GraphQL Pagination

In this lesson, we will be introducing the GraphQL feature: pagination.

Why Do we Need Pagination?

This is where we return to the concept of **pagination** mentioned in the previous chapter. Imagine we have a list of repositories in our GitHub organization, but we only want to retrieve a few of them to display on our UI. It could take ages to fetch a list of repositories from a large organization. In GraphQL, you can request paginated data by providing arguments to a **list field**, such as an argument that specifies how many items you are expecting from the list.

Environment Variables	⌃

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

```
query OrganizationForLearningReact {
  organization(login: "the-road-to-learn-react") {
    name
    url
    repositories(first: 2) {
      edges {
        node {
          name
        }
      }
    }
  }
}
```

An argument, namely `first`, is passed to the `repositories` list field that specifies how many items from the list are expected in the result. The query shape doesn't need to follow the `edges` and `node` structure, but it's one from a few solutions to define paginated data structures and lists with GraphQL. Actually, it follows the interface description of Facebook's GraphQL client called Relay. GitHub followed this approach and adopted it for their own GraphQL pagination API.

Solution: Cursor Field

After executing the query, we should see two items from the list in the repositories field. However, we still need to figure out how to fetch the next two repositories in the list. The first result of the query is the first **page** of the paginated list, the second query result should be the second page. In the following code, we will see how the query structure for paginated data allows us to retrieve meta information to execute successive queries. This is done by allowing each edge to come with its own cursor field to identify its position in the list.

Environment Variables	^

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

```
query OrganizationForLearningReact {
  organization(login: "the-road-to-learn-react") {
    name
    url
    repositories(first: 2) {
      edges {
        node {
          name
        }
        cursor
      }
    }
  }
}
```

The result should be similar to the following:

Environment Variables

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

```json
{
  "data": {
    "organization": {
      "name": "The Road to learn React",
      "url": "https://github.com/the-road-to-learn-react",
      "repositories": {
        "edges": [
          {
            "node": {
              "name": "the-road-to-learn-react"
            },
            "cursor": "Y3Vyc29yOnYyOpHOA8awSw=="
          },
          {
            "node": {
              "name": "hackernews-client"
            },
            "cursor": "Y3Vyc29yOnYyOpHOBGhimw=="
          }
        ]
      }
    }
  }
}
```

Now, we can use the cursor of the first repository in the list to execute a second query. By using the `after` argument for the `repositories` list field, we can specify an entry point to retrieve our next page of paginated data. What would the result look like when executing the following query?

Environment Variables

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

```
query OrganizationForLearningReact {
  organization(login: "the-road-to-learn-react") {
    name
    url
    repositories(first: 2, after: "Y3Vyc29yOnYyOpHOA8awSw==") {
      edges {
        node {
          name
        }
        cursor
```

```
        }
      }


    }
  }
```

In the previous result, only the second item is retrieved along with a new third item. However, the first item is not retrieved because you have used its cursor as an `after` argument to retrieve all the items after it. Now you can imagine how to make successive queries for paginated lists:

- Execute the initial query without a cursor argument
- Then, execute every following query using a cursor argument in which the cursor will be that of the previous queries' last item's cursor.

To keep the query dynamic, we extract its arguments as variables. Afterward, you can use the query with a dynamic `cursor` argument by providing a variable for it. The `after` argument can be `undefined` to retrieve the first page. In conclusion, this would be everything we need to fetch pages of lists from one large list by using a feature called pagination. In short, we need a mandatory argument specifying how many items should be retrieved and an optional argument specifying the starting point for the list. In our case, this is the `after` argument.

There are also a couple of helpful ways to use meta information for your paginated list. Retrieving the `cursor` field for every repository may become redundant when using only the `cursor` of the last repository. One solution would be to remove the `cursor` field for an individual edge and instead add the `pageInfo` object with its `endCursor` and `hasNextPage` fields. You can also request the `totalCount` of the list.

Environment Variables   ∧

Key: Value:

REACT_APP_GITHUB...   Not Specified...

GITHUB_PERSONAL...   Not Specified...

```
query OrganizationForLearningReact {
  organization(login: "the-road-to-learn-react") {
    name
    url
    repositories(first: 2, after: "Y3Vyc29yOnYyOpHOA8awSw==") {
      totalCount
```

```
        edges {
          node {
            name
          }
        }
        pageInfo {
          endCursor
          hasNextPage
        }
      }
    }
  }
}
```

The `totalCount` field discloses the total number of items in the list, while the `pageInfo` field gives you information about two things:

- `endCursor` : Similar to the `cursor` field, `endCursor` can be used to retrieve the successive list. Except, in this case, we only need one meta field to perform it. The cursor of the last list item is sufficient to request the next page of the list.

- `hasNextPage` : This gives us information about whether or not there is a next page to retrieve from the GraphQL API because sometimes we have already fetched the last page from our server. For applications that use infinite scrolling to load more pages when scrolling lists, you can stop fetching when there are no more pages available.

This meta information completes the pagination implementation. Information is made accessible using the GraphQL API to implement paginated lists and infinite scroll.

Note: If you use a different GraphQL API for pagination, it might use different naming conventions for the fields, exclude meta information, or employ different mechanisms altogether. The lesson above was specifically regarding pagination implemented using GitHub's GraphQL API.

For more clarity, read more about pagination in GraphQL.

In the next chapter, we will move on to combining React with GraphQL.