React

Data Fetching / Mutations

Data Fetching, Mutations?

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

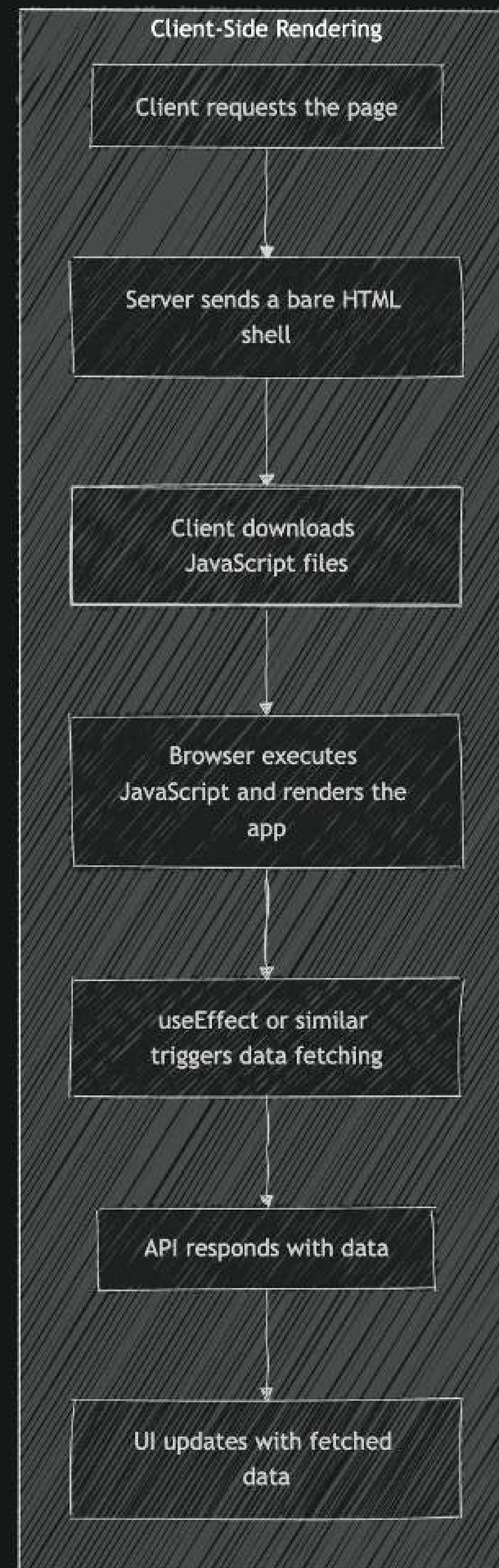
CSR vs SSR

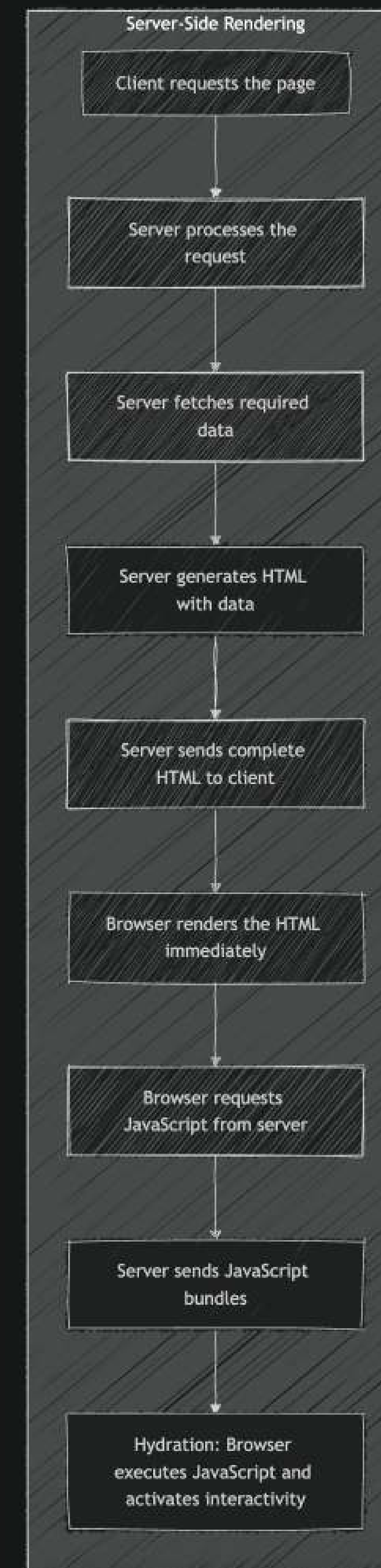
Client-Side Rendering (CSR)

- 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.





https://nextjs.org/docs/app/building-your-application/routing/loading-ui-and-streaming

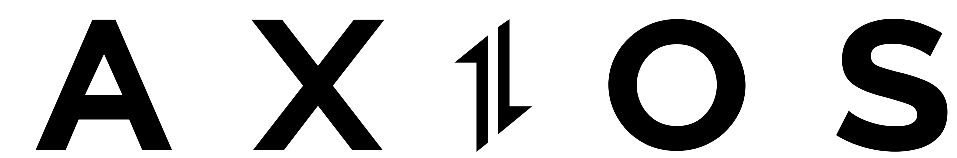
Data Fetching · CSR
Fetch

App.tsx

```tsx
import * as React from "react";

interface Post {
  id: string;
  title: string;
  views: string;
}

function App() {
  const [items, setItems] = React.useState<Post[]>([]);
  const [isLoading, setIsLoading] = React.useState(false);
  const [error, setError] = React.useState<Error>();

  React.useEffect(() => {
    const fetchData = async () => {
      setIsLoading(true);

      try {
        const response = await fetch(`${import.meta.env.VITE_API_URL}/posts`);

        if (!response.ok) {
          throw new Error(`Fetch error: ${response.statusText}`);
        }

        const json = (await response.json()) as Post[];

        setItems(json);
      } catch (error) {
        setError(error as Error);
      }

      setIsLoading(false);
    };

    fetchData();
  }, []);

  if (isLoading) {
    return "Loading...";
  }

  if (error) {
    return "Something went wrong, please try again later.";
  }

  return (
    <ul>
      {items.map((item) => (
        <li key={item.id}>{item.title}</li>
      ))}
    </ul>
  );
}

export default App;
```

API Clients

Simplifying API requests with an API Client



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.

axios.ts

```ts
import axios from "axios";

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

ky.ts

```ts
import ky from "ky";

export const getPosts = async ({ page = 1, perPage = 5 }: GetPostsInput) => {
  return ky
    .get("posts", {
      prefixUrl: process.env.API_URL,
      searchParams: { _page: page, _per_page: perPage, _sort: "-createdAt" },
    })
    .json<List<Post>>();
};
```

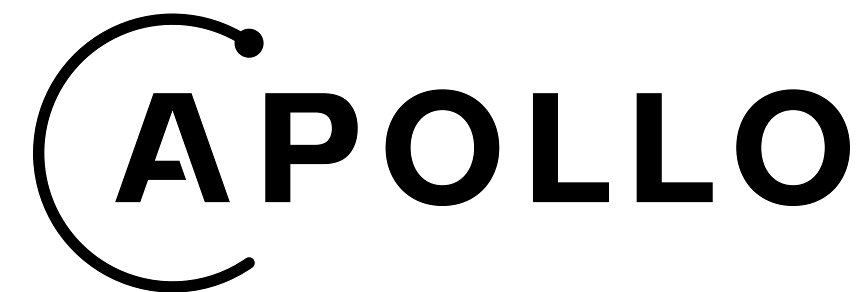
wretch.ts

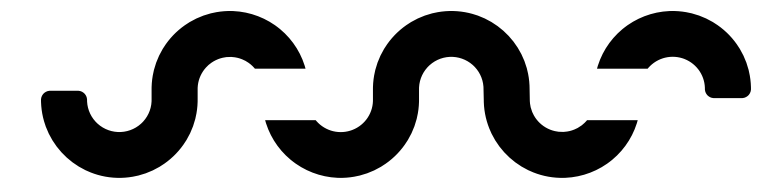
```ts
import wretch from "wretch";

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

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

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

// ...

const { data, isLoading, error } = useQuery({
  queryKey: ["posts"],
  queryFn: () => getPosts({ page: 1, perPage: 5 }),
});
```

apollo.ts

```ts
import { useQuery, gql } from "@apollo/client";

const GET_POSTS = gql`
  query GetPosts($page: Int, $perPage: Int) {
    posts(page: $page, perPage: $perPage) {
      id
      title
    }
  }
`;

// ...

const { data, loading, error } = useQuery(GET_POSTS, {
  variables: { page: 1, perPage: 5 },
});
```

swr.ts

```ts
import useSWR from "swr";

const fetcher = (url: string) => getPosts({ page: 1, perPage: 5 });

// ...

const { data, isLoading, error } = useSWR("posts", fetcher);
```

Data Fetching : CSR
React Query

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.

```tsx
import { useQuery } from "@tanstack/react-query";
import ky from "ky";

interface Post {
  id: string;
  title: string;
  views: string;
}

const getPosts = () =>
  ky<Post[]>(`${import.meta.env.VITE_API_URL}/posts`).json();

function App() {
  const { data, isLoading, error } = useQuery({
    queryKey: ["posts"],
    queryFn: getPosts,
  });

  if (isLoading) {
    return "Loading...";
  }

  if (error) {
    return "Something went wrong, please try again later.";
  }

  return (
    <ul>
      {data?.map((item) => (
        <li key={item.id}>{item.title}</li>
      ))}
    </ul>
  );
}

export default App;
```

Caching

Request Deduplication

Window Focus Refetching

Handling Loading States

Pause while Offline

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

Retries

and Error States

Exponential Backoff

Tracked Queries

stale-while-revalidate

Global State Management

Structural Sharing

Automatic Garbage Collection

https://tkdodo.eu/blog/react-query-api-design-lessons-learned

Data Fetching · CSR (React Query)

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.

https://tkdodo.eu/blog/inside-react-query

App.tsx

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

interface Post {
  id: string;
  title: string;
  views: string;
}

const getPosts = () =>
  ky<Post[]>(`${import.meta.env.VITE_API_URL}/posts`).json();

function App() {
  const { data, isLoading, error } = useQuery({
    queryKey: ["posts"],
    queryFn: getPosts,
  });

  if (isLoading) {
    return "Loading...";
  }

  if (error) {
    return "Something went wrong, please try again later.";
  }

  return (
    <ul>
      {data?.map((item) => (
        <li key={item.id}>{item.title}</li>
      ))}
    </ul>
  );
}

export default App;
```

App.tsx

```tsx
import * as React from "react";

interface Post {
  id: string;
  title: string;
  views: string;
}

function App() {
  const [items, setItems] = React.useState<Post[]>([]);
  const [isLoading, setIsLoading] = React.useState(false);
  const [error, setError] = React.useState<Error>();

  React.useEffect(() => {
    const fetchData = async () => {
      setIsLoading(true);

      try {
        const response = await fetch(`${import.meta.env.VITE_API_URL}/posts`);

        if (!response.ok) {
          throw new Error(`Fetch error: ${response.statusText}`);
        }

        const json = (await response.json()) as Post[];

        setItems(json);
      } catch (error) {
        setError(error as Error);
      }

      setIsLoading(false);
    };

    fetchData();
  }, []);

  if (isLoading) {
    return "Loading...";
  }

  if (error) {
    return "Something went wrong, please try again later.";
  }

  return (
    <ul>
      {items.map((item) => (
        <li key={item.id}>{item.title}</li>
      ))}
    </ul>
  );
}

export default App;

```

App.tsx

```tsx
import { useQuery } from "@tanstack/react-query";
import ky from "ky";

interface Post {
  id: string;
  title: string;
  views: string;
}

const getPosts = () =>
  ky<Post[]>(`${import.meta.env.VITE_API_URL}/posts`).json();

function App() {
  const { data, isLoading, error } = useQuery({
    queryKey: ["posts"],
    queryFn: getPosts,
  });

  if (isLoading) {
    return "Loading...";
  }

  if (error) {
    return "Something went wrong, please try again later.";
  }

  return (
    <ul>
      {data?.map((item) => (
        <li key={item.id}>{item.title}</li>
      ))}
    </ul>
  );
}

export default App;
```

Mutations · CSR
Fetch

App.tsx

```tsx
import * as React from "react";

interface Post {
  id: string;
  title: string;
  views: string;
}

function App() {
  const [posts, setPosts] = React.useState<Post[]>([]);
  const [isLoading, setIsLoading] = React.useState(false);
  const [error, setError] = React.useState<Error>();

  const [value, setValue] = React.useState("");

  const fetchPosts = React.useCallback(async () => {
    setIsLoading(true);

    try {
      const response = await fetch(`${import.meta.env.VITE_API_URL}/posts`);

      if (!response.ok) {
        throw new Error(`Fetch error: ${response.statusText}`);
      }

      const posts = (await response.json()) as Post[];

      setPosts(posts);
    } catch (error) {
      setError(error as Error);
    }

    setIsLoading(false);
  }, []);

  React.useEffect(() => {
    fetchPosts();
  }, [fetchPosts]);

  if (isLoading) {
    return "Loading...";
  }

  if (error) {
    return "Something went wrong, please try again later.";
  }

  const handleSubmit = async (event: React.FormEvent<HTMLFormElement>) => {
    event.preventDefault();

    try {
      const response = await fetch(`${import.meta.env.VITE_API_URL}/posts`, {
        method: "POST",
        body: JSON.stringify({ title: value }),
        headers: {
          "content-type": "application/json",
        },
      });

      if (!response.ok) {
        throw new Error(`Fetch error: ${response.statusText}`);
      }

      await fetchPosts();

      setValue("");
    } catch (error) {
      setError(error as Error);
    }
  };

  return (
    <div>
      <h1>Mutation Basic CSR</h1>
      <form onSubmit={handleSubmit}>
        <input
          value={value}
          onChange={(event) => setValue(event.target.value)}
        />
        <button type="submit">Add</button>
      </form>
      <ul>
        {posts.map((item) => (
          <li key={item.id}>{item.title}</li>
        ))}
      </ul>
    </div>
  );
}

export default App;
```

App.tsx

```tsx
import * as React from "react";

interface Post {
  id: string;
  title: string;
  views: string;
}

function App() {
  // ...

  const handleSubmit = async (event: React.FormEvent<HTMLFormElement>) => {
    event.preventDefault();

    try {
      const response = await fetch(`${import.meta.env.VITE_API_URL}/posts`, {
        method: "POST",
        body: JSON.stringify({ title: value }),
        headers: {
          "content-type": "application/json",
        },
      });

      if (!response.ok) {
        throw new Error(`Fetch error: ${response.statusText}`);
      }

      // fetch `posts` to ensure `posts` are updated after mutation.
      await fetchPosts();

      setValue("");
    } catch (error) {
      setError(error as Error);
    }
  };

  return (
    // ...
  );
}

export default App;
```

Mutations · CSR
React Query

App.tsx

```tsx
import { useMutation, useQuery, useQueryClient } from "@tanstack/react-query";
import ky from "ky";
import * as React from "react";

interface Post {
  id: string;
  title: string;
  views: string;
}

const getPosts = () =>
  ky<Post[]>(`${import.meta.env.VITE_API_URL}/posts`).json();

const createPost = (title: string) =>
  ky
    .post<Post>(`${import.meta.env.VITE_API_URL}/posts`, {
      json: { title },
    })
    .json();

function App() {
  const [value, setValue] = React.useState("");

  const queryClient = useQueryClient();

  const { data, isLoading, error } = useQuery({
    queryKey: ["posts"],
    queryFn: getPosts,
  });

  const { mutate } = useMutation({
    mutationFn: createPost,
    onSuccess: async () => {
      await queryClient.invalidateQueries({ queryKey: ["posts"] });
      setValue("");
    },
  });

  if (isLoading) {
    return "Loading...";
  }

  if (error) {
    return "Something went wrong, please try again later.";
  }

  const handleSubmit = async (event: React.FormEvent<HTMLFormElement>) => {
    event.preventDefault();

    mutate(value);
  };

  return (
    <div>
      <h1>Mutation React Query CSR</h1>
      <form onSubmit={handleSubmit}>
        <input
          value={value}
          onChange={(event) => setValue(event.target.value)}
        />
        <button type="submit">Add</button>
      </form>
      <ul>
        {data?.map((item) => (
          <li key={item.id}>{item.title}</li>
        ))}
      </ul>
    </div>
  );
}

export default App;
```

App.tsx

```tsx
// ...

function App() {
  const [value, setValue] = React.useState("");

  const queryClient = useQueryClient();

  const { data, isLoading, error } = useQuery({
    queryKey: ["posts"],
    queryFn: getPosts,
  });

  const { mutate } = useMutation({
    mutationFn: createPost,
    onSuccess: async () => {
      await queryClient.invalidateQueries({ queryKey: ["posts"] });
      setValue("");
    },
  });

  if (isLoading) {
    return "Loading...";
  }

  if (error) {
    return "Something went wrong, please try again later.";
  }

  const handleSubmit = async (event: React.FormEvent<HTMLFormElement>) => {
    event.preventDefault();

    mutate(value);
  };

  return (
    // ...
  );
}

export default App;
```

Data Fetching / Mutations · CSR
Fetch vs React Query

TS posts.ts

```ts
export const useGetPosts = ({ page = 1, perPage = 5 }: GetPostsInput) => {
  const [data, setData] = React.useState<List<Post>>();
  const [isLoading, setIsLoading] = React.useState(false);
  const [error, setError] = React.useState<Error>();

  const fetchPosts = React.useCallback(async () => {
    setIsLoading(true);

    try {
      const response = await fetch(
        `${import.meta.env.VITE_API_URL}/posts?_page=${page}&_per_page=${perPage}&_sort=-createdAt`,
      );

      if (!response.ok) {
        throw new Error(`Fetch error: ${response.statusText}`);
      }

      const data = (await response.json()) as List<Post>;

      setData(data);
    } catch (error) {
      setError(error as Error);
    }

    setIsLoading(false);
  }, [page, perPage]);

  React.useEffect(() => {
    fetchPosts();
  }, [fetchPosts]);

  return {
    data,
    isLoading,
    error,
    fetchPosts,
  };
};
```

TS posts.ts

```ts
const getPosts = ({ page = 1, perPage = 5 }: GetPostsInput) =>
  ky<List<Post>>(`${import.meta.env.VITE_API_URL}/posts`, {
    searchParams: {
      _page: page,
      _per_page: perPage,
      _sort: "-createdAt",
    },
  }).json();

export const useGetPostsQuery = (input: GetPostsInput) =>
  useQuery({
    queryKey: ["posts", input],
    queryFn: () => getPosts(input),
  });
```

App.tsx

```tsx
// ...

function App() {
  // ...

  const { data, isLoading, error, fetchPosts } = useGetPosts({
    page: pagination.pageIndex + 1,
    perPage: pagination.pageSize,
  });
  const { createPost } = useCreatePost();
  const { deletePost } = useDeletePost();

  const handleSubmit = async (event: React.FormEvent<HTMLFormElement>) => {
    event.preventDefault();
    await createPost(value);
    await fetchPosts();
    setValue("");
  };

  const handleDeletePost = async (postId: string) => {
    await deletePost(postId);
    await fetchPosts();
  };

  // ...

  const [value, setValue] = React.useState("");

  if (isLoading) {
    return "Loading...";
  }

  if (error) {
    return "Something went wrong, please try again later.";
  }

  return (
    // ...
  );
}

export default App;
```

App.tsx

```tsx
// ...

function App() {
  // ...

  const { data, isLoading, error } = useGetPostsQuery({
    page: pagination.pageIndex + 1,
    perPage: pagination.pageSize,
  });
  const { mutate: createPost } = useCreatePostMutation();
  const { mutate: deletePost } = useDeletePostMutation();

  const handleSubmit = (event: React.FormEvent<HTMLFormElement>) => {
    event.preventDefault();
    createPost(value);
    setValue("");
  };

  const handleDeletePost = (postId: string) => {
    deletePost(postId);
  };

  // ...

  const [value, setValue] = React.useState("");

  if (isLoading) {
    return "Loading...";
  }

  if (error) {
    return "Something went wrong, please try again later.";
  }

  return (
    // ...
  );
}

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

```ts
"use server";

// ...

export const getPosts = async ({ page = 1, perPage = 5 }: GetPostsInput) => {
  const response = await fetch(
    `${process.env.API_URL}/posts?_page=${page}&_per_page=${perPage}&_sort=-createdAt`,
  );

  if (!response.ok) {
    throw new Error(`Fetch error: ${response.statusText}`);
  }

  const data = (await response.json()) as List<Post>;

  return data;
};

// ...
```

page.tsx

```tsx
import { getPosts } from "@/actions/posts";
import { Posts } from "@/components/posts";

export default async function Home({
  searchParams,
}: {
  searchParams: Promise<{ [key: string]: string | undefined }>;
}) {
  const { page = "1", perPage = "5" } = await searchParams;

  const data = await getPosts({
    page: Number(page),
    perPage: Number(perPage),
  });

  return <Posts data={data} />;
}
```

posts.tsx

```tsx
"use client";

// ...

import { usePathname, useRouter, useSearchParams } from "next/navigation";

// ...

interface PostsProps {
  data: List<Post>;
}

export const Posts = ({ data }: PostsProps) => {
  const searchParams = useSearchParams();
  const router = useRouter();
  const pathname = usePathname();

  const pagination = React.useMemo(
    () =>
      ({
        pageIndex: Number(searchParams.get("page") ?? "1") - 1,
        pageSize: Number(searchParams.get("perPage") ?? "5"),
      }) as const satisfies PaginationState,
    [searchParams],
  );

  // We could've used `useActionState` here. Visit https://github.com/vercel/next.js/tree/canary/examples/next-forms for more examples.
  const handleSubmit = async (event: React.FormEvent<HTMLFormElement>) => {
    event.preventDefault();
    await createPost(value);
    setValue("");
  };

  const handleDeletePost = async (postId: string) => {
    await deletePost(postId);
  };

  const handlePaginationChange = React.useCallback(
    (updater: Updater<PaginationState>) => {
      if (typeof updater === "function") {
        const newPagination = updater(pagination);
        const params = new URLSearchParams(searchParams.toString());
        params.set("page", `${newPagination.pageIndex + 1}`);
        params.set("perPage", `${newPagination.pageSize}`);
        router.push(`${pathname}?${params.toString()}`);
      }
    },
    [pagination, router.push, searchParams, pathname],
  );

  const [value, setValue] = React.useState("");

  // ...

  return (
    // ...
  );
};
```

Data Fetching · CSR (React Query)

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.

TS post.ts

```ts
"use server";

import { revalidatePath } from "next/cache";

// ...

export const createPost = async (title: string) => {
  const response = await fetch(`${process.env.API_URL}/posts`, {
    method: "POST",
    body: JSON.stringify({ title, createdAt: new Date().toISOString() }),
    headers: {
      "content-type": "application/json",
    },
  });

  if (!response.ok) {
    throw new Error(`Fetch error: ${response.statusText}`);
  }

  const post = (await response.json()) as Post;

  revalidatePath("/");

  return post;
};
```

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

```tsx

// Using server api.
import { getPosts } from "@/apis/posts.server";

export const getPostsQueryOptions = (...input: Parameters<typeof getPosts>) =>
  queryOptions({
    queryKey: ["posts", ...input],
    queryFn: () => getPosts(...input),
    placeholderData: {
      first: 0,
      prev: null,
      next: null,
      last: 0,
      pages: 0,
      items: 0,
      data: [],
    },
  });
```

page.tsx

```tsx
import { Posts } from "@/components/posts";
import { getPostsQueryOptions } from "@/queries/posts.server";
import {
  HydrationBoundary,
  QueryClient,
  dehydrate,
} from "@tanstack/react-query";
import * as React from "react";

export default async function Home({
  searchParams,
}: {
  searchParams: Promise<{ [key: string]: string | undefined }>;
}) {
  const { page = "1", perPage = "5" } = await searchParams;

  const queryClient = new QueryClient();

  await queryClient.prefetchQuery(
    getPostsQueryOptions({
      page: Number(page),
      perPage: Number(perPage),
    }),
  );

  return (
    <HydrationBoundary state={dehydrate(queryClient)}>
      <React.Suspense fallback={<>Loading...</>}>
        <Posts />
      </React.Suspense>
    </HydrationBoundary>
  );
}

```

posts.tsx

```tsx
"use client";

import {
  getPostsQueryOptions,
  useCreatePostMutation,
  useDeletePostMutation,
} from "@/queries/posts.client";

// ...

export const Posts = () => {
  // ...

  const { data } = useSuspenseQuery(
    getPostsQueryOptions({
      page,
      perPage,
    }),
  );

  const { mutate: createPost, isPending, error } = useCreatePostMutation();

  const { mutate: deletePost } = useDeletePostMutation();

  const pagination = React.useMemo(
    () =>
      ({
        pageIndex: page - 1,
        pageSize: perPage,
      }) as const satisfies PaginationState,
    [page, perPage],
  );

  const handleSubmit = async (event: React.FormEvent<HTMLFormElement>) => {
    event.preventDefault();
    createPost(value);
    setValue("");
  };

  const handleDeletePost = async (postId: string) => {
    deletePost(postId);
  };

  const handlePaginationChange = React.useCallback(
    (updater: Updater<PaginationState>) => {
      if (typeof updater === "function") {
        const newPagination = updater(pagination);
        const params = new URLSearchParams(searchParams.toString());
        params.set("page", `${newPagination.pageIndex + 1}`);
        params.set("perPage", `${newPagination.pageSize}`);
        router.push(`${pathname}?${params.toString()}`);
      }
    },
    [pagination, router.push, searchParams, pathname],
  );

  const [value, setValue] = React.useState("");

  // ...

  return (
    // ...
  );
};
```

TS posts.ts

```ts
// src/actions/posts.ts

// Server actions with "use server";
"use server";

import * as posts from "@/apis/posts.server";

export const createPost = async (title: string) => {
  if (!title) {
    return { message: "Title is required" };
  }

  return posts.createPost(title);
};

// src/lib/react-query.ts

// Typescript helper function used for type inference.
export function mutationOptions<
  TData = unknown,
  TError = DefaultError,
  TVariables = void,
  TContext = unknown,
>(
  options: UseMutationOptions<TData, TError, TVariables, TContext>,
): UseMutationOptions<TData, TError, TVariables, TContext> {
  return options;
}

// src/queries/posts.client.ts

const createPostMutationOptions = mutationOptions({
  mutationFn: createPost,
});

export const useCreatePostMutation = ({
  onSuccess,
  ...options
}: Omit<typeof createPostMutationOptions, "mutationFn"> = {}) => {
  const queryClient = useQueryClient();

  return useActionMutation({
    ...createPostMutationOptions,
    // Idiomatic server actions should not throw, useMutation expects error to be thrown.
    transformError: (result) => {
      if ("message" in result) {
        // We take the result of the server action call, and return an error so that `useActionMutation` throws it.
        return new Error(result.message);
      }
      return null;
    },
    onSuccess: (...args) => {
      queryClient.invalidateQueries(getPostsQueryOptions());

      return onSuccess?.(...args);
    },
    ...options,
  });
};
```

Optimistic Updates
React Query

posts.tsx

```tsx
"use client";

// ...

export const Posts = () => {
  // ...

  const {
    mutate: createPost,
    variables: optimisticTitle,
    isPending,
  } = useCreatePostMutation({
    onError: (error) => alert(error.message),
  });

  // ...

  const table = useReactTable({
    data:
      (isPending
        ? [
            {
              id: "temp",
              title: optimisticTitle,
              createdAt: "",
              views: "0",
            },
            ...data.data.slice(0, -1),
          ]
        : data?.data) ?? [],
    rowCount: isPending ? data?.items + 1 : data?.items,
    columns,
    getCoreRowModel: getCoreRowModel(),
    getRowId: (post) => post.id,
    manualPagination: true,
    onPaginationChange: handlePaginationChange,
    state: { pagination },
    meta: {
      onPostDelete: ({ row }) => handleDeletePost(row.id),
    },
  });

  return (
    // ...
  );
};
```

Infinite Scrolling
React Query

TS posts.ts

```ts


import { getPosts } from "@/apis/posts.client";
import { infiniteQueryOptions } from "@tanstack/react-query";

export const getPostsInfiniteQueryOptions = (
  ...input: Parameters<typeof getPosts>
) =>
  infiniteQueryOptions({
    queryKey: ["posts", ...input],
    queryFn: async ({ pageParam }) => {
      // Added this to simulate a slight delay to that we can see the infinite scrolling more clearly.
      await new Promise((res) => setTimeout(res, 500));
      return getPosts(pageParam);
    },
    initialPageParam: {
      page: 1,
      perPage: 10,
      ...input.at(0),
    },
    getNextPageParam: (lastPage, allPages, lastPageParam) => {
      if (!lastPage.next) {
        return null;
      }

      return {
        ...lastPageParam,
        page: lastPage.next,
      };
    },
    getPreviousPageParam: (firstPage, allPages, firstPageParam) => {
      if (!firstPage.prev) {
        return null;
      }

      return {
        ...firstPageParam,
        page: firstPage.prev,
      };
    },
  });
```

TS posts.tsx

```tsx
"use client";

import * as React from "react";

import { getPostsInfiniteQueryOptions } from "@/queries/posts.client";
import { useInfiniteQuery } from "@tanstack/react-query";
import { InView } from "react-intersection-observer";

export const Posts = () => {
  const { data, hasNextPage, isFetching, fetchNextPage } = useInfiniteQuery(
    getPostsInfiniteQueryOptions(),
  );

  const items = data?.pages?.flatMap((page) => page.data) ?? [];

  return (
    <div>
      <h1>Infinite Scrolling</h1>
      <div
        style={{
          maxHeight: 300,
          overflowY: "scroll",
          border: "1px solid black",
        }}
      >
        <ul>
          {items.map((item) => (
            <li key={item.id}>{item.title}</li>
          ))}
          {hasNextPage && !isFetching && (
            <InView
              onChange={(inView) => inView && !isFetching && fetchNextPage()}
            />
          )}
          {isFetching && "Loading..."}
        </ul>
      </div>
    </div>
  );
};
```

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.

eslint.config.js

```js
import pluginQuery from '@tanstack/eslint-plugin-query'

export default [
  ...pluginQuery.configs['flat/recommended'],
  // Any other config...
]
```

posts.ts

```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: [],
    },
  });
```

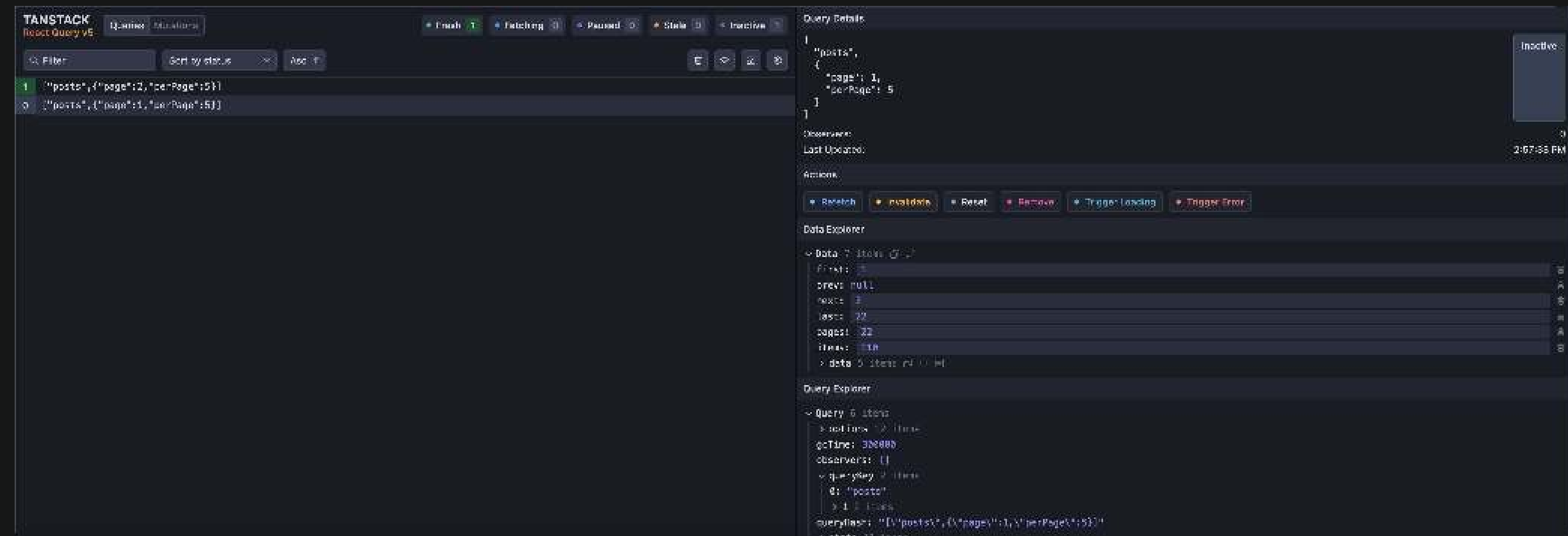
React Query Complementary Tools

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!

App.tsx

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

function App() {
  return (
    <QueryClientProvider client={queryClient}>
      {/* The rest of your application */}
      <ReactQueryDevtools initialIsOpen={false} />
    </QueryClientProvider>
  )
}
```



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/

10 Reputation

22 Longest streak

280 Posts read

Jun Wen Kwan

@junwenk • Aug 19

#webdev #javascript #react #open-source #frontend

daily.dev

https://linkedin.com/in/junwenk

https://forms.gle/rcmjcrXPa6TEJ2hf8

https://github.com/junwen-k