Extracted from:

Test-Driving JavaScript Applications

Rapid, Confident, Maintainable Code

The Pragmatic Bookshelf
Raleigh, North Carolina

The Pragmatic Programmers

Test-Driving JavaScript Applications

Rapid, Confident, Maintainable Code

Venkat Subramaniam

Edited by Jacquelyn Carter

Test-Driving JavaScript Applications

Rapid, Confident, Maintainable Code

Venkat Subramaniam

The Pragmatic Bookshelf

Raleigh, North Carolina

The team that produced this book includes:

Jacquelyn Carter (editor)
Potomac Indexing, LLC (index)
Liz Welch (copyedit)
Gilson Graphics (layout)
Janet Furlow (producer)

Printed in the United States of America.
ISBN-13: 978-1-68050-174-2
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—October 2016

CHAPTER 3

Test Asynchrony

JavaScript libraries are filled with async functions. For example, to read a file, give the filename to the *fs* library and it'll get back later with data or an error. Likewise, talking to services involves async functions—we can't escape asynchrony. Let's embrace asynchrony with automated tests.

Writing and executing automated tests for asynchronous functions poses some challenges. A call to a synchronous function blocks and waits for the result. However, a call to an asynchronous function is nonblocking and the result or error will arrive later through callbacks or promises. To write automated verifications for async functions, you need to go beyond the techniques you learned in the previous chapter.

Let's focus now on testing asynchronous functions and temporarily set aside our desires for test-first design. Doing so will help us to more easily explore the nitty-gritty details of writing tests for asynchronous functions. We will write tests that run within Node.js and tests that run within browsers. To meet this goal, we'll start with pre-created asynchronous functions. The techniques you learn here will help you in the later chapters when we take on the test-first approach.

To verify async functions, you have to deal with two issues. Since results may not arrive immediately after you make the calls, you'll have to make the tests wait for the results to show up in the callbacks or through promises. Also, you have to decide how long to wait for the results—the timeout. Set the timeout too low, tests fail prematurely. Set it too long, and you'll wait longer than necessary if a function were to become nonresponsive. These efforts may seem overwhelming, but tools and techniques are available to effectively deal with these challenges.

We'll first explore testing functions that use callbacks; then we'll dig into promises. Let's get started.

Server-Side Callbacks

A caller of a synchronous function blocks until it gets the result or the intended action is completed. But calls to asynchronous functions are nonblocking. A caller of such functions typically sends one or more callback functions as extra parameters and moves on. Asynchronous functions that use callbacks eventually invoke one or more callbacks when the processing is completed. It's through these callbacks that they indirectly send a response back to the caller. The difference in the nature of the functions poses some challenges from the point of view of testing.

Due to the nature of the call, a test for a synchronous function automatically waits for the result to arrive. The test for an asynchronous function needs to induce a wait since the call is nonblocking. Introducing a sleep or delay in execution will not suffice. For one, it will make tests run slower. Furthermore, there is no guarantee the asynchronous function has responded within the duration of delay. Rather, we need a reliable mechanism to test these functions. Let's explore this with an example.

The function we'll test reads a given file and returns the number of lines in it. It currently has no tests—we'll write the tests for it together.

We'll first write a test naively for the asynchronous function, like we did for the synchronous functions. You'll learn from that experience the reasons why asynchronous testing needs to be different. Then we'll write a proper asynchronous test and make it pass. Finally we'll write a negative test for the asynchronous function as well. Let's get started.

A Naive Attempt to Test

Let's first approach testing an asynchronous function like we approached tests of synchronous functions. This exercise will get you familiar with the code to be tested and also help you see why a different testing approach is necessary.

Switch to a new `files` project by changing to the `tdjsa/async/files` directory in your workspace. The tools we'll use in this project are Mocha and Chai. Let's get them installed right away by running the `npm install` command in the current project directory—this will be our routine each time we step into a project directory and see a `package.json` file in the directory.

Take a look at the code in the src/files.js file. You'll see a function that takes a filename and eventually returns, through callbacks, either the number of lines in the file or an error.

async/files/src/files.js
```
var fs = require('fs');

var linesCount = function(fileName, callback, onError) {
  var processFile = function(err, data) {
    if(err) {
      onError('unable to open file ' + fileName);
    } else {
      callback(data.toString().split('\n').length);
    }
  };

  fs.readFile(fileName, processFile);
};

module.exports = linesCount;
```

Let's write a test for this function, using the approach we've used for synchronous functions so far. Open the empty file test/files-test.js in the current files project in the workspace and enter the following code:

```
var expect = require('chai').expect;
var linesCount = require('../src/files');

describe('test server-side callback', function() {
  it('should return correct lines count for a valid file', function() {
  //Good try, but this will not actually work
    var callback = function(count) {
      expect(count).to.be.eql(-2319);
    };

    linesCount('src/files.js', callback);
  });
});
```

We want to verify that the linesCount function correctly returns the number of lines in a given file. But for that, we need to pass a filename as a parameter. It's hard to predict what files are available on different systems. But we both know that the source code file exists on our system, so we will use that filename as a parameter to the linesCount function.

At the top of files-test.js, the file containing the code under test is loaded and the function within that file is assigned to the variable named linesCount. The test calls the linesCount function, sends it the name of the source file as the filename, and registers a callback function. Within the callback the test asserts that the count received as the parameter is -2319. We know that the count of number of lines can't be negative—clearly the test is broken. If all went well this test should report a failure, but we'll see what happens.

Let's run the test with the command npm test. When run, the test passes instead of failing, as we see in the output:

```
test server-side callback

  ✓ should return correct lines count for a valid file

1 passing (5ms)
```

From the automation point of view, there's nothing worse than tests that lie. Tests should be highly deterministic and should pass only for the right reasons. This test called the linesCount function, passed a filename and a callback, and immediately exited. There is nothing in the test that tells Mocha to wait for the callback to be executed. So, the test did not actually wait to exercise the assert that's within the callback when it's eventually called. It would be nice if tests failed when there are no asserts in their execution path, but that's not the case, as we saw.

We need to tell Mocha not to assume that the test is complete when it exits out of the test function. We need the tool to wait for the execution of the callback function, and the assert within that, before it can declare the test to have passed or failed.

This example serves as a good reminder to make each test fail first and then, with minimum code, make it pass. Let's make the test fail first.

Writing an Asynchronous Test

Tests written using Mocha can include a parameter that can be used to signal the actual completion of tests. When a test exits, Mocha will wait for the signal that the test is actually completed. If an assertion fails before this signal is received or if the signal is not received within a reasonable time, it will declare the test as failure.

Let's edit the test so that exiting the test does not imply completion of the test:

```
it('should return correct lines count for a valid file', function(done) {
  var callback = function(count) {
    expect(count).to.be.eql(-2319);
  };

  linesCount('src/files.js', callback);
});
```

Unlike the previous tests, this test takes on a parameter—it can be named anything you like, but done is quite logical. It's a way to signal to Mocha when a test is really complete. In other words, if a parameter is present, Mocha does not assume a test is done when it completes the test function. Instead, it waits for a signal through that parameter to declare that the test is done. Whether the function being tested is synchronous or asynchronous, this technique may be used to verify results in callbacks.

Let's run the test now and see it go up in flames, like it should:

```
test server-side callback

  1) should return correct lines count for a valid file

0 passing (9ms)
1 failing

1) test server-side callback
  should return correct lines count for a valid file:

    Uncaught AssertionError: expected 15 to deeply equal -2319
    + expected - actual

    -15
    +-2319

    at callback (test/files-test.js:8:27)
    at processFile (src/files.js:8:7)
    at FSReqWrap.readFileAfterClose [as oncomplete] (fs.js:404:3)
```

To make the test pass, let's change -2319 to 15 in the body of the callback. As an astute reader you may protest, "Wait, won't this test break if the file is changed?" Yes it will, but let's keep our eyes on asynchrony at this time; we'll focus on other concerns later in the book. Here's the change to the callback with the correct expected value:

```
it('should return correct lines count for a valid file', function(done) {
  var callback = function(count) {
    expect(count).to.be.eql(15);
  };

  linesCount('src/files.js', callback);
});
```

The callback verifies the value passed, but when npm test is run again, Mocha reports

```
test server-side callback

  1) should return correct lines count for a valid file

0 passing (2s)
1 failing
```

```
1) test server-side callback
  should return correct lines count for a valid file:
   Error: timeout of 2000ms exceeded.
     Ensure the done() callback is being called in this test.
```

Even though the assert in the callback passed, the test failed after a 2 second wait—the default timeout. That's because the test never signaled its completion. To fix that, we'll add a call to done() at the end of the callback function. You'll soon see how to change the default timeout. Here's the modified test:

```
it('should return correct lines count for a valid file', function(done) {
  var callback = function(count) {
    expect(count).to.be.eql(15);
    done();
  };

  linesCount('src/files.js', callback);
});
```

Let's now run npm test and see the test passing, but this time for the right reasons:

```
test server-side callback

  ✓ should return correct lines count for a valid file

1 passing (7ms)
```

Let's write another asynchronous test to gain practice, but this time we'll make it a negative test.

A Negative Asynchronous Test

The test we wrote covers only the happy path of the function. The behavior of the function when an invalid file is given needs to be verified as well. Let's write a test for that, again in the test/files-test.js file.

```
it('should report error for an invalid file name', function(done) {
  var onError = function(error) {
    expect(error).to.be.eql('unable to open file src/flies.js');
    done();
  };
  linesCount('src/flies.js', undefined, onError);
});
```

The second test sends an invalid filename—flies instead of files—to the function under test. The test assumes such a misnamed file doesn't exist. Such tests are troublesome—they're brittle and may fail if the dependency changes. Again, we'll address that concern later on.

The second argument to the function is undefined since it will not be used during this call. The new third argument is a callback that verifies the error details. Once again, this test takes in the done parameter and the callback signals the completion of the test through that. Let's run the test and Mocha should report great success:

```
test server-side callback
  ✓ should return correct lines count for a valid file
  ✓ should report error for an invalid file name
2 passing (8ms)
```

Joe asks:

Does the 3-As Pattern Apply to Async Tests?

Good tests follow the 3-As pattern mentioned in *Create Positive Tests*, on page ?. Asynchronous tests are no exception. The arrange part is followed by the act part, but the assert part is embedded within the callbacks. Even though it may not be apparent, the execution flow of the tests follows the sequence of arrange, act, and assert.

Mocha relies on the parameter of its test functions to know when a test on an asynchronous function is completed. Now that you know how to test asynchronous functions running on the server side, let's explore doing the same for the client side. Along the way, you'll pick up a few more tricks.