How to use Redux with Apollo Client and GraphQL in React

JUNE 26, 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 minimal Apollo Client in React Application
Part 2: A apollo-link-state Tutorial for Local State Management

In a previous application, you have used Apollo Link State to substitute React's local state management with it. Even though it wasn't necessary, because it is most often sufficient to manage local state with React only, it has shown you how it can be achieved with local queries and local mutations when using Apollo Link State. However, it needs to be clear that this only happens when you have the following three requirements for your application:

- you need a sophisticated state management layer (e.g. mobx, redux, apollo-link-state)
- you have a GraphQL server, because otherwise Apollo Client may makes no sense in the first place
- you embrace React's local state as still being useful for co-located component state

Basically you have learned that Apollo Link State can be used as replacement for Redux or MobX to manage local data in a global store in your application. However, Redux is often already used in React applications for historical reasons, because it was the successor of the state management libraries for React applications, but also because people consciously decide to use it. It's not without reason that people use Redux as their powerful state management solution. So there is no real use for Apollo Link State in these applications.

So even though the server-side application makes a transition to being GraphQL powered instead of being RESTful, it is not often requested to abandon Redux as state layer for the client-side application. The big question, which is asked by many developers who introduced GraphQL to their server-side and Apollo Client to their client-side application, is: **How to use Redux and Apollo Client together?** The application that you are going to build in the following section should show you only one approach of how Redux can be used together with Apollo Client in a React application. It gives you a implementation-wise scenario on how it can be done and is also followed by a couple of recommendations on how to use Redux and Apollo Client together in larger applications.

I also want to point out that Redux may not be needed at all when introducing Apollo Client to your tech stack. Whereas Apollo Client would be used for your remote data, React's local state may be sufficient to manage local data. Only if the state management for your local data becomes complex, you want to introduce a sophisticated state management solution such as Redux.

If you are not interested in Redux at all, it may be still valuable to read up the following approach and discussion for using other state management libraries in combination with Apollo Client. In general, if you are interested in using Redux with Apollo Client, but you have never learned Redux before, you should take the time to go through the The Road to Redux course to get a high level yet pragmatic understanding about it.

REDUX FOR LOCAL DATA, APOLLO CLIENT FOR REMOTE DATA

The requirement for this application is identical to the requirement for the previous application where you have introduced Apollo Link State as replacement for React's local state. In this application, you want to introduce Redux instead of Apollo Link State to manage the list of repository identifiers to power the selection feature.

In order to get you started with the project, you can either clone this starter repository from GitHub and follow its installation instructions or follow the steps from the previous application where you have built a minimal Apollo Client with React application.

Before you start to replace React's local state with Redux, take your time to try out the example application. You should be familiar with which part of the application is responsible for the local data and which part is responsible for the remote data. Afterward, you can introduce Redux by installing it on the command line:

```
npm install redux --save
```

The Redux store creation can happen in the same file as the Apollo Client creation for the sake of keeping everything at one place: the *src/index.js* file. Keep in mind that you may want to separate those things, setting up Apollo Client and Redux, in their own files when building a larger application with these libraries.

Before you can create a Redux store instance, you need to have at least one reducer. Reducers are used to take an action and the current state from the Redux store, and use the action payload to derive the next state for the Redux store: `(state, action) => nextState`. In addition, an action comes with a mandatory `type` so that it can be identified by the reducer(s) to do something with it. Moreover, an action can have an optional payload. In your *src/index.js* file, next to the Apollo Client creation, you can define your reducer to manage the state of selected repositories.

```
function repositoryReducer(state, action) {
  switch (action.type) {
    case 'TOGGLE_SELECT_REPOSITORY': {
      return applyToggleSelectRepository(state, action);
    }
    default:
      return state;
  }
}

function applyToggleSelectRepository(state, action) {
  const { id, isSelected } = action;

  const selectedRepositoryIds = isSelected
    ? state.selectedRepositoryIds.filter(itemId => itemId !== id)
```

```
      : state.selectedRepositoryIds.concat(id);

    return { ...state, selectedRepositoryIds };
  }
```

Basically the Redux reducer does the same as the React local state, but only for this particular action which is identified by its type. It only merges the local data in a global state which is managed by the Redux store rather than React's local state managed co-located in a component.

Next, you can setup the Redux store in the same file and give it the reducer and an initial state which has the identical structure as the local state in the Repositories component.

```
import { createStore } from 'redux';

...

const initialState = {
  selectedRepositoryIds: [],
};

const store = createStore(repositoryReducer, initialState);
```

Last but not least, the Redux store needs to be provided along with Apollo Client instance to the React view-layer. Therefore, you need to install the connecting react-redux package for it on the command line:

```
npm install react-redux --save
```

Afterward, you can import the Provider component and use it next to the ApolloProvider to provide the Redux store to the view-layer as its context. Afterward it can be used implicitly by React components the same as the Apollo Client instance.

```
import { Provider } from 'react-redux';
import { createStore } from 'redux';

...

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

Now the Redux state-layer is paired with the React view-layer just as the Apollo Client state-layer was paired before. In the next steps, React's local state management for the selection feature needs to be replaced in the *src/App.js* file. You can remove the Repositories component, because you will create a connected Component by using the `connect()` higher-order component from the React Redux package. This HOC makes it possible to map state from the Redux store to your React component as props.

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

...

const App = () => (
  <Query query={GET_REPOSITORIES_OF_ORGANIZATION}>
    {({ data: { organization }, loading }) => {
      ...

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

const RepositoryList = ({ repositories, selectedRepositoryIds }) => (
  <ul>
    ...
  </ul>
);

const mapStateToProps = state => ({
  selectedRepositoryIds: state.selectedRepositoryIds,
});

const Repositories = connect(mapStateToProps)(RepositoryList);

...
```

The previous part was for reading data from the Redux store in a React component by using the `connect()` higher-order component and mapping the state from the store to the component. The next part will implement the writing data to the Redux store from a React component by using the `connect()` higher-order component again, but this time for dispatching an action.

First, you can remove the `toggleSelectRepository()` callback function from the RepositoryList component, but still keep it in the Select component for now.

```
const RepositoryList = ({ repositories, selectedRepositoryIds }) => (
  <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}>
          <SelectContainer id={node.id} isSelected={isSelected} />{' '}
          <a href={node.url}>{node.name}</a>{' '}
          {!node.viewerHasStarred && <Star id={node.id} />}
        </li>
      );
    })}
  </ul>
);
```

The callback function is not passed anymore from above as prop, but passed as prop from the `connect()` higher-order component which is this time used for the Select component.

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

const mapDispatchToProps = (dispatch, { id, isSelected }) => ({
  toggleSelectRepository: () =>
    dispatch({
      type: 'TOGGLE_SELECT_REPOSITORY',
      id,
      isSelected,
    }),
});

const SelectContainer = connect(null, mapDispatchToProps)(Select);
```

Now the Select component is able to dispatch the action for the Redux store instead of using the React's local state. Once you start your application again, it should work as before, but this time with Redux instead of React's local state. As for now, Redux is responsible to manage the local data and Apollo Client is responsible to manage the remote data.

INTERACTION BETWEEN REDUX AND APOLLO CLIENT

In this section, I want to give a recommendation on how to use both state management layers, Apollo Client and Redux, together. However, you should take these as my personal opinion and not as best practice or whatsoever, because there are not many people out there which went down the road to use these state layers together in a large application. It may be too early to give any useful recommendation here. However, I want to give it a shot to giving advice for using both state layers in a growing React application. Therefore I want to introduce three different levels of togetherness for using both state layers. With every level of togetherness, the degree of separation of both state layers decreases and thus it is easier to use both together as a unit. However, easier doesn't mean simpler. So by interweaving both state layers to a certain level, it will increase the complexity of dealing with both state layers. In conclusion, it comes with the cost of complexity to have a higher level of togetherness. A rule of thumb could be to keep the level of togetherness as low as possible.

Togetherness Level 1: Apollo and Redux

In the previous application, you have seen how both state management tools can exist side by side to manage local data and remote data. When using Apollo Client instead of Redux for remote data, you can 100% rely on its caching, normalization and request states. If you would use Redux for the remote data, by using an asynchronous action library such as Redux Thunk or Redux Saga, you would have to implement the caching and normalization (e.g. using normalizr) features yourself. Furthermore, all the request states, such as loading state, error state and success state, are handled by Apollo Client for you. In Redux you would have to introduce a couple of actions to deal with those request states:

- GET_TODOS_LOADING
- GET_TODOS_ERROR
- GET_TODOS_SUCCESS

Apollo Client takes away all this pain when dealing with remote data. It boils down to the mentioned requirements: If you have a GraphQL backend, you can introduce Apollo Client as GraphQL client library to deal with the remote data. If your state management for local data grows eventually, and you cannot deal with a larger part of it with React's local state only, you can introduce Redux for it. In order to meet all these requirements, your application should be fairly mature to justify having two state layers side by side. Otherwise, the common sense for your application should be to keep it simple with only one state layer (Redux or Apollo Client along with React's local state).

```
+----------------------+ +----------------------+
|        Redux         | |    Apollo Client     |
| +------------------+ | | +------------------+ |
| |                  | | | |                  | |
| |    Local Data    | | | |   Remote Data    | |
| |                  | | | |                  | |
| +------------------+ | | +------------------+ |
|                      | |                      |
+----------------------+ +----------------------+
```

The tenor from the previous paragraph was mostly to take Redux and Apollo Client for separated areas of state management: local data and remote data. For instance, while a list of repositories coming from the remote GitHub GraphQL API is stored as normalized entities in Apollo Client's Cache, a selection of repositories which is purely based on user interaction is stored in the Redux store. If you keep up those clear constraints and boundaries for the two state layers, you might be alright managing two sources of truth instead of a single source of truth for the state in your React application. In the end, the most important rule should be to either store state in Apollo Client's Cache or the Redux store. There shouldn't be any duplications.

Togetherness Level 2: Apollo with Redux, Redux with Apollo

However, at some point it may go beyond the basic example which was shown by implementing the last application. In the example, both state layers were clearly separated. But then comes the time where you have to implement a feature that needs to access state from Redux for a Apollo Client query/mutation or needs to access state from Apollo Client for a Redux mapStateToProps/mapDispatchToProps. In order to give a recommendation for it, you can see those as analogies for reading and writing state:

- read state:
  - local data: Redux mapStateToProps
  - remote data: Apollo Client query (either from cache or network request)
- write state:
  - local data: Redux mapDispatchToProps
  - remote data: Apollo Client mutation

The key is to compose both state layers into each other when they have to interact. For instance, imagine you would want to perform a batch mutation with Apollo Client to star selected repositories that are stored in the Redux store. You would compose the Redux layer around the Apollo Client layer to pass the selected repositories from the React Redux mapStateToProps to your React component and ultimately to your React Apollo Mutation component to be used there as variables. The same would apply for using React Apollo Query component (e.g. fetching issues of all the selected repositories).

```
+----------------------+
|         Redux        |
| +------------------+ |
| |                  | |
| |   Apollo Client  | |
| |                  | |
| +------------------+ |
|                      |
+----------------------+
```

On the other side, if you need to pass state from the Apollo Client layer to your Redux layer, because you need the state for a Redux action, you can compose the Apollo Client layer around the Redux layer to access the state from a Query (or Mutation) component's result from the child function. The result could be passed down as props to a connected Redux component which would then have access to the result in mapDispatchToProps (or mapStateToProps) as props.

```
+----------------------+
|     Apollo Client    |
| +------------------+ |
| |                  | |
| |       Redux      | |
| |                  | |
| +------------------+ |
|                      |
+----------------------+
```

In conclusion, following these constraints, you can always pass state from one to another state layer to either derive state (query, mapStateToProps) from the particular state or to alter state (mutation, mapDispatchToProps).

Togetherness Level 3: Apollo in Redux, Redux in Apollo

So far, I introduced two levels of using Apollo Client and Redux together. The first level was a clear separation between both layers with almost no interaction. The remote data wasn't much used for the local data and vice versa. The second level is enabling an interaction by composing both state layers into each other. Then the remote and local data are used with another to derive local/remote data or to alter local/remote data of the other state layer. Now there could be also a third level which enables a state layer (Apollo Client or Redux) to be used within the control flow of the other state layer. I wouldn't recommend to go this far, but I have seen projects where it was implemented this way.

For instance, imagine you reached a point where you have introduced Redux Saga for a fine-grained control flow for your business logic in your React and Redux powered application. By using the second level of togetherness, you would be only able to pass (remote) data from Apollo Client to the Redux Sagas. But what if you would need to be able to make a GraphQL query/mutation from within this control flow of Redux Saga? That's the point where people start to pass the Apollo Client instance (e.g. by using the ApolloConsumer component) to Redux Saga in order to execute imperative mutations or queries. This could be the case for prefetching data (query) or altering remote data (mutate) from within the Redux Saga control flow.

However, by doing so the boundary between local data and remote data or Redux and Apollo Client becomes blurry. It adds another level of complexity, because someone has to untangle the code afterward. As mentioned before, going down the road by increasing the level of togetherness gives you more flexibility in using both state layers within each other, but adds lots of complexity which you might be able to avoid in the first place.

The last paragraphs tried to give you advice on how to use React and Redux together to a certain degree. However, you need to note that it isn't common to reach this certain degree for many application out there. If you start out with an application, it doesn't happen too often that you have a GraphQL server to be consumed by Apollo Client in your React application. If it is the case, your client-side application needs to reach a certain point of unease of managing your local state with React itself. Not until then you would introduce Redux (or another sophisticated state management solution). Afterward, you have to decide whether you apply the first or second level of togetherness in your application. Then again there needs to be the need to introduce a library such as Redux Saga, because all of your asynchronous operations for the remote data are done with Apollo Client anyway. Only then, when introducing such a side-effect library for Redux for having an improved control flow, you can consider to increase the level of togetherness for the state layers to the third level. Last but least, I want to mention again that this area is very new to many people, so don't take all of this advice as best practice or anything. It is just my opinion for this topic.

. . .

The last sections have shown you how to set up and use Apollo Client with Redux in a React application. The example application was only one approach of doing it. Furthermore, another section gave you advice on how to fit Redux and Apollo Client together in larger applications. Since there is not much experience in this area yet, everyone is waiting for people gathering more experience about this topic. Another neat way to set up Redux when having Apollo Client is using the apollo-redux-cache (example) package instead of the apollo-cache-inmemory package to create Apollo Client's Cache. When using the Redux Cache instead, you get all the functionality around Redux and associated packages, such as redux-persist, for free. However, it must be a conscious decision to exchange the Apollo Client Cache for it.

Continue Reading: Mocking a GraphQL Server for Apollo Client

Continue Reading: How to build a GraphQL client library for React

Show Comments

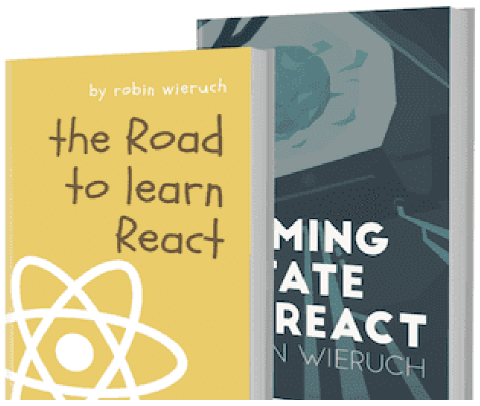
KEEP READING ABOUT REACT

MOBX REACT: REFACTOR YOUR APPLICATION FROM REDUX TO MOBX

MobX is a state management solution. It is a standalone pure technical solution without being opinionated about the architectural state management app design. The 4 pillars State, Actions, Reactions...

A APOLLO-LINK-STATE TUTORIAL FOR LOCAL STATE IN REACT

There are many people out there questioning how to deal with local data in a React application when using Apollo Client for remote data with its queries and mutations. As shown in previous...



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.