React GraphQL Pagination

In this lesson, we will explore and implement pagination with list fields GraphQL in React.

WE'LL COVER THE FOLLOWING

In the last lesson, we implemented a list field in GraphQL query, which fits into the flow of structuring the query with nested objects and a list responsible for showing partial results of the query in React. Now, we will combine that knowledge with pagination.

Initially, we will learn more about the arguments of list fields. Further, we will add one more nested list field to the query. Finally, we will fetch another page of the paginated `issues` list with the query.

Let's start by extending the `issues` list field in your query with one more argument:

Environment Variables

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

```
const GET_ISSUES_OF_REPOSITORY = `
  query ($organization: String!, $repository: String!) {
    organization(login: $organization) {
      name
      url
      repository(name: $repository) {
        name
        url
        issues(last: 5, states: [OPEN]) {
          edges {
            node {
              id
```

```
            title
            url
          }
        }
      }
    }
  }
`;
```

If you read the arguments for the `issues` list field using the “Docs” sidebar in GraphQL, you can explore which arguments you can pass to the field. One of these is the `states` argument, which defines whether or not to fetch open or closed issues. The previous implementation of the query has shown you how to refine the list field, in case you only want to show open issues. You can explore more arguments for the `issues` list field, but also for other list fields, using the documentation from Github’s API.

Implement Another Nested List for Pagination

Now we’ll implement another nested list field that could be used for pagination. Each issue in a repository can have reactions, essentially emoticons like a smiley or a thumbs up. Reactions can be seen as another list of paginated items. First, extend the query with the nested list field for reactions:

Environment Variables

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

```
const GET_ISSUES_OF_REPOSITORY = `
  query ($organization: String!, $repository: String!) {
    organization(login: $organization) {
      name
      url
      repository(name: $repository) {
        name
        url
        issues(last: 5, states: [OPEN]) {
          edges {
            node {
              id
              title
              url
              reactions(last: 3) {
                edges {
```

```
                  node {
                    id
                    content

                  }
                }
              }
            }
          }
        }
      }
    }
  }
`;
```

Second, render the list of reactions in one of your React components again. Implement dedicated List and Item components, such as ReactionsList and ReactionItem for it. As an exercise, try keeping the code for this application readable and maintainable.

Environment Variables

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

```
const Repository = ({ repository }) => (
  <div>
    ...
    <ul>
      {repository.issues.edges.map(issue => (
        <li key={issue.node.id}>
          <a href={issue.node.url}>{issue.node.title}</a>

          <ul>
            {issue.node.reactions.edges.map(reaction => (
              <li key={reaction.node.id}>{reaction.node.content}</li>
            ))}
          </ul>
        </li>
      ))}
    </ul>
  </div>
);
```

You extended the query and React's component structure to render the result. It's a straightforward implementation when you are using a GraphQL API as your data source which has a well defined underlying schema for these field relationships.

Implement Real Pagination

Now let us implement real pagination with the `issues` list field. We need to implement a button to fetch more issues from the GraphQL API:

Environment Variables

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

```
const Repository = ({
  repository,
  onFetchMoreIssues,
}) => (
  <div>
    ...
    <ul>
      ...
    </ul>
    <hr />
    <button onClick={onFetchMoreIssues}>More</button>
  </div>
);
```

The handler for the button passes through all the components to reach the Repository component:

Environment Variables

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

```
const Organization = ({
  organization,
  errors,
  onFetchMoreIssues,
}) => {
  ...
  return (
    <div>
      <p>
        <strong>Issues from Organization:</strong>
        <a href={organization.url}>{organization.name}</a>
      </p>
      <Repository
        repository={organization.repository}
        onFetchMoreIssues={onFetchMoreIssues}
      />
```

```
    </div>
  );
};
```

Logic for the function is implemented in the App component as class method. It passes to the Organization component as well.

Environment Variables

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

```
class App extends Component {
  ...
  onFetchMoreIssues = () => {
    ...
  };

  render() {
    const { path, organization, errors } = this.state;
    return (
      <div>
        ...
        {organization ? (
          <Organization
            organization={organization}
            errors={errors}
            onFetchMoreIssues={this.onFetchMoreIssues}
          />
        ) : (
          <p>No information yet ...</p>
        )}
      </div>
    );
  }
}
```

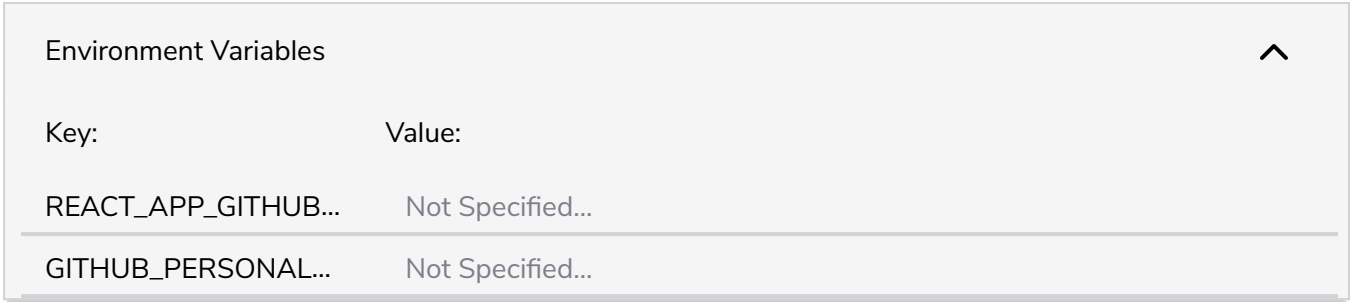
Before implementing the logic for it, there needs to be a way to identify the next page of the paginated list. To extend the inner fields of a list field with fields for meta information such as the `pageInfo` or the `totalCount` information, use `pageInfo` to define the next page on button-click. Also, the `totalCount` is only a nice way to see how many items are in the next list:

Environment Variables

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

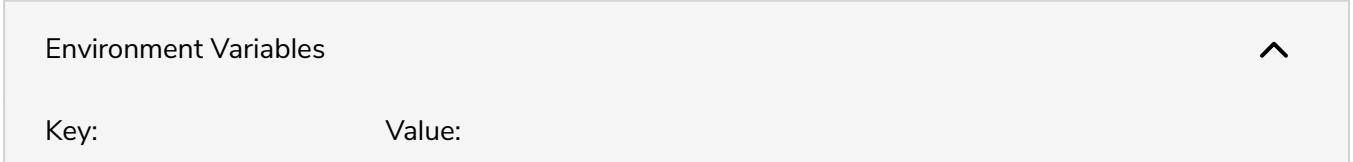
```
const GET_ISSUES_OF_REPOSITORY = `
  query ($organization: String!, $repository: String!) {
    organization(login: $organization) {

      name
      url
      repository(name: $repository) {
        ...
        issues(last: 5, states: [OPEN]) {
          edges {
            ...
          }
          totalCount
          pageInfo {
            endCursor
            hasNextPage
          }
        }
      }
    }
  }
`;
```

Now, you can use this information to fetch the next page of issues by providing the cursor as a variable to your query. The cursor, or the `after` argument, defines the starting point to fetch more items from the paginated list.




```
class App extends Component {
  ...
  onFetchMoreIssues = () => {
    const {
      endCursor,
    } = this.state.organization.repository.issues.pageInfo;
    this.onFetchFromGitHub(this.state.path, endCursor);
  };
  ...
}
```

The second argument wasn't introduced to the `onFetchFromGitHub()` class method yet. Let's see how that turns out.



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

```
const getIssuesOfRepository = (path, cursor) => {
  const [organization, repository] = path.split('/');

  return axiosGitHubGraphQL.post('', {
    query: GET_ISSUES_OF_REPOSITORY,
    variables: { organization, repository, cursor },
  });
};

class App extends Component {
  ...
  onFetchFromGitHub = (path, cursor) => {
    getIssuesOfRepository(path, cursor).then(queryResult =>
      this.setState(resolveIssuesQuery(queryResult, cursor)),
    );
  };

  ...
}
```

The argument is simply passed to the `getIssuesOfRepository()` function, which makes the GraphQL API request, and returns the promise with the query result. Check the other functions that call the `onFetchFromGitHub()` class method, and notice how they don't make use of the second argument, so the cursor parameter will be `undefined` when it's passed to the GraphQL API call. Either the query uses the cursor as argument to fetch the next page of a list, or it fetches the initial page of a list by having the cursor not defined at all:

Environment Variables

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

```
const GET_ISSUES_OF_REPOSITORY = `
  query (
    $organization: String!,
    $repository: String!,
    $cursor: String
  ) {
    organization(login: $organization) {
      name
      url
      repository(name: $repository) {
        ...
        issues(first: 5, after: $cursor, states: [OPEN]) {
          edges {
            ...
          }
```

```
          totalCount
          pageInfo {
            endCursor
            hasNextPage
          }
        }
      }
    }
  }
`;
```

In the previous template string, the `cursor` is passed as variable to the query and used as `after` argument for the list field. The variable is not enforced though, because there is no exclamation mark next to it, so it can be `undefined`. This happens for the initial page request for a paginated list, when you only want to fetch the first page. Further, the argument `last` has been changed to `first` for the `issues` list field, because there won't be another page after you fetched the last item in the initial request. Thus, you have to start with the first items of the list to fetch more items until you reach the end of the list.

That's it for fetching the next page of a paginated list with GraphQL in React, except one final step. Nothing updates the local state of the App component about a page of issues yet, so there are still only the issues from the initial request. You want to merge the old pages of issues with the new page of issues in the local state of the App component, while keeping the organization and repository information in the deeply nested state object intact. The perfect time for doing this is when the promise for the query resolves. You already extracted it as a function outside of the App component, so you can use this place to handle the incoming result and return a result with your own structure and information. Keep in mind that the incoming result can be an initial request when the App component mounts for the first time, or after a request to fetch more issues happens, such as when the "More" button is clicked.


```
const resolveIssuesQuery = (queryResult, cursor) => state => {
  const { data, errors } = queryResult.data;
```

```
  const { data, errors } = queryResult.data;

  if (!cursor) {
    return {
      organization: data.organization,
      errors,
    };
  }

  const { edges: oldIssues } = state.organization.repository.issues;
  const { edges: newIssues } = data.organization.repository.issues;
  const updatedIssues = [...oldIssues, ...newIssues];

  return {
    organization: {
      ...data.organization,
      repository: {
        ...data.organization.repository,
        issues: {
          ...data.organization.repository.issues,
          edges: updatedIssues,
        },
      },
    },
    errors,
  };
};
```

We will have to completely rewrite the function, because the update mechanism is more complex now. First, you passed the `cursor` as an argument to the function, which determines whether it was an initial query or a query to fetch another page of issues.

Second, if the `cursor` is `undefined`, the function can return early with the state object that encapsulates the plain query result, same as before. There is nothing to keep intact in the state object from before, because it is an initial request that happens when the App component mounts or when a user submits another request which should overwrite the old state anyway.

Third, if it is a fetch more query and the cursor is there, the old and new issues from the state and the query result get merged in an updated list of issues. In this case, a JavaScript de-structuring alias is used to make naming both issue lists more obvious.

Finally, the function returns the updated state object. Since it is a deeply nested object with multiple levels to update, use the JavaScript spread operator syntax to update each level with a new query result. Only the `edges` property should be updated with the merged list of issues.

Next, use the `hasNextPage` property from the `pageInfo` that you requested to show a “More” button (or not). If there are no more issues in the list, the button should disappear.

Environment Variables

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

```
const Repository = ({ repository, onFetchMoreIssues }) => (
  <div>
    ...
    <hr />
    {repository.issues.pageInfo.hasNextPage && (
      <button onClick={onFetchMoreIssues}>More</button>
    )}
  </div>
);
```

Now you’ve implemented pagination with GraphQL in React. For practice, try more arguments for your issues and reactions list fields on your own. Check the “Docs” sidebar in GraphQL to find out about arguments you can pass to list fields. Some arguments are generic, but have arguments that are specific to lists. These arguments should show you how finely-tuned requests can be with a GraphQL query.

Environment Variables

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

```
import React, { Component } from 'react';
import axios from 'axios';

const axiosGitHubGraphQL = axios.create({
  baseURL: 'https://api.github.com/graphql',
  headers: {
    Authorization: `bearer ${
      process.env.REACT_APP_GITHUB_PERSONAL_ACCESS_TOKEN
    }`,
  },
});

const resolveIssuesQuery = (queryResult, cursor) => state => {
  const { data, errors } = queryResult.data;
```

```
  if (!cursor) {
    return {
      organization: data.organization,

      errors,
    };
  }

  const { edges: oldIssues } = state.organization.repository.issues;
  const { edges: newIssues } = data.organization.repository.issues;
  const updatedIssues = [...oldIssues, ...newIssues];

  return {
    organization: {
      ...data.organization,
      repository: {
        ...data.organization.repository,
        issues: {
          ...data.organization.repository.issues,
          edges: updatedIssues,
        },
      },
    },
    errors,
  };
};


const TITLE = 'React GraphQL GitHub Client';
const getIssuesOfRepository = (path, cursor) => {
  const [organization, repository] = path.split('/');

  return axiosGitHubGraphQL.post('', {
    query: GET_ISSUES_OF_REPOSITORY,
    variables: { organization, repository, cursor },
  });
};
const GET_ISSUES_OF_REPOSITORY = `
  query ($organization: String!, $repository: String!,  $cursor: String) {
    organization(login: $organization) {
      name
      url
      repository(name: $repository) {
        name
        url
                                issues(first: 5, after: $cursor, states: [OPEN]) {
          edges {
            node {
              id
              title
              url
               reactions(last: 3) {
                edges {
                  node {
                    id
                    content
                  }
                }
              }
            }
          }
                                      totalCount
          pageInfo {
```

```
            endCursor
            hasNextPage
          }
        }

      }
    }
  }
`;
class App extends Component {

   state = {
    path: 'the-road-to-learn-react/the-road-to-learn-react',
  };

   componentDidMount() {
    // fetch data
        this.onFetchFromGitHub(this.state.path);
  }

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

  onSubmit = event => {
    // fetch data
         this.onFetchFromGitHub(this.state.path);
    event.preventDefault();
  };

          onFetchFromGitHub = (path, cursor) => {
    getIssuesOfRepository(path, cursor).then(queryResult =>
      this.setState(resolveIssuesQuery(queryResult, cursor)),
    );
  };

  onFetchMoreIssues = () => {
   const {
      endCursor,
    } = this.state.organization.repository.issues.pageInfo;

    this.onFetchFromGitHub(this.state.path, endCursor);
  };

  render() {
      const { path, organization, errors } = this.state;

    return (
      <div>
        <h1>{TITLE}</h1>
        <form onSubmit={this.onSubmit}>
          <label htmlFor="url">
            Show open issues for https://github.com/
          </label>
          <input
            id="url"
            type="text"
            value={path}
            onChange={this.onChange}
            style={{ width: '300px' }}
          />
          <button type="submit">Search</button>
```

```
        </form>

        <hr />

       {organization ? (
          <Organization organization={organization} errors={errors}
            onFetchMoreIssues={this.onFetchMoreIssues}
          />
        ) : (
          <p>No information yet ...</p>
        )}
      </div>
    );
  }
}

const Organization = ({ organization, errors,  onFetchMoreIssues, }) => {
  if (errors) {
    return (
      <p>
        <strong>Something went wrong:</strong>
        {errors.map(error => error.message).join(' ')}
      </p>
    );
  }

  return (
  <div>
    <p>
      <strong>Issues from Organization:</strong>
      <a href={organization.url}>{organization.name}</a>
    </p>
    <Repository repository={organization.repository}
       onFetchMoreIssues={onFetchMoreIssues} />
  </div>
);
};

const Repository = ({
  repository,
         onFetchMoreIssues,}) => (
  <div>
    <p>
      <strong>In Repository:</strong>
      <a href={repository.url}>{repository.name}</a>
    </p>
     <ul>
      {repository.issues.edges.map(issue => (
        <li key={issue.node.id}>
          <a href={issue.node.url}>{issue.node.title}</a>
          <ul>
            {issue.node.reactions.edges.map(reaction => (
              <li key={reaction.node.id}>{reaction.node.content}</li>
            ))}
          </ul>
        </li>
      ))}
    </ul>
     <hr />
                {repository.issues.pageInfo.hasNextPage && (
      <button onClick={onFetchMoreIssues}>More</button>
    )}
```

```
  </div>
);



export default App;
```

In the next lesson we will be learning about Mutation in GraphQL.