VUE.JS ENTERPRISE CRASH COURSE - PDF

LESSON 3

UNIT TESTING

VUE.JS DEVELOPERS

Lesson 3

Unit Testing

Welcome to the third lesson of this crash course on enterprise web development with Vue.js.

One of the important aspects of enterprise apps is that they should be reliable and easy for developers to add to.

For that reason it's important to provide tests for the key functionality.

The two types of tests you should consider for a full-stack Vue app are unit tests and browser automation tests.

In this lesson, I'll show you how to add unit tests to a full-stack Vue app, first for the frontend components, and secondly, for the API endpoints.

Unit testing

```
const operations = {
  add(a, b) {
    return a + b;
  }
};

it("should add numbers correctly", () => {
  expect(operations.add(2,3)).toBe(5);
});
```

The idea of a "unit" in testing, is that code is broken down into atomic parts which provide a fixed output for a given input.

Method

For example, say we have an object called `operations`. It has an `add` method which adds two numbers.

Test function

We can unit test that method by declaring a test function,

Assertion

and then we can write our test logic where we'll supply an input to the method, in this case the arguments `2` and `3`, and assert that the output will be `5`.

Jest - JavaScript test runner

Terminal

```
$ vue add unit-jest
```

In order to run a unit test, we'll need a test runner.

One of the most popular test runners for JavaScript right now is Jest. It's easy to add Jest to a project using the Vue CLI plugin, and we can use Jest for both our component and API tests.

Unit testing components

tests/unit/client/App.spec.js

```
import App from "@/App";

describe("App.vue", () => {
  it("should render correctly", () => {
    // testing logic
  });
});
```

In the client-side app, the "units" of our tests will be our components. To test them, we'll normally create a separate test file for each component.

File transform

One handy feature of Jest is that it can transform source files. This means we can import a Vue single-file component and get back an object.

Describe

The typical way to create a Jest component test is to start with a describe block announcing the component you're testing.

it

You then include one or more `it` blocks in which you'll declare a test, for example, `it`, meaning the component, `should render correctly`.

Test logic

And, finally, we write the logic for our test.

Vue Test Utils

tests/unit/client/App.spec.js

```
import { mount } from "vue-test-utils";
import App from "@/App";

describe("App.vue", () => {
  it("should render correctly", () => {
    const wrapper = mount(App);
    // INPUT: e.g. wrapper.setProps(...)
    // OUTPUT: e.g. expect(wrapper.html()).toBe(...)
  });
});
```

You'll almost always want to use the Vue Test Utils library when testing Vue components.

This wraps a component in a convenient API allowing you to easily work with it in isolation from the Vue app to which it belongs.

Mount

One approach to this is to use the `mount` function of Vue Test Utils.

Wrapper

You can then call that function to create a wrapper around the component you're testing.

API methods

This wrapper allows you to easily work with the inputs and outputs of the component. For example, you could use the `setProps` method to add a particular set of props, then then assert something about the subsequent output.

Snapshot tests

tests/unit/client/App.vue

```
describe("App.vue", () => {
  it("should render correctly", () => {
    const wrapper = mount(App);
    expect(wrapper).toMatchSnapshot();
  });
});
```

tests/unit/client/snapshots/App.snap

```
<div id="app">
  <nav />
  <router-view />
  <footer />
</div>
```

Speaking of which, Jest offers an easy way to test rendered output with a feature called "snapshots".

Snapshot file

The first time the test runs, the rendered output is captured and stored in a file. This snapshot is assumed to be the correct state of the component.

Then, each time the test runs again, the output is compared to this master snapshot to see if the component is outputting what it should.

API endpoint unit tests

tests/unit/server/items.spec.js

```
const request = require("supertest");

...

describe("GET /items", () => {
  it("should get all items", async () => {
    const res = await request(app)
      .get("/items")
      .expect(200);
    expect(res.body.length).toBe(seedItems.length);
  });
});
```

Moving now to the backend of app, the units we want to test here are the API endpoints.

For example, we may want to check if we send a POST request to the items endpoint, that a new item is successfully created.

Supertest

Just like we needed Vue Test Utils to create a friendly API around our components, we can use a package called `supertest` to create an API around our Express app.

The supertest API provides a method for each HTTP verb allowing you to easily create requests and get the response back as a return.

In this example, we're testing the GET /items endpoint, which should return all the items in the database.

Assertions

We can also use supertest to make assertions. For example, we can assert that we get a 200 status in the response.

Body content

We can also assert the body content of the response. For example, we can assert that the number of items in the response is the same as the number of items that are seeded in the database.

In the next video...(E2E testing)

That wraps up our overview of unit testing, so in the next video we'll be talking about the other kind of testing you commonly find in enterprise web app development which is...

end-to-end testing. Thanks for watching and I'll see you in the next video.