A apollo-link-state Tutorial for Local State in React

MAY 13, 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 2 of 3 in this series.

Part 1: A minimal Apollo Client in React Application
Part 3: How to use Redux with Apollo Client and GraphQL 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 applications, it can be done with React's local state management. When the state management for the local data reaches a point where it becomes too complex, it may be beneficial to introduce a state management library such as Redux or MobX. However, this leaves the issue of not having a single source of truth as state. There are two state layer then. This topic is revisited later in another application, where I will point out how to use Redux side by side with Apollo Client.

In this section though, I want to show how to use apollo-link-state instead of introducing Redux or MobX in a React example application. When using Apollo Link State, the Apollo Client Cache becomes your single source of truth for state. It manages remote data and local data then. It is important to note that Apollo Link State makes only sense when having a GraphQL backend which is consumed by Apollo Client in the frontend. Only then the Apollo Link State add-on can be used as state management solution for the local data.

However, it is still important to remember that React's local state is often sufficient when dealing with co-located state, which is not reaching outside to the general application but is close to its components, even though there is a sophisticated state management layer in place. You apply the same rules as for introducing Redux or MobX: React's local state stays important even with sophisticated state management solutions such as Redux, MobX or Apollo Link State. Not everything belongs in the global state which is established by these libraries.

TABLE OF CONTENTS

SETUP OF APOLLO LINK STATE: RESOLVERS AND DEFAULTS

Before you can start to set up Apollo Link State in your React with Apollo Client application, you have to create a boilerplate project or use an existing project that is out there. In a previous section, you have built a minimal Apollo Client with React application which you will use as your starter project now. You can find it in this GitHub repository for cloning it.

In this boilerplate application, you have managed a list of identifiers that represent selected repositories in the local state of a React component. In order to manage the list of identifiers in Apollo Client's Cache instead of React's local state, you have to install Apollo Link State on the command line first. In addition, you have to install the Apollo Link package for combining multiple links for your Apollo Client creation.

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

Now comes to the Apollo Link State setup, but as you have learned before, Apollo Links can be composed in a straight forward way by using the Apollo Link package. You have already set up the Apollo HTTP Link. Now it is accompanied by the Apollo Link State in the composition. The composed link can then be used by the Apollo Client creation.

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

...

const stateLink = withClientState({
  cache,
  defaults: {},
  resolvers: {},
});

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

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

...
```

It is important to have the `stateLink` not far away from the end of your link chain, but it shouldn't be the last terminating link (in this case the `httpLink`) which makes the network requests. If you would add the Apollo Link Error package, the `stateLink` should come after it, so that the `errorLink` can deal with the errors on behalf of it.

The `withClientState()` is a new function from Apollo Link State to create an instance of the link. It takes an object with a Apollo Client Cache instance, defaults and resolvers. You already have the cache, so what's left to explain are the defaults and resolvers.

The **Apollo Link State Defaults** are used to define an initial state for the Apollo Client's Cache. After all, the Apollo Link State is used to manage a state object. So it is mandatory to have an initial state for it. Otherwise your later introduced local queries and mutations may hit an undefined local state which would lead to an error.

The **Apollo Link State Resolvers** is a map of mutation and query functions for implementing the logic of these **local GraphQL operations**. If you have implemented a GraphQL server before, you may be aware of these resolver functions already. For instance, in case of a query, they may be used to retrieve the remote data from a database. On the other side, in case of a mutation, they are used to alter the remote data in the database. That's for the server-side though. On the client-side, you can define those resolver functions when using Apollo Link State too. But they are used to retrieve local data from the local state or to alter local data in the local state. The normal case is that there is no remote data involved at all.

In your setup, the `resolvers` and `defaults` are empty objects. You will use both of them in the next sections to read and write local data to your local state with Apollo Link State instead of using React's local state for it. This way, you will have a better understanding of what they are really used for.

APOLLO LINK STATE FOR LOCAL STATE: READ DATA

So how to read and write data to the local state? You will do it with GraphQL queries and mutations, but this time they will be used for local data instead of remote data. That's why you have set up Apollo Link State to manage this local data for you instead of React's local state.

First, let's give the `defaults` a filled initial state object when creating the Apollo Link State instance for the sake of querying it afterward. In your application, you have queried a list of repositories before. Since the local state in Apollo Link State should store a list of repository identifiers to keep track of selected repositories, you can insert one of the queried repository identifiers in it. This repository should be pre-selected then when starting the application. In the case of the `the-road-to-learn-react` organization which you may have queried before, one of the queried repositories has the id `MDEwOlJlcG9zaXRvcnk2MzM1MjkwNw==`. So you can go with this one in your initial state object and use the object for the defaults configuration.

```
...

const initialState = {
  selectedRepositoryIds: ['MDEwOlJlcG9zaXRvcnk2MzM1MjkwNw=='],
};

const stateLink = withClientState({
  cache,
  defaults: initialState,
  resolvers: {},
});

...
```

Second, since this state is defined in the local state of Apollo Link State, you can remove React's local state for the list of repository identifiers from the Repository component. In addition, you can remove the handler which toggles the repository selection. After removing all this implementations, the Repositories component becomes a functional stateless component again.

```
const Repositories = ({ repositories }) => (
  <RepositoryList
    repositories={repositories}
    selectedRepositoryIds={selectedRepositoryIds}
  />
);
```

So where does the list of selected repository identifiers come from? Since they are in Apollo Client's Cache due to Apollo Link State and not in React's local state anymore, you can query them with a normal GraphQL query and the Query component which you have used before for querying remote data. This time they are used for querying local data though.

```
const GET_SELECTED_REPOSITORIES = gql`
  query {
    selectedRepositoryIds @client
  }
`;

...

const Repositories = ({ repositories }) => (
  <Query query={GET_SELECTED_REPOSITORIES}>
    {({ data: { selectedRepositoryIds } }) => (
      <RepositoryList
        repositories={repositories}
        selectedRepositoryIds={selectedRepositoryIds}
      />
    )}
  </Query>
);
```

The query for local data works almost identical to the query for remote data. There is only one difference: the @client directive. As you may have learned before, there exist directives such as the @skip or @include directive in the GraphQL specification. Apollo came up with an own directive to annotate objects and fields in a query (or mutation) as local data. Basically the @client directive tells Apollo Client Cache to look up the data in the local state instead of doing a network request.

Just for the sake of demonstration, the @client directive affects all underlying fields too. So when annotating a field which fields itself, all the underlying fields are derived from the local state too.

```
const GET_SELECTED_REPOSITORIES = gql`
  query {
    repositoryInformation @client {
      selectedRepositoryIds
    }
  }
`;
```

Since a GraphQL operation is fine-tuned of a field level, the @client directive can be used for only a part of the data. All the remaining fields are fetched by using a network request with the Apollo HTTP Link. The following query gives you an example of how one query can be used to fetch local data and remote data.

```
const GET_SELECTED_REPOSITORIES = gql`
  query {
    repositoryInformation @client {
      selectedRepositoryIds
    }
    organization {
      name
      url
    }
  }
`;
```

Nevertheless, let's stick to the initial implementation of the query for not adding too much noise and keeping the example simple.

```
const GET_SELECTED_REPOSITORIES = gql`
  query {
    selectedRepositoryIds @client
  }
`;
```

When you start your application again, you should see that one of the repositories is selected, because you have defined the identifier in the `defaults` of the Apollo Link State initialization.

Basically it is pre-selected because of the initial state. It is similar to telling React's local state to have an initial state:

```
class SomeComponent extends Component {
  state = {
    selectedRepositoryIds: ['MDEwOlJlcG9zaXRvcnk2MzM1MjkwNw=='],
  }

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

The only difference is that Apollo Link State manages a global state and React' local state only a component co-located state. After all, by using Apollo Link State, you have made your state globally accessible by using GraphQL queries.

The reading local data part of the equation works. What about the writing local data part then? The Select component is broken as it is now, because in the last implementation the `toggleSelectRepository()` callback function was removed since it cannot be used anymore to update the identifiers in React's local state. The state lives in Apollo Client Cache now.

APOLLO LINK STATE FOR LOCAL STATE: WRITE DATA

You have seen how a GraphQL query is used for reading local data from the local state in the previous section. That's half of what GraphQL operations are used for (for the savvy reader: this statement is wrong, because it is only true if you don't include GraphQL subscriptions). The second half is writing local data to the local state. Whereas you have used previously a GraphQL query to read local data, now you will use a GraphQL mutation to write local data to the local state. In the Select component, you can remove the `toggleSelectRepository()` callback function which was used before to update React's local state that was removed in a previous step. In addition, the function can be removed from the RepositoryList component too.

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

  ...

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

Since the state managed by Apollo Link State is global now, you don't need to pass any callback functions anymore. Instead, identical to the reading local state part with the GraphQL query and Query component, you can use the Mutation component from React Apollo and a GraphQL mutation to write data to the state.

```
const SELECT_REPOSITORY = gql`
  mutation($id: ID!, $isSelected: Boolean!) {
    toggleSelectRepository(id: $id, isSelected: $isSelected) @client
  }
`;

...

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

The GraphQL mutation for local data isn't any different from a mutation used for remote data except for the @client directive. The directive was used for the local query too in order to signalize Apollo Client that it is a local mutation, because there is no remote data involved here.

Almost identical to the `toggleSelectRepository()` callback function which was used before to update React's local state, the exposed mutation function `toggleSelectRepository()` gets implicit access to the `id` and `isSelected` values via the variables in the Mutation component.

The big question mark: How to define on the client-side what happens after executing this mutation? If you would send this mutation to your GraphQL server, a resolver would take care about it. So that's why you can define those resolvers for your local state on the client-side, only when using Apollo Link State, as well. You can define a resolver function for the actual `toggleSelectRepository` mutation which you have used in the previously defined GraphQL mutation.

```
const initialState = {
  selectedRepositoryIds: [],
};

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

const stateLink = withClientState({
  cache,
  defaults: initialState,
  resolvers: {
    Mutation: {
      toggleSelectRepository,
    },
  },
});

...
```

A client-side resolver function has the same signature as an usually on the server-side used resolver function. The arguments are `parent`, `args`, `context` and `info`.

In this example, the `info` argument isn't needed and thus doesn't appear in the function signature. The same applies for the `parent` argument, but it appears in the signature because it is the first argument of it. It can be named _ to keep it out of your sight, because it isn't used for the mutation in this example. You may wonder when you would need the `parent` argument. You may only need it when your query or mutation becomes deeply nested and fields in the particular operation need to be resolved with their own resolver functions which would naturally lead to multiple resolver functions. Then the `parent` argument can be used to pass results from one to another resolver function. In our case, you can ignore the `parent` argument because it is not needed. However, if you are curious,you can read more about it.

After all, you only need the `args` and `context` arguments. While the former has all the parameters which where provided as GraphQL arguments to the GraphQL mutation, the latter has access to the Apollo Client's Cache instance. Thus, both can be used to write the local data to the local state.

Before you can write data to the local state, you often need to read data from it in order to update it. In this case, you need to read the list of selected repositories from the local state in order to update it with the new selected or unselected identifier. Therefore, you can use the same query which you have used in your Repositories component. In order to use it in the *src/index.js* file for the resolver function, you have to export it from the *src/App.js* file first:

```
export const GET_SELECTED_REPOSITORIES = gql`
  query {
    selectedRepositoryIds @client
  }
`;
```

Afterward, you can import it in the *src/index.js* file for your resolver function:

```
import App, { GET_SELECTED_REPOSITORIES } from './App';
```

Finally, as first step, the query can be used in the resolver function to retrieved the list of selected repository identifiers. The cache instance offers methods such as `readQuery()` or `readFragment()` to read data from it. That's why you had to import the query.

```
const toggleSelectRepository = (_, { id, isSelected }, { cache }) => {
  let { selectedRepositoryIds } = cache.readQuery({
    query: GET_SELECTED_REPOSITORIES,
  });

  ...
};
```

In the second step, the list of selected repository identifiers can be updated with the provided information in the `args` argument.

```
const toggleSelectRepository = (_, { id, isSelected }, { cache }) => {
  let { selectedRepositoryIds } = cache.readQuery({
    query: GET_SELECTED_REPOSITORIES,
  });

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

  ...
};
```

Third, the updated data can be written with one of the `writeData()`, `writeQuery()` or `writeFragment()` methods that are available for the cache instance in order to write data. In this case, since the data was read with the `readQuery()` method, it makes most sense to write it again with the analogous method `writeQuery()` because then it matches the identical data structure requirements.

```
const toggleSelectRepository = (_, { id, isSelected }, { cache }) => {
  let { selectedRepositoryIds } = cache.readQuery({
    query: GET_SELECTED_REPOSITORIES,
  });

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

  cache.writeQuery({
    query: GET_SELECTED_REPOSITORIES,
    data: { selectedRepositoryIds },
  });

  ...
};
```

Last but not least, a mutation result should be returned. In this case, no result is needed in the previously used Mutation component, so it can be null.

```
const toggleSelectRepository = (_, { id, isSelected }, { cache }) => {
  let { selectedRepositoryIds } = cache.readQuery({
    query: GET_SELECTED_REPOSITORIES,
  });

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

  cache.writeQuery({
    query: GET_SELECTED_REPOSITORIES,
    data: { selectedRepositoryIds },
  });

  return null;
};
```

That's it for writing local data in Apollo's local state by using a GraphQL mutation which is only used locally. Once you start your application again, the select interaction should work. But this time, the data is stored and retrieved in/from Apollo Link State by using GraphQL operations instead of React's local state.

Last but not least, for the sake of mentioning it, when you would want to add a result to your mutation, you could start out by adding the wanted fields in the mutation definition:

```
const SELECT_REPOSITORY = gql`
  mutation($id: ID!, $isSelected: Boolean!) {
    toggleSelectRepository(id: $id, isSelected: $isSelected) @client {
      id
      isSelected
    }
  }
`;
```

Next, the resolver function can return the updated result:

```
const toggleSelectRepository = (_, { id, isSelected }, { cache }) => {
  let { selectedRepositoryIds } = cache.readQuery({
    query: GET_SELECTED_REPOSITORIES,
  });

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

  cache.writeQuery({
    query: GET_SELECTED_REPOSITORIES,
    data: { selectedRepositoryIds },
  });

  return { id, isSelected: !isSelected };
};
```

And finally you would be able to access it in the Mutation's render prop child function as second argument.

```
const Select = ({ id, isSelected }) => (
  <Mutation
```

```
      mutation={SELECT_REPOSITORY}
      variables={{ id, isSelected }}
    >
      {(toggleSelectRepository, result) => (
        <button type="button" onClick={toggleSelectRepository}>
          {isSelected ? 'Unselect' : 'Select'}
        </button>
      )}
    </Mutation>
  );
```

In the end, you should be able to access the result with the previous implementations. However, in the case of this mutation it is not really needed. In case you need it in the future, you have the necessary knowledge to do it. The application which you have implemented in the previous sections can be found here as GitHub repository.

Exercises:

- Implement select and unselect all repositories in the list mutations
- Implement a batch star and unstar mutation for all selected repositories

---

ASYNC QUERY RESOLVER AND DEVICE DATA

You will not go any deeper implementation wise for this application. Instead, this last paragraph should only give you an outline what's possible with Apollo Link State. Foremost, Apollo Link State is used for **local data** which is created in the client application by having user interactions while having Apollo Client itself for **remote data**. You have used both kinds of data in the previous application. But what about other data? For instance, there could be **device data** which can be queried from a mobile phone when using Apollo Client there. Then you can use Apollo Link State as well. You can define a **async query resolver** in your Apollo Link State resolvers when setting up Apollo Client with Apollo Link State. In this resolver you can define your query (as you have done with your mutation before) and its implementation: how it accessed the device API (with optionally given arguments) and how it returns the result. In case you are interested in this topics, you have to dig deeper into the Apollo Link State documentation. One could argue that every other side-effect can be done in Apollo Link State too. For instance, you can perform requests to other API endpoints in a query resolver too. However, you should be cautious when mixing different use cases into Apollo Link State. It's main focus is local data after all.

. . .

The last application has shown you how to use Apollo Link State instead of React's local state to establish state management in your application for local and remote data. As mentioned, the use case isn't the best choice to demonstrate Apollo Link State, because the local data which is manage as state is only needed in one component. It would be the best choice to keep this data in React's local state to have it co-located to your component. However, if you imagine a scenario where this local data is needed and thus shared across the whole application, it might be a valid step to manage it in Apollo Link State (instead of Redux or another state management solution). Then the Apollo Client's Cache becomes the single source of truth for remote data and local data.

This tutorial is part 2 of 3 in this series.

Part 1: A minimal Apollo Client in React Application

Part 3: How to use Redux with Apollo Show Comments in React

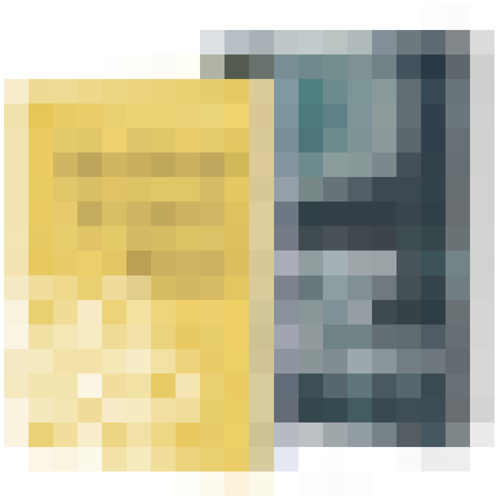
KEEP READING ABOUT REACT

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...

HOW TO USE REDUX WITH APOLLO CLIENT AND GRAPHQL IN REACT

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...



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