GraphQL Caching of Queries with Apollo Client in React

In this lesson, you will implement the Organization Component and learn how to cache queries using GraphQL with Apollo Client in React.

WE'LL COVER THE FOLLOWING

In this lesson, we will introduce React Router to show two separate pages for your application. At the moment, we are only showing one page with a `Profile` component that displays all your repositories. We want to add another `Organization` component that shows repositories by an organization, and there could be a search field as well, to look up individual organizations with their repositories on that page. Let's do this by introducing React Router to our application. If you haven't used React Router before, make sure to conduct the exercises of this lesson to learn more about it.

In your *src/constants/routes.js* file, you can specify both routes you want to make accessible by React Router. The `ORGANIZATION` route points to the base URL, while the `PROFILE` route points to a more specific URL.

Environment Variables

Key:	Value:
REACT_APP_GITHUB...	Not Specified...
GITHUB_PERSONAL...	Not Specified...

```
export const ORGANIZATION = '/';
export const PROFILE = '/profile';
```

src/constants/routes.js

Next, map both routes to their components. The `App` component is the perfect place to do it because the two routes will exchange the `Organization` and `Profile` components based on the URL there.

Environment Variables

Key:	Value:
REACT_APP_GITHUB...	Not Specified...
GITHUB_PERSONAL...	Not Specified...

```
import React, { Component } from 'react';
import { BrowserRouter as Router, Route } from 'react-router-dom';

import Profile from '../Profile';
import Organization from '../Organization';

import * as routes from '../constants/routes';

import './style.css';

class App extends Component {
  render() {
    return (
      <Router>
        <div className="App">
          <div className="App-main">
            <Route
              exact
              path={routes.ORGANIZATION}
              component={() => (
                <div className="App-content_large-header">
                  <Organization />
                </div>
              )}
            />
            <Route
              exact
              path={routes.PROFILE}
              component={() => (
                <div className="App-content_small-header">
                  <Profile />
                </div>
              )}
            />
          </div>
        </div>
      </Router>
    );
  }
}

export default App;
```

src/App/index.js

The `Organization` component wasn't implemented yet, but you can start with a functional stateless component in the *src/Organization/index.js* file, that acts as a placeholder to keep the application working for now.

Environment Variables

Key:	Value:
REACT_APP_GITHUB...	Not Specified...
GITHUB_PERSONAL...	Not Specified...

```
import React from 'react';

const Organization = () => <div>Organization</div>;

export default Organization;
```

src/Organization/index.js

Since you mapped both routes to their respective components, so you want to implement navigation from one route to another. For this, introduce a **`Navigation`** component in the `App` component.

Environment Variables

Key:	Value:
REACT_APP_GITHUB...	Not Specified...
GITHUB_PERSONAL...	Not Specified...

```
...

import Navigation from './Navigation';
import Profile from '../Profile';
import Organization from '../Organization';

...

class App extends Component {
  render() {
    return (
      <Router>
        <div className="App">
          <Navigation />

          <div className="App-main">
            ...
          </div>
        </div>
```

```
      </Router>
    );
  }
}

export default App;
```

src/App/index.js

Next, we'll implement the `Navigation` component, which is responsible for displaying the two links to navigate between your routes using React Router's Link component.

Environment Variables

Key:	Value:
REACT_APP_GITHUB...	Not Specified...
GITHUB_PERSONAL...	Not Specified...

```
import React from 'react';
import { Link } from 'react-router-dom';

import * as routes from '../../constants/routes';

import './style.css';

const Navigation = () => (
  <header className="Navigation">
    <div className="Navigation-link">
      <Link to={routes.PROFILE}>Profile</Link>
    </div>
    <div className="Navigation-link">
      <Link to={routes.ORGANIZATION}>Organization</Link>
    </div>
  </header>
);

export default Navigation;
```

src/App/Navigation/index.js

The Profile page works as before, but the Organization page is empty. In the last step, you defined the two routes as constants, used them in the `App` component to map to their respective components, and introduced `Link` components to navigate to them in the `Navigation` component.

Query Caching by Apollo Client

Another great feature of the Apollo Client is that it caches query requests.

When navigating from the Profile page to the Organization page and back to the Profile page, the results appear immediately because the Apollo Client checks its cache before making the query to the remote GraphQL API. It's a pretty powerful tool.

`Organization` Component

The next part of this lesson is the `Organization` component. It is the same as the `Profile` component, except that the query differs because it takes a variable for the organization name to identify the organization's repositories.

Environment Variables

Key:	Value:
REACT_APP_GITHUB...	Not Specified...
GITHUB_PERSONAL...	Not Specified...

```
import React from 'react';
import gql from 'graphql-tag';
import { Query } from 'react-apollo';

import { REPOSITORY_FRAGMENT } from '../Repository';

const GET_REPOSITORIES_OF_ORGANIZATION = gql`
  query($organizationName: String!) {
    organization(login: $organizationName) {
      repositories(first: 5) {
        edges {
          node {
            ...repository
          }
        }
      }
    }
  }
  ${REPOSITORY_FRAGMENT}
`;

const Organization = ({ organizationName }) => (
  <Query
    query={GET_REPOSITORIES_OF_ORGANIZATION}
    variables={{
      organizationName,
    }}
    skip={organizationName === ''}
  >
    {({ data, loading, error }) => {
      ...
    }}
  </Query>
);

export default Organization;
```

src/Organization/index.js

The `Query` component in the `Organization` component takes a query tailored to the organization being the top level field of the query. It takes a variable to identify the organization, and it uses the newly introduced `skip` prop to skip executing the query if no organization identifier is provided. Later, you will pass an organization identifier from the `App` component. You may have noticed that the repository fragment you introduced earlier to update the local state in the cache can be reused here. It saves lines of code, and more importantly, ensures the returned list of repositories have identical structures to the list of repositories in the `Profile` component.

Next, extend the query to fit the requirements of the pagination feature. It requires the `cursor` argument to identify the next page of repositories. The `notifyOnNetworkStatusChange` prop is used to update the `loading` boolean for paginated requests as well.

Environment Variables

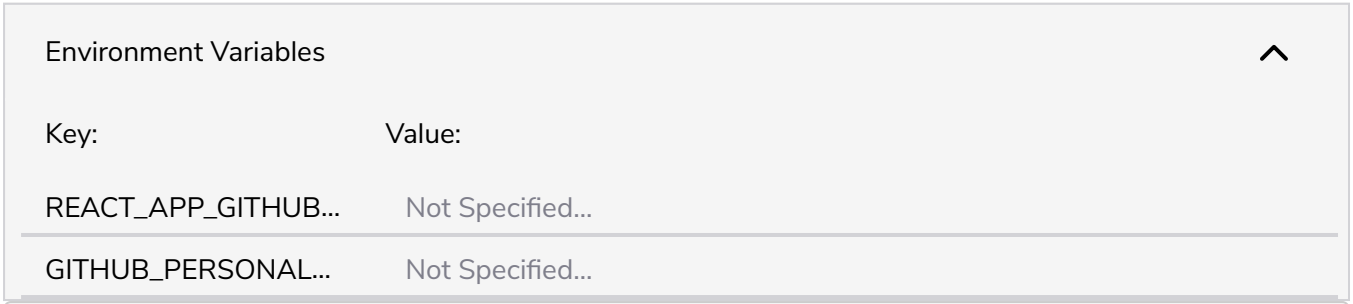
Key:	Value:
REACT_APP_GITHUB...	Not Specified...
GITHUB_PERSONAL...	Not Specified...

```
...

const GET_REPOSITORIES_OF_ORGANIZATION = gql`
  query($organizationName: String!, $cursor: String) {
    organization(login: $organizationName) {
      repositories(first: 5, after: $cursor) {
        edges {
          node {
            ...repository
          }
        }
        pageInfo {
          endCursor
          hasNextPage
        }
      }
    }
  }
  ${REPOSITORY_FRAGMENT}
`;

const Organization = ({ organizationName }) => (
  <Query
    query={GET_REPOSITORIES_OF_ORGANIZATION}
    variables={{
      organizationName,
```

```
    }}
    skip={organizationName === ''}
    notifyOnNetworkStatusChange={true}
  >
    {({ data, loading, error, fetchMore }) => {
      ...
    }}
  </Query>
);

export default Organization;
```

src/Organization/index.js

Lastly, the render prop child function needs to be implemented. It doesn't differ much from the Query's content in the `Profile` component. Its purpose is to handle edge cases like loading and `no data` errors, and eventually, to show a list of repositories. Because the `RepositoryList` component handles the pagination feature, this improvement is included in the newly implemented `Organization` component.

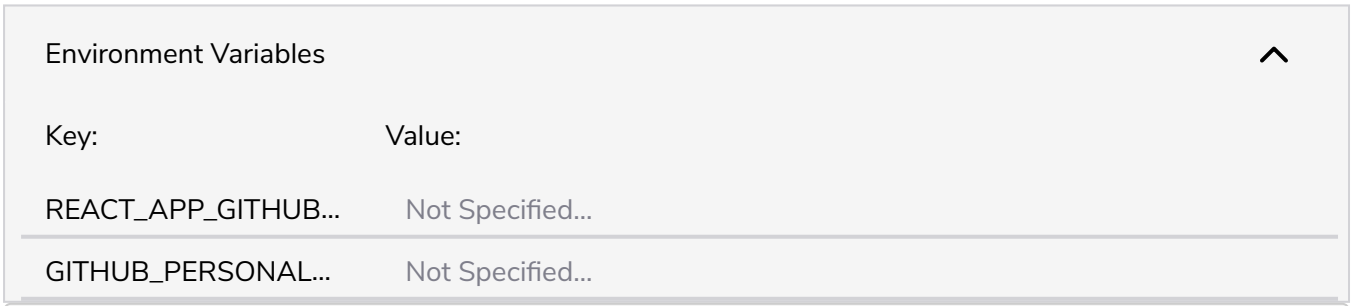



```
...

import RepositoryList, { REPOSITORY_FRAGMENT } from '../Repository';
import Loading from '../Loading';
import ErrorMessage from '../Error';

...

const Organization = ({ organizationName }) => (
  <Query ... >
    {({ data, loading, error, fetchMore }) => {
      if (error) {
        return <ErrorMessage error={error} />;
      }

      const { organization } = data;

      if (loading && !organization) {
        return <Loading />;
      }

      return (
        <RepositoryList
          loading={loading}
          repositories={organization.repositories}
          fetchMore={fetchMore}
```

```
        />
      );
    }}
  </Query>
);

export default Organization;
```

src/Organization/index.js

Provide a `organizationName` as prop when using the `Organization` in the `App` component, and leave it inlined for now. Later, you will make it dynamic with a search field.




```
class App extends Component {
  render() {
    return (
      <Router>
        <div className="App">
          <Navigation />

          <div className="App-main">
            <Route
              exact
              path={routes.ORGANIZATION}
              component={() => (
                <div className="App-content_large-header">
                  <Organization
                    organizationName={'the-road-to-learn-react'}
                  />
                </div>
              )}
            />
            ...
          </div>
        </div>
      </Router>
    );
  }
}
```

The `Organization` component should almost work now, as the `More` button is the only incomplete part. The remaining issue is the resolving block for the pagination feature in the `updateQuery` function. It assumes that the nested data structure always starts with a `viewer` object. It does for the Profile page,

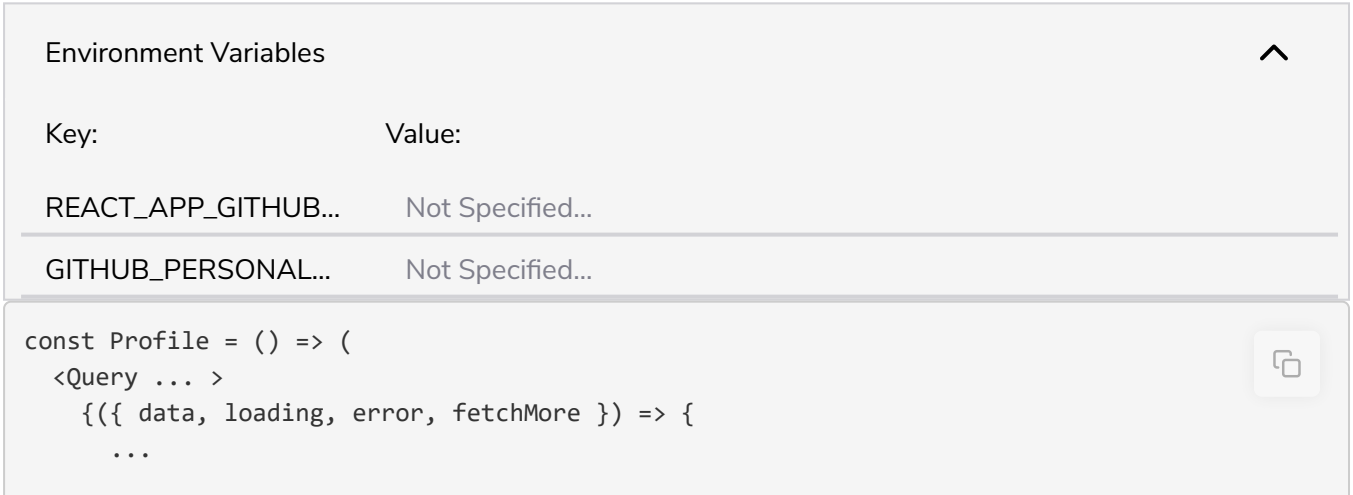
but not for the Organization page. There the top level object is the

`organization` followed by the list of `repositories`. Only the top-level object changes from page to page, where the underlying structure stays identical.

When the top level object changes from page to page, the ideal next step is to tell the `RepositoryList` component its top-level object from the outside. With the `Organization` component, it's the top-level object `organization`, which could be passed as a string and reused as a dynamic key later:

src/Organization/index.js

With the `Profile` component, the `viewer` would be the top-level object:



```
      return (
        <RepositoryList
          loading={loading}
          repositories={viewer.repositories}
          fetchMore={fetchMore}
          entry={'viewer'}
        />
      );
    }}
  </Query>
);
```

src/Profile/index.js

Now you can handle the new case in the `RepositoryList` component by passing the entry as computed property name to the `updateQuery` function. Instead of passing the `updateQuery` function directly to the `FetchMore` component, it can be derived from a higher-order function needed to pass the new `entry` property.

Environment Variables

Key:	Value:
REACT_APP_GITHUB...	Not Specified...
GITHUB_PERSONAL...	Not Specified...

```
const RepositoryList = ({
  repositories,
  loading,
  fetchMore,
  entry,
}) => (
  <Fragment>
    ...

    <FetchMore
      loading={loading}
      hasNextPage={repositories.pageInfo.hasNextPage}
      variables={{
        cursor: repositories.pageInfo.endCursor,
      }}
      updateQuery={getUpdateQuery(entry)}
      fetchMore={fetchMore}
    >
      Repositories
    </FetchMore>
  </Fragment>
);
```

src/Repository/RepositoryList/index.js

The higher-order function next to the `RepositoryList` component is completed

as such:

Environment Variables

Key:	Value:
REACT_APP_GITHUB...	Not Specified...
GITHUB_PERSONAL...	Not Specified...

```
const getUpdateQuery = entry => (
  previousResult,
  { fetchMoreResult },
) => {
  if (!fetchMoreResult) {
    return previousResult;
  }

  return {
    ...previousResult,
    [entry]: {
      ...previousResult[entry],
      repositories: {
        ...previousResult[entry].repositories,
        ...fetchMoreResult[entry].repositories,
        edges: [
          ...previousResult[entry].repositories.edges,
          ...fetchMoreResult[entry].repositories.edges,
        ],
      },
    },
  };
};
```

src/Repository/RepositoryList/index.js

That's how a deeply-nested object is updated with the `fetchMoreResult`, even though the top level component from the query result is not static. The pagination feature should work on both pages now. Take a moment to recap the last implementations again and why these were necessary.

Next, we'll implement the search function I mentioned earlier. The best place to add the search field would be the `Navigation` component, but only when the Organization page is active. React Router comes with a useful higher-order component to access the current URL, which can be used to show a search field.

Environment Variables

Key:	Value:

Key:	Value:
REACT_APP_GITHUB...	Not Specified...
GITHUB_PERSONAL...	Not Specified...

```
import React from 'react';
import { Link, withRouter } from 'react-router-dom';

import * as routes from '../../constants/routes';

import './style.css';

const Navigation = ({
  location: { pathname },
}) => (
  <header className="Navigation">
    <div className="Navigation-link">
      <Link to={routes.PROFILE}>Profile</Link>
    </div>
    <div className="Navigation-link">
      <Link to={routes.ORGANIZATION}>Organization</Link>
    </div>

    {pathname === routes.ORGANIZATION && (
      <OrganizationSearch />
    )}
  </header>
);

export default withRouter(Navigation);
```

src/App/Navigation/index.js

The `OrganizationSearch` component is implemented next to the `Navigation` component in the next steps. Before that can work, there needs to be some kind of initial state for the `OrganizationSearch`, as well as a callback function to update the initial state in the `Navigation` component. To accommodate this, the `Navigation` component becomes a class component.

Environment Variables

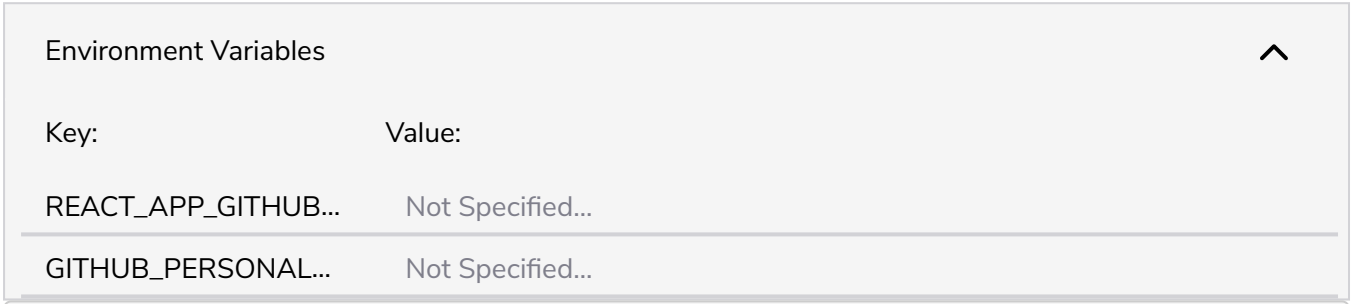
Key:	Value:
REACT_APP_GITHUB...	Not Specified...
GITHUB_PERSONAL...	Not Specified...

```
...

class Navigation extends React.Component {
  state = {
    organizationName: 'the-road-to-learn-react',
  };

  onOrganizationSearch = value => {
    this.setState({ organizationName: value });
  };
```

```
  render() {
    const { location: { pathname } } = this.props;

    return (
      <header className="Navigation">
        <div className="Navigation-link">
          <Link to={routes.PROFILE}>Profile</Link>
        </div>
        <div className="Navigation-link">
          <Link to={routes.ORGANIZATION}>Organization</Link>
        </div>

        {pathname === routes.ORGANIZATION && (
          <OrganizationSearch
            organizationName={this.state.organizationName}
            onOrganizationSearch={this.onOrganizationSearch}
          />
        )}
      </header>
    );
  }
}

export default withRouter(Navigation);
```

src/App/Navigation/index.js

The `OrganizationSearch` component implemented in the same file would also work with the following implementation. It handles its own local state, the value that shows up in the input field, but uses it as an initial value from the parent component. It also receives a callback handler, which can be used in the `onSubmit()` class method to propagate the search fields value on a submit interaction up the component tree.




```
...

import Button from '../../Button';
import Input from '../../Input';

import './style.css';

const Navigation = ({ ... }) => ...

class OrganizationSearch extends React.Component {
  state = {
    value: this.props.organizationName,
```

```
  };

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

  onSubmit = event => {
    this.props.onOrganizationSearch(this.state.value);

    event.preventDefault();
  };

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

    return (
      <div className="Navigation-search">
        <form onSubmit={this.onSubmit}>
          <Input
            color={'white'}
            type="text"
            value={value}
            onChange={this.onChange}
          />{' '}
          <Button color={'white'} type="submit">
            Search
          </Button>
        </form>
      </div>
    );
  }
}

export default withRouter(Navigation);
```

src/App/Navigation/index.js

The `Input` component is a slightly styled input element that is defined in *src/Input/index.js* as its own component.

Environment Variables

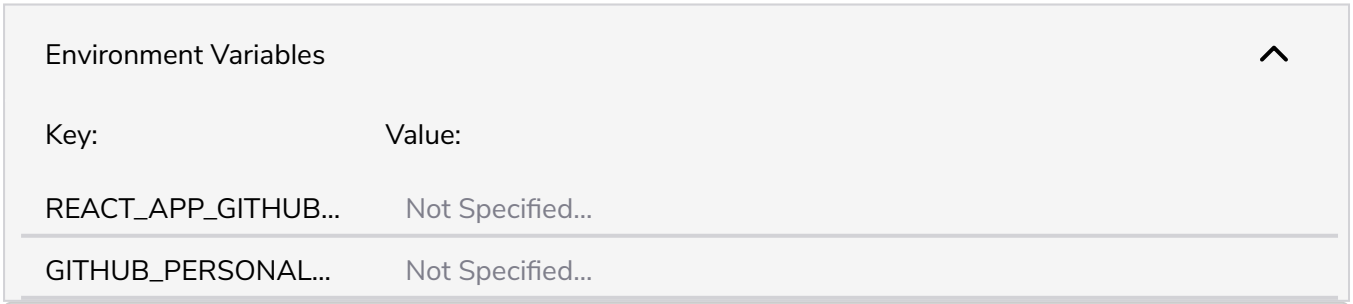
Key:	Value:
REACT_APP_GITHUB...	Not Specified...
GITHUB_PERSONAL...	Not Specified...

```
import React from 'react';

import './style.css';

const Input = ({ children, color = 'black', ...props }) => (
  <input className={`Input Input_${color}`} {...props}>
    {children}
  </input>
);
```

```
export default Input;
```

src/Input/index.js

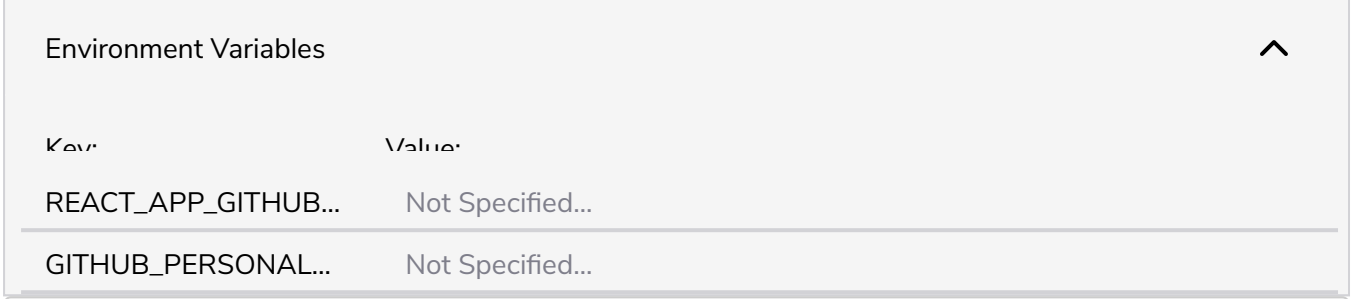
While the search field works in the `Navigation` component, it doesn't help the rest of the application. It only updates the state in the `Navigation` component when a search request is submitted. However, the value of the search request is needed in the `Organization` component as a GraphQL variable for the query, so the local state needs to be lifted up from the Navigation component to the App component. The `Navigation` component becomes a stateless functional component again.




```
const Navigation = ({
  location: { pathname },
  organizationName,
  onOrganizationSearch,
}) => (
  <header className="Navigation">
    <div className="Navigation-link">
      <Link to={routes.PROFILE}>Profile</Link>
    </div>
    <div className="Navigation-link">
      <Link to={routes.ORGANIZATION}>Organization</Link>
    </div>

    {pathname === routes.ORGANIZATION && (
      <OrganizationSearch
        organizationName={organizationName}
        onOrganizationSearch={onOrganizationSearch}
      />
    )}
  </header>
);
```

src/App/Navigation/index.js

The `App` component takes over the responsibility from the `Navigation` component, managing the local state, passing the initial state and a callback function to update the state to the Navigation component, and passing the state itself to the `Organization` component to perform the query:




```
...

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

  onOrganizationSearch = value => {
    this.setState({ organizationName: value });
  };

  render() {
    const { organizationName } = this.state;

    return (
      <Router>
        <div className="App">
          <Navigation
            organizationName={organizationName}
            onOrganizationSearch={this.onOrganizationSearch}
          />

          <div className="App-main">
            <Route
              exact
              path={routes.ORGANIZATION}
              component={() => (
                <div className="App-content_large-header">
                  <Organization organizationName={organizationName} />
                </div>
              )}
            />
            ...
          </div>
        </div>
      </Router>
    );
  }
}

export default App;
```

src/App/index.js

We have implemented a dynamic GraphQL query with a search field. Once a new `organizationName` is passed to the `Organization` component from a local state change, the `Query` component triggers another request due to a re-render. The request is not always made to the remote GraphQL API, though. The Apollo Client cache is used when an organization is searched twice. Also,

we have used the well-known technique called lifting state in React to share the state across components.
Check it out below:

Environment Variables

Key:	Value:
REACT_APP_GITHUB...	Not Specified...
GITHUB_PERSONAL...	Not Specified...

```
import React from 'react';

import Link from '../../Link';

import './style.css';

const Footer = () => (
  <div className="Footer">
    <div>
      <small>
        <span className="Footer-text">Built by</span>{' '}
        <Link
          className="Footer-link"
          href="https://www.robinwieruch.de"
        >
          Robin Wieruch
        </Link>{' '}
        <span className="Footer-text">with &hearts;</span>
      </small>
    </div>
    <div>
      <small>
        <span className="Footer-text">
          Interested in GraphQL, Apollo and React?
        </span>{' '}
        <Link
          className="Footer-link"
          href="https://www.getrevue.co/profile/rwieruch"
        >
          Get updates
        </Link>{' '}
        <span className="Footer-text">
          about upcoming articles, books &
        </span>{' '}
        <Link className="Footer-link" href="https://roadtoreact.com">
          courses
        </Link>
        <span className="Footer-text">.</span>
      </small>
    </div>
  </div>
);

export default Footer;
```

Exercise

1. Confirm your source code for the last section
2. If you are not familiar with React Router, try it out in this pragmatic tutorial