# **Working with Fragments in GraphQL**

**GraphQL Fragments** are reusable units of query logic that can be used to avoid repetition and to ensure consistency across queries and mutations. They allow you to define a piece of a query that can be shared and reused in multiple queries, mutations, or subscriptions.

**Why Use GraphQL Fragments**

1. **Reusability**: Fragments allow you to define reusable pieces of query logic that can be included in multiple operations. This is especially useful when you have several queries or mutations that need to fetch the same set of fields.
2. **Maintainability**: With fragments, you can define a common set of fields in one place, reducing the risk of errors and inconsistencies. If you need to update the set of fields, you can do it in one place, and all operations that use the fragment will be updated automatically.
3. **Readability**: Fragments can make your queries and mutations cleaner and easier to read by abstracting repetitive parts of the query into a separate, named fragment.
4. **Modularity**: They help in breaking down complex queries into smaller, more manageable parts, making your queries easier to understand and maintain.

**When to Use GraphQL Fragments**

1. **Repeated Field Sets**: When multiple queries or mutations need to fetch the same fields, use fragments to avoid duplication.
2. **Consistent Data Requirements**: When you want to ensure that all parts of your application that fetch a certain type of data request the same fields.
3. **Modular Queries**: When you have complex queries that can be broken down into smaller, reusable parts.
4. **API Evolution**: When the schema evolves and you need to update the fields being fetched, using fragments ensures you only need to update the fragment definition.

**Detailed Explanation and Implementation**

Let's use a scenario where we have an application that manages employee data. We'll use fragments to fetch common employee fields across different queries and mutations.

**Step 1: Define the Fragment**

Create a fragment that defines the common fields for an employee.

// src/fragments/employeeFragments.js

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

export const EMPLOYEE\_FIELDS = gql`

fragment EmployeeFields on Employee {

id

name

email

designation {

title

}

department {

name

}

manager {

name

}

}

`;

**Step 2: Use the Fragment in Queries**

1. **Get Employees Query**

// src/queries/employeeQueries.js

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

import { EMPLOYEE\_FIELDS } from '../fragments/employeeFragments';

export const GET\_EMPLOYEES = gql`

query GetEmployees {

employees {

...EmployeeFields

}

}

${EMPLOYEE\_FIELDS}

`;

export const GET\_EMPLOYEE = gql`

query GetEmployee($id: Int!) {

employee(id: $id) {

...EmployeeFields

}

}

${EMPLOYEE\_FIELDS}

`;

**Step 3: Use the Fragment in Mutations**

1. **Add Employee Mutation**

// src/mutations/employeeMutations.js

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

import { EMPLOYEE\_FIELDS } from '../fragments/employeeFragments';

export const ADD\_EMPLOYEE = gql`

mutation AddEmployee($name: String!, $email: String!, $designationId: Int!, $departmentId: Int!, $managerId: Int) {

addEmployee(name: $name, email: $email, designationId: $designationId, departmentId: $departmentId, managerId: $managerId) {

...EmployeeFields

}

}

${EMPLOYEE\_FIELDS}

`;

export const UPDATE\_EMPLOYEE = gql`

mutation UpdateEmployee($id: Int!, $name: String!, $email: String!, $designationId: Int!, $departmentId: Int!, $managerId: Int) {

updateEmployee(id: $id, name: $name, email: $email, designationId: $designationId, departmentId: $departmentId, managerId: $managerId) {

...EmployeeFields

}

}

${EMPLOYEE\_FIELDS}

`;

**Step 4: Use the Queries and Mutations in Components**

1. **EmployeeList Component**

// src/components/EmployeeList.js

import React from 'react';

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

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

import { DELETE\_EMPLOYEE } from '../mutations/employeeMutations';

const EmployeeList = () => {

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

const [deleteEmployee] = useMutation(DELETE\_EMPLOYEE, {

update(cache, { data: { deleteEmployee } }) {

const { employees } = cache.readQuery({ query: GET\_EMPLOYEES });

cache.writeQuery({

query: GET\_EMPLOYEES,

data: { employees: employees.filter(emp => emp.id !== deleteEmployee.id) },

});

},

onError: (error) => {

console.error('Error deleting employee:', error.message);

}

});

const handleDelete = async (id) => {

try {

await deleteEmployee({ variables: { id } });

} catch (error) {

console.error('Error:', error);

}

};

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

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

return (

<div>

<h2>Employees</h2>

<ul>

{data.employees.map(employee => (

<li key={employee.id}>

{employee.name} - {employee.email}

<button onClick={() => handleDelete(employee.id)}>Delete</button>

</li>

))}

</ul>

</div>

);

};

export default EmployeeList;

1. **AddEmployeeForm Component**

// src/components/AddEmployeeForm.js

import React from 'react';

import { useForm } from 'react-hook-form';

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

import { ADD\_EMPLOYEE } from '../mutations/employeeMutations';

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

const AddEmployeeForm = () => {

const { register, handleSubmit, reset } = useForm();

const [addEmployee, { loading, error }] = useMutation(ADD\_EMPLOYEE, {

update(cache, { data: { addEmployee } }) {

const { employees } = cache.readQuery({ query: GET\_EMPLOYEES });

cache.writeQuery({

query: GET\_EMPLOYEES,

data: { employees: [...employees, addEmployee] },

});

},

onError: (error) => {

console.error('Error adding employee:', error.message);

}

});

const onSubmit = async (formData) => {

try {

await addEmployee({ variables: { ...formData, designationId: parseInt(formData.designationId), departmentId: parseInt(formData.departmentId), managerId: formData.managerId ? parseInt(formData.managerId) : null } });

reset(); // Reset the form after successful submission

} catch (error) {

console.error('Error:', error);

}

};

return (

<div>

<h2>Add Employee</h2>

<form onSubmit={handleSubmit(onSubmit)}>

<div className="form-group">

<label>Name:</label>

<input className="form-control" {...register('name', { required: true })} />

</div>

<div className="form-group">

<label>Email:</label>

<input className="form-control" type="email" {...register('email', { required: true })} />

</div>

<div className="form-group">

<label>Designation ID:</label>

<input className="form-control" type="number" {...register('designationId', { required: true })} />

</div>

<div className="form-group">

<label>Department ID:</label>

<input className="form-control" type="number" {...register('departmentId', { required: true })} />

</div>

<div className="form-group">

<label>Manager ID:</label>

<input className="form-control" type="number" {...register('managerId')} />

</div>

<button type="submit" className="btn btn-primary mt-3" disabled={loading}>

{loading ? 'Adding...' : 'Add Employee'}

</button>

{error && <p>Error adding employee: {error.message}</p>}

</form>

</div>

);

};

export default AddEmployeeForm;

**Detailed Explanation**

**Step 1: Define the Fragment**

* **Fragment Definition**: The EMPLOYEE\_FIELDS fragment is defined to include the common fields for the Employee type. This fragment can be reused in multiple queries and mutations.

**Step 2: Use the Fragment in Queries**

* **Queries**: The GET\_EMPLOYEES and GET\_EMPLOYEE queries use the EMPLOYEE\_FIELDS fragment to ensure they fetch the same set of fields for employees. The fragment is included in the query using the ...EmployeeFields syntax, and the fragment is appended to the query using ${EMPLOYEE\_FIELDS}.

**Step 3: Use the Fragment in Mutations**

* **Mutations**: The ADD\_EMPLOYEE and UPDATE\_EMPLOYEE mutations use the EMPLOYEE\_FIELDS fragment to ensure they return the same set of fields after performing the mutation. This allows the client to update its cache with the latest data, using a consistent structure.

**Step 4: Use the Queries and Mutations in Components**

* **Components**: The EmployeeList and AddEmployeeForm components use the queries and mutations that include the fragment. This ensures that the components receive a consistent set of fields for employees, regardless of whether the data is fetched by a query or returned by a mutation.

**Conclusion**

Fragments in GraphQL are a powerful tool for creating reusable, maintainable, and modular query logic. They help avoid repetition, ensure consistency, and make your queries and mutations more readable. By defining fragments for common field sets and using them across your operations, you can simplify your GraphQL code and improve the maintainability of your application. This detailed implementation demonstrates how to define and use fragments in queries and mutations, ensuring a consistent and efficient way to manage data fetching and updating in a React application using Apollo Client.