This approach is efficient as the route component is only rendered once because the data is available on the first render.

For more information on React Router loaders, see the following link: `https://reactrouter.com/en/main/route/loader`. For more information on the `useLoaderData` hook, see the following link: `https://reactrouter.com/en/main/hooks/use-loader-data`.

Now that we are starting to understand data loading in React Router, we will use this in our app.

Using React Router for data loading

Carry out the following steps to use a React Router data loader in our app:

1. Open `App.tsx` and add the following import statement:

```
import {
  createBrowserRouter,
  RouterProvider
} from 'react-router-dom';
```

2. Also, import the `getPosts` function:

```
import { getPosts } from './posts/getPosts';
```

`getPosts` will be the loader function.

3. Add the following router definition above the `App` component:

```
const router = createBrowserRouter([
  {
    path: "/",
    element: <PostsPage />,
    loader: getPosts
  }
]);
```

4. In the `App` component, replace `PostsPage` with `RouterProvider`:

```
function App() {
  return <RouterProvider router={router} />;
}
```

5. Open `PostsPage.tsx` and remove the React import statement, as this is no longer required in this component.

6. Also, add `assertIsPosts` to the `getPosts` import statement and remove `getPosts`:

```
import { assertIsPosts } from './getPosts';
```

   We will eventually need `assertIsPosts` to type the data.

7. Still in `PostsPage.tsx`, add the following import statement for a hook in React Router that allows us to access the loader data:

```
import { useLoaderData } from 'react-router-dom';
```

8. Inside the `PostsPage` component, remove the `isLoading` and `posts` state definitions. These won't be needed because we will get the data from React Router without having to do any waiting.
9. Remove the call to `useEffect` that currently gets the data.
10. Remove the second line of the `handleSave` function that sets the `posts` state. `handleSave` should now read as follows:

```
async function handleSave(newPostData: NewPostData) {
  await savePost(newPostData);
}
```

11. Remove the loading indicator as well.
12. Now at the top of the `PostsPage` component, make a call to `useLoaderData` and assign the result to a `posts` variable:

```
export function PostsPage() {
  const posts = useLoaderData();
  …
}
```

13. Unfortunately, `posts` is of the `unknown` type, so there is a type error where it is passed to the `PostsLists` component. Use the `assertsIsPosts` function to type the data with `PostData[]`:

```
const posts = useLoaderData();
assertIsPosts(posts);
```

    The type errors are now resolved.

    Note that `PostData` from the `types` import statement is unused. Leave it intact because we will use this again in the next section.

14. The running app should look and behave similarly to how it previously did. One thing you may notice is that when a new blog post is added using the form, it doesn't appear in the list – you have to manually refresh the page for it to appear. This will be resolved when we use React Query later in this chapter.

Notice how much code we just removed – this indicates that the code is much simpler now. Another benefit of using React Router to load the data is that `PostsPage` isn't re-rendered after the data is fetched – the data is fetched before `PostsPage` is rendered.

Next, we will improve the user experience of the data-fetching process.

Deferred React Router data fetching

If the data-fetching process is slow, there will be a noticeable delay before a component is rendered by React Router. Fortunately, there is a solution to this using React Router's `defer` function and `Await` component, along with React's `Suspense` component. Carry out the following steps to add these to our app:

1. Start by opening `App.tsx` and add the `defer` function to the React Router import statement:

```
import {
  createBrowserRouter,
  RouterProvider,
  defer
} from 'react-router-dom';
```

2. Update the `loader` function as follows in the route definition:

```
const router = createBrowserRouter([
  {
    path: "/",
    element: ...,
    loader: async () => defer({ posts: getPosts() })
  }
]);
```

   React Router's `defer` function takes in an object of promised data. The property name in the object is a unique key for the data, which is `posts` in our case. The value is the function that fetches the data, which is `getPosts` in our case.

   Notice that we don't await `getPosts` because we want the loader to complete and `PostsPage` to immediately render.

3. Open `PostsPage.tsx` and add an import statement for React's `Suspense` component:

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

4. Add the `Await` component to the React Router import statement:

```
import { useLoaderData, Await } from 'react-router-dom';
```

5. In the component, update the call to `useLoaderData` to assign the result to a `data` variable instead of `posts`:

```
const data = useLoaderData();
```

The shape of the loader data is a little different now – it will be an object containing a `posts` property containing the blog posts. The blog posts also won't immediately be there as they previously were – the `data.posts` property will contain a promise for the blog posts instead.

6. Also, remove the call to `assertIsPosts` – we will use this later in *step 9*.

7. The `data` variable is of the `unknown` type, so add a type assertion function beneath the component that can be used to strongly type it:

```
type Data = {
  posts: PostData[];
};
export function assertIsData(
  data: unknown
): asserts data is Data {
  if (typeof data !== 'object') {
    throw new Error("Data isn't an object");
  }
  if (data === null) {
    throw new Error('Data is null');
  }
  if (!('posts' in data)) {
    throw new Error("data doesn't contain posts");
  }
}
```

The type assertion function checks that the `data` parameter is an object containing a `posts` property.

8. We can now use the assertion function to type the `data` variable in the component:

```
const data = useLoaderData();
assertIsData(data);
```

9. In the JSX, wrap `Suspense` and `Await` around `PostsList` as follows:

```
<Suspense fallback={<div>Fetching...</div>}>
  <Await resolve={data.posts}>
    {(posts) => {
      assertIsPosts(posts);
      return <PostsList posts={posts} />;
    }}
  </Await>
</Suspense>
```

`Suspense` and `Await` work together to only render `PostsLists` when the data has been fetched. We use `Suspense` to render a **Fetching…** message while the data is being fetched. We also use `assertIsPosts` to ensure that `posts` is typed correctly.

10. In the running app, you will now notice the **Fetching…** message when the page loads:

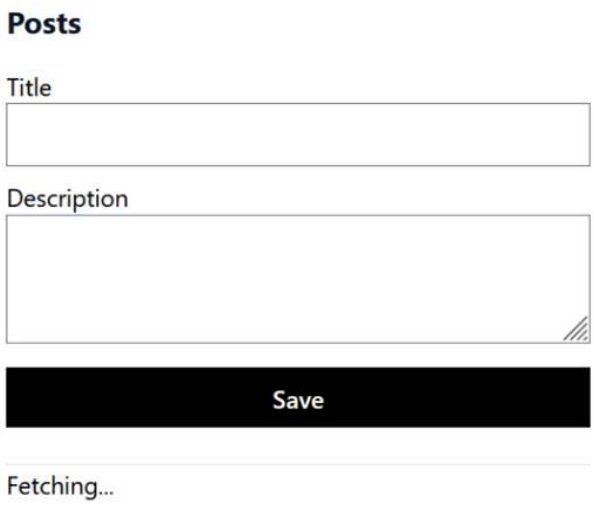

Figure 9.7 – Fetching message during data fetching

11. Stop the app from running by pressing *Ctrl* + *C* in the terminal that is running the app but keep the API running.

The great thing about this solution is that a re-render still doesn't occur when `PostsPage` is rendered because of the use of `Suspense` and `Await`.

We will now quickly recap what we have learned with React Router's data-fetching capabilities:

- React Router's `loader` allows us to efficiently load fetched data into a route component
- React Router's `defer` allows the route component not to be blocked from rendering the component while data is being fetched
- React Router's `useLoaderData` hook allows a component to access a route's loader data
- React's `Suspense` and React Router's `Await` allow a component to render while data is still being fetched

For more information on deferred data in React Router, see the following link: `https://reactrouter.com/en/main/guides/deferred`.

In the next section, we will use another popular library for managing server data to further improve the user experience.

Using React Query

React Query is a popular library for interacting with REST APIs. The key thing it does is manage the state surrounding REST API calls. One thing that it does that React Router doesn't is that it maintains a cache of the fetched data, which improves the perceived performance of an app.

In this section, we will refactor the app to use React Query rather than React Router's loader capability. We will then refactor the app again to use both React Query and React Router's loader to get the best of both these worlds.

Installing React Query

Our first job is to install React Query, which we can do by running the following command in a terminal:

```
npm i @tanstack/react-query
```

This library includes TypeScript types, so no additional package is required to be installed.

Adding the React Query provider

React Query requires a provider component in the component tree above the components that need access to the data it manages. Eventually, React Query will hold the blog post data in our app. Carry out the following steps to add the React Query provider component to the `App` component:

1. Open `App.tsx` and add the following import statement:

```
import {
  QueryClient,
```

```
  QueryClientProvider,
} from '@tanstack/react-query';
```

`QueryClient` provides access to the data. `QueryClientProvider` is the provider component we need to place in the component tree.

2. Wrap `QueryClientProvider` around `RouterProvider` as follows:

```
const queryClient = new QueryClient();
const router = createBrowserRouter( ... );
function App() {
  return (
    <QueryClientProvider client={queryClient}>
      <RouterProvider router={router} />
    </QueryClientProvider>
  );
}
```

`QueryClientProvider` requires an instance of `QueryClient` to be passed into it, so we create this instance outside of the `App` component. We place the `queryClient` variable above the router definition because we will eventually use it in the router definition.

The `PostsPage` component now has access to React Query. Next, we will use React Query in `PostsPage`.

Getting data using React Query

React Query refers to a request to fetch data as a **query** and has a `useQuery` hook to carry out this. We will use React Query's `useQuery` hook in the `PostsPage` component to call the `getPosts` function and store the data it returns. This will temporarily replace the use of React Router's loader. Carry out the following steps:

1. Import `useQuery` from React Query:

```
import { useQuery } from '@tanstack/react-query';
```

2. Add `getPosts` to the `getPosts` import statement:

```
import { assertIsPosts, getPosts } from './getPosts';
```

We will eventually use `getPosts` to fetch data and store it within React Query.

3. In the `PostPage` component, comment out the `data` variable:

```
// const data = useLoaderData();
// assertIsData(data);
```

We are commenting these lines out rather than removing them because we will use them again in the next section when we use React Router and React Query together.

4. Now, add a call to `useQuery` as follows:

```
export function PostsPage() {
  const {
    isLoading,
    isFetching,
    data: posts,
  } = useQuery(['postsData'], getPosts);
  // const data = useLoaderData();
  // assertIsData(data);
  ...
}
```

The first argument passed to `useQuery` is a unique key for the data. This is because React Query can store many datasets and uses the key to identify each one. The key is an array containing the name given to the data in our case. However, the key array could include things like the ID of a particular record we want to fetch or a page number if we want to only fetch a page of records.

The second argument passed to `useQuery` is the fetching function, which is our existing `getPosts` function.

We have destructured the following state variables:

- `isLoading` – Whether the component is being loaded for the first time.
- `isFetching` – Whether the fetching function is being called. React Query will refetch data when it thinks it is stale. We will experience refetching later when we play with the app.
- `data` – The data that has been fetched. We have aliased this `posts` variable to match the previous `posts` state value. Keeping the same name minimizes the changes required in the rest of the component.

> **Note**
>
> There are other useful state variables that can be destructured from `useQuery`. An example is `isError`, which indicates whether the `fetch` function errored. See the following link for more information: `https://tanstack.com/query/v4/docs/reference/useQuery`.

5. Add a loading indicator above the return statement:

```
if (isLoading || posts === undefined) {
  return (
    <div className="w-96 mx-auto mt-6">
```

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

The check for the `posts` state being `undefined` means that the TypeScript compiler knows that `posts` isn't `undefined` when referenced in the JSX.

6. In the JSX, comment out `Suspense` and its children:

```
return (
  <div className="w-96 mx-auto mt-6">
    <h2 className="text-xl text-slate-900 font-
      bold">Posts</h2>
    <NewPostForm onSave={mutate} />
    {/* <Suspense fallback={<div>Fetching ...</div>}>
        <Await resolve={data.posts}
          errorElement={<p>Error!</p>}>
          {(posts) => {
            assertIsPosts(posts);
            return <PostsList posts={posts} />;
          }}
        </Await>
      </Suspense> */}
  </div>
);
```

We have commented this block out rather than removing it because we will revert to it in the next section when we use React Router and React Query together.

7. When data is being fetched, display a fetching indicator and render the blog posts when the data has been fetched:

```
<div className="w-96 mx-auto mt-6">
  <h2 className="text-xl text-slate-900 font-
    bold">Posts</h2>
  <NewPostForm onSave={handleSave} />
  {isFetching ? (
    <div>Fetching ...</div>
  ) : (
```

```
      <PostsList posts={posts} />
    )}
    ...
  </div>
```

8. Run the app by running `npm start` in the terminal. The blog post page will appear the same as it did before. A technical difference is that the `PostsPage` is re-rendered after the data has been fetched.
9. Leave the browser window and set the focus to a different window, such as your code editor. Now, set your focus back on the browser window and notice that the fetching indicator appears for a split second:

Figure 9.8 – Fetching indicator when data is refetched

This is because React Query, by default, assumes that data is stale when the browser regains focus. For more information on this behavior, see the following link in the React Query documentation: `https://tanstack.com/query/v4/docs/guides/window-focus-refetching`.

10. A great feature of React Query is that it maintains a cache of the data. This allows us to render components with data from the cache while fresh data is being fetched. To experience this, in the `PostsPage` JSX, remove the `isFetching` condition for when `PostsList` is rendered:

```
<PostsList posts={posts} />
```

So, `PostsList` will now render even if the data is stale.

11. In the running app, press *F5* to refresh the page. Then, leave the browser window and set the focus to a different window. Set your focus back on the browser window and notice that no fetching indicator appears and the blog posts list remains intact.
12. Repeat the previous step but this time, observe the **Network** tab in the browser's DevTools. Notice that a second network request is made when the app is refocused:

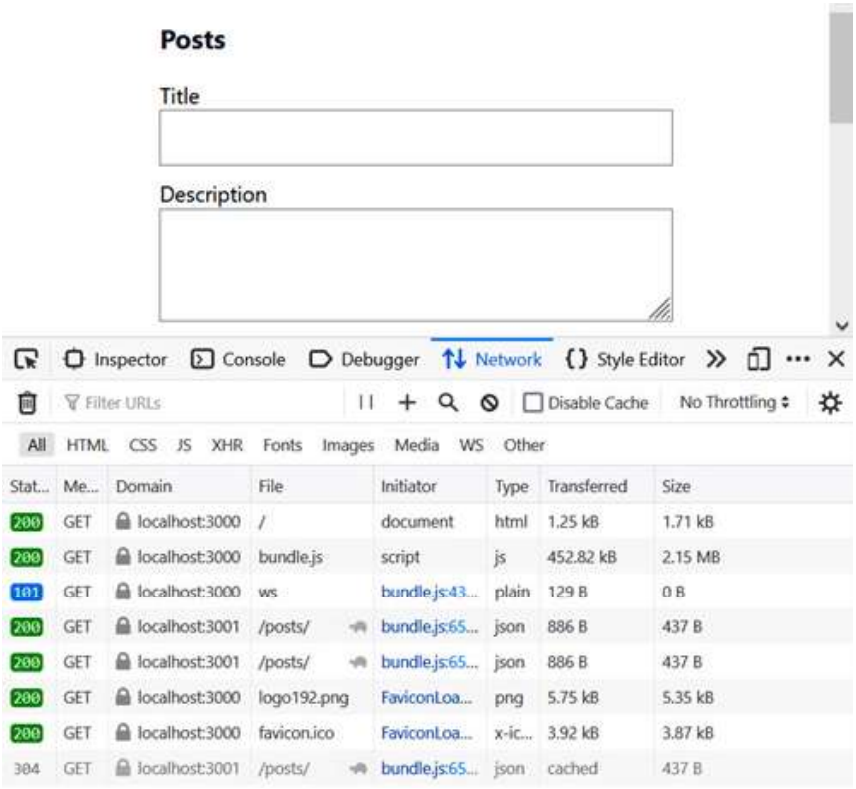

Figure 9.9 – Two API requests for blog posts

So, React Query seamlessly allows the component to render the old data and re-renders it with the new data after it has been fetched.

Next, we will continue to refactor the posts page to use React Query when a new blog post is sent to the API.

Updating data using React Query

React Query can update data using a feature called **mutations** using a `useMutation` hook. Carry out the following steps in `PostsPage.tsx` to change the saving of a new blog post to use a React Query mutation:

1. Update the React Query import as follows:

```
import {
  useQuery,
  useMutation,
  useQueryClient,
} from '@tanstack/react-query';
```

The `useMutation` hook allows us to carry out a mutation. The `useQueryClient` hook will enable us to get the instance of `queryClient` that the component is using and access and update the cached data.

2. Add a call to `useMutation` after the call to `useQuery` as follows:

```
const {
  isLoading,
  data: posts,
  isFetching,
} = useQuery(['postsData'], getPosts);

const { mutate } = useMutation(savePost);
```

We pass `useMutation` the function that performs the REST API HTTP `POST` request. We destructure the `mutate` function from the return value of `useMutation`, which we will use in *step 4* to trigger the mutation.

> **Note**
>
> There are other useful state variables that can be destructured from `useMutation`. An example is `isError`, which indicates whether the `fetch` function errored. See the following link for more information: `https://tanstack.com/query/v4/docs/reference/useMutation`.

3. When the mutation has successfully been completed, we want to update the `posts` cache to contain the new blog post. Make the following highlighted changes to implement this:

```
const queryClient = useQueryClient();
const { mutate } = useMutation(savePost, {
  onSuccess: (savedPost) => {
    queryClient.setQueryData<PostData[]>(
      ['postsData'],
      (oldPosts) => {
        if (oldPosts === undefined) {
          return [savedPost];
        } else {
          return [savedPost, ...oldPosts];
        }
      }
    );
```

```
    },
  });
```

The second argument on `useMutation` allows the mutation to be configured. The `onSuccess` configuration option is a function called when the mutation has been successfully completed.

`useQueryClient` returns the query client that the component is using. This query client has a method called `setQueryData`, which allows the cached data to be updated. `setQueryData` has arguments for the key of the cached data and the new copy of data to be cached.

4. We can trigger the mutation when the new post is saved by calling the destructured `mutate` function in the `onSave` prop on the `NewPostForm` JSX element:

```
<NewPostForm onSave={mutate} />
```

5. Now, we can remove the `handleSave` function because this is now redundant.
6. The imported `NewPostData` type can be removed as well. This type's import statement should now be as follows:

```
import { PostData } from './types';
```

7. In the running app, if you enter and save a new blog post, it will appear in the list as in the previous implementation:

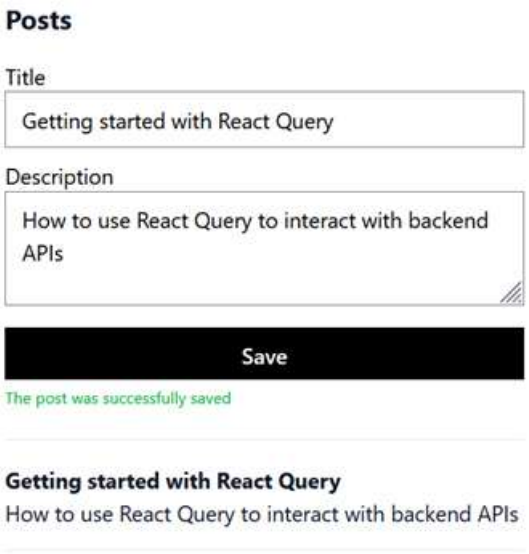

Figure 9.10 – New blog post added to posts list

That completes the refactoring of saving new blog posts to use a React Query mutation. That also completes this section on React Query – here's a recap of the key points:

- React Query is a popular library that manages data from a backend API in a cache, helping to improve performance
- React Query doesn't actually make the HTTP requests – the browser's `fetch` function can be used to do this
- React Query's `QueryClientProvider` component needs to be placed high in the component tree above where backend data is needed
- React Query's `useQuery` hook allows data to be fetched and cached in state
- React Query's `useMutation` hook allows data to be updated

For more information on React Query, visit the library's documentation site: `https://tanstack.com/query`.

Next, we will learn how to integrate React Query into React Router's data-fetching capabilities.

Using React Router with React Query

So far, we have experienced the benefits of both React Router and React Query data fetching. React Router reduces the number of re-renders, while React Query provides a client-side cache of the data. In this section, we will use these libraries together in our app so that it has both these benefits.

Carry out the following steps:

1. Start by opening `App.tsx` and change the loader function on the route definition to the following:

```
const router = createBrowserRouter([
  {
    path: '/',
    element: ...,
    loader: async () => {
      const existingData = queryClient.getQueryData([
        'postsData',
      ]);
      if (existingData) {
        return defer({ posts: existingData });
      }
      return defer({
        posts: queryClient.fetchQuery(
```

```
            ['postsData'],
            getPosts
          )
        });
      }
    }
])
```

Inside the loader, we use React Query's `getQueryData` function on the query client to get the existing data from its cache. If there is cached data, it is returned; otherwise, the data is fetched, deferred, and added to the cache.

2. Open `PostsPage.tsx` and remove the use of React Query's `useQuery` because the React Router loader manages the data loading process now.
3. Remove the `getPosts` function from the `getPosts` import statement because this is used in the React Router loader now.
4. Also, remove the loading indicator because we will revert to using React Suspense in *step 6*.
5. The data will be retrieved using React Router's `useLoaderData` hook again, so uncomment those two lines of code:

```
export function PostsPage() {
  const queryClient = useQueryClient();
  const { mutate } = useMutation( ... );
  const data = useLoaderData();
  assertIsData(data);
  return ...
}
```

6. Also, reinstate the use of `Suspense` and `Await` in the JSX. The JSX should be as follows now:

```
<div className="w-96 max-w-xl mx-auto mt-6">
  <h2 className="text-xl text-slate-900 font-bold">
    Posts
  </h2>
  <NewPostForm onSave={mutate} />
  <Suspense fallback={<div>Fetching ...</div>}>
    <Await resolve={data.posts}>
      {(posts) => {
        assertIsPosts(posts);
```

```
          return <PostsList posts={posts} />;
        }}
      </Await>
    </Suspense>
  </div>
```

7. The running app will appear and display the blog posts just as before, but a second render of `PostsPage` will no longer occur when the app is first loaded. However, after adding a new blog post using the form, it doesn't appear in the list. We will resolve this in the next step.

8. After the new blog post has been saved, we need to cause the route component to re-render in order to get the latest data. We can do this by causing the router to navigate to the page we are already on, as follows:

```
import {
  useLoaderData,
  Await,
  useNavigate
} from 'react-router-dom';
...
export function PostsPage() {
  const navigate = useNavigate();
  const queryClient = useQueryClient();
  const { mutate } = useMutation(savePost, {
    onSuccess: (savedPost) => {
      queryClient.setQueryData<PostData[]>(
        ['postsData'],
        (oldPosts) => {
          if (oldPosts === undefined) {
            return [savedPost];
          } else {
            return [savedPost, ...oldPosts];
          }
        }
      );
      navigate('/');
    },
  });
  ...
}
```

We perform the navigation after the blog post has been saved and added to the cache. This means the route's loader will execute and populate its data from the cache. `PostsPage` will then be rendered with `useLoaderData` returning the up-to-date data.

That completes this final revision of the app and this section on using React Router with React Query. By integrating these two libraries, we get the following key benefits of each library:

- React Router's data loader prevents an unnecessary re-render when data is loaded onto a page
- React Query's cache prevents unnecessary calls to the REST API

The way these two libraries integrate is to get and set data in the React Query cache, in the React Router loader.

Summary

In this chapter, we used the browser's `fetch` function to make HTTP `GET` and `POST` requests. The request's URL is the first argument on the `fetch` function. The second argument on `fetch` allows the request options to be specified, such as the HTTP method and body.

A type assertion function can be used to strongly type the data in the response body of an HTTP request. The function takes in the data having an `unknown` type. The function then carries out checks to validate the type of data and throws an error if it is invalid. If no errors occur, the asserted type for the data is specified in the functions assertion signature.

React's `useEffect` hook can be used to execute a call to fetch data from a backend API and store the data in the state when the component is mounted. A flag can be used inside `useEffect` to ensure the component is still mounted after the HTTP request before the data state is set.

React Query and React Router replace the use of `useEffect` and `useState` in the data-fetching process and simplify our code. React Router's loader function allows data to be fetched and passed into the component route removing an unnecessary re-render. React Query contains a cache that can be used in components to render data optimistically while up-to-date data is being fetched. React Query also contains a `useMutation` hook to enable data to be updated.

In the next chapter, we will cover how to interact with GraphQL APIs.

Questions

Answer the following questions to check what you have learned in this chapter:

1. The following effect attempts to fetch data from a REST API and store it in the state:

```
useEffect(async () => {
  const response = await fetch('https://some-rest-api/');
```

```
    const data = await response.json();
    setData(data);
  }, []);
```

What are the problems with this implementation?

2. The following fetching function returns an array of first names:

```
export async function getFirstNames() {
  const response = await fetch('https://some-
    firstnames/');
  const body = await response.json();
  return body;
}
```

However, the return type of the function is `any`. So, how can we improve the implementation to have a return type of `string[]`?

3. In the `fetch` function argument, what should be specified in the `method` option for it to make an HTTP `PUT` request?

```
fetch(url, {
  method: ???,
  body: JSON.stringify(data),
});
```

4. How do you specify a bearer token in an HTTP `Authorization` header when making an HTTP request to a protected resource using `fetch`?

5. A component uses React Query's `useQuery` to fetch data but the component errors with the following error:

   Uncaught Error: No QueryClient set, use QueryClientProvider to set one

   What do you think the problem is?

6. What state variable can be destructured from React Query's `useMutation` to determine whether the HTTP request has returned an error?

Answers

1. There are two problems with the implementation:

   - `useEffect` doesn't support top-level `async/await`
   - If the component is umounted during the HTTP request, an error will occur when the `data` state is set

Here is an implementation with those issues resolved:

```
useEffect(() => {
  let cancel = false;
  fetch('https://some-rest-api/')
    .then((response) => data.json())
    .then((data) => {
      if (!cancel) {
        setData(data);
      }
    });
  return () => {
    cancel = true;
  };
}, []);
```

2. An `assert` function can be used on the response body object as follows:

```
export async function getFirstNames() {
  const response = await fetch('https://some-
    firstnames/');
  const body = await response.json();
  assertIsFirstNames(body);
  return body;
}

function assertIsFirstNames(
  firstNames: unknown
): asserts firstNames is string[] {
  if (!Array.isArray(firstNames)) {
    throw new Error('firstNames isn't an array');
  }
  if (firstNames.length === 0) {
    return;
  }
  firstNames.forEach((firstName) => {
    if (typeof firstName !== 'string') {
      throw new Error('firstName is not a string');
```

```
    }
  });
}
```

3. The method option should be `'PUT'`:

```
fetch(url, {
  method: 'PUT',
  body: JSON.stringify(data),
});
```

4. The `headers.Authorization` option can be used to specify a bearer token when making an HTTP request to a protected resource using `fetch`:

```
fetch(url, {
  headers: {
    Authorization: 'Bearer some-bearer-token',
    'Content-Type': 'application/json',
  },
});
```

5. The problem is that React Query's `QueryClientProvider` hasn't been placed above the component using `useQuery` in the component tree.

6. The `isError` state variable can be destructured from React Query's `useMutation` to determine whether the HTTP request has returned an error. Alternatively, the `status` state variable can be checked for a value of `'error'`.

10
Interacting with GraphQL APIs

GraphQL APIs are web APIs that have a special language for interacting with them. These APIs are a very popular alternative to REST APIs with React frontends.

In this chapter, we'll first understand the special GraphQL language, executing some basic queries on the GitHub GraphQL API. We will then build a React app that allows users to search for a GitHub repository and star it, experiencing the benefits of GraphQL over REST.

The app will use the browser's `fetch` function with React Query to interact with the GitHub GraphQL API. We will then refactor the implementation of the app to use a specialized GraphQL client called **Apollo Client**.

We'll cover the following topics:

- Understanding the GraphQL syntax
- Getting set up
- Using the React Query with fetch
- Using Apollo Client

Technical requirements

We will use the following technologies in this chapter:

- **Node.js** and **npm**: You can install them at `https://nodejs.org/en/download/`.
- **Visual Studio Code**: You can install it at `https://code.visualstudio.com/`.
- **GitHub**: You'll need a GitHub account. If you haven't got an account, you can sign up at the following link: `https://github.com/join`.
- **GitHub GraphQL API Explorer**: We'll use this tool to play with the syntax of GraphQL queries and mutations. The tool can be found at `https://docs.github.com/en/graphql/overview/explorer`.

All the code snippets in this chapter can be found online at `https://github.com/PacktPublishing/Learn-React-with-TypeScript-2nd-Edition/tree/main/Chapter10`.

Understanding the GraphQL syntax

Like React Query, GraphQL refers to a request to fetch data as a **query**. In the following subsections, we'll learn how to write a basic GraphQL query that returns data from a couple of fields. These fields will have primitive values and so the result will be flat. We'll then learn how to write a more advanced query containing object-based field values that have their own properties. Lastly, we will learn how to make queries more reusable using parameters.

Returning flat data

Carry out the following steps to use the GitHub GraphQL API Explorer to get information about your GitHub user account:

1. Open the following URL in a browser to open the GitHub GraphQL API Explorer: `https://docs.github.com/en/graphql/overview/explorer`.
2. Sign in using the **Sign in with GitHub** button if you aren't signed in already. A GraphQL API Explorer page appears, as follows:

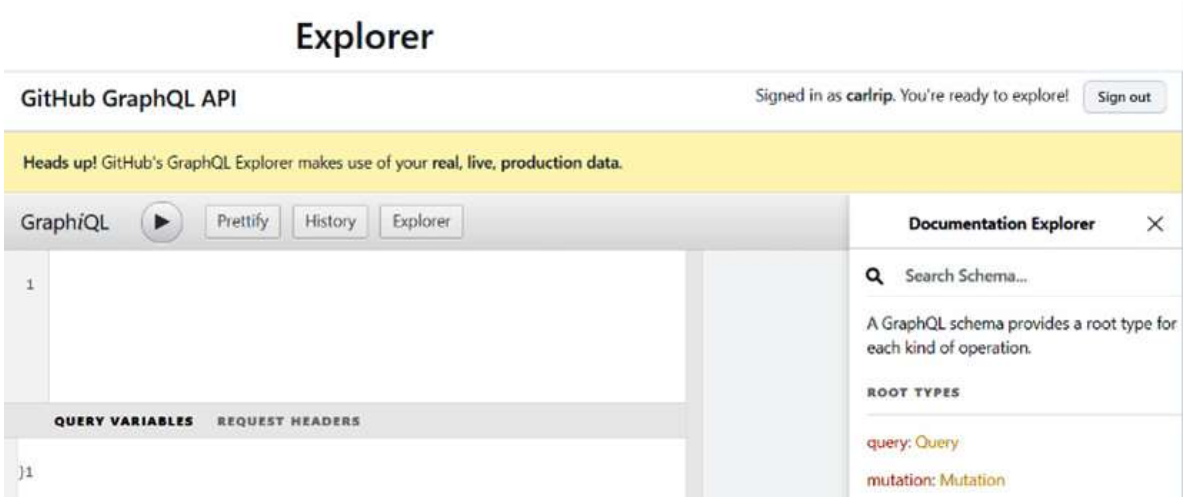

Figure 10.1 – GitHub GraphQL API Explorer

3. In the top-left panel of the GraphQL API Explorer, enter the following query:

```
query {
  viewer {
    name
  }
}
```

The query starts with the `query` keyword to specify that the operation is a query to fetch data (rather than update data). It is worth noting that the `query` keyword is optional because the operation defaults to a query.

After the operation, the data to be returned is specified by specifying the required objects and fields. In our example, we have specified that the `name` field in the `viewer` object is returned.

4. Click the **Execute Query** button, which is the round button containing the black triangle.

   The query result appears to the right of the query as follows:

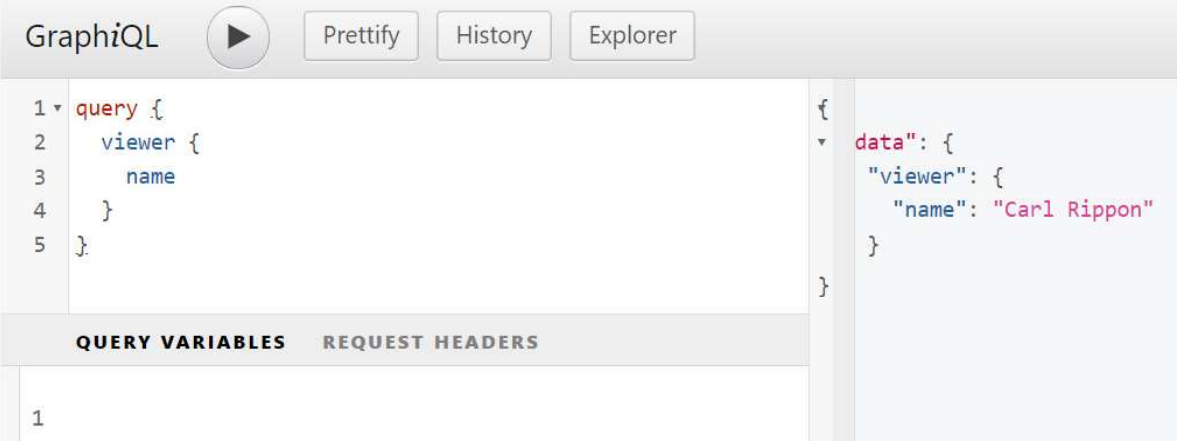

Figure 10.2 – GitHub GraphQL

The data we requested is returned as a JSON object. The `name` field value should be your name stored in your GitHub account.

5. On the right-hand side of the query results is **Documentation Explorer**. Expand this panel if it's not already expanded:

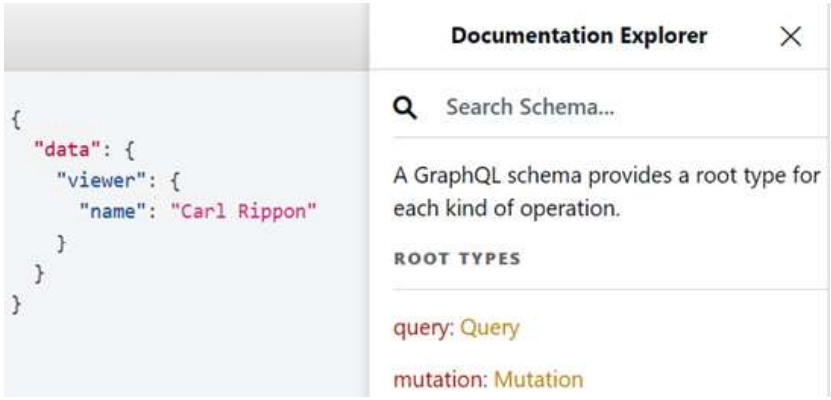

Figure 10.3 – Documentation Explorer

6. Click on the **Query** link in **Documentation Explorer**. All the objects are shown that can be queried, including `viewer`, which is the one we just queried. The object type appears to the right of the object name, and the object description appears underneath.

> **Note**
>
> Like many languages, GraphQL's fields have types – there are built-in types such as `String`, `Int`, and `Boolean`, as well as the ability to create custom types. See the following link for more information: `https://graphql.org/learn/schema/#type-language`.

7. Scroll down to the `viewer` object in **Documentation Explorer** (it should be right at the bottom):

Figure 10.4 – viewer object in Documentation Explorer

8. Click on the `User` type next to the `viewer` object name in **Documentation Explorer**. All the fields available in the `User` type are listed:

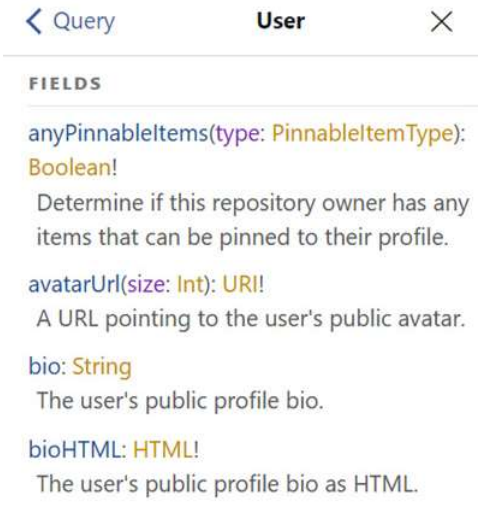

Figure 10.5 – Fields in the User type

9. Let's add `avatarUrl` to our query, as this is an additional field available to us:

```
query {
  viewer {
    name
    avatarUrl
  }
}
```

We simply add the `avatarUrl` field inside the `viewer` object with a carriage return between the `name` and `avatarUrl` fields.

10. The `avatarUrl` field is added to the JSON result if you execute the query. This should be a path to an image of you:

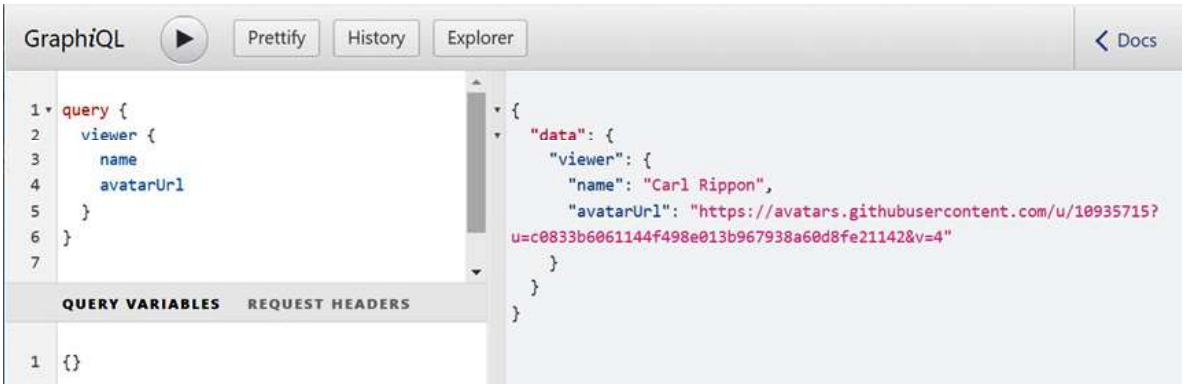

Figure 10.6 – Updated query result with avatarUrl

That completes our first graphQL query.

11. We are already seeing how flexible GraphQL is with being able to specify which fields we want to be returned in the response. Next, we'll create another query that returns a hierarchical structure.

Returning hierarchical data

We will make a more complex query now by returning an object-based field rather than just fields with primitive values. This will mean the result will have a hierarchical structure rather than being flat. We'll query for a GitHub repository, returning its name, description, and the number of stars it has. So, carry out the following steps:

1. Start by entering the following query into the query panel of the GitHub GraphQL API explorer:

```
query {
  repository (owner:"facebook", name:"react") {
```

```
      id
      name
      description
    }
  }
```

The query asks for the `id`, `name`, and `description` fields in the `repository` object. After the `repository` object is specified, two parameters for the `owner` and `name` of the repository are specified.

2. Let's now request the number of stars against the repository. To do this, add the `totalCount` field within the `stargazers` object as follows:

```
query {
  repository (owner:"facebook", name:"react") {
    id
    name
    description
    stargazers {
      totalCount
    }
  }
}
```

3. If you execute the query, the result will appear like the following screenshot:

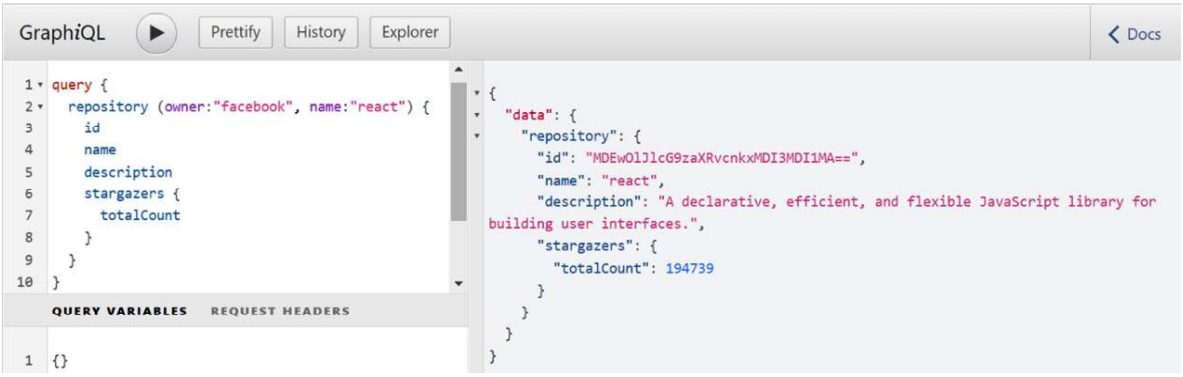

Figure 10.7 – Query for a specific repository

That completes our second GraphQL query.

So, GraphQL allows us to make a single web request for different bits of data, returning just the fields that we require. Doing a similar thing with a REST API would probably require multiple requests and we'd get a lot more data than we need to return. It is in these types of queries where GraphQL shines over REST.

Next, we will learn how to allow query parameter values to vary.

Specifying query parameters

The query we have just made already has parameters for the repository name and owner. However, the `owner` parameter is hardcoded to have a value of `"facebook"`, and the `name` parameter to have a value of `"react"`.

You may have noticed the **QUERY VARIABLES** panel under the query panel. This allows query parameter values to be specified. The query parameters then reference variable names instead of hardcoded values.

Carry out the following steps to adjust the repository query so that the query parameters can vary:

1. Add the following query variables in the **QUERY VARIABLES** panel:

```
{
  "owner": "facebook",
  "name": "react"
}
```

   As you can see, the variables are specified using JSON syntax. We have named the variable for the repository owner, `owner`, and the variable for the repository name, `name`.

2. Update the query to reference the query variables as follows:

```
query ($owner: String!, $name: String!) {
  repository (owner:$owner, name:$name) {
    ...
  }
}
```

   The query parameters are specified in parentheses after the `query` keyword. The parameter names must be prefixed with a dollar sign (`$`). The type for each parameter is specified after a colon (`:`) – both parameters are `String` in our case. The exclamation mark (`!`) after the type means it is a required query parameter. The parameters can then be referenced within the query, which, in our case, is where we request the repository object.

3. If we execute the query, the JSON result will be the same as the query with the hardcoded repository owner and name criteria.

4. Now, change the variable values to target a different repository and rerun the query. The JSON result will contain the same fields but with values for the repository requested. The following is the query and result for the TypeScript repository:

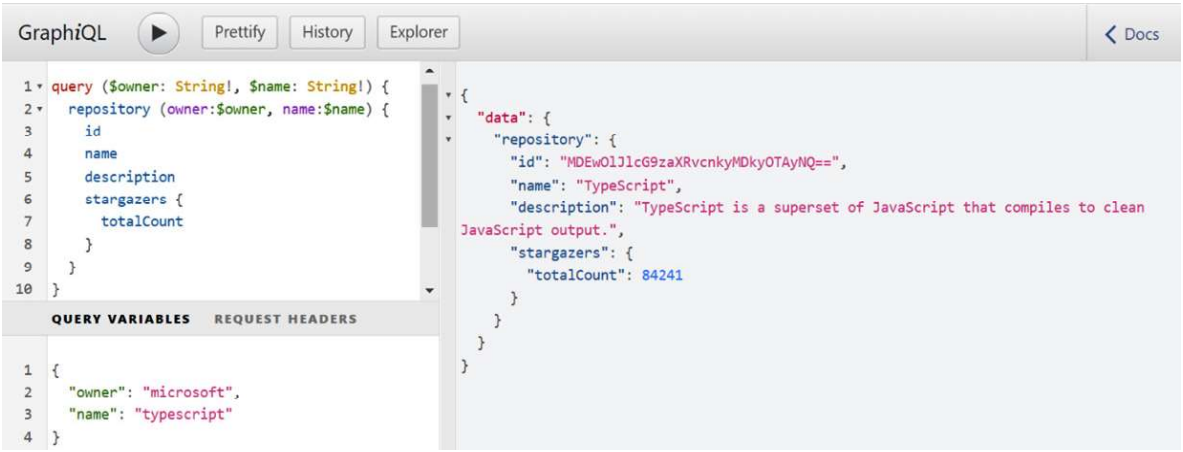

Figure 10.8 – Query with parameters for the TypeScript repository

5. We are now getting comfortable with reading data from a GraphQL server. Next, we'll learn how to request changes to GraphQL data.

GraphQL mutations

A change to data in GraphQL is referred to as a **mutation**. Starring a repository is a change to the underlying data, so we can class this as an example of a mutation.

Carry out the following steps to create a mutation that stars a GitHub repository:

1. In order to star a repository, we need the repository ID. So, copy the repository ID of the last query result into your clipboard. The following is the ID of the TypeScript repository:

```
"MDEwOlJlcG9zaXRvcnkyMDkyOTAyNQ=="
```

2. Replace the query variables with a variable for the repository ID we want to star:

```
{
  "repoId": "MDEwOlJlcG9zaXRvcnkyMDkyOTAyNQ=="
}
```

3. Replace the content in the query panel with the following mutation:

```
mutation ($repoId: ID!) {
  addStar(input: { starrableId: $repoId }) {
    starrable {
```

```
        stargazers {
          totalCount
        }
      }
    }
  }
```

Let's break down this code:

- We prefix a mutation with the `mutation` keyword.
- We put parameters to be passed into the mutation after the `mutation` keyword in parentheses. In our case, we have a single parameter for the repository ID we want to star.
- `addStar` is the mutation function we call, which has a parameter called `input` that we need to pass.
- `input` is actually an object that has a field called `starrableId` that we need to include. The value of this is the repository ID we want to star, so we set it to our `$repoId` repository ID variable.
- After the mutation parameters, we specify what we want to return in the response. In our case, we want to return the number of stars on the repository.

4. If we execute the mutation, the star will be added to the repository, and the new total number of stars will be returned:

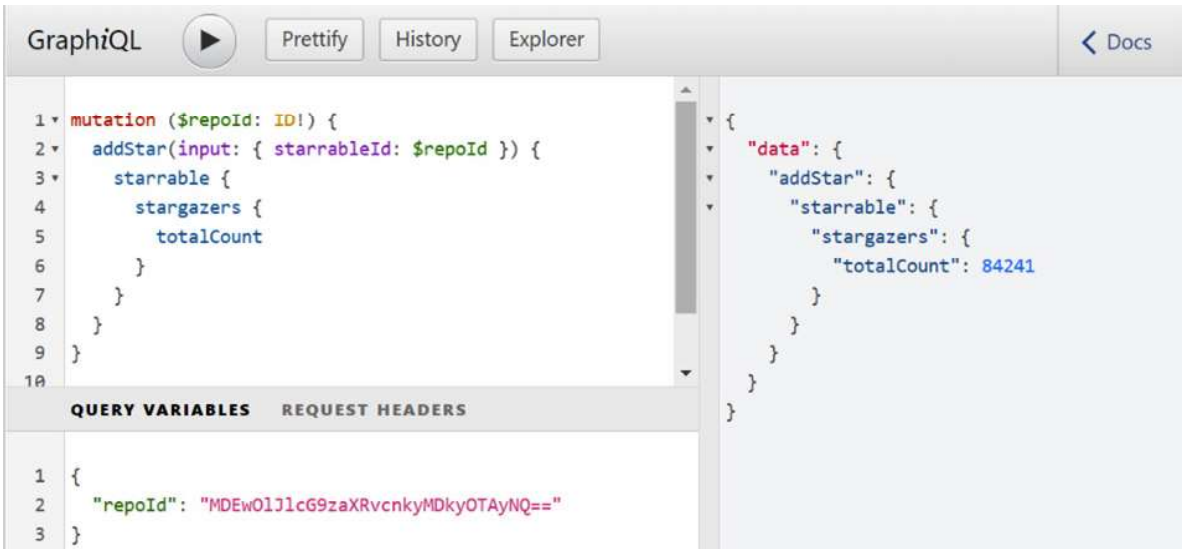

Figure 10.9 – Mutation to star the repository

That completes this section on getting comfortable with the GraphQL syntax. To recap, here are some key points:

- A GraphQL query fetches data, and a mutation changes data. These operations are specified with the `query` and `mutation` keywords, respectively.
- The data required in the response can be specified in the query/mutation, which helps the backend interactions be efficient.
- Query parameter variables can be specified to allow a query/mutation to be reusable.

Next, we will set up a React project that will eventually interact with the GitHub GraphQL API.

Setting up the project

In this section, we will start by creating the project for the app we will build. We will build a React app that allows users to search for a GitHub repository and star it. It will use the GitHub GraphQL API, so we will generate a **personal access token** (**PAT**) for this and store it in an environment variable.

Creating the project

We will develop the app using Visual Studio Code and a new Create React App-based project setup. We've previously covered this several times, so we will not cover the steps in this chapter – instead, see *Chapter 3*, *Setting Up React and TypeScript*.

We will style the app with Tailwind CSS. We have previously covered how to install and configure Tailwind in a Create React App in *Chapter 5*, *Approaches to Styling Frontends*. So, after you have created the React and TypeScript project, install and configure Tailwind.

We will use React Hook Form to implement the form that creates blog posts, and the `@tailwindcss/forms` plugin to style the form. So, install the `@tailwindcss/forms` plugin and React Hook Form (see *Chapter 7*, *Working with Forms,* if you can't remember how to do this).

Now that the project is set up, next, we will gain access to the GitHub GraphQL API.

Creating a PAT for the GitHub GraphQL API

The GitHub GraphQL API is protected by a PAT, which is a string of characters and is a common mechanism for protecting web APIs. Carry out the following steps to generate a PAT:

1. In a browser, go to GitHub: `https://github.com/`.
2. Sign in to your GitHub account if you aren't already signed in. You can create a GitHub account if you haven't got one by using the **Sign Up** button.
3. Now, open the menu under your avatar and click **Settings**.

4. Next, access the **Developer Settings** option at the bottom of the left-hand bar.
5. Go to the **Personal access tokens** page on the left-hand bar.
6. Click the **Generate new token** button to start creating the PAT. You will likely be prompted to input your password after clicking the button.
7. Before the token is generated, you will be asked to specify the scopes. Enter a token description, tick the repo and user scopes, and then click the **Generate token** button.

The token is then generated and displayed on the page. Take a copy of this because we'll need this in the next section when building our app.

Creating environment variables

Before writing code that interacts with the GitHub GraphQL API, we will create environment variables for the API URL and the PAT:

1. Let's start by creating an environment file to store the URL for the GitHub GraphQL API. Create a file called `.env` in the root of the project containing this variable, as follows:

   ```
   REACT_APP_GITHUB_URL = https://api.github.com/graphql
   ```

   This environment variable is injected into the code at build time and can be accessed by code using `process.env.REACT_APP_GITHUB_URL`. Environment variables in Create React App projects must be prefixed with `React_APP_`.

   For more information on environment variables, see the following link: `https://create-react-app.dev/docs/adding-custom-environment-variables/`.

2. Now, we will create a second environment variable for the GitHub PAT token. However, we don't want to commit this file to source code control, so place it in a file called `.env.local` at the root of the project:

   ```
   REACT_APP_GITHUB_PAT = your-token
   ```

   `.env.local` is in the `.gitignore` file, so this file won't get committed to source code control, reducing the risk of your PAT getting stolen. Replace `your-token` with your PAT token in the preceding code snippet.

That completes the creation of the environment variables.

In the next section, we will start to build the app that will interact with the GitHub GraphQL API.

Using React Query with fetch

In this section, we will build an app containing a form that allows users to search and star GitHub repositories. The app will also have a header containing our name from GitHub. We will use the browser `fetch` function with React Query to interact with the GitHub GraphQL API. Let's get started.

Creating the header

We will create the header for the app, which will contain our GitHub name. We will create a `Header` component containing this, which will be referenced from the `App` component. The `Header` component will use React Query to execute a function that gets our GitHub name calling the GitHub GraphQL API.

Creating a function to get viewer information

Carry out the following steps to create a function that makes a request to the GitHub GraphQL API to get details about the logged-in viewer:

1. We will start by creating a folder for the API calls. Create an `api` folder in the `src` folder.
2. Now, we will create a type that the function will use. Create a file called `types.ts` in the `src/api` folder with the following content:

```
export type ViewerData = {
  name: string;
  avatarUrl: string;
};
```

   This type represents the logged-in viewer.

3. Create a file called `getViewer.ts` in the `api` folder that will contain the function we need to implement. Add an import statement for the type we just created:

```
import { ViewerData } from './types';
```

4. Under the import statement, add a constant assigned to the following GraphQL query:

```
export const GET_VIEWER_QUERY = `
  query {
    viewer {
      name
      avatarUrl
    }
  }
`;
```

This is the same query we used earlier in the chapter to get the current viewer's name and avatar URL.

5. Add the following type, which represents the response from the GraphQL API call within this file:

```
type GetViewerResponse = {
  data: {
    viewer: ViewerData;
  };
};
```

6. Start to implement the function as follows:

```
export async function getViewer() {
  const response = await fetch(
    process.env.REACT_APP_GITHUB_URL!
  );
}
```

We use the `fetch` function to make the request to the GraphQL API. We have used the `REACT_APP_GITHUB_URL` environment variable to specify the GraphQL API URL. Environment variable values can be `undefined`, but we know this isn't the case, so we have added a not null assertion (`!`) after it.

7. Specify the GraphQL query in the request body as follows:

```
export async function getViewer() {
  const response = await fetch(
    process.env.REACT_APP_GITHUB_URL!,
    {
      body: JSON.stringify({
        query: GET_VIEWER_QUERY
      }),
      headers: {
        'Content-Type': 'application/json'
      }
    }
  );
}
```

GraphQL queries are specified in the request body in an object structure with a `query` property containing the GraphQL query string, which is `GET_VIEWER_QUERY` in our case. We have also specified that the request is in JSON format using the `Content-Type` HTTP header.

8. The HTTP `POST` method must be used for GraphQL API requests. So, let's specify this in the request:

```
export async function getViewer() {
  const response = await fetch(
    process.env.REACT_APP_GITHUB_URL!,
    {
      method: 'POST',
      body: ...,
      headers: ...,
    }
  );
}
```

9. A PAT protects the GitHub GraphQL API, so let's add this to the request:

```
export async function getViewer() {
  const response = await fetch(
    process.env.REACT_APP_GITHUB_URL!,
    {
      ...,
      headers: {
        'Content-Type': 'application/json',
        Authorization: `bearer ${process.env.REACT_APP_
          GITHUB_PAT}`
      },
    }
  );
}
```

10. The last steps in the function are to get the JSON response body and type it appropriately before returning it:

```
export async function getViewer() {
  const response = await fetch(
    ...
  );
```

```
    const body = (await response.json()) as unknown;
    assertIsGetViewerResponse(body);
    return body.data;
  }
```

11. We narrow the type of `body` using a type assertion function called `assertIsGetViewerResponse`. The implementation of this function is lengthy and follows the same pattern as the ones we implemented in *Chapter 9*, *Interacting with RESTful APIs*, so we won't list it in this step, but see `https://github.com/PacktPublishing/Learn-React-with-TypeScript-2nd-Edition/blob/main/Chapter10/Using-React-Query/src/api/getViewer.ts` for the implementation of this function.

    One difference is that the function's parameter is of the `any` type rather than `unknown`. This is due to a known TypeScript issue of not being able to narrow the `unknown` type when it is an object. For more information on this, see the following link: `https://github.com/microsoft/TypeScript/issues/25720`. Using the `any` type is fine in this case – the `assertIsGetViewerResponse` function will work perfectly fine.

That completes the implementation of a function that gets the details of the logged-in GitHub viewer.

Next, we will create the component for the header.

Creating the header component

We will create a component for the app header, which will call the `getViewer` function we just implemented and show the viewer's name and avatar:

1. We will use React Query to call `getViewer` and manage the data it returns. So, let's start by installing this package by running the following command in a terminal:

```
npm i @tanstack/react-query
```

2. Create a file for the component called `Header.tsx` in the `src` folder.
3. Add the following import statements in `Header.tsx` to import React Query's `useQuery` hook and our `getViewer` function:

```
import { useQuery } from '@tanstack/react-query';
import { getViewer } from './api/getViewer';
```

4. Start to implement the `Header` component as follows:

```
export function Header() {
  const { isLoading, data } = useQuery(['viewer'],
    getViewer);
}
```

We use the `useQuery` hook to call `getViewer`. The data returned from `getViewer` will be in the destructured `data` variable. We have also destructured an `isLoading` variable to implement a loading indicator in the next step.

5. Add a loading indicator as follows:

```
export function Header() {
  const { isLoading, data } = useQuery(['viewer'],
    getViewer);
  if (isLoading || data === undefined) {
    return <div>...</div>;
  }
}
```

6. Finish the component implementation with the following JSX:

```
export function Header() {
  ...
  return (
    <header className="flex flex-col items-center text-
      slate-50 bg-slate-900 h-40 p-5">
      <img
        src={data.viewer.avatarUrl}
        alt="Viewer"
        className="rounded-full w-16 h-16"
      />
      <div>{data.viewer.name}</div>
      <h1 className="text-xl font-bold">GitHub Search</
        h1>
    </header>
  );
}
```

A `header` element with a very dark gray background is rendered. The `header` element contains the viewer's avatar, name, and a heading of **GitHub Search**, all horizontally centered.

That completes the implementation of the header. Next, we will add it to the component tree.

Adding the Header component to the app

Carry out the following steps to add the `Header` component to the `App` component:

1. Open `App.tsx` and remove all the existing content.
2. Add import statements for React Query's provider component and client as well as our `Header` component:

```
import {
  QueryClient,
  QueryClientProvider,
} from '@tanstack/react-query';
import { Header } from './Header';
```

3. Implement the component by wrapping React Query's provider component around the `Header` component:

```
const queryClient = new QueryClient();

function App() {
  return (
    <QueryClientProvider client={queryClient}>
      <Header />
    </QueryClientProvider>
  );
}

export default App;
```

4. Now, let's try the app by running `npm start` in the terminal. A header containing your avatar and name should appear:

Figure 10.10 – Header containing the viewer's avatar and name

That completes the app header. Next, we will start to implement the main part of the app.

Creating the repository page

The main part of the app will be a page that allows a user to search for a GitHub repository and star it. The page component will be called `RepoPage` and will reference three other components, as follows:

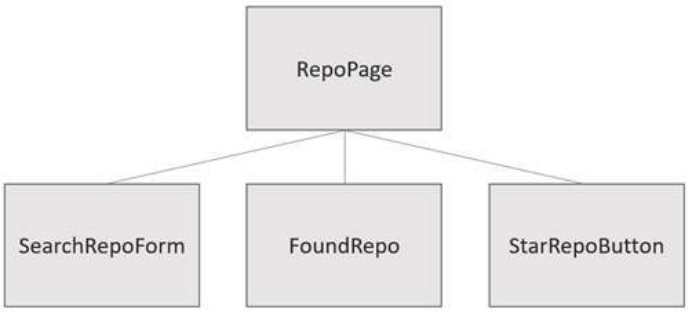

Figure 10.11 – Repository page component structure

Here's an explanation of the components:

- The form that allows users to enter their search criteria will be contained in a `SearchRepoForm` component
- The `FoundRepo` component will render the matched repository after a search
- The `StarRepoButton` component will render the button that the user can click to star a repository
- The `RepoPage` component will use React Query to manage calls to the GitHub GraphQL API and store the returned data

Next, we will make a start on the repository page by implementing a function to do the repository search.

Creating the search function

We will start by implementing the function that calls the GitHub GraphQL API to find a repository. Carry out the following steps:

1. We will start by creating a couple of types that the function will use. Open `src/api/types.ts` and add the following types:

```
export type SearchCriteria = {
  org: string,
  repo: string,
};
export type RepoData = {
  repository: {
```

```
    id: string,
    name: string,
    description: string,
    viewerHasStarred: boolean,
    stargazers: {
      totalCount: number,
    },
  },
};
```

The `SearchCriteria` type represents the information we need in the GraphQL query parameters to find the GitHub repository. The `RepoData` type represents the data returned from the repository search.

2. Create a file for the function called `getRepo.ts` in the `src/api` folder.
3. Open `getRepo.ts` and start by importing the types just created:

```
import { RepoData, SearchCriteria } from './types';
```

4. Add a constant for the following GraphQL query:

```
export const GET_REPO = `
  query GetRepo($org: String!, $repo: String!) {
    repository(owner: $org, name: $repo) {
      id
      name
      description
      viewerHasStarred
      stargazers {
        totalCount
      }
    }
  }
`;
```

This is the same query we created earlier in the chapter in the GitHub GraphQL API explorer.

5. Add the following type beneath the constant:

```
type GetRepoResponse = {
  data: RepoData;
};
```

The `GetRepoResponse` type represents the data returned from the GraphQL query – it references the `RepoData` type we created in *step 1*.

6. Implement the function as follows:

```
export async function getRepo(searchCriteria:
SearchCriteria) {
  const response = await fetch(process.env.REACT_APP_
    GITHUB_URL!, {
    method: 'POST',
    body: JSON.stringify({
      query: GET_REPO,
      variables: {
        org: searchCriteria.org,
        repo: searchCriteria.repo,
      },
    }),
    headers: {
      'Content-Type': 'application/json',
      Authorization: `bearer ${process.env.REACT_APP_
        GITHUB_PAT}`,
    },
  });
  const body = (await response.json()) as unknown;
  assertIsGetRepoResponse(body);
  return body.data;
}
```

This follows the same pattern as the function we created earlier to get the viewer's information. One difference is that we have specified GraphQL query `org` and `repo` parameters, which are set to the properties in the `searchCriteria` function parameter.

7. The `assertIsGetRepoResponse` type assertion function follows the same pattern as previous type assertion functions. The implementation is lengthy, so it isn't listed here. You can find the implementation here: `https://github.com/PacktPublishing/Learn-React-with-TypeScript-2nd-Edition/blob/main/Chapter10/Using-React-Query/src/api/getRepo.ts`.

That completes the implementation of a function that finds the GitHub repository.

Next, we will create the component for the repository search form.

Creating the search form component

We will implement a form component that allows the user to search for a repository. The form will contain fields for the organization and repository name. The component won't call the GitHub GraphQL API when the form is submitted; instead, it will pass the submitted search criteria back to a page component to do this.

We will use React Hook Form for implementation, which should already be installed. The pattern for the implementation is very similar to previous forms we have implemented, so the steps to do this implementation aren't listed in detail here. The implemention for the `SearchRepoForm` component can be copied from the book's GitHub repository as follows:

1. Create a new folder called `repoPage` in the `src` folder and then create a new file called `SearchRepoForm.tsx` in this folder.
2. Open `SearchRepoForm.tsx` and copy and paste the contents into it from `https://github.com/PacktPublishing/Learn-React-with-TypeScript-2nd-Edition/blob/main/Chapter10/Using-React-Query/src/repoPage/SearchRepoForm.tsx`.

The implementation for the `SearchRepoForm` component is now in place in our project.

Next, we will implement a component that renders the found repository.

Creating the FoundRepo component

The `FoundRepo` component will display the repository name, description, and number of stars. Carry out the following steps to implement this component:

1. Create a file in the `src/repoPage` folder called `FoundRepo.tsx`.
2. Start the implementation by adding the following type for the component props:

```
type Props = {
  name: string;
  description: string;
  stars: number;
};
```

   So, the repository name, description, and the number of stars will be passed into the component.

3. Add the following component implementation:

```
export function FoundRepo({ name, description, stars }:
Props) {
  return (
    <div className="py-4">
```

```
        <div className="flex flex-row items-center justify-
          between mb-2">
          <h2 className="text-xl font-bold">{name}</h2>
          <div className="px-4 py-2 rounded-xl text-
            gray-800 bg-gray-200 font-semibold text-sm flex
            align-center w-max">
            {stars} Stars
          </div>
        </div>
        <p>{description}</p>
      </div>
    );
  }
```

The repository name is rendered as a bold heading. The number of stars is rendered in a gray rounded background to the right of the repository name. The description is rendered underneath the name.

That completes the implementation of the found repository component.

Next, we will implement the function that calls the GitHub GraphQL API to star a repository.

Creating a function to star a repository

We will use the same GraphQL mutation we used earlier in the chapter to star the GitHub repository. The pattern used in the function will be similar to the `getViewer` function we created earlier. Carry out the following steps:

1. Create a file called `starRepo.ts` in the `src/api` folder, with the following GraphQL mutation:

```
export const STAR_REPO = `
  mutation ($repoId: ID!) {
    addStar(input: { starrableId: $repoId }) {
      starrable {
        stargazers {
          totalCount
        }
      }
    }
  }
`;
```

This is the same mutation we created earlier in the chapter in the GitHub GraphQL API explorer.

2. Add the function implementation as follows:

```
export async function starRepo(repoId: string) {
  const response = await fetch(process.env.REACT_APP_
    GITHUB_URL!, {
    method: 'POST',
    body: JSON.stringify({
      query: STAR_REPO,
      variables: {
        repoId,
      },
    }),
    headers: {
      'Content-Type': 'application/json',
      Authorization: `bearer ${process.env.REACT_APP_
        GITHUB_PAT}`,
    },
  });
  await response.json();
}
```

This follows the same pattern as the other functions that call the GitHub GraphQL API.

That completes the function that calls the GitHub GraphQL API to star a repository.

Next, we will implement the component for the star button.

Creating the star button

The star button is a regular button styled to be black with white text.

Create a file called `StarRepoButton.tsx` in the `src/repoPage` folder and add the following implementation to it:

```
type Props = {
  onClick: () => void;
};
export function StarRepoButton({ onClick }: Props) {
  return (
    <button
```

```
      type="button"
      className="mt-2 h-10 px-6 font-semibold bg-black text-
        white"
      onClick={onClick}
    >
      Star
    </button>
  );
}
```

That completes the implementation of the star button.

Next, we will create the main page component for the app.

Creating the repository page

The repository page component will reference the `SearchRepoForm`, `FoundRepo`, and `StarRepoButton` components we just created. This component will also call the `getRepo` and `starRepo` functions we created using React Query. To do this, carry out the following steps:

1. Create a file called `RepoPage.tsx` in the `src/repoPage` folder with the following import statements:

```
import { useState } from 'react';
import {
  useQuery,
  useMutation,
  useQueryClient,
} from '@tanstack/react-query';
import { getRepo } from '../api/getRepo';
import { starRepo } from '../api/starRepo';
import { RepoData, SearchCriteria } from '../api/types';
import { SearchRepoForm } from './SearchRepoForm';
import { FoundRepo } from './FoundRepo';
import { StarRepoButton } from './StarRepoButton';
```

   We have imported the components and the data functions we created earlier, along with React Query's hooks and client. We have also imported React's state hook because we need to store a piece of state outside React Query.

2. Start the component implementation as follows:

```
export function RepoPage() {
  const [searchCriteria, setSearchCriteria] = useState<
    SearchCriteria | undefined
  >();
}
```

We store the search criteria in state so that we can feed it into `useQuery` in the next step. We will set this state when the search repository form is submitted in *step 6*.

3. Next, call the `useQuery` hook to get the repository for the given search criteria as follows:

```
export function RepoPage() {
  const [searchCriteria, setSearchCriteria] = ...
  const { data } = useQuery(
    ['repo', searchCriteria],
    () => getRepo(searchCriteria as SearchCriteria),
    {
      enabled: searchCriteria !== undefined,
    }
  );
}
```

We don't want the query to execute when the component is mounted, so we use the `enabled` option to only run the query when `searchCriteria` is set, which will be when the search repository form is submitted.

We use the search criteria in the query key as well and pass it to the `getRepo` function. We use a type assertion on the `getRepo` argument to remove `undefined` from it. This is safe because we know it can't be `undefined` when `getRepo` is called because of the `enabled` option expression.

4. Define the star mutation as follows:

```
export function RepoPage() {
  const [searchCriteria, setSearchCriteria] = ...
  const { data } = useQuery(...);
  const queryClient = useQueryClient();
  const { mutate } = useMutation(starRepo, {
    onSuccess: () => {
      queryClient.setQueryData<RepoData>(
```

```
          ['repo', searchCriteria],
          (repo) => {
            if (repo === undefined) {
              return undefined;
            }
            return {
              ...repo,
              viewerHasStarred: true,
            };
          }
        );
      }
    });
}
```

The mutation calls the `getRepo` function we created earlier. We use the mutation's `onSuccess` option to update the React Query's cached repository data with the `viewerHasStarred` property set to `true`.

5. Return the following JSX from the component:

```
export function RepoPage() {
  ...
  return (
    <main className="max-w-xs ml-auto mr-auto">
      <SearchRepoForm onSearch={handleSearch} />
      {data && (
        <>
          <FoundRepo
            name={data.repository.name}
            description={data.repository.description}
            stars={data.repository.stargazers.totalCount}
          />
          {!data.repository.viewerHasStarred && (
            <StarRepoButton onClick={handleStarClick} />
          )}
        </>
      )}
```

```
      </main>
    );
  }
```

The component is wrapped in a `main` element, which centers its content. The repository search form is placed inside the `main` element. The found repository is rendered (if there is a found repository) along with a star button if the repository hasn't already been starred.

We will implement the `handleSearch` and `handleStarClick` handlers in the following steps.

6. Create the `handleSearch` handler as follows:

```
export function RepoPage() {
  ...
  function handleSearch(search: SearchCriteria) {
    setSearchCriteria(search);
  }
  return ...
}
```

The handler sets the `searchCriteria` state, which triggers a re-render and the `useQuery` hook to call `getRepo` with the search criteria.

7. Create the `handleStarClick` handler as follows:

```
export function RepoPage() {
  ...
  function handleStarClick() {
    if (data) {
      mutate(data.repository.id);
    }
  }
  return ...
}
```

The handler calls the mutation with the found repository's ID, which will call the `starRepo` function.

This completes the implementation of the repository page component.

8. Open `App.tsx` and add the `RepoPage` component we just created under the app header:

```
...
import { RepoPage } from './repoPage/RepoPage';
...
function App() {
  return (
    <QueryClientProvider client={queryClient}>
      <Header />
      <RepoPage />
    </QueryClientProvider>
  );
}
```

9. Now, let's try the app by running `npm start` in the terminal. The repository search form should appear, as follows:

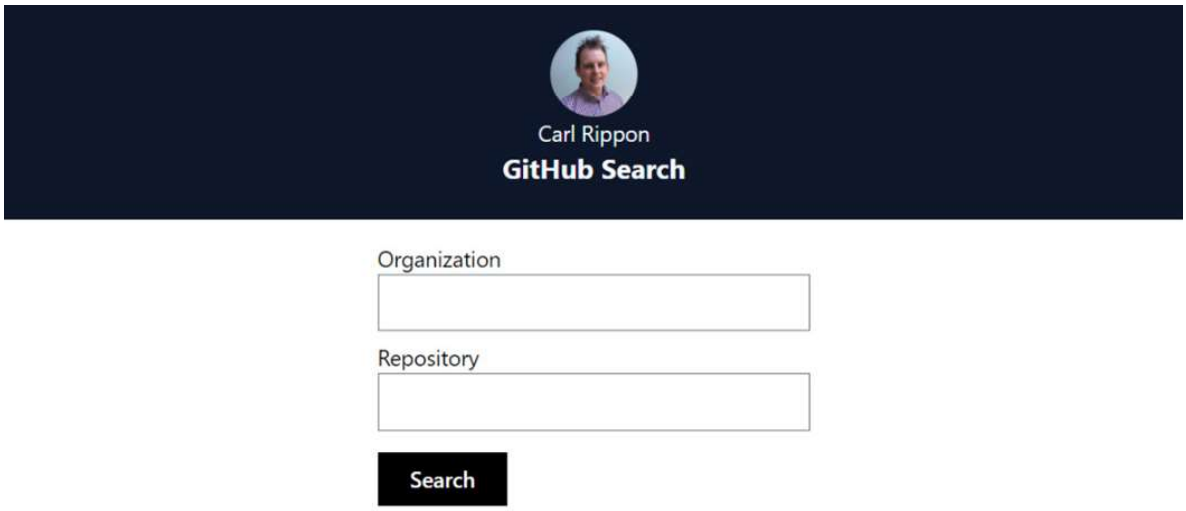

Figure 10.12 – Repository search form

10. Enter a GitHub organization and repository you haven't starred and press **Search**. The found repository will appear with a **Star** button:

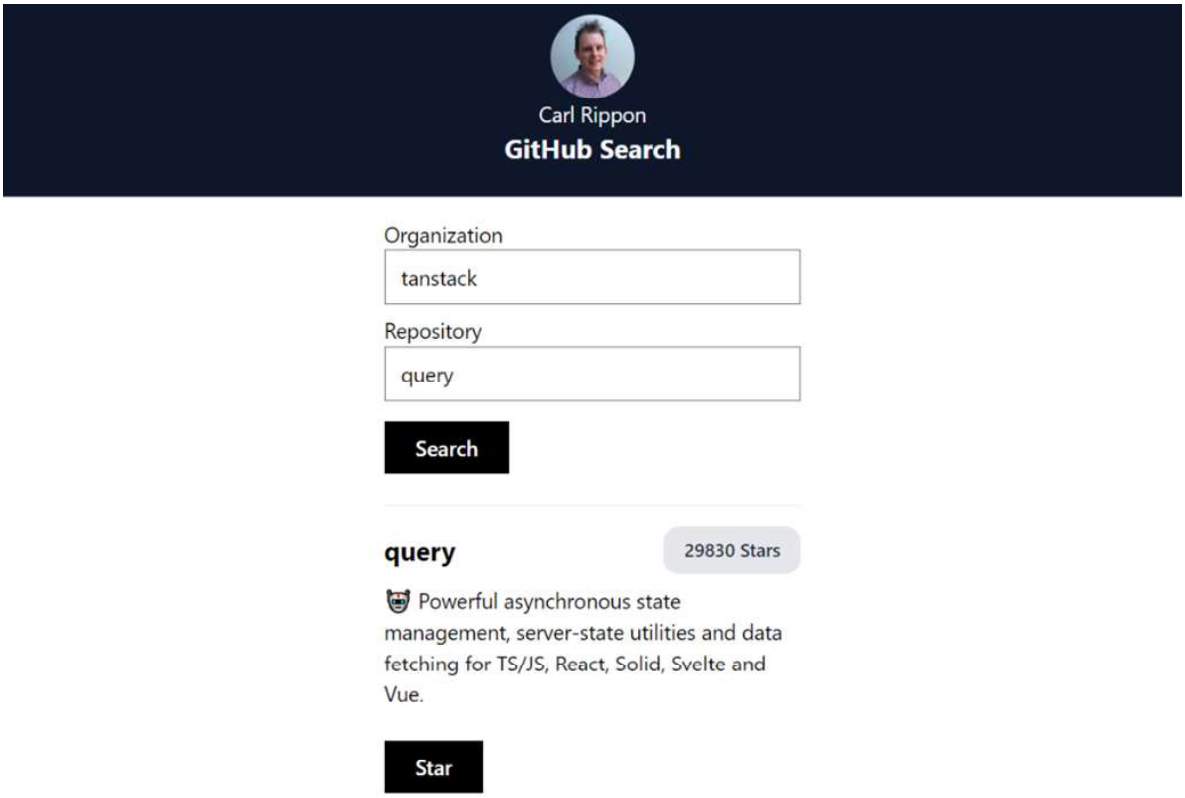

Figure 10.13 – Found repository with a Star button

11. Click the **Star** button to star the repository. The **Star** button will then disappear.
12. Stop the app from running before continuing by pressing *Ctrl* + *C* in the terminal.

That completes the first iteration of the app. Here's a recap of the key points of using `fetch` and React Query to interact with a GraphQL API:

- The `fetch` function can call a GraphQL API by putting the query or mutation in the request body and using the HTTP `POST` method.
- React Query can execute the function containing `fetch` and manage the response data.
- The `enabled` option on `useQuery` and `useMutation` can execute the function containing `fetch` when the user interacts with the app. We used this feature to execute a query when the repository search form was submitted.

In the next section, we will refactor the code to use a specialized GraphQL client.

Using Apollo Client

In this section, we will learn about Apollo Client and use it within the app we have built, replacing the use of React Query and `fetch`.

Understanding Apollo Client

Apollo Client is a client library for interacting with GraphQL servers. It has query and mutation hooks called `useQuery` and `useMutation`, like React Query. Apollo Client also stores the data in a client cache like React Query and requires a provider component placed above the components requiring GraphQL data.

One thing that Apollo Client does that React Query doesn't is that it interacts with the GraphQL API directly instead of requiring a function to do this.

Installing Apollo Client

Our first job is to install Apollo Client, which we can do by running the following command in a terminal:

```
npm i @apollo/client graphql
```

This library includes TypeScript types, so no additional package is required to be installed.

Refactoring the App component

The first component we are going to refactor is the `App` component. Carry out the following steps:

1. Open `App.tsx` and replace the React Query import with the following Apollo Client import statement:

```
import {
  ApolloClient,
  InMemoryCache,
  ApolloProvider,
} from '@apollo/client';
```

2. Update the `queryClient` variable assignment as follows:

```
const queryClient = new ApolloClient({
  uri: process.env.REACT_APP_GITHUB_URL!,
  cache: new InMemoryCache(),
  headers: {
```

```
    Authorization: `bearer ${process.env.REACT_APP_
GITHUB_PAT}`,
  }
});
```

We are now using Apollo Client. We have specified the URL to the API and the PAT because Apollo Client will directly call the API.

3. The last step is to replace `QueryClientProvider` with `ApolloProvider` in the JSX:

```
<ApolloProvider client={queryClient}>
  <Header />
  <RepoPage />
</ApolloProvider>
```

The `App` component is now using Apollo Client.

Next, we will refactor the `Header` component.

Refactoring the Header component

Now, we will refactor the `Header` component to use Apollo Client. Carry out the following steps:

1. Open `getViewer.ts`. The `getViewer` and `assertIsGetViewerResponse` functions and the `GetViewerResponse` type can be removed because Apollo Client doesn't require these. The `ViewerData` import statement can also be removed.
2. Add the following import statement into `getViewer.tsx`:

```
import { gql } from '@apollo/client';
```

`gql` is a function that we will use in the next step to wrap around the GraphQL query string constant.

3. Add the `gql` function in front of the GraphQL query string constant as follows:

```
export const GET_VIEWER_QUERY = gql`
  query {
    viewer {
      name
      avatarUrl
    }
  }
`;
```

So, `GET_VIEWER_QUERY` is now assigned to a tagged template literal rather than a plain string. We covered tagged template literals in *Chapter 5*, when we used Emotion's `css` prop. The `gql` function converts the query string into a query object that Apollo Client can use.

4. Open `Header.tsx` and update the `useQuery` import statement to come from Apollo Client. Also, import the constant we just exported from `getViewer.ts`. We no longer need to import `getViewer`.

```
import { useQuery } from '@apollo/client';
import { GET_VIEWER_QUERY } from './api/getViewer';
```

5. Now update the `useQuery` hook as follows:

```
const { loading: isLoading, data } = useQuery(
  GET_VIEWER_QUERY
);
```

   Apollo Client's `useQuery` hook takes in a parameter for the query definition object and returns useful state variables similar to React Query. We have aliased the `loading` state variable as `isLoading` so that the rendering of the loading indicator remains unchanged.

   For more information on Apollo Client queries, see the following link: `https://www.apollographql.com/docs/react/data/queries/`.

That completes the `Header` component.

Next, we will refactor the repository page.

Refactoring the repository page

Refactoring the repository page will be a similar process. Carry out the following steps:

1. Open `getRepo.ts` and remove the `getRepo` and `assertIsGetResponse` functions and the `GetRepoReponse` type. Remove the imported `RepoData` and `SearchCriteria` types as well.
2. Import the `gql` function and add it in front of the query string:

```
import { gql } from '@apollo/client';

export const GET_REPO = gql`
  query ...
`;
```

3. Open `starRepo.ts` and remove the `starRepo` function.
4. Import the `gql` function and add it in front of the query string:

```
import { gql } from '@apollo/client';

export const STAR_REPO = gql`
  mutation ...
`;
```

5. Open `RepoPage.tsx` and replace the React Query import statement with an Apollo Client import statement. Also, import the GraphQL query constants we just changed in the previous two steps:

```
import {
  useLazyQuery,
  useMutation,
  useApolloClient,
} from '@apollo/client';
import { GET_REPO } from '../api/getRepo';
import { STAR_REPO } from '../api/starRepo';
```

We will use the `useLazyQuery` hook rather than `useQuery` because we want to trigger the query during form submission rather than when the component mounts.

6. Replace the call to `useQuery` with the following call to `useLazyQuery`:

```
const [getRepo, { data }] = useLazyQuery(GET_REPO);
```

`useLazyQuery` returns a tuple, with the first element being a function that can be called to trigger the query. We have called this trigger function `getRepo`. The second tuple element is an object containing useful state variables, such as the data from the API response, which we have destructured.

For more information on `useLazyQuery`, see the following link: `https://www.apollographql.com/docs/react/data/queries/#manual-execution-with-uselazyquery`.

7. Next, replace the `queryClient` variable assignment and the `useMutation` call with the following:

```
const queryClient = useApolloClient();
const [starRepo] = useMutation(STAR_REPO, {
  onCompleted: () => {
    queryClient.cache.writeQuery({
```

```
        query: GET_REPO,
        data: {
          repository: {
            ...data.repository,
            viewerHasStarred: true,
          },
        },
        variables: searchCriteria,
      });
  },
});
```

The first parameter in Apollo Client's `useMutation` hook is the mutation definition object, which is `STAR_REPO` in our case. The second parameter contains options for the mutation. We have specified the `onCompleted` option, which is a function called after the mutation has been completed. We have used this option to update the data cache to indicate that the viewer has now starred the repository.

For more information on Apollo Client mutations, see the following link: `https://www.apollographql.com/docs/react/data/mutations`.

8. Update the `handleSearch` function to call the `useLazyQuery` trigger function:

```
function handleSearch(search: SearchCriteria) {
  getRepo({
    variables: { ...search },
  });
  setSearchCriteria(search);
}
```

9. Update the `handleStarClick` function to call the `useMutation` trigger function:

```
async function handleStarClick() {
  if (data) {
    starRepo({ variables: { repoId: data.repository.id }
});
  }
}
```

That completes the refactoring of the repository page.

10. Now, try the app by running `npm start` in the terminal. Try searching for a repository and starring it – it should behave as it previously did.

That completes the second iteration of the app and our use of Apollo Client. Here are the key points on using Apollo Client:

- Apollo Client is a specialized library for interacting with GraphQL APIs
- Unlike React Query, Apollo Client interacts directly with the GraphQL API and, therefore, doesn't require a separate function that uses `fetch`
- Apollo Client's `ApolloProvider` component needs to be placed high in the component tree above where backend data is needed
- Apollo Client's `useQuery` hook allows data to be fetched and cached in state
- Apollo Client's `useMutation` hook allows data to be updated

Next, we will summarize the chapter.

Summary

In this chapter, we started by learning the GraphQL syntax for queries and mutations. A great feature of GraphQL is the ability to request and receive only the required objects and fields. This can really help the performance of our apps.

We used React Query and `fetch` to interact with a GraphQL API. This is very similar to interacting with a REST API, but the HTTP method needs to be `POST`, and the query or mutation needs to be placed in the request body. A new feature we learned about in React Query is the ability to trigger queries when the user interacts with the app using the `enabled` option.

We refactored the app to use Apollo Client, which is a specialized GraphQL client. It is very similar to React Query in that it has `useQuery` and `useMutation` hooks and a provider component. One advantage over React Query is that Apollo Client interacts directly with the GraphQL API, which means we write less code.

In the next chapter, we will cover patterns that help us build reusable components.

Questions

Answer the following questions to check what you have learned in this chapter:

1. The following is an attempt at a GraphQL query to get a GitHub viewer's name and email address:

```
viewer: {
  name,
  email
}
```

   The query errors though – what is the problem?

2. What is the mutation that would unstar a GitHub repository? The mutation should have a parameter for the repository ID.
3. The following use of `fetch` is an attempt to call a GraphQL API:

```
const response = await fetch(process.env.REACT_APP_API_
URL!, {
  body: JSON.stringify({
    query: GET_DATA_QUERY,
  }),
});
```

   This doesn't work though – what is the problem?
4. Where does the authorization access token in a protected GraphQL API get specified when using Apollo Client?
5. A component uses Apollo Client's `useQuery` hook to fetch data from a GraphQL API, but the component errors with the following error:

   Could not find "client" in the context or passed in as an option. Wrap the root component in an <ApolloProvider>, or pass an ApolloClient instance in via options

   What do you think the problem is?
6. The following attempts to use Apollo Client's `useQuery` hook to fetch data from a GraphQL API:

```
const { loading, data } = useQuery(`query {
  contact {
    name
    email
  }
}
`);
```

   The call errors, though – what do you think the problem is?
7. What state variable can be destructured from Apollo Client's `useMutation` hook to determine whether the request has returned an error?

Answers

1. The query syntax is incorrect – the syntax is like JSON but doesn't have colons and commas. Also, the `query` keyword can be omitted, but it is best practice to include this. Here is the corrected query:

```
query {
  viewer {
    name
    email
  }
}
```

2. The following mutation will unstar a GitHub repository:

```
mutation ($repoId: ID!) {
  removeStar(input: { starrableId: $repoId }) {
    starrable {
      stargazers {
        totalCount
      }
    }
  }
}
```

3. The request is missing the HTTP `POST` method:

```
const response = await fetch(process.env.REACT_APP_API_
URL!, {
  method: 'POST',
  body: JSON.stringify({
    query: GET_DATA_QUERY,
  }),
});
```

4. The authorization access token gets specified when Apollo Client is created, which is passed into the provider component:

```
const queryClient = new ApolloClient({
  ...,
  headers: {
```

```
    Authorization: `bearer ${process.env.REACT_APP_
ACCESS_TOKEN}`,
  },
});

function App() {
  return (
    <ApolloProvider
      client={queryClient}
    >
      ...
    </ApolloProvider>
  );
}
```

5. The problem is that Apollo Client's `ApolloProvider` component hasn't been placed above the component using `useQuery` in the component tree.
6. The `gql` function must be applied to the query string to convert it into the object format that Apollo Client expects:

```
const { loading, data } = useQuery(gql`
  query {
    viewer {
      name
      email
    }
  }
`);
```

7. The `error` state variable can be destructured from React Query's `useMutation` hook to determine whether the HTTP request has returned an error.