React Query / SWR: Server State, Query Keys, Optimistic Updates, Cache Invalidation

1. Problem Statement

Modern web applications frequently need to fetch, update, and manage data from remote APIs. Managing server state—data that originates from and is stored on the server—introduces challenges such as keeping the UI in sync, handling loading and error states, avoiding redundant API calls, and ensuring data consistency across multiple components and user sessions. Traditional approaches using global state managers or manual `fetch` logic can become complex, error-prone, and difficult to maintain as applications grow.

For example, consider a dashboard that displays user data fetched from a backend. Users expect to see up-to-date information, smooth loading experiences, and immediate feedback when making changes. Without robust tools, applications risk displaying stale data, over-fetching, or providing poor user experiences during network delays or errors.

React Query and SWR address these challenges by providing powerful abstractions for server state management, including automatic caching, background updates, and seamless synchronization between client and server

---

2. Learning Objectives

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

- **Understand the difference between server state and client state.**
- **Use query keys (React Query) or cache keys (SWR) to manage and identify cached data.**
- **Implement optimistic updates for responsive user interfaces.**
- **Invalidate and update the cache to keep data fresh and in sync.**
- **Leverage React Query and SWR for efficient, scalable data management in React applications.**

---

3. Concept Introduction with Analogy

Imagine your application as a newsroom, where reporters (the frontend) need to get the latest stories (data) from the wire (the server). The newsroom keeps track of stories it already has (caching), asks for updates when a story might be outdated (stale data detection), and sometimes publishes breaking news before confirming all details (optimistic updates). If a story changes, the newsroom quickly updates its records to keep everyone informed (cache invalidation).

React Query and SWR act as the newsroom's editorial system, automating these processes so reporters can focus on storytelling rather than managing logistics

---

4. Technical Deep Dive

Server State vs. Client State

- **Server State:** Data stored and managed on the server, fetched by the client. Requires synchronization, caching, and background updates
- **Client State:** Data managed solely on the client (e.g., UI state, form input). Does not require synchronization with a server.

Query Keys (React Query) / Cache Keys (SWR)

- **Purpose:** Uniquely identify and manage cached data for queries. Used for caching, invalidation, and refetching
- **Examples:**
  - `'todos'` (simple string key)
  - `['todo', id]` (array key for dynamic data)
  - `['todo', {id, type: 'comments', commentId}]` (complex key for nested data)
  - In SWR: `['/api/user', token]` or `{ url: '/api/orders', args: user }`

Optimistic Updates

- **Purpose:** Update the UI immediately in response to user actions, before the server responds. If the server call fails, roll back the UI for a seamless experience[
- **Implementation:** Use mutation hooks (React Query: `useMutation`, SWR: `mutate`) to update the cache before the request completes.

Cache Invalidation

- **Purpose:** Ensure cached data is up-to-date after mutations or when data may have changed on the server
- **Methods:**
  - **Invalidate:** Mark cached data as stale so it is refetched on next use.
  - **Update:** Directly update the cache with new data from the server response.
  - **SWR:** Automatic revalidation on focus, interval, or manual trigger.

---

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

1. Fetching Data with Query Keys (React Query Example)

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

const { data, isLoading, error } = useQuery({
  queryKey: ['users'],
  queryFn: () => fetch('/api/users').then(res => res.json()),
});

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>;
```

The query key `['users']` uniquely identifies this query in the cache

2. SWR Example with Cache Keys

```
import useSWR from 'swr';

const { data, error } = useSWR('/api/users', fetcher);

if (error) return <p>Error loading users</p>;
if (!data) return <p>Loading...</p>;
return <ul>{data.map(user => <li key={user.id}>{user.name}</li>)}</ul>;
```

SWR uses the URL as the cache key by default, but you can use arrays or objects for more complex scenarios.

3. Optimistic Updates (React Query Example)

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

const queryClient = useQueryClient();
const mutation = useMutation({
  mutationFn: updateUser,
  onMutate: async (newUser) => {
    await queryClient.cancelQueries(['users']);
    const previousUsers = queryClient.getQueryData(['users']);
    queryClient.setQueryData(['users'], (old) => [...old, newUser]);
    return { previousUsers };
  },
  onError: (err, newUser, context) => {
    queryClient.setQueryData(['users'], context.previousUsers);
  },
  onSettled: () => {
    queryClient.invalidateQueries(['users']);
  },
});
```

This updates the UI immediately and rolls back if the server request fails

4. Cache Invalidation (React Query Example)

```
// Invalidate all users queries
queryClient.invalidateQueries(['users']);

// Invalidate a specific user query
queryClient.invalidateQueries(['user', userId]);
```

This marks the data as stale and triggers a refetch on next use

5. SWR Cache Invalidation

```
const { data, mutate } = useSWR('/api/users', fetcher);

// Manually trigger a refetch
mutate();
```

SWR also supports automatic revalidation on focus, interval, or manual trigger

---

6. Challenge Problem

Assignment Steps

1. **Set Up Your Project**
    - Create a new React app.
    - Install React Query (or SWR) and Axios (optional, you can use `fetch`).
    - Set up the QueryClient and wrap your app with `QueryClientProvider`.
2. **Fetch and Display Data**
    - Use the `useQuery` hook to fetch a list of posts from `https://jsonplaceholder.typicode.com/posts`.
    - Display the posts in a simple list.
    - Show a loading state while the data is being fetched.

  - Show an error message if the fetch fails.

3. **Add a Form to Create a New Post**
   - Create a form with fields for `title` and `body`.
   - Use the `useMutation` hook to post the new data to `https://jsonplaceholder.typicode.com/posts`.
   - After submitting the form, display a success message and update the list of posts.
4. **Implement Optimistic Updates**
   - When you submit the form, immediately add the new post to the list before the server responds.
   - If the server request fails, remove the optimistically added post.
   - Show a loading indicator while the mutation is in progress.
5. **Invalidate the Cache**
   - After a successful mutation, invalidate the posts query so the list is refreshed and up-to-date.
   - Alternatively, update the cache manually with the response from the server.
6. **Bonus: Use DevTools**
   - Add React Query DevTools to your app to monitor and debug your queries and mutations.

---

7. Common Pitfalls & Best Practices

Pitfall	Best Practice
Not using unique query/cache keys	Always use keys to manage and identify cached data
Ignoring optimistic update rollback	Roll back UI changes if the server request fails
Forgetting to invalidate cache after mutations	Invalidate or update cache to keep data fresh
Mixing server and client state unnecessarily	Use dedicated tools for server state, not client state
Not handling loading and error states	Always provide feedback during loading and errors

---

8. Optional: Programmer's Workflow Checklist

- **Identify server state and use React Query or SWR to manage it.**
- **Use query keys (React Query) or cache keys (SWR) to uniquely identify and cache data.**
- **Implement optimistic updates for a responsive UI.**
- **Invalidate or update the cache after mutations.**
- **Handle loading and error states for all network requests.**
- **Test edge cases (e.g., network errors, concurrent updates).**
- **Monitor and debug queries and mutations with DevTools.**
