A minimal Apollo Client in React Example

JUNE 05, 2018 BY ROBIN WIERUCH - EDIT THIS POST

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

This tutorial is part 3 of 3 in this series.

Part 1: A complete React with GraphQL Tutorial
Part 2: React with Apollo and GraphQL Tutorial

It's time to get you started with a minimal Apollo Client in React application that can be used as boilerplate project. There will be sections later on where you can use this application as starter project, but also you may want to experiment with it on your own. After all, it gives you all the necessary parts to consume GitHub's GraphQL API in your React application by using Apollo Client in a minimal

starter project. In addition though, there will be some local state management with React only to show you that local state management for local data is still used when having Apollo Client for your remote data.

APOLLO CLIENT IN REACT STARTER PROJECT

In the following case study application, you will consume GitHub's GraphQL API to query a bunch of repositories from an organization. You have learned those steps before. Basically it is how your remote data is managed in Apollo Client's Cache. However, this time you will introduce local data along the way. Imagine a use case where you would have to select the queried repositories in a list to make a batch operation (e.g. mutation) on them. For instance, you maybe want to star 3 of the 10 repositories. Therefore, you would have to introduce local data to track the selected repositories which are managed in a local state. In the following you will implement this use case, first by using React's local state but then transition to Apollo Link State as alternative.

It is up to you to create a React application with create-react-app. Afterward, you will have to setup Apollo Client in your React application as you have done in previous applications in the *src/index.js* file.

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

import App from './App';

import registerServiceWorker from './registerServiceWorker';

const cache = new InMemoryCache();

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

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

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

```
});

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

registerServiceWorker();
```

Don't forget to install the necessary packages for GraphQL, Apollo Client and React Apollo on the command line:

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

And furthermore, don't forget to add your personal access token from GitHub as value to the key in the *.env* file which you have to create in your project folder.

In the next step, implement the components to display the remote data which gets queried with React Apollo's Query component eventually.

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

import './App.css';

const GET_REPOSITORIES_OF_ORGANIZATION = gql`
  {
    organization(login: "the-road-to-learn-react") {
      repositories(first: 20) {
        edges {
          node {
            id
            name
            url
            viewerHasStarred
          }
        }
      }
    }
  }
`;

const App = () => (
  <Query query={GET_REPOSITORIES_OF_ORGANIZATION}>
    {({ data: { organization }, loading }) => {
      if (loading || !organization) {
```

```
        return <div>Loading ...</div>;
      }

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

const RepositoryList = ({ repositories }) => (
  <ul>
    {repositories.edges.map(({ node }) => {
      return (
        <li key={node.id}>
          <a href={node.url}>{node.name}</a>
        </li>
      );
    })}
  </ul>
);

export default App;
```

Once you run this application, you should see initially a loading indicator and afterward the list of repositories fetched from the defined GitHub organization in your GraphQL query. In addition, it could be possible to star a repository by executing a GraphQL mutation with the Mutation component.

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

...

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

...

const RepositoryList = ({ repositories }) => (
  <ul>
    {repositories.edges.map(({ node }) => {
      return (
        <li key={node.id}>
          <a href={node.url}>{node.name}</a>{' '}
```

```
          {!node.viewerHasStarred && <Star id={node.id} />}
        </li>
      );
    })}
  </ul>
);

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

export default App;
```

Nevertheless, there are a couple of potential improvements that you can do as exercise already before continuing with the tutorial. For instance, there is only a star mutation but not a unstar mutation when the repository is already starred. Another thing could be a search field to pass in a dynamic login of an organization to be flexible in querying repositories from different organizations. All of these improvements are up to your implementation to internalize the learnings from previous applications which you have built while learning about GraphQL in React.

Exercises:

- Implement the unstar mutation
- Implement a flexible way to query repositories from different organizations
  - Implement a search field that is managed with React's local state
  - When submitting the value from the search field, pass this value as variable to the Query component to use it as dynamic value for the `login` argument

---

STARTING WITH REACT'S LOCAL STATE MANAGEMENT FOR LOCAL DATA

Another requirement for this application was it to be able to select (and unselect) repositories in the list of repositories for performing batch operations. Such a batch operation could be to star (and unstar) selected repositories. Before being able to execute such an operation, it must be possible to select the repositories from the list in the first place. Therefore, React's local state management is the most straight forward choice for this problem to keep track of selected repositories. Each rendered repository row will have a button next to it. When clicking the button, the repository's identifier will be stored in React's local state. When clicking it again, the identifier will be removed again.

In order to keep the components lightweight and suited to their responsibilities (e.g. fetching data, rendering data), you can introduce a Repositories component which is used as container component in between of the App component and the RepositoryList component to manage local state.

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

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

The Repositories component in between manages the state of selected repositories by storing their identifiers in React's local state. in the end, it renders the RepositoryList component which was rendered previously in the App component. After all, you only introduced a component in between which has the responsibility to manage local state (container component) while the RepositoryList component only needs to render data (presentational component).

```
class Repositories extends React.Component {
  state = {
    selectedRepositoryIds: [],
  };

  toggleSelectRepository = (id, isSelected) => {
    ...
  };

  render() {
    return (
      <RepositoryList
        repositories={this.props.repositories}
        selectedRepositoryIds={this.state.selectedRepositoryIds}
        toggleSelectRepository={this.toggleSelectRepository}
      />
    );
  }
}
```

Now implement the business logic of the class method in Repositories component which adds and removes (toggles) the identifier of a repository depending on its incoming selection state.

```
class Repositories extends React.Component {
  state = {
    selectedRepositoryIds: [],
  };

  toggleSelectRepository = (id, isSelected) => {
    let { selectedRepositoryIds } = this.state;

    selectedRepositoryIds = isSelected
      ? selectedRepositoryIds.filter(itemId => itemId !== id)
      : selectedRepositoryIds.concat(id);

    this.setState({ selectedRepositoryIds });
  };

  render() {
    ...
  }
}
```

Since the list of selected repository identifiers and the class method to actually toggle a repository are passed to the RepositoryList component, you can implement a new Select component there to make use of those props.

```
const RepositoryList = ({
  repositories,
  selectedRepositoryIds,
  toggleSelectRepository,
}) => (
  <ul>
    {repositories.edges.map(({ node }) => {
      const isSelected = selectedRepositoryIds.includes(node.id);

      return (
        <li key={node.id}>
          <Select
            id={node.id}
            isSelected={isSelected}
            toggleSelectRepository={toggleSelectRepository}
          />{' '}
          <a href={node.url}>{node.name}</a>{' '}
          {!node.viewerHasStarred && <Star id={node.id} />}
        </li>
      );
    })}
  </ul>
);
```

The Select component is only a button which acts as toggle to select and unselect a repository.

```
const Select = ({ id, isSelected, toggleSelectRepository }) => (
  <button
    type="button"
    onClick={() => toggleSelectRepository(id, isSelected)}
  >
    {isSelected ? 'Unselect' : 'Select'}
  </button>
);
```

The select interaction should work after starting your application. It is indicated by a toggling "Select" and "Unselect" label after clicking the new button multiple times. But you can do better by adding some conditional styling to each row in the RepositoryList component.

```
const RepositoryList = ({ ... }) => (
  <ul>
    {repositories.edges.map(({ node }) => {
      const isSelected = selectedRepositoryIds.includes(node.id);

      const rowClassName = ['row'];

      if (isSelected) {
        rowClassName.push('row_selected');
      }

      return (
        <li className={rowClassName.join(' ')} key={node.id}>
          ...
        </li>
      );
    })}
  </ul>
);
```

Last but not least, you have to define the CSS classed which were used for the repository row in the *src/App.css* file:

```
.row {
  padding: 5px;
}

.row:hover {
  background-color: lightblue;
}

.row_selected {
  background-color: orange;
}

.row_selected:hover {
  background-color: orange;
```

```
}
```

That's it for the selection feature implementation. You should be able to select and unselect repositories in your list when starting your application now.

. . .

Remember that this solution with React's local state would already be sufficient to deal with this problem. No one else than the one component is interested in the selected repositories. So the state is co-located to the component. But following applications will show you how to replace React's local state management with Apollo Link State or Redux which is used side by side with Apollo Client. The minimal boilerplate application can be found in this GitHub repository as boilerplate project.

This tutorial is part 1 of 3 in this series.

Part 2: Mocking a GraphQL Server for Apollo Client
Part 3: Writing Tests for Apollo Client in React

This tutorial is part 1 of 3 in this series.

Part 2: A apollo-link-state Tutorial for Local State in React
Part 3: How to use Redux with Apollo Client and GraphQL in React

KEEP READING ABOUT GRAPHQL

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...

REACT WITH APOLLO AND GRAPHQL TUTORIAL

In this tutorial, you will learn how to combine React with GraphQL in your application using Apollo. The Apollo toolset can be used to create a GraphQL client, GraphQL

server, and other complementary...

✔ Join 50.000+ Developers

✔ Learn Web Development

✔ Learn JavaScript

✔ Access Tutorials, eBooks and Courses

✔ Personal Development as a Software Engineer

SUBSCRIBE >

View our Privacy Policy.

PORTFOLIO

Online Courses

Open Source

Tutorials

ABOUT

About me

What I use

How to work with me

How to support me

Contact Me Privacy & Terms