Mock External Module Dependencies

In this lesson, we will learn about External Modules Dependencies.

WE'LL COVER THE FOLLOWING

Testing Methods

What should we test in methods? That's a question that we had when we started doing unit tests. Everything comes down to **testing what that method does**. This means that we need to *avoid calls to any dependency*; so, we'll need to mock them.

Let's add a submit event to the form in the `Form.vue` component that we created in the last chapter:

```
require('./check-versions')()

process.env.NODE_ENV = 'production'

var ora = require('ora')
var rm = require('rimraf')
var path = require('path')
var chalk = require('chalk')
var webpack = require('webpack')
var config = require('../config')
var webpackConfig = require('./webpack.prod.conf')

var spinner = ora('building for production...')
spinner.start()

rm(path.join(config.build.assetsRoot, config.build.assetsSubDirectory), err => {
  if (err) throw err
  webpack(webpackConfig, function (err, stats) {
    spinner.stop()
    if (err) throw err
    process.stdout.write(stats.toString({
```

```
      colors: true,
      modules: false,
      children: false,
      chunks: false,
      chunkModules: false
    }) + '\n\n')

    console.log(chalk.cyan('  Build complete.\n'))
    console.log(chalk.yellow(
      '  Tip: built files are meant to be served over an HTTP server.\n' +
      '  Opening index.html over file:// won\'t work.\n'
    ))
  })
})
```

The `.prevent` modifier is just a convenient way to call `event.preventDefault()` in order to prevent reloading the page. Now make some modifications to call an API and store the result, by adding a `results` array to the data and an `onSubmit` method:

```
require('./check-versions')()

process.env.NODE_ENV = 'production'

var ora = require('ora')
var rm = require('rimraf')
var path = require('path')
var chalk = require('chalk')
var webpack = require('webpack')
var config = require('../config')
var webpackConfig = require('./webpack.prod.conf')

var spinner = ora('building for production...')
spinner.start()

rm(path.join(config.build.assetsRoot, config.build.assetsSubDirectory), err => {
  if (err) throw err
  webpack(webpackConfig, function (err, stats) {
    spinner.stop()
    if (err) throw err
    process.stdout.write(stats.toString({
      colors: true,
      modules: false,
      children: false,
      chunks: false,
      chunkModules: false
    }) + '\n\n')

    console.log(chalk.cyan('  Build complete.\n'))
    console.log(chalk.yellow(
      '  Tip: built files are meant to be served over an HTTP server.\n' +
      '  Opening index.html over file:// won\'t work.\n'
    ))
  })
})
```

The method is using axios to perform an HTTP call to the “reqres” endpoint of

The method is using *axios* to perform an HTTP call to the "posts" endpoint of *jsonplaceholder*, which is just a RESTful API for this kind of example. With the

`q` query parameter, we can search for posts using the `value` provided as parameter.

For testing the `onSubmit` method:

- We don't want to call the `axios.get` actual method
- We want to check that it is calling axios (but not the real one) and that it returns a promise
- We want to check that the promise callback is setting `this.results` to the promise result

This is probably one of the hardest things to test when you have external dependencies. Now, what we need to do is to **mock the external dependencies**.

Mock External Module Dependencies

Jest provides a really great mocking system that allows you to mock everything in quite a convenient way. You don't need any extra libraries for that. We have already seen `jest.spyOn` and `jest.fn` for spying and creating stub functions, although they are not enough for this case.

We need to mock the whole `axios` module. Here's where `jest.mock` comes onto the stage. It allows us to easily mock module dependencies by writing this at the top of our file:

```
jest.mock("dependency-path", implementationFunction);
```

You must know that `jest.mock` **is hoisted**, which means that it will be placed at the top. So:

```
jest.mock("something", jest.fn);
import foo from "bar";
// ...
```

Is equivalent to:

```
import foo from "bar";
```

```
jest.mock("something", jest.fn); // this will end up above all imports and everything
// ...
```

Let's write the mock for *axios* at the top of the `Form.test.js` test file and the corresponding test case:

Form.test.js

```
jest.mock("axios", () => ({
  get: jest.fn()
}));

import { shallowMount } from "@vue/test-utils";
import Form from "../src/components/Form";
import axios from "axios"; // axios here is the mock from above!

// ...

it("Calls axios.get", () => {
  cmp.vm.onSubmit("an");
  expect(axios.get).toBeCalledWith(
    "https://jsonplaceholder.typicode.com/posts?q=an"
  );
});
```

This is great; we're indeed mocking *axios*, so neither the original *axios* nor any HTTP call is called. And, we're even checking by using `toBeCalledWith` that it's been called with the right parameters. But we're still missing something: ***we're not checking that it returns a promise***.

First, we need to make sure that our mocked `axios.get` method returns a promise. `jest.fn` accepts a factory function as a parameter, so we can use it to define its implementation:

```
jest.mock("axios", () => ({
  get: jest.fn(() => Promise.resolve({ data: 3 }))
}));
```

But still, we cannot access the promise, because we're not returning it.

Returning the Promise

In testing, it is a good practice to return something from a function when possible as it makes testing much easier. Let's do it then in the `onSubmit` method of the `Form.vue` component:

Form.vue

```
export default {
  methods: {
    // ...
    onSubmit(value) {
      const getPromise = axios.get(
        "https://jsonplaceholder.typicode.com/posts?q=" + value
      );

      getPromise.then(results => {
        this.results = results.data;
      });

      return getPromise;
    }
  }
};
```

Then we can use the very clean ES2017 `async/await` syntax in the test to check the promised result:

```
it("Calls axios.get and checks promise result", async () => {
  const result = await cmp.vm.onSubmit("an");

  expect(result).toEqual({ data: [3] });
  expect(cmp.vm.results).toEqual([3]);
  expect(axios.get).toBeCalledWith(
    "https://jsonplaceholder.typicode.com/posts?q=an"
  );
});
```

You can see that not only do we check the promised result but also that the `results` internal state of the component is updated as expected, by doing `expect(cmp.vm.results).toEqual([3])`.

In the next lesson, we'll keep the mocks externalized.