## useQuery Hook

In this React GraphQL code, when is the query GetUserById invoked? Is it at <p>{getUserByIdData.getUserById.name}</p>?

const GET\_USER\_BY\_ID = gql`

query GetUserById($id: ID!) {

getUserById(id: $id) {

name

}

}

`;

const {

data: getUserByIdData, loading: getUserByIdLoading

} = useQuery(GET\_USER\_BY\_ID,

{

variables: { id: "2" },

}

);

return (

<p>{getUserByIdData.getUserById.name}</p>

)

The query GetUserById is invoked **as soon as the component renders** and the useQuery hook is called, not specifically when <p>{getUserByIdData.getUserById.name}</p> is rendered. The invocation process can be broken down as follows:

**Step-by-Step Explanation:**

1. **useQuery Hook Initialization**:
   * When the component renders, the useQuery hook is called with the GET\_USER\_BY\_ID query and the specified variables (id: "2").
   * This causes Apollo Client to send the GetUserById query to the GraphQL server automatically unless caching satisfies the request.

Code:

const { data: getUserByIdData, loading: getUserByIdLoading } = useQuery(GET\_USER\_BY\_ID, {

variables: { id: "2" },

});

1. **Query Execution**:
   * Apollo Client checks its cache for the GetUserById query with id: "2".
   * If the data is not in the cache or if the cache policy requires a fresh fetch (e.g., cache-and-network), Apollo sends the query to the GraphQL server.
2. **Data Fetching**:
   * The server processes the GetUserById query and returns a response, which Apollo Client updates in its cache.
   * The useQuery hook then updates its data and loading properties.
3. **Component Re-Render**:
   * When data is updated, the component re-renders to display the new data.
   * At this point, <p>{getUserByIdData.getUserById.name}</p> uses the fetched name value.

**Key Insights:**

* **Query Execution Timing**:
  + The query is invoked as soon as the useQuery hook runs, not when the JSX <p> tag referencing getUserByIdData.getUserById.name is evaluated.
* **Conditional Rendering During Loading**:
  + Since the query takes time to fetch data, the loading state (getUserByIdLoading) helps manage what is displayed before the data arrives.
  + Example:
  + if (getUserByIdLoading) {
  + return <p>Loading...</p>;
  + }
* **Error Handling**:
  + If the query fails, you might handle it using Apollo's error property from useQuery.
  + const { data, loading, error } = useQuery(GET\_USER\_BY\_ID, { variables: { id: "2" } });
  + if (error) return <p>Error: {error.message}</p>;

**When Is <p> Rendered?**

* Initially, the <p> tag will try to render but may throw an error if getUserByIdData or getUserById is undefined during the first render.
* To prevent this, ensure that data exists before accessing nested fields:
* return (
* <p>{getUserByIdData?.getUserById?.name || "Loading..."}</p>
* );

**Conclusion:**

The GetUserById query is invoked when the useQuery hook is executed, which occurs immediately when the component renders. The <p> tag displays the name field after the data is fetched and the component re-renders with the updated data.

## Component’s render invoking useQueries

The rendering of the component invokes the query GetUserById because of the **useQuery hook** from Apollo Client. Here's a detailed explanation of how this happens:

**1. useQuery Hook's Functionality**

The useQuery hook in Apollo Client is a declarative way to fetch data when a component is rendered. It automatically handles:

* Query execution.
* Tracking the loading and error states.
* Re-fetching data when necessary.

When you call useQuery inside a functional component, it triggers the GraphQL query execution process.

**2. The Component's Render Triggers the Hook**

When the component renders:

1. **The useQuery Hook Executes**:
   * The useQuery hook is called as part of the component's render logic.
   * It initializes a data-fetching process by:
     + Checking Apollo Client's cache to see if the requested data (GetUserById with id: "2") is already available.
     + Sending a query to the GraphQL server if the cache does not satisfy the query or if the policy requires a fresh fetch.

Code:

const { data, loading, error } = useQuery(GET\_USER\_BY\_ID, {

variables: { id: "2" },

});

1. **Apollo Client Handles Query Execution**:
   * If the data is not cached or a fresh fetch is needed:
     + Apollo Client sends the query to the GraphQL server over the network.
     + The server processes the query, fetches the required data, and returns it as a response.
   * The useQuery hook listens for the response and updates its state (data, loading, error).
2. **React Re-Render**:
   * When useQuery updates the data or loading state, React re-renders the component.
   * During this re-render, the updated data is available for rendering.

**3. Query Execution Process**

The entire process looks like this:

1. **First Render**:
   * The useQuery hook is executed.
   * Initially, data is undefined, and loading is true.
2. **Apollo Client Fetches Data**:
   * If required, Apollo sends the GetUserById query to the GraphQL server.
   * The server processes the query and returns the requested data.
3. **State Update and Re-Render**:
   * The useQuery hook receives the response and updates its internal state (data and loading).
   * React re-renders the component with the updated data.
4. **Render Updated Data**:
   * On the next render, the component can access data and display the fetched information.

**4. Example Flow**

Consider this code:

const GET\_USER\_BY\_ID = gql`

query GetUserById($id: ID!) {

getUserById(id: $id) {

name

}

}

`;

const UserComponent = () => {

const { data, loading, error } = useQuery(GET\_USER\_BY\_ID, {

variables: { id: "2" },

});

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

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

return <p>{data.getUserById.name}</p>;

};

**How It Works**:

1. **Initial Render**:
   * useQuery is invoked, and the query execution begins.
   * loading is true, so <p>Loading...</p> is displayed.
2. **Data Fetching**:
   * Apollo Client fetches data from the server (if not cached).
3. **State Update**:
   * Once the data is fetched, data is updated, and loading becomes false.
4. **Re-Render**:
   * The component re-renders, now displaying <p>{data.getUserById.name}</p>.

**Conclusion**

The rendering of the component invokes the GetUserById query indirectly because the useQuery hook is executed during the render phase. The hook triggers Apollo Client's data-fetching mechanism, retrieves the required data, and updates the component upon completion.

## Does the Apollo client check all useQuery hooks when the component renders for the first time?

Yes, the Apollo Client processes all useQuery hooks invoked during the **initial render** of a component. Each useQuery hook operates independently, and the Apollo Client handles them in the following manner:

**How Apollo Client Handles Multiple useQuery Hooks**

1. **Initialization During Render**:
   * When a component renders for the first time, each useQuery hook is executed.
   * Each hook sends its GraphQL query to Apollo Client, providing details such as:
     + The query document.
     + Any variables required for the query.
     + Cache policies (e.g., cache-first, network-only).
2. **Apollo Client Query Processing**:
   * Apollo Client checks its **cache** for each query's data based on the specified caching policy.
     + If the data exists in the cache and satisfies the policy, it is returned immediately without a network request.
     + If the data is missing or a fresh fetch is required, Apollo Client sends the query to the GraphQL server.
3. **Asynchronous Data Fetching**:
   * Queries requiring a network request are processed independently.
   * Apollo sends these queries to the server concurrently, not sequentially, ensuring efficient data fetching.
4. **Updates and Re-Renders**:
   * As each query's data becomes available (either from the cache or the server), Apollo updates the corresponding useQuery hook.
   * React re-renders the component as the data for each hook is updated.

**Example: Multiple useQuery Hooks**

const GET\_USER = gql`

query GetUser($id: ID!) {

user(id: $id) {

id

name

}

}

`;

const GET\_POSTS = gql`

query GetPosts($userId: ID!) {

posts(userId: $userId) {

id

title

}

}

`;

const MyComponent = () => {

const { data: userData, loading: userLoading } = useQuery(GET\_USER, {

variables: { id: "1" },

});

const { data: postsData, loading: postsLoading } = useQuery(GET\_POSTS, {

variables: { userId: "1" },

});

if (userLoading || postsLoading) return <p>Loading...</p>;

return (

<div>

<h1>User: {userData.user.name}</h1>

<ul>

{postsData.posts.map((post) => (

<li key={post.id}>{post.title}</li>

))}

</ul>

</div>

);

};

**Execution in the Example**

1. **First Render**:
   * Both useQuery hooks (GET\_USER and GET\_POSTS) are executed.
   * Apollo Client processes these queries concurrently:
     + It checks the cache for GET\_USER and GET\_POSTS.
     + If the cache doesn't have the data or requires a fresh fetch, network requests are sent for both queries.
2. **Data Arrival**:
   * When the server responds with the GET\_USER data, the userData state is updated, and the component re-renders.
   * Similarly, when the GET\_POSTS data arrives, the postsData state updates, triggering another re-render.
3. **Final Render**:
   * When both queries have completed and their data is available, the component displays the user's name and posts.

**Key Insights**

* **Concurrency**:
  + Multiple useQuery hooks can be executed in parallel, and Apollo Client manages them independently.
* **Cache Efficiency**:
  + If the data for any query is already in the cache, Apollo Client avoids unnecessary network requests.
* **Independent Lifecycle**:
  + Each useQuery hook has its own lifecycle for fetching, updating, and managing data.

**Conclusion**

Yes, Apollo Client checks and processes all useQuery hooks during the first render of a component. Each hook's query is handled independently, ensuring efficient data fetching and cache management. If multiple queries are involved, they are executed concurrently, minimizing delays.

## Can useQuery hooks be triggered by user actions?

Absolutely, `useQuery` hooks can indeed be triggered by user actions. Here's how you can achieve this:

1. \*\*Conditional Fetch\*\*: Use the `skip` option to conditionally fetch data. You can set up a state variable that changes based on user actions.

2. \*\*Refetch\*\*: You can manually trigger a refetch using the `refetch` function provided by `useQuery`.

Here's an example using both approaches:

```javascript

import { useState } from 'react';

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

const MY\_QUERY = gql`

query MyQuery($variable: String!) {

myField(variable: $variable) {

data

}

}

`;

function MyComponent() {

const [shouldFetch, setShouldFetch] = useState(false);

const { data, loading, error, refetch } = useQuery(MY\_QUERY, {

variables: { variable: 'value' },

skip: !shouldFetch, // Condition to skip or not

});

const handleButtonClick = () => {

setShouldFetch(true); // This will trigger the query

};

const handleRefetch = () => {

refetch({ variable: 'newValue' }); // Manually refetch with new variable

};

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

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

return (

<div>

<button onClick={handleButtonClick}>Fetch Data</button>

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

<div>Data: {data?.myField.data}</div>

</div>

);

}

```

In this example:

- The `skip` option prevents the initial query from running until the user clicks the "Fetch Data" button, changing `shouldFetch` to true.

- The `refetch` function allows you to manually trigger the query again with different variables when the "Refetch Data" button is clicked.