React with Apollo and GraphQL Tutorial

MAY 01, 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 5 of 5 in this series.

Part 1: Getting Started with GitHub's GraphQL API
Part 2: GraphQL Tutorial for Beginners
Part 3: A complete React with GraphQL Tutorial
Part 4: Apollo Client Tutorial for Beginners

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 applications, but you will use the Apollo Client for your React client-side application. Along the way, you will build a simplified GitHub client that consumes GitHub's GraphQL API using Apollo instead of plain HTTP requests like the previous application. Apollo Client can be used to perform queries and mutations, and to read and write data. By the end, you should be able to showcase a React application using GraphQL and Apollo that can be used by other developers as a learning tool. You can find the final project as a repository on GitHub.

TABLE OF CONTENTS

WRITING YOUR FIRST REACT APPLICATION WITH GRAPHQL AND APOLLO CLIENT

Now we'll focus on using Apollo Client in React by building another client application. Basically, you will learn how to connect the data-layer to the view-layer. We'll cover how to send queries and mutations from the view-layer, and how to update the view-layer to reflect the result. Further, you will learn to use GraphQL features like pagination, optimistic UI, caching, local state management, and prefetching with Apollo Client in React.

For this application, no elaborate React setup is needed. Simply use create-react-app to create your React application. If you want to have an elaborate React setup instead, see this setup guide for using Webpack with React. To get started, the following steps have to be performed:

- Create a new React application with create-react-app
- Create a folder/file structure for your project (recommendation below)

You can create your own folder and file structure for your components in the *src/* folder; the following top level structure is only a recommendation. If you adjust it to your own needs, keep in mind that the JavaScript import statements with their paths will need to be adjusted to match. If you don't want to create everything, you can clone this GitHub repository instead and follow its installation instructions.



- App/
  - index.js
- Button/
- Error/
- FetchMore/
- Input/
- Issue/
  - IssueList/
  - IssueItem/
  - index.js
- Link/
- Loading/
- Organization/
- Profile/
- Repository/
  - RepositoryList/
  - RepositoryItem/
  - index.js
- TextArea/
- constants/
  - routes.js

- index.js
- serviceWorker.js
- style.css

The folders primarily represent React components. Some components will be reusable UI components such as the Input and Link components, while other components like Repository and Profile components are domain specific for the GitHub client application. Only the top level folders are specified for now, though more can be introduced later if you choose. Moreover, the *constants* folder has only one file to specify the application's routes, which will be introduced later. You may want to navigate from a page that shows repositories of an organization (Organization component) to a page which shows repositories of yourself (Profile component).

This application will use plain CSS classes and CSS files. By following the plain CSS classes, you can avoid difficulties that may occur with other tools. You will find all the CSS files and their content in the appendix section for this application. The components will use their class names without explaining them. The next sections should be purely dedicated to JavaScript, React, and GraphQL.

Exercises:

- If you are not familiar with React, read up *The Road to learn React*
- Set up the recommended folder/file structure (if you are not going with your own structure and didn't clone the repository)
  - Create the CSS *style.css* files in their specified folders from the CSS appendix section
  - Create the *index.js* files for the components
  - Create further folders on your own for non top level components (e.g. Navigation) when conducting the following sections
- Run the application with `npm start`
  - Make sure there are no errors
  - Render only a basic App component with *src/App/index.js* in the *src/index.js* file
- Invest 3 minutes of your time and take the quiz

CONFIGURE APOLLO CLIENT FOR REACT AND GITHUB'S GRAPHQL API

In this section, you will set up an Apollo Client instance like we did previously. However, this time you will use Apollo Client directly without the zero-configuration package Apollo Boost, meaning you'll need to configure the Apollo Client yourself without sensitive defaults. While it's best to use

a tool with sensitive defaults for learning, configuring Apollo yourself exposes the composable ecosystem of Apollo Client, how to use it for an initial setup, and how to advance this setup later.

The Apollo Client setup can be completed in the top-level *src/index.js* file, where the React to HTML entry point exists as well. First, install the Apollo Client in your project folder using the command line:

```
npm install apollo-client --save
```

Two utility packages are required for two mandatory configurations used to create the Apollo Client. The apollo-cache-inmemory is a recommended cache (read also as: store or state) for your Apollo Client to manage the data, while apollo-link-http is used to configure the URI and additional network information once for an Apollo Client instance.

```
npm install apollo-cache-inmemory apollo-link-http --save
```

As you can see, nothing has been mentioned about React, only the Apollo Client plus two packages for its configuration. There are two additional packages required for Apollo Client to work with GraphQL, to be used as internal dependencies by Apollo. The latter is also used to define queries and mutations. Previously, these utilities came directly from Apollo Boost.

```
npm install graphql graphql-tag --save
```

That's it for package installation, so now we enter the Apollo Client setup and configuration. In your top level *src/index.js* file, where all the Apollo Client setup will be done in this section, import the necessary classes for the Apollo Client setup from the previously installed packages.

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

import './style.css';
import App from './App';

...
```

The `ApolloClient` class is used to create the client instance, and the `HttpLink` and `InMemoryCache` are used for its mandatory configurations. First, you can create a configured

`HttpLink` instance, which will be fed to the Apollo Client creation.

```
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
    }`,
  },
});
```

You may recall the mandatory configuration from previous applications. The `uri` is a mandatory value to define the only GraphQL API endpoint used by the Apollo Client. In this case, Github's GraphQL endpoint is passed as value. When consuming the GitHub GraphQL API, you have to authorize yourself with your personal access token. You should have already created the token in a previous section, which you can now define in a *.env* file in your project folder. Afterward, it should be accessible with `process.env`. Keep in mind that you have to use the `REACT_APP` prefix when using create-react-app, because that's how it is required by create-react-app. Otherwise, you would be free to choose your own naming for it.

Second, create the cache as the place where the data is managed in Apollo Client. The cache normalizes your data, caches requests to avoid duplicates, and makes it possible to read and write data to the cache. You will use it multiple times while developing this application. The cache instantiation is straightforward, as it doesn't require you to pass any arguments to it. Check the API to explore further configurations.

```
const cache = new InMemoryCache();
```

Finally, you can use both instantiated configurations, the link and the cache, to create the instance of the Apollo Client in the *src/index.js* file.

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

To initialize Apollo Client, you must specify link and cache properties on the config object. Once you start your application again, there should be no errors. If there are any, check whether you

have implemented a basic App component in your *src/App/index.js* file because the ReactDOM API needs to hook this component into the HTML.

Exercises:

- Confirm your source code for the last section
  - Confirm the changes from the last section
- Read more about the network layer configuration in Apollo Client
- Invest 3 minutes of your time and take the quiz

CONNECT DATA-LAYER TO VIEW-LAYER: INTRODUCING REACT APOLLO

All we've done thus far has been the framework agnostic part of Apollo Client. However, without connecting React to it, you'd have a hard time making effective use of GraphQL. That's why there is an official library to connect both worlds: react-apollo. The great thing about those connecting libraries is that there are solutions for other view-layer solutions like Angular and Vue, too, so you can use the Apollo Client in a framework agnostic way. In the following, it needs two steps to connect the Apollo Client with React. First, install the library in the command line in your project folder:

```
npm install react-apollo --save
```

Second, import its ApolloProvider component, and use it as a composing component around your App component in the *src/index.js* file. Under the hood, it uses React's Context API to pass the Apollo Client through your application.

```
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';

...

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

```
);
```

Now you have implicit access to the Apollo Client in your React view-layer. It says implicit because most often you will not use the client explicitly. You will see in the next section what this means.

Exercises:

- Confirm your source code for the last section
  - Confirm the changes from the last section
- Read more about configuring and connecting Apollo Client to React
- Invest 3 minutes of your time and take the quiz

GRAPHQL QUERY WITH APOLLO CLIENT IN REACT

In this section, you will implement your first GraphQL query using Apollo Client in React. You've seen how different entities, such as the current user (viewer) or repositories, can be queried from GitHub's GraphQL API. This time you will do it in React. A Profile component might be the best place to render the current user and its associated repositories. Start by using the not-yet-implemented Profile component in your App component in the *src/App/index.js* file, which we'll take care of next. It makes sense to extract the Profile component now, because the App component will be the static frame around the application later. Components like Navigation and Footer are static, and components such as Profile and Organization are dynamically rendered based on routing (URLs).

```
import React, { Component } from 'react';

import Profile from '../Profile';

class App extends Component {
  render() {
    return <Profile />;
  }
}

export default App;
```

In your *src/Profile/index.js* file, add a simple functional stateless component. In the next step you will extend it with a GraphQL query.

```
import React from 'react';

const Profile = () =>
  <div>Profile</div>

export default Profile;
```

Now we'll learn to query data with GraphQL and Apollo Client. The Apollo Client was provided in a previous section with React's Context API in a top level component. You have implicit access to it, but never use it directly for standard queries and mutations. It says "standard" here, because there will be situations where you use the Apollo Client instance directly while implementing this application.

The React Apollo package grants access to a Query component, which takes a query as prop and executes it when its rendered. That's the important part: it executes the query when it is rendered. It uses React's render props pattern, using a child as a function implementation where you can access the result of the query as an argument.

```
import React from 'react';
import { Query } from 'react-apollo';

const Profile = () => (
  <Query query={}>
    {() => <div>My Profile</div>}
  </Query>
);

export default Profile;
```

This is a function that returns only JSX, but you have access to additional information in the function arguments. First, define the GraphQL query to request your authorizations. You can use a previously installed utility package to define the query.

```
import React from 'react';
import gql from 'graphql-tag';
import { Query } from 'react-apollo';

const GET_CURRENT_USER = gql`
  {
    viewer {
      login
      name
    }
  }
`;
```

```
const Profile = () => (
  <Query query={GET_CURRENT_USER}>
    {() => <div>My Profile</div>}
  </Query>
);

export default Profile;
```

Use the children as a function pattern to retrieve the query result as a data object, and render the information in your JSX.

```
import React from 'react';
import gql from 'graphql-tag';
import { Query } from 'react-apollo';

const GET_CURRENT_USER = gql`
  {
    viewer {
      login
      name
    }
  }
`;

const Profile = () => (
  <Query query={GET_CURRENT_USER}>
    {({ data }) => {
      const { viewer } = data;

      return (
        <div>
          {viewer.name} {viewer.login}
        </div>
      );
    }}
  </Query>
);

export default Profile;
```

Make sure to give some type of visual feedback until your view-layer can be rendered with actual data:

```
const Profile = () => (
  <Query query={GET_CURRENT_USER}>
    {({ data }) => {
      const { viewer } = data;
```

```
        if (!viewer) {
          return null;
        }

        return (
          <div>
            {viewer.name} {viewer.login}
          </div>
        );
      }}
    </Query>
  );
```

That's how you define a GraphQL query in a declarative way in React. Once the Query component renders, the request is executed. The Apollo Client is used, provided in a top level component, to perform the query. The render props pattern makes it possible to access the result of the query in the child function. You can try it in your browser to verify that it actually works for you.

There is more information found in the render prop function. Check the official React Apollo API for additional information beyond the examples in this application. Next, let's show a loading indicator when a query is pending:

```
const Profile = () => (
  <Query query={GET_CURRENT_USER}>
    {({ data, loading }) => {
      const { viewer } = data;

      if (loading || !viewer) {
        return <div>Loading ...</div>;
      }

      return (
        <div>
          {viewer.name} {viewer.login}
        </div>
      );
    }}
  </Query>
);
```

The application now shows a loading indicator when there is no `viewer` object or the `loading` boolean is set to true. As you can assume that the request will be pending when there is no `viewer`, you can show the loading indicator from the beginning. At this point, it's best to extract the loading indicator as its own component because you will have to reuse it later for other queries. You created a Loading folder for it before, which will house the *src/Loading/index.js* file. Then, use it in your Profile component.

```
```

```
import React from 'react';

const Loading = () =>
  <div>Loading ...</div>

export default Loading;
```

Next, extend the query with a nested list field for querying your own GitHub repositories. You have done it a few times before, so the query structure shouldn't be any different now. The following query requests a lot of information you will use in this application:

```
const GET_REPOSITORIES_OF_CURRENT_USER = gql`
  {
    viewer {
      repositories(
        first: 5
        orderBy: { direction: DESC, field: STARGAZERS }
      ) {
        edges {
          node {
            id
            name
            url
            descriptionHTML
            primaryLanguage {
              name
            }
            owner {
              login
              url
            }
            stargazers {
              totalCount
            }
            viewerHasStarred
            watchers {
              totalCount
            }
            viewerSubscription
          }
        }
      }
    }
  }
`;
```

Use this extended and renamed query in your Query component to request additional information about repositories. Pass these repositories from the query result to a new

RepositoryList component which should do all the rendering for you. It's not the responsibility of the Profile component, and you may want to render a list of repositories somewhere else.

```
...

import RepositoryList from '../Repository';
import Loading from '../Loading';

...

const Profile = () => (
  <Query query={GET_REPOSITORIES_OF_CURRENT_USER}>
    {({ data, loading }) => {
      const { viewer } = data;

      if (loading || !viewer) {
        return <Loading />;
      }

      return <RepositoryList repositories={viewer.repositories} />;
    }}
  </Query>
);
```

In your *src/Repository/index.js* file, create your first import/export statements for the RepositoryList component from a dedicated file in this folder. The *index.js* file is used as your entry point to this Repository module. Everything used from this module should be accessible by importing it from this *index.js* file.

```
import RepositoryList from './RepositoryList';

export default RepositoryList;
```

Next, define the RepositoryList component in your *src/Repository/RepositoryList/index.js* file. The component only takes the array of repositories as props, which will be retrieved by the GraphQL query to render a list of RepositoryItem components. The identifier of each repository can be passed as key attribute to the rendered list. Otherwise, all props from one repository node are passed to the RepositoryItem using the JavaScript spread operator.

```
import React from 'react';

import RepositoryItem from '../RepositoryItem';

import '../style.css';

const RepositoryList = ({ repositories }) =>
```

```
  repositories.edges.map(({ node }) => (
    <div key={node.id} className="RepositoryItem">
      <RepositoryItem {...node} />
    </div>
  ));

export default RepositoryList;
```

Finally, define the RepositoryItem component in the *src/Repository/RepositoryItem/index.js* file to render all the queried information about each repository. The file already uses a couple of stylings which you may have defined in a CSS file as suggested before. Otherwise, the component renders only static information for now.

```
import React from 'react';

import Link from '../../Link';

import '../style.css';

const RepositoryItem = ({
  name,
  url,
  descriptionHTML,
  primaryLanguage,
  owner,
  stargazers,
  watchers,
  viewerSubscription,
  viewerHasStarred,
}) => (
  <div>
    <div className="RepositoryItem-title">
      <h2>
        <Link href={url}>{name}</Link>
      </h2>

      <div className="RepositoryItem-title-action">
        {stargazers.totalCount} Stars
      </div>
    </div>

    <div className="RepositoryItem-description">
      <div
        className="RepositoryItem-description-info"
        dangerouslySetInnerHTML={{ __html: descriptionHTML }}
      />
      <div className="RepositoryItem-description-details">
        <div>
          {primaryLanguage && (
            <span>Language: {primaryLanguage.name}</span>
          )}
```

```
        </div>
        <div>
          {owner && (
            <span>
              Owner: <a href={owner.url}>{owner.login}</a>
            </span>
          )}
        </div>
      </div>
    </div>
  </div>
);

export default RepositoryItem;
```

The anchor element to link to the repository is already extracted as a Link component. The Link component in the *src/Link/index.js* file could look like the following, to make it possible to open those URLs in an extra browser tab:

```
import React from 'react';

const Link = ({ children, ...props }) => (
  <a {...props} target="_blank" rel="noopener noreferrer">
    {children}
  </a>
);

export default Link;
```

Once you restart your application, you should see a styled list of repositories with a name, url, description, star count, owner, and the project's implementation language. If you can't see any repositories, check to see if your GitHub account has any public repositories. If it doesn't, then it's normal that nothing showed up. I recommend you make yourself comfortable with GitHub by creating a couple of repositories, both for the sake of learning about GitHub and to use this data to practice with this tutorial. Another way to create repositories for your own account is forking repositories from other people.

What you have done in the last steps of this section were pure React implementation, but this is only one opinionated way on how to structure components. The most important part from this section though happens in the Profile component. There, you introduced a Query component that takes a query as prop. Once the Query component renders, it executes the GraphQL query. The result of the query is made accessible as an argument within React's render props pattern.

Exercises:

- Confirm your source code for the last section
  - Confirm the changes from the last section
- Read more about queries with Apollo Client in React
- Invest 3 minutes of your time and take the quiz

APOLLO CLIENT ERROR HANDLING IN REACT

Before diving into GraphQL mutations in React with Apollo Client, this section should clarify error handling with Apollo in React. The error handling happens on two levels: the application level and the query/mutation level. Both can be implemented with the two cases that follow. On a query level, in your Profile component, you have access to the query `data` and `loading` properties. Apart from these, you can also access the `error` object, which can be used to show a conditional error message.

```
...

import RepositoryList from '../Repository';
import Loading from '../Loading';
import ErrorMessage from '../Error';

...

const Profile = () => (
  <Query query={GET_REPOSITORIES_OF_CURRENT_USER}>
    {({ data, loading, error }) => {
      if (error) {
        return <ErrorMessage error={error} />;
      }

      const { viewer } = data;

      if (loading || !viewer) {
        return <Loading />;
      }

      return <RepositoryList repositories={viewer.repositories} />;
    }}
  </Query>
);

export default Profile;
```

Whereas the ErrorMessage component from the *src/Error/index.js* could look like the following:

```
import React from 'react';

import './style.css';

const ErrorMessage = ({ error }) => (
  <div className="ErrorMessage">
    <small>{error.toString()}</small>
  </div>
);

export default ErrorMessage;
```

Try to change the name of a field in your query to something not offered by GitHub's GraphQL API, and observe what's rendered in the browser. You should see something like this: *Error: GraphQL error: Field 'viewers' doesn't exist on type 'Query'*. Or, if you simulate offline functionality, you'll see: *Error: Network error: Failed to fetch*. That's how errors can be separated into GraphQL errors and network errors. You can handle errors on a component or query level, but it will also help with mutations later. To implement error handling on an application level, install another Apollo package:

```
npm install apollo-link-error --save
```

You can import it in your *src/index.js* file and create such an error link:

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

...

const errorLink = onError(({ graphQLErrors, networkError }) => {
  if (graphQLErrors) {
    // do something with graphql error
  }

  if (networkError) {
    // do something with network error
  }
});
```

You could differentiate the error handling at the application level into development and production mode. During development, it might be sufficient to console log the errors to a developer console in the browser. In production mode, you can setup an error tracking service like Sentry. It will teach you to identify bugs in a web dashboard more efficiently.

Now you have two links in your application: `httpLink` and `errorLink`. To combine them for use with the Apollo Client instance, we'll download yet another useful package in the Apollo ecosystem that makes link compositions possible in the command line:

```
npm install apollo-link --save
```

And second, use it to combine your two links in the *src/index.js* file:

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

...

const httpLink = ...

const errorLink = ...

const link = ApolloLink.from([errorLink, httpLink]);

const cache = new InMemoryCache();

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

That's how two or multiple links can be composed for creating an Apollo Client instance. There are several links developed by the community and Apollo maintainers that extend the Apollo Client with advanced functionality. Remember, it's important to understand that links can be used to access and modify the GraphQL control flow. When doing so, be careful to chain the control flow in the correct order. The `apollo-link-http` is called a **terminating link** because it turns an operation into a result that usually occurs from a network request. On the other side, the `apollo-link-error` is a **non-terminating link**. It only enhances your terminating link with features, since a terminating link has to be the last entity in the control flow chain.

Exercises:

- Confirm your source code for the last section
  - Confirm the changes from the last section
- Read more about different Apollo Error types and error policies
- Read more about Apollo Links
- Read more about composable Apollo Links
- Implement the apollo-link-retry in case a network request fails
- Invest 3 minutes of your time and take the quiz

GRAPHQL MUTATION WITH APOLLO CLIENT IN REACT

The previous sections have taught you how to query data with React Apollo and the Apollo Client. In this section, you will learn about mutations. As in other applications before, you will implement starring a repository with GitHub's exposed `addStar` mutation.

The mutation starts out with a variable to identify the repository to be starred. We haven't used a variable in Query component yet, but the following mutation works the same way, which can be defined in the *src/Repository/RepositoryItem/index.js* file.

```
import React from 'react';
import gql from 'graphql-tag';

...

const STAR_REPOSITORY = gql`
  mutation($id: ID!) {
    addStar(input: { starrableId: $id }) {
      starrable {
        id
        viewerHasStarred
      }
    }
  }
`;

...
```

The mutation definition takes the `id` variable as input for the `addStar` mutation. As before, you can decide what should be returned in case of a successful mutation. Now, you can use a Mutation component that represents the previously used Query component, but this time for

mutations. You have to pass the mutation prop, but also a variable prop for passing the identifier for the repository.

```
import React from 'react';
import gql from 'graphql-tag';
import { Mutation } from 'react-apollo';

...

const RepositoryItem = ({
  id,
  name,
  url,
  descriptionHTML,
  primaryLanguage,
  owner,
  stargazers,
  watchers,
  viewerSubscription,
  viewerHasStarred,
}) => (
  <div>
    <div className="RepositoryItem-title">
      <h2>
        ...
      </h2>

      <div>
        <Mutation mutation={STAR_REPOSITORY} variables={{ id }}>
          {addStar => <div>{stargazers.totalCount} Star</div>}
        </Mutation>
      </div>
    </div>

    <div className="RepositoryItem-description">
      ...
    </div>
  </div>
);
```

Note: The div element surrounding the Mutation component is there for other mutations you will implement in this section.

The `id` for each repository should be available due to previous query result. It has to be used as a variable for the mutation to identify the repository. The Mutation component is used like the Query component, because it implements the render prop pattern as well. The first argument is different, though, as it is the mutation itself instead of the mutation result. Use this function to trigger the mutation before expecting a result. Later, you will see how to retrieve the mutation

result; for now, the mutating function can be used in a button element. In this case, it is already in a Button component:

```
...

import Link from '../../Link';
import Button from '../../Button';

...

const RepositoryItem = ({ ... }) => (
  <div>
    <div className="RepositoryItem-title">
      ...

      <div>
        <Mutation mutation={STAR_REPOSITORY} variables={{ id }}>
          {(addStar) => (
            <Button
              className={'RepositoryItem-title-action'}
              onClick={addStar}
            >
              {stargazers.totalCount} Star
            </Button>
          )}
        </Mutation>
      </div>
    </div>

    ...
  </div>
);
```

The styled Button component could be implemented in the *src/Button/index.js* file. It's already extracted, because you will use it in this application later.

```
import React from 'react';

import './style.css';

const Button = ({
  children,
  className,
  color = 'black',
  type = 'button',
  ...props
}) => (
  <button
    className={`${className} Button Button_${color}`}
    type={type}
```

```
      {...props}
    >
      {children}
    </button>
  );

  export default Button;
```

Let's get to the mutation result which was left out before. Access it as a second argument in your child function of the render prop.

```
const RepositoryItem = ({ ... }) => (
  <div>
    <div className="RepositoryItem-title">
      ...

      <div>
        <Mutation mutation={STAR_REPOSITORY} variables={{ id }}>
          {(addStar, { data, loading, error }) => (
            <Button
              className={'RepositoryItem-title-action'}
              onClick={addStar}
            >
              {stargazers.totalCount} Star
            </Button>
          )}
        </Mutation>
      </div>
    </div>

    ...
  </div>
);
```

A mutation works like a query when using React Apollo. It uses the render prop pattern to access the mutation and the result of the mutation. The mutation can be used as a function in the UI. It has access to the variables that are passed in the Mutation component, but it can also override the variables when you pass them in a configuration object to the function (e.g. `addStar({ variables: { id } })`). That's a general pattern in React Apollo: You can specify information like variables in the Mutation component, or when you call the mutating function to override it.

Note that if you use the `viewerHasStarred` boolean from the query result to show either a "Star" or "Unstar" button, you can do it with a conditional rendering:

```
const RepositoryItem = ({ ... }) => (
  <div>
    <div className="RepositoryItem-title">
```

```
        ...

        <div>
          {!viewerHasStarred ? (
            <Mutation mutation={STAR_REPOSITORY} variables={{ id }}>
              {(addStar, { data, loading, error }) => (
                <Button
                  className={'RepositoryItem-title-action'}
                  onClick={addStar}
                >
                  {stargazers.totalCount} Star
                </Button>
              )}
            </Mutation>
          ) : (
            <span>{/* Here comes your removeStar mutation */}</span>
          )}

        {/* Here comes your updateSubscription mutation */}
        </div>
      </div>

      ...
    </div>
  );
```

When you star a repository as above, the "Star" button disappears. This is what we want, because it means the `viewerHasStarred` boolean has been updated in Apollo Client's cache for the identified repository. Apollo Client was able to match the mutation result with the repository identifier to the repository entity in Apollo Client's cache, the props were updated, and the UI re-rendered. Yet, on the other side, the count of stargazers who have starred the repository isn't updated because it cannot be retrieved from GitHub's API. The count must be updated in Apollo Client's cache. You will find out more about this topic in one of the following sections.

Exercises:

- Confirm your source code for the last section
  - Confirm the changes from the last section
- Read more about mutations with Apollo Client in React
- Implement other mutations in the RepositoryItem component
  - Implement the `removeStar` mutation when the `viewerHasStarred` boolean is true
  - Show a button with the watchers count which should be used to watch/unwatch a repository
    - Implement the `updateSubscription` mutation from GitHub's GraphQL API to watch/unwatch a repository based on the `viewerSubscription` status
- Invest three minutes of your time and take the quiz

GRAPHQL QUERY/MUTATION WITH HIGHER-ORDER COMPONENTS IN REACT

We've done Query and Mutation components from React Apollo to connect a data-layer (Apollo Client) with a view-layer (React). The Query component executes the query when it is rendered, whereas the Mutation component gives access to a function that triggers the mutation. Both components use the render props pattern to make the results accessible in their child functions.

Higher-Order Components (HOC) is a widely accepted alternative to React's render prop pattern. The React Apollo package implements a Higher-Order Component for queries and mutations as well, though the team behind Apollo doesn't advertise it, and even spoke in favor of render props as their first choice. Nonetheless, this section shows you the alternative, using a Higher-Order Component instead of a Render Prop, though this application will continue to use the render prop pattern afterward. If you already have access to the query result in the Profile component's arguments, there is no Query component needed in the component itself:

```
const Profile = ({ data, loading, error }) => {
  if (error) {
    return <ErrorMessage error={error} />;
  }

  const { viewer } = data;

  if (loading || !viewer) {
    return <Loading />;
  }

  return <RepositoryList repositories={viewer.repositories} />;
};
```

There is no GraphQL involved here, because all you see is the pure view-layer. Instead, the data-layer logic is extracted into a Higher-Order Component. We import the `graphql` HOC from the React Apollo package in order to apply it on the Profile component, which takes the query definition as argument.

```
import React from 'react';
import gql from 'graphql-tag';
import { graphql } from 'react-apollo';

...

const GET_REPOSITORIES_OF_CURRENT_USER = gql`
  {
    viewer {
```

```
      ...
    }
  }
`;

const Profile = ({ data, loading, error }) => {
  ...
};

export default graphql(GET_REPOSITORIES_OF_CURRENT_USER)(Profile);
```

I find the HOC approach cleaner than the render props, because it co-locates both the data-layer and view-layer instead of inserting the one into the other. However, the team behind Apollo made the decision to favor render props instead. While I find the HOC approach more concise, the render prop pattern comes with its own advantages for mutating and querying data. For instance, imagine a query depends on a prop used as variable. It would be cumbersome to access the incoming prop in a statically-defined Higher-Order Component, but it can be dynamically used in a render prop because it is used within the Profile component where the props are naturally accessible. Another advantage is the power of composition for render props, which is useful when one query depends on the result of another. It can be achieved with HOCs as well, but again, it is more cumbersome. It boils down to seemingly never ending "Higher-Order Components vs Render Props" discussions.

Exercises:

- Confirm your source code for the last section
  - Confirm the changes from the last section
- Come up with your own opinion about the advantages and disadvantages of using a Higher-Order Component or Render Prop
- Try to implement one of your mutations with a Higher-Order Component
- Invest 3 minutes of your time and take the quiz

LOCAL STATE MANAGEMENT WITH APOLLO CLIENT IN REACT

Let's get back to the Repository component. You have experienced that the `viewerHasStarred` boolean updates in the Apollo Client's cache after a mutation was successful. That's great, because Apollo Client handles this for you, based on the mutation result. If you have followed the exercises of the mutation section, you should probably see something like a toggling "Star" and "Unstar" label for the button. All of this happens because you returned the `viewerHasStarred`

boolean in your mutation result. Apollo Client is clever enough to update the repository entity, which is normalized and accessible in the cache. That's powerful default behavior, isn't it? You don't need to handle the local state management yourself, since Apollo Client figures it out for you as long as you provide useful information in the mutation's result.

Apollo Client doesn't update the count of stars after the mutation, though. Normally, it is assumed that the count of stars increments by one when it is starred, with the opposite for unstarring. Since we don't return a count of stargazers in the mutation result, you have to handle the update in Apollo Client's cache yourself. Using Apollo Client's `refetchQueries` option is the naive approach for a mutation call, or a Mutation component to trigger a refetch for all queries, where the query result might be affected by the mutation. But that's not the best way to deal with this problem. It costs another query request to keep the data consistent after a mutation. In a growing application, this approach will eventually become problematic. Fortunately, the Apollo Client offers other functionalities to read/write manually from/to the cache locally without more network requests. The Mutation component offers a prop where you can insert update functionality that has access to the Apollo Client instance for the update mechanism.

Before implementing the update functionality for the local state management, let's refactor another piece of code that will be useful for a local state update mechanism. The query definition next to your Profile component has grown to several fields with multiple object nestings. Previously, you learned about GraphQL fragments, and how they can be used to split parts of a query to reuse later. Next, we will split all the field information you used for the repository's node. You can define this fragment in the *src/Repository/fragments.js* file to keep it reusable for other components.

```
import gql from 'graphql-tag';

const REPOSITORY_FRAGMENT = gql`
  fragment repository on Repository {
    id
    name
    url
    descriptionHTML
    primaryLanguage {
      name
    }
    owner {
      login
      url
    }
    stargazers {
      totalCount
    }
    viewerHasStarred
    watchers {
      totalCount
```

```
    }
    viewerSubscription
  }
`;

export default REPOSITORY_FRAGMENT;
```

You split this partial query (fragment), because it is used more often in this application in the next sections for a local state update mechanism, hence the previous refactoring.

The fragment shouldn't be imported directly from the *src/Repository/fragments.js* path to your Profile component, because the *src/Repository/index.js* file is the preferred entry point to this module.

```
import RepositoryList from './RepositoryList';
import REPOSITORY_FRAGMENT from './fragments';

export { REPOSITORY_FRAGMENT };

export default RepositoryList;
```

Finally, import the fragment in the Profile component's file to use it again.

```
...

import RepositoryList, { REPOSITORY_FRAGMENT } from '../Repository';
import Loading from '../Loading';
import ErrorMessage from '../Error';

const GET_REPOSITORIES_OF_CURRENT_USER = gql`
  {
    viewer {
      repositories(
        first: 5
        orderBy: { direction: DESC, field: STARGAZERS }
      ) {
        edges {
          node {
            ...repository
          }
        }
      }
    }
  }

  ${REPOSITORY_FRAGMENT}
`;

...
```

The refactoring is done. Your query is now more concise, and the fragment in its natural repository module can be reused for other places and functionalities. Next, use Mutation component's `update` prop to pass a function which will update the local cache eventually.

```
...

const updateAddStar = (client, mutationResult) => {
  ...
};

const RepositoryItem = ({ ... }) => (
  <div>
    <div className="RepositoryItem-title">
      ...

      <div>
        {viewerHasStarred ? (
          ...
        ) : (
          <Mutation
            mutation={STAR_REPOSITORY}
            variables={{ id }}
            update={updateAddStar}
          >
            ...
          </Mutation>
        )}
      </div>
    </div>

    ...
  </div>
);

export default RepositoryItem;
```

The function is extracted as its own JavaScript variable, otherwise ends up too verbose in the RepositoryItem component when keeping it inlined in the Mutation component. The function has access to the Apollo Client and the mutation result in its argument, and you need both to update data so you can destructure the mutation result in the function signature. If you don't know how the mutation result looks like, check the `STAR_REPOSITORY` mutation definition again, where you defined all fields that should appear in the mutation result. For now, the `id` of the repository to be updated is the important part.

```
const updateAddStar = (
  client,
```

```
  { data: { addStar: { starrable: { id } } } },
) => {
  ...
};
```

You could have passed the `id` of the repository to the `updateAddStar()` function, which was a higher-order function in the Mutation component's render prop child function. You already have access to the repository's identifier in the Repository component.

Now comes the most exciting part of this section. You can use the Apollo Client to read data from the cache, but also to write data to it. The goal is to read the starred repository from the cache, which is why we need the `id` to increment its stargazers count by one and write the updated repository back to the cache. You got the repository by its `id` from the cache by extracting the repository fragment. You can use it along with the repository identifier to retrieve the actual repository from Apollo Client's cache without querying all the data with a naive query implementation.

```
...

import REPOSITORY_FRAGMENT from '../fragments';
import Link from '../../Link';
import Button from '../../Button';

...

const updateAddStar = (
  client,
  { data: { addStar: { starrable: { id } } } },
) => {
  const repository = client.readFragment({
    id: `Repository:${id}`,
    fragment: REPOSITORY_FRAGMENT,
  });

  // update count of stargazers of repository

  // write repository back to cache
};
```

The Apollo Client's cache that you set up to initialize the Apollo Client normalizes and stores queried data. Otherwise, the repository would be a deeply nested entity in a list of repositories for the query structure used in the Profile component. Normalization of a data structure makes it possible to retrieve entities by their identifier and their GraphQL `__typename` meta field. The combination of both is the default key, which is called a composite key, to read or write an entity

from or to the cache. You may find out more about changing this default composite key in the exercises of this section.

Furthermore, the resulting entity has all properties specified in the fragment. If there is a field in the fragment not found on the entity in the cache, you may see the following error message: *Can't find field __typename on object ...*. That's why we use the identical fragment to read from the local cache to query the GraphQL API.

After you have retrieved the repository entity with a fragment and its composite key, you can update the count of stargazers and write back the data to your cache. In this case, increment the number of stargazers.

```
const updateAddStar = (
  client,
  { data: { addStar: { starrable: { id } } } },
) => {
  const repository = client.readFragment({
    id: `Repository:${id}`,
    fragment: REPOSITORY_FRAGMENT,
  });

  const totalCount = repository.stargazers.totalCount + 1;

  client.writeFragment({
    id: `Repository:${id}`,
    fragment: REPOSITORY_FRAGMENT,
    data: {
      ...repository,
      stargazers: {
        ...repository.stargazers,
        totalCount,
      },
    },
  });
};
```

Let's recap all three steps here. First, you have retrieved (read) the repository entity from the Apollo Client using an identifier and the fragment; second, you updated the information of the entity; and third, you wrote back the data with updated information, but kept all remaining information intact using the JavaScript spread operator. This is a manual update mechanism that can be used when a mutation is missing data.

It is a good practice to use an identical fragment for all three parts: the initial query, the `readFragment()`, and `writeFragment()` cache method. Your data structure for the entity stays consistent in your cache. For instance, if you forget to include a property defined by the

fragment's fields in data object of the `writeFragment()` method, you get a warning: *Missing field __typename in ....*

On an implementation level, you learned about extracting fragments from a query or mutation. Fragments allow you to define your shared entities by GraphQL types. You can reuse those in your queries, mutations or local state management methods to update the cache. On a higher level, you learned that Apollo Client's cache normalizes your data, so you can retrieve entities that were fetched with a deeply nested query using their type and identifier as composite key. Without it, you'd have to perform normalizations for all the fetched data before putting it in your store/state.

Exercises:

- Confirm your source code for the last section
  - Confirm the changes from the last section
- Read more about Local State Management in Apollo Client
- Read more about Fragments in Apollo Client
- Implement local cache updates for all the other mutations from the previous exercises
  - Implement the identical local cache update, but with decreasing the count of stargazers, for your `removeStar` mutation
  - Implement the local cache update for the `updateSubscription` mutation
  - You will see in the next section a working solution for it
- Read more about Caching in Apollo Client and the composite key to identify entities
- Invest 3 minutes of your time and take the quiz

APOLLO CLIENT OPTIMISTIC UI IN REACT

We've covered the basics, so now it's time for the advanced topics. One of those topics is the optimistic UI with React Apollo, which makes everything onscreen more synchronous. For instance, when liking a post on Twitter, the like appears immediately. As developers, we know there is a request that sends the information for the like to the Twitter backend. This request is asynchronous and doesn't resolve immediately with a result. The optimistic UI immediately assumes a successful request and mimics the result of such request for the frontend so it can update its UI immediately, before the real response arrives later. With a failed request, the optimistic UI performs a rollback and updates itself accordingly. Optimistic UI improves the user experience by omitting inconvenient feedback (e.g. loading indicators) for the user. The good thing is that React Apollo comes with this feature out of the box.

In this section, you will implement an optimistic UI for when a user clicks the watch/unwatch mutation you implemented in a previous exercise. If you haven't, it's time to implement it now, or you can substitute it with the star or unstar mutation. Either way, completing the optimistic UI behavior for all three mutations is the next exercise. For completeness, this is a possible implementation of the watch mutation as a button next to the "Star"/"Unstar" buttons. First, the mutation:

```
const WATCH_REPOSITORY = gql`
  mutation ($id: ID!, $viewerSubscription: SubscriptionState!) {
    updateSubscription(
      input: { state: $viewerSubscription, subscribableId: $id }
    ) {
      subscribable {
        id
        viewerSubscription
      }
    }
  }
`;
```

Second, the usage of the mutation with a Mutation render prop component:

```
const VIEWER_SUBSCRIPTIONS = {
  SUBSCRIBED: 'SUBSCRIBED',
  UNSUBSCRIBED: 'UNSUBSCRIBED',
};

const isWatch = viewerSubscription =>
  viewerSubscription === VIEWER_SUBSCRIPTIONS.SUBSCRIBED;

const updateWatch = () => {
  ...
};

const RepositoryItem = ({ ... }) => (
  <div>
    <div className="RepositoryItem-title">
      ...

      <div>
        ...

        <Mutation
          mutation={WATCH_REPOSITORY}
          variables={{
            id,
            viewerSubscription: isWatch(viewerSubscription)
              ? VIEWER_SUBSCRIPTIONS.UNSUBSCRIBED
              : VIEWER_SUBSCRIPTIONS.SUBSCRIBED,
```

```
          }}
          update={updateWatch}
        >
          {(updateSubscription, { data, loading, error }) => (
            <Button
              className="RepositoryItem-title-action"
              onClick={updateSubscription}
            >
              {watchers.totalCount}{' '}
              {isWatch(viewerSubscription) ? 'Unwatch' : 'Watch'}
            </Button>
          )}
        </Mutation>

        ...
      </div>
    </div>

    ...
  </div>
);
```

And third, the missing update function that is passed to the Mutation component:

```
const updateWatch = (
  client,
  {
    data: {
      updateSubscription: {
        subscribable: { id, viewerSubscription },
      },
    },
  },
) => {
  const repository = client.readFragment({
    id: `Repository:${id}`,
    fragment: REPOSITORY_FRAGMENT,
  });

  let { totalCount } = repository.watchers;
  totalCount =
    viewerSubscription === VIEWER_SUBSCRIPTIONS.SUBSCRIBED
      ? totalCount + 1
      : totalCount - 1;

  client.writeFragment({
    id: `Repository:${id}`,
    fragment: REPOSITORY_FRAGMENT,
    data: {
      ...repository,
      watchers: {
        ...repository.watchers,
        totalCount,
```

```
      },
    },
  });
};
```

Now let's get to the optimistic UI feature. Fortunately, the Mutation component offers a prop for the optimistic UI strategy called `optimisticResponse`. It returns the same result, which is accessed as argument in the function passed to the `update` prop of the Mutation component. With a watch mutation, only the `viewerSubscription` status changes to subscribed or unsubscribed. This is an optimistic UI.

```
const RepositoryItem = ({ ... }) => (
  <div>
    <div className="RepositoryItem-title">
      ...

      <div>
        ...

        <Mutation
          mutation={WATCH_REPOSITORY}
          variables={{
            id,
            viewerSubscription: isWatch(viewerSubscription)
              ? VIEWER_SUBSCRIPTIONS.UNSUBSCRIBED
              : VIEWER_SUBSCRIPTIONS.SUBSCRIBED,
          }}
          optimisticResponse={{
            updateSubscription: {
              __typename: 'Mutation',
              subscribable: {
                __typename: 'Repository',
                id,
                viewerSubscription: isWatch(viewerSubscription)
                  ? VIEWER_SUBSCRIPTIONS.UNSUBSCRIBED
                  : VIEWER_SUBSCRIPTIONS.SUBSCRIBED,
              },
            },
          }}
          update={updateWatch}
        >
          ...
        </Mutation>

        ...
      </div>
    </div>

    ...
  </div>
);
```

When you start your application and watch a repository, the "Watch" and "Unwatch" label of the button changes immediately after clicking it. This is because the optimistic response arrives synchronously, while the real response is pending and resolves later. Since the `__typename` meta field comes with every Apollo request, include those as well.

An additional benefit of the optimistic response is that it makes the count of watchers updates optimistic, too. The function used in the `update` prop is called twice now, the first time with the optimistic response, and the second with a response from GitHub's GraphQL API. It makes sense to capture identical information in the optimistic response expected as a mutation result in the function passed to the `update` prop of the Mutation component. For instance, if you don't pass the `id` property in the `optimisticResponse` object, the function passed to the `update` prop throws an error, because it can't retrieve the repository from the cache without an identifier.

At this point, it becomes debatable whether or not the Mutation component becomes too verbose. Using the Render Props pattern co-locates the data layer even more to the view-layer than Higher-Order Components. One could argue it doesn't co-locate the data-layer, but inserts it into the view-layer. When optimizations like the `update` and `optimisticResponse` props are put into the Render Prop Component, it can become too verbose for a scaling application. I advise using techniques you've learned as well as your own strategies to keep your source code concise. I see four different ways to solve this issue:

1. Keep the declarations inlined (see: `optimisticUpdate`)
2. Extracting the inlined declarations as variable (see: `update`).
3. Perform a combination of 1 and 2 whereas only the most verbose parts are extracted
4. Use Higher-Order Components instead of Render Props to co-locate data-layer, instead of inserting it in the view-layer

The first three are about **inserting** a data-layer into the view-layer, while the last is about **co-locating** it. Each comes with drawbacks. Following the second way, you might yourself declaring functions instead of objects, or higher-order functions instead of functions because you need to pass arguments to them. With the fourth, you could encounter the same challenge in keeping HOCs concise. There, you could use the other three ways too, but this time in a HOC rather than a Render Prop.

Exercises:

- Confirm your source code for the last section
  - Confirm the changes from the last section
- Throttle your internet connection (often browsers offers such functionality) and experience how the `optimisticResponse` takes the `update` function into account even though the request is

slow

- Try different ways of co-locating or inserting your data-layer with render props and higher-order components
- Implement the optimistic UIs for the star and unstar mutations
- Read more about Apollo Optimistic UI in React with GraphQL
- Invest 3 minutes of your time and take the quiz

GRAPHQL PAGINATION WITH APOLLO CLIENT IN REACT

Finally, you are going to implement another advanced feature when using a GraphQL API called **pagination**. In this section, you implement a button that allows successive pages of repositories to be queries, a simple "More" button rendered below the list of repositories in the RepositoryList component. When is clicked, another page of repositories is fetched and merged with the previous list as one state into Apollo Client's cache.

First, extend the query next for your Profile component with the necessary information to allow pagination for the list of repositories:

```
const GET_REPOSITORIES_OF_CURRENT_USER = gql`
  query($cursor: String) {
    viewer {
      repositories(
        first: 5
        orderBy: { direction: DESC, field: STARGAZERS }
        after: $cursor
      ) {
        edges {
          node {
            ...repository
          }
        }
        pageInfo {
          endCursor
          hasNextPage
        }
      }
    }
  }
  ${REPOSITORY_FRAGMENT}
`;
```

The `endCursor` can be used as `$cursor` variable when fetching the next page of repositories, but the `hasNextPage` can disable the functionality (e.g. not showing the "More" button) to fetch another page. The initial request to fetch the first page of repositories will have a `$cursor` variable of `undefined`, though. GitHub's GraphQL API will handle this case gracefully and return the first items from the list of repositories without considering the `after` argument. Every other request to fetch more items from the list will send a defined `after` argument with the cursor, which is the `endCursor` from the query.

Now we have all information to fetch more pages of repositories from GitHub's GraphQL API. The Query component exposes a function to retrieve them in its child function. Since the button to fetch more repositories fits best in the the RepositoryList component, you can pass this function as prop to it.

```
const Profile = () => (
  <Query query={GET_REPOSITORIES_OF_CURRENT_USER}>
    {({ data, loading, error, fetchMore }) => {
      ...

      return (
        <RepositoryList
          repositories={viewer.repositories}
          fetchMore={fetchMore}
        />
      );
    }}
  </Query>
);
```

Next, use the function in the RepositoryList component, and add a button to fetch successive pages of repositories that appears when another page is available.

```
import React, { Fragment } from 'react';

...

const RepositoryList = ({ repositories, fetchMore }) => (
  <Fragment>
    {repositories.edges.map(({ node }) => (
      ...
    ))}

    {repositories.pageInfo.hasNextPage && (
      <button
        type="button"
        onClick={() =>
          fetchMore({
            /* configuration object */
```

```
          })
        }
      >
        More Repositories
      </button>
    )}
  </Fragment>
);

export default RepositoryList;
```

The `fetchMore()` function performs the query from the initial request, and takes a configuration object, which can be used to override variables. With pagination, this means you pass the `endCursor` of the previous query result to use it for the query as `after` argument. Otherwise, you would perform the initial request again because no variables are specified.

```
const RepositoryList = ({ repositories, fetchMore }) => (
  <Fragment>
    ...

    {repositories.pageInfo.hasNextPage && (
      <button
        type="button"
        onClick={() =>
          fetchMore({
            variables: {
              cursor: repositories.pageInfo.endCursor,
            },
          })
        }
      >
        More Repositories
      </button>
    )}
  </Fragment>
);
```

If you attempt to click the button, you should get the following error message: *Error: updateQuery option is required.*. The `updateQuery` function is needed to tell Apollo Client how to merge the previous result with a new one. Define the function outside of the button, because it would become too verbose otherwise.

```
const updateQuery = (previousResult, { fetchMoreResult }) => {
  ...
};

const RepositoryList = ({ repositories, fetchMore }) => (
  <Fragment>
```

```
      ...

      {repositories.pageInfo.hasNextPage && (
        <button
          type="button"
          onClick={() =>
            fetchMore({
              variables: {
                cursor: repositories.pageInfo.endCursor,
              },
              updateQuery,
            })
          }
        >
          More Repositories
        </button>
      )}
    </Fragment>
  );
```

The function has access to the previous query result, and to the next result that resolves after the button click:

```
const updateQuery = (previousResult, { fetchMoreResult }) => {
  if (!fetchMoreResult) {
    return previousResult;
  }

  return {
    ...previousResult,
    viewer: {
      ...previousResult.viewer,
      repositories: {
        ...previousResult.viewer.repositories,
        ...fetchMoreResult.viewer.repositories,
        edges: [
          ...previousResult.viewer.repositories.edges,
          ...fetchMoreResult.viewer.repositories.edges,
        ],
      },
    },
  };
};
```

In this function, you can merge both results with the JavaScript spread operator. If there is no new result, return the previous result. The important part is merging the `edges` of both repositories objects to have a merge list of items. The `fetchMoreResult` takes precedence over the `previousResult` in the `repositories` object because it contains the new `pageInfo`, with its `endCursor` and `hasNextPage` properties from the last paginated result. You need to have those

when clicking the button another time to have the correct cursor as an argument. If you want to checkout an alternative to the verbose JavaScript spread operator when dealing with deeply nested data, checkout the changes in this GitHub Pull Request that uses Lenses from Ramda.js.

To add one more small improvement for user friendliness, add a loading indicator when more pages are fetched. So far, the `loading` boolean in the Query component of the Profile component is only true for the initial request, but not for the following requests. Change this behavior with a prop that is passed to the Query component, and the loading boolean will be updated accordingly.

```
const Profile = () => (
  <Query
    query={GET_REPOSITORIES_OF_CURRENT_USER}
    notifyOnNetworkStatusChange={true}
  >
    {({ data, loading, error, fetchMore }) => {
      ...
    }}
  </Query>
);
```

When you run your application again and try the "More" button, you should see odd behavior. Every time you load another page of repositories, the loading indicator is shown, but the list of repositories disappears entirely, and the merged list is rendered as assumed. Since the `loading` boolean becomes true with the initial and successive requests, the conditional rendering in the Profile component will always show the loading indicator. It returns from the Profile function early, never reaching the code to render the RepositoryList. A quick change from || to && of the condition will allow it to show the loading indicator for the initial request only. Every request after that, where the `viewer` object is available, is beyond this condition, so it renders the RepositoryList component.

```
const Profile = () => (
  <Query
    query={GET_REPOSITORIES_OF_CURRENT_USER}
    notifyOnNetworkStatusChange={true}
  >
    {({ data, loading, error, fetchMore }) => {
      ...

      const { viewer } = data;

      if (loading && !viewer) {
        return <Loading />;
      }
```

```
        return (
          <RepositoryList
            loading={loading}
            repositories={viewer.repositories}
            fetchMore={fetchMore}
          />
        );
      }}
    </Query>
  );
```

The boolean can be passed down to the RepositoryList component. There it can be used to show a loading indicator instead of the "More" button. Since the boolean never reaches the RepositoryList component for the initial request, you can be sure that the "More" button only changes to the loading indicator when there is a successive request pending.

```
import React, { Fragment } from 'react';

import Loading from '../../Loading';
import RepositoryItem from '../RepositoryItem';

...

const RepositoryList = ({ repositories, loading, fetchMore }) => (
  <Fragment>
    ...

    {loading ? (
      <Loading />
    ) : (
      repositories.pageInfo.hasNextPage && (
        <button
          ...
        >
          More Repositories
        </button>
      )
    )}
  </Fragment>
);
```

The pagination feature is complete now, and you are fetching successive pages of an initial page, then merging the results in Apollo Client's cache. In addition, you show your user feedback about pending requests for either the initial request or further page requests.

Now we'll take it a step further, making the button used to fetch more repositories reusable. Let me explain why this would be a neat abstraction. In an upcoming section, you have another list field that could potentially implement the pagination feature. There, you have to introduce the

`More` button, which could be nearly identical to the `More` button you have in the RepositoryList component. Having only one button in a UI would be a satisfying abstraction, but this abstraction wouldn't work in a real-world coding scenario. You would have to introduce a second list field first, implement the pagination feature for it, and then consider an abstraction for the `More` button. For the sake of the tutorial, we implement this abstraction for the pagination feature only in this section, though you should be aware this is a premature optimization put in place for you to learn it.

For another way, imagine you wanted to extract the functionality of the `More` button into a FetchMore component. The most important thing you would need is the `fetchMore()` function from the query result. The `fetchMore()` function takes an object to pass in the necessary `variables` and `updateQuery` information as a configuration. While the former is used to define the next page by its cursor, the latter is used to define how the results should be merged in the local state. These are the three essential parts: fetchMore, variables, and updateQuery. You may also want to shield away the conditional renderings in the FetchMore component, which happens because of the `loading` or `hasNextPage` booleans. Et voilà! That's how you get the interface to your FetchMore abstraction component.

```
import React, { Fragment } from 'react';

import FetchMore from '../../FetchMore';
import RepositoryItem from '../RepositoryItem';

...

const RepositoryList = ({ repositories, loading, fetchMore }) => (
  <Fragment>
    {repositories.edges.map(({ node }) => (
      <div key={node.id} className="RepositoryItem">
        <RepositoryItem {...node} />
      </div>
    ))}

    <FetchMore
      loading={loading}
      hasNextPage={repositories.pageInfo.hasNextPage}
      variables={{
        cursor: repositories.pageInfo.endCursor,
      }}
      updateQuery={updateQuery}
      fetchMore={fetchMore}
    >
      Repositories
    </FetchMore>
  </Fragment>
);

export default RepositoryList;
```

Now this FetchMore component can be used by other paginated lists as well, because every part that can be dynamic is passed as props to it. Implementing a FetchMore component in the *src/FetchMore/index.js* is the next step. First, the main part of the component:

```
import React from 'react';

import './style.css';

const FetchMore = ({
  variables,
  updateQuery,
  fetchMore,
  children,
}) => (
  <div className="FetchMore">
    <button
      type="button"
      className="FetchMore-button"
      onClick={() => fetchMore({ variables, updateQuery })}
    >
      More {children}
    </button>
  </div>
);

export default FetchMore;
```

Here, you can see how the `variables` and `updateQuery` are taken as configuration object for the `fetchMore()` function when it's invoked. The button can be made cleaner using the Button component you defined in a previous section. To add a different style, let's define a specialized ButtonUnobtrusive component next to the Button component in the *src/Button/index.js* file:

```
import React from 'react';

import './style.css';

const Button = ({ ... }) => ...

const ButtonUnobtrusive = ({
  children,
  className,
  type = 'button',
  ...props
}) => (
  <button
    className={`${className} Button_unobtrusive`}
    type={type}
```

```
      {...props}
    >
      {children}
    </button>
  );

  export { ButtonUnobtrusive };

  export default Button;
```

Now the ButtonUnobtrusive component is used as button instead of the button element in the FetchMore component. In addition, the two booleans `loading` and `hasNextPage` can be used for the conditional rendering, to show the Loading component or nothing, because there is no next page which can be fetched.

```
import React from 'react';

import Loading from '../Loading';
import { ButtonUnobtrusive } from '../Button';

import './style.css';

const FetchMore = ({
  loading,
  hasNextPage,
  variables,
  updateQuery,
  fetchMore,
  children,
}) => (
  <div className="FetchMore">
    {loading ? (
      <Loading />
    ) : (
      hasNextPage && (
        <ButtonUnobtrusive
          className="FetchMore-button"
          onClick={() => fetchMore({ variables, updateQuery })}
        >
          More {children}
        </ButtonUnobtrusive>
      )
    )}
  </div>
);

export default FetchMore;
```

That's it for the abstraction of the FetchMore button for paginated lists with Apollo Client. Basically, you pass in everything needed by the `fetchMore()` function, including the function

itself. You can also pass all booleans used for conditional renderings. You end up with a reusable FetchMore button that can be used for every paginated list.

Exercises:

- Confirm your source code for the last section
  - Confirm the changes from the last section
- Read more about pagination with Apollo Client in React
- Invest 3 minutes of your time and take the quiz

GRAPHQL CACHING OF QUERIES WITH APOLLO CLIENT IN REACT

In this section, you introduce React Router to show two separate pages for your application. At the moment, you are only showing one page with a Profile component that displays all your repositories. We want to add another Organization component that shows repositories by an organization, and there could be a search field as well, to lookup individual organizations with their repositories on that page. Let's do this by introducing React Router to your application. If you haven't used React Router before, make sure to conduct the exercises of this section to learn more about it.

```
npm install react-router-dom --save
```

In your *src/constants/routes.js* file, you can specify both routes you want to make accessible by React Router. The `ORGANIZATION` route points to the base URL, while the `PROFILE` route points to a more specific URL.

```
export const ORGANIZATION = '/';
export const PROFILE = '/profile';
```

Next, map both routes to their components. The App component is the perfect place to do it because the two routes will exchange the Organization and Profile components based on the URL there.

```
import React, { Component } from 'react';
import { BrowserRouter as Router, Route } from 'react-router-dom';

import Profile from '../Profile';
```

```
import Organization from '../Organization';

import * as routes from '../constants/routes';

import './style.css';

class App extends Component {
  render() {
    return (
      <Router>
        <div className="App">
          <div className="App-main">
            <Route
              exact
              path={routes.ORGANIZATION}
              component={() => (
                <div className="App-content_large-header">
                  <Organization />
                </div>
              )}
            />
            <Route
              exact
              path={routes.PROFILE}
              component={() => (
                <div className="App-content_small-header">
                  <Profile />
                </div>
              )}
            />
          </div>
        </div>
      </Router>
    );
  }
}

export default App;
```

The Organization component wasn't implemented yet, but you can start with a functional stateless component in the *src/Organization/index.js* file, that acts as a placeholder to keep the application working for now.

```
import React from 'react';

const Organization = () => <div>Organization</div>;

export default Organization;
```

Since you mapped both routes to their respective components, so you want to implement navigation from one route to another. For this, introduce a **Navigation** component in the App component.

```
...

import Navigation from './Navigation';
import Profile from '../Profile';
import Organization from '../Organization';

...

class App extends Component {
  render() {
    return (
      <Router>
        <div className="App">
          <Navigation />

          <div className="App-main">
            ...
          </div>
        </div>
      </Router>
    );
  }
}

export default App;
```

Next, we'll implement the Navigation component, which is responsible for displaying the two links to navigate between your routes using React Router's Link component.

```
import React from 'react';
import { Link } from 'react-router-dom';

import * as routes from '../../constants/routes';

import './style.css';

const Navigation = () => (
  <header className="Navigation">
    <div className="Navigation-link">
      <Link to={routes.PROFILE}>Profile</Link>
    </div>
    <div className="Navigation-link">
      <Link to={routes.ORGANIZATION}>Organization</Link>
    </div>
  </header>
);
```

```
export default Navigation;
```

The Profile page works as before, but the Organization page is empty. In the last step, you defined the two routes as constants, used them in the App component to map to their respective components, and introduced Link components to navigate to them in the Navigation component.

Another great feature of the Apollo Client is that it caches query requests. When navigating from the Profile page to the Organization page and back to the Profile page, the results appear immediately because the Apollo Client checks its cache before making the query to the remote GraphQL API. It's a pretty powerful tool.

The next part of this section is the Organization component. It is the same as the Profile component, except the query differs because it takes a variable for the organization name to identify the organization's repositories.

```
import React from 'react';
import gql from 'graphql-tag';
import { Query } from 'react-apollo';

import { REPOSITORY_FRAGMENT } from '../Repository';

const GET_REPOSITORIES_OF_ORGANIZATION = gql`
  query($organizationName: String!) {
    organization(login: $organizationName) {
      repositories(first: 5) {
        edges {
          node {
            ...repository
          }
        }
      }
    }
  }
  ${REPOSITORY_FRAGMENT}
`;

const Organization = ({ organizationName }) => (
  <Query
    query={GET_REPOSITORIES_OF_ORGANIZATION}
    variables={{
      organizationName,
    }}
    skip={organizationName === ''}
  >
    {({ data, loading, error }) => {
      ...
    }}
  </Query>
```

```
);

export default Organization;
```

The Query component in the Organization component takes a query tailored to the organization being the top level field of the query. It takes a variable to identify the organization, and it uses the newly introduced `skip` prop to skip executing the query if no organization identifier is provided. Later, you will pass an organization identifier from the App component. You may have noticed that the repository fragment you introduced earlier to update the local state in the cache can be reused here. It saves lines of code, and more importantly, ensures the returned list of repositories have identical structures to the list of repositories in the Profile component.

Next, extend the query to fit the requirements of the pagination feature. It requires the `cursor` argument to identify the next page of repositories. The `notifyOnNetworkStatusChange` prop is used to update the `loading` boolean for paginated requests as well.

```
...

const GET_REPOSITORIES_OF_ORGANIZATION = gql`
  query($organizationName: String!, $cursor: String) {
    organization(login: $organizationName) {
      repositories(first: 5, after: $cursor) {
        edges {
          node {
            ...repository
          }
        }
        pageInfo {
          endCursor
          hasNextPage
        }
      }
    }
  }
  ${REPOSITORY_FRAGMENT}
`;

const Organization = ({ organizationName }) => (
  <Query
    query={GET_REPOSITORIES_OF_ORGANIZATION}
    variables={{
      organizationName,
    }}
    skip={organizationName === ''}
    notifyOnNetworkStatusChange={true}
  >
    {({ data, loading, error, fetchMore }) => {
      ...
    }}
```

```
  </Query>
);

export default Organization;
```

Lastly, the render prop child function needs to be implemented. It doesn't differ much from the Query's content in the Profile component. Its purpose is to handle edge cases like loading and 'no data' errors, and eventually, to show a list of repositories. Because the RepositoryList component handles the pagination feature, this improvement is included in the newly implemented Organization component.

```
...

import RepositoryList, { REPOSITORY_FRAGMENT } from '../Repository';
import Loading from '../Loading';
import ErrorMessage from '../Error';

...

const Organization = ({ organizationName }) => (
  <Query ... >
    {({ data, loading, error, fetchMore }) => {
      if (error) {
        return <ErrorMessage error={error} />;
      }

      const { organization } = data;

      if (loading && !organization) {
        return <Loading />;
      }

      return (
        <RepositoryList
          loading={loading}
          repositories={organization.repositories}
          fetchMore={fetchMore}
        />
      );
    }}
  </Query>
);

export default Organization;
```

Provide a `organizationName` as prop when using the Organization in the App component, and leave it inlined for now. Later, you will make it dynamic with a search field.

```
class App extends Component {
  render() {
    return (
      <Router>
        <div className="App">
          <Navigation />

          <div className="App-main">
            <Route
              exact
              path={routes.ORGANIZATION}
              component={() => (
                <div className="App-content_large-header">
                  <Organization
                    organizationName={'the-road-to-learn-react'}
                  />
                </div>
              )}
            />
            ...
          </div>
        </div>
      </Router>
    );
  }
}
```

The Organization component should almost work now, as the `More` button is the only incomplete part. The remaining issue is the resolving block for the pagination feature in the `updateQuery` function. It assumes that the nested data structure always starts with a `viewer` object. It does for the Profile page, but not for the Organization page. There the top level object is the `organization` followed by the list of `repositories`. Only the top level object changes from page to page, where the underlying structure stays identical.

When the top level object changes from page to page, the ideal next step is to tell the RepositoryList component its top level object from the outside. With the Organization component, its the top-level object `organization`, which could be passed as a string and reused as a dynamic key later:

```
const Organization = ({ organizationName }) => (
  <Query ... >
    {({ data, loading, error, fetchMore }) => {
      ...

      return (
        <RepositoryList
          loading={loading}
          repositories={organization.repositories}
```

```
          fetchMore={fetchMore}
          entry={'organization'}
        />
      );
    }}
  </Query>
);
```

With the Profile component, the `viewer` would be the top level object:

```
const Profile = () => (
  <Query ... >
    {({ data, loading, error, fetchMore }) => {
      ...

      return (
        <RepositoryList
          loading={loading}
          repositories={viewer.repositories}
          fetchMore={fetchMore}
          entry={'viewer'}
        />
      );
    }}
  </Query>
);
```



Now you can handle the new case in the RepositoryList component by passing the entry as computed property name to the `updateQuery` function. Instead of passing the `updateQuery` function directly to the FetchMore component, it can be derived from a higher-order function needed to pass the new `entry` property.

```
const RepositoryList = ({
  repositories,
  loading,
  fetchMore,
  entry,
}) => (
  <Fragment>
    ...

    <FetchMore
      loading={loading}
      hasNextPage={repositories.pageInfo.hasNextPage}
      variables={{
        cursor: repositories.pageInfo.endCursor,
      }}
      updateQuery={getUpdateQuery(entry)}
      fetchMore={fetchMore}
```

```
      >
        Repositories
      </FetchMore>
    </Fragment>
  );
```

The higher-order function next to the RepositoryList component is completed as such:

```
const getUpdateQuery = entry => (
  previousResult,
  { fetchMoreResult },
) => {
  if (!fetchMoreResult) {
    return previousResult;
  }

  return {
    ...previousResult,
    [entry]: {
      ...previousResult[entry],
      repositories: {
        ...previousResult[entry].repositories,
        ...fetchMoreResult[entry].repositories,
        edges: [
          ...previousResult[entry].repositories.edges,
          ...fetchMoreResult[entry].repositories.edges,
        ],
      },
    },
  };
};
```

That's how a deeply-nested object is updated with the `fetchMoreResult`, even though the top level component from the query result is not static. The pagination feature should work on both pages now. Take a moment to recap the last implementations again and why these were necessary.

Next, we'll implement the search function I mentioned earlier. The best place to add the search field would be the Navigation component, but only when the Organization page is active. React Router comes with a useful higher-order component to access to the current URL, which can be used to show a search field.

```
import React from 'react';
import { Link, withRouter } from 'react-router-dom';

import * as routes from '../../constants/routes';

import './style.css';
```

```
const Navigation = ({
  location: { pathname },
}) => (
  <header className="Navigation">
    <div className="Navigation-link">
      <Link to={routes.PROFILE}>Profile</Link>
    </div>
    <div className="Navigation-link">
      <Link to={routes.ORGANIZATION}>Organization</Link>
    </div>

    {pathname === routes.ORGANIZATION && (
      <OrganizationSearch />
    )}
  </header>
);

export default withRouter(Navigation);
```

The OrganizationSearch component is implemented next to the Navigation component in the next steps. Before that can work, there needs to be some kind of initial state for the OrganizationSearch, as well as a callback function to update the initial state in the Navigation component. To accommodate this, the Navigation component becomes a class component.

```
...

class Navigation extends React.Component {
  state = {
    organizationName: 'the-road-to-learn-react',
  };

  onOrganizationSearch = value => {
    this.setState({ organizationName: value });
  };

  render() {
    const { location: { pathname } } = this.props;

    return (
      <header className="Navigation">
        <div className="Navigation-link">
          <Link to={routes.PROFILE}>Profile</Link>
        </div>
        <div className="Navigation-link">
          <Link to={routes.ORGANIZATION}>Organization</Link>
        </div>

        {pathname === routes.ORGANIZATION && (
          <OrganizationSearch
            organizationName={this.state.organizationName}
```

```
            onOrganizationSearch={this.onOrganizationSearch}
          />
        )}
      </header>
    );
  }
}

export default withRouter(Navigation);
```

The OrganizationSearch component implemented in the same file would also work with the following implementation. It handles its own local state, the value that shows up in the input field, but uses it as an initial value from the parent component. It also receives a callback handler, which can be used in the `onSubmit()` class method to propagate the search fields value on a submit interaction up the component tree.



```
...

import Button from '../../Button';
import Input from '../../Input';

import './style.css';

const Navigation = ({ ... }) => ...

class OrganizationSearch extends React.Component {
  state = {
    value: this.props.organizationName,
  };

  onChange = event => {
    this.setState({ value: event.target.value });
  };

  onSubmit = event => {
    this.props.onOrganizationSearch(this.state.value);

    event.preventDefault();
  };

  render() {
    const { value } = this.state;

    return (
      <div className="Navigation-search">
        <form onSubmit={this.onSubmit}>
          <Input
            color={'white'}
            type="text"
            value={value}
            onChange={this.onChange}
```

```
          />{' '}
          <Button color={'white'} type="submit">
            Search
          </Button>
        </form>
      </div>
    );
  }
}

export default withRouter(Navigation);
```

The Input component is a slightly styled input element that is defined in *src/Input/index.js* as its own component.

```
import React from 'react';

import './style.css';

const Input = ({ children, color = 'black', ...props }) => (
  <input className={`Input Input_${color}`} {...props}>
    {children}
  </input>
);

export default Input;
```

While the search field works in the Navigation component, it doesn't help the rest of the application. It only updates the state in the Navigation component when a search request is submitted. However, the value of the search request is needed in the Organization component as a GraphQL variable for the query, so the local state needs to be lifted up from the Navigation component to the App component. The Navigation component becomes a stateless functional component again.

```
const Navigation = ({
  location: { pathname },
  organizationName,
  onOrganizationSearch,
}) => (
  <header className="Navigation">
    <div className="Navigation-link">
      <Link to={routes.PROFILE}>Profile</Link>
    </div>
    <div className="Navigation-link">
      <Link to={routes.ORGANIZATION}>Organization</Link>
    </div>

    {pathname === routes.ORGANIZATION && (
```

```
        <OrganizationSearch
          organizationName={organizationName}
          onOrganizationSearch={onOrganizationSearch}
        />
      )}
    </header>
  );
```

The App component takes over the responsibility from the Navigation component, managing the local state, passing the initial state and a callback function to update the state to the Navigation component, and passing the state itself to the Organization component to perform the query:

```
...

class App extends Component {
  state = {
    organizationName: 'the-road-to-learn-react',
  };

  onOrganizationSearch = value => {
    this.setState({ organizationName: value });
  };

  render() {
    const { organizationName } = this.state;

    return (
      <Router>
        <div className="App">
          <Navigation
            organizationName={organizationName}
            onOrganizationSearch={this.onOrganizationSearch}
          />

          <div className="App-main">
            <Route
              exact
              path={routes.ORGANIZATION}
              component={() => (
                <div className="App-content_large-header">
                  <Organization organizationName={organizationName} />
                </div>
              )}
            />
            ...
          </div>
        </div>
      </Router>
    );
  }
}
```

```
export default App;
```

You have implemented a dynamic GraphQL query with a search field. Once a new `organizationName` is passed to the Organization component from a local state change, the Query component triggers another request due to a re-render. The request is not always made to the remote GraphQL API, though. The Apollo Client cache is used when an organization is searched twice. Also, you have used the well-known technique called lifting state in React to share the state across components.

Exercises:

- Confirm your source code for the last section
  - Confirm the changes from the last section
- If you are not familiar with React Router, try it out in this pragmatic tutorial
- Invest 3 minutes of your time and take the quiz



IMPLEMENTING THE ISSUES FEATURE: SETUP

In the previous sections you have implemented most of the common Apollo Client features in your React application. Now you can start implementing extensions for the application on your own. This section showcases how a full-fledged feature can be implemented with Apollo Client in React.

So far, you have dealt with GitHub repositories from organizations and your account. This will take that one step further, fetching GitHub issues that are made available using a list field associated to a repository in a GraphQL query. However, this section doesn't only show you how to render a nested list field in your React application.

The foundation will be rendering the list of issues. You will implement client-side filtering with plain React to show opened, closed, or no issue. Finally, you will refactor the filtering to a server-side filtering using GraphQL queries. We will only fetch the issues by their state from the server rather than filtering the issue's state on the client-side. Implementing pagination for the issues will be your exercise.

First, render a new component called 'Issues' in your RepositoryList component. This component takes two props that are used later in a GraphQL query to identify the repository from which you want to fetch the issues.

```
...
```

```
import FetchMore from '../../FetchMore';
import RepositoryItem from '../RepositoryItem';
import Issues from '../../Issue';

...

const RepositoryList = ({
  repositories,
  loading,
  fetchMore,
  entry,
}) => (
  <Fragment>
    {repositories.edges.map(({ node }) => (
      <div key={node.id} className="RepositoryItem">
        <RepositoryItem {...node} />

        <Issues
          repositoryName={node.name}
          repositoryOwner={node.owner.login}
        />
      </div>
    ))}

    ...
  </Fragment>
);

export default RepositoryList;
```

In the *src/Issue/index.js* file, import and export the Issues component. Since the issue feature can be kept in a module on its own, it has this *index.js* file again. That's how you can tell other developers to access only this feature module, using the *index.js* file as its interface. Everything else is kept private.

```
import Issues from './IssueList';

export default Issues;
```

Note how the component is named Issues, not IssueList. The naming convention is used to break down the rendering of a list of items: Issues, IssueList and IssueItem. Issues is the container component, where you query the data and filter the issues, and the IssueList and IssueItem are only there as presentational components for rendering. In contrast, the Repository feature module hasn't a Repositories component, because there was no need for it. The list of repositories already came from the Organization and Profile components and the Repository module's

components are mainly only there for the rendering. This is only one opinionated approach of naming the components, however.

Let's start implementing Issues and IssueList components in the *src/Issue/IssueList/index.js* file. You could argue to split both components up into their own files, but for the sake of this tutorial, they are kept together in one file.

First, there needs to be a new query for the issues. You might wonder: Why do we need a new query here? It would be simpler to include the issues list field in the query at the top next to the Organization and Profile components. That's true, but it comes with a cost. Adding more nested (list) fields to a query often results into performance issues on the server-side. There you may have to make multiple roundtrips to retrieve all the entities from the database.

- Roundtrip 1: get organization by name
- Roundtrip 2: get repositories of organization by organization identifier
- Roundtrip 3: get issues of repository by repository identifier

It is simple to conclude that nesting queries in a naive way solves all of our problems. Whereas it solves the problem of only requesting the data once and not with multiple network request (similar roundtrips as shown for the database), GraphQL doesn't solve the problem of retrieving all the data from the database for you. That's not the responsibility of GraphQL after all. So by having a dedicated query in the Issues component, you can decide **when** to trigger this query. In the next steps, you will just trigger it on render because the Query component is used. But when adding the client-side filter later on, it will only be triggered when the "Filter" button is toggled. Otherwise the issues should be hidden. Finally, that's how all the initial data loading can be delayed to a point when the user actually wants to see the data.

First, define the Issues component which has access to the props which were passed in the RepositoryList component. It doesn't render much yet.

```
import React from 'react';

import './style.css';

const Issues = ({ repositoryOwner, repositoryName }) =>
  <div className="Issues">
  </div>

export default Issues;
```

Second, define the query in the *src/Issue/IssueList/index.js* file to retrieve issues of a repository. The repository is identified by its owner and name. Also, add the `state` field as one of the fields

for the query result. This is used for client-side filtering, for showing issues with an open or closed state.

```
import React from 'react';
import gql from 'graphql-tag';

import './style.css';

const GET_ISSUES_OF_REPOSITORY = gql`
  query($repositoryOwner: String!, $repositoryName: String!) {
    repository(name: $repositoryName, owner: $repositoryOwner) {
      issues(first: 5) {
        edges {
          node {
            id
            number
            state
            title
            url
            bodyHTML
          }
        }
      }
    }
  }
`;

...
```



Third, introduce the Query component and pass it the previously defined query and the necessary variables. Use its render prop child function to access the data, to cover all edge cases and to render a IssueList component eventually.

```
import React from 'react';
import { Query } from 'react-apollo';
import gql from 'graphql-tag';

import IssueItem from '../IssueItem';
import Loading from '../../Loading';
import ErrorMessage from '../../Error';

import './style.css';

const Issues = ({ repositoryOwner, repositoryName }) => (
  <div className="Issues">
    <Query
      query={GET_ISSUES_OF_REPOSITORY}
      variables={{
        repositoryOwner,
        repositoryName,
```

```
      }}
    >
      {({ data, loading, error }) => {
        if (error) {
          return <ErrorMessage error={error} />;
        }

        const { repository } = data;

        if (loading && !repository) {
          return <Loading />;
        }

        if (!repository.issues.edges.length) {
          return <div className="IssueList">No issues ...</div>;
        }

        return <IssueList issues={repository.issues} />;
      }}
    </Query>
  </div>
);

const IssueList = ({ issues }) => (
  <div className="IssueList">
    {issues.edges.map(({ node }) => (
      <IssueItem key={node.id} issue={node} />
    ))}
  </div>
);

export default Issues;
```

Finally, implement a basic IssueItem component in the *src/Issue/IssueItem/index.js* file. The snippet below shows a placeholder where you can implement the Commenting feature, which we'll cover later.

```
import React from 'react';

import Link from '../../Link';

import './style.css';

const IssueItem = ({ issue }) => (
  <div className="IssueItem">
    {/* placeholder to add a show/hide comment button later */}

    <div className="IssueItem-content">
      <h3>
        <Link href={issue.url}>{issue.title}</Link>
      </h3>
      <div dangerouslySetInnerHTML={{ __html: issue.bodyHTML }} />
```

```
      {/* placeholder to render a list of comments later */}
    </div>
  </div>
);

export default IssueItem;
```

Once you start your application again, you should see the initial page of paginated issues rendered below each repository. That's a performance bottleneck. Worse, the GraphQL requests are not bundled in one request, as with the issues list field in the Organization and Profile components. In the next steps you are implementing client-side filtering. The default is to show no issues, but it can toggle between states of showing none, open issues, and closed issues using a button, so the issues will not be queried before toggling one of the issue states.

Exercises:

- Confirm your source code for the last section
- Confirm the changes from the last section

Read more about the rate limit when using a (or in this case GitHub's) GraphQL API

IMPLEMENTING THE ISSUES FEATURE: CLIENT-SIDE FILTER

In this section, we enhance the Issue feature with client-side filtering. It prevents the initial issue querying because it happens with a button, and it lets the user filter between closed and open issues.

First, let's introduce our three states as enumeration next to the Issues component. The `NONE` state is used to show no issues; otherwise, the other states are used to show open or closed issues.

```
const ISSUE_STATES = {
  NONE: 'NONE',
  OPEN: 'OPEN',
  CLOSED: 'CLOSED',
};
```

Second, let's implement a short function that decides whether it is a state to show the issues or not. This function can be defined in the same file.

```
const isShow = issueState => issueState !== ISSUE_STATES.NONE;
```

Third, the function can be used for conditional rendering, to either query the issues and show the IssueList, or to do nothing. It's not clear yet where the `issueState` property comes from.

```
const Issues = ({ repositoryOwner, repositoryName }) => (
  <div className="Issues">
    {isShow(issueState) && (
      <Query ... >
        ...
      </Query>
    )}
  </div>
);
```

The `issueState` property must come from the local state to toggle it via a button in the component, so the Issues component must be refactored to a class component to manage this state.



```
class Issues extends React.Component {
  state = {
    issueState: ISSUE_STATES.NONE,
  };

  render() {
    const { issueState } = this.state;
    const { repositoryOwner, repositoryName } = this.props;

    return (
      <div className="Issues">
        {isShow(issueState) && (
          <Query ... >
            ...
          </Query>
        )}
      </div>
    );
  }
}
```

The application should be error-free now, because the initial state is set to `NONE` and the conditional rendering prevents the query and the rendering of a result. However, the client-side filtering is not done yet, as you still need to toggle the `issueState` property with React's local state. The ButtonUnobtrusive component has the appropriate style, so we can reuse it to implement this toggling behavior to transition between the three available states.

```
...

import IssueItem from '../IssueItem';
import Loading from '../../Loading';
import ErrorMessage from '../../Error';
import { ButtonUnobtrusive } from '../../Button';

class Issues extends React.Component {
  state = {
    issueState: ISSUE_STATES.NONE,
  };

  onChangeIssueState = nextIssueState => {
    this.setState({ issueState: nextIssueState });
  };

  render() {
    const { issueState } = this.state;
    const { repositoryOwner, repositoryName } = this.props;

    return (
      <div className="Issues">
        <ButtonUnobtrusive
          onClick={() =>
            this.onChangeIssueState(TRANSITION_STATE[issueState])
          }
        >
          {TRANSITION_LABELS[issueState]}
        </ButtonUnobtrusive>

        {isShow(issueState) && (
          <Query ... >
            ...
          </Query>
        )}
      </div>
    );
  }
}
```

In the last step, you introduced the button to toggle between the three states. You used two enumerations, `TRANSITION_LABELS` and `TRANSITION_STATE`, to show an appropriate button label and to define the next state after a state transition. These enumerations can be defined next to the `ISSUE_STATES` enumeration.

```
const TRANSITION_LABELS = {
  [ISSUE_STATES.NONE]: 'Show Open Issues',
  [ISSUE_STATES.OPEN]: 'Show Closed Issues',
  [ISSUE_STATES.CLOSED]: 'Hide Issues',
};
```

```
const TRANSITION_STATE = {
  [ISSUE_STATES.NONE]: ISSUE_STATES.OPEN,
  [ISSUE_STATES.OPEN]: ISSUE_STATES.CLOSED,
  [ISSUE_STATES.CLOSED]: ISSUE_STATES.NONE,
};
```

As you can see, whereas the former enumeration only matches a label to a given state, the latter enumeration matches the next state to a given state. That's how the toggling to a next state can be made simple. Last but not least, the `issueState` from the local state has to be used to filter the list of issues after they have been queried and should be rendered.

```
class Issues extends React.Component {
  ...

  render() {
    ...

    return (
      <div className="Issues">
        ...

        {isShow(issueState) && (
          <Query ... >
            {({ data, loading, error }) => {
              if (error) {
                return <ErrorMessage error={error} />;
              }

              const { repository } = data;

              if (loading && !repository) {
                return <Loading />;
              }

              const filteredRepository = {
                issues: {
                  edges: repository.issues.edges.filter(
                    issue => issue.node.state === issueState,
                  ),
                },
              };

              if (!filteredRepository.issues.edges.length) {
                return <div className="IssueList">No issues ...</div>;
              }

              return (
                <IssueList issues={filteredRepository.issues} />
              );
            }}
```

```
          </Query>
        )}
      </div>
    );
  }
}
```

You have implemented client-side filtering. The button is used to toggle between the three states managed in the local state of the component. The issues are only queried in filtered and rendered states. In the next step, the existing client-side filtering should be advanced to a server-side filtering, which means the filtered issues are already requested from the server and not filtered afterward on the client.

Exercises:

- Confirm your source code for the last section
  - Confirm the changes from the last section
- Install the recompose library which implements many higher-order components
- Refactor the Issues component from class component to functional stateless component
- Use the `withState` HOC for the Issues component to manage the `issueState`

IMPLEMENTING THE ISSUES FEATURE: SERVER-SIDE FILTER

Before starting with the server-side filtering, let's recap the last exercise in case you had difficulties with it. Basically you can perform the refactoring in three steps. First, install recompose as package for your application on the command line:

```
npm install recompose --save
```

Second, import the `withState` higher-order component in the *src/Issue/IssueList/index.js* file and use it to wrap your exported Issues component, where the first argument is the property name in the local state, the second argument is the handler to change the property in the local state, and the third argument is the initial state for that property.

```
import React from 'react';
import { Query } from 'react-apollo';
import gql from 'graphql-tag';
import { withState } from 'recompose';
```

```
...

export default withState(
  'issueState',
  'onChangeIssueState',
  ISSUE_STATES.NONE,
)(Issues);
```

Finally, refactor the Issues component from a class component to a functional stateless component. It accesses the `issueState` and `onChangeIssueState()` function in its props now. Remember to change the usage of the `onChangeIssueState` prop to being a function and not a class method anymore.

```
...

const Issues = ({
  repositoryOwner,
  repositoryName,
  issueState,
  onChangeIssueState,
}) => (
  <div className="Issues">
    <ButtonUnobtrusive
      onClick={() => onChangeIssueState(TRANSITION_STATE[issueState])}
    >
      {TRANSITION_LABELS[issueState]}
    </ButtonUnobtrusive>

    ...
  </div>
);

...
```

The previous section makes writing stateful components, where the state is much more convenient. Next, advance the filtering from client-side to server-side. We use the defined GraphQL query and its arguments to make a more exact query by requesting only open or closed issues. In the *src/Issue/IssueList/index.js* file, extend the query with a variable to specify the issue state:

```
const GET_ISSUES_OF_REPOSITORY = gql`
  query(
    $repositoryOwner: String!
    $repositoryName: String!
    $issueState: IssueState!
  ) {
    repository(name: $repositoryName, owner: $repositoryOwner) {
```

```
      issues(first: 5, states: [$issueState]) {
        edges {
          node {
            id
            number
            state
            title
            url
            bodyHTML
          }
        }
      }
    }
  }
`;
```

Next, you can use the `issueState` property as variable for your Query component. In addition, remove the client-side filter logic from the Query component's render prop function.

```
const Issues = ({
  repositoryOwner,
  repositoryName,
  issueState,
  onChangeIssueState,
}) => (
  <div className="Issues">
    ...

    {isShow(issueState) && (
      <Query
        query={GET_ISSUES_OF_REPOSITORY}
        variables={{
          repositoryOwner,
          repositoryName,
          issueState,
        }}
      >
        {({ data, loading, error }) => {
          if (error) {
            return <ErrorMessage error={error} />;
          }

          const { repository } = data;

          if (loading && !repository) {
            return <Loading />;
          }

          return <IssueList issues={repository.issues} />;
        }}
      </Query>
    )}
```

```
    </div>
  );
```

You are only querying open or closed issues. Your query became more exact, and the filtering is no longer handled by the client.

Exercises:

- Confirm your source code for the last section
  - Confirm the changes from the last section
- Implement the pagination feature for the Issue feature
  - Add the pageInfo information to the query
  - Add the additional cursor variable and argument to the query
  - Add the FetchMore component to the IssueList component

APOLLO CLIENT PREFETCHING IN REACT

This section is all about prefetching data, though the user doesn't need it immediately. It is another UX technique that can be deployed to the optimistic UI technique you used earlier. You will implement the prefetching data feature for the list of issues, but feel free to implement it for other data fetching later as your exercise.

When your application renders for the first time, there no issues fetched, so no issues are rendered. The user has to toggle the filter button to fetch open issues, and do it again to fetch closed issues. The third click will hide the list of issues again. The goal of this section is to prefetch the next bulk of issues when the user hovers the filter button. For instance, when the issues are still hidden and the user hovers the filter button, the issues with the open state are prefetched in the background. When the user clicks the button, there is no waiting time, because the issues with the open state are already there. The same scenario applies for the transition from open to closed issues. To prepare this behavior, split out the filter button as its own component in the *src/Issue/IssueList/index.js* file:

```
const Issues = ({
  repositoryOwner,
  repositoryName,
  issueState,
  onChangeIssueState,
}) => (
  <div className="Issues">
    <IssueFilter
      issueState={issueState}
```

```
        onChangeIssueState={onChangeIssueState}
      />

      {isShow(issueState) && (
        ...
      )}
    </div>
  );

const IssueFilter = ({ issueState, onChangeIssueState }) => (
  <ButtonUnobtrusive
    onClick={() => onChangeIssueState(TRANSITION_STATE[issueState])}
  >
    {TRANSITION_LABELS[issueState]}
  </ButtonUnobtrusive>
);
```

Now it is easier to focus on the IssueFilter component where most of the logic for data prefetching is implemented. Like before, the prefetching should happen when the user hovers over the button. There needs to be a prop for it, and a callback function which is executed when the user hovers over it. There is such a prop (attribute) for a button (element). We are dealing with HTML elements here.



```
const prefetchIssues = () => {};

...

const IssueFilter = ({ issueState, onChangeIssueState }) => (
  <ButtonUnobtrusive
    onClick={() => onChangeIssueState(TRANSITION_STATE[issueState])}
    onMouseOver={prefetchIssues}
  >
    {TRANSITION_LABELS[issueState]}
  </ButtonUnobtrusive>
);
```

The `prefetchIssue()` function has to execute the identical GraphQL query executed by the Query component in the Issues component, but this time it is done in an imperative way instead of declarative. Rather than using the Query component for it, use the the Apollo Client instance directly to execute a query. Remember, the Apollo Client instance is hidden in the component tree, because you used React's Context API to provide the Apollo Client instance the component tree's top level. The Query and Mutation components have access to the Apollo Client, even though you have never used it yourself directly. However, this time you use it to query the prefetched data. Use the ApolloConsumer component from the React Apollo package to expose the Apollo Client instance in your component tree. You have used the ApolloProvider somewhere to provide the client instance, and you can use the ApolloConsumer to retrieve it now. In the

src/Issue/IssueList/index.js file, import the ApolloConsumer component and use it in the IssueFilter component. It gives you access to the Apollo Client instance via its render props child function.

```
import React from 'react';
import { Query, ApolloConsumer } from 'react-apollo';
import gql from 'graphql-tag';
import { withState } from 'recompose';

...

const IssueFilter = ({ issueState, onChangeIssueState }) => (
  <ApolloConsumer>
    {client => (
      <ButtonUnobtrusive
        onClick={() =>
          onChangeIssueState(TRANSITION_STATE[issueState])
        }
        onMouseOver={() => prefetchIssues(client)}
      >
        {TRANSITION_LABELS[issueState]}
      </ButtonUnobtrusive>
    )}
  </ApolloConsumer>
);
```

Now you have access to the Apollo Client instance to perform queries and mutations, which will enable you to query GitHub's GraphQL API imperatively. The variables needed to perform the prefetching of issues are the same ones used in the Query component. You need to pass those to the IssueFilter component, and then to the `prefetchIssues()` function.

```
...

const Issues = ({
  repositoryOwner,
  repositoryName,
  issueState,
  onChangeIssueState,
}) => (
  <div className="Issues">
    <IssueFilter
      repositoryOwner={repositoryOwner}
      repositoryName={repositoryName}
      issueState={issueState}
      onChangeIssueState={onChangeIssueState}
    />

    {isShow(issueState) && (
      ...
    )}
  </div>
```

```
);

const IssueFilter = ({
  repositoryOwner,
  repositoryName,
  issueState,
  onChangeIssueState,
}) => (
  <ApolloConsumer>
    {client => (
      <ButtonUnobtrusive
        onClick={() =>
          onChangeIssueState(TRANSITION_STATE[issueState])
        }
        onMouseOver={() =>
          prefetchIssues(
            client,
            repositoryOwner,
            repositoryName,
            issueState,
          )
        }
      >
        {TRANSITION_LABELS[issueState]}
      </ButtonUnobtrusive>
    )}
  </ApolloConsumer>
);

...
```

Use this information to perform the prefetching data query. The Apollo Client instance exposes a `query()` method for this. Make sure to retrieve the next `issueState`, because when prefetching open issues, the current `issueState` should be `NONE`.

```
const prefetchIssues = (
  client,
  repositoryOwner,
  repositoryName,
  issueState,
) => {
  const nextIssueState = TRANSITION_STATE[issueState];

  if (isShow(nextIssueState)) {
    client.query({
      query: GET_ISSUES_OF_REPOSITORY,
      variables: {
        repositoryOwner,
        repositoryName,
        issueState: nextIssueState,
      },
    });
```

```
    }
};
```

That's it. Once the button is hovered, it should prefetch the issues for the next `issueState`. The Apollo Client makes sure that the new data is updated in the cache like it would do for the Query component. There shouldn't be any visible loading indicator in between except when the network request takes too long and you click the button right after hovering it. You can verify that the request is happening in your network tab in the developer development tools of your browser. In the end, you have learned about two UX improvements that can be achieved with ease when using Apollo Client: optimistic UI and prefetching data.

Exercises:

- Confirm your source code for the last section
  - Confirm the changes from the last section
- Read more about Apollo Prefetching and Query Splitting in React
- Invest 3 minutes of your time and take the quiz

EXERCISE: COMMENTING FEATURE

This last section is for hands-on experience with the application and implementing features yourself. I encourage you to continue implementing features for the application and improving it. There are a couple of guiding points to help you implementing the Commenting feature. In the end it should be possible to show a list of paginated comments per issue on demand. Finally, a user should be able to leave a comment. The source code of the implemented feature can be found here.

- Introduce components for fetching a list of comments (e.g. Comments), rendering a list of comments (e.g. CommentList), and rendering a single comment (e.g. CommentItem). They can render sample data for now.

- Use the top level comments component (e.g. Comments), which will be your container component that is responsible to query the list of comments, in the *src/Issue/IssueItem/index.js* file. In addition, add a toggle to either show or hide comments. The IssueItem component has to become a class component or needs to make use of the `withState` HOC from the recompose library.

- Use the Query component from React Apollo in your container Comments component to fetch a list of comments. It should be similar to the query that fetches the list of issues. You only need to

identify the issue for which the comments should be fetched.

- Handle all edge cases in the Comments to show loading indicator, no data, or error messages. Render the list of comments in the CommentList component and a single comment in the CommentItem component.
- Implement the pagination feature for comments. Add the necessary fields in the query, the additional props and variables to the Query component, and the reusable FetchMore component. Handle the merging of the state in the `updateQuery` prop.
- Enable prefetching of the comments when hovering the "Show/Hide Comments" button.
- Implement an AddComment component that shows a `textarea` and a `submit` button to enable user comments. Use the `addComment` mutation from GitHub's GraphQL API and the Mutation component from React Apollo to execute the mutation with the submit button.
- Improve the AddComment component with the optimistic UI feature (perhaps read again the Apollo documentation about the optimistic UI with a list of items). A comment should show up in the list of comments, even if the request is pending.

I hope this section, building your own feature in the application with all the learned tools and techniques, matched your skills and challenged you to implement React applications with Apollo and GraphQL. I would recommend working to improve and extend the existing application. If you haven't implemented a GraphQL server yet, find other third-party APIs that offer a GraphQL API and build your own React with Apollo application by consuming it. Keep yourself challenged to grow your skills as a developer.

APPENDIX: CSS FILES AND STYLES

This section has all the CSS files as well as their content and locations, to give your React with GraphQL and Apollo Client application a nice touch. It even makes it responsive for mobile and tablet devices. These are only recommendations, though; you can experiment with them, or come up with your own styles.

src/style.css

```
#root,
html,
body {
  height: 100%;
}
```

```
body {
  margin: 0;
  padding: 0;
  font-family: 'Source Sans Pro', sans-serif;
  font-weight: 200;
  text-rendering: optimizeLegibility;
}

h2 {
  font-size: 24px;
  font-weight: 600;
  line-height: 34px;
  margin: 5px 0;
}

h3 {
  font-size: 20px;
  font-weight: 400;
  line-height: 27px;
  margin: 5px 0;
}

ul,
li {
  list-style: none;
  padding-left: 0;
}

a {
  text-decoration: none;
  color: #000;
  opacity: 1;
  transition: opacity 0.25s ease-in-out;
}

a:hover {
  opacity: 0.35;
  text-decoration: none;
}

a:active {
  text-decoration: none;
}

pre {
  white-space: pre-wrap;
}
```

src/App/style.css

```
.App {
  min-height: 100%;
```

```
  display: flex;
  flex-direction: column;
}

.App-main {
  flex: 1;
}

.App-content_large-header,
.App-content_small-header {
  margin-top: 54px;
}

@media only screen and (max-device-width: 480px) {
  .App-content_large-header {
    margin-top: 123px;
  }

  .App-content_small-header {
    margin-top: 68px;
  }
}
```

src/App/Navigation/style.css

```
.Navigation {
  overflow: hidden;
  position: fixed;
  top: 0;
  width: 100%;
  z-index: 1;
  background-color: #24292e;
  display: flex;
  align-items: baseline;
}

@media only screen and (max-device-width: 480px) {
  .Navigation {
    flex-direction: column;
    justify-content: center;
    align-items: center;
  }
}

.Navigation-link {
  font-size: 12px;
  letter-spacing: 3.5px;
  font-weight: 500;
  text-transform: uppercase;
  padding: 20px;
  text-decoration: none;
}
```

```
.Navigation-link a {
  color: #ffffff;
}

.Navigation-search {
  padding: 0 10px;
}

@media only screen and (max-device-width: 480px) {
  .Navigation-link {
    padding: 10px;
  }

  .Navigation-search {
    padding: 10px 10px;
  }
}
```

src/Button/style.css

```
.Button {
  padding: 10px;
  background: none;
  cursor: pointer;
  transition: color 0.25s ease-in-out;
  transition: background 0.25s ease-in-out;
}

.Button_white {
  border: 1px solid #fff;
  color: #fff;
}

.Button_white:hover {
  color: #000;
  background: #fff;
}

.Button_black {
  border: 1px solid #000;
  color: #000;
}

.Button_black:hover {
  color: #fff;
  background: #000;
}

.Button_unobtrusive {
  padding: 0;
  color: #000;
  background: none;
  border: none;
```

```
  cursor: pointer;
  opacity: 1;
  transition: opacity 0.25s ease-in-out;
  outline: none;
}

.Button_unobtrusive:hover {
  opacity: 0.35;
}

.Button_unobtrusive:focus {
  outline: none;
}
```

src/Error/style.css

```
.ErrorMessage {
  margin: 20px;
  display: flex;
  justify-content: center;
}
```

src/FetchMore/style.css

```
.FetchMore {
  display: flex;
  flex-direction: column;
  align-items: center;
}

.FetchMore-button {
  margin: 20px 0;
}
```

src/Input/style.css

```
.Input {
  border: none;
  padding: 10px;
  background: none;
  outline: none;
}

.Input:focus {
  outline: none;
}

.Input_white {
```

```
  border-bottom: 1px solid #fff;
  color: #fff;
}

.Input_black {
  border-bottom: 1px solid #000;
  color: #000;
}
```

src/Issue/IssueItem/style.css

```
.IssueItem {
  margin-bottom: 10px;
  display: flex;
  align-items: baseline;
}

.IssueItem-content {
  margin-left: 10px;
  padding-left: 10px;
  border-left: 1px solid #000;
}
```

src/Issue/IssueList/style.css

```
.Issues {
  display: flex;
  flex-direction: column;
  align-items: center;
  margin: 0 20px;
}

.Issues-content {
  margin-top: 20px;
  display: flex;
  flex-direction: column;
}

.IssueList {
  margin: 20px 0;
}

@media only screen and (max-device-width: 480px) {
  .Issues-content {
    align-items: center;
  }
}
```

src/Loading/style.css

```
.LoadingIndicator {
  display: flex;
  flex-direction: column;
  align-items: center;
  margin: 20px 0;
}

.LoadingIndicator_center {
  margin-top: 30%;
}
```

src/Repository/style.css

```
.RepositoryItem {
  padding: 20px;
  border-bottom: 1px solid #000;
}

.RepositoryItem-title {
  display: flex;
  justify-content: space-between;
  align-items: baseline;
}

@media only screen and (max-device-width: 480px) {
  .RepositoryItem-title {
    flex-direction: column;
    align-items: center;
  }
}

.RepositoryItem-title-action {
  margin-left: 10px;
}

.RepositoryItem-description {
  margin: 10px 0;
  display: flex;
  justify-content: space-between;
}

@media only screen and (max-device-width: 480px) {
  .RepositoryItem-description {
    flex-direction: column;
    align-items: center;
  }
}

.RepositoryItem-description-info {
  margin-right: 20px;
}
```

```
@media only screen and (max-device-width: 480px) {
  .RepositoryItem-description-info {
    text-align: center;
    margin: 20px 0;
  }
}

.RepositoryItem-description-details {
  text-align: right;
  white-space: nowrap;
}

@media only screen and (max-device-width: 480px) {
  .RepositoryItem-description-details {
    text-align: center;
  }
}
```

Show Comments

KEEP READING ABOUT REACT.

You can find the final repository on GitHub that showcases most of the exercise tasks. The application is not feature-complete and it doesn't cover all edge cases, but it should give insight into using GraphQL with Apollo in React applications. If you want to dive more deeply into different topics like testing and state management with GraphQL on the client-side, you can start here: A Minimal Apollo Client in React Example. Try to apply what you've learned in this application (e.g. testing, state management). Otherwise, I encourage you to try to build your own GraphQL client library, which helps you understand more of the GraphQL internals: How to build a GraphQL client library for React. Whichever you decide, keep tinkering on this application, or start with another GraphQL client application to fortify your skill set. You have finished all the GraphQL client chapters now.

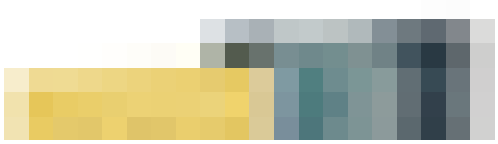
WRITING TESTS FOR APOLLO CLIENT IN REACT

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 I

A COMPLETE REACT WITH GRAPHQL TUTORIAL

In this client-sided GraphQL application we'll build together, you will learn how to combine React with GraphQL. There is no clever library like Apollo Client or Relay to help you get started yet...

Continue Reading: GraphQL Server Tutorial with Apollo Server and Express



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