Implementing the Issues Feature: Setup

In this lesson, we will fetch GitHub issues using a list field associated with a repository in a GraphQL query.

WE'LL COVER THE FOLLOWING

- Exercise
- Reading Task

In the previous lessons, we have implemented most of the common Apollo Client features in our React application. Now we can start implementing extensions for the application on our own. This lesson showcases how a full-fledged feature can be implemented with Apollo Client in React.

So far, you have dealt with GitHub repositories from organizations and your account. This will take us one step further, fetching GitHub issues that are made available using a list field associated with a repository in a GraphQL query. However, this lesson doesn't only show you how to render a nested list field in your React application.

The foundation will be rendering the list of issues. You will implement client-side filtering with plain React to show opened, closed, or no issue. Finally, you will refactor the filtering to server-side filtering using GraphQL queries. We will only fetch the issues by their state from the server rather than filtering the issue's state on the client-side. Implementing pagination for these issues will be your exercise.

First, we will render a new component called `Issues` in our `RepositoryList` component. This component takes two props that are used later in a GraphQL query to identify the repository from which you want to fetch the issues.

Environment Variables

Key: Value:

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

```
import FetchMore from '../../FetchMore';
import RepositoryItem from '../RepositoryItem';
import Issues from '../../Issue';

...

const RepositoryList = ({
  repositories,
  loading,
  fetchMore,
  entry,
}) => (
  <Fragment>
    {repositories.edges.map(({ node }) => (
      <div key={node.id} className="RepositoryItem">
        <RepositoryItem {...node} />

        <Issues
          repositoryName={node.name}
          repositoryOwner={node.owner.login}
        />
      </div>
    ))}

    ...
  </Fragment>
);

export default RepositoryList;
```

src/Repository/RepositoryList/index.js

In the *src/Issue/index.js* file, import and export the `Issues` component. Since the issue feature can be kept in a module on its own, it has this *index.js* file again. That's how you can tell other developers to access only this feature module, using the *index.js* file as its interface. Everything else is kept private.

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

```
import Issues from './IssueList';

export default Issues;
```

src/Issues/index.js

Note how the component is named `Issues`, not `IssueList`. The naming convention is used to break down the rendering of a list of items:

- `Issues`
- `IssueList`
- `IssueItem`.

`Issues` is the container component, where you query the data and filter the issues, and the `IssueList` and `IssueItem` are only there as presentational components for rendering. In contrast, the Repository feature module hasn't a `Repositories` component, because there was no need for it. The list of repositories already came from the `Organization` and `Profile` components and the Repository module's components are mainly only there for the rendering. This is only one opinionated approach of naming the components, however.

Let's start implementing `Issues` and `IssueList` components in the *src/Issue/IssueList/index.js* file. You could argue to split both components up into their own files, but for the sake of this tutorial, they are kept together in one file.

First, there needs to be a new query for the issues. You might wonder: Why do we need a new query here? It would be simpler to include the issues list field in the query at the top next to the `Organization` and `Profile` components. That's true, but it comes with a cost. Adding more nested (list) fields to a query often results in performance issues on the server-side. There you may have to make multiple roundtrips to retrieve all the entities from the database.

- Roundtrip 1: get organization by name
- Roundtrip 2: get repositories of an organization by organization identifier
- Roundtrip 3: get issues of a repository by repository identifier

It is simple to conclude that nesting queries are a naive way to solve all of our problems. Although they solve the problem of only requesting the data once and not with multiple network request (similar roundtrips as shown for the database), GraphQL doesn't solve the problem of retrieving all the data from the database for you. That's not the responsibility of GraphQL after all. So by having a dedicated query in the `Issues` component, we can decide **when** to trigger this query. In the next steps, we will just trigger it on render because the `Query` component is used but when adding the client-side filter, later on, it

will only be triggered when the **“Filter”** button is toggled. Otherwise, the

issues should be hidden. Finally, that’s how all the initial data loading can be delayed to a point when the user actually wants to see the data.

First, let’s define the `Issues` component which has access to the props which were passed in the `RepositoryList` component. It doesn’t render much yet.

Environment Variables

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

```
import React from 'react';

import './style.css';

const Issues = ({ repositoryOwner, repositoryName }) =>
  <div className="Issues">
  </div>

export default Issues;
```

src/Issue/IssueList/index.js

Second, let’s define the query in the *src/Issue/IssueList/index.js* file to retrieve issues of a repository. The repository is identified by its owner and name. Also, add the `state` field as one of the fields for the query result. This is used for client-side filtering, for showing issues with an open or closed state.

Environment Variables

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

```
import React from 'react';
import gql from 'graphql-tag';

import './style.css';

const GET_ISSUES_OF_REPOSITORY = gql`
  query($repositoryOwner: String!, $repositoryName: String!) {
    repository(name: $repositoryName, owner: $repositoryOwner) {
      issues(first: 5) {
        edges {
```

```
          node {
            id
            number

            state
            title
            url
            bodyHTML
          }
        }
      }
    }
  }
`;

...
```

src/Issue/IssueList/index.js

Third, we'll introduce the `Query` component and pass it the previously defined query and the necessary variables. We'll use its render prop child function to access the data, to cover all edge cases and to render an `IssueList` component eventually.

Environment Variables

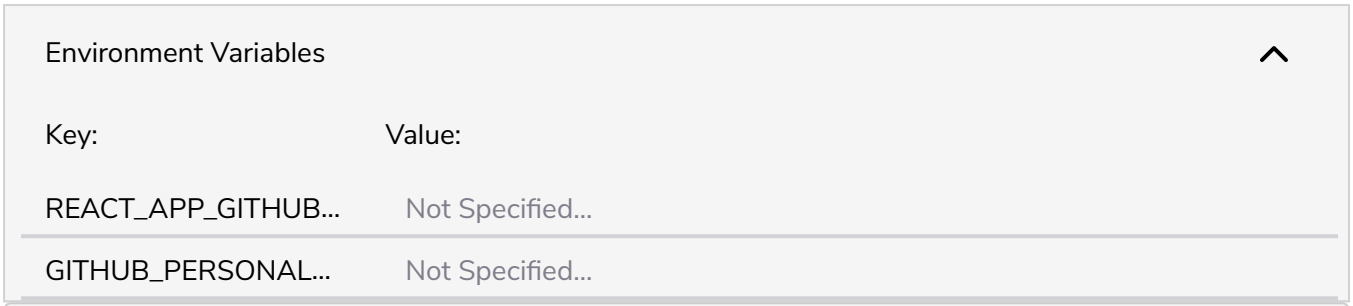
Key:	Value:
REACT_APP_GITHUB...	Not Specified...
GITHUB_PERSONAL...	Not Specified...

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

import IssueItem from '../IssueItem';
import Loading from '../../Loading';
import ErrorMessage from '../../Error';

import './style.css';

const Issues = ({ repositoryOwner, repositoryName }) => (
  <div className="Issues">
    <Query
      query={GET_ISSUES_OF_REPOSITORY}
      variables={{
        repositoryOwner,
        repositoryName,
      }}
    >
      {({ data, loading, error }) => {
        if (error) {
          return <ErrorMessage error={error} />;
        }

        const { repository } = data;
```

```
        if (loading && !repository) {
          return <Loading />;

        }

        if (!repository.issues.edges.length) {
          return <div className="IssueList">No issues ...</div>;
        }

        return <IssueList issues={repository.issues} />;
      }}
    </Query>
  </div>
);

const IssueList = ({ issues }) => (
  <div className="IssueList">
    {issues.edges.map(({ node }) => (
      <IssueItem key={node.id} issue={node} />
    ))}
  </div>
);

export default Issues;
```

src/Issue/IssueList/index.js

Finally, we'll implement a basic `IssueItem` component in the *src/Issue/IssueItem/index.js* file. The snippet below shows a placeholder where you can implement the Commenting feature, which we'll cover later.

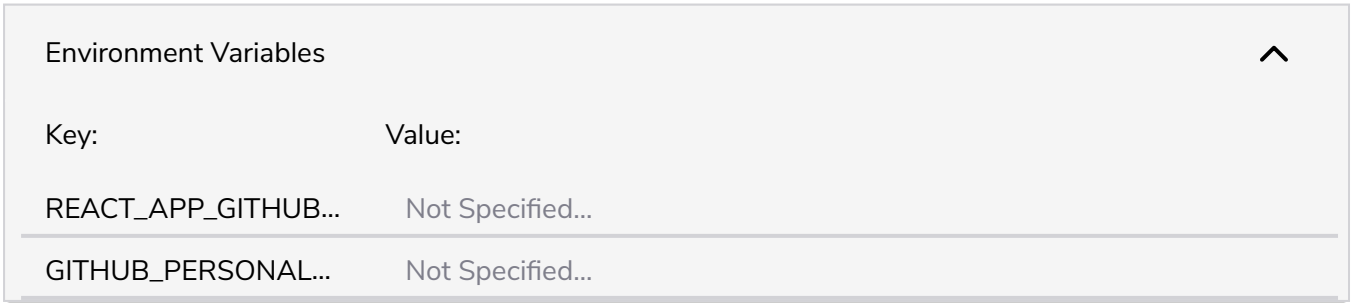



```
import React from 'react';

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

import './style.css';

const IssueItem = ({ issue }) => (
  <div className="IssueItem">
    {/* placeholder to add a show/hide comment button later */}

    <div className="IssueItem-content">
      <h3>
        <Link href={issue.url}>{issue.title}</Link>
      </h3>
      <div dangerouslySetInnerHTML={{ __html: issue.bodyHTML }} />

      {/* placeholder to render a list of comments later */}
    </div>
```

```
  </div>
);


export default IssueItem;
```

src/Issue/IssueItem/index.js

Once you start the application again, you should see the initial page of paginated issues rendered below each repository. That's a performance bottleneck. Worse, the GraphQL requests are not bundled in one request, as with the issues list field in the `Organization` and `Profile` components. In the next step, we will be implementing client-side filtering. By default, it shows no issues, but we can toggle between states of showing no issues, open issues, and closed issues using a button.




```
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

Reading Task

1. Read more about the rate limit when using a (or in this case GitHub's) GraphQL API