Writing Tests for Apollo Client in React

OCTOBER 19, 2018 BY ROBIN WIERUCH - EDIT THIS POST

Interested in reading this tutorial as one of many chapters in my GraphQL book? Checkout the entire The Road to GraphQL book that teaches you to become a fullstack developer with JavaScript.

This tutorial is part 3 of 3 in this series.

Part 1: A minimal Apollo Client in React Application
Part 2: Mocking a GraphQL Server for Apollo Client

In a previous application, you have learned how to mock a GraphQL server in different ways when having Apollo Client as GraphQL client in your React application. The following application shows you how you can take this knowledge to the next level for writing tests for your Apollo Client queries and

mutations. So far, the Apollo Client instance can be mocked, but one unsolved question keeps popping up: How to test Apollo Client in a React application?

SEPARATION OF APOLLO CLIENT AND MOCK CLIENT

If you have the previous application with the mocked Apollo Client at your disposal, you can start writing tests with it. Otherwise, you find the application with the mocking of the Apollo Client in this GitHub repository. Let's start to separate both concerns, the actual Apollo Client and the mocked Apollo Client, before using the former for the actual application and the latter for testing the application. The Apollo Client setup for the React application can be done in a couple of steps for the GitHub client application:

```
import { ApolloClient } from 'apollo-client';
import { HttpLink } from 'apollo-link-http';
import { InMemoryCache } from 'apollo-cache-inmemory';

const cache = new InMemoryCache();

const GITHUB_BASE_URL = 'https://api.github.com/graphql';

const httpLink = new HttpLink({
  uri: GITHUB_BASE_URL,
  headers: {
    authorization: `Bearer ${
      process.env.REACT_APP_GITHUB_PERSONAL_ACCESS_TOKEN
    }`,
  },
});

export default new ApolloClient({
  link: httpLink,
  cache,
});
```

Afterward, the Apollo Client instance can be imported in your React root component for using it in React Apollo's Provider component:

```
import React from 'react';
import ReactDOM from 'react-dom';
import { ApolloProvider } from 'react-apollo';

import App from './App';
import client from './client';
```

```
ReactDOM.render(
  <ApolloProvider client={client}>
    <App />
  </ApolloProvider>,
  document.getElementById('root'),
);
```

That's the part for the actual application. So what about the mocked Apollo Client from the previous application? You can implement it in another file which is only used by your tests later on.

```
import { ApolloClient } from 'apollo-client';
import { InMemoryCache } from 'apollo-cache-inmemory';
import { SchemaLink } from 'apollo-link-schema';
import { makeExecutableSchema } from 'graphql-tools';

import { schema, resolvers } from './schema';

const cache = new InMemoryCache();

const executableSchema = makeExecutableSchema({
  typeDefs: schema,
  resolvers,
  resolverValidationOptions: {
    requireResolversForResolveType: false,
  },
});

export default new ApolloClient({
  link: new SchemaLink({ schema: executableSchema }),
  cache,
});
```

In this case, a client-side schema is used (and no GraphQL introspection) to define the executable GraphQL schema with its resolvers. Whereas the resolvers deliver all the mock data for your tests, the schema itself defines all the GraphQL types and their structure. You have implemented both, client-side schema and resolvers, in the previous application where you have mocked the Apollo Client.

Now you have an actual Apollo Client instance for your application in one file and the mocked Apollo Client in another file. The latter should be used in the following sections for testing your interaction between React and the mocked Apollo Client.

Since the application is set up with create-react-app, it already comes with Jest as testing framework. Jest can be used as test runner and assertion library at the same time. You can read more about it in this comprehensive React testing tutorial which covers Jest but also other libraries such as Sinon and Enzyme, which you are going to use in the following sections, for React applications. In order to have access to Enzyme and Sinon next to Jest, you have to install them as development dependencies to your project.

```
npm install enzyme enzyme-adapter-react-16 sinon --save-dev
```

So what are these libraries, Enzyme and Sinon, including Jest doing for us to test a React application with Apollo Client?

- **Jest**: Since this application already comes with Jest, Jest is used as test runner (e.g. tests can be started from the command line, tests can be grouped in test suites and test cases) and assertion library (e.g. making expectations such as "to equal" or "to be" between result and expected result).
- **Enzyme**: The library is used for rendering React components in tests. Afterward, components rendered by Enzyme have an API to access them (e.g. find all input HTML nodes in the component) to conduct assertions with them. In addition, it is possible to simulate events such as a click on a button element or writing in an input element.
- **Sinon**: The library is used to spy, stub and mock functions. It is often used to make expectations on how many times a function is called, with which arguments a function is called or to return dummy output from a stubbed/mocked function.

Before you can start with using Enzyme in your Jest test files (by default all files which end with the *test.js* suffix are executed as tests by the Jest test runner), you have to setup Enzyme with the recent React version. You can do this in a separate file which you only have to import once in your test files. As alternative, you can do the Enzyme setup in your test files too.

```
import Adapter from 'enzyme-adapter-react-16';
import { configure } from 'enzyme';

configure({ adapter: new Adapter() });
```

Now you are ready to write your tests with Jest, Enzyme and Sinon for your React components which are using Apollo Client for GraphQL queries and mutations. In case of the tests, it will be the mocked Apollo Client and not the actual Apollo Client connecting to the real API.

TESTING A APOLLO CLIENT MUTATION IN REACT

If you continued with the previous application, most of your React component implementation should be in the *src/App.js* file. So what about writing the tests for a couple of its React components in a *src/App.test.js* file next to it? In the following, you will test the execution of a GraphQL mutation which is conducted with the mocked Apollo Client. In your *src/App.js* file, the Star component is a perfect candidate to be tested in isolation. It only receives an identifier as prop which is used for the GraphQL mutation when clicking the button in the component at some point. In order to make the component

accessible in other files (e.g. test file), you have to export it. Along with it you have to export the mutation to make assertions with it in your test file.

```
...

const Star = ({ id }) => (
  <Mutation mutation={STAR_REPOSITORY} variables={{ id }}>
    {starRepository => (
      <button type="button" onClick={starRepository}>
        Star
      </button>
    )}
  </Mutation>
);

...

export { Star, STAR_REPOSITORY };

export default App;
```

Now comes the exciting part: writing a test for a GraphQL mutation. In your *src/App.test.js* file, import all the parts which are needed for the testing. If you are wondering about the *src/test/setup.js* file, it is the part where you had to set up Enzyme with its adapter to React from the previous section.

```
import React from 'react';

import './test/setup';

import {
  Star,
  STAR_REPOSITORY,
} from './App';

describe('Star', () => {
  it('calls the mutate method on Apollo Client', () => {

  });
});
```

Now you can use the mocked Apollo Client instance and React Apollo's Provider component to render the Star component with Enzyme.

```
import React from 'react';
import { ApolloProvider } from 'react-apollo';
import { mount } from 'enzyme';

import './test/setup';
```

```
import clientMock from './test/client-mock';

import {
  Star,
  STAR_REPOSITORY,
} from './App';

describe('Star', () => {
  it('calls the mutate method on Apollo Client', () => {
    const wrapper = mount(
      <ApolloProvider client={clientMock}>
        <Star id={'1'} />
      </ApolloProvider>,
    );
  });
});
```

If you revisit your resolvers where you have set up all the mocked data in the previous application, it should have a repository with the `id` property, because this is the repository you are going to star in the test.

```
...

describe('Star', () => {
  it('calls the mutate method on Apollo Client', () => {
    const wrapper = mount(
      <ApolloProvider client={clientMock}>
        <Star id={'1'} />
      </ApolloProvider>,
    );

    wrapper.find('button').simulate('click');
  });
});
```

The mutation should be called by Apollo Client in the test now. If you are not sure about what's happening when clicking the button, check again the Star component. But how do you find out that the mutation is actually executed in your test? That's where Sinon comes into play to spy methods of your Apollo Client instance.

```
import React from 'react';
import { ApolloProvider } from 'react-apollo';
import { mount } from 'enzyme';
import { spy } from 'sinon';

...

describe('Star', () => {
  it('calls the mutate method on Apollo Client', () => {
```

```
    spy(clientMock, 'mutate');

    const wrapper = mount(
      <ApolloProvider client={clientMock}>
        <Star id={'1'} />
      </ApolloProvider>,
    );

    wrapper.find('button').simulate('click');

    expect(clientMock.mutate.calledOnce).toEqual(true);

    clientMock.mutate.restore();
  });
});
```

The spy on the `mutate()` method on the Apollo Client instance wraps the method itself into a testable function. That's why it has the `calledOnce` property at its disposal to conduct assertions with it. So basically after clicking the button, you want to assert that the mutation was executed once by the mocked Apollo Client. Afterward, you remove the spy again with the `restore()` method on the spied method for leaving your tests without any footprint. Once you run your tests with `npm test`, it should give you a green output for the previous test. That's basically it for testing a GraphQL mutation which is executed by Apollo Client. Since Apollo Client itself is tested by the Apollo Client package, you can be assured that the executed method reaches the network and thus your GraphQL API eventually.

But the testing doesn't stop here. Since you have used a spy on the `mutate()` method of the Apollo Client which result in a network request eventually, you can use the spy to make further assertions. Basically you have access to all the props which you have passed to the Mutation component in the arguments of the call of the spy.

```
...

describe('Star', () => {
  it('calls the mutate method on Apollo Client', () => {
    spy(clientMock, 'mutate');

    const wrapper = mount(
      <ApolloProvider client={clientMock}>
        <Star id={'1'} />
      </ApolloProvider>,
    );

    wrapper.find('button').simulate('click');

    expect(clientMock.mutate.calledOnce).toEqual(true);

    expect(clientMock.mutate.getCall(0).args[0].variables).toEqual({
      id: '1',
    });
```

```
    expect(clientMock.mutate.getCall(0).args[0].mutation).toEqual(
      STAR_REPOSITORY,
    );

    clientMock.mutate.restore();
  });
});
```

That's it for testing the GraphQL mutation in Apollo Client in a React application. In the next section, you will test a GraphQL query. There you will also see how the result of the GraphQL operation can be tested. Since the mutation result from this section wasn't used in the Star component, it wasn't necessary to test it.

TESTING A APOLLO CLIENT QUERY IN REACT

This time you are going to test the App component itself which queries a list of items (repositories). The list of items is defined as mocked data in your client-side resolvers which are used for the mocked Apollo Client. Therefore, make sure the App component is exported from the *src/App.js* file, which should already be there with a default export, along with its query, which is used in the Query component, to make them accessible for your test file.

```
...

const App = () => (
  <Query query={GET_REPOSITORIES_OF_ORGANIZATION}>
    {({ data: { organization }, loading }) => {
      if (loading || !organization) {
        return <div>Loading ...</div>;
      }

      return (
        <Repositories repositories={organization.repositories} />
      );
    }}
  </Query>
);

...

export {
  Star,
  STAR_REPOSITORY,
  GET_REPOSITORIES_OF_ORGANIZATION,
};

export default App;
```

In your *src/App.test.js* file, import these things and create a new test suite with two test cases. Whereas the former test case is similar to the mutation test case from before, the latter test case should make an assertion about the rendered component after the queried (mocked) data arrived and thus is used to display something with it.

```
...

import App, {
  Star,
  STAR_REPOSITORY,
  GET_REPOSITORIES_OF_ORGANIZATION,
} from './App';

describe('Star', () => {
  ...
});

describe('App', () => {
  it('calls the query method on Apollo Client', () => {

  });

  it('renders correctly after the query method on Apollo Client executed', (

  });
});
```

The former test case for the query is similar to the mutation test case and thus can be tested in a similar way:

```
...

describe('App', () => {
  it('calls the query method on Apollo Client', () => {
    spy(clientMock, 'watchQuery');

    const wrapper = mount(
      <ApolloProvider client={clientMock}>
        <App />
      </ApolloProvider>,
    );

    expect(clientMock.watchQuery.calledOnce).toEqual(true);

    expect(clientMock.watchQuery.getCall(0).args[0].query).toEqual(
      GET_REPOSITORIES_OF_ORGANIZATION,
    );

    clientMock.watchQuery.restore();
```

```
  });

  it('renders correctly after the query method on Apollo Client executed', (

  });
});
```

Internally in Apollo Client, not the `query()` method is called, but the `watchQuery()` method. Hence you have to spy this method for making assertions on it. Once you start your tests again, they should turn out to be green. So what about the second test case?

```
...

describe('App', () => {
  it('calls the query method on Apollo Client', () => {
    ...
  });

  it('renders correctly after the query method on Apollo Client executed', (
    const wrapper = mount(
      <ApolloProvider client={clientMock}>
        <App />
      </ApolloProvider>,
    );

    expect(
      wrapper
        .find('Repositories')
        .find('RepositoryList')
        .find('li').length,
    ).toEqual(2);

    expect(
      wrapper.find('Repositories').props().repositories.edges[0].node
        .id,
    ).toEqual('1');

    expect(
      wrapper.find('Repositories').props().repositories.edges[1].node
        .id,
    ).toEqual('2');
  });
});
```

The second test case differs from the previous query test and the mutation test too. Yet it isn't any more spectacular. It could have been conducted for the mutation as well, but there was no result used from the GraphQL mutation after executing the mutation in the first place. However, the test verifies for you whether everything is rendered accordingly to the mocked query result from the Query

component's child function. Since the resolver returns two repositories, you can test the rendered HTML elements and the props that are passed to child components. The final application can be found in this GitHub repository.

APOLLO CLIENT TESTING UTILITIES

Whereas the previous application has shown you how to mock Apollo Client for your GraphQL server, the last two sections have shown you how to write tests for your React components which are using the Query and Mutation components from React Apollo. In the case of the GraphQL query, you have tested both: the query itself and the query result. Most of the time the shown testing patterns should be sufficient. Nevertheless, this section shows you a couple of additional techniques which you can use to test your GraphQL operations in React.

Stubbing the fetch API

If you are not able to create a mock for your GraphQL server, you can intercept the actual request made by your Apollo Client instance and stub the result instead. At this time, Apollo Client is using the native fetch API as default to conduct HTTP requests under the hood. That's why you can use it as your advantage to stub the fetch API with Sinon. The following code snippets demonstrate how it could work:

```
import sinon from 'sinon';
import { print } from 'graphql/language/printer';

const mockData = [
  { id: '1', title: 'Foo' },
  { id: '2', title: 'Bar' },
];
const uri = 'https://api.github.com/graphql';

// Promise implementation for a returned result from the fetch API
const promise = Promise.resolve({
  text: () => Promise.resolve(JSON.stringify({ data: mockData })),
});

sinon
  .stub(global, 'fetch')
  .withArgs(uri)
  .returns(promise);
```

That's basically your test setup for stubbing your GraphQL API endpoint and having control over the returned data by having a promise in place. Then it should be possible to resolve the promise in your test and expect the correct data from the stubbed fetch API.

```
test('query result of Query component', done => {
  // using the real Apollo Client instance
  const wrapper = mount(
    <ApolloProvider client={client}>
      <App />
    </ApolloProvider>,
  );

  expect(wrapper.find('[data-test-id="loading"]')).toHaveLength(1);

  promise.then().then(() => {
    setImmediate(() => {
      wrapper.update();

      expect(wrapper.find('li')).toHaveLength(2);

      expect(wrapper.find('li').at(0).text())
        .toEqual(mockData[0].title);

      done();
    });
  });
});
```

This way, you are able to stub your GraphQL query, but also get more fine-grained control over the resolving promise(s) and the different rendering states (e.g. loading, finish) of your React component. You can even stub your request more fine-grained by providing the arguments that are expected in the native fetch API request when using Apollo Client.

```
import sinon from 'sinon';
import { print } from 'graphql/language/printer';

const mockData = [
  { id: '1', title: 'Foo' },
  { id: '2', title: 'Bar' },
];
const uri = 'https://api.github.com/graphql';
const mockInput = {
  query: print(GET_REPOSITORIES_OF_ORGANIZATION),
};

const promise = Promise.resolve({
  text: () => Promise.resolve(JSON.stringify({ data: mockData })),
});

const args = {
  method: 'POST',
  headers: { accept: '*/*', 'content-type': 'application/json' },
  credentials: undefined,
  body: JSON.stringify({
    operationName: mockInput.operationName || null,
    variables: mockInput.variables || {},
```

```
    query: print(mockInput.query),
  }),
};

sinon
  .stub(global, 'fetch')
  .withArgs(uri, args)
  .returns(promise);
```

Keep in mind, that you can provide Apollo Client something else (e.g. axios) than the default fetch API. Then you would have to stub this (e.g. axios) instead of the fetch API. In addition, the structure of the arguments (here `args`) can change in the future, because they are internally provided by Apollo Client to the fetch API and you don't have any control over their structure.

Testing the Children Function in a Render Prop Component

Both components, the Query and the Mutation component, come with the render props pattern where you use a child function. The children function has access to the query/mutation results, but also to the function which calls the mutation itself. The following example will show you how you can get access to the child function of a render prop component (Mutation) in order to make assertions (with a spy) on it. You will use Jest to manipulate the Mutation component and Sinon to give you a spy for the mutation function which is then available in the children's arguments.

```
import React from 'react';
import * as ReactApollo from 'react-apollo';
import sinon from 'sinon';

const spy = sinon.spy();

ReactApollo.Mutation = ({ mutation, variables, children }) => (
  <div>{children(() => spy({ mutation, variables }))}</div>
);

jest.setMock('react-apollo', ReactApollo);
```

That's again basically your test setup to spy the mutation function from every Mutation component that is used in your tested components. In this scenario, you mock the Mutation component from the React Apollo package. The spy is used for the mutation function. Afterward, when testing a component which has the Mutation component, you can use the spy to verify that it was called. For instance, in case of the Star component:

```
const Star = ({ id }) => (
  <Mutation mutation={STAR_REPOSITORY} variables={{ id }}>
    {starRepository => (
      <button type="button" onClick={starRepository}>
        Star
```

```
        </button>
      )}
    </Mutation>
  );
```

You can verify that the mutation was called (also with the correct arguments if you wish to do so) after the button has been clicked:

```
test('interaction with mutation function from the Mutation component', () =>
  const wrapper = mount(
    <ApolloProvider client={client}>
      <Star id={'1'} />
    </ApolloProvider>,
  );

  wrapper.find('button').simulate('click');

  expect(sinonSpy.calledOnce).toEqual(true);
});
```

That's how you get access to the `starRepository()` function in your tests from the arguments of the child function of the Mutation component. If you want to advance the previous test setup, you can even provide a mutation result as second argument to your child function and verify the rendered output in your Mutation component (only when the mutation result is used there) after the button has been clicked.

```
import React from 'react';
import * as ReactApollo from 'react-apollo';
import sinon from 'sinon';

const mockData = { id: '1', starred: true };
const spy = sinon.spy();

ReactApollo.Mutation = ({ mutation, variables, children }) => (
  <div>{children(() => spy({ mutation, variables }), mockData)}</div>
);

jest.setMock('react-apollo', ReactApollo);
```

That's how you get full control over the Mutation component (but also Query component) in your tests. Jest enables you to mock the render prop components.

All the previous techniques, stubbing the GraphQL API and mocking the Query/Mutation components, can be found in this experimental library for testing Apollo Client. It is not an official library, so I

wouldn't advice you to use it, but it is a great place to check again the previous techniques and its usage in the example application which can be found in the GitHub repository as well.

. . .

After all, keep in mind that the testing setup and how you use the tools at your hand (Jest, Enzyme, Sinon) is up to you. It doesn't have to be a Apollo specific testing library. In the previous sections, you have learned how you can test your GraphQL queries and mutations in React when having a mocked Apollo Client. Everything you need is a test runner (Jest or Mocha), an assertion library (Jest or Chai), and a library to spy/stub/mock functions (Sinon). Afterward, you can test whether your spied/stubbed methods of the mocked Apollo Client are called. Furthermore, the client-side resolvers return mock data which can be used for testing the rendered output when using the query or mutation results in your component. In conclusion, if you follow these straight forward test patterns, you don't need to reinvent the wheel every time you test Apollo Client with its Mutation and Query components in React.

Continue Reading: A apollo-link-state Tutorial for Local State in React

Continue Reading: How to build a GraphQL client library for React

KEEP READING ABOUT REACT

MOCKING A GRAPHQL SERVER FOR APOLLO CLIENT

Often you run into the case where you have to mock your GraphQL server for your GraphQL client application. It can be for testing your GraphQL client or when your GraphQL server is not (always…

REACT WITH APOLLO AND GRAPHQL TUTORIAL

In this tutorial, you will learn how to combine React with GraphQL in your application

using Apollo. The Apollo toolset can be used to create a GraphQL client, GraphQL server, and other complementary...

TAKE PART

NEVER MISS AN ARTICLE ABOUT WEB DEVELOPMENT AND JAVASCRIPT.

- ✔ Join 50.000+ Developers
- ✔ Learn Web Development
- ✔ Learn JavaScript
- ✔ Access Tutorials, eBooks and Courses
- ✔ Personal Development as a Software Engineer

SUBSCRIBE >

View our Privacy Policy.

PORTFOLIO

Online Courses

Open Source

Tutorials

ABOUT

About me

What I use

How to work with me

How to support me