Creating an App using React and Apollo Graphql

#graphql #react #testing #tutorial

Felipe Armoni Feb 7, 2020 • *Updated on Jun 23, 2020* • 10 min read

Managing and Testing Local State with Apollo Graphql and React (3 Part Series)

1. **Creating an App using React and Apollo Graphql**
2. Use Apollo to Manage the App's Local State
3. Unit Tests with Enzyme and Apollo Graphql

This is a three part tutorial series in which we will build a simple shopping cart app using React and Apollo Graphql. The idea is to build a table in which the user can choose which Rick and Morty action figures he wants to buy.

- Part 1: Creating an App using React and Apollo Graphql.
- Part 2: Use Apollo to manage the app's local state.
- Part 3: Unit Tests with Enzyme and Apollo Graphql

In this first part we will build a table that will show the available action figures, using data from the Rick and Morty API.

> Note: this tutorial assumes that you have a working knowledge of React and Typescript.

You can find the complete code in this repository and the website here: https://komyg.github.io/rm-shop-v1/.

Getting Started

To get started, create a new React App using the CRA - Create React App tool:

```
yarn create react-app rm-shop-v1 --template typescript
```

Material UI

To make our App prettier we will use the Material UI:

```
yarn add @material-ui/core
```

Apollo Graphql

Add the necessary packages from Apollo:

```
yarn add graphql apollo-client apollo-cache-inmemory apollo-link-http apollo-link-error apollo
```

> Note: I chose not to use the Apollo Boost for this tutorial.

Creating the Apollo Client

Now that we've added all the necessary Apollo packages we have to create and initialize the Apollo Client. For this tutorial we are going to connect it to the Rick and Morty API. Click on this link to see the playground with the graphql schema and the available data.

Create a folder called *config* to place the configuration files for our Apollo Client.

> Note: I've chosen to separate each configuration in its own file, because I believe it makes them more readable and clean, however you could use a single file as the Apollo documentation suggests.

Configuring Apollo HTTP Link

The Apollo HTTP link handles the connection between the client app and the Graphql server. Let's create a new config file called: apollo-http-link.ts and add the contents below:

```
import { HttpLink } from 'apollo-link-http';

export const httpLink = new HttpLink({
  uri: 'https://rickandmortyapi.com/graphql',
});
```

The `uri` param is the endpoint that contains the graphql API that we are using.

Configuring the Apollo Error Link

The Apollo Error Link receives and logs any errors that may occur in the Graphql calls. Create a new config file named: *apollo-error-link.ts* and paste the contents below:

```
import { onError } from 'apollo-link-error';

export const errorLink = onError(({ graphQLErrors, networkError, response, operation }) => {
  if (graphQLErrors) {
    for (const error of graphQLErrors) {
      console.error(
        `[GraphQL error]: Message: ${error.message}, Location: ${error.locations}, Path: ${err
        operation,
        response
      );
    }
  }
  if (networkError) {
    console.error(`[Network error]: ${networkError}`, operation, response);
  }
});
```

Notice that the errors here are split into two kinds: **Grapqhl Errors** and **Network Error**. The first kind concerns errors that occur in queries and mutations, such as constraint errors while saving data, incorrect

data formats, etc. The second kind concerns errors that occur in the network and on the POST requests made by the Apollo, such as timeouts or any error code >= 400.

If you have an error reporting tool like Sentry, this is a good place to add them.

Configuring the Local Cache

The `InMemoryCache` is a module that stores the results of the queries and mutations locally so that you don't have to go to the server twice to get the same results. It can also be used for the application state management as we will see in the next parts of this tutorial. For now, create a new file named *apollo-local-cache.ts* and paste these contents:

```
import { InMemoryCache } from 'apollo-cache-inmemory';

export const localCache = new InMemoryCache({
  freezeResults: true,
});
```

The current version of Apollo doesn't require that the cached data be immutable, but we can get a performance boost if we design our cache this way. The `freezeResults` parameter helps us make sure our data is immutable, by throwing an error if we try to change an existing object while running our app in development mode.

Configuring the Apollo Client

Now we will configure the Apollo Client itself and import the configurations we made above. To do this, first create a new file called: *apollo-client.ts* and then paste the contents below:

```
import { ApolloClient } from 'apollo-client';
import { ApolloLink } from 'apollo-link';
import { httpLink } from './apollo-http-link';
import { errorLink } from './apollo-error-link';
import { localCache } from './apollo-local-cache';

export const apolloClient = new ApolloClient({
  link: ApolloLink.from([errorLink, httpLink]),
  connectToDevTools: process.env.NODE_ENV !== 'production',
  cache: localCache,
  assumeImmutableResults: true,
});
```

There is a lot going on this file:

- First we created the `ApolloClient` using its constructor and passed a configuration object to it.
- The first parameter of the configuration is the `ApolloLink`. It works as a chain of Apollo Link Objects that will either:
  - Receive the request, transform it and pass it forward.
  - Receive the request and pass it forward as it is.
  - Receive the request, execute it and return the result to the previous object in the chain.

In our case, we have just two links: the `errorLink` and the `httpLink`. Notice that the order here is important, because we want the `errorLink` to capture any errors that are returned by the `httpLink`, so the `errorLink` must come before it.

You can have as many links as you want, for example: `link: ApolloLink.from([authLink, errorLink, timeoutLink, restLink, httpLink])`. In this example, the `authLink` must come first, because it adds an `Authorization` header that is used to authenticate all requests. Then comes the `errorLink` to capture and log all the errors thrown further down the chain. Then we have the `timeoutLink` that will return an error if the requests made down the chain take longer than a specified period of time. Then we have the `restLink` that is used to make rest calls and finally we have the `httpLink` that handles the Graphql requests.

- The second parameter in the configuration is the `connectToDevTools`. It is active only on non production environments and it allows the Apollo Dev Tools to work.
- The third parameter is the `InMemoryCache`.
- The last parameter is `assumeImmutableResults: true`, it tells the Apollo Client that we intend to make our cached data immutable for a performance gain. Please note that we have to enforce the immutability by ourselves, but the parameter `freezeResults` that we configured on the `InMemoryCache` will help us do this, by throwing an error if we try to change an immutable object while on development.

The Apollo Provider

Now that we have successfully configured the Apollo Client, we have to add the `ApolloProvider` so that all of our components can access it. To do this, we will change our *index.tsx* file to:

```
import { ApolloProvider } from '@apollo/react-hooks';
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
import { apolloClient } from './config/apollo-client';
import * as serviceWorker from './serviceWorker';
```

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

serviceWorker.unregister();
```

Graphql Codegen

The Graphql Codegen is a tool that automatically generates typescript types and classes based on your Grapqhql Schema. It is very useful to ensure type safety.

Configuring the Grapqhl Codegen

The Grapqhl Codegen comes with a CLI tool that helps you create a configuration file. To use it follow these steps:

Install the CLI:

```
yarn add -D @graphql-codegen/cli
```

Execute the wizard:

```
yarn graphql-codegen init
```

Choose the following options:

1. Application built with React.
2. For this tutorial we will use the Rick and Morty Graphql API. Its endpoint is this one: https://rickandmortyapi.com/graphql.
3. Use the default value (`src/**/*.graphql`) for the fragment and operations.
4. Then pick the following plugins:
   - TypeScript
   - TypeScript Operations
   - TypeScript React Apollo
   - Introspection Fragment Matcher
5. Use the default value for the output (`src/generated/graphql.tsx`).
6. Answer *no* when it asks if you want to generate an introspection file.
7. Use the default value for the name of the config file (`codegen.yml`).
8. Type in `gen-graphql` when it asks the name of the script in the *package.json* that will be used to generate the graphql files.

After the wizard finishes, run `yarn install` to install all the necessary plugins added by the Grapqhl Code Gen.

Now, open your *codegen.yml* file and add the `config` param to tell the codegen that we want to use hooks. The final file looks like the one below:

```
overwrite: true
schema: "https://rickandmortyapi.com/graphql"
documents: "src/**/*.graphql"
generates:
  src/generated/graphql.tsx:
    plugins:
      - "typescript"
      - "typescript-operations"
      - "typescript-react-apollo"
      - "fragment-matcher"

    # Add this to use hooks:
    config:
      withHooks: true
```

Creating Our First Query

Now that we have added all necessary packages, let's create our first graphql query to retrieve all characters from the Rick and Morty API. To do this, create a folder called *graphql* inside our *src* folder. Next, create a new file called: *get-characters.query.graphql* and paste the contents below:

```
query GetCharacters {
  characters {
    __typename
    results {
      id
      __typename
      name
```

```
      image
      species
      origin {
        id
        __typename
        name
      }
      location {
        id
        __typename
        name
      }
    }
  }
}
```

> Note: I have added the `id` and `__typename` parameters to our query, even though they are not necessary at this point in our tutorial. I did this, because we will use the `id` field later and because the Apollo Dev Tools needs to have both parameters to show what is in our cache.

Now run the Graphql Codegen to generate the typescript types:

```
yarn gen-graphql
```

If the command ran successfully, you should see that a *graphql.tsx* file was created inside our *generated* folder and that it contains our query.

Displaying the query data

Now that we have our first query, we would like to display its data as a table. To do this, create a new folder called *src/components*.

Creating the character table

Create a new folder: *src/components/character-table* and create the file: *character-table.tsx* inside it. This component will execute our query and display its data inside a table.

Copy and paste the code below into the *character-table.tsx* file:

```
import {
  CircularProgress,
  Paper,
  Table,
  TableBody,
  TableCell,
  TableContainer,
  TableHead,
  TableRow,
  Typography,
} from '@material-ui/core';
import React, { ReactElement } from 'react';
import { useGetCharactersQuery } from '../../generated/graphql';
import CharacterData from '../character-data/character-data';

interface Props {}

export default function CharacterTable(props: Props): ReactElement {
```

```
  // Use hook to retrieve data from the backend
  const { data, loading, error } = useGetCharactersQuery();

  // Query state management
  if (loading) {
    return <CircularProgress />;
  } else if (error) {
    return (
      <Typography variant='h5'>
        Error retrieving data, please reload the page to try again.
      </Typography>
    );
  } else if (!data || !data.characters || !data.characters.results) {
    return (
      <Typography variant='h5'>No data available, please reload the page to try again.</Typogr
    );
  }

  // Display the data
  return (
    <TableContainer component={Paper}>
      <Table>
        <TableHead>
          <TableRow>
            <TableCell>
              <strong>Name</strong>
            </TableCell>
            <TableCell>
              <strong>Species</strong>
            </TableCell>
            <TableCell>
              <strong>Origin</strong>
```

```
              </TableCell>
              <TableCell>
                <strong>Location</strong>
              </TableCell>
            </TableRow>
          </TableHead>
          <TableBody>
            {data.characters.results.map(character => (
              <CharacterData character={character} key={character?.id!} />
            ))}
          </TableBody>
        </Table>
      </TableContainer>
    );
}
```

As you can see there are a lot of things happening in this file:

- First we use the `useGetCharactersQuery` hook. It executes our query as soon as the component finishes mounting. We have also destructured its output using: `{ data, loading, error }`.
- Then we have a state management code in which we display different outputs depending on the query state. For example, we show a progress spinner when the query is retrieving data from the server or we show an error message if something goes wrong or if no data is available.
- Finally, if the query successfully retrieves the character data from the server, then we display it inside the `<Table>` element. Notice that we are mapping the array of characters that is returned by the query into a `<CharacterData />` component that we will create shortly.

- Also notice that we are passing a `key` attribute to the `<CharacterData />` component. This is a good practice to improve React's rendering speed.

Creating the character data

Create a new folder: *src/components/character-data* and create the file: *character-data.tsx* inside it. This component will display our data as a table row.

Copy and paste the code below into the *character-data.tsx* file:

```
import React, { ReactElement } from 'react';
import { Character, Maybe } from '../../generated/graphql';
import {
  TableRow,
  TableCell,
  Box,
  createStyles,
  Theme,
  makeStyles,
  Typography,
} from '@material-ui/core';

interface Props {
  character: Maybe<Character | null>;
}

const useStyles = makeStyles((theme: Theme) =>
  createStyles({
    nameTableCell: {
      display: 'flex',
```

```
      alignItems: 'center',
    },
    characterImg: {
      maxHeight: '3rem',
      width: 'auto',
      borderRadius: '50%',
    },
    characterName: {
      paddingLeft: theme.spacing(2),
    },
  })
);

export default function CharacterData(props: Props): ReactElement {
  const classes = useStyles();

  return (
    <TableRow>
      <TableCell className={classes.nameTableCell}>
        <Box>
          <img src={props.character?.image!} alt='' className={classes.characterImg} />
        </Box>
        <Typography variant='body2' className={classes.characterName}>
          {props.character?.name}
        </Typography>
      </TableCell>
      <TableCell>{props.character?.species}</TableCell>
      <TableCell>{props.character?.origin?.name}</TableCell>
      <TableCell>{props.character?.location?.name}</TableCell>
    </TableRow>
  );
}
```

This component is pretty straight forward. But it is worth noticing that the data type that we are using on the `character` prop was generated by the Graphql Codegen. It indicates that the `character` might be null.

We are using the new Optional Chaining Operator (`?.`) to simplify our code. What it does is return `undefined` if the `character` property is also `undefined` or `null` instead of throwing an error.

And we are also using the Material UI styling tools that rely on jss.

Create a new app component

Finally let's create a new App component to display our data. To start, please delete the *App.tsx*, *App.test.tsx* and *App.css* files. Then create a new folder: *components/app* and create a new *app.tsx* file inside it.

Copy and paste the following code:

```
import React, { ReactElement } from 'react';
import { Container, Box, Theme, makeStyles, createStyles } from '@material-ui/core';
import CharacterTable from '../character-table/character-table';

const useStyles = makeStyles((theme: Theme) =>
  createStyles({
    root: {
      paddingTop: theme.spacing(2),
      paddingBottom: theme.spacing(2),
    },
  })
);
```

```
export default function App(): ReactElement {
  const classes = useStyles();

  return (
    <Container className={classes.root}>
      <Box display='flex' justifyContent='center' alignContent='center'>
        <CharacterTable />
      </Box>
    </Container>
  );
}
```

Notice that we are using the `createStyles` hook to avoid using css. (see: https://material-ui.com/customization/components/#overriding-styles-with-classes).

Update the index.tsx file

Now, update the *index.tsx* file to use our new `App` component:

```
import { ApolloProvider } from '@apollo/react-hooks';
import React from 'react';
import ReactDOM from 'react-dom';
import App from './components/app/app';
import { apolloClient } from './config/apollo-client';
import * as serviceWorker from './serviceWorker';

ReactDOM.render(
  <ApolloProvider client={apolloClient}>
    <App />
```

```
  </ApolloProvider>,
  document.getElementById('root')
);

serviceWorker.unregister();
```

Running our App

Now we have everything we need to run our App. Open a console and type `yarn start` to run the development server and open a browser in this address: http://localhost:3000.

If all goes well, you should see our table with the characters from Rick and Morty.

Discussion (2)

Subscribe

Add to the discussion

Michael • Jan 8

Could be a n00b thing, but I had to add `rm-shop-v1/` in front of both the `documents:` and `generates:` portion of `codegen.yml` to get it to generate properly.

Also, it seems like the more recent create react scripts have the service worker as optional? You may want the `--template cra-template-pwa-typescript` option.

2 likes Reply

Felipe Armoni • Jan 11

That's odd. Are you sure you were running the codegen inside the app root folder (the one that has the codegen.yml file)?

Also, thanks for the tip on the service worker.

1 like Reply

Read next

Pointer vs Reference in C++: The Final Guide

ZigRazor - Sep 1

Best Practices Are The Best For Whom?

Kevin Murphy - Aug 27

Master objects in JS 🤯 (Part 2)

Ben Matt, Jr. - Sep 6

The 10 REST Commandments

Vedran Cindrić - Sep 6

Felipe Armoni

Follow

LOCATION

São Paulo

WORK

Senior Engineer at Runa HR

JOINED

Jan 5, 2020

More from Felipe Armoni

Angular Fire and Forget Polling with NgRx, RxJS and Unit Tests

#angular #ngrx #tutorial #testing

Unit Tests with Enzyme and Apollo Graphql

#graphl #react #testing #tutorial

Use Apollo to Manage the App's Local State

#graphql #react #testing #tutorial