# **Updating Cache Query Results: Polling and Refetching**

When building applications that interact with a GraphQL backend using Apollo Client, you often need to keep your UI in sync with the server. This can be achieved through **polling** and **refetching** mechanisms to update the cache query results. Let's delve into why updating cache results is needed, the scenarios where it can be used, and the implementation details.

**Why It Is Needed**

1. **Real-Time Data Updates**: In many applications, data changes frequently, and you want the UI to reflect these changes in real-time. For example, dashboards, social media feeds, stock market updates, etc.
2. **User Interactions**: Users might perform actions that change the data (e.g., adding, updating, or deleting items), and you want these changes to be reflected immediately in the UI.
3. **Consistency**: Keeping the UI consistent with the server state ensures a seamless user experience. Users should always see the latest data without needing to refresh the page manually.

**Scenarios Where It Can Be Used**

1. **Live Data Feeds**: Applications displaying live data, such as news feeds, sports scores, or stock prices, benefit from polling and refetching to ensure the displayed information is current.
2. **Collaborative Tools**: Tools where multiple users can interact with the same data in real-time, such as task managers or document editors.
3. **Dynamic Dashboards**: Dashboards that aggregate and display real-time statistics or metrics require frequent updates to present accurate information.
4. **Notification Systems**: Applications that provide real-time notifications about events or status changes.

## **Detailed Explanation and Implementation**

**Polling**

Polling is a mechanism where Apollo Client repeatedly fetches data at a specified interval. This is useful for real-time updates without needing to set up subscriptions.

**Example Implementation**

1. **Define the Query**

// src/queries/employeeQueries.js

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

export const GET\_EMPLOYEES = gql`

query GetEmployees {

employees {

id

name

email

designation {

title

}

department {

name

}

manager {

name

}

}

}

`;

1. **Use Polling in a Component**

// src/components/EmployeeList.js

import React from 'react';

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

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

const EmployeeList = () => {

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

pollInterval: 5000, // Poll every 5 seconds

});

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}

</li>

))}

</ul>

</div>

);

};

export default EmployeeList;

**Explanation**:

* **pollInterval**: This property specifies the interval (in milliseconds) at which the query should be re-fetched. Here, it is set to 5000 milliseconds (5 seconds).

**Refetching**

Refetching is manually triggering a query to fetch the latest data. This is useful in scenarios where you need to update the UI in response to specific user actions.

**Example Implementation**

1. **Use Refetching in a Component**

// src/components/EmployeeListWithRefetch.js

import React from 'react';

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

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

const EmployeeListWithRefetch = () => {

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

const handleRefetch = () => {

refetch(); // Manually trigger refetch

};

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

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

return (

<div>

<h2>Employees</h2>

<button onClick={handleRefetch}>Refresh Data</button>

<ul>

{data.employees.map(employee => (

<li key={employee.id}>

{employee.name} - {employee.email}

</li>

))}

</ul>

</div>

);

};

export default EmployeeListWithRefetch;

**Explanation**:

* **refetch**: This function is returned by useQuery and can be called to manually refetch the query data.

**Detailed Use Case Scenario**

Imagine a dashboard application that displays real-time data about employees, including their status, which can change frequently. You want the data to be updated every 10 seconds automatically, but you also want to provide a button for users to manually refresh the data whenever they want.

1. **Set up the Query with Polling**

const { loading, error, data, refetch, startPolling, stopPolling } = useQuery(GET\_EMPLOYEES, {

pollInterval: 10000, // Poll every 10 seconds

});

1. **Provide a Manual Refresh Option**

<button onClick={() => refetch()}>Refresh Data</button>

1. **Render the Data**

<ul>

{data.employees.map(employee => (

<li key={employee.id}>

{employee.name} - {employee.email} - Status: {employee.status}

</li>

))}

</ul>

**Complete Component**

import React from 'react';

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

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

const EmployeeDashboard = () => {

const { loading, error, data, refetch, startPolling, stopPolling } = useQuery(GET\_EMPLOYEES, {

pollInterval: 10000, // Poll every 10 seconds

});

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

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

return (

<div>

<h2>Employee Dashboard</h2>

<button onClick={() => refetch()}>Refresh Data</button>

<ul>

{data.employees.map(employee => (

<li key={employee.id}>

{employee.name} - {employee.email} - Status: {employee.status}

</li>

))}

</ul>

</div>

);

};

export default EmployeeDashboard;

**Conclusion**

Updating cache query results through polling and refetching is essential for keeping your application's UI in sync with the server, providing real-time data updates, and ensuring a consistent user experience. Polling automatically fetches data at regular intervals, which is useful for live updates, while refetching allows manual data refresh in response to user actions. By implementing these techniques, you can ensure that your application always displays the most current data.