Sinon.js in Real-Life

Jani Hartikainen - codeutopia.net

Contents

Introduction

Real-life testing is often different from the theoretical side. Whenever I teach something, my goal is always to make it as practical as possible - what use is theory if you don't know where, why or how to apply it?

That's why I've put together this ebook for you. Here, you'll find several examples of how Sinon can be used to solve real-life problems. In addition, I've also included a useful reference of Sinon best practices, so you can avoid common problems.

If you have any questions or comments, don't hesitate to email me. My address is jani@codeutopia.net

How to test Ajax requests with Sinon

Ajax is a great example of something that can be difficult to test. You've got a specific URL where a request is being sent, and usually need to get some specific data back from the server. Since it's often used to both save and load data, it's vital any code handling ajax results works correctly.

Thankfully with Sinon, testing Ajax requests becomes simple!

Setup

For this example, we'll be running the test in a browser. For this, we need to set up a *test runner page*. The page is then loaded in a browser to run our tests, as one might expect based on the name.

If you prefer using command-line Mocha, don't worry - the tests themselves are written almost exactly the same as you would write them when testing CommonJS modules / Node.js modules.

Although we're using this in a browser, we can still make use of npm to install our tooling:

```
npm install mocha sinon chai
```

Below is the test runner file we will use. I'm going to call it `testrunner.html`

```
<!DOCTYPE html>
<html>
  <head>
    <title>Mocha Tests</title>
    <link rel="stylesheet" href="node_modules/mocha/mocha.css">
  </head>
  <body>
    <div id="mocha"></div>
    <script src="node_modules/mocha/mocha.js"></script>
    <script src="node_modules/sinon/pkg/sinon-1.12.2.js"></script>
    <script src="node_modules/chai/chai.js"></script>
    <script>mocha.setup('bdd')</script>
    <script src="myapi.js"></script>
    <script src="test.js"></script>
    <script>
      mocha.run();
    </script>
  </body>
</html>
```

Note the paths for `mocha.css`, `mocha.js`, `sinon-1.12.2.js` and `chai.js`. Since we installed them using npm, they are within the `node_modules` directory. For

Sinon, you may need to adjust the filename to match the installed version.

Also note the `myapi.js` and `test.js` files. These are our example module and test case, which I'll introduce next.

Example module

Below I've created a basic module which does some Ajax requests. I'll use this to show you the techniques for testing Ajax code.

I'm calling this file `myapi.js`

```
var myapi = {
  get: function(callback) {
    var xhr = new XMLHttpRequest();
    xhr.open('GET', 'http://jsonplaceholder.typicode.com/posts/1', true);

    xhr.onreadystatechange = function() {
      if(xhr.readyState == 4) {
        if(xhr.status == 200) {
          callback(null, JSON.parse(xhr.responseText));
        }
        else {
          callback(xhr.status);
        }
      }
    };

    xhr.send();
  },

  post: function(data, callback) {
    var xhr = new XMLHttpRequest();
    xhr.open('POST', 'http://jsonplaceholder.typicode.com/posts', true);

    xhr.onreadystatechange = function() {
      if(xhr.readyState == 4) {
        callback();
      }
    };

    xhr.send(JSON.stringify(data));
  }
};
```

Here we have two functions, one for fetching data, another for posting data. I'm using the JSONPlaceholder API, which is nice for quick testing.

This should look like a fairly typical case - quite often, you'll have some code like this, which performs ajax requests to specific URLs, and doing something with the result.

Test case skeleton

Let's create a quick skeleton where we can add tests as we go through each scenario. I'm calling this file `test.js`

```
var expect = chai.expect;

describe('MyAPI', function() {
  //Tests etc. go here
});
```

Here, we've only set up a variable for `expect` and a `describe` block where we can put our tests. The variable is so that we can easily use Chai's convenient expect-api for doing assertions.

Testing a GET request

We'll start by creating a test to verify the fetched data is parsed from JSON correctly. But since we don't want to send HTTP requests from unit tests, what can we do about the `XMLHttpRequest`?

Remember the basic process of using Sinon: replace the problematic function with a test double. In this case, Sinon provides us a convenient way of doing exactly that.

Let's make use of Sinon's *fake XMLHttpRequests feature* to stub out the problem:

```
var expect = chai.expect;

describe('MyAPI', function() {
  var xhr;
  var requests;
  beforeEach(function() {
    xhr = sinon.useFakeXMLHttpRequest();

    requests = [];
    xhr.onCreate = function(xhr) {
      requests.push(xhr);
    };
  });

  afterEach(function() {
```

```
    xhr.restore();
  });

  //Tests etc. go here
});
```

As their names might lead you to guess, `beforeEach` and `afterEach` let us run code before and after each test. Before each test, we set up Sinon's fake XHR functionality. We can then use the `xhr` variable within our tests to control what happens. In addition, we set up a `onCreate` callback - this is a special feature of Sinon's fake XHR, which lets us get a reference to each created fake XMLHttpRequest object.

After each test, we use `xhr.restore()` to restore the original XMLHttpRequest object back. This is important to remember to do, as without it, you may see your tests behave strangely.

Now with that out of the way, we can actually write our test...

```
it('should parse fetched data as JSON', function(done) {
  var data = { foo: 'bar' };
  var dataJson = JSON.stringify(data);

  myapi.get(function(err, result) {
    expect(result).to.deep.equal(data);
    done();
  });

  requests[0].respond(200, { 'Content-Type': 'text/json' }, dataJson);
});
```

First some data: An object and its JSON version. We define these to avoid using having to repeat the values within the test. Next, we call `myapi.get`. In its callback, we validate result with the test data. We also call `done()`, which tells Mocha the asynchronous test is complete. Note that `done` was a parameter on the test function.

Finally, we call `requests[0].respond`. Remember that we created a listener in the beforeEach function to put all created fake XMLHttpRequests into the `requests` array. Since `myapi.get` creates an XMLHttpRequest, we can access it here thanks to Sinon.

Normally XMLHttpRequests don't have a `respond` function. `respond` is a special function of the fake XHR, which we can use to send a response to the request. In this case, we set the status code as `200` to indicate success. We also set a `Content-Type` header as `text/json` to show the data is JSON formatted. The last parameter is the response body, which we set to the `dataJson` variable we set up earlier.

This way, the fake XMLHttpRequest looks like it behaves like a normal JSON

request to our code in `myapi.get`. The fake XMLHttpRequest pretends the JSON we gave it was sent by a web server. Then `myapi.get` sees it and calls the callback. Looking at the callback, notice we compare the result against the data variable:

```
myapi.get(function(err, result) {
  expect(result).to.deep.equal(data);
  done();
});
```

If the response is handled correctly, `myapi.get` should call its callback with the data parsed from JSON. Since we set up the `data` and `dataJson` variables to represent that earlier, we simply compare against that.

We can now run the tests in our browser of choice. Open the `testrunner.html` file, and you should see a message about the test passing.

Testing a POST request

The posted data needs to be encoded as JSON and sent in the request body. Let's write a test for this.

```
it('should send given data as JSON body', function() {
  var data = { hello: 'world' };
  var dataJson = JSON.stringify(data);

  myapi.post(data, function() { });

  expect(requests[0].requestBody).to.equal(dataJson);
});
```

Like before, we first define the test data. Then, we call `myapi.post`. We only need to verify the data is correctly converted into JSON, so we leave the callback empty.

The final line has an assertion to verify the behavior. Like in the previous test, we access the created XMLHttpRequest, but this time we only need to verify it contains the correct data. We can do this by comparing the requestBody property with the test data we defined earlier.

Testing for failures

As the final example, let's test a failing request. This important because network connections can have problems and the server can have problems, and we shouldn't leave the user wondering "What happened?".

The example module uses node-style callbacks – Meaning it should pass errors into the callback as the first parameter. We can test it like below:

```
it('should return error into callback', function(done) {
  myapi.get(function(err, result) {
    expect(err).to.exist;
    done();
  });

  this.requests[0].respond(500);
});
```

This time we don't need any data. We just call `myapi.get`, and verify the error parameter exists. The last line sends an erroneous response code, 500 Internal Server Error, to trigger the error handling code.

Conclusion

Ajax requests are important to test. If you verify they work correctly, the rest of your application can trust it will not get bad values from them. That can save you a lot of headaches down the line.

What if you're using jQuery Ajax instead of plain XMLHttpRequest? No problem. You can use exact same methods shown here to test your code. You can use the same approach with Sinon's fake XMLHttpRequests, as behind the scenes, jQuery also uses the basic XMLHttpRequest object.

How to test Node.js HTTP requests with Sinon

When unit testing, you don't want HTTP requests to go out and affect the result. We can easily solve this with Sinon. However, there is one slightly bigger challenge involved: Streams. They can be a bit complicated to deal with.

Let's look at some typical examples of using the `http` module in Node, and how to approach unit testing the resulting module. Here's what we'll look at in this chapter:

- Sending a GET request and testing response handling
- Sending a POST request and testing the parameter behavior
- Testing that failures are handled correctly

Example scenario

This example is based on real-world code I encountered in a project. I did simplify it a little bit, so that it's easier to follow what's going on. The techniques are

still the same as you would use in real production code.

To start with, here's a basic Node.js module which has some functionality to do HTTP requests to an external API. I'm going to call this file `api.js`

```
var http = require('http');

module.exports = {
    get: function(callback) {
        var req = http.request({
            hostname: 'jsonplaceholder.typicode.com',
            path: '/posts/1'
        }, function(response) {
            var data = '';
            response.on('data', function(chunk) {
                data += chunk;
            });

            response.on('end', function() {
                callback(null, JSON.parse(data));
            });
        });

        req.end();
    },

    post: function(data, callback) {
        var req = http.request({
            hostname: 'jsonplaceholder.typicode.com',
            path: '/posts',
            method: 'POST'
        }, function(response) {
            var data = '';
            response.on('data', function(chunk) {
                data += chunk;
            });

            response.on('end', function() {
                callback(null, JSON.parse(data));
            });
        });

        req.write(JSON.stringify(data));

        req.end();
    }
};
```

This should look familiar if you've used the `http` module. One function for fetching data, another for sending data. The code uses the JSONPlaceholder API, a simple REST API useful for this kind of prototyping and testing.

Setting up the test suite

Next, let's create the test suite. We will use this to contain each individual test, and do some basic setup. I'm going to put this file into `test/apiSpec.js`.

```
var assert = require('assert');
var sinon = require('sinon');
var PassThrough = require('stream').PassThrough;
var http = require('http');

var api = require('../api.js');

describe('api', function() {
    var request;
    beforeEach(function() {
        request = sinon.stub(http, 'request');
    });

    afterEach(function() {
        request.restore();
    });


    //We'll place our tests cases here

});
```

The `assert` module can be used as a simple way to do assertions for our tests. You can also use Chai if you wish, but I'll use `assert` here to keep things simple. We also include `PassThrough` from the `stream` module. This is a stream which simply passes all data through it, and we can use it as a test-double for streams.

The key things here are `beforeEach` and `afterEach`. `beforeEach` creates a stub to replace `http.request`, and `afterEach` restores the original functionality. We're storing the stub into `request` within the test case, so that we can alter its behavior in each individual test that we'll add later.

An important aspect to note is how we're stubbing something from another module. We use the `http` module in two places. How can stubbing it here work? The key lies in how Node.js *caches* requires - that is, each module requiring something gets the same copy. This is why stubbing the function here also affects the other file.

Testing the GET request

Let's start by adding a test that validates the get request handling. When the code runs, it calls `http.request`. Since we stubbed it, the idea is we'll use the stub control what happens when `http.request` is used, so that we can recreate all the scenarios without a real HTTP request ever happening.

```
it('should convert get result to object', function(done) {
    var expected = { hello: 'world' };
    var response = new PassThrough();
    response.write(JSON.stringify(expected));
    response.end();

    var request = new PassThrough();

    request.yields(response)
           .returns(request);

    api.get(function(err, result) {
        assert.deepEqual(result, expected);
        done();
    });
});
```

First, we define the expected data that we will use in our test and create a response stream using PassThrough. When an `http.request` stream is open, it would normally let us read the response string. To mimic that behavior for the test, we write a JSON version of our expected data into the PassThrough response and end it. This way, when the code we're testing tries to read it, the stream will give it the data it should.

We create a PassThrough stream for the request as well. In this case, we don't need it to do anything - instead, its purpose is again to mimic how calling `http.request` would normally return a stream.

A successful `http.request` will pass the response stream to its callback. We can use `yields` to tell our stub to call the first callback function it encounters, optionally passing a parameter to it - in this case, the `response` stream we created earlier. We also tell the stub to return the `request` stream, so it behaves like the not-stubbed function.

With the setup out of the way, we call `api.get` with a callback which verifies the behavior.

We can run the test using `mocha`. You should find the test passes with a green light.

Testing the POST request

For the POST scenario, we want to ensure the parameters are passed correctly to the http request.

```
it('should send post params in request body', function() {
    var params = { foo: 'bar' };
    var expected = JSON.stringify(params);

    var request = new PassThrough();
    var write = sinon.spy(request, 'write');

    request.returns(request);

    api.post(params, function() { });

    sinon.assert.calledWith(write, expected);
});
```

When calling `api.post`, it should send parameters in the request body. That's why in this test, we only need to verify `request.write` is called with the correct value.

Like previously, we define the expected data first. Doing this helps make the test easier to maintain, as we're not using magic values all over the place.

We then define a request stream. To make sure the expected data is written into it, we need a way to check write was called. In this case, we can use a spy as a convenient way of checking the function gets called, without affecting its behavior.

The `http.request` function only needs to return the request stream in this test. Whether the callback gets called or not will not affect the result, therefore we should omit it for clarity.

We call `api.post`, passing in our parameters and an empty callback, as the callback's behavior is not related to this particular test.

In this test, we're using `sinon.assert.calledWith` instead of `assert`. Sinon has a number of built-in assertions like these, which make it convenient to check more complex requirements, like what we have here.

We can run the tests again using `mocha`, and they'll pass.

Note: in a real app, you might want to have a test for the callback, but it should go into its own test case. A single test case should ideally have one assertion only. The test for a callback would resemble the test with a get request, so I won't go into detail on it here.

Testing a failure scenario

Something that's easy to forget is failure scenarios. Often, people only write tests for the so-called happy path. Let's make sure the sad path (ha) is also covered.

Most http request testing related libraries on npm had difficulties solving this, but using the PassThrough stream solves it cleanly as we'll see. It goes to show how powerful Sinon is when combined with other simple components.

```
it('should pass request error to callback', function(done) {
    var expected = 'some error';
    var request = new PassThrough();

    request.returns(request);

    api.get(function(err) {
        assert.equal(err, expected);
        done();
    });

    request.emit('error', expected);
});
```

This should be starting to look familiar. We define the expected data, create a request stream, tell our stub to return it and call the api. The callback passed to the api checks the error parameter is correct.

The new thing here is the last line. When an http request fails due to network errors, http parsing errors or such, it emits an error event. To simulate an error, we emit one from the test.

Run tests with mocha, and they'll... fail? Huh?

This is why unit tests are important, and especially testing the failure cases! It turns out a bug snuck into the code.

No problem, let's add handling for the error event.

```
get: function(callback) {
    var req = http.request({
        hostname: 'jsonplaceholder.typicode.com',
        path: '/posts/1'
    }, function(response) {
        var data = '';
        response.on('data', function(chunk) {
            data += chunk;
        });
```

```
        response.on('end', function() {
            callback(null, JSON.parse(data));
        });
    });

    req.on('error', function(err) {
        callback(err);
    });

    req.end();
},
```

The only thing added here is the three lines to handle the error event.

Run tests with `mocha` and now they will pass. Done!

Testing Timers with Sinon

Another common problem scenario is dealing with timers. Sometimes you need to have code which uses `setTimeout` or `setInterval`. How could we go about testing that?

Sinon provides a convenient *fake timers* feature, which makes it easy to do this.

Example scenario

Let's imagine we want to have some animations play, but only after they've been visible for some amount of time. This would typically mean you need to set a timeout to wait, and play the animation after that time has passed.

A similar approach could be used for playing video when it's visible on the user's screen - if you use Facebook, I'm sure you've seen videos autoplay in the timeline when you see them.

The code for animating after a delay might look something like this:

```
function waitForAnimation() {
    var time = 5000;

    setTimeout(function() {
        playAnimation();
    }, time);
}
```

Let's see how we can test this.

Using Fake Timers

We can enable fake timers with Sinon using `sinon.useFakeTimers()`. This gives us a *clock* object, which can be used to control time.

```
var clock = sinon.useFakeTimers();

//advance time by 500 milliseconds
clock.tick(500);

//remove fake timers
clock.restore();
```

As you can see, this works similar to other test-doubles: We call a function to set it up. The returned object can be used to control time by calling `tick`. And lastly, we need to remember to clean up any timers using `restore`, just like with other test-doubles.

A convenient way of using fake timers is with `beforeEach` and `afterEach` hooks:

```
describe('Delayed animation', function() {
    var clock;
    beforeEach(function() {
        clock = sinon.useFakeTimers();
    });

    afterEach(function() {
        clock.restore();
    });

    //we'll put our tests here
});
```

With this set up, we can easily use `clock.tick` to test our timeout, interval or `Date` using code.

Testing the animation plays when expected

First, let's create a test to see if the animations will play at the time we want them to.

```
it('should play animations after 5 seconds', function() {
    var playAnimation = sinon.stub(window, 'playAnimation');

    waitForAnimation();
    clock.tick(5000);

    playAnimation.restore();
```

```
    sinon.assert.calledOnce(playAnimation);
}));
```

Here, we first stub out the `playAnimation` function. Note that we're using `window` as the first parameter to `sinon.stub`. If the function you want to stub is a global, we can use `window` like this to stub it.

Next, we call `waitForAnimation` - this sets up the timer. We then call `clock.tick(5000)` to advance the time by 5 seconds.

Finally, we restore the stub and do an assertion. This test will only pass if the `playAnimation` stub was called, which should happen, since we set a timer for 5 seconds, and also advanced the time by 5 seconds.

Testing the animation doesn't play too early

We can also easily check the animation doesn't play early:

```
it('should not play animations too early', function() {
    var playAnimation = sinon.stub(window, 'playAnimation');

    waitForAnimation();
    clock.tick(500);

    playAnimation.restore();
    sinon.assert.notCalled(playAnimation);
}));
```

This test is mostly the same as the previous one, with two differences: Instead of advancing the time by 5 seconds, we only advance it by 0.5 seconds. Second, we've changed the assertion to `notCalled` to verify the `playAnimation` stub wasn't called at all.

Conclusion

Sinon's fake timers are very simple to use, but like other features in Sinon, they are very powerful and convenient for testing. You can use them with any time-related functionality: `setTimeout`, `setInterval`, or even `Date`.

Sinon.js best practices

Sometimes getting started with Sinon can be tricky. You get a lot of functionality in the form of spies, stubs and mocks, but it can be difficult to choose when to use what. They also have some gotchas, so you need to know what you're doing to avoid problems.

In this last chapter of the ebook, I'll show you the differences between spies, stubs and mocks, when and how to use them, and give you a set of best practices to help you avoid common pitfalls.

Example Function

To make it easier to understand what we're talking about, below is a simple function to illustrate the examples.

```
function setupNewUser(info, callback) {
  var user = {
    name: info.name,
    nameLowercase: info.name.toLowerCase()
  };

  try {
    Database.save(user, callback);
  }
  catch(err) {
    callback(err);
  }
}
```

The function takes two parameters — an object with some data we want to save and a callback function. We put the data from the `info` object into the `user` variable, and save it to a database. For the purpose of this tutorial, what `Database.save` does is irrelevant — it could send an Ajax request, or, if this was Node.js code, maybe it would talk directly to the database, but the specifics don't matter. Just imagine it does some kind of a data-saving operation.

Spies, Stubs and Mocks

Together, spies, stubs and mocks are known as *test doubles*. Similar to how stunt doubles do the dangerous work in movies, we use test doubles to replace troublemakers and make tests easier to write.

When Do You Need Test Doubles?

To best understand when to use test-doubles, we need to understand the two different types of functions we can have. We can split functions into two categories:

- Functions without side effects
- And functions with side effects

Functions without side effects are simple: the result of such a function is only dependent on its parameters — the function always returns the same value given the same parameters.

A function with side effects can be defined as a function that depends on something external, such as the state of some object, the current time, a call to a database, or some other mechanism that holds some kind of state. The result of such a function can be affected by a variety of things in addition to its parameters.

If you look back at the example function, we call two functions in it — `toLowerCase`, and `Database.save`. The former has no side effects - the result of `toLowerCase` only depends on the value of the string. However, the latter has a side effect - as previously mentioned, it does some kind of a save operation, so the result of `Database.save` is also affected by that action.

If we want to test `setupNewUser`, we may need to use a test-double on `Database.save` because it has a side effect. In other words, we can say that we need test-doubles when the function has side effects.

In addition to functions with side effects, we may occasionally need test doubles with functions that are causing problems in our tests. A common case is when a function performs a calculation or some other operation which is very slow and which makes our tests slow. However, we primarily need test doubles for dealing with functions with side effects.

When to Use Spies

As the name might suggest, spies are used to get information about function calls. For example, a spy can tell us how many times a function was called, what arguments each call had, what values were returned, what errors were thrown, etc.

As such, a spy is a good choice whenever the goal of a test is to verify something happened. Combined with Sinon's assertions, we can check many different results by using a simple spy.

The most common scenarios with spies involve...

- Checking how many times a function was called
- Checking what arguments were passed to a function

We can check how many times a function was called using `sinon.assert.callCount`, `sinon.assert.calledOnce`, `sinon.assert.notCalled`, and similar. For example, here's how to verify the save function was being called:

```
it('should call save once', function() {
  var save = sinon.spy(Database, 'save');
```

```
  setupNewUser({ name: 'test' }, function() { });

  save.restore();
  sinon.assert.calledOnce(save);
});
```

We can check what arguments were passed to a function using `sinon.assert.calledWith`, or by accessing the call directly using `spy.lastCall` or `spy.getCall()`. For example, if we wanted to verify the aforementioned save function receives the correct parameters, we would use the following spec:

```
it('should pass object with correct values to save', function() {
  var save = sinon.spy(Database, 'save');
  var info = { name: 'test' };
  var expectedUser = {
    name: info.name,
    nameLowercase: info.name.toLowerCase()
  };

  setupNewUser(info, function() { });

  save.restore();
  sinon.assert.calledWith(save, expectedUser);
});
```

These are not the only things you can check with spies though — Sinon provides many other assertions you can use to check a variety of different things. The same assertions can also be used with stubs.

If you spy on a function, the function's behavior is not affected. If you want to change how a function behaves, you need a stub.

When to Use Stubs

Stubs are like spies, except in that they replace the target function. They can also contain custom behavior, such as returning values or throwing exceptions. They can even automatically call any callback functions provided as parameters.

Stubs have a few common uses:

- You can use them to replace problematic pieces of code
- You can use them to trigger code paths that wouldn't otherwise trigger - such as error handling
- You can use them to help test asynchronous code more easily

Stubs can be used to replace problematic code, i.e. the code that makes writing tests difficult. This is often caused by something external - a network connection, a database, or some other non-JavaScript system. The problem with these is that

they often require manual setup. For example, we would need to fill a database with test data before running our tests, which makes running and writing them more complicated.

If we stub out a problematic piece of code instead, we can avoid these issues entirely. Our earlier example uses `Database.save` which could prove to be a problem if we don't set up the database before running our tests. Therefore, it might be a good idea to use a stub on it, instead of a spy.

```
it('should pass object with correct values to save', function() {
  var save = sinon.stub(Database, 'save');
  var info = { name: 'test' };
  var expectedUser = {
    name: info.name,
    nameLowercase: info.name.toLowerCase()
  };

  setupNewUser(info, function() { });

  save.restore();
  sinon.assert.calledWith(save, expectedUser);
});
```

By replacing the database-related function with a stub, we no longer need an actual database for our test. A similar approach can be used in nearly any situation involving code that is otherwise hard to test.

Stubs can also be used to trigger different code paths. If the code we're testing calls another function, we sometimes need to test how it would behave under unusual conditions — most commonly if there's an error. We can make use of a stub to trigger an error from the code:

```
it('should pass the error into the callback if save fails', function() {
  var expectedError = new Error('oops');
  var save = sinon.stub(Database, 'save');
  save.throws(expectedError);
  var callback = sinon.spy();

  setupNewUser({ name: 'foo' }, callback);

  save.restore();
  sinon.assert.calledWith(callback, expectedError);
});
```

Thirdly, stubs can be used to simplify testing asynchronous code. If we stub out an asynchronous function, we can force it to call a callback right away, making the test synchronous and removing the need of asynchronous test handling.

```
it('should pass the database result into the callback', function() {
```

```
  var expectedResult = { success: true };
  var save = sinon.stub(Database, 'save');
  save.yields(null, expectedResult);
  var callback = sinon.spy();

  setupNewUser({ name: 'foo' }, callback);

  save.restore();
  sinon.assert.calledWith(callback, null, expectedResult);
});
```

Stubs are highly configurable, and can do a lot more than this, but most follow these basic ideas.

When to Use Mocks

You should take care when using mocks - it's easy to overlook spies and stubs when mocks can do everything they can, but mocks also easily make your tests overly specific, which leads to brittle tests that break easily. A brittle test is a test that easily breaks unintentionally when changing your code.

Mocks should be used primarily when you would use a stub, but need to verify multiple more specific behaviors on it.

For example, here's how we could verify a more specific database saving scenario using a mock:

```
it('should pass object with correct values to save only once', function() {
  var info = { name: 'test' };
  var expectedUser = {
    name: info.name,
    nameLowercase: info.name.toLowerCase()
  };
  var database = sinon.mock(Database);
  database.expects('save').once().withArgs(expectedUser);

  setupNewUser(info, function() { });

  database.verify();
  database.restore();
});
```

Note that, with a mock, we define our expectations up front. Normally, the expectations would come last in the form of an assert function call. With a mock, we define it directly on the mocked function, and then only call verify in the end.

In this test, we're using `once` and `withArgs` to define a mock which checks both the number of calls and the arguments given. If we use a stub, checking multiple conditions require multiple assertions, which can be a code smell.

Because of this convenience in declaring multiple conditions for the mock, it's easy to go overboard. We can easily make the conditions for the mock more specific than is needed, which can make the test harder to understand and easy to break. This is also one of the reasons to avoid multiple assertions, so keep this in mind when using mocks.

Best Practices and Tips

Follow these best practices to avoid common problems with spies, stubs and mocks.

Use sinon.test Whenever Possible

When you use spies, stubs or mocks, wrap your test function in `sinon.test`. This allows you to use Sinon's automatic clean-up functionality. Without it, if your test fails before your test-doubles are cleaned up, it can cause a cascading failure - more test failures resulting from the initial failure. Cascading failures can easily mask the real source of the problem, so we want to avoid them where possible.

Using `sinon.test` eliminates this case of cascading failures. Here's one of the tests we wrote earlier:

```
it('should call save once', function() {
  var save = sinon.spy(Database, 'save');

  setupNewUser({ name: 'test' }, function() { });

  save.restore();
  sinon.assert.calledOnce(save);
});
```

If `setupNewUser` threw an exception in this test, that would mean the spy would never get cleaned up, which would wreak havoc in any following tests.

We can avoid this by using `sinon.test` as follows:

```
it('should call save once', sinon.test(function() {
  var save = this.spy(Database, 'save');

  setupNewUser({ name: 'test' }, function() { });
```

```
  sinon.assert.calledOnce(save);
}));
```

Note the three differences: in the first line, we wrap the test function with `sinon.test`. In the second line, we use `this.spy` instead of `sinon.spy`. And lastly, we removed the `save.restore` call, as it's now being cleaned up automatically.

You can make use of this mechanism with all three test doubles:

- sinon.spy becomes this.spy
- sinon.stub becomes this.stub
- sinon.mock becomes this.mock

Async Tests with sinon.test

You may need to disable fake timers for async tests when using `sinon.test`. This is a potential source of confusion when using Mocha's asynchronous tests together with `sinon.test`.

To make a test asynchronous with Mocha, you can add an extra parameter into the test function:

```
it('should do something async', function(done) {
```

This can break when combined with `sinon.test`:

```
it('should do something async', sinon.test(function(done) {
```

Combining these can cause the test to fail for no apparent reason, displaying a message about the test timing out. This is caused by Sinon's fake timers which are enabled by default for tests wrapped with `sinon.test`, so you'll need to disable them.

This can be fixed by changing `sinon.config` somewhere in your test code or in a configuration file loaded with your tests:

```
sinon.config = {
  useFakeTimers: false
};
```

`sinon.config` controls the default behavior of some functions like `sinon.test`. It also has some other available options.

Create Shared Stubs in beforeEach

If you need to replace a certain function with a stub in all of your tests, consider stubbing it out in a `beforeEach` hook. For example, all of our tests were using a test-double for `Database.save`, so we could do the following:

```
describe('Something', function() {
  var save;
  beforeEach(function() {
    save = sinon.stub(Database, 'save');
  });

  afterEach(function() {
    save.restore();
  });

  it('should do something', function() {
    //you can use the stub in tests by accessing the variable
    save.yields('something');
  });
});
```

Make sure to also add an `afterEach` and clean up the stub. Without it, the stub may be left in place and it may cause problems in other tests.

Word of caution though: If you put too much stuff into `beforeEach`, it may become difficult to tell what's going on in your tests. Especially if some of your tests are not using the values defined in `beforeEach`, you should consider using another approach. Using a test helper function is a good alternative.

Use Test Helper Functions to Arrange Code

Sometimes your test code becomes messy. But remember - test code is just like any other code!

You can refactor your test code to make use of helper functions, just like you would refactor any other code where you see duplication.

A common scenario is where you have several stubs or other values being initialized in the beginning of your test:

```
it('should do something complicated', function() {
  var x = sinon.stub();
  var y = something;
  var z = moreStuffGoingOn();

  // ...
})

it('should do something else complicated', function() {
  var x = sinon.stub();
  var y = something;
  var z = moreStuffGoingOn();
```

```
  // ...
});
```

In this case, you can create a function within your test file and move the code there:

```
function setupComplicatedThing() {
  var x = sinon.stub();
  var y = something;
  var z = moreStuffGoingOn();
}

it('should do something complicated', function() {
  setupComplicatedThing();

  // ...
})

it('should do something else complicated', function() {
  setupComplicatedThing();

  // ...
});
```

By doing this, you can make your tests both easier to understand and easier to maintain.

Checking the Order of Function Calls or Values Being Set

If you need to check that certain functions are called in order, you can use spies or stubs together with `sinon.assert.callOrder`:

```
var a = sinon.spy();
var b = sinon.spy();

a();
b();

sinon.assert.callOrder(a, b);
```

If you need to check that a certain value is set before a function is called, you can use the third parameter of stub to insert an assertion into the stub:

```
var object = { };
var expectedValue = 'something';
var func = sinon.stub(example, 'func', function() {
  assert.equal(object.value, expectedValue);
```

```
});

doSomethingWithObject(object);

sinon.assert.calledOnce(func);
```

The assertion within the stub ensures the value is set correctly before the stubbed function is called. Remember to also include a `sinon.assert.calledOnce` check to ensure the stub gets called. Without it, your test will not fail when the stub is not called.

In Closing

Sinon is a very powerful tool to have in your testing toolset. You can use it to make almost any kind of code testable - even if it's completely crazy!

> When in doubt, stub it out.

Well, maybe not, but you can solve most testing problems with stubs.

Please send me any comments or feedback on this ebook to jani@codeutopia.net

Thanks for reading

– Jani