Ferran Negre Follow
Software Engineer at @Callstackio • React • React Native • Jest • Created @audioprofiles (Android) …
May 18, 2016 · 4 min read

Unit testing with Jest: Redux + async actions + fetch

I have been using Jest since its beginnings. I won't lie, the path has been painful: Things like updating from version 0.x.0 to version 0.y.0 (specially to version 0.4.0) was particularly terrible for my tests. Furthermore, I struggled myself more than once with the way Jest works and I even bothered the community with questions like 1, 2, and 3.

But stay with me, Jest has also good things that I am going to show you in a moment. I hope somebody will find this particularly useful because, in my humble opinion, the biggest drawback of Jest is its lack of examples.

Recently, I've been writing an app with the React/Redux combo. It uses an external API and I decided to use a fetch *polyfill* in order to make my network requests. Following the *Redux-way*, I've used Async Action Creators in combination with redux-thunk. Here is how my action looks like:

```
import 'whatwg-fetch';

...

export function fetchData(id) {
  return (dispatch, getState) => {
    if(getState().id === id)) {
      return; // No need to fetch
    }
    dispatch(requestData(id));
    return fetch('/api/data/' + id)
      .then(checkResponse)
```

Pretty net, no?

Ok, now, let’s test it step by step based on the WritingTests from Redux docs. Going with Jest and, first things first, if you read its docs too, you will notice that its core part is:

> *Automatically mocks dependencies for you when running your tests.*

Thus, we actually need to manually *unmock* some stuff here:

```
jest.unmock('../api');
jest.unmock('redux-mock-store');
jest.unmock('redux-thunk');
```

The first *unmock* is the module where our async action is (we want to test the real module, not a mocked version of it). The second *unmock* is *redux-mock-store* which is a small package that will help us mocking a redux store (so its getState function too). The third *unmock* is the actual redux-thunk (we want it to be the real one for the async action to work as expected).

We also import everything we need:

```
import { fetchData } from '../api';
import configureMockStore from 'redux-mock-store'
import thunk from 'redux-thunk'

const middlewares = [ thunk ];
const mockStore = configureMockStore(middlewares);
```

And I am also going to write a helper method to mock a fetch response (we don’t need nock as the Redux Writing Test example):

```
const mockResponse = (status, statusText, response) => {
  return new window.Response(response, {
    status: status,
    statusText: statusText,
    headers: {
      'Content-type': 'application/json'
    }
  });
};
```

After that, let's actually write our first *it*, well, in this case, pit (since the test returns a promise):

```
pit('calls request and success actions if the fetch re

  window.fetch = jest.fn().mockImplementation(() =>
    Promise.resolve(mockResponse(200, null, '{"id":"12

  return store.dispatch(fetchData(1234))
    .then(() => {
      const expectedActions = store.getActions();
      expect(expectedActions.length).toBe(2);
```

Do you see what I am doing? I don't care what the real fetch does, I will reinvent it with *jest.fn().mockImplementation* and force it to return the response I want to test with (in this case a successful one). And with that, I assert that two and only two actions have been called and that they are *FETCH_DATA_REQUEST* and *FETCH_DATA_SUCCESS*.

Very similar with the error test case:

```
pit('calls request and failure actions if the fetch re

  window.fetch = jest.fn().mockImplementation(() => Pr
    400, 'Test Error', '{"status":400, "statusText": T

  return store.dispatch(fetchData(1234))
    .then(() => {
      const expectedActions = store.getActions();
      expect(expectedActions.length).toBe(2);
      expect(expectedActions[0]).toEqual({type: types.
```

Well done! Let's go one step further, remember this?

```
if(getState().id === id)) {
      return; // No need to fetch
}
```

Well, that might be the most basic caching system in the world. Let's see how can we easily test that too:

```
it('does check if we already fetched that id and only
  const store = mockStore({id: 1234, isFetching: false
  window.fetch = jest.fn().mockImplementation(() => Pr

  store.dispatch(fetchData(1234)); // Same id
  expect(window.fetch).not.toBeCalled();

  store.dispatch(fetchData(1234 + 1)); // Different id
```

See, I am mocking a new store that already has the id 1234 'fetched', so if we dispatch our action, *window.fetch* should not be called since is the same id. Instead, it should be called if it is a different one. As you can see, I don't even care what fetch returns here so I just make it to return a Promise.

With that, we got our async action fully covered. But wait! It's not over, there is a bonus and this is **very** important. Do you remember the import at the beginning of our example action code?

```
import 'whatwg-fetch'
```

If you or your team forget to use the *polyfill* version of fetch, Chrome or FF will still work and you won't notice it! But poor the user who tries to use your website on Safari if you know what I mean. For that reason, we will write a unit test to avoid such case too. Look at the *polyfill* code, it has:

```
self.fetch.polyfill = true
```

Awesome, we can write a test like:

```
it('makes sure window.fetch is the one from the polyfill',
() => {
  expect(window.fetch.polyfill).toBe(true);
});
```

Splendid Jest!

Notice that we don't need to *unmock 'whatwg-fetch'* since whatwg-fetch is not really a module and just injects fetch into the global window object.

And that's it my friends, this is actually everything I wanted to show you. If you find a similar example with Jest, please share it because it does get really hard to find them. Thanks to the Redux docs for the great explanation of how we can test those async actions and for the mocha example.

By the way, this is my first one in Medium, I hope you enjoyed, thanks for reading and hopefully, *write* you soon!