Advanced Unit Testing Techniques in JavaScript

Guido Kessels on Jun 25th 2013 with 9 Comments

Tutorial Details

-
- **Difficulty**: Intermediate
- **Estimated Completion Time**: 30 minutes

View post on Tuts+ Beta**Tuts+ Beta** is an optimized, mobile-friendly and easy-to-read version of the Tuts+ network.

By now, everyone knows about Test-Driven Development and unit testing. But are you using the testing frameworks to their fullest?

Introduction

In this tutorial, I'll introduce you to some of the more advanced techniques available to you.

As this tutorial will cover some advanced topics, I assume you've already created unit tests before and are familiar with the basics and its terminology. If not, here's an excellent article to you get started: TDD Terminology Simplified.

We'll be using SinonJS. This is a standalone framework which provides an API for mocks, stubs, spies and more. You can use it with any testing framework of your choosing, but for this tutorial, we'll be using BusterJS, as it ships with SinonJS built–in.

Up & Running With BusterJS

To install BusterJS just run the following in a terminal: `npm install -g buster`

Note that you need Node v0.6.3 or newer.

BusterJS requires a `buster.js` configuration file which tells Buster where your sources and tests files are.

Create the `buster.js` file and paste in the following:

```
1  var config = module.exports;
2
3  config["Nettuts Tests"] = {
4      rootPath: "./",
5      environment: "browser",
6      sources: [
7          "src/*.js"
8      ],
9      tests: [
10         "spec/*-test.js"
11     ]
12 }
```

Now we've told Buster that our tests can be found in the `spec` folder, and our implementation code in the `src` folder. You can reference files by their filenames, or use wildcards as we have done here. Note that these are relative to the `rootPath` that we specified.

You can have as many configurations as you want. This allows you to set up different test groups.

To run a test with Buster you'll first need to start its server and hook it up to a browser. You can do so by running `buster server` in your terminal. You should see the following:

Now open your favorite browser and point it to http://localhost:1111. You should see the following screen:

Click the big **Capture Browser** button to start capturing this browser. Your tests will now run in this browser as long as you leave it open. You can hook up as many browsers as you want, giving you the ability to test in multiple browsers *simultaneously* (yes, even good ol' IE)!

You can also hook up BusterJS with PhantomJS to execute your tests without a browser, but that's outside the scope of this tutorial.

Make sure to leave the server and the browser tab running for the remainder of this tutorial.

To run your tests, simply type `buster test` in a new terminal tab/window. Once

you've added some tests you will see an output similar to the following:

Test Doubles

Before we dive into mocks and stubs, lets talk a bit about **test doubles**; A test double is an object which looks and behaves more or less like the real thing. They are used in a test when using the real object would be difficult or undesirable, and they make testing much easier.

It's commonly compared to using a *stunt double* instead of the real actor in a movie scene.

In this tutorial we'll cover the following types of test doubles:

- stubs
- spies
- mocks

Spies

A spy is a function that records all calls made to it. It will keep track of arguments, return values, the value of `this`, exceptions thrown (if any), etc. It can be an anonymous function or it can wrap an existing function. If used as a wrapper it will *not* modify the underlying function in any way; the original function will still be executed as usual.

Here's how you create a spy:

```
1 var spy = sinon.spy();
```

This creates an anonymous function that records arguments, the value of `this`,

exceptions, and return values for all calls it receives.

```
var spy = sinon.spy(my_function);
```

This spies on the provided function.

```
var spy = sinon.spy(object, "method");
```

This creates a spy for `object.method` and replaces the original method with the spy. The spy still executes the original method, but will record all calls.
You can access this spy via the newly created `spy` variable or by calling `object.method` directly. `object.method` can be restored by calling `spy.restore()` or `object.method.restore()`.

The returned spy object has the following methods and properties:

```
spy.withArgs(arg1[, arg2, ...]);
```

Creates a spy that *only* records calls when the received arguments match those passed to `withArgs`.

```
spy.callCount
```

Returns the number of recorded calls.

```
spy.called
```

Returns `true` if the spy was called at least once.

```
spy.calledOnce
```

Returns `true` if spy was called *exactly* one time.

```
spy.calledWith(arg1, arg2, ...);
```

Returns `true` if the spy was called *at least once* with the provided arguments. This can be used for partial matching. SinonJS will only check the provided arguments against actual arguments. So a call that receives the provided arguments (and possibly others) will return `true`.

```
spy.threw([exception]);
```

Returns `true` if the spy threw an exception at least once. If you want, you can pass

in a string or an object to test for a specific exception type or object.

```
var spyCall = spy.getCall(n);
```

Returns the `n`th call made to the spy. Spy calls have their own API, which you can find here: Spy Call API

```
spy.args
```

An array of arguments received per call. `spy.args[0]` is an array of arguments received in the first call, `spy.args[1]` is an array of arguments received in the second call, etc.

```
spy.reset()
```

Resets the state of a spy.

This was just a small excerpt of the methods available to you. For the full list of all available API methods check the documentation here: Spy API

Spies Example

Now lets look at an example on how to use a spy. In the following test we're checking if `jQuery.get()` is using `jQuery.ajax()`. We do this by *spy*ing on `jQuery.ajax()`.

```
buster.testCase("Spies", {

    tearDown: function() {
        jQuery.ajax.restore();
    },

    "should call jQuery.ajax when using jQuery.get": function() {

        sinon.spy(jQuery,"ajax");

        jQuery.get("/user");

        assert(jQuery.ajax.calledOnce);

    }

});
```

The original method is unaffected, and is still being executed. We just wrapped it in

a spy so we can record the calls to it.

After the test has run, we remove the spy from `jQuery.ajax` by calling `.restore()`.

Stubs

A stub is a test double with preprogrammed behavior. Stubs are used to make a piece of code work without actually using the *real implementation* of it.

It provides preprogrammed responses to calls, and it doesn't care about how many times it's called, in which order, or with which arguments.

Stubs have their own API, but also implement the full Spy API. Just like spies they can be either anonymous or wrap an existing function. Unlike spies, they will *not* execute the wrapped function. Instead, you can specify what the stub should do when it is called.

Because you can control exactly how a stub behaves, it allows you to easily test different flows in your code, or to prevent undesirable behavior to be executed.

Here's an excerpt from Sinon's Stub API:

```
1 var stub = sinon.stub();
```

This creates an anonymous stub function.

```
1 var stub = sinon.stub(object, "method");
```

This replaces `object.method` with a stub function. When stubbing an existing method like this, the original method will *not* be executed whenever `object.method()` is called.

The original function can be restored by calling `object.method.restore()` or `stub.restore()`.

```
1 var stub = sinon.stub(obj);
```

Stubs *all* the object's methods. It's usually considered a better practice to stub individual methods, which are less prone to unexpected behavior.

```
stub.withArgs(arg1[, arg2, ...]);
```

Stubs the method *only* for the provided arguments.

```
stub.returns(value);
```

Makes the stub return the provided `value`.

```
stub.returnsArg(index);
```

Causes the stub to return the argument at the provided index; `stub.returnsArg(0)` causes the stub to return the first argument.

```
stub.throws();
```

Causes the stub to throw an exception. Optionally, you can pass in the type of error to throw, e.g. `stub.throws("TypeError")`.

You can find the full API reference here: Stubs API

Stubs Examples

The simplest way to use a stub is by creating an anonymous stub function:

```
buster.testCase("Stubs Example", {

    "should demonstrate anonymous stub usage": function() {

        var callback = sinon.stub();

        callback.returns("result");

        assert.equals(callback(), "result");

    }

});
```

Here's a different example. It demonstrates how to stub methods to force the code flow down a certain path:

```
buster.testCase("User", {

    setUp : function() {

        this.user = new User({
```

```
            name  : 'John',
            age   : 24,
            loves : 'coffee'
        });

    },

    tearDown : function() {

        Database.saveRecord.restore();

    },

    "should return `User saved successfully` when save in database

        sinon.stub(Database, 'saveRecord').returns(true);

        var result = this.user.save();

        assert.equals(result, 'User saved successfully');

    },

    "should return `Error saving user` when save in database fails

        sinon.stub(Database, 'saveRecord').returns(false);

        var result = this.user.save();

        assert.equals(result, 'Error saving user');

    }

});
```

In the above tests we have a `User` class which uses the `Database` class to save data. Our goal is to test if the `User` class responds with the correct message when the `Database` is done saving the user data. We want to test both the good and bad scenarios.

In a production environment the `Database` class might do various things to save the data (connect to a real database, do some AJAX calls, etc.) which are not of interest for this test. It could even have a negative effect on our test results. If something in the `Database` class is broken, we want the `Database` class's own unit tests to break and indicate the problem for us. Other classes which use the `Database` class as a dependency should still work as expected. Mocking or stubbing dependencies allows

us to do this, which is a strong argument for using these in the first place.

In the above test we use a stub to preprogram the `Database.saveRecord()` method behavior. This allows us to test both code paths we need for our test.

After each test we call `.restore()` on the method we stubbed to restore the original method.

In the above example we stub *all* calls to `Database.saveRecord()`. We can also limit our stub to calls which have a certain collection of arguments.

Here's a quick example of how to force different actions based on the passed arguments:

```
buster.testCase("Stubs", {

    "should stub different behaviour based on arguments": function

        var callback = sinon.stub();

        // Stub the same method in 3 different ways, based on the
        callback.withArgs('success').returns(true);
        callback.withArgs('getOrder').returns(['pizza', 'icecream'
        callback.withArgs(false).throws("My Error");

        // Verify each stub
        assert( callback('success') );
        assert.equals( callback('getOrder'), ['pizza', 'icecream']

        try {
            callback(false)
        } catch(e) {}
```

```
19
20          assert( callback.threw("My Error"), "Exception 'My Error'
21
22      }
23
24  });
```

Mocks

Mocks are stubs with **preprogrammed expectations**. They allow you to verify the *behavior* of a piece of software, as opposed to verifying the *state* of something, as you'd do with normal assertions.

Here's a list of Sinon's Mock API:

```
1  var mock = sinon.mock(obj);
```

This creates a mock for the provided object. It does not modify the object, but returns a mock object to set expectations on the object's methods.

```
1  var expectation = mock.expects("method");
```

This overrides `obj.method` with a mock function and returns it. Expectations come with their own API, which we'll cover later.

```
1  mock.restore();
```

Restores all mocked methods to their original functions.

```
1  mock.verify();
```

Verifies all expectations on the mock. If any expectation is not met, an exception is thrown. This will also restore the mocked methods to their original functions.

Mocks also implement the full Stub API.

Mocks Examples

Now lets see how we can implement this in the `User` example we used earlier when talking about stubs.

Remember how it used the `Database.saveRecord` method? We never wrote a test to make sure the `User` class is actually calling this method correctly, we just *assumed* it would.

We don't have any tests to verify the communication between the two objects, but we can fix this easily by writing the following test:

```
buster.testCase("User", {

    setUp : function() {

        var userdata = this.userdata = {
            name  : 'John',
            age   : 24,
            loves : 'coffee'
        };

        this.user = new User(userdata);

    },

    "should use Database class to save userdata": function() {

        var mock = sinon.mock(Database);

        mock
            .expects('saveRecord')
            .withExactArgs(this.userdata)
            .once();

        this.user.save();

        mock.verify();

    }

});
```

As you can see, we mocked the `Database` object and explicitly stated how we expect the `saveRecord` method to be called. In this test we expect the method to be called only once, with the `userdata` object as the only parameter.

Because our expectations are already in our mock, we do not need to write any assertions, instead we just tell the mock to verify its expectations by using `mock.verify()`.

If the mock was called more than once, or with parameters other than those we specified, it would throw an error which would make the test fail:

Lets look at an other example where mocks could come in handy.

If you've worked with unit tests before in a PubSub system, you'll probably have seen something similar to the following:

```
"should execute subscribers with correct data": function() {

    var pubsub = new PubSub(),
        called = false,
        eventdata = { foo : 'bar' },
        callback = function(data) {
            called = (data === eventdata);
        };

    pubsub.subscribe("message", callback);
    pubsub.publish("message", eventdata);

    assert(called);

}
```

This test verifies that the subscriber is called when an event is published.

The `callback` function is acting more or less like a mock, as it's verifying if it was called with the correct arguments. Lets improve the test by turning `callback` into a real mock:

```
"should execute subscribers with correct data (using mocks)": func

    var pubsub = new PubSub(),
        eventdata = { foo : 'bar' },
        callback = sinon.mock().withExactArgs(eventdata).once();

    pubsub.subscribe("message", callback);
    pubsub.publish("message", eventdata);

    callback.verify();

}
```

Easy as pie. And it also improved the readability of the test!

Expectations

The `.once()` and `.withExactArgs()` methods used above are **expectations**. Sinon offers a ton of different expectations that you can use for your mocks. Here are a few of my favorites:

```
1 expectation.atLeast(n)
```

Expect the method to be called a *minimum* of `n` times.

```
1 expectation.atMost(n)
```

Expect the method to be called a *maximum* of `n` times.

```
1 expectation.never()
```

Expect the method to never be called.

```
1 expectation.once()
```

Expect the method to be called *exactly* once.

```
1 expectation.exactly(n)
```

Expect the method to be called exactly `n` times.

```
1 expectation.withArgs(arg1, arg2, ...)
```

Expect the method to be called with the provided arguments, and possibly others.

```
1 expectation.withExactArgs(arg1, arg2, ...)
```

Expect the method to be called with the provided arguments, *and no others*.

```
1 expectation.verify()
```

Verifies the expectation and throws an exception if it's not met.

The full list of expectations can be found here: Expectations API

Expectations Examples

You can chain these expectations to your heart's content. So this is totally valid:

```
sinon.mock(obj)
     .expects('method')
     .withExactArgs(data)
     .atLeast(1)
     .atMost(3);
```

Also, you can set expectations for multiple methods on the same mock simultaneously:

```
var mock = sinon.mock(obj);

mock.expects('method1')
    .atLeast(1)
    .atMost(3);

mock.expects('method2')
    .withArgs(data)
    .once();
```

Or even set multiple expectations on the *same* method:

```
var mock = sinon.mock(obj);

mock.expects('myMethod')
    .withArgs('foo')
    .atLeast(1)
    .atMost(3);

mock.expects('myMethod')
    .withArgs('bar')
    .exactly(4);
```

Both expectations will have to be met for the test to pass.

Now that we've covered test doubles, lets talk about something completely different, but equally awesome: *time travel*!

Time-Travel in Unit Tests

I don't always bend time and space in unit tests, but when I do, I use

Buster.JS + Sinon.JS ~ Brian Cavalier, Cujo.JS

Do you often use `setTimeout`, `clearTimeout`, `setInterval`, or `clearInterval` to delay execution of a piece of code? If so, then you've probably encountered tests like this:

```
buster.testCase("EggTimer", {

    "should execute callback method after 5000ms": function(done)

        // Overwrite BusterJS default test timeout of 250ms
        this.timeout = 6000;

        var mock = sinon.mock().once();

        EggTimer.start(5000, mock);

        setTimeout(function() {
            mock.verify();

            // Because of the asynchronous nature of setTimeout,
            // we need to tell BusterJS when our test is done:
            done();
        }, 5001);

    }

});
```

This test verifies if the **EggTimer.start** method executes the callback after a certain period of time. But by doing so, it forces you to wait for five plus seconds *every time you run the test*!

Imagine having ten tests which rely on `setTimeout` in this way; Your test suite will quickly become so slow, you'll start hating to run it.

Fortunately, SinonJS provides **fake timers** which allow us to override the browser's clock and travel forward in time — Great Scott!

We can do this by using the `sinon.useFakeTimers()` method. By doing so, SinonJS will create a clock object and override the browser's default timer functions with its own.

The returned clock object has only two methods:

```
clock.tick(time)
```

Tick the clock ahead for `time` milliseconds. This causes all timers scheduled within the specified time period to be executed.

```
clock.restore()
```

This call is usually done in the *tearDown* step of a test(suite). It resets the timer functions back to the browser's native ones.

Fake Timers Example

Now that we know about fake timers, lets see how we can use them to rewrite the above test:

```
buster.testCase("EggTimer (with fake timers)", {

    setUp: function () {
        this.clock = sinon.useFakeTimers();
    },

    tearDown: function () {
        this.clock.restore();
    },

    "should execute callback method after 5000ms": function() {

        var mock = sinon.mock().once();

        EggTimer.start(5000, mock);
        this.clock.tick(5001);

        mock.verify();

    }

});
```

First we've added `setUp` and `tearDown` methods to override and restore the browser's clock before and after each test.

Then we used the `clock.tick()` method to travel forward in time. Because SinonJS's fake timers are synchronous implementations, we no longer need the `done()` call. As an added benefit, our test is now much easier to read.

Here's a speed comparison:

Our rewritten test takes the total test execution time down from 5012ms to 12ms! We saved exactly 5000ms, which was the value we used in the first test's `setTimeout()` call!

By using fake timers, having ten of these tests is no big deal. It will only increase the total test execution time by a few milliseconds, as opposed to 5000ms per added test!

More information on Sinon's clock and timer functions can be found here: Clock API

Conclusion

We've looked at various advanced techniques which you can use in your JavaScript unit tests. We discussed **spies**, **stubs**, **mocks**, and how to **fake the browser's timer functions**.

We used SinonJS for this, but most other testing frameworks (like Jasmine) have support for these functionalities (although with their own API).

If you're interested in more in-depth knowledge on unit testing in JavaScript, I highly recommend the Test-Driven JavaScript Development book by Christian Johansen (the creator of SinonJS.)

I hope this article was helpful, and that you've learned some new techniques that you can use the next time you write a unit test. Thank you for reading.

Tags: javascriptunit testing

By Guido Kessels

This author has yet to write their bio.

Note: Want to add some source code? Type <pre><code> before it and </code></pre> after it. Find out more