# **Implementing Pagination in Apollo Client**

**Pagination** is a technique used to divide a large set of data into smaller, more manageable chunks or pages. It enhances the performance and user experience by loading only a subset of data at a time instead of fetching and displaying all the data at once.

**Why Use Pagination**

1. **Performance**: Loading large datasets can be resource-intensive and slow. Pagination improves performance by fetching data in smaller chunks.
2. **User Experience**: Displaying all data at once can overwhelm users. Pagination provides a more manageable way to navigate data.
3. **Scalability**: It allows applications to handle large datasets efficiently by loading data on demand.
4. **Network Efficiency**: Reduces the amount of data transferred over the network, which is especially important for mobile and low-bandwidth connections.

**Where to Use Pagination**

1. **Data Tables**: When displaying large lists of data in tables, such as user lists, product catalogs, etc.
2. **Feed or Timeline**: Social media feeds, news feeds, or any infinite scrolling lists.
3. **Search Results**: When displaying search results that can potentially return a large number of items.
4. **API Responses**: Any API response that returns a large dataset can benefit from pagination.

**Detailed Explanation and Implementation**

**Scenario: Paginating a List of Employees**

We'll implement pagination for fetching and displaying a list of employees. The GraphQL server should support pagination, typically using arguments like first and after for cursor-based pagination or limit and offset for offset-based pagination.

**Step 1: Define the GraphQL Query with Pagination**

Assume our GraphQL API uses cursor-based pagination.

1. **GraphQL Query**

# On the server-side

type Query {

employees(first: Int, after: String): EmployeeConnection

}

type EmployeeConnection {

edges: [EmployeeEdge]

pageInfo: PageInfo

}

type EmployeeEdge {

cursor: String

node: Employee

}

type PageInfo {

endCursor: String

hasNextPage: Boolean

}

// src/queries/employeeQueries.js

import { gql } from '@apollo/client';

export const GET\_EMPLOYEES = gql`

query GetEmployees($first: Int, $after: String) {

employees(first: $first, after: $after) {

edges {

cursor

node {

id

name

email

designation {

title

}

department {

name

}

manager {

name

}

}

}

pageInfo {

endCursor

hasNextPage

}

}

}

`;

**Step 2: Use the Query in a Component**

1. **EmployeeList Component**

// src/components/EmployeeList.js

import React, { useState } from 'react';

import { useQuery } from '@apollo/client';

import { GET\_EMPLOYEES } from '../queries/employeeQueries';

const PAGE\_SIZE = 10;

const EmployeeList = () => {

const [cursor, setCursor] = useState(null);

const { loading, error, data, fetchMore } = useQuery(GET\_EMPLOYEES, {

variables: { first: PAGE\_SIZE, after: cursor },

});

const loadMore = () => {

fetchMore({

variables: {

first: PAGE\_SIZE,

after: data.employees.pageInfo.endCursor,

},

updateQuery: (previousResult, { fetchMoreResult }) => {

const newEdges = fetchMoreResult.employees.edges;

const pageInfo = fetchMoreResult.employees.pageInfo;

return newEdges.length

? {

employees: {

\_\_typename: previousResult.employees.\_\_typename,

edges: [...previousResult.employees.edges, ...newEdges],

pageInfo,

},

}

: previousResult;

},

});

};

if (loading) return <p>Loading...</p>;

if (error) return <p>Error: {error.message}</p>;

return (

<div>

<h2>Employees</h2>

<ul>

{data.employees.edges.map(({ node }) => (

<li key={node.id}>

{node.name} - {node.email}

</li>

))}

</ul>

{data.employees.pageInfo.hasNextPage && (

<button onClick={loadMore}>Load More</button>

)}

</div>

);

};

export default EmployeeList;

**Explanation**

1. **GraphQL Schema**:
   * The GraphQL schema defines a Query type for fetching employees with pagination support.
   * EmployeeConnection, EmployeeEdge, and PageInfo types are used to structure the paginated response.
2. **Client-Side Query**:
   * GET\_EMPLOYEES query takes first (number of items to fetch) and after (cursor) as arguments.
   * The query returns edges (list of employees) and pageInfo (pagination information).
3. **Component State**:
   * cursor: State to keep track of the current cursor for pagination.
   * PAGE\_SIZE: Constant to define the number of items to fetch per page.
4. **useQuery Hook**:
   * The useQuery hook is used to execute the GET\_EMPLOYEES query with initial pagination variables.
5. **fetchMore Function**:
   * fetchMore is used to load more data when the "Load More" button is clicked.
   * updateQuery function is used to merge the new data with the existing data in the cache.
6. **Handling Loading and Error States**:
   * The component handles loading and error states using the values returned by the useQuery hook.
7. **Rendering Data**:
   * The employees are rendered in a list, and a "Load More" button is displayed if there are more pages to load.

**Conclusion**

Pagination is essential for handling large datasets efficiently and providing a better user experience. It reduces the amount of data loaded at once, improves performance, and makes navigation easier for users. By implementing pagination using Apollo Client and GraphQL, you can ensure that your application can handle large datasets gracefully. This detailed explanation and implementation guide demonstrate how to set up cursor-based pagination in a React application using Apollo Client.