GraphQL Pagination with Apollo Client in React

In this lesson, we will use GraphQL API to implement the pagination feature for our application.

Finally, you are going to implement another advanced feature while using GraphQL API called **pagination**.

In this lesson, we will implement a button that allows successive pages of repositories to be queried and a simple **"More"** button rendered below the list of repositories in the `RepositoryList` component. When the button is clicked, another page of repositories is fetched and merged with the previous list as one state into Apollo Client's cache.

First, let's extend the query next for your `Profile` component with the necessary information to allow pagination for the list of repositories:

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

```
const GET_REPOSITORIES_OF_CURRENT_USER = gql`
  query($cursor: String) {
    viewer {
      repositories(
        first: 5
        orderBy: { direction: DESC, field: STARGAZERS }
        after: $cursor
      ) {
        edges {
          node {
            ...repository
          }
```

```
        }
      pageInfo {
        endCursor
        hasNextPage
      }
    }
  }
}
${REPOSITORY_FRAGMENT}
`;
```

src/Profile/index.js

The `endCursor` can be used as `$cursor` variable when fetching the next page
of repositories, but the `hasNextPage` can disable the functionality (e.g. not
showing the **"More"** button) to fetch another page. The initial request to fetch
the first page of repositories will have a `$cursor` variable of `undefined`,
though. GitHub's GraphQL API will handle this case gracefully and return the
first items from the list of repositories without considering the `after`
argument. Every other request to fetch more items from the list will send a
defined `after` argument with the cursor, which is the `endCursor` from the
query.

Now we have all the information to fetch more pages of repositories from
GitHub's GraphQL API. The `Query` component exposes a function to retrieve
them in its child function. Since the button to fetch more repositories fits best
in the `RepositoryList` component, you can pass this function as a prop to it.

Environment Variables   ⌃

Key: Value:

REACT_APP_GITHUB...   Not Specified...

GITHUB_PERSONAL...   Not Specified...

```
const Profile = () => (
  <Query query={GET_REPOSITORIES_OF_CURRENT_USER}>
    {({ data, loading, error, fetchMore }) => {
      ...

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

Next, we will use the function in the `RepositoryList` component, and add a button to fetch successive pages of repositories that appears when another page is available.

```js
import React, { Fragment } from 'react';

...

const RepositoryList = ({ repositories, fetchMore }) => (
  <Fragment>
    {repositories.edges.map(({ node }) => (
      ...
    ))}

    {repositories.pageInfo.hasNextPage && (
      <button
        type="button"
        onClick={() =>
          fetchMore({
            /* configuration object */
          })
        }
      >
        More Repositories
      </button>
    )}
  </Fragment>
);

export default RepositoryList;
```

The `fetchMore()` function performs the query from the initial request, and takes a configuration object, which can be used to override variables. With pagination, this means you pass the `endCursor` of the previous query result to use it for the query as `after` argument. Otherwise, you would perform the initial request again because no variables are specified.

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

    {repositories.pageInfo.hasNextPage && (
      <button
        type="button"
        onClick={() =>
          fetchMore({
            variables: {
              cursor: repositories.pageInfo.endCursor,
            },
          })
        }
      >
        More Repositories
      </button>
    )}
  </Fragment>
);
```

src/Repository/RepositoryList/index.js

If you attempt to click the button now, you should get the following error message: *Error: updateQuery option is required.*. The `updateQuery` function is needed to tell Apollo Client how to merge the previous result with a new one. We will now define the function outside of the button because it would become too verbose otherwise.

```
const updateQuery = (previousResult, { fetchMoreResult }) => {
  ...
};

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

    {repositories.pageInfo.hasNextPage && (
      <button
        type="button"
        onClick={() =>
          fetchMore({
```

```
            variables: {
                cursor: repositories.pageInfo.endCursor,
            },

                updateQuery,
            })
        }
      >
        More Repositories
      </button>
    )}
  </Fragment>
);
```

The function has access to the previous query result, and to the next result that resolves after the button click:

Environment Variables   ^

Key: Value:

REACT_APP_GITHUB...   Not Specified...

GITHUB_PERSONAL...   Not Specified...

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

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

In this function, you can merge both results with the JavaScript spread operator. If there is no new result, return the previous result. The important part is merging the `edges` of both repositories objects to have a merge list of items. The `fetchMoreResult` takes precedence over the `previousResult` in the

`repositories` object because it contains the new `pageInfo`, with its `endCursor` and `hasNextPage` properties from the last paginated result. We need to have those when clicking the button another time to have the correct cursor as an argument. If you want to check out an alternative to the verbose JavaScript spread operator when dealing with deeply nested data, check out the changes in this GitHub Pull Request that uses Lenses from Ramda.js.

To add one more small improvement for user-friendliness, let's add a loading indicator when more pages are fetched. So far, the `loading` boolean in the `Query` component of the `Profile` component is only true for the initial request, but not for the following requests. Now, we'll change this behavior with a prop that is passed to the `Query` component, and the loading boolean will be updated accordingly.

Environment Variables ⌄

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

```
const Profile = () => (
  <Query
    query={GET_REPOSITORIES_OF_CURRENT_USER}
    notifyOnNetworkStatusChange={true}
  >
    {({ data, loading, error, fetchMore }) => {
      ...
    }}
  </Query>
);
```

src/Profile/index.js

When you run your application again and try the **"More"** button, you should see odd behavior. Every time you load another page of repositories, the loading indicator is shown, but the list of repositories disappears entirely, and the merged list is rendered as assumed. Since the `loading` boolean becomes true with the initial and successive requests, the conditional rendering in the `Profile` component will always show the loading indicator. It returns from the Profile function early, never reaching the code to render the `RepositoryList`. A quick change from `||` to `&&` of the condition will allow it to show the loading indicator for the initial request only. Every request after that, where the `viewer` object is available, is beyond this condition, so it

renders the `RepositoryList` component.

```
const Profile = () => (
  <Query
    query={GET_REPOSITORIES_OF_CURRENT_USER}
    notifyOnNetworkStatusChange={true}
  >
    {({ data, loading, error, fetchMore }) => {
      ...

      const { viewer } = data;

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

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

src/Profile/index.js

The boolean can be passed down to the `RepositoryList` component. There it can be used to show a loading indicator instead of the **"More"** button. Since the boolean never reaches the `RepositoryList` component for the initial request, you can be sure that the **"More"** button only changes to the loading indicator when there is a successive request pending.

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

import Loading from '../../Loading';
```

```
import RepositoryItem from '../RepositoryItem';

...

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

    {loading ? (
      <Loading />
    ) : (
      repositories.pageInfo.hasNextPage && (
        <button
          ...
        >
          More Repositories
        </button>
      )
    )}
  </Fragment>
);
```

src/Repository/RepositoryList/index.js

The pagination feature is complete now, and we are fetching successive pages of an initial page, then merging the results in Apollo Client's cache. In addition, we show our user feedback about pending requests for either the initial request or further page requests.

Now we'll take it a step further, making the button used to fetch more repositories reusable. Let me explain why this would be a neat abstraction. In an upcoming lesson, we have another list field that could potentially implement the pagination feature. There, you have to introduce the **More** button, which could be nearly identical to the **More** button you have in the `RepositoryList` component. Having only one button in a UI would be a satisfying abstraction, but this abstraction wouldn't work in a real-world coding scenario. You would have to introduce a second list field first, implement the pagination feature for it, and then consider an abstraction for the **More** button. For the sake of the tutorial, we implement this abstraction for the pagination feature only in this lesson, though you should be aware this is a premature optimization put in place for you to learn it.

For another way, imagine you wanted to extract the functionality of the **More** button into a `FetchMore` component. The most important thing you would need is the `fetchMore()` function from the query result. The `fetchMore()` function takes an object to pass in the necessary `variables` and `updateQuery` information as a configuration. While the former is used to define the next
```

page by its cursor, the latter is used to define how the results should be merged in the local state. These are the three essential parts:

1. `fetchMore`
2. variables
3. `updateQuery`

You may also want to shield away from the conditional renderings in the `FetchMore` component, which happens because of the `loading` or `hasNextPage` booleans. Et voilà! That's how you get the interface to your `FetchMore` abstraction component.

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

import FetchMore from '../../FetchMore';
import RepositoryItem from '../RepositoryItem';

...

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

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

export default RepositoryList;
```

Now this `FetchMore` component can be used by other paginated lists as well because every part that can be dynamic is passed as props to it. Implementing a `FetchMore` component in the *src/FetchMore/index.js* is the next step. First, the main part of the component:

```
import React from 'react';

import './style.css';

const FetchMore = ({
  variables,
  updateQuery,
  fetchMore,
  children,
}) => (
  <div className="FetchMore">
    <button
      type="button"
      className="FetchMore-button"
      onClick={() => fetchMore({ variables, updateQuery })}
    >
      More {children}
    </button>
  </div>
);

export default FetchMore;
```

src/FetchMore/index.js

Here, you can see how the `variables` and `updateQuery` are taken as a configuration object for the `fetchMore()` function when it's invoked. The button can be made cleaner using the **Button** component you defined in a previous lesson. To add a different style, let's define a specialized `ButtonUnobtrusive` component next to the `Button` component in the *src/Button/index.js* file:

```
import React from 'react';

import './style.css';

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

const ButtonUnobtrusive = ({
  children,
  className,
  type = 'button',
  ...props
}) => (
  <button
    className={`${className} Button_unobtrusive`}
    type={type}
    {...props}
  >
    {children}
  </button>
);

export { ButtonUnobtrusive };

export default Button;
```

src/Button/index.js

Now the `ButtonUnobtrusive` component is used as a button instead of the button element in the `FetchMore` component. In addition, the two booleans `loading` and `hasNextPage` can be used for the conditional rendering, to show the Loading component or nothing, because there is no next page which can be fetched.

Environment Variables ⌃

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

```
import React from 'react';

import Loading from '../Loading';
import { ButtonUnobtrusive } from '../Button';

import './style.css';

const FetchMore = ({
  loading,
  hasNextPage,
```

```
    variables,
    updateQuery,
    fetchMore,

    children,
}) => (
    <div className="FetchMore">
        {loading ? (
            <Loading />
        ) : (
            hasNextPage && (
                <ButtonUnobtrusive
                    className="FetchMore-button"
                    onClick={() => fetchMore({ variables, updateQuery })}
                >
                    More {children}
                </ButtonUnobtrusive>
            )
        )}
    </div>
);


export default FetchMore;
```

src/Fetchmore/index.js

That's it for the abstraction of the `FetchMore` button for paginated lists with
Apollo Client. Basically, you pass in everything needed by the `fetchMore()`
function, including the function itself. You can also pass all booleans used for
conditional renderings. You end up with a reusable `FetchMore` button that can
be used for every paginated list.

The code below has the entire pagination feature implemented with additions
to `FetchMore` and `Button` Components:

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

Reading Task

1. Read more about pagination with Apollo Client in React