TanStack Query

1. Problem Statement

Modern web applications often rely on fetching, caching, and updating data from APIs. Developers face challenges such as managing loading and error states, keeping data in sync with the server, handling pagination, and minimizing unnecessary API calls. Without robust tools, applications can become slow, unreliable, and difficult to maintain.

For example, consider a project management dashboard that displays tasks fetched from a backend API. Users expect to see up-to-date task lists, smooth loading experiences, and immediate feedback when adding or updating tasks. Traditional approaches using `fetch` or global state managers like Redux often lead to complex code, manual caching, and inconsistent data.

TanStack Query addresses these challenges by providing a powerful, intuitive solution for managing server state in web applications.

---

2. Learning Objectives

By the end of this tutorial, you will be able to:

- **Understand the role and benefits of TanStack Query in modern web development.**
- **Fetch data from APIs efficiently using the `useQuery` hook.**
- **Modify server data with the `useMutation` hook.**
- **Leverage automatic caching, background updates, and error handling.**
- **Integrate TanStack Query with your React application.**
- **Use TanStack Query DevTools for debugging and monitoring.**

---

3. Concept Introduction with Analogy

Imagine your application as a busy office manager who needs to keep track of a large team's tasks. The manager must:

- **Fetch the latest task list** from the team's shared system (the server).
- **Update the list** when new tasks are added or completed.
- **Handle errors** if the system is unavailable.
- **Cache frequently accessed information** to avoid constantly asking for updates.

TanStack Query is like an assistant who automates all these tasks, ensuring the manager always has the most current information, handles errors gracefully, and keeps the office running smoothly.

---

4. Technical Deep Dive

TanStack Query (formerly React Query) is a library for managing server state in React and other modern frameworks. It provides:

- **Automatic Caching:** Data fetched from APIs is cached and reused across components, reducing unnecessary network requests.
- **Background Updates:** TanStack Query automatically refreshes stale data in the background, keeping the UI up-to-date.
- **Query Deduplication:** Multiple requests for the same data are combined into a single API call.
- **Optimistic Updates:** The UI can be updated immediately when a mutation is triggered, with automatic rollback if the server request fails.
- **Error Handling:** Built-in support for loading and error states, making it easy to display feedback to users.
- **TypeScript Support:** Fully typed API for better developer experience and fewer runtime errors.
- **DevTools:** A graphical interface for monitoring and debugging queries and mutations.

---

5. Step-by-Step Data Modeling & Code Walkthrough

1. Installation

Install TanStack Query in your React project:

```
npm install @tanstack/react-query
```

2. Setting Up QueryClient

Wrap your application with `QueryClientProvider`:

```
// main.jsx
import { QueryClient, QueryClientProvider } from '@tanstack/react-query';
import App from './App';

const queryClient = new QueryClient();

function Root() {
  return (
    <QueryClientProvider client={queryClient}>
      <App />
    </QueryClientProvider>
  );
}

export default Root;
```

3. Fetching Data with `useQuery`

Create a component to fetch and display a list of users:

```
// UsersList.jsx
import { useQuery } from '@tanstack/react-query';

async function fetchUsers() {
  const response = await fetch('https://jsonplaceholder.typicode.com/users');
  if (!response.ok) throw new Error('Network response was not ok');
  return response.json();
}

function UsersList() {
  const { data, error, isLoading } = useQuery(['users'], fetchUsers);

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

```
  return (
    <ul>
      {data.map(user => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
}

export default UsersList;
```

4. Modifying Data with `useMutation`

Add a component to create a new user:

```
// AddUser.jsx
import { useMutation, useQueryClient } from '@tanstack/react-query';

async function addUser(newUser) {
  const response = await fetch('https://jsonplaceholder.typicode.com/users', {
    method: 'POST',
    body: JSON.stringify(newUser),
  });
  return response.json();
}

function AddUser() {
  const queryClient = useQueryClient();
  const mutation = useMutation({
    mutationFn: addUser,
    onSuccess: () => {
      // Refresh the users list after adding a new user
      queryClient.invalidateQueries(['users']);
    },
  });

  const handleAddUser = () => {
    mutation.mutate({ name: 'John Doe' });
  };

  return (
    <button onClick={handleAddUser} disabled={mutation.isLoading}>
      Add User
    </button>
  );
}
```

5. Enabling DevTools

Install and add DevTools to your application:

```
npm install @tanstack/react-query-devtools


import { ReactQueryDevtools } from '@tanstack/react-query-devtools';

function App() {
  return (
    <>
      <UsersList />
      <AddUser />
      <ReactQueryDevtools initialIsOpen={false} />
    </>
  );
}
```

---

6. Challenge

Your Turn!

- **Fetch a list of posts** from `https://jsonplaceholder.typicode.com/posts` using `useQuery`.
- **Add a form** to create a new post using `useMutation`.
- **Display loading and error states** for both queries and mutations.
- **Use DevTools** to monitor active queries and mutations.
- **Bonus:** Implement optimistic updates for the post creation form.

---

7. Common Pitfalls & Best Practices

Pitfall	Best Practice
Not invalidating queries after mutations	Always invalidate queries to keep data in sync
Ignoring error and loading states	Display feedback to users during loading and errors
Overfetching or underfetching data	Use query keys and `staleTime` to optimize caching
Not using DevTools for debugging	Leverage DevTools to monitor and troubleshoot queries
Mixing client and server state unnecessarily	Use TanStack Query for server state, not client state

---

8. Optional: Programmer's Workflow Checklist

- **Install TanStack Query and DevTools.**
- **Wrap your app with `QueryClientProvider`.**
- **Fetch data with `useQuery` and display loading/error states.**
- **Modify data with `useMutation` and invalidate queries on success.**
- **Add DevTools to monitor and debug queries and mutations.**
- **Test with different API endpoints and edge cases.**
- **Optimize cache settings and query keys for your use case.**
- **Document your data fetching and mutation logic for your team.**