toBeComponent();
    });
});
```

Nesting describe Blocks

Calls to our `describe()` function can be nested with specs at any level or point of execution. This allows you to create your tests as a related tree of nested functions. Please note that before a spec is executed, TestBox walks down the tree executing each `beforeEach()` and `afterEach()` function in the declared order. This is a great way to logically group specs in any level as you see fit.

```
describe("A spec", function() {
    beforeEach(function( currentSpec ) {
        coldbox = 22;
        application.wirebox = new coldbox.system
                                        .ioc.Injector();
    });
    afterEach(function( currentSpec ) {
        coldbox = 0;
        structDelete( application, "wirebox" );
    });
    it("is just a function, so it can contain any code",
        function() {
        expect( coldbox ).toBe( 22 );
    });
    it("can have more than one expectation and talk to
       scopes", function() {
        expect( coldbox ).toBe( 22 );
        expect( application.wirebox.getInstance( 'MyService' ) )
                                    .toBeComponent();
    });
```

Cont.

Included Matchers Cont.

```
        expect( application.wirebox.getInstance( 'MyService' ) )
                                    .toBeComponent();
    });
    describe("nested inside a second describe", function() {
        beforeEach(function( currentSpec ){
            awesome = 22;
        });
        afterEach(function( currentSpec ){
            awesome = 22 + 8;
        });
        it("can reference both scopes as needed ",
           function() {
            expect( coldbox ).toBe( awesome );
        });
    });
    it("can be declared after nested suites and have access to
       nested variables", function() {
        expect( awesome ).toBe( 30 );
    });
});
```

SPECS AND SUITE LABELS

Specs and suites can be tagged with TestBox labels. Labels allows you to further categorize different specs or suites so that when a runner executes with labels attached, only those specs and suites will be executed, the rest will be skipped.

```
describe(title="A spec", labels="stg,railo",
         body=function() {
    it("executes if its in staging or in railo", function() {
        coldbox = 0;
        coldbox++;
        expect( coldbox ).toBe( 1 );
    });
});
describe("A spec", function() {
    it("is just a closure, so it can contain any code",
       function() {
        coldbox = 0;
        coldbox++;
        expect( coldbox ).toBe( 1 );
    });
it(title="can have more than one expectation and labels",
   labels="dev,stg,qa,shopping", body=function() {
        coldbox = 0;
        coldbox++;
        expect( coldbox ).toBe( 1 );
        expect( coldbox ).toBeTrue();
    });
});
```

SKIPPING SPECS AND SUITES

Specs and suites can be skipped from execution by using the `xdescribe()` and `xit()` functions or by using the **skip** argument in each of them. The reporters will show that these suites or specs where skipped from execution.

```
xdescribe("A spec", function() {
    it("was just skipped, so I will never execute",
       function() {
        coldbox = 0;
        coldbox++;
        expect( coldbox ).toBe( 1 );
    });
});
describe("A spec", function() {
    it("is just a closure, so it can contain any code",
       function() {
        coldbox = 0;
        coldbox++;
        expect( coldbox ).toBe( 1 );
    });
    xit("can have more than one expectation, but I am
        skipped", function() {
        coldbox = 0;
        coldbox++;
        expect( coldbox ).toBe( 1 );
        expect( coldbox ).toBeTrue();
    });
});
```

The skip Argument

The skip argument can be a boolean value or a closure. If the value is **true** then the suite or spec is skipped. If the return value of the closure is **true** then the suite or spec is skipped. Using the closure approach allows you to dynamically at runtime figure out if the desired spec or suite is skipped. This is such a great way to prepare tests for different CFML engines.

```
describe(title="A railo suite", body=function() {
    it("can be expected to run", function() {
        coldbox = 0;
        coldbox++;
        expect( coldbox ).toBe( 1 );
    });
    it(title="can have more than one expectation and another
       skip closure", body=function() {
        coldbox = 0;
        coldbox++;
        expect( coldbox ).toBe( 1 );
        expect( coldbox ).toBeTrue();
    }, skip=function(){
        return false;
    });
}, skip=function(){
    return !structKeyExists( server, "railo" );
});
```

SPIES & MOCKING

Please refer to our MockBox guide to take advantage of all the mocking and stubbing you can do. However, every BDD TestBundle 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

As you can see from our arguments for a test suite, you can pass an `asyncAll` argument to the `describe()` blocks that will allow TestBox to execute all specs in separate threads for you concurrently.

```
describe(title="A spec (with setup and tear-down)",
         asyncAll=true, body=function() {
    beforeEach(function( currentSpec ) {
        coldbox = 22;
        application.wirebox = new coldbox.system
                                        .ioc.Injector();
    });
    afterEach(function( currentSpec ) {
        coldbox = 0;
        structDelete( application, "wirebox" );
    });
    it("is just a function, so it can contain any code",
       function() {
        expect( coldbox ).toBe( 22 );
    });
    it("can have more than one expectation and talk to
       scopes", function() {
        expect( coldbox ).toBe( 22 );
        expect( application.wirebox.getInstance( 'MyService' )
)                                   .toBeComponent();
    });
});
```

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 & produce reporter results
run()
// Run tests & get raw TestResults object
runRaw()
// Run tests & 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.

Here are the arguments you can use for initializing TestBox or executing the `run() or runRemote()` methods. However, please always refer to the latest included API docs:

`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]. It can also be just the mapping path.

`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.

`testBundles` - A list or array of bundle path names that are the ones that will be executed ONLY!

`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

Bundle CFC Runner

If you make your bundle CFCs inherit from **testbox.system.testing.BaseSpec** then you can execute your tests directly from the bundle CFC by using the following:

```
http://localhost/path/bundle.cfc?method=runRemote
```

```
component extends="testbox.system.testing.BaseSpec"{}
```

Another added benefit of the inheritance is that some IDEs will give you introspection and your tests will actually run faster. The URL arguments you can use when executing via the bundle CFC are:

`reporter` -The name of the reporter to use

`testBundles` - A list of bundle path names that are the ones that will be executed ONLY!

`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

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:

Codexwiki : Creates wiki complaint markup for usage in Codex Wiki or any other Mediawiki compliant wiki.

Console : Sends report to console

Doc : Builds semantic HTML, great for documentation

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

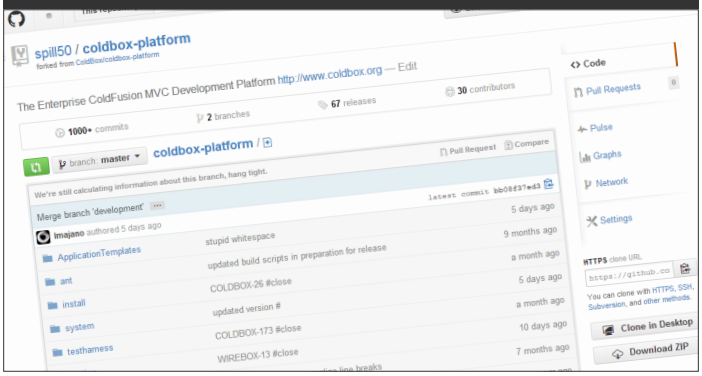
For API docs to see class definitions and method signatures, please visit the API docs located at http://www.coldbox.org/api

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



An Introduction to TestBox BDD

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

www.coldbox.org | www.ortussolutions.com

First Edition - October 2013