MightyMock

Last Updated : 03/11/2009

Table of Contents

What It Is

MightyMock is a simple lightweight for mocking component interaction in ColdFusion. It provides you the ability to easily and quickly define behaviors for dependencies, including both mocking and stubbing. When mocking, selective verification is possible, and when stubbing you can define return data or throw exceptions.

One slick feature of the MightyMock is the ability to match invocations by argument pattern. This gives you the flexibility to specify either literal arguments *or* patterns in your mock. More on this in the next section.

How To Use It

In general the steps will follow this simple pattern:

1. Create the mock
2. Define behavior
3. Inject the mock into the component under test
4. Run the component under test
5. Optionally verify how the mock was called

Stubbing

Suppose you have a component you want to test and that component calls a method on another component which returns a number needed by the 1st component:

```
<cfcomponent hint="Example Component To Mock" output="false">

<cffunction name="myMethod">
  <cfargument name="foo" />
  <cfset myData = myOtherComponent.doSomething('foo') />
  <!--- Do something with myData --->
  <cfreturn true />
</cffunction>

</cfcomponent>
```

Using MightyMock, you cans mock MyOtherComponent like this:

```
<cfcomponent extends="mxunit.framework.TestCase" output="false">

  <cffunction name="testMyComponentInteractions">
   <!--- Create the mock --->
   <cfset mock = createObject('component','MightyMock').init('MyOtherComponent');
   <!--- Define Behavior --->
   <cfset mock.doSomething('foo').returns( 123456 ) />
   <!--- Inject into component --->
   <cfset myComponent.setMyOtherComponent(mock) />
   <!--- Exercise MyComponent --->
   <cfset myComponent.myMethod('foo') />
   <!--- When doSomething('foo') is called by MyComponent, MightyMock
         will return the value 123456 to MyComponent
     --->
  </cffunction>

</cfcomponent>
```

Side Bar The init('component.name') statement above is optional. It's only required if your component under test requires an exact type; e.g.,

```
  <cffunction name="setMyOtherComponent">
   <cfargument name='aCollaborator' type='MyOtherComponent' />
   ...
  </cffunction>
```

Mocking

If your dependency does not return any data, but rather *does* something (returns void), you will want to verify that it was called, but maybe you don't want to incur side-effects, such as emails or logging:

```
<cffunction name="myMethod">
  <cfargument name="foo" />
  <cfset myOtherComponent.writeToLog('Hello.') />
  <!--- do a bunch of other stuff ... --->
  <cfset myOtherComponent.writeToLog('Good bye.') />
</cffunction>
```

A mock would like something like this:

```
<cffunction name="testMyComponent">
    <!--- Create the  mock --->
    <cfset mock = createObject('component','MightyMock').init('MyOtherComponent')
/>
    <!--- Define Behavior --->
    <cfset mock.writeToLog ('Hello.').returns() />
    <cfset mock.writeToLog ('Good Bye.').returns() />
    <cfset myComponent.setMyOtherComponent(mock) />
    <!--- Exercise MyComponent --->
    <cfset myComponent.myMethod('foo') />
    <!--- Verify --->
    <cfset mock.verify().writeToLog ('Hello.') />
    <cfset mock.verify().writeToLog ('Good Bye.') />
</cffunction>
```

If you are not overly concerned with the literal details of the method calls, you could simplify this with *argument patterns*:

```
<cffunction name="testMyComponent">
    <!--- Create the mock --->
    <cfset mock = createObject('component','MightyMock').init('MyOtherComponent')
/>
    <!--- Define Behavior --->
    <cfset mock.writeToLog('{string}').returns() />
    <cfset myComponent.setMyOtherComponent(mock) />
    <!--- Exercise MyComponent --->
    <cfset myComponent.myMethod('foo') />
    <!--- Verify --->
    <cfset mock.verifyTimes(2).writeToLog('{string}') />

</cffunction>
```

In the above example, we use the MightyMock keyword, {string}, instead of literal arguments. What happens under the hood is that when the method is called by the component that is passing in any single string argument as a parameter, the defined mock behavior is invoked.

Getting Bad-Ass with Mocks

Verification

MightyMock offers a couple of different ways perform verification of mocks.

Note: Verifying stubs is probably ultra-redundant because if your component under test cares anything about what the mock returns, it will probably fail before it can be verified.

Syntax: verify*Type*(int *count*).mockedMethod(params);

Types: verifyAtLeast(int *count*), verifyAtMost(int *count*), verifyTimes(int *count*), verifyOnce(), verifyNever(), verify(). verifyOnce(), verifyNever(), and verify() do not require any parameters.

Simple:

```
<cfset mock.verify().foo(1) />
```

This is the same as mock.verifyOnce().foo(1); or mock.verifyTimes(1).foo(1);

Chaining is also possible like this (assuming foo(1) was invoked 5 times):

```
   <cfset mock.verifyTimes(5)  .foo(1).
              verifyAtLeast(1).foo(1).
              verifyAtMost(5) .foo(1);   />
```

This can very powerful when verifying multiple mocked methods in the component under test.

Verifying Order

Frequently you will want to know if and how your mock executed and compare that with expectations. MightyMock offers you the ability to intuitively establish expectations and perform flexible verification. Instead of wiring this into a mock, we create an Order object and pass in the mocks to be verified:

```
  <cfset mock.one().returns() />
  <cfset mock.two().returns() />
  <cfset mock.three().returns() />
  <!---
    Inject mock into component and run it ...
   --->

 <!---
  Ok, now let's see what happened:
```

```
--->
<cfset order = createObject('component','mightymock.Order').init(mock) />

<cfset order.one().
             two().
             three().
             verify() />
```

To Do: verify() verifies range – makes sure that one was called before the other. verifyExactly() verifies the exact number and calls – everything must match *exactly*.

Argument Matching

MightyMock gives you the ability to mock using literal arguments or argument patterns. Imagine you have a component that sets a dozen HTTP headers. You have the option of explicitly mocking each header set or you can grab all by specifying a pattern:

Explicit literals

```
<cfset mock.myCollaborator.setHeader('X-Foo','Bar').returns() />
<cfset mock.myCollaborator.setHeader('X-Bar','Foo').returns() />
<cfset mock.myCollaborator.setHeader('X-Name','Mouse').returns() />
<cfset mock.myCollaborator.setHeader('X-Value','Cheese').returns() />
```

...

To Do: Need an easy way to verify all calls in one statement ... verifyAll().foo(1);

Alternatively, you could use an argument pattern:

```
<cfset mock.myCollaborator.setHeader('{string}','{string}').returns() />
```

MightyMock will invoke and record and calls made to setHeader(...) that have exactly two string parameters.

Partial Mocks and Spying

MightyMock provides you the ability to optionally mock methods in a real object and inject that object into your component under test. All calls made to the real object are recorded and can be inspected later, regardless if the method is mocked or not.

```
<cffunction name="testMyComponent">
    <!--- Create a mock --->
    <cfset mock = createObject('component','MightyMock').init('MyRealComponent',
true) />
   <!---
      Note that the optional TRUE init param tells MightyMock to create a reference
to the real object
   --->

 <!--- Define Behavior --->
```

To mock methods in a spy you need to tell MIghtyMock that you want to change the behavior of a method

```
  <cfset mock.mockSpy().foo().returns('bar') />


</cffunction>
```

Finé ...