YOU DON'T NEED A BACKEND...(YET)

Building Frontend Applications By Mocking Your Entire API With Testing Tools

TypeScript, React, GraphQL and Apollo Client

Jason Lai Follow
Sep 11, 2020 · 5 min read

Photo by Nick Fewings on Unsplash

If you're a frontend developer, you've probably experienced being blocked while waiting on the backend API to be built. You could just hard code the templates and leave the integration until later, or import a JSON file of mock data to use, but neither of these solutions are particularly efficient as they don't allow you to see the result of the actual code that will run in production.

Even if your backend API is ready and you have a local development environment to work with, configuring your application and populating your database can be a laborious process.

Luckily, Apollo Client ships with `MockedProvider`, a component that allows you to test your React components by mocking calls to your GraphQL endpoint. How does this help? The idea came to me one day while I was writing unit tests — if the `MockedProvider` could be used to hydrate my React App in the test environment JSDOM, then why not in the browser DOM?

This probably is not the intended use for `MockedProvider`, as the Apollo documentation only covers implementing it in the context of a test environment. However, I set out to put my theory to the test and found that this method works really well, improving my workflow and efficiency.

Schema first design

The first thing you must do is agree on a schema with the team who will be building the backend. This will save a lot of refactoring later on, and if all

goes according to plan you will be able to swap out the `MockedProvider` for the real one and your application should work seamlessly.

Since Apollo Client uses GraphQL, you can start by writing the GraphQL types. In the example below, we'll be using a simple `Book` and `Author` model.

```
import { SchemaLink } from '@apollo/client/link/schema'
import { makeExecutableSchema } from '@graphql-tools/schema'

import gql from 'graphql-tag'

const typeDefs = gql`
    type Query {
        books: [Book!]!
    }

    type Book {
        id: ID!
        title: String!
        publishDate: String!
        author: Author!
    }

    type Author {
        id: ID!
        firstName: String!
        lastName: String!
        avatarUrl: String!
    }
`
export const schemaLink = new SchemaLink({ schema: makeExecutableSchema({ typeDefs }) })
```

apm-schema.ts hosted with ❤ by GitHub view raw

You can optionally create a client side schema to pass to your instance of Apollo Client, if you want to be able to inspect your schema with the Apollo

Client Dev Tools.

```ts
import { ApolloClient, ApolloLink, InMemoryCache, createHttpLink } from '@apollo/client'
import { schemaLink } from './schema'

export const cache = new InMemoryCache()

const link = createHttpLink({
    /** Your graphql endpoint */
    uri: 'http://localhost:4000/',
})

export const client = new ApolloClient({
    connectToDevTools: true,
    link: ApolloLink.from([link, (schemaLink as unknown) as ApolloLink]),
    cache,
    resolvers: {},
    defaultOptions: {
        query: {
            errorPolicy: 'all',
        },
    },
})
```

apm-client.ts hosted with ❤️ by GitHub view raw

Generate TypeScript types

This example is in TypeScript, so if you are using regular JS you can skip this step. GraphQL Code Generator is an excellent tool to convert GraphQL types into TypeScript types. You can use their web tool or install their CLI tool. Here is the output:

```ts
export type Maybe<T> = T | null
export type Exact<T extends { [key: string]: unknown }> = { [K in keyof T]: T[K] }
/** All built-in and custom scalars, mapped to their actual values */
export type Scalars = {
    ID: string
    String: string
```

```
    String: string
    Boolean: boolean
    Int: number
    Float: number
}

export type Query = {
    __typename?: 'Query'
    books: Array<Book>
}

export type Book = {
    __typename?: 'Book'
    id: Scalars['ID']
    title: Scalars['String']
    publishDate: Scalars['String']
    author: Author
}

export type Author = {
    __typename?: 'Author'
    id: Scalars['ID']
    firstName: Scalars['String']
    lastName: Scalars['String']
    avatarUrl: Scalars['String']
}
```

apm-types.ts hosted with ❤️ by GitHub view raw

Write your GraphQL query

Now that we know the structure of our data, we can write our query to fetch all our books.

```
import gql from 'graphql-tag'

export const GET_BOOKS = gql`
    query getBooks {
        books {
            id
            title
            publishDate
```

```
            author {
                id
                firstName
                lastName
                avatarUrl
            }
        }
    }
`
```

apm-query.ts hosted with ❤️ by GitHub view raw

Note: If you are working with a REST API using `apollo-link-rest` you will need to include the `@type` directive in your queries to patch the `__typename` in your query results. For more info visit https://www.apollographql.com/docs/link/links/rest/#typename-patching

Create your mocks

Apollo's `MockedProvider` expects a `mocks` prop, which is an array of `MockedResponse<T>`. You could hard code these, but it is much less brittle to use a mock generator. We'll be using Factory.ts with Faker, but you can use any libraries that suit your needs. Jose Silva has a good article explaining how to use Factory.ts and Faker with TDD.

For each GraphQL type we will create a factory, which will generate instances of that type. We can also provide the corresponding TypeScript type to the factory to validate the shape of the mock data. Faker will also be used to generate realistic data for each property on our `Book` and `Author`.

Finally we can create our `MockedResponse<Query>` and leverage the `buildList` function on our factory to generate as many books as we want.

```ts
import * as Factory from 'factory.ts'
import faker from 'faker'
import { Book, Author, Query } from './types'
import { MockedResponse } from '@apollo/client/testing'
import { GET_BOOKS } from './queries'

export const AuthorMock = Factory.Sync.makeFactory<Author>({
    __typename: 'Author',
    id: Factory.each(() => faker.random.uuid()),
    firstName: Factory.each(() => faker.name.firstName()),
    lastName: Factory.each(() => faker.name.lastName()),
    avatarUrl: Factory.each(() => faker.image.avatar()),
})

export const BookMock = Factory.Sync.makeFactory<Book>({
    __typename: 'Book',
    id: Factory.each(() => faker.random.uuid()),
    publishDate: Factory.each(() => faker.date.past().toISOString()),
    title: Factory.each(() => faker.random.words()),
    author: Factory.each(() => AuthorMock.build()),
})

export const booksQueryMock: MockedResponse<Query> = {
    request: {
        query: GET_BOOKS,
    },
    result: {
        data: {
            books: BookMock.buildList(10),
        },
    },
}
```

apm-mocks.ts hosted with ❤️ by GitHub view raw

Prepare your Provider

You'll want an easy way to switch between the real `ApolloProvider` and the `MockedProvider`. There are many ways you can do this, for example you might conditionally render it depending on the node env (dev/production).

In this example, I'm just going to use a basic boolean `useMocks` prop to switch between the two.

```tsx
import React from 'react'
import { MockedProvider } from '@apollo/client/testing'
import { ApolloProvider } from '@apollo/client'
import { client } from '../../apollo/config'
import { booksQueryMock } from '../../apollo/mocks'

interface ProviderProps {
    useMocks?: boolean
}

export const Provider: React.FC<ProviderProps> = ({ useMocks, children }) => {
    if (useMocks)
        return (
            <MockedProvider mocks={[booksQueryMock]}>
                <>{children}</>
            </MockedProvider>
        )
    return (
        <ApolloProvider client={client}>
            <>{children}</>
        </ApolloProvider>
    )
}
```

apm-provider.tsx hosted with ❤️ by GitHub view raw

Now wrap the main contents of your App with the `Provider` component, so that it can access the GraphQL data.

```tsx
import React from 'react'
import { Books } from './components/Books/Books'
import { Provider } from './components/Provider/Provider'

export const App = () => {
    return (
        <Provider useMocks>
```

```
            <Books />
          </Provider>
      )
  }
```

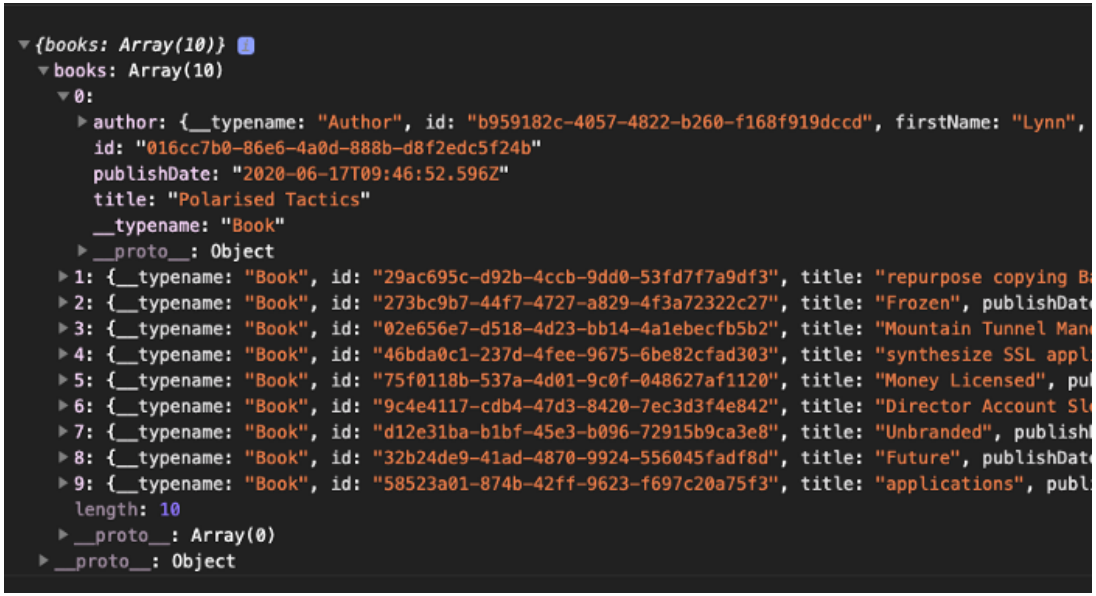
apm-app.tsx hosted with ❤️ by GitHub view raw

Using the mock data in your component

In our `Books` component we are going to use the `useQuery` hook to fetch the mock data from our `MockedProvider`.

```
const { data, loading, error } = useQuery<Query>(GET_BOOKS)
```

If you `console.log(data)` you should be able to see the generated mock data in the browser console.



Now we can display this data in our component — the example below just renders it in a simple list.

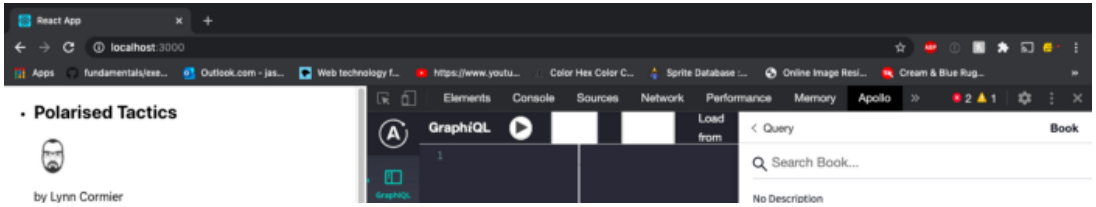
```
import React from 'react'
import { useQuery } from '@apollo/client'
import { Query } from '../../apollo/types'
import { GET_BOOKS } from '../../apollo/queries'

export const Books: React.FC = () => {
    const { data, loading, error } = useQuery<Query>(GET_BOOKS)

    if (error) return <p>{error.message}</p>
    if (loading) return <p>Loading...</p>
    if (data)
        return (
            <ul>
                {data.books.map((book) => (
                    <li key={book.id}>
                        <h2>{`${book.title}`}</h2>
                        <img src={book.author.avatarUrl} width="50px" height="50px" alt=
                        <p>{`by ${book.author.firstName} ${book.author.lastName}`}</p>
                    </li>
                ))}
            </ul>
        )
    return null
}
```

apm-books.tsx hosted with ❤️ by GitHub view raw

You should be able to see your React app hydrated with mock data rendered in the browser.



This is a very basic example, but you can also do more complex mocks with `MockedProvider`, such as paginated queries or mutations, by creating a `MockedResponse<T>` for each `query` or `mutation`. For mutations, you can create a mock factory to generate the mutation payload.

This makes it much easier to build the frontend, especially interactive UI which would normally be dependent on an API response, such as CRUD operations.

You can read more about how to use `MockedProvider` at:

- https://www.apollographql.com/docs/react/development-testing/testing/#mockedprovider
- https://www.apollographql.com/docs/react/development-testing/testing/#testing-mutation-components

Gotchas and common mistakes

When working with `MockedProvider` I've come across a few idiosyncrasies that are good to be aware of to ensure it works properly.

- If your queries or mutations require variables, for example, a pagination limit and offset, the mock you pass to the `MockedProvider` must have the exact same variables defined that the `useQuery` hook will be called with in your component.
- If your mock data objects *do not* have the `__typename` field, then you must *add* the prop `addTypeName={false}` to your `MockedProvider`.
- If your mock data objects *do* have the `__typename` field, you must *remove* the prop `addTypeName={false}` (it is true by default).

Conclusion

Using Apollo's `MockedProvider` is an easy way to build frontend applications without relying on the backend API. There are a lot of benefits to this approach.

- The code you write is the code that will be used in production — your queries and mutations will work exactly the same with both the `MockedProvider` and the `ApolloProvider`. You don't need to comment out chunks of code that handle your API requests and replace it with an imported JSON file to hydrate your UI.
- You can easily test the performance of your application — using your mock factories you can generate hundreds of entries of data to see how your frontend app renders it.
- You can easily do visual checks for edge cases by tweaking your mock factories. For example what if some of the books have extremely long titles?

- You can reuse your mocks factories for unit/integration tests.
- Development time is much quicker as you don't have to wait on API changes to write and test your code, and view the results in the browser.

Although this article focuses specifically on Apollo Client, this approach should work for any API client that provides tools for mocking requests, making development and testing faster and easier.

The full code repository for the examples shown in this article can be found at: https://github.com/laij84/apollo-mocked-provider

React Typescript Apollo Client Testing Tools GraphQL