How to build a GraphQL client library for React

JUNE 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 3 of 3 in this series.

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

You may have used a GraphQL client library that was view-layer agnostic and thus able to work with React or other solutions like Angular or Vue. Other GraphQL client libraries like Relay and Urql aren't so powerful, because they are used only for React. The next chapter is to illustrate how

to implement a GraphQL client library that works with React. The takeaway shouldn't be "build your own GraphQL client for your production ready applications", however, you should instead learn:

- how a GraphQL client library works under the hood
- how simple a GraphQL client library can be implemented your way
- how it is capable of contributing to the GraphQL ecosystem

There is lots of room to contribute to the GraphQL ecosystem, because the tools surrounding it are still fairly new. A diverse set of tools would speed this along, instead of Apollo pushing its maturation forward alone. This is not only a useful addition for your web development skillset, it is also an opportunity to contribute to the early stages of GraphQL.

Before diving into implementing your own GraphQL client for React, consider the essentials for consuming a GraphQL API in a React application:

- A **GraphQL client** must be used. It can be any HTTP library or even the native fetch API, but it must be able to send HTTP methods with a payload across the wire. While the GraphQL specification isn't opinionated about the transportation layer, the GitHub GraphQL API you consume with a GraphQL client is using HTTP. Because we are using their API, our GraphQL client must be able to execute GraphQL operations using HTTP methods.
- There must be a way to **provide the GraphQL client instance to the React view layer**. It is the perfect use for React's Context API to provide the GraphQL client instance at the top level of the React component tree, and to consume it in every React component interested in it.
- There must be a way to **execute GraphQL operations, like a query or a mutation, in a declarative way in React**. You will implement a Query component and a Mutation component that exposes an API to execute the GraphQL operations and to access its result. Because you are implementing these components, you won't touch the GraphQL client provided with React's Context API explicitly in your React components, but only in the Query and Mutation components.

The first part is React agnostic, but the second and third glue the GraphQL client (data layer) to React (view layer). It can be seen as an analog to the *redux* and *react-redux* or *apollo-client* and *react-apollo* libraries. The former is view layer agnostic, the latter is used to connect it to the view layer.

While you implement a GraphQL client for React in the following sections, you will also implement a GitHub client application with React that consumes GitHub's GraphQL API, using GraphQL client.

IMPLEMENTING YOUR GRAPHQL CLIENT

Next, you will separate the domain specific application (GitHub client) and the GraphQL client with its connecting parts to the React world. The latter could be extracted later, as a standalone library, and published on npm. It could even be split up into two libraries, where the first part is the view layer agnostic GraphQL client, and the second is used to connect the former to the view layer..

First, bootstrap your React application with create-react-app where you will implement your GraphQL client and the connecting parts to the view layer.

Second, create a file to implement your standalone GraphQL client. You are going to use axios as HTTP client to send queries and mutations with HTTP POST methods.

```
npm install axios --save
```

The GraphQL client build with axios could be as lightweight as the following:

```
import axios from 'axios';

const graphQLClient = axios.create();

export default graphQLClient;
```

Since you may need greater control for creating the GraphQL client instance--passing in the GraphQL API endpoint or HTTP headers, for example--you can also expose it with a function that returns the configured GraphQL client instance.

```
import axios from 'axios';

const createGraphQLClient = (baseURL, headers) =>
  axios.create({
    baseURL,
    headers,
  });

export default createGraphQLClient;
```

Maybe you want to avoid using the GraphQL client with HTTP methods (e.g. `graphQLClient.post()`), or you may want to expose different functions for the query and mutation methods (e.g. `graphQLClient.query()`) called from the outside. That way, you never

see the behind the scenes HTTP POST when interacting with the GraphQL client. For this, JavaScript class makes sense.

```
import axios from 'axios';

class GraphQLClient {
  axios;

  constructor({ baseURL, headers }) {
    this.axios = axios.create({
      baseURL,
      headers,
    });
  }

  query({ query, variables }) {
    return this.axios.post('', {
      query,
      variables,
    });
  }

  mutate({ mutation, variables }) {
    return this.axios.post('', {
      query: mutation,
      variables,
    });
  }
}

export default GraphQLClient;
```

That's it for the GraphQL client. You created an instance of the GraphQL client and executed GraphQL operations (query and mutation) with it. You may wonder: Where is the state, the caching of requests, and the normalization of the data? You don't need them. The lightweight GraphQL client operates without any extra features, though I invite you to extend the feature set of the GraphQL client after you implement it in the following sections.

Next, use the instantiated GraphQL Client in your top level React component.

```
import React from 'react';
import ReactDOM from 'react-dom';

import App from './App';
import GraphQLClient from './my-graphql-client';
import registerServiceWorker from './registerServiceWorker';

const client = new GraphQLClient({
  baseURL: 'https://api.github.com/graphql',
```

```
  headers: {
    Authorization: `bearer ${
      process.env.REACT_APP_GITHUB_PERSONAL_ACCESS_TOKEN
    }`,
  },
});

ReactDOM.render(
  <App />,
  document.getElementById('root'),
);
registerServiceWorker();
```

Necessary information is provided for the GraphQL client constructor to create a GitHub GraphQL client instance. In previous applications, you have seen how to obtain the personal access token from GitHub to access their data and how to use it in a *.env* file for environment variables, to make it securely accessible for the GraphQL client instantiation.

IMPLEMENTING YOUR GRAPHQL TO REACT BRIDGE

In this section, you connect your GraphQL client instance to your React view layer, and the best way is to use React's Context API. In a separate file, you can create the necessary parts for creating the context used to tunnel the GraphQL client instance from a Provider component to all Consumer components.

```
import { createContext } from 'react';

const GraphQLClientContext = createContext();

export default GraphQLClientContext;
```

To provide the GraphQL client instance to your React component tree, use the previous context and its Provider component to make it available to the underlying React component hierarchy.

```
import React from 'react';
import ReactDOM from 'react-dom';

import App from './App';
import GraphQLClient from './my-graphql-client';
import GraphQLClientContext from './my-graphql-client-react/context';
import registerServiceWorker from './registerServiceWorker';

const client = new GraphQLClient({
```

```
  baseURL: 'https://api.github.com/graphql',
  headers: {
    Authorization: `bearer ${
      process.env.REACT_APP_GITHUB_PERSONAL_ACCESS_TOKEN
    }`,
  },
});

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

Since you provided the GraphQL client instance to your React component tree, you can use the Consumer component from the context object to retrieve the client as a value. You can implement a higher-order component (HOC) to make the GraphQL client instance accessible to React components using this HOC.

```
import React from 'react';
import GraphQLClientContext from '../context';

const withClient = Component => props => (
  <GraphQLClientContext.Consumer>
    {client => <Component {...props} client={client} />}
  </GraphQLClientContext.Consumer>
);

export default withClient;
```

Rather than using the Consumer component directly in your React components, use it implicitly with a higher-order component to expose the GraphQL client instance to the props. By now you have implemented all the parts necessary to bridge the data layer to the view layer. You have a Provider component providing the GraphQL client instance for the whole React component tree, and a higher-order component using the Consumer component to make the GraphQL client instance available to all React components interested in it.

IMPLEMENTING THE QUERY COMPONENT IN REACT

Now you are going to explore on how to execute GraphQL queries and mutations with your own GraphQL client instance. However, rather than using the client directly in your React components

by using the `withClient()` HOC, which would be possible as well, you will implement two components, called Query and Mutation, which are performing the GraphQL operations in your component tree in a declarative way.

Both the Query and Mutation component implement the render props pattern to pass information to the component using Query or Mutation component. The minimal implementation of the Query component could look like the following:

```
import React from 'react';

import withClient from './withClient';

class Query extends React.Component {
  state = {
    data: null,
    loading: null,
    errors: null,
  };

  componentDidMount() {
    const { query, variables } = this.props;

    this.query({ query, variables });
  }

  query = ({ query, variables }) => {
    this.props.client
      .query({ query, variables })
      .then(result =>
        this.setState({
          data: result.data.data,
          errors: result.data.errors,
          loading: false,
        }),
      )
      .catch(error =>
        this.setState({
          errors: [error],
          loading: false,
        }),
      );
  };

  render() {
    return this.props.children({
      ...this.state,
    });
  }
}

export default withClient(Query);
```

The Query component receives a GraphQL query and optional variables as props. Once it mounts, it executes the query using the GraphQL client instance injected with the `withClient` higher-order component. If the request resolves successfully, all data and GraphQL errors are stored in the local state of the Query component. Otherwise, a network error is stored in the local state, in an array of errors. Also, a `loading` boolean tracks the request state. The Query component uses the render prop as a children function to pass in the local state of the component. The user of the Query component decides what should be rendered in response to the information (data, loading, errors) from the children function.

In your App component's file, you can import the component, pass in a query and optional variables, and let the Query component execute the GraphQL query once it mounts. You will receive the information from the Query component in the children function during each render.

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

import { Query } from './my-graphql-client-react';

const GET_ORGANIZATION = `
  query (
    $organizationLogin: String!,
  ) {
    organization(login: $organizationLogin) {
      name
      url
    }
  }
`;

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

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

  onSubmit = event => {
    this.setState({ organizationLogin: this.state.value });

    event.preventDefault();
  };

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

    return (
      <div>
        <h1>React GraphQL GitHub Client</h1>
```

```
        <form onSubmit={this.onSubmit}>
          <label htmlFor="url">
            Show organization for https://github.com/
          </label>
          <input
            id="url"
            type="text"
            value={value}
            onChange={this.onChange}
            style={{ width: '300px' }}
          />
          <button type="submit">Search</button>
        </form>

        <hr />

        <Query
          query={GET_ORGANIZATION}
          variables={{
            organizationLogin,
          }}
        >
          {({ data, loading, errors, fetchMore }) => {
            if (!data) {
              return <p>No information yet ...</p>;
            }

            const { organization } = data;

            if (loading) {
              return <p>Loading ...</p>;
            }

            if (errors) {
              return (
                <p>
                  <strong>Something went wrong:</strong>
                  {errors.map(error => error.message).join(' ')}
                </p>
              );
            }

            return (
              <Organization organization={organization} />
            );
          }}
        </Query>
      </div>
    );
  }
}

const Organization = ({ organization }) => (
  <div>
```

```
    <h1>
      <a href={organization.url}>{organization.name}</a>
    </h1>
  </div>
);

export default App;
```

For the sake of completion, the implementation could also add a list of repositories that belong to the organization. This part of the application provides a good reason to implement pagination later, as well as a mutation with your GraphQL client, Query component, and Mutation component.

```
...

const GET_ORGANIZATION = `
  query (
    $organizationLogin: String!,
  ) {
    organization(login: $organizationLogin) {
      name
      url
      repositories(first: 5) {
        edges {
          node {
            id
            name
            url
            watchers {
              totalCount
            }
            viewerSubscription
          }
        }
      }
    }
  }
`;

const isWatch = viewerSubscription =>
  viewerSubscription === 'SUBSCRIBED';

...

const Organization = ({ organization }) => (
  <div>
    <h1>
      <a href={organization.url}>{organization.name}</a>
    </h1>
    <Repositories
      repositories={organization.repositories}
    />
  </div>
```

```
);

const Repositories = ({ repositories }) => (
  <div>
    <ul>
      {repositories.edges.map(repository => (
        <li key={repository.node.id}>
          <a href={repository.node.url}>{repository.node.name}</a>{' '}
          {repository.node.watchers.totalCount}
          {isWatch(repository.node.viewerSubscription)
            ? ' Watched by you'
            : ' Not watched by you'
          }
        </li>
      ))}
    </ul>
  </div>
);

export default App;
```

The GraphQL query works now, using the Query component. But it only works for the initial request, not when searching for another GitHub organization with the input element. This is because the Query component executes the GraphQL query only when mounts, but not when the `organizationLogin` variable changes. Let's add this little feature in the Query component.

```
import React from 'react';
import { isEqual } from 'lodash';

import withClient from './withClient';

class Query extends React.Component {
  state = {
    data: null,
    loading: null,
    errors: null,
  };

  componentDidMount() {
    ...
  }

  componentDidUpdate(prevProps) {
    if (!isEqual(this.props.variables, prevProps.variables)) {
      const { query, variables } = this.props;

      this.query({ query, variables });
    }
  }

  query = ({ query, variables }) => {
    ...
```

```
  };

  render() {
    ...
  }
}

export default withClient(Query);
```

In this case, lodash is used to make an equal check on the previous and next variables which are passed as props to the Query component. So don't forget to install lodash or any other utility library which can do the check for you.

```
npm install lodash --save
```

Once the variables change, the GraphQL query is executed again. When you try your application, the search for another GitHub organization works now, because when the variable for the `organizationLogin` changes on a submit click, the GraphQL query in the Query component executes again.

IMPLEMENTING THE QUERY COMPONENT WITH PAGINATION IN REACT

We've added some functionality, but the application only fetches the first page of repositories, and there is currently no means to fetch the next page. You have to add a mechanism that executes a query to fetch more pages from the GraphQL backend. To do this, we extend the Query component:

```
...

class Query extends React.Component {
  state = {
    data: null,
    loading: null,
    fetchMoreLoading: null,
    errors: null,
  };

  componentDidMount() {
    ...
  }

  componentDidUpdate(prevProps) {
```

```
    ...
  }

  query = ({ query, variables }) => {
    ...
  };

  queryMore = ({ query, variables }) => {
    this.props.client
      .query({ query, variables })
      .then(result =>
        this.setState(state => ({
          data: this.props.resolveFetchMore(result.data.data, state),
          errors: result.data.errors,
          fetchMoreLoading: false,
        })),
      )
      .catch(error =>
        this.setState({
          errors: [error],
          fetchMoreLoading: false,
        }),
      );
  };

  render() {
    return this.props.children({
      ...this.state,
      fetchMore: this.queryMore,
    });
  }
}

export default withClient(Query);
```

The `queryMore()` method, exposed with the children function as `fetchMore()` function, is used similar to the `query()` method. You switch from a declarative query execution to a imperative query execution using the `fetchMore()` function in React now. There, pass in a query and variables with a pagination argument to the function.

The one crucial difference to the `query()` method is the `resolveFetchMore()` function that is passed to the Query component as prop. It is used when a query resolves successfully, to merge the result with the component state. You can define from the outside how to merge this information.

First, the query needs to provide a cursor argument in the GitHub GraphQL API to fetch another page of repositories. The `pageInfo` field is used to retrieve the cursor for the next page, and to see whether there is a next page.

```
const GET_ORGANIZATION = `
  query (
    $organizationLogin: String!,
    $cursor: String
  ) {
    organization(login: $organizationLogin) {
      name
      url
      repositories(first: 5, after: $cursor) {
        pageInfo {
          endCursor
          hasNextPage
        }
        edges {
          node {
            id
            name
            url
            watchers {
              totalCount
            }
            viewerSubscription
          }
        }
      }
    }
  }
`;
```

Second, the `fetchMore()` function is accessed in the Query's children as a function. The function can be passed down as a wrapped higher-order function to the next component that makes use of it. This way, the next component doesn't have to worry about passing arguments to the function anymore, as it is handled in the App component.

```
...

const resolveFetchMore = (data, state) => {
  ...
}

class App extends Component {
  ...

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

    return (
      <div>
        ...

        <Query
```

```
          query={GET_ORGANIZATION}
          variables={{
            organizationLogin,
          }}
          resolveFetchMore={resolveFetchMore}
        >
          {({ data, loading, errors, fetchMore }) => {
            ...

            return (
              <Organization
                organization={organization}
                onFetchMoreRepositories={() =>
                  fetchMore({
                    query: GET_ORGANIZATION,
                    variables: {
                      organizationLogin,
                      cursor:
                        organization.repositories.pageInfo.endCursor,
                    },
                  })
                }
              />
            );
          }}
        </Query>
      </div>
    );
  }
}

...

export default App;
```

Third, the Repositories component can use the function to fetch the next page of the paginated list of repositories with a button. The button becomes available only when there is a next page of the paginated list.

```
const Organization = ({
  organization,
  onFetchMoreRepositories,
}) => (
  <div>
    <h1>
      <a href={organization.url}>{organization.name}</a>
    </h1>
    <Repositories
      repositories={organization.repositories}
      onFetchMoreRepositories={onFetchMoreRepositories}
    />
  </div>
```

```
);

const Repositories = ({
  repositories,
  onFetchMoreRepositories,
}) => (
  <div>
    <ul>
      ...
    </ul>

    {repositories.pageInfo.hasNextPage && (
      <button onClick={onFetchMoreRepositories}>More</button>
    )}
  </div>
);
```

Next, implement the `resolveFetchMore()` function which was already passed in a previous step to the Query component. In this function, you have access to the query result when you fetch another page, as well as the state of the Query component.

```
const resolveFetchMore = (data, state) => {
  const { edges: oldR } = state.data.organization.repositories;
  const { edges: newR } = data.organization.repositories;

  const updatedRepositories = [...oldR, ...newR];

  return {
    organization: {
      ...data.organization,
      repositories: {
        ...data.organization.repositories,
        edges: updatedRepositories,
      },
    },
  };
};
```

The function merges the edges of the repositories from the state and new result into a new list of edges. The list is used in the returned object, which is used in the Query function for the `data` property in the state. Check the Query component again to verify it. With this resolving function, you can decide how to treat a paginated query, by merging component state of the Query component and the query result into a new state for the Query component.

IMPLEMENTING THE MUTATION COMPONENT IN REACT

So far, you have implemented data reading part with your GraphQL client using a Query component, with pagination. Now you'll add its counterpart, a Mutation component:

```
import React from 'react';

import withClient from './withClient';

class Mutation extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      data: null,
      loading: null,
      errors: null,
    };
  }

  mutate = ({
    mutation = this.props.mutation,
    variables = this.props.variables,
  }) => {
    const { client } = this.props;

    client
      .mutate({ mutation, variables })
      .then(result => {
        this.setState({
          data: result.data.data,
          errors: result.data.errors,
          loading: false,
        });
      })
      .catch(error =>
        this.setState({
          errors: [error],
          loading: false,
        }),
      );
  };

  render() {
    return this.props.children(this.mutate, this.state);
  }
}

export default withClient(Mutation);
```

It is similar to the initial Query component except for three things: the render method, the arguments passed to the mutate method, and the point when the mutate method is executed.

First, the render method gives access to the state of the component, but also to the mutate method to use imperatively from the outside. Second, arguments passed to the mutate method are either the arguments passed to the mutate method at execution or the props passed to the Mutation component as fallback. Third, the mutate method exposed to the outside as a function is used any time except when the Mutation component mounts, as it did in the Query component. It is up to the user of the Mutation component to trigger it.

So how to use it in your App component's file? First, you can implement a mutation which works for GitHub's GraphQL API.

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

Use the new Mutation component in your Repositories component for each repository to watch or unwatch it on GitHub with the mutation.

```
...
import { Query, Mutation } from './my-graphql-client-react';

...

const Repositories = ({
  repositories,
  onFetchMoreRepositories,
}) => (
  <div>
    <ul>
      {repositories.edges.map(repository => (
        <li key={repository.node.id}>
          ...

          <Mutation
            mutation={WATCH_REPOSITORY}
          >
            {(toggleWatch, { data, loading, errors }) => (
              <button
                type="button"
                onClick={() =>
                  toggleWatch({
```

```
                    variables: {
                      id: repository.node.id,
                      viewerSubscription: isWatch(
                        repository.node.viewerSubscription,
                      )
                        ? 'UNSUBSCRIBED'
                        : 'SUBSCRIBED',
                    },
                  })
                }
              >
                {repository.node.watchers.totalCount}
                {isWatch(repository.node.viewerSubscription)
                  ? ' Unwatch'
                  : ' Watch'}
              </button>
            )}
          </Mutation>
        </li>
      ))}
    </ul>

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

The Mutation component grants access to the mutation function and the mutation result in its child as a function. The button can then use the function to watch or unwatch the repository. In this case, the variables are passed in the mutate function, but you could pass them in the Mutation component too.

You may notice your mutation works only once now, as every other mutation keeps same count of watchers, meaning it doesn't toggle between watch and unwatch. This is because the repository prop with the `viewerSubscription` and the `totalCount` properties doesn't change after a mutation, since it is a prop from the Query component above. It is managed in the Query component, not in the Mutation component. You need to manage the data in the Mutation component instead, to update it after a mutation accordingly.

```
import React from 'react';

import withClient from './withClient';

class Mutation extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      data: this.props.initial,
      loading: null,
      errors: null,
```

```
      };
    }

    mutate = ({
      mutation = this.props.mutation,
      variables = this.props.variables,
    }) => {
      const { client, resolveMutation } = this.props;

      client
        .mutate({ mutation, variables })
        .then(result => {
          this.setState(state => ({
            data: resolveMutation(result.data.data, state),
            errors: result.data.errors,
            loading: false,
          }));
        })
        .catch(error =>
          this.setState({
            errors: [error],
            loading: false,
          }),
        );
    };

    render() {
      return this.props.children(this.mutate, this.state);
    }
  }

  export default withClient(Mutation);
```

The previous additions to the Mutation component implemented two requirements:

- The Mutation component has to take over the state of the data to be mutated. In this case, the `initial` prop makes it possible to set an initial state with the data it takes over.
- The Mutation component needs a way to update the state after a successful mutation, to retrieve recent data from it. In this case, the `resolveMutation()` function is passed as prop to the Mutation component, which is used to merge the Mutation component state with the mutation result into a new Mutation component state. This is similar to the `resolveFetchMore()` function from the Query component used for pagination.

After these improvements, you can update the Mutation component in your GitHub client application. Give it the initial state using the prop for it, which should give all the information needed for the Mutation component's render prop function.

```
  const resolveWatchMutation = (data, state) => {
```

```
  ...
};

const Repositories = ({
  repositories,
  onFetchMoreRepositories,
}) => (
  <div>
    <ul>
      {repositories.edges.map(repository => (
        <li key={repository.node.id}>
          <a href={repository.node.url}>{repository.node.name}</a>{' '}
          <Mutation
            mutation={WATCH_REPOSITORY}
            initial={{
              repository: {
                viewerSubscription:
                  repository.node.viewerSubscription,
                totalCount: repository.node.watchers.totalCount,
              },
            }}
            resolveMutation={resolveWatchMutation}
          >
            {(toggleWatch, { data, loading, errors }) => (
              <button
                type="button"
                onClick={() =>
                  toggleWatch({
                    variables: {
                      id: repository.node.id,
                      viewerSubscription: isWatch(
                        data.repository.viewerSubscription,
                      )
                        ? 'UNSUBSCRIBED'
                        : 'SUBSCRIBED',
                    },
                  })
                }
              >
                {data.repository.totalCount}
                {isWatch(data.repository.viewerSubscription)
                  ? ' Unwatch'
                  : ' Watch'}
              </button>
            )}
          </Mutation>
        </li>
      ))}
    </ul>

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

Rather than letting a user outside the Mutation component dictate its data, the Mutation component takes over, only using data provided by its child function for rendering. Once you execute the mutation, the state of the Mutation component should change and the new state should be reflected in the return value of the child function. What's missing is the update to the Mutation component's state using the `resolveMutation` function. It could look like the following, to merge the previous state with the mutation result to a new state object.

```
const resolveWatchMutation = (data, state) => {
  const { totalCount } = state.data.repository;
  const { viewerSubscription } = data.updateSubscription.subscribable;

  return {
    repository: {
      viewerSubscription,
      totalCount:
        viewerSubscription === 'SUBSCRIBED'
          ? totalCount + 1
          : totalCount - 1,
    },
  };
}
```

The resolving function updates the Mutation component's internal state. See the Mutation component's usage of the resolving function again. It takes the `totalCount` of watchers of the repository and increments or decrements it based on the `viewerSubscription` property from the mutation result. The new state is passed as data to the Mutation component's child function. What's important is that the resolving function has to return the identical structure of the data provided to the Mutation component with the `initial` prop. Otherwise, your rendering may break, because data from the render prop function has lost its identical structure.

If the props used for the `initial` prop of the Mutation component changes in the Mutation component, nothing reflects this update. We'll need to add a lifecycle method in the Mutation component to update its local state when a new `initial` prop comes in.

```
import React from 'react';
import { isEqual } from 'lodash';

import withClient from './withClient';

class Mutation extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      data: this.props.initial,
      loading: null,
```

```
        errors: null,
      };
    }

    componentDidUpdate(prevProps) {
      if (!isEqual(this.props.initial, prevProps.initial)) {
        this.setState({ data: this.props.initial });
      }
    }

    ...
}

export default withClient(Mutation);
```

Everything is in place for the Mutation component now. It manages its own state, which is used as data in the Mutation component's render prop function. We've implemented a Mutation component that handles the GraphQL mutation using your GraphQL client in a React application.

. . .

There's a big difference between this lightweight GraphQL client and a sophisticated GraphQL client like Apollo Client. Obviously, the Apollo Client has more features, probably has better performance, and certainly gives more options. The key difference is where the data is stored. Apollo Client has a central cache to manage all normalized data, but the lightweight GraphQL client manages the data in the intermediate Query and Mutation components. They are only locally available to the React components using the Query and Mutation components and the components below them using React's props.

This implementation of a GraphQL client in React should have illustrated that it's not as complicated as it seems. Hopefully this will eventually inspire you to contribute to the GraphQL and React ecosystem with your own libraries. Perhaps you'll create more sophisticated GraphQL client libraries on top of the previous ideas. You can find the previous GraphQL client as library and repository on GitHub. Check it out to find your own solutions and open source them on npm as library. I am looking forward to seeing what you come up with, so please contact me when you'd like to discuss contributing to the ecosystem.

Show Comments
Continue Reading: A minimal Apollo Client in React Example

KEEP READING ABOUT GRAPHQL ›

APOLLO CLIENT TUTORIAL FOR BEGINNERS

Apollo is an entire ecosystem built by developers as an infrastructure for GraphQL applications. You can use it on the client-side for a GraphQL client application, or server-side for a GraphQL server...

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...

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