Mocking a GraphQL Server for Apollo Client

MAY 28, 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 2 of 3 in this series.

Part 1: A minimal Apollo Client in React Application
Part 3: Writing Tests for Apollo Client in React

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) available for development. Then it comes in handy to know how to mock your GraphQL server. The following tutorial will show you how to do it for Apollo Client which is used in a React application.

The following sections are split up into two parts. The first part will show you how to mock a GraphQL server with a client-side implemented GraphQL schema. You may wonder: *When would you do it this way?* For instance, it happens when you are not able to get a *schema.json* file from your GraphQL server or when you are not able to run a GraphQL introspection against your GraphQL server. So this approach can be used when the schema from the GraphQL server, which you are trying to mock, is out of your hands. The second part shows you the alternative way, when you are able to access the schema from your GraphQL server, by using a GraphQL introspection.

In order to get you started, clone this minimal React application from GitHub and follow its installation instructions or use the minimal Apollo Client in React boilerplate project from a previous section. Afterward, get to know the source code of the project and run it on the command line with `npm start`. It is a minimal React application which consumes the official GitHub GraphQL API by using Apollo Client.

HOW TO MOCK A GRAPHQL SERVER FROM A CLIENT-SIDE SCHEMA

In the following, the *src/index.js* file is the only part you are going to focus on. That's the place where the Apollo Client instance with its HTTP link and cache is instantiated and where you will hook-in the mocking of your GraphQL server. You will need a Apollo Link called Apollo Link Schema to provide a client-side GraphQL schema to your Apollo Client setup. In addition, you need GraphQL Tools helper functions to create the client-sided schema in the first place. Therefore, install the packages on the command line for your project:

```
npm install apollo-link-schema graphql-tools --save
```

Next, import the SchemaLink along with your other Apollo Client dependencies. Apollo Client's HttpLink is not needed for the first part, because it is replaced entirely by the SchemaLink. In the second part of the sections it is needed though.

```
import React from 'react';
import ReactDOM from 'react-dom';

import { ApolloProvider } from 'react-apollo';
import { ApolloClient } from 'apollo-client';
import { InMemoryCache } from 'apollo-cache-inmemory';
import { SchemaLink } from 'apollo-link-schema';

import App from './App';

const cache = new InMemoryCache();
```

```
const link = ...

const client = new ApolloClient({
  link,
  cache,
});

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

Everything is in place except for the `link` property which is mandatory for the Apollo Client constructor. Since you have imported the SchemaLink class, you can use it to create a client-sided GraphQL schema by using the `makeExecutableSchema()` function.

```
import React from 'react';
import ReactDOM from 'react-dom';

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

import App from './App';

const cache = new InMemoryCache();

const typeDefs = ...

const resolvers = ...

const executableSchema = makeExecutableSchema({
  typeDefs,
  resolvers,
});

const link = new SchemaLink({ schema: executableSchema });

const client = new ApolloClient({
  link,
  cache,
});

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

```
);
```

Perhaps you are familiar with the function to generate a GraphQL schema, because it is used for Apollo Server on the Node.js server-side to generate your GraphQL schema from GraphQL types and resolvers. You will implement a small GraphQL schema with those types and resolvers in the next part to mimic the small part you are using from the GitHub GraphQL API in your application.

Let's start with the GraphQL type definitions. The one GraphQL query you are using in your *src/App.js* is retrieving an organization with its repositories based on a `login` string type which identifies the organization.

```
const typeDefs = `
  type Query {
    organization(login: String!): Organization!
  }
`;
```

The query returns an object of the type `Organization` which has GraphQL scalar types (name and url) but also an object type (RepositoryConnection) for the repositories. Since the repositories are not a plain list of repositories, but follow one of the opinionated structures for paginated lists in GraphQL, the type structure is a bit more complex by using a list of RepositoryEdge types. However, in the end there is a `Repository` type represented as node which has only scalar types and thus is the leaf of query. You can double check the query structure by looking at the query from the *src/App.js* file to make more sense of it.

```
const typeDefs = `
  type Query {
    organization(login: String!): Organization!
  }

  type Organization {
    name: String!
    url: String!
    repositories: RepositoryConnection!
  }

  type RepositoryConnection {
    edges: [RepositoryEdge!]!
  }

  type RepositoryEdge {
    node: Repository!
  }

  type Repository {
    id: ID!
    name: String!
```

```
    url: String!
    viewerHasStarred: Boolean!
  }
`;
```

At this point you may wonder: How to come up with the type names? In this case, it is only important to reconstruct the correct **type structure** from the query you are performing in your application, but not the **type names**. The latter are not relevant and you could come up with your own. However, in this case the correct type names from the GitHub GraphQL API are reflected. In addition, you don't need to reconstruct the whole GitHub GraphQL schema, but only the part you are using in your application.

In the next step, you have to implement the type definitions for the Mutation which is used in the *src/App.js* file. The `addStar` mutation takes an **input type** with the type `AddStarInput` and returns an object type of `AddStarPayload`.

```
const typeDefs = `
  type Query {
    organization(login: String!): Organization!
  }

  type Organization {
    name: String!
    url: String!
    repositories: RepositoryConnection!
  }

  type RepositoryConnection {
    edges: [RepositoryEdge!]!
  }

  type RepositoryEdge {
    node: Repository!
  }

  type Repository {
    id: ID!
    name: String!
    url: String!
    viewerHasStarred: Boolean!
  }

  type Mutation {
    addStar(input: AddStarInput!): AddStarPayload!
  }

  input AddStarInput {
    starrableId: ID!
  }

  type AddStarPayload {
```

```
    starrable: Starrable!
  }
`;
```

Last but not least, the `Starrable` type needs to be defined, because it is already used in the `AddStarPayload` type to return the `starrable` object. It could be a GraphQL type identical to all the types from before. However, in the following you are going to define it as a **GraphQL interface** instead which is used on the Repository type too. Doing it this way, it is possible to associate the entity, which is mutated by the `addStar` mutation, with an entity from the result of the query with the list of repositories. After all, that's how Apollo Client is able to update the cache by resolving the relations between those types by using an `id` and `__typename` from the returned entities from GraphQL queries and mutations.

```
const typeDefs = `
  type Query {
    organization(login: String!): Organization!
  }

  interface Starrable {
    id: ID!
    viewerHasStarred: Boolean!
  }

  type Organization {
    name: String!
    url: String!
    repositories: RepositoryConnection!
  }

  type RepositoryConnection {
    edges: [RepositoryEdge!]!
  }

  type RepositoryEdge {
    node: Repository!
  }

  type Repository implements Starrable {
    id: ID!
    name: String!
    url: String!
    viewerHasStarred: Boolean!
  }

  type Mutation {
    addStar(input: AddStarInput!): AddStarPayload!
  }

  input AddStarInput {
    starrableId: ID!
  }
```

```
  type AddStarPayload {
    starrable: Starrable!
  }
`;
```

That's it for the type definitions. You should have implemented all the GraphQL types that are needed to create a small GraphQL schema which reflects all necessary parts for the used query and mutation from the App component. The complementary part in order to create an executable schema for the Apollo Client are resolvers. You may have used them before for Apollo Link State or Apollo Server. Basically it is the place to define how every field in your GraphQL operations get resolved. Usually the information for the resolvers is taken from a database (Apollo Server) or local state (Apollo Link State), but in this case, it is the place where you simply return mocked data which reflects the schema structure from before.

First, define the resolver for the organization field in your query. It can return the whole object going all the way down to the repositories as nodes in a list. In order to give the mock data a dynamic touch, you can use the `login` argument from the second argument of the resolver function to use it for the mock data. These are all the arguments which are passed into your query (or mutation).



```
const resolvers = {
  Query: {
    organization: (parent, { login }) => ({
      name: login,
      url: `https://github.com/${login}`,
      repositories: {
        edges: [
          {
            node: {
              id: '1',
              name: 'the-road-to-learn-react',
              url: `https://github.com/${login}/the-road-to-learn-react`,
              viewerHasStarred: false,
            },
          },
          {
            node: {
              id: '2',
              name: 'the-road-to-learn-react-chinese',
              url: `https://github.com/${login}/the-road-to-learn-react-chir
              viewerHasStarred: false,
            },
          },
        ],
      },
    }),
  },
};
```

Second, you can define the `addStar` mutation in the Mutation resolver the same way:

```
const resolvers = {
  Query: {
    ...
  },
  Mutation: {
    addStar: (parent, { input }) => ({
      starrable: {
        id: input.starrableId,
        viewerHasStarred: true,
      },
    }),
  },
};
```

And third, you have to define the resolveType for the GraphQL interface that you have defined and implemented for the Repository type before. Since the GraphQL interface is only implemented by one GraphQL type, it can simply return this one GraphQL type. Otherwise, if the interface would be implemented by many types, the resolveType function would have to handle it.

```
const resolvers = {
  Query: {
    ...
  },
  Mutation: {
    ...
  },
  Starrable: {
    __resolveType: () => 'Repository',
  },
};
```

If you would't implement the resolveType, you would get the following error when having the interface implemented as before and when executing the `addStar` mutation: "*Abstract type Starrable must resolve to an Object type at runtime for field AddStarPayload.starrable with value "[object Object]", received "undefined". Either the Starrable type should provide a "resolveType" function or each possible types should provide an "isTypeOf" function.*"

That's it for defining your GraphQL type definitions and schema. Both are used in the `makeExecutableSchema()` function to produce a schema which is used in the SchemaLink constructor. It is the one part which replaced the HttpLink which would have been used to send the GraphQL operations across the network to a actual GraphQL server. Now it should work with the client-side GraphQL schema instead which resolves with the mocked data. Once you start your

application again, you should see the mocked data from the GraphQL query and the mocking of the GraphQL mutation, because the mutation result updates the Apollo Client's Cache.

HOW TO MOCK A GRAPHQL SERVER FROM A INTROSPECTION

The next part of the series shows you the alternative way of creating a mocked GraphQL server by using the GraphQL schema from the actual GraphQL server. Therefore, you don't need to reconstruct the exact schema as you did before. However, the GraphQL schema from the server must be accessible for you in order to pull this off. The common way to retrieve the schema is a GraphQL introspection. In case of GitHub's GraphQL API, you could perform a HTTP GET request against their GraphQL endpoint to retrieve their schema (see instructions). However, there exists a convenient helper function to retrieve the schema with one asynchronous function call: introspectSchema.

```
import React from 'react';
import ReactDOM from 'react-dom';
import { ApolloProvider } from 'react-apollo';
import { ApolloClient } from 'apollo-client';
import { HttpLink } from 'apollo-link-http';
import { InMemoryCache } from 'apollo-cache-inmemory';
import { SchemaLink } from 'apollo-link-schema';
import { printSchema } from 'graphql/utilities/schemaPrinter';
import {
 makeExecutableSchema,
  introspectSchema,
} from 'graphql-tools';

import App from './App';

const resolvers = ...

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
    }`,
  },
});

const schema = introspectSchema(httpLink);

const executableSchema = makeExecutableSchema({
  typeDefs: printSchema(schema),
```

```
  resolvers,
});

const client = new ApolloClient({
  link: new SchemaLink({ schema: executableSchema }),
  cache,
});

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

As you can see, you need to have the working network access to the GraphQL server to retrieve the schema from it. As alternative, the GraphQL schema could be also provided by a *schema.json* file instead of using the GraphQL introspection. A schema file should be used when you don't have network access to your GraphQL server, but you are able to retrieve the *schema.json* file in another way. You will see this approach at the end of this section. Furthermore, the `printSchema()` utility function is used to stringify the schema definitions from the GraphQL server, because the schema is returned as a JavaScript object from the GraphQL server when performing the introspection.

You may have noticed that only the `typeDefs` property has changed for the `makeExecutableSchema()` object argument, because it is the GraphQL schema which comes from your GraphQL server. Thus you don't have to reconstruct the type definitions anymore on your client-side as you did before. You can be assured to have the exact schema on the client-side for mocking your GraphQL server now. However, the second property in the configuration object, the resolvers, are still defined by you on the client-side. It is not possible to retrieve the resolvers from the GraphQL server and it wouldn't make any sense whatsoever, because they are most likely connected to your database on the GraphQL server. That's why you can use the resolver from the previous section to return your mocked data from them for the query and mutation you are using in your application.

Last but not least, since the introspection is an asynchronous request, you need to resolve a promise or use async/await for it:

```
...

const resolvers = ...

async function render() {
  const cache = new InMemoryCache();

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

  const httpLink = ...
```

```
  const schema = await introspectSchema(httpLink);

  const executableSchema = ...

  const client = ...

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

render();
```

Notice how you may receive several similar warnings in your console logs in the browser once you start your application:

```
Type "Starrable" is missing a "resolveType" resolver. Pass false into "resol
```

You should receive similar warnings, but not the warning shown for the `Starrable` type. It is because you have already defined its resolveType function in your resolvers:

```
const resolvers = {
  Query: {
    ...
  },
  Mutation: {
    ...
  },
  Starrable: {
    __resolveType: () => 'Repository',
  },
};
```

All the other GraphQL interfaces from the GraphQL server schema (which is the entire schema and not only a part of it as before) are not resolved. But you don't need to resolve them, because they are not used in your actual GraphQL queries and mutations in your application. Thus, you can deactivate these warnings:

```
async function render() {
  ...

  const executableSchema = makeExecutableSchema({
    typeDefs: printSchema(schema),
```

```
    resolvers,
    resolverValidationOptions: {
      requireResolversForResolveType: false,
    },
  });

  ...
}
```

Now, start your application to verify that your GraphQL operations are still working. The mocking of your GraphQL server should work identical to the mocking from the previous section with the client-sided GraphQL schema. In the previous section, you have defined your client-side schema which mimics/reconstructs the necessary parts used in your application of the GraphQL server schema. It was only important to reconstruct the type definition structure but not necessarily the type names. In the last section though, you have used the actual GraphQL schema from the GraphQL server by using a GraphQL introspection. For both approaches, the resolvers have been the same to mock your data. The final repository can be found on GitHub.

If you cannot use an introspection for your GraphQL server, but need to rely on a *schema.json* file which you have retrieved at another point in time, the following example shows you how to create a client-side schema with a *schema.json* file.

```
import { addResolveFunctionsToSchema } from 'graphql-tools';
import { buildClientSchema } from 'graphql/utilities';

import schema from './schema.json';

const resolvers = ...

const executableSchema = buildClientSchema(schema.data);

addResolveFunctionsToSchema({
  schema: executableSchema,
  resolvers,
});
```

The last function adds your resolver functions to the schema by mutating it directly. This way, you can use the *schema.json* file instead of an introspection for mocking your GraphQL server.

. . .

The last sections have shown you two approaches to create a GraphQL schema which matches (partly) your GraphQL server schema. The reconstructed/fetched schema can be used with client-sided resolvers to mock your data for the Apollo Client. Once the executable schema is created, it is used for the Apollo Client instantiation. It may be also possible to consider one or the other approach for mocking the GraphQL data for testing your React components which depend on a query or

mutation. After all, hopefully the last sections have helped you to mock your GraphQL server data for your GraphQL client-side application.

This tutorial is part 2 of 3 in this series.

Part 1: A minimal Apollo Client in React Application
Part 3: Writing Tests for Apollo Client in React

Show Comments

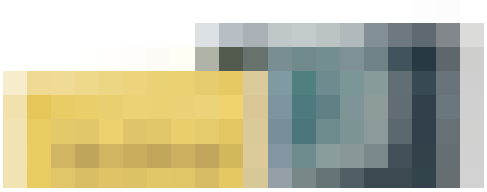
KEEP READING ABOUT GRAPHQL ›

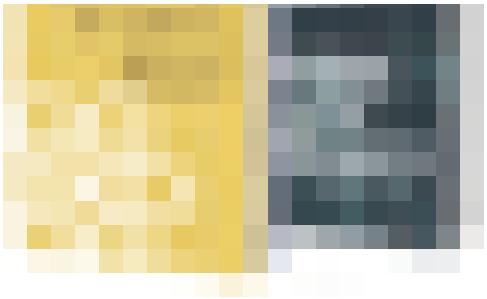
GRAPHQL TUTORIAL FOR BEGINNERS

Before we start to build full-fledged GraphQL applications, on the client- and server-side, let's explore GraphQL with the tools we have installed in the previous sections. You can either use GraphiQL…

WRITING TESTS FOR APOLLO CLIENT IN REACT

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…





THE ROAD TO REACT

Learn React by building real world applications. No setup configuration. No tooling. Plain React in 200+ pages of learning material. Learn React like **50.000+ readers**.

GET THE BOOK

Get it on Amazon.

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

Contact Me Privacy & Terms