

React

Data Fetching / Mutations

Data Fetching,
Mutations?

What is Data Fetching & Mutation?

Data Fetching

- The process of retrieving data from a server or external source.
- Commonly done using **GET** requests, fetching data in formats like **JSON** or **XML**.

Mutation

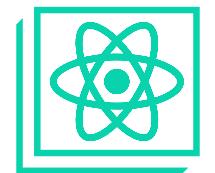
- The process of modifying or creating data on a server.
- Often performed through **POST**, **PUT**, **PATCH** or **DELETE** requests.

CSR vs SSR

CSR vs SSR

Client-Side Rendering (CSR)

- The browser fetches data and renders content directly on the client (the user's device).
- Common in single-page applications (SPAs).



Server-Side Rendering (SSR)

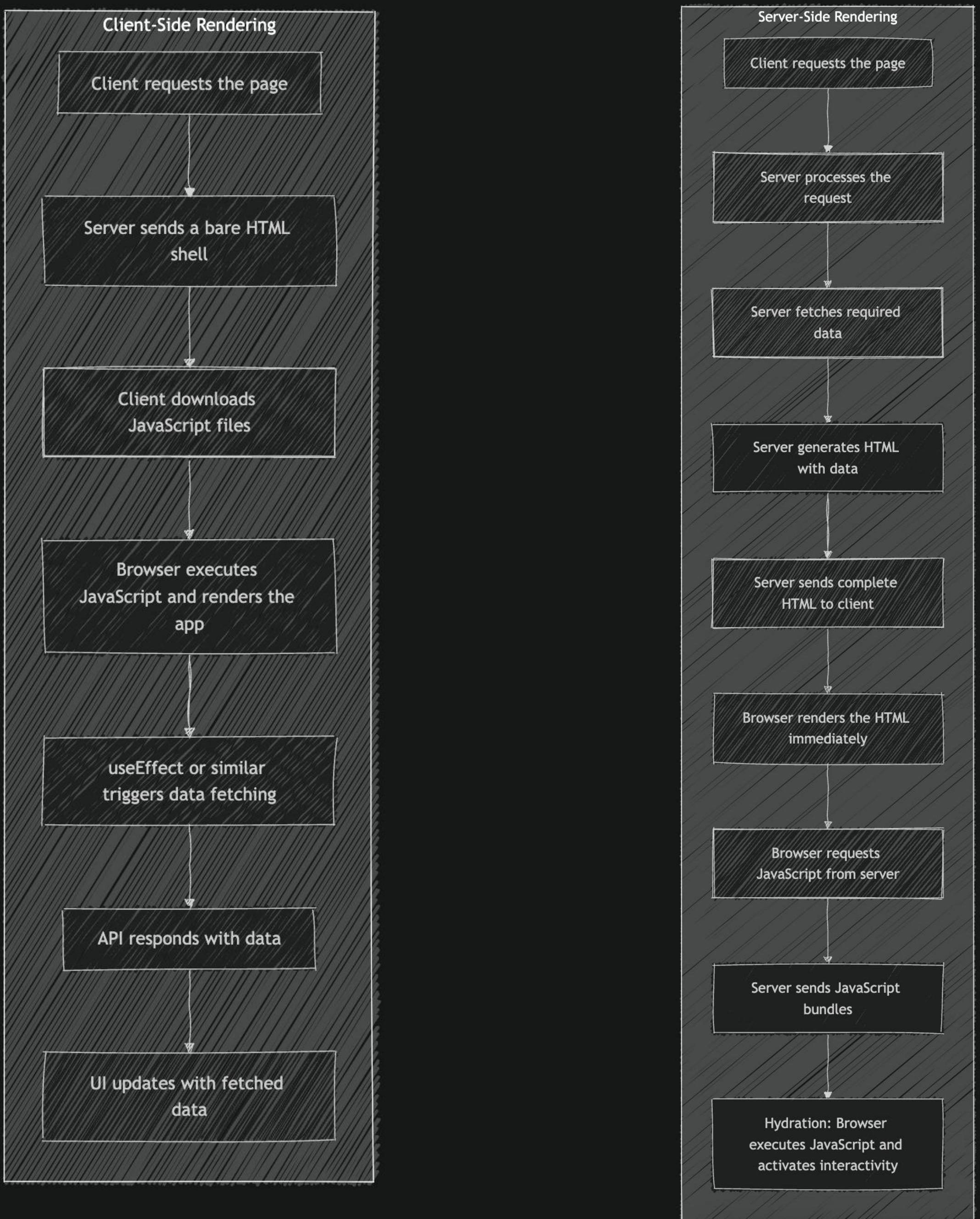
- The server generates the HTML and sends it to the client.
- The content is rendered before it reaches the browser, improving SEO and initial page load.

NEXT.js Remix



Vike

vinxi



Data Fetching · CSR Fetch

Data Fetching · CSR (Fetch)

Basic Client-Side Data Fetching with *fetch*

This example demonstrates client-side data fetching in React using the *fetch* API and *useState*. It fetches a list of posts, handles loading and error states, and displays the data once retrieved. This is a very basic data fetching implementation when using **Client-Side Rendering (CSR)**.

```
○ ○ ○ App.tsx

1 import * as React from "react";
2
3 interface Post {
4   id: string;
5   title: string;
6   views: string;
7 }
8
9 function App() {
10   const [items, setItems] = React.useState([]);
11   const [isLoading, setIsLoading] = React.useState(false);
12   const [error, setError] = React.useState<Error>();
13
14   React.useEffect(() => {
15     const fetchData = async () => {
16       setIsLoading(true);
17
18       try {
19         const response = await fetch(`.${import.meta.env.VITE_API_URL}/posts`);
20
21         if (!response.ok) {
22           throw new Error(`Fetch error: ${response.statusText}`);
23         }
24
25         const json = (await response.json()) as Post[];
26
27         setItems(json);
28       } catch (error) {
29         setError(error as Error);
30       }
31
32       setIsLoading(false);
33     };
34
35     fetchData();
36   }, []);
37
38   if (isLoading) {
39     return "Loading...";
40   }
41
42   if (error) {
43     return "Something went wrong, please try again later.";
44   }
45
46   return (
47     <ul>
48       {items.map((item) => (
49         <li key={item.id}>{item.title}</li>
50       ))}
51     </ul>
52   );
53 }
54
55 export default App;
56
```

API Clients

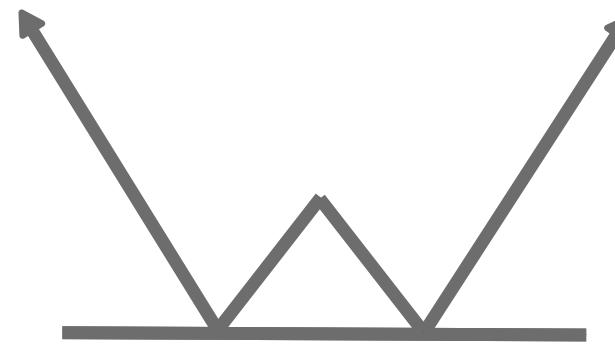
Simplifying API requests with an API Client

A X I O S

Axios: A promise-based HTTP client that simplifies request handling, with features like interceptors, automatic JSON parsing, and error handling.



Ky: A lightweight HTTP client built on fetch, providing a simpler API with automatic JSON parsing and modern features.



Wretch: A small, feature-rich alternative to fetch, offering a chainable API, built-in retries, and automatic JSON parsing.

○ ○ ○

TS axios.ts

```
1 import axios from "axios";
2
3 export const getPosts = async ({ page = 1, perPage = 5 }: GetPostsInput) => {
4   const { data } = await axios.get<List<Post>>(`${process.env.API_URL}/posts`, {
5     params: { _page: page, _per_page: perPage, _sort: "-createdAt" },
6   });
7   return data;
8 }
```

○ ○ ○

TS wretch.ts

```
1 import wretch from "wretch";
2
3 export const getPosts = async ({ page = 1, perPage = 5 }: GetPostsInput) => {
4   return wretch(process.env.API_URL)
5     .url("/posts")
6     .query({ _page: page, _per_page: perPage, _sort: "-createdAt" })
7     .get()
8     .json<List<Post>>();
9 }
```

○ ○ ○

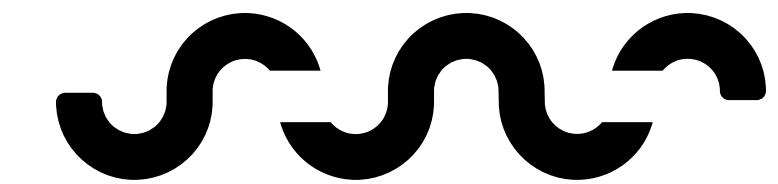
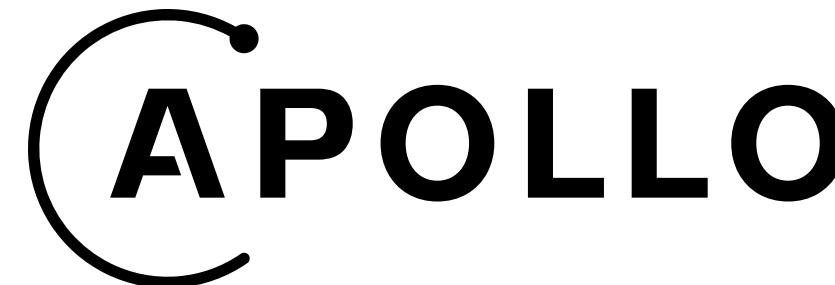
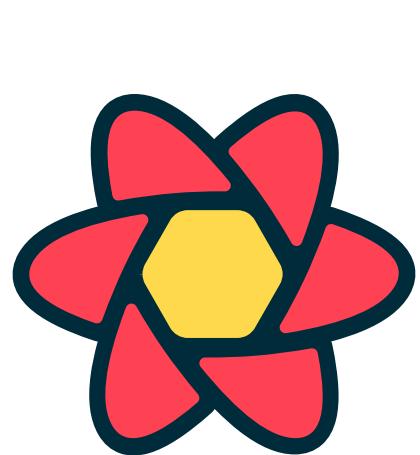
TS ky.ts

```
1 import ky from "ky";
2
3 export const getPosts = async ({ page = 1, perPage = 5 }: GetPostsInput) => {
4   return ky
5     .get("posts", {
6       prefixUrl: process.env.API_URL,
7       searchParams: { _page: page, _per_page: perPage, _sort: "-createdAt" },
8     })
9     .json<List<Post>>();
10 }
```

Data Fetching Libraries

Data Fetching Libraries

Efficient Data Fetching and Mutations



Tanstack Query (React Query)

- Automates caching, background updates, and data synchronization.
- Handles pagination, mutations, and server state management.

Apollo Client

- Full-featured GraphQL client with caching and state management.
- Advanced support for pagination and real-time data with subscriptions.

SWR (stale-while-revalidate)

- Lightweight, with built-in caching and revalidation.
- Optimized for background data fetching and performance.

○ ○ ○

⚛ react-query.tsx

```
1 import { useQuery } from "@tanstack/react-query";
2
3 // ...
4
5 const { data, isLoading, error } = useQuery({
6   queryKey: ["posts"],
7   queryFn: () => getPosts({ page: 1, perPage: 5 }),
8 });
9
```

○ ○ ○

⚛ apollo.ts

```
1 import { useQuery, gql } from "@apollo/client";
2
3 const GET_POSTS = gql`query GetPosts($page: Int, $perPage: Int) {
4   posts(page: $page, perPage: $perPage) {
5     id
6     title
7   }
8 }
9 `;
10
11 // ...
12
13
14 const { data, loading, error } = useQuery(GET_POSTS, {
15   variables: { page: 1, perPage: 5 },
16 });
17
```

○ ○ ○

⚛ swr.ts

```
1 import useSWR from "swr";
2
3 const fetcher = (url: string) => getPosts({ page: 1, perPage: 5 });
4
5 // ...
6
7 const { data, isLoading, error } = useSWR("posts", fetcher);
8
```

Data Fetching · CSR
React Query

Client-Side Data Fetching with React Query

Using *React Query* and *Ky*, data fetching becomes more declarative, reducing boilerplate and improving readability. The combination automates data caching, loading states, and error handling, simplifying the code and enhancing maintainability.

```
 1 import { useQuery } from "@tanstack/react-query";
 2 import ky from "ky";
 3
 4 interface Post {
 5   id: string;
 6   title: string;
 7   views: string;
 8 }
 9
10 const getPosts = () =>
11   ky<Post[]>(`${
12     import.meta.env.VITE_API_URL
13   }/posts`).json();
14
15 function App() {
16   const { data, isLoading, error } = useQuery({
17     queryKey: ["posts"],
18     queryFn: getPosts,
19   });
20
21   if (isLoading) {
22     return "Loading...";
23   }
24
25   if (error) {
26     return "Something went wrong, please try again later.";
27   }
28
29   return (
30     <ul>
31       {data?.map((item) => (
32         <li key={item.id}>{item.title}</li>
33       ))}
34     </ul>
35   );
36 }
37
38 export default App;
```

Request Deduplication

Caching

Window Focus Refetching

Pause while Offline

Handling Loading States

Retries

```
const { data, status } = useQuery({  
  queryKey: ['tasks'],  
  queryFn: fetchTasks  
})
```

and Error States

Exponential Backoff

Tracked Queries

stale-while-revalidate

Global State Management

Automatic Garbage Collection

Structural Sharing

Caching with React Query

Simplifying a lot, React Query uses a cache to store data by query key. When you call `useQuery` with the same key (e.g., `["posts"]`), it fetches the data once and stores it. On subsequent renders, it pulls the data from the cache instead of making another network request. This makes your app faster by reducing unnecessary API calls! You can also configure cache expiration and refetching behaviors.

```
○ ○ ○ App.tsx

1 import { useQuery } from "@tanstack/react-query";
2 import ky from "ky";
3
4 interface Post {
5   id: string;
6   title: string;
7   views: string;
8 }
9
10 const getPosts = () =>
11   ky<Post[]>(`${
12     import.meta.env.VITE_API_URL}/posts`).json();
13
14 function App() {
15   const { data, isLoading, error } = useQuery({
16     queryKey: ["posts"],
17     queryFn: getPosts,
18   });
19
20   if (isLoading) {
21     return "Loading...";
22   }
23
24   if (error) {
25     return "Something went wrong, please try again later.";
26   }
27
28   return (
29     <ul>
30       {data?.map((item) => (
31         <li key={item.id}>{item.title}</li>
32       ))}
33     </ul>
34   );
35
36 export default App;
```

```
App.tsx

1 import * as React from "react";
2
3 interface Post {
4   id: string;
5   title: string;
6   views: string;
7 }
8
9 function App() {
10  const [items, setItems] = React.useState<Post[]>([]);
11  const [isLoading, setIsLoading] = React.useState(false);
12  const [error, setError] = React.useState<Error>();
13
14  React.useEffect(() => {
15    const fetchData = async () => {
16      setIsLoading(true);
17
18      try {
19        const response = await fetch(`.${import.meta.env.VITE_API_URL}/posts`);
20
21        if (!response.ok) {
22          throw new Error(`Fetch error: ${response.statusText}`);
23        }
24
25        const json = (await response.json()) as Post[];
26
27        setItems(json);
28      } catch (error) {
29        setError(error as Error);
30      }
31
32      setIsLoading(false);
33    };
34
35    fetchData();
36  }, []);
37
38  if (isLoading) {
39    return "Loading...";
40  }
41
42  if (error) {
43    return "Something went wrong, please try again later.";
44  }
45
46  return (
47    <ul>
48      {items.map((item) => (
49        <li key={item.id}>{item.title}</li>
50      ))}
51    </ul>
52  );
53}
54
55 export default App;
56
```

```
App.tsx

1 import { useQuery } from "@tanstack/react-query";
2 import ky from "ky";
3
4 interface Post {
5   id: string;
6   title: string;
7   views: string;
8 }
9
10 const getPosts = () =>
11   ky<Post[]>(`.${import.meta.env.VITE_API_URL}/posts`).json();
12
13 function App() {
14  const { data, isLoading, error } = useQuery({
15    queryKey: ["posts"],
16    queryFn: getPosts,
17  });
18
19  if (isLoading) {
20    return "Loading...";
21  }
22
23  if (error) {
24    return "Something went wrong, please try again later.";
25  }
26
27  return (
28    <ul>
29      {data?.map((item) => (
30        <li key={item.id}>{item.title}</li>
31      ))}
32    </ul>
33  );
34}
35
36 export default App;
```

Mutations · CSR
Fetch

Mutations · CSR (Fetch)

Implementing Basic Mutations with State Updates

Here's a simple example of adding mutations to our client-side app. By submitting a new post through the form, this code sends a POST request, updates the server, and refetches the posts to keep the UI in sync. This approach demonstrates handling mutations and data updates from scratch without using any data fetching library.

```
App.tsx

1 import * as React from "react";
2
3 interface Post {
4   id: string;
5   title: string;
6   views: string;
7 }
8
9 function App() {
10   const [posts, setPosts] = React.useState([]);
11   const [isLoading, setIsLoading] = React.useState(false);
12   const [error, setError] = React.useState<Error>();
13
14   const [value, setValue] = React.useState("");
15
16   const fetchPosts = React.useCallback(async () => {
17     setIsLoading(true);
18
19     try {
20       const response = await fetch(`${import.meta.env.VITE_API_URL}/posts`);
21
22       if (!response.ok) {
23         throw new Error(`Fetch error: ${response.statusText}`);
24       }
25
26       const posts = (await response.json()) as Post[];
27
28       setPosts(posts);
29     } catch (error) {
30       setError(error as Error);
31     }
32
33     setIsLoading(false);
34   }, []);
35
36   React.useEffect(() => {
37     fetchPosts();
38   }, [fetchPosts]);
39
40   if (isLoading) {
41     return "Loading...";
42   }
43
44   if (error) {
45     return "Something went wrong, please try again later.";
46   }
47
48   const handleSubmit = async (event: React.FormEvent<HTMLFormElement>) => {
49     event.preventDefault();
50
51     try {
52       const response = await fetch(`${import.meta.env.VITE_API_URL}/posts`, {
53         method: "POST",
54         body: JSON.stringify({ title: value }),
55         headers: {
56           "content-type": "application/json",
57         },
58       });
59
59       if (!response.ok) {
60         throw new Error(`Fetch error: ${response.statusText}`);
61       }
62
63       await fetchPosts();
64
65       setValue("");
66     } catch (error) {
67       setError(error as Error);
68     }
69   };
70
71
72   return (
73     <div>
74       <h1>Mutation Basic CSR</h1>
75       <form onSubmit={handleSubmit}>
76         <input
77           value={value}
78           onChange={(event) => setValue(event.target.value)}
79         />
80         <button type="submit">Add</button>
81       </form>
82       <ul>
83         {posts.map((item) => (
84           <li key={item.id}>{item.title}</li>
85         )));
86       </ul>
87     </div>
88   );
89 }
90
91 export default App;
```

Mutations · CSR (Fetch)

Ensuring UI Sync · Revalidation

By calling `fetchPosts` after submitting the new post, we ensure our UI reflects the latest data from the server. This approach mirrors `react-query's` `invalidateQueries / refetch`, where refetching invalidated data keeps the client and server in sync.

Note:

While we could immediately add the new post via awaiting the response from create post API to our list, invalidating and refetching is often simpler and more maintainable, reducing errors and ensuring accurate data. This applies to `react-query` as well.

```
App.tsx

1 import * as React from "react";
2
3 interface Post {
4   id: string;
5   title: string;
6   views: string;
7 }
8
9 function App() {
10   // ...
11
12   const handleSubmit = async (event: React.FormEvent<HTMLFormElement>) => {
13     event.preventDefault();
14
15     try {
16       const response = await fetch(`.${import.meta.env.VITE_API_URL}/posts`, {
17         method: "POST",
18         body: JSON.stringify({ title: value }),
19         headers: {
20           "content-type": "application/json",
21         },
22       });
23
24       if (!response.ok) {
25         throw new Error(`Fetch error: ${response.statusText}`);
26       }
27
28       // fetch `posts` to ensure `posts` are updated after mutation.
29       await fetchPosts();
30
31       setValue("");
32     } catch (error) {
33       setError(error as Error);
34     }
35   };
36
37   return (
38     // ...
39   );
40 }
41
42 export default App;
```

Mutations · CSR
React Query

Mutations · CSR (React Query)

Implementing Mutations with React Query

In this example, we use `useMutation` from React Query to handle adding new posts, simplifying our mutation workflow and ensuring the UI remains up-to-date.

After creating a new post, we use `invalidateQueries` to trigger a refetch of the "posts" query, synchronizing the latest data with minimal code. This pattern provides a clean, reliable way to keep our UI and backend in sync without manually managing state updates.

This approach also highlights the key benefit of React Query: automated cache management and query invalidation, reducing the chance of stale data and making our code easier to maintain over time.

```
App.tsx

1 import { useMutation, useQuery, useQueryClient } from "@tanstack/react-query";
2 import ky from "ky";
3 import * as React from "react";
4
5 interface Post {
6   id: string;
7   title: string;
8   views: string;
9 }
10
11 const getPosts = () => ky<Post[]>(`.${import.meta.env.VITE_API_URL}/posts`).json();
12
13 const createPost = (title: string) =>
14   ky
15     .post<Post>(`.${import.meta.env.VITE_API_URL}/posts`, {
16       json: { title },
17     })
18     .json();
19
20 function App() {
21   const [value, setValue] = React.useState("");
22
23   const queryClient = useQueryClient();
24
25   const { data, isLoading, error } = useQuery({
26     queryKey: ["posts"],
27     queryFn: getPosts,
28   });
29
30   const { mutate } = useMutation({
31     mutationFn: createPost,
32     onSuccess: async () => {
33       await queryClient.invalidateQueries({ queryKey: ["posts"] });
34       setValue("");
35     },
36   });
37
38   if (isLoading) {
39     return "Loading...";
40   }
41
42   if (error) {
43     return "Something went wrong, please try again later.";
44   }
45
46   const handleSubmit = async (event: React.FormEvent<HTMLFormElement>) => {
47     event.preventDefault();
48
49     mutate(value);
50   };
51
52   return (
53     <div>
54       <h1>Mutation React Query CSR</h1>
55       <form onSubmit={handleSubmit}>
56         <input
57           value={value}
58           onChange={(event) => setValue(event.target.value)}
59         />
60         <button type="submit">Add</button>
61       </form>
62       <ul>
63         {data?.map((item) => (
64           <li key={item.id}>{item.title}</li>
65         ))}
66       </ul>
67     </div>
68   );
69 }
70
71 export default App;
```

 App.tsx

```
1 // ...
2
3 function App() {
4   const [value, setValue] = React.useState("");
5
6   const queryClient = useQueryClient();
7
8   const { data, isLoading, error } = useQuery({
9     queryKey: ["posts"],
10    queryFn: getPosts,
11  });
12
13   const { mutate } = useMutation({
14     mutationFn: createPost,
15     onSuccess: async () => {
16       await queryClient.invalidateQueries({ queryKey: ["posts"] });
17       setValue("");
18     },
19   });
20
21   if (isLoading) {
22     return "Loading...";
23   }
24
25   if (error) {
26     return "Something went wrong, please try again later.";
27   }
28
29   const handleSubmit = async (event: React.FormEvent<HTMLFormElement>) => {
30     event.preventDefault();
31
32     mutate(value);
33   };
34
35   return (
36     // ...
37   );
38 }
39
40 export default App;
```

Data Fetching / Mutations · CSR Fetch vs React Query

○ ○ ○

TS posts.ts

```
1 export const useGetPosts = ({ page = 1, perPage = 5 }: GetPostsInput) => {
2   const [data, setData] = React.useState<List<Post>>();
3   const [isLoading, setIsLoading] = React.useState(false);
4   const [error, setError] = React.useState<Error>();
5
6   const fetchPosts = React.useCallback(async () => {
7     setIsLoading(true);
8
9     try {
10       const response = await fetch(
11         `${import.meta.env.VITE_API_URL}/posts?
12 _page=${page}&_per_page=${perPage}&_sort=-createdAt`,
13       );
14
15       if (!response.ok) {
16         throw new Error(`Fetch error: ${response.statusText}`);
17       }
18
19       const data = (await response.json()) as List<Post>;
20
21       setData(data);
22     } catch (error) {
23       setError(error as Error);
24     }
25
26     setIsLoading(false);
27   }, [page, perPage]);
28
29   React.useEffect(() => {
30     fetchPosts();
31   }, [fetchPosts]);
32
33   return {
34     data,
35     isLoading,
36     error,
37     fetchPosts,
38   };
39 }
```

○ ○ ○

TS posts.ts

```
1 const getPosts = ({ page = 1, perPage = 5 }: GetPostsInput) =>
2   ky<List<Post>>(`${import.meta.env.VITE_API_URL}/posts`, {
3     searchParams: {
4       _page: page,
5       _per_page: perPage,
6       _sort: "-createdAt",
7     },
8   }).json();
9
10 export const useGetPostsQuery = (input: GetPostsInput) =>
11   useQuery({
12     queryKey: ["posts", input],
13     queryFn: () => getPosts(input),
14   });
15 }
```

App.tsx

```
1 // ...
2
3 function App() {
4   // ...
5
6   const { data, isLoading, error, fetchPosts } = useGetPosts({
7     page: pagination.pageIndex + 1,
8     perPage: pagination.pageSize,
9   });
10  const { createPost } = useCreatePost();
11  const { deletePost } = useDeletePost();
12
13  const handleSubmit = async (event: React.FormEvent<HTMLFormElement>) => {
14    event.preventDefault();
15    await createPost(value);
16    await fetchPosts();
17    setValue("");
18  };
19
20  const handleDeletePost = async (postId: string) => {
21    await deletePost(postId);
22    await fetchPosts();
23  };
24
25  // ...
26
27  const [value, setValue] = React.useState("");
28
29  if (isLoading) {
30    return "Loading...";
31  }
32
33  if (error) {
34    return "Something went wrong, please try again later.";
35  }
36
37  return (
38    // ...
39  );
40}
41
42 export default App;
43
```

App.tsx

```
1 // ...
2
3 function App() {
4   // ...
5
6   const { data, isLoading, error } = useGetPostsQuery({
7     page: pagination.pageIndex + 1,
8     perPage: pagination.pageSize,
9   });
10  const { mutate: createPost } = useCreatePostMutation();
11  const { mutate: deletePost } = useDeletePostMutation();
12
13  const handleSubmit = (event: React.FormEvent<HTMLFormElement>) => {
14    event.preventDefault();
15    createPost(value);
16    setValue("");
17  };
18
19  const handleDeletePost = (postId: string) => {
20    deletePost(postId);
21  };
22
23  // ...
24
25  const [value, setValue] = React.useState("");
26
27  if (isLoading) {
28    return "Loading...";
29  }
30
31  if (error) {
32    return "Something went wrong, please try again later.";
33  }
34
35  return (
36    // ...
37  );
38}
39
40 export default App;
```

Plain Fetch vs React Query

I prefer creating and exporting custom `useXQuery` (or `queryOptions` directly in `react-query v5`) or `useXMutation` hooks as reusable utilities across the app. This approach makes it easier to maintain consistent data-fetching logic.

It is clear that how React Query streamlines data fetching and mutations compared to manually managing state with plain fetch. Here's why React Query outshines manual data fetching / mutation management with plain fetch 99.9% of the time.

- Simplified Error and Loading State Handling
- Automated Caching and Refetching via `InvalidateQueries`
- Configuration and Flexibility (retrying requests, stale time, polling, `infiniteQuery` etc.)

Data Fetching / Mutations · SSR
Fetch

Server-Side Data Fetching / Mutations with Fetch

In this example, we see a basic implementation of server-side data fetching and mutations without using React Query. Instead of relying on `useEffect` in the client, data fetching is handled on the server via the `getPosts` function and passed directly to the client component (`Posts`) as props.

For pagination and other client-side states, we use `searchParams` to handle state across pages, allowing consistent pagination and navigation. The pagination state is updated with `router.push`, helping to maintain query parameters and refresh the data as needed.

- Server-side Fetching: Fetches data on the server and passes it as props, ideal for initial load optimizations.
- Pagination Handling: Updates pagination state in `searchParams` and triggers navigation via `router.push`.
- CRUD Operations: Supports `createPost` and `deletePost` server actions directly within the client component for mutation actions.

TS post.ts

```
1 "use server";
2
3 // ...
4
5 export const getPosts = async ({ page = 1, perPage = 5 }: GetPostsInput) => {
6   const response = await fetch(
7     `${process.env.API_URL}/posts?_page=${page}&_per_page=${perPage}&_sort=-createdAt`,
8   );
9
10 if (!response.ok) {
11   throw new Error(`Fetch error: ${response.statusText}`);
12 }
13
14 const data = (await response.json()) as List<Post>;
15
16 return data;
17 };
18 // ...
19
```

page.tsx

```
1 import { getPosts } from "@/actions/posts";
2 import { Posts } from "@/components/posts";
3
4 export default async function Home({
5   searchParams,
6 }: {
7   searchParams: Promise<{ [key: string]: string | undefined }>;
8 }) {
9   const { page = "1", perPage = "5" } = await searchParams;
10
11   const data = await getPosts({
12     page: Number(page),
13     perPage: Number(perPage),
14   });
15
16   return <Posts data={data} />;
17 }
```

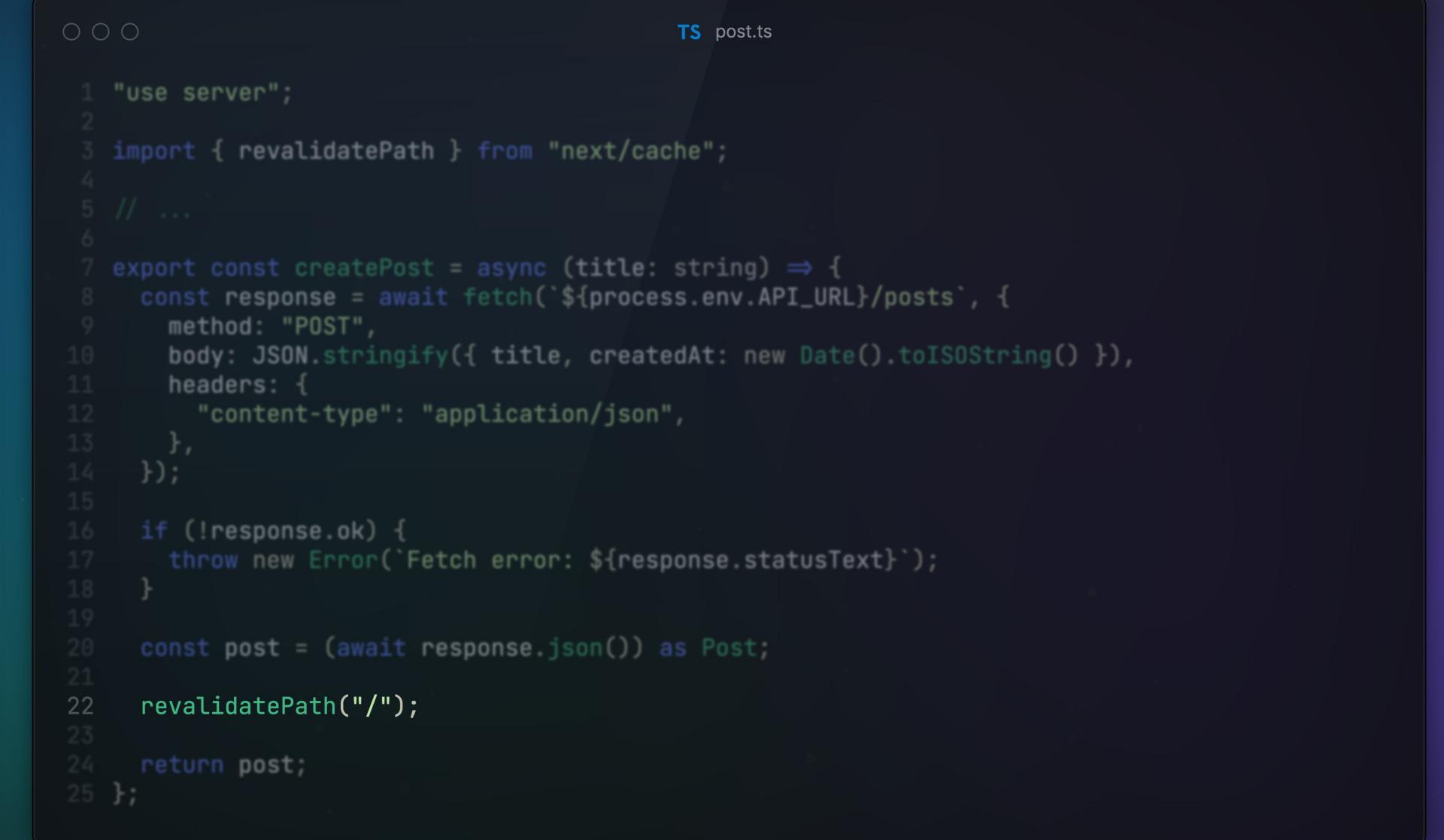
posts.tsx

```
1 "use client";
2
3 // ...
4
5 import { usePathname, useRouter, useSearchParams } from "next/navigation";
6
7 // ...
8
9 interface PostsProps {
10   data: List<Post>;
11 }
12
13 export const Posts = ({ data }: PostsProps) => {
14   const searchParams = useSearchParams();
15   const router = useRouter();
16   const pathname = usePathname();
17
18   const pagination = React.useMemo(
19     () =>
20     ({
21       pageIndex: Number(searchParams.get("page") ?? "1") - 1,
22       pageSize: Number(searchParams.get("perPage") ?? "5"),
23     }) as const satisfies PaginationState,
24     [searchParams],
25   );
26
27   // We could've used `useActionState` here. Visit
28   // https://github.com/vercel/next.js/tree/canary/examples/next-forms for more
29   // examples.
30   const handleSubmit = async (event: React.FormEvent<HTMLFormElement>) => {
31     event.preventDefault();
32     await createPost(value);
33     setValue("");
34   };
35
36   const handleDeletePost = async (postId: string) => {
37     await deletePost(postId);
38   };
39
40   const handlePaginationChange = React.useCallback(
41     (updater: Updater<PaginationState>) => {
42       if (typeof updater === "function") {
43         const newPagination = updater(pagination);
44         const params = new URLSearchParams(searchParams.toString());
45         params.set("page", `${newPagination.pageIndex + 1}`);
46         params.set("perPage", `${newPagination.pageSize}`);
47         router.push(`${pathname}?${params.toString()}`);
48       }
49     },
50     [pagination, router.push, searchParams, pathname],
51   );
52
53   const [value, setValue] = React.useState("");
54
55   return (
56     // ...
57   );
58 }
```

Cache Invalidation with revalidatePath

In this example, `revalidatePath` is used to tell Next.js to invalidate the cache for a specified path ("/" in this case) after creating a new post. By calling `revalidatePath("/")`, we ensure that the `getPosts` function will be triggered again when the page is re-rendered, so any newly created posts are included in the updated data.

This cache invalidation step guarantees that our UI stays up-to-date without requiring a full page refresh, enabling a more responsive experience.



```
 1 "use server";
 2
 3 import { revalidatePath } from "next/cache";
 4
 5 // ...
 6
 7 export const createPost = async (title: string) => {
 8   const response = await fetch(`${process.env.API_URL}/posts`, {
 9     method: "POST",
10     body: JSON.stringify({ title, createdAt: new Date().toISOString() }),
11     headers: {
12       "content-type": "application/json",
13     },
14   });
15
16   if (!response.ok) {
17     throw new Error(`Fetch error: ${response.statusText}`);
18   }
19
20   const post = (await response.json()) as Post;
21
22   revalidatePath("/");
23
24   return post;
25 }
```

Data Fetching / Mutations · SSR React Query

Server-Side Data Fetching / Mutations with React Query

We demonstrate the advanced server-side rendering (SSR) technique with React Query in Next.js. By using `prefetchQueries` on the server side and then dehydrating it back to the client, we achieve a smooth data handover, ensuring that the client has immediate access to pre-fetched data without needing to refetch on load.

```
○○○ page.tsx

1 // Using server api.
2 import { getPosts } from "@/apis/posts.server";
3
4 export const getPostsQueryOptions = (...input: Parameters<typeof getPosts>) =>
5   queryOptions({
6     queryKey: ["posts", ...input],
7     queryFn: () => getPosts(...input),
8     placeholderData: {
9       first: 0,
10      prev: null,
11      next: null,
12      last: 0,
13      pages: 0,
14      items: 0,
15      data: [],
16    },
17  },
18});
```

```
○○○ page.tsx

1 import { Posts } from "@/components/posts";
2 import { getPostsQueryOptions } from "@/queries/posts.server";
3 import {
4   HydrationBoundary,
5   QueryClient,
6   dehydrate,
7 } from "@tanstack/react-query";
8 import * as React from "react";
9
10 export default async function Home({
11   searchParams,
12 }: {
13   searchParams: Promise<{ [key: string]: string | undefined }>;
14 }) {
15   const { page = "1", perPage = "5" } = await searchParams;
16
17   const queryClient = new QueryClient();
18
19   await queryClient.prefetchQuery(
20     getPostsQueryOptions({
21       page: Number(page),
22       perPage: Number(perPage),
23     }),
24   );
25
26   return (
27     <HydrationBoundary state={dehydrate(queryClient)}>
28       <React.Suspense fallback={<>Loading...</>}>
29         <Posts />
30       </React.Suspense>
31     </HydrationBoundary>
32   );
33 }
34
```

posts.tsx

```
1 "use client";
2
3 import {
4   getPostsQueryOptions,
5   useCreatePostMutation,
6   useDeletePostMutation,
7 } from "@/queries/posts.client";
8
9 // ...
10
11 export const Posts = () => {
12   // ...
13
14   const { data } = useSuspenseQuery(
15     getPostsQueryOptions({
16       page,
17       perPage,
18     }),
19   );
20
21   const { mutate: createPost, isPending, error } = useCreatePostMutation();
22
23   const { mutate: deletePost } = useDeletePostMutation();
24
25   const pagination = React.useMemo(
26     () =>
27       ({
28         pageIndex: page - 1,
29         pageSize: perPage,
30       }) as const satisfies PaginationState,
31     [page, perPage],
32   );
33
34   const handleSubmit = async (event: React.FormEvent<HTMLFormElement>) => {
35     event.preventDefault();
36     createPost(value);
37     setValue("");
38   };
39
40   const handleDeletePost = async (postId: string) => {
41     deletePost(postId);
42   };
43
44   const handlePaginationChange = React.useCallback(
45     (updater: Updater<PaginationState>) => {
46       if (typeof updater === "function") {
47         const newPagination = updater(pagination);
48         const params = new URLSearchParams(searchParams.toString());
49         params.set("page", `${newPagination.pageIndex + 1}`);
50         params.set("perPage", `${newPagination.pageSize}`);
51         router.push(`${pathname}?${params.toString()}`);
52       }
53     },
54     [pagination, router.push, searchParams, pathname],
55   );
56
57   const [value, setValue] = React.useState("");
58
59   // ...
60
61   return (
62     // ...
63   );
64 }
```

TS posts.ts

```
1 // src/actions/posts.ts
2
3 // Server actions with "use server";
4 "use server";
5
6 import * as posts from "@apis/posts.server";
7
8 export const createPost = async (title: string) => {
9   if (!title) {
10     return { message: "Title is required" };
11   }
12
13   return posts.createPost(title);
14 };
15
16 // src/lib/react-query.ts
17
18 // Typescript helper function used for type inference.
19 export function mutationOptions<
20   TData = unknown,
21   TError = DefaultError,
22   TVariables = void,
23   TContext = unknown,
24 >(
25   options: UseMutationOptions<TData, TError, TVariables, TContext>,
26 ): UseMutationOptions<TData, TError, TVariables, TContext> {
27   return options;
28 }
29
30 // src/queries/posts.client.ts
31
32 const createPostMutationOptions = mutationOptions({
33   mutationFn: createPost,
34 });
35
36 export const useCreatePostMutation = ({
37   onSuccess,
38   ...options
39 }: Omit<typeof createPostMutationOptions, "mutationFn"> = {}) => {
40   const queryClient = useQueryClient();
41
42   return useActionMutation({
43     ...createPostMutationOptions,
44     // Idiomatic server actions should not throw, useMutation expects error to be thrown.
45     transformError: (result) => {
46       if ("message" in result) {
47         // We take the result of the server action call, and return an error so that `useActionMutation` throws it.
48         return new Error(result.message);
49       }
50       return null;
51     },
52     onSuccess: (...args) => {
53       queryClient.invalidateQueries(getPostsQueryOptions());
54
55       return onSuccess?.(...args);
56     },
57     ...options,
58   });
59 }
```

Optimistic Updates

React Query

Optimistic Updates React Query

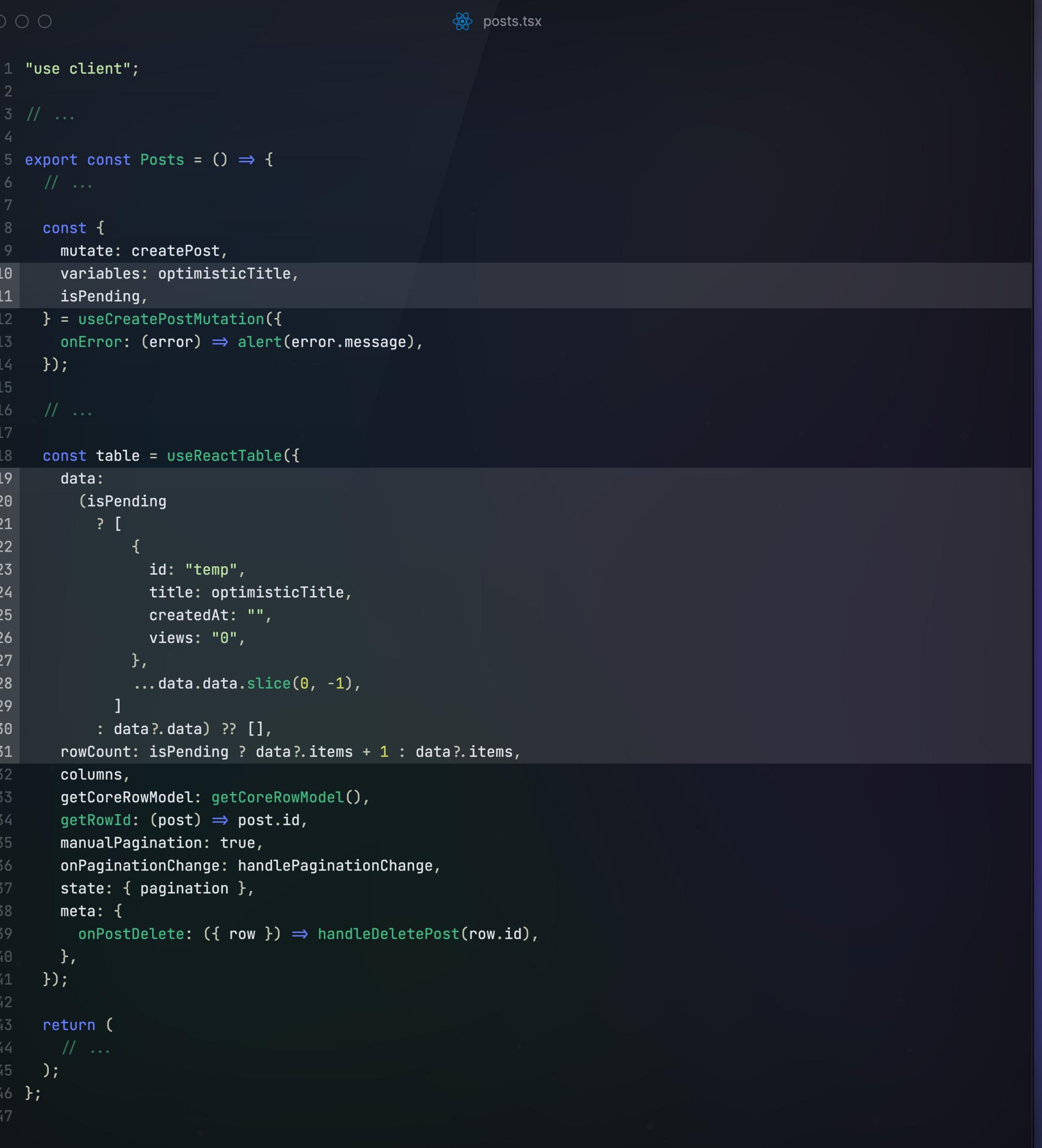
Optimistic Updates via UI

Optimistic updates allow you to update the UI immediately as if a mutation has succeeded while the mutation is still running, creating a seamless user experience.

Note:

Avoid using optimistic updates for the sake of "performance optimization.". For most cases, invalidateQueries is sufficient and simpler, making the code much more maintainable. Some example valid use cases for optimistic updates would be:

- Liking a post
- Bookmarking a post by clicking on a toggle star button



```
 1 "use client";
 2
 3 // ...
 4
 5 export const Posts = () => {
 6   // ...
 7
 8   const {
 9     mutate: createPost,
10     variables: optimisticTitle,
11     isPending,
12   } = useCreatePostMutation({
13     onError: (error) => alert(error.message),
14   });
15
16   // ...
17
18   const table = useReactTable({
19     data:
20       (isPending
21        ? [
22          {
23            id: "temp",
24            title: optimisticTitle,
25            createdAt: "",
26            views: "0",
27          },
28          ...data.data.slice(0, -1),
29        ]
30        : data?.data) ?? [],
31     rowCount: isPending ? data?.items + 1 : data?.items,
32     columns,
33     getCoreRowModel: getCoreRowModel(),
34     getRowId: (post) => post.id,
35     manualPagination: true,
36     onPaginationChange: handlePaginationChange,
37     state: { pagination },
38     meta: {
39       onPostDelete: ({ row }) => handleDeletePost(row.id),
40     },
41   });
42
43   return (
44     // ...
45   );
46 };
47
```

Infinite Scrolling React Query

Infinite Scrolling React Query

“Scroll to load more” via useInfiniteQuery

Infinite scrolling allows you to load additional content dynamically as the user scrolls, creating a seamless user experience.

useInfiniteQuery: A hook designed to handle implementation of infinite scrolling, managing states and exposes utility function like `hasNextPage` and `fetchNextPage`.

InView: A utility to detect when the user scrolls near the end of the list.

```
TS posts.ts

1
2
3 import { getPosts } from "@/apis/posts.client";
4 import { infiniteQueryOptions } from "@tanstack/react-query";
5
6 export const getPostsInfiniteQueryOptions = (
7   ...input: Parameters<typeof getPosts>
8 ) =>
9   infiniteQueryOptions({
10   queryKey: ["posts", ...input],
11   queryFn: async ({ pageParam }) => {
12     // Added this to simulate a slight delay to that we can see the infinite scrolling more clearly.
13     await new Promise((res) => setTimeout(res, 500));
14     return getPosts(pageParam);
15   },
16   initialPageParam: {
17     page: 1,
18     perPage: 10,
19     ...input.at(0),
20   },
21   getNextPageParam: (lastPage, allPages, lastPageParam) => {
22     if (!lastPage.next) {
23       return null;
24     }
25
26     return {
27       ...lastPageParam,
28       page: lastPage.next,
29     };
30   },
31   getPreviousPageParam: (firstPage, allPages, firstPageParam) => {
32     if (!firstPage.prev) {
33       return null;
34     }
35
36     return {
37       ...firstPageParam,
38       page: firstPage.prev,
39     };
40   },
41 });


```

○ ○ ○

TS posts.tsx

```
1 "use client";
2
3 import * as React from "react";
4
5 import { getPostsInfiniteQueryOptions } from "@/queries/posts.client";
6 import { useInfiniteQuery } from "@tanstack/react-query";
7 import {InView} from "react-intersection-observer";
8
9 export const Posts = () => {
10   const { data, hasNextPage, isFetching, fetchNextPage } = useInfiniteQuery(
11     getPostsInfiniteQueryOptions(),
12   );
13
14   const items = data?.pages?.flatMap((page) => page.data) ?? [];
15
16   return (
17     <div>
18       <h1>Infinite Scrolling</h1>
19       <div
20         style={{
21           maxHeight: 300,
22           overflowY: "scroll",
23           border: "1px solid black",
24         }}
25       >
26         <ul>
27           {items.map((item) => (
28             <li key={item.id}>{item.title}</li>
29           ))}
30           {hasNextPage && !isFetching && (
31             <InView
32               onChange={(inView) => inView && !isFetching && fetchNextPage()}
33             />
34           )}
35           {isFetching && "Loading..."}
36         </ul>
37       </div>
38     </div>
39   );
40 };
```

React Query Complementary Tools

React Query Complementary Tools

ESLint Plugin Query

Using the official eslint plugin is highly recommended as it is used to enforce best practices and to help you avoid common mistakes.

The image shows a dark-themed code editor interface with two code snippets displayed in separate tabs.

eslint.config.js

```
import pluginQuery from '@tanstack/eslint-plugin-query'
export default [
  ...pluginQuery.configs['flat/recommended'],
  // Any other config...
]
```

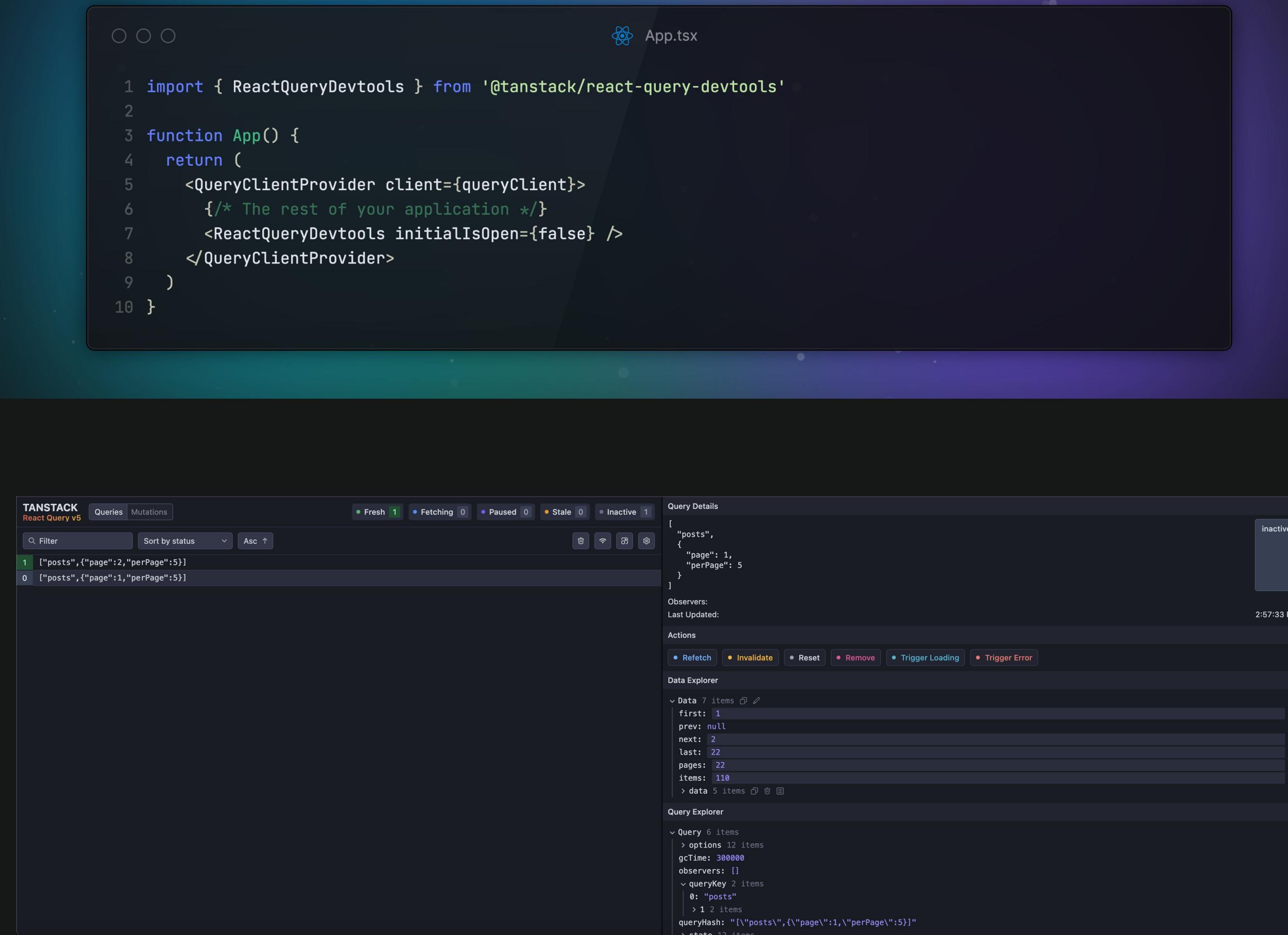
posts.ts

```
export const getPostsQueryOptions = (...input: Parameters<typeof getPosts>) =>
  queryOptions({
    // The following dependencies are missing in your queryKey: input eslint(@tanstack/query/exhaustive-deps)
    queryKey: ['posts'],
    queryFn: () => getPosts(...input),
    placeholderData: {
      first: 0,
      prev: null,
      next: null,
      last: 0,
      pages: 0,
      items: 0,
      data: [],
    },
  });
};
```

React Query Complementary Tools

React Query Devtools

Setting up React Query Devtools is also highly recommended as they help visualize all the inner workings of React Query and will likely save you hours of debugging if you find yourself in a pinch!



The screenshot shows the React Query Devtools integrated into a code editor. At the top, there's a code snippet for `App.tsx`:

```
1 import { ReactQueryDevtools } from '@tanstack/react-query-devtools'
2
3 function App() {
4   return (
5     <QueryClientProvider client={queryClient}>
6       {/* The rest of your application */}
7       <ReactQueryDevtools initialIsOpen={false} />
8     </QueryClientProvider>
9   )
10 }
```

Below the code editor is the React Query Devtools interface. It has several sections:

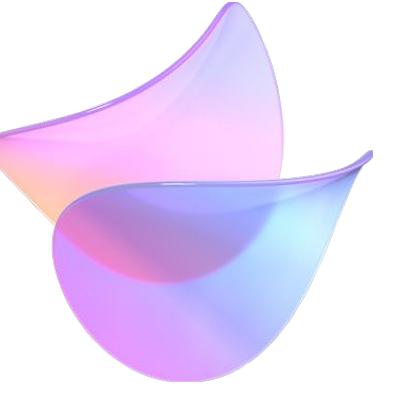
- TANSTACK React Query v5**: The title bar.
- Queries**: A table showing two active queries:
 - Index 1: ["posts", {"page":2,"perPage":5}]
 - Index 0: ["posts", {"page":1,"perPage":5}]
- Actions**: Buttons for Refetch, Invalidate, Reset, Remove, Trigger Loading, and Trigger Error.
- Query Details**: Shows the query configuration:

```
[ "posts",
  {
    "page": 1,
    "perPage": 5
  }
]
```
- Observers**: Shows 0 observers.
- Last Updated**: Shows 2:57:33 PM.
- Data Explorer**: Shows the data structure:

```
first: 1
prev: null
next: 2
last: 22
pages: 22
items: 110
> data 5 items
```
- Query Explorer**: Shows the query configuration:

```
> options 12 items
gcTime: 300000
observers: []
< queryKey 2 items
  0: "posts"
  > 1 2 items
queryHash: "\\"posts\\",{\"page\":1,\"perPage\":5}]"
> state 12 items
```

You don't have to
re-invent The Wheel



Opening the floor for questions...

Thank You

Resources

<https://tkdodo.eu/blog/tags/react-query>

<https://github.com/junwen-k/react-data-fetching-mutation>

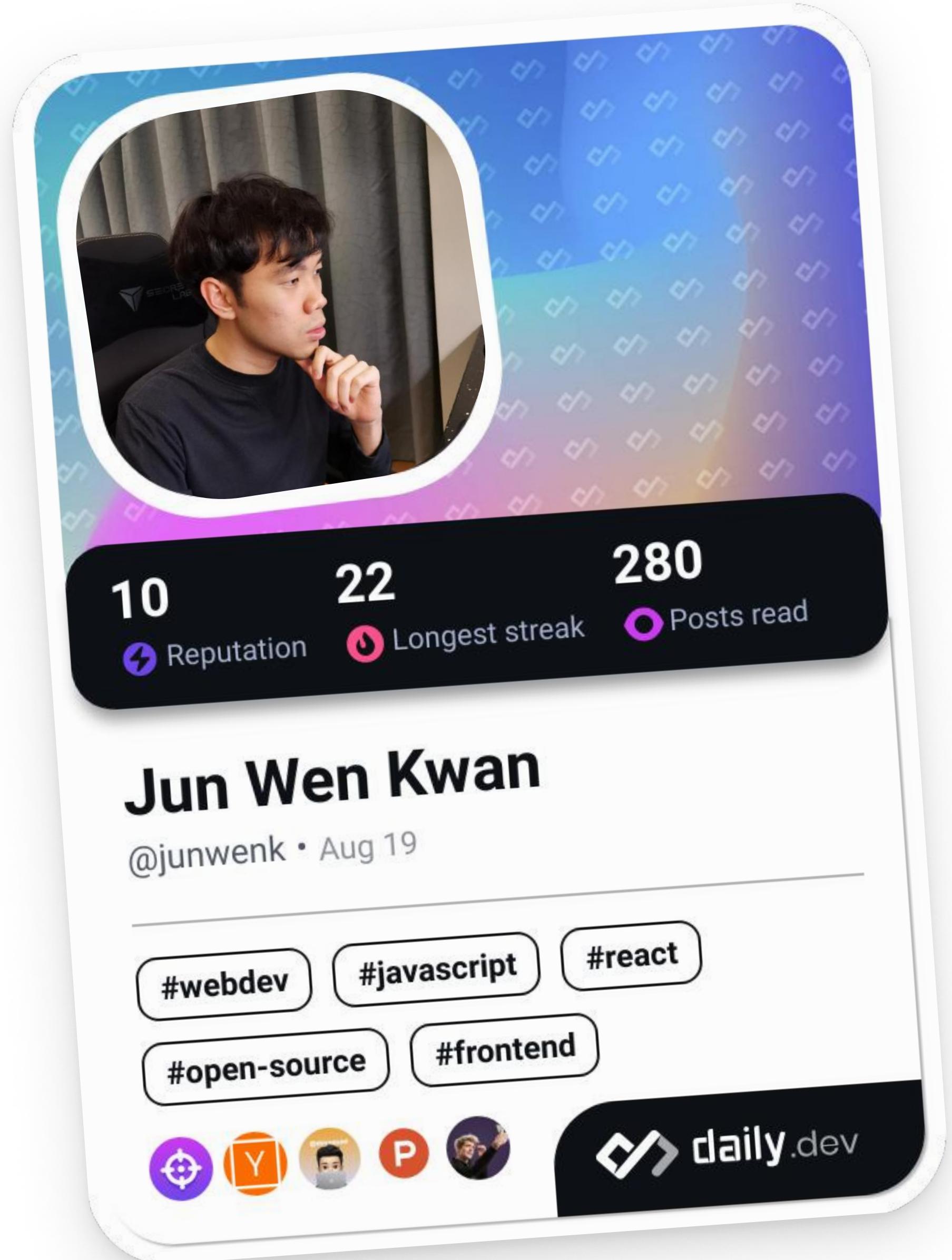
<https://nextjs.org/docs/app/building-your-application/data-fetching>

Special Thanks

<https://chalk.ist/>

<https://github.com/typicode/json-server>

<https://www.kl-react.com/>



<https://linkedin.com/in/junwenk>



<https://forms.gle/rcmjcrXPa6TEJ2hf8>



<https://github.com/junwen-k>