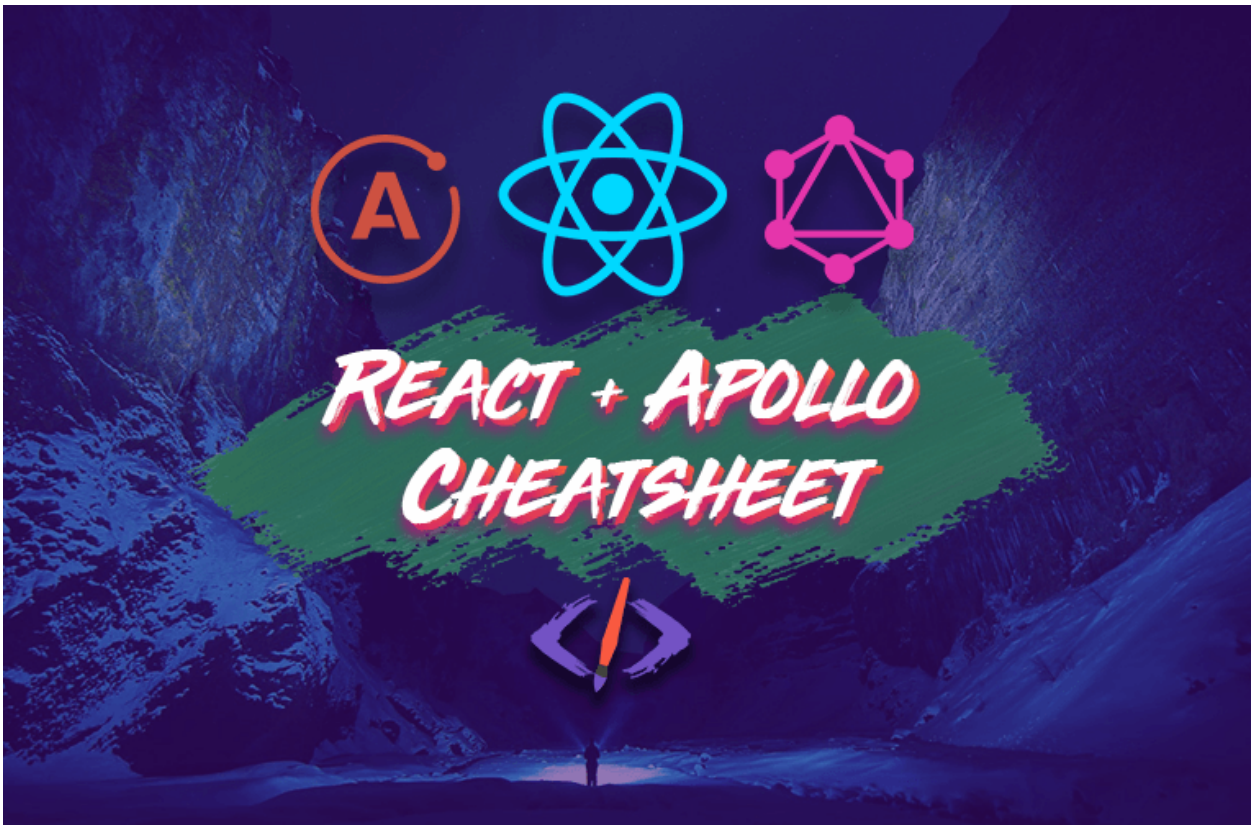
React + Apollo Client 2020 Cheatsheet



💡 Note: This cheatsheet does assume familiarity with React and GraphQL. If you need a quick refresher on GraphQL and how to write it, a great resource is the official GraphQL website.

Table of Contents

Getting Started

- Using the client directly
- Writing GraphQL in .js files with gql

Core Apollo React Hooks

- useQuery Hook
- useLazyQuery Hook
- useMutation Hook
- useSubscription Hook

Essential Recipes

- Manually setting the fetch policy
- Updating the cache upon a mutation
- Refetching queries with useQuery
- Refetching queries with useMutation
- Accessing the client with useApolloClient

What is Apollo and why do we need it?

Apollo is a library that brings together two incredibly useful technologies used to build web and mobile apps—React and GraphQL.

React was made for creating great user experiences with JavaScript. GraphQL is a very straightforward and declarative new language to more easily and efficiently fetch and change data, whether it is from a database or even from static files.

Apollo is the glue that binds these two tools together, plus makes working with them a lot easier by giving us a lot of custom React hooks and features that enable us to both write GraphQL operations and JavaScript files and execute them with JavaScript code. We'll cover these features in depth throughout the course of this guide.

Apollo Client basic setup

If you are starting a project with a React template like Create React App, you will need to install the following as your base dependencies to get up and running with Apollo Client.

```
// with npm:
npm i @apollo/react-hooks apollo-boost graphql

// with yarn:
yarn add @apollo/react-hooks apollo-boost graphql
```

`@apollo/react-hooks` gives us React hooks that make performing our operations and working with Apollo client better

`apollo-boost` helps us set up the client along with parse our GraphQL operations

`graphql` also takes care of parsing the GraphQL operations (along with gql)

Apollo Client + subscriptions setup

To use all manner of GraphQL operations (queries, mutations, and subscriptions), we need to install more specific dependencies as compared to just `apollo-boost`.

```
// with npm:
npm i @apollo/react-hooks apollo-client graphql graphql-tag apollo-cache-inmemory apollo-link-ws

// with yarn:
yarn add @apollo/react-hooks apollo-client graphql graphql-tag apollo-cache-inmemory apollo-link-ws
```

`apollo-client` gives us the client directly, instead of from `apollo-boost`

`graphql-tag` is integrated into `apollo-boost`, but not included in `apollo-client`

`apollo-cache-inmemory` is needed to setup our own cache (which `apollo-boost`, in comparison, does automatically)

`apollo-link-ws` is needed for communicating over websockets, which subscriptions require

Creating a new Apollo Client (basic setup)

The most straightforward setup for creating an Apollo client is by instantiating a new client and providing just the `uri` property, which will be your GraphQL endpoint:

```
import ApolloClient from "apollo-boost";

const client = new ApolloClient({
  uri: "https://your-graphql-endpoint.com/api/graphql"
});
```

`apollo-boost` was developed in order to make doing things like creating an Apollo Client as easy as possible. What it lacks for the time being, however, is support for GraphQL subscriptions over a websocket connection.

By default, it performs the operations over an http connection (as you can see through our provided uri above).

In short, use `apollo-boost` to create your client if you only need to execute queries and mutations in your app.

It setups an in-memory cache by default, which is helpful for storing our app data locally. We can read from and write to our cache to prevent having to execute our queries after our data is updated. We'll cover how to do that a bit later.

Creating a new Apollo Client (+ subscriptions setup)

Subscriptions are useful for more easily displaying the result of data changes (through mutations) in our app.

Generally speaking, we use subscriptions as an improved kind of query. Subscriptions use a websocket connection to 'subscribe' to updates and data, enabling new or updated data to be immediately displayed to our users without having to reexecute queries or update the cache.

```
import ApolloClient from "apollo-client";
import { WebSocketLink } from "apollo-link-ws";
import { InMemoryCache } from "apollo-cache-inmemory";

const client = new ApolloClient({
  link: new WebSocketLink({
    uri: "wss://your-graphql-endpoint.com/v1/graphql",
    options: {
      reconnect: true,
      connectionParams: {
        headers: {
          Authorization: "Bearer yourauthtoken"
        }
      },
    },
  }),
  cache: new InMemoryCache(),
});
```

Providing the client to React components

After creating a new client, passing it to all components is essential in order to be able to use it within our components to perform all of the available GraphQL operations.

The client is provided to the entire component tree using React Context, but instead of creating our own context, we import a special context provider from `@apollo/react-hooks` called `ApolloProvider` . We can see how it differs from the regular React Context due to it having a special prop, `client`, specifically made to accept the created client.

Note that all of this setup should be done in your index.js or App.js file (wherever your Routes declared) so that the Provider can be wrapped around all of your components.

```
import { ApolloProvider } from "@apollo/react-hooks";

const rootElement = document.getElementById("root");
ReactDOM.render(
```

```
  <React.StrictMode>
    <ApolloProvider client={client}>
      <BrowserRouter>
        <Switch>
          <Route exact path="/" component={App} />
          <Route exact path="/new" component={NewPost} />
          <Route exact path="/edit/:id" component={EditPost} />
        </Switch>
      </BrowserRouter>
    </ApolloProvider>
  </React.StrictMode>,
  rootElement
)
```

Using the client directly

The Apollo client is most important part of the library due to the fact that it is responsible for executing all of the GraphQL operations that we want to perform with React.

We can use the created client directly to perform any operation we like. It has methods corresponding to queries (`client.query()`), mutations (`client.mutate()`), and subscriptions (`client.subscribe()`).

Each method accepts an object and it's own corresponding properties:

```
// executing queries
client.query({
  query: GET_POSTS,
  variables: { limit: 5 }
})
.then(response => console.log(response.data))
.catch(err => console.error(err));

// executing mutations
client.mutate({
  mutation: CREATE_POST,
  variables: { title: "Hello", body: "World" }
})
.then(response => console.log(response.data))
.catch(err => console.error(err));

// executing subscriptions
client.subscribe({
  subscription: GET_POST,
  variables: { id: "8883346c-6dc3-4753-95da-0cc0df750721" }
})
.then(response => console.log(response.data))
.catch(err => console.error(err));
```

Using the client directly can be a bit tricky, however, since in making a request, it returns a promise. To resolve each promise, we either need `.then()` and `.catch()` callbacks as above or to `await` each promise within a function declared with the `async` keyword.

Writing GraphQL operations in .js files (gql)

Notice above that I didn't specify the contents of the variables `GET_POSTS`, `CREATE_POST`, and `GET_POST`.

They are the operations written in the GraphQL syntax which specify how to perform the query, mutation, and subscription respectively. They are what we would write in any GraphiQL console to get and change data.

The issue here, however, is that we can't write and execute GraphQL instructions in JavaScript (.js) files, like our React code has to live in.

To parse the GraphQL operations, we use a special function called a tagged template literal to allow us to express them as JavaScript strings. This function is named `gql`.

```
// if using apollo-boost
import { gql } from 'apollo-boost';
// else, you can use a dedicated package graphql-tag
import gql from 'graphql-tag';

// query
const GET_POSTS = gql`
 query GetPosts($limit: Int) {
    posts(limit: $limit) {
      id
      body
      title
      createdAt
    }
  }
`;

// mutation
const CREATE_POST = gql`
  mutation CreatePost($title: String!, $body: String!) {
    insert_posts(objects: { title: $title, body: $body }) {
      affected_rows
    }
  }
`;

// subscription
const GET_POST = gql`
  subscription GetPost($id: uuid!) {
    posts(where: {id: {_eq: $id}}) {
      id
      body
      title
      createdAt
    }
  }
`;
```

useQuery Hook

The `useQuery` hook is arguably the most convenient way of performing a GraphQL query, considering that it doesn't return a promise that needs to be resolved.

It is called at the top of any function component (as all hooks should be) and receives as a first required argument—a query parsed with `gql`.

It is best used when you have queries that should be executed immediately, when a component is rendered, such as a list of data which the user would want to see immediately when the page loads.

`useQuery` returns an object from which we can easily destructure the values that we need. Upon executing a query, there are three primary values will need to use within every component in which we fetch data. They are `loading`, `error`, and `data`.

```
const GET_POSTS = gql`
 query GetPosts($limit: Int) {
    posts(limit: $limit) {
      id
      body
      title
      createdAt
    }
  }
`;

function App() {
  const { loading, error, data } = useQuery(GET_POSTS, {
    variables: { limit: 5 }
  });

  if (loading) return <div>Loading...</div>;
  if (error) return <div>Error!</div>;

  return data.posts.map(post => (
     <Post key={post.id} post={post} />
  ))
}
```

Before we can display the data that we're fetching, we need to handle when we're loading (when `loading` is set to true) and we are attempting to fetch the data.

At that point, we display a div with the text 'Loading' or a loading spinner. We also need to handle the possibility that there is an error in fetching our query, such as if there's a network error or if we made a mistake in writing our query (syntax error).

Once we're done loading and there's no error, we can use our data in our component, usually to display to our users (as we are in the example above).

There are other values which we can destructure from the object that `useQuery` returns, but you'll need `loading`, `error`, and `data` in virtually every component where you execute

`useQuery`. You can see a full list of all of the data we can get back from useQuery here.

useLazyQuery Hook

The `useLazyQuery` hook provides another way to perform a query, which is intended to be executed at some time after the component is rendered or in response to a given data change.

`useLazyQuery` is very useful for things that happen at any unknown point of time, such as in response to a user's search operation.

```
function Search() {
  const [query, setQuery] = React.useState("");
  const [searchPosts, { data }] = useLazyQuery(SEARCH_POSTS, {
    variables: { query: `%${query}%` }
  });
  const [results, setResults] = React.useState([]);

  React.useEffect(() => {
    if (!query) return;
    // function for executing query doesn't return a promise
    searchPosts();
    if (data) {
      setResults(data.posts);
    }
  }, [query, data, searchPosts]);

  if (called && loading) return <div>Loading...</div>

  return results.map(result => (
    <SearchResult key={result.id} result={result} />
  ))
}
```

`useLazyQuery` differs from `useQuery`, first of all, in what's returned from the hook. It returns an array which we can destructure, instead of an object.

Since we want to perform this query sometime after the component is mounted, the first element that we can destructure is a function which you can call to perform that query when you choose. This query function is named `searchPosts` in the example above.

The second destructured value in the array is an object, which we can use object destructuring on and from which we can get all of the same properties as we did from `useQuery`, such as `loading`, `error`, and `data`.

We also get an important property named `called`, which tells us if we've actually called this function to perform our query. In that case, if `called` is true and `loading` is true, we want to return "Loading..." instead of our actual data, because are waiting for the data to be

returned. This is how `useLazyQuery` handles fetching data in a synchronous way without any promises.

Note that we again pass any required variables for the query operation as a property, variables, to the second argument. However, if we need, we can pass those variables on an object provided to the query function itself.

useMutation Hook

Now that we know how to execute lazy queries, we know exactly how to work with the `useMutation` hook.

Like the `useLazyQuery` hook, it returns an array which we can destructure into its two elements. In the first element, we get back a function, which in this case, we can call it to perform our mutation operation. For next element, we can again destructure an object which returns to us `loading`, `error` and `data`.

```
import { useMutation } from "@apollo/react-hooks";
import { gql } from "apollo-boost";

const CREATE_POST = gql`
  mutation CreatePost($title: String!, $body: String!) {
    insert_posts(objects: { body: $body, title: $title }) {
      affected_rows
    }
  }
`;

function NewPost() {
  const [title, setTitle] = React.useState('');
  const [body, setBody] = React.useState('');
  const [createPost, { loading, error }] = useMutation(CREATE_POST);

  function handleCreatePost(event) {
    event.preventDefault()
    // the mutate function also doesn't return a promise
    createPost({ variables: { title, body } });
  }

  return (
    <div>
      <h1>New Post</h2>
      <form onSubmit={handleCreatePost}>
        <input
          onChange={event => setTitle(event.target.value)}
        />
        <textarea
          onChange={event => setBody(event.target.value)}
        />
        <button disabled={loading} type="submit">
          Submit
        </button>
        {error && <p>{error.message}</p>}
      </form>
    </div>
```

```
  );
}
```

Unlike with queries, however, we don't use `loading` or `error` in order to conditionally render something. We generally use `loading` in such situations as when we're submitting a form to prevent it being submitted multiple times, to avoid executing the same mutation needlessly (as you can see in the example above).

We use `error` to display what goes wrong with our mutation to our users. If for example, some required values to our mutation are not provided, we can easily use that error data to conditionally render an error message within the page so the user can hopefully fix what's going wrong.

As compared to passing variables to the second argument of `useMutation`, we can access a couple of useful callbacks when certain things take place, such as when the mutation is completed and when there is an error. These callbacks are named `onCompleted` and `onError`.

The `onCompleted` callback gives us access to the returned mutation data and it's very helpful to do something when the mutation is done, such as going to a different page. The `onError` callback gives us the returned error when there is a problem with the mutation and gives us other patterns for handling our errors.

```
const [createPost, { loading, error }] = useMutation(CREATE_POST, {
   onCompleted: (data) => console.log('Data from a successful mutation', data),
   onError: (error) => console.error('Error creating a post', error)
 });
```

useSubscription Hook

The useSubscription hook works just like the useQuery hook.

useSubscription returns an object that we can destructure, that includes the same properties, loading, data, and error.

It executes our subscription immediately when the component is rendered. This means we need to handle loading and error states, and only afterwards display/use our data.

```
import { useSubscription } from '@apollo/react-hooks';
import gql from 'graphql-tag';

const GET_POST = gql`
  subscription GetPost($id: uuid!) {
    posts(where: {id: {_eq: $id}}) {
      id
      body
      title
      createdAt
    }
```

```
  }
`;

// where id comes from route params -> /post/:id
function PostPage({ id }) {
  const { loading, error, data } = useSubscription(GET_POST, {
    variables: { id },
    // shouldResubscribe: true (default: false)
    // onSubscriptionData: data => console.log('new data', data)
    // fetchPolicy: 'network-only' (default: 'cache-first')
  });

  if (loading) return <div>Loading...</div>;
  if (error) return <div>Error!</div>;

  const post = data.posts[0];

  return (
    <div>
      <h1>{post.title}</h1>
      <p>{post.body}</p>
    </div>
  );
}
```

Just like useQuery, useLazyQuery and useMutation, useSubscription accepts `variables` as a property provided on the second argument.

It also accepts, however, some useful properties such as `shouldResubscribe`. This is a boolean value, which will allow our subscription to automatically resubscribe, when our props change. This is useful for when we're passing variables to our you subscription hub props that we know will change.

Additionally, we have a callback function called `onSubscriptionData`, which enables us to call a function whenever the subscription hook receives new data. Finally, we can set the `fetchPolicy`, which defaults to 'cache-first'.

Manually Setting the Fetch Policy

What can be very useful about Apollo is that it comes with its own cache, which it uses to manage the data that we query from our GraphQL endpoint.

Sometimes, however, we find that due to this cache, things aren't updated in the UI in the way that we want.

In many cases we don't, as in the example below, where we are editing a post on the edit page, and then after editing our post, we navigate to the home page to see it in a list of all posts, but we see the old data instead:

```
// route: /edit/:postId
function EditPost({ id }) {
  const { loading, data } = useQuery(GET_POST, { variables: { id } });
```

```
  const [title, setTitle] = React.useState(loading ? data?.posts[0].title : "")
  const [body, setBody] = React.useState(loading ? data?.posts[0].body : "")
  const [updatePost] = useMutation(UPDATE_POST, {
    // after updating the post, we go to the home page
    onCompleted: () => history.push("/")
  });

  function handleUpdatePost(event) {
    event.preventDefault();
    updatePost({ variables: { title, body, id } });
  }

  return (
    <form onSubmit={handleUpdatePost}>
      <input
        onChange={event => setTitle(event.target.value)}
        defaultValue={title}
      />
      <input
        onChange={event => setBody(event.target.value)}
        defaultValue={body}
      />
      <button type="submit">Submit</button>
    </form>
  );
}

// route: / (homepage)
function App() {
  const { loading, error, data } = useQuery(GET_POSTS, {
    variables: { limit: 5 }
  });

  if (loading) return <div>Loading...</div>;
  if (error) return <div>Error!</div>;

  // updated post not displayed, still see old data
  return data.posts.map(post => (
     <Post key={post.id} post={post} />
  ))
}
```

This not only due to the Apollo cache, but also the instructions for what data the query should fetch. We can changed how the query is fetched by using the `fetchPolicy` property.

By default, the `fetchPolicy` is set to 'cache-first'. It's going to try to look at the cache to get our data instead of getting it from the network.

An easy way to fix this problem of not seeing new data is to change the fetch policy. However, this approach is not ideal from a performance standpoint, because it requires making an additional request (using the cache directly does not, because it is local data).

There are many different options for the fetch policy listed below:

```
{
  fetchPolicy: "cache-first" // default
```

```
    /*
      cache-and-network
      cache-first
      cache-only
      network-only
      no-cache
      standby
    */
}
```

I won't go into what each policy does exactly, but to solve our immediate problem, if you always want a query to get the latest data by requesting it from the network, we set `fetchPolicy` to 'network-first'.

```
const { loading, error, data } = useQuery(GET_POSTS, {
  variables: { limit: 5 },
  fetchPolicy: "network-first"
});
```

Updating the cache upon a mutation

Instead of bypassing the cache by changing the fetch policy of `useQuery`, let's attempt to fix this problem by manually updating the cache.

When performing a mutation with `useMutation`. We have access to another callback, known as `update`.

`update` gives us direct access to the cache as well as the data that is returned from a successful mutation. This enables us to read a given query from the cache, take that new data and write the new data to the query, which will then update what the user sees.

Working with the cache manually is a tricky process that a lot of people tend to avoid, but it's very helpful because it saves some time and resources by not having to perform the same request multiple times to update the cache manually.

```
function EditPost({ id }) {
   const [updatePost] = useMutation(UPDATE_POST, {
    update: (cache, data) => {
      const { posts } = cache.readQuery(GET_POSTS)
      const newPost = data.update_posts.returning;
      const updatedPosts = posts.map(post => post.id === id ? newPost : post);
      cache.writeQuery({query: GET_POSTS, data: { posts: updatedPosts }});
    },
    onCompleted: () => history.push("/")
  });
}
```

We first want to read the query and get the previous data from it. Then we need to take the new data. In this case, to find the post with a given id and replace it with `newPost` data, otherwise have it be the previous data, and then write that data back to the same query, making sure that it has the same data structure as before.

After all this, whenever we edit a post and are navigated back to the home page, we should see that new post data.

Refetching queries with useQuery

Let's say we display a list of posts using a `GET_POSTS` query and are deleting one of them with a `DELETE_POST` mutation.

When a user deletes a post, what do we want to happen?

Naturally, we want it to be removed from the list, both the data and what is displayed to the users. When a mutation is performed, however, the query doesn't know that the data is changed.

There are a few ways of updating what we see, but one approach is to reexecute the query.

We can do so by grabbing the `refetch` function which we can destructure from the object returned by the `useQuery` hook and pass it down to the mutation to be executed when it is completed, using the `onCompleted` callback function:

```
function Posts() {
  const { loading, data, refetch } = useQuery(GET_POSTS);

  if (loading) return <div>Loading...</div>;

  return data.posts.map(post => (
     <Post key={post.id} post={post} refetch={refetch} />
   ))
}

function Post({ post, refetch}) {
   const [deletePost] = useMutation(DELETE_POST, {
     onCompleted: () => refetch()
   });

  function handleDeletePost(id) {
    if (window.confirm("Are you sure you want to delete this post?")) {
      deletePost({ variables: { id } });
    }
  }

  return (
    <div>
      <h1>{post.title}</h1>
      <p>{post.body}</p>
      <button onClick={() => handleDeletePost(post.id)}>Delete</button>
    </div>
  )
}
```

Refetching Queries with useMutation

Note that we can also utilize the `useMutation` hook to reexecute our queries through an argument provided to the mutate function, called `refetchQueries`.

It accepts an array of queries that we want to refetch after a mutation is performed. Each queries is provided within an object, just like we would provide it to client.query(), and consists of a query property and a variables property.

Here is a minimal example to refetch our `GET_POSTS` query after a new post is created:

```
function NewPost() {
  const [createPost] = useMutation(CREATE_POST, {
    refetchQueries: [{ query: GET_POSTS, variables: { limit: 5 } }],
  });
}
```

Using the client with useApolloClient

We can get access to the client across our components with the help of a special hook called use Apollo client. This execute the hook at the top of our function component and we get back the client itself.

```
function Logout() {
  const client = useApolloClient();
  // client is the same as what we created with new ApolloClient()

  function handleLogout() {
    // handle logging out user, then clear stored data
    logoutUser();
    client
     .resetStore()
     .then(() => console.log('logged out!'));
    /* Be aware that .resetStore() is async */
  }

  return (
    <button onClick={handleLogout}>Logout</button>
  );
}
```

And from there we can execute all the same queries, mutations, and subscriptions.

Note that there are a ton more features that come with methods that come with the client. Using the client, we can also write and read data to and from the cache that Apollo sets up (using client.readData() and client.writeData()).

Working with the Apollo cache deserves its own crash course in itself. A great benefit of working with Apollo, is that we can also use it as a state management system to replace solutions like Redux for our global state. If you want to learn more about using Apollo to manage global app state you can check out the following link.

What's Next

I attempted to make this cheatsheet as comprehensive as possible, though it still leaves out many Apollo features that are worth investigating.

If you want to learn Apollo further, be sure to check out the official Apollo documentation.