TAYLOR JONES
SOFTWARE CHEF AT IZEA

Test-Driven Development for JavaScript

About the Author.

Taylor is Software Chef at IZEA, slowly simmering in the Florida heat. He is a contributor to the Codeship blog and mainly writes about how to transition to a microservice structure with your software team.

Codeship is a fully customizable hosted Continuous Integration and Delivery platform that helps you build, test, and deploy web applications fast and with confidence.

Learn more about Codeship here.

Test-Driven Development for JavaScript

JavaScript is handsdown the strangest language I've ever had to test. Its also one of the most popular ones out there right now. The influx of JavaScript developers tells us that a lot of modern-day web development is starting to focus more and more on the frontend.

This trend is interesting because it represents a shift in development that we've never seen before. Our applications are being composed of more and more JavaScript. However, are we effectively testing all of this newfound client-side code?

I believe that we must follow a more robust methodology to test our growing JavaScript applications.

ABOUT THIS EBOOK

In this eBook we will explore the idea of practicing Test-Driven Development (TDD) for client-side JavaScript.

What's So Different About Testing JavaScript?

In web development, we can separate our concerns into two sections: the client-side (stuff that happens in a user's web browser) and the server-side (things that happen on our application servers).

When it comes to testing, we've historically covered our server-side code with unit and functional tests and our client-side with integration tests. We tend to give a lot of thought and investment into server-side tests and simply verify the results with an integration test on the client-side.

However, server-side code is and was never the complete picture of web development. What the user sees on the client-side is often what has the most impact. The most impressive technical features mean nothing if the view layer is messed up!

Integration tests are helpful and effective when we're only concerned with what shows up on the webpage. It says nothing about what goes on inside the client-side code processing; it only shows the output of such logic.

There was a period of time where integration tests could really serve as proper test coverage for what's going on in the client-side of our applications. However, with more and more applications having bigger shares of

JavaScript, we've got to come up with a better way of testing the client-side other than: is x showing up on the page?

To start things off, let's look at what's familiar about TDD on the client-side. It turns out a lot of things aren't that different after all!

JavaScript TDD Isn't That Different

No matter how experienced in TDD you are, don't panic! If you've done TDD before, applying it to client-side JavaScript isn't that different on a high level. With TDD, we want to stay true to a cycle of three steps:

- Write tests that reflect the expected behavior of your code.
- Write code to make those tests pass.
- (optional) Refactor and ensure the tests still pass.

Let's run through a quick example in EmberJS.

A unit TDD example with EmberJS

Ember is really cool because it comes with a lot of great tools for testing right out of the box. If you run something as simple as: `ember g model user`, you'll be gifted with two important files: `app/models/user.js` and `tests/unit/user-test.js`. If you run ember test, all of your tests related to user.js should pass.

Currently, our files look something like this:

user.js

```
CODE
1  import DS from 'ember-data';
2  export default DS.Model.extend({
3  });
```

user-test.js

```
CODE
1  import { moduleForModel, test } from 'ember-qunit';
2
3  moduleForModel('user', 'Unit | Model | user', {
4    // Specify the other units that are required for this test.
5    needs: []
6  });
7
8  test('it exists', function(assert) {
9    let model = this.subject();
10   assert.ok(!!model);
11 });
```

We want to expand our model to do two things: have a name and be able to retrieve a metric called "karma" for a user. Let's write up some tests to reflect these desires:

user-test.js

```
CODE
1  import { moduleForModel, test } from 'ember-qunit';
2
3  moduleForModel('user', 'Unit | Model | user', {
4    // Specify the other units that are required for this test.
5  });
6
7  test('it exists', function(assert) {
8    let model = this.subject();
9    assert.ok(!!model);
10 });
```

CODE

```
11  test('must contain a name', function(assert) {
12    let model = this.subject({ name: 'testing name'});
13    assert.equal(model.get('name'), 'testing name');
14  });
15
16  // Note: Karma should be the combo of upvotes and downvotes against a user
17  test('should be able to retrieve the karma of a user', function(assert) {
18    let model = this.subject({ name: 'karma user', upvotes: 10, downvotes: 5});
19    assert.equal(model.get('karma'), 5);
20  });
```

If we've changed nothing to **user.js**, we'll find that our Ember test suite will now fail gracefully. Next up, we'll work to make these tests actually pass.

user.js

CODE

```
1  import Ember from 'ember';
2  import DS from 'ember-data';
3
4  export default DS.Model.extend({
5    name: DS.attr('string'),
6    upvotes: DS.attr('number'),
7    downvotes: DS.attr('number'),
8    karma: Ember.computed( function() {
9      return this.get('upvotes') - this.get('downvotes');
10   })
11 });
```

Now our tests and code are good to go!

With just a few simple steps, we were able to run through the TDD process and develop some simple but stable code! I chose not to refactor on this run through since my changes were pretty simple. However, the more complex

our code becomes, the more there becomes a need for refactoring at every step.

Other frameworks and JavaScript code

Not every JavaScript framework or codebase has a built-in test runner like EmberJS (or EmberCLI) has. There still are a lot of excellent tools out there like Mocha and Jasmine. However there's still a long way to go when it comes to JavaScript test tooling.

While client-side unit testing doesn't look that different from other kinds of testing we've experienced, there are some noticeable differences to how we should approach it.

Javascript TDD Is Different

When implementing TDD on the client-side, we'll need to divide our testing mindset into client-side and server-side concerns.

Client-side JavaScript cannot query a database. It can make calls for a query or query data presented on the client-side. It cannot, however, directly fetch and insert into a database. This is important because we want to assume that the data being given to the client-side (in our tests) is gospel. We should treat the server-side data being sent as an external dependency on our test.

In many ways, we're splitting up our applications into two contained testing areas: Server-side tests are concerned with things that go on inside the server; client-side tests are concerned with stuff that happens in the web browser. In order to achieve proper test coverage of these two aspects, the amount of tests we're hauling along is going to increase. In some cases, we could have all of the following:

- Client-side integration tests
- Server-side functional tests
- Server-side unit tests
- Client-side unit tests
- Client-side functional tests

Adding a lot more test weight to your application might not seem like the most attractive option. Yet, its a consequence of having complex processing on the server- and client-sides of your application. There's a lot more space to be covered, and understanding more of what's going on is essential!

What Does All This Mean?

If you're implementing TDD practices for the first time, JavaScript is a great place to start. The basics of TDD aren't too hard to get down, but it takes some practice to grasp what aspects of the domain you should test.

If you're starting to apply previous knowledge of TDD to your frontend JavaScript, don't forget what you know. Embrace your historical knowledge and learn to understand the domain layout of the client-side.

Ultimately, JavaScript test coverage is a hard thing to fully visualize and understand. The way we leverage the language to help us develop apps is ever changing as well. You might be adding a lot more tests to your test suite, but the understanding and coverage you'll have is worth it!

Share this Codeship Guide

More Codeship Resources.

EBOOKS

Running a MEAN Web App in Docker Containers on AWS.

In this eBook, we break down all the steps it takes to successfully install and run a web application built on the MEAN stack.

Download this eBook

EBOOKS

Understanding the Docker Ecosystem.

In this book you will learn about technologies in the Docker Ecosystem and how they fit together.

Download this eBook

EBOOKS

A Beginner's Guide to the Dockerfile.

In this eBook we will take a look at the Dockerfile – the building block of Docker images and containers.

Download this eBook

CODESHIP

Share this Codeship Guide

About Codeship.

Codeship is a hosted Continuous Integration service that fits all your needs. Codeship Basic provides pre-installed dependencies and a simple setup UI that let you incorporate CI and CD in only minutes. Codeship Pro has native Docker support and gives you full control of your CI and CD setup while providing the convenience of a hosted solution.

Codeship Basic

A simple out-of-the-box Continuous Integration service that just works.

Starting at $0/month.


Works out of the box


Preinstalled CI dependencies


Optimized hosted infrastructure


Quick & simple setup

LEARN MORE

Codeship Pro

A fully customizable hosted Continuous Integration service.

Starting at $0/month.


Customizability & Full Autonomy


Local CLI tool


Dedicated single-tenant instances


Deploy anywhere

LEARN MORE