Experiment 3

Aim

Manage complex state with Redux or Context API

Theory

When to use what (opinionated)

- **Redux (use Redux Toolkit)**: Best for **complex, cross-cutting, frequently updated** state (entities, caching, optimistic updates, undo/redo). Predictable updates, great devtools, middleware, time-travel.
- **Context API**: Best for **stable, read-mostly** cross-cutting values (theme, i18n, auth session). Avoid for high-frequency granular updates—can cause tree-wide re-renders.

Redux basics (with Redux Toolkit, RTK)

- **Single source of truth**: Central store; updates via dispatched actions.
- **RTK slices**: Co-locate reducer + actions; Immer gives **mutable-looking** immutable updates.
- **Selectors**: Derive data; memoize with `createSelector` to prevent wasteful renders.
- **Normalization**: Store entities by id (`createEntityAdapter`) to keep reducers simple and updates efficient.
- **Async**: Use **RTK Query** (preferred) or `createAsyncThunk` for data fetching, caching, and invalidation.
- **DevEx**: Redux DevTools, middleware (logging, analytics), predictable flows useful at scale.

Context basics (without overusing it)

- **Providers** wrap app; `useContext` reads nearest value.
- **Performance**: Re-renders all consumers on value change. Mitigate with:
  - Split contexts (state vs actions).
  - Memoize provider values (`useMemo`, `useCallback`).
  - Use context **selectors** (e.g., `use-context-selector`) if updates are frequent.

- **Pattern**: Context for **config + coarse state**, local `useState` for component concerns, external libs for complex global state.

Best practices

- Prefer **RTK** over “raw” Redux; never hand-write action types/creators/reducers.
- Keep reducers **pure**, effects in thunks/RTK Query; avoid side effects in reducers.
- Co-locate slice logic with features; export **selectors** as the read API.
- Treat server state differently from client state; use **RTK Query or React Query** for server data.

Common pitfalls

- Overusing Context for fast-changing state → re-render storms.
- Storing **derived**/UI state (like “isModalOpen” per component) globally without need.
- Skipping selectors/memoization → unnecessary renders.
- Mixing data fetching logic inside components instead of a dedicated layer (RTK Query).

30% extra part

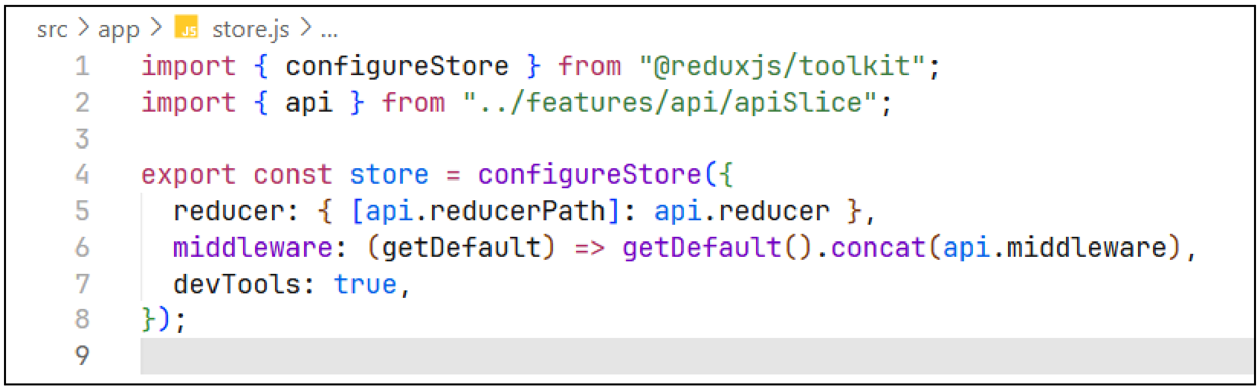
What we’re doing (RTK Query)

RTK Query centralizes **server-state** (fetching, caching, revalidation, mutations) with minimal code. We’ll define an **API slice** that auto-generates React hooks (e.g., `useGetPostsQuery`, `useAddPostMutation`) and handles cache lifecycles for us.

Goals (standard practice):

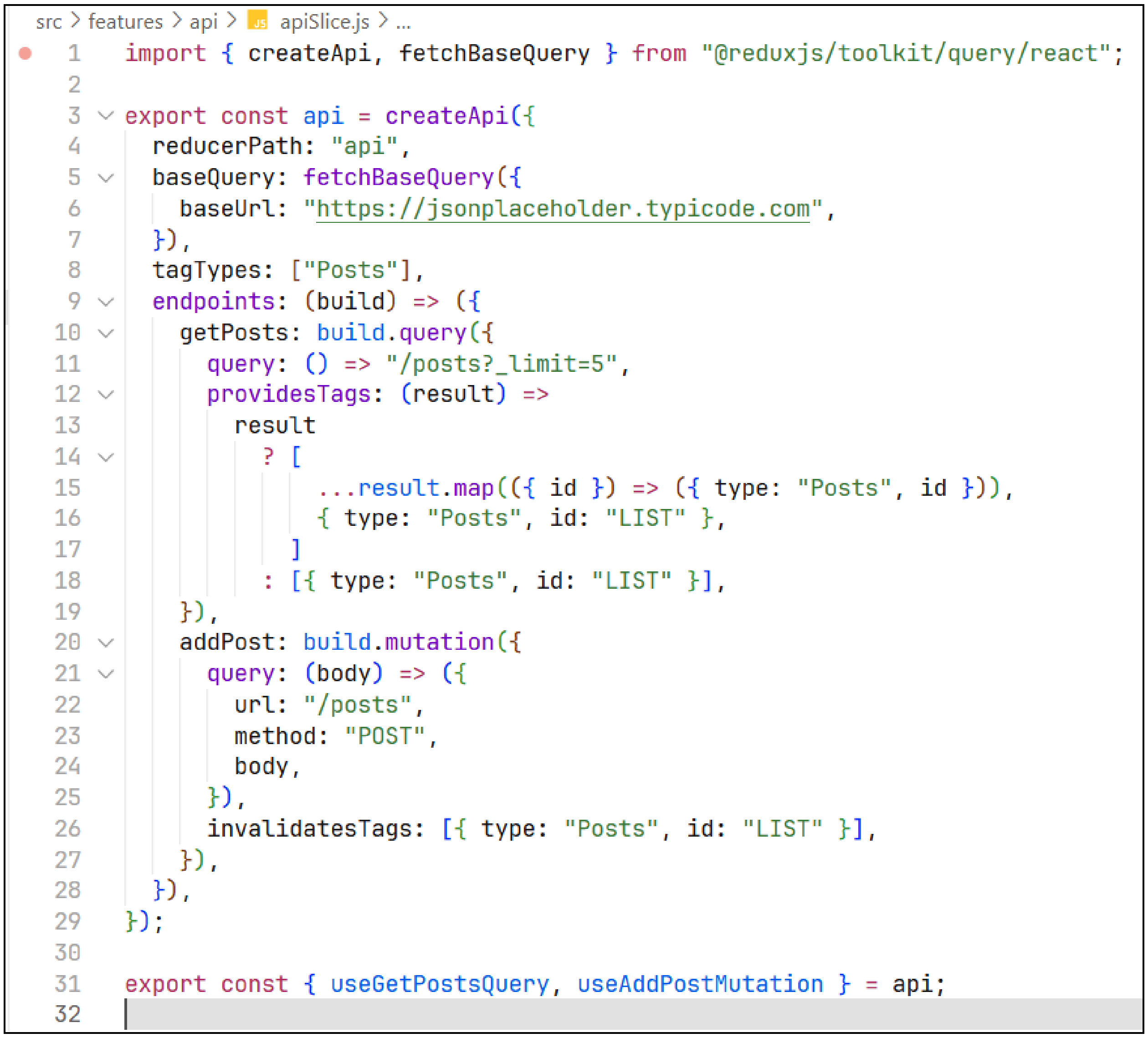
- Single API layer (`createApi` + `fetchBaseQuery`).
- Cache by endpoint + args; **tag** invalidation keeps lists fresh.
- Co-locate API definitions with a feature; keep components lean (just call hooks).

Source code


src > app > store.js > ...

```
import { configureStore } from "@reduxjs/toolkit";
import { api } from "../features/api/apiSlice";

export const store = configureStore({
  reducer: { [api.reducerPath]: api.reducer },
  middleware: (getDefault) => getDefault().concat(api.middleware),
  devTools: true,
});

```

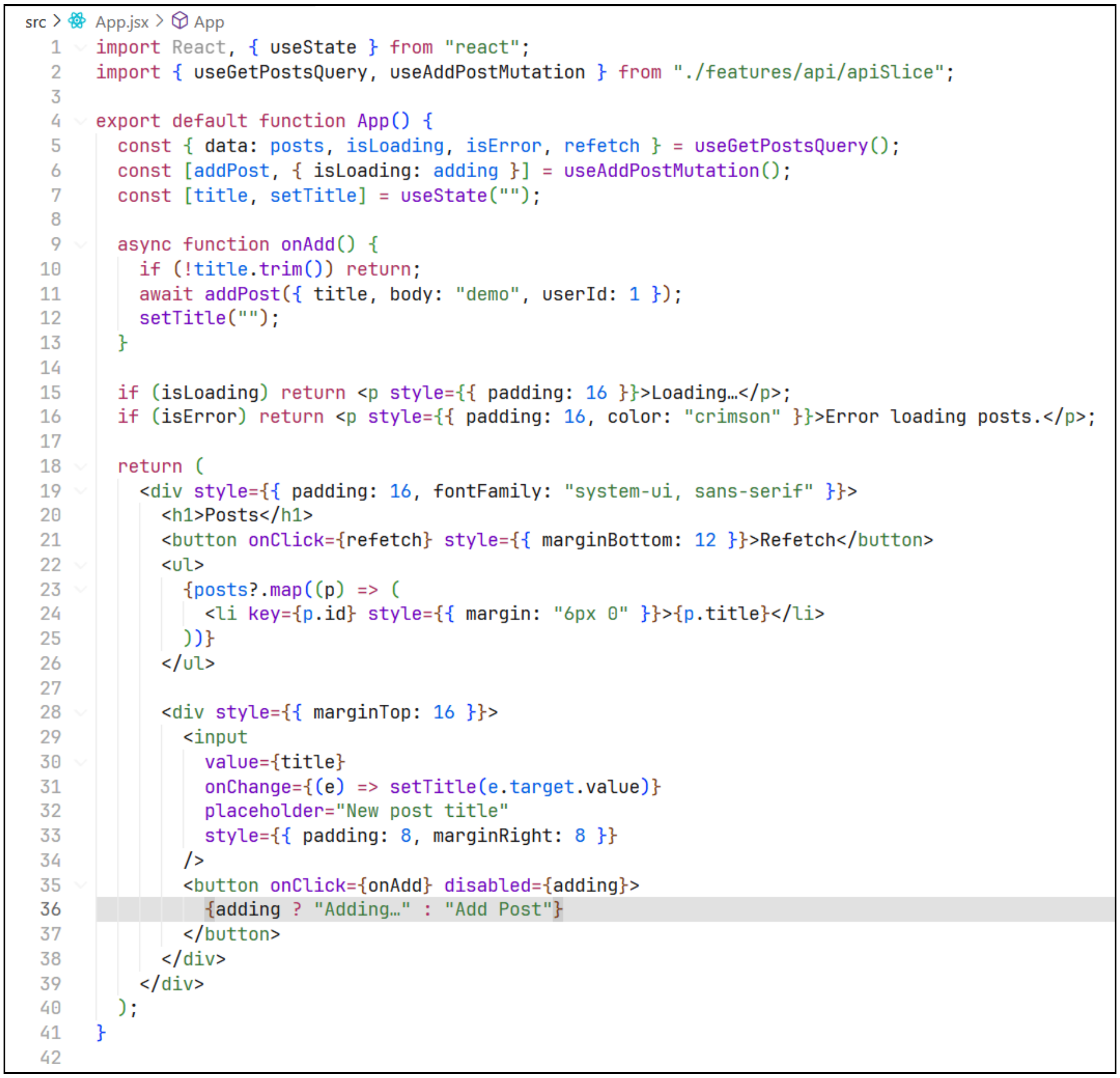
Fig 1.1



src > features > api > apiSlice.js > ...

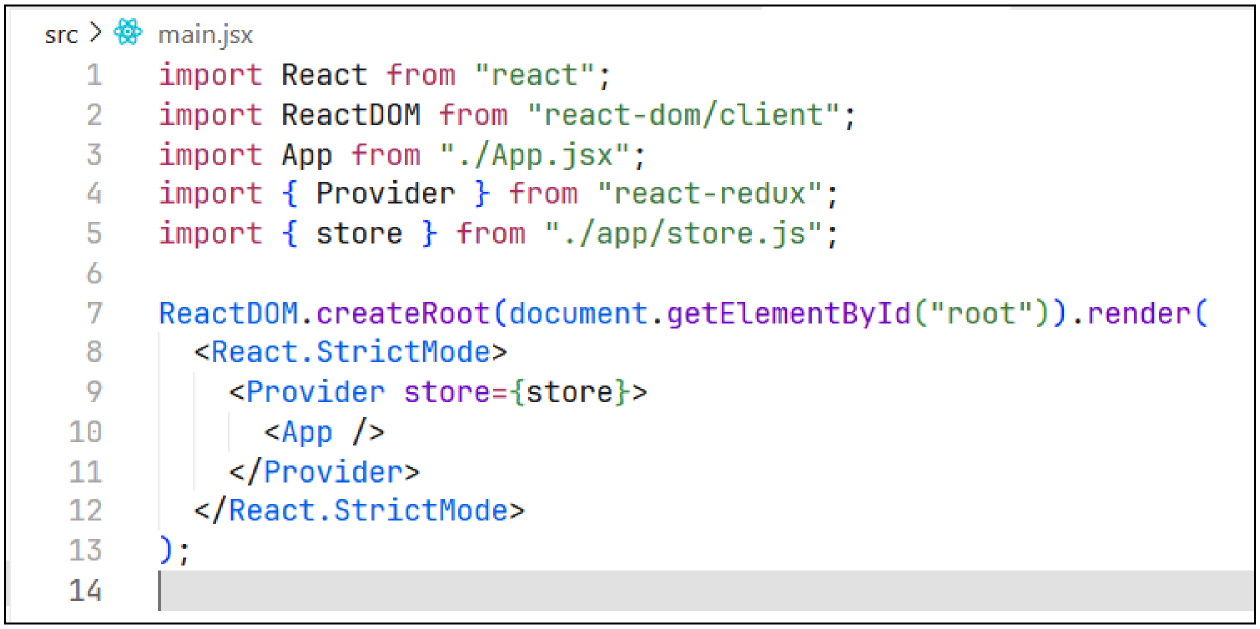
```
import { createApi, fetchBaseQuery } from "@reduxjs/toolkit/query/react";

export const api = createApi({
  reducerPath: "api",
  baseQuery: fetchBaseQuery({
    baseUrl: "https://jsonplaceholder.typicode.com",
  }),
  tagTypes: ["Posts"],
  endpoints: (build) => ({
    getPosts: build.query({
      query: () => "/posts?_limit=5",
      providesTags: (result) =>
        result
          ? [
              ...result.map(({ id }) => ({ type: "Posts", id })),
              { type: "Posts", id: "LIST" },
            ]
          : [{ type: "Posts", id: "LIST" }],
    }),
    addPost: build.mutation({
      query: (body) => ({
        url: "/posts",
        method: "POST",
        body,
      }),
      invalidatesTags: [{ type: "Posts", id: "LIST" }],
    }),
  }),
});

export const { useGetPostsQuery, useAddPostMutation } = api;

```

Fig 1.2


src > App.jsx > App

```jsx
import React, { useState } from "react";
import { useGetPostsQuery, useAddPostMutation } from "./features/api/apiSlice";

export default function App() {
  const { data: posts, isLoading, isError, refetch } = useGetPostsQuery();
  const [addPost, { isLoading: adding }] = useAddPostMutation();
  const [title, setTitle] = useState("");

  async function onAdd() {
    if (!title.trim()) return;
    await addPost({ title, body: "demo", userId: 1 });
    setTitle("");
  }

  if (isLoading) return <p style={{ padding: 16 }}>Loading…</p>;
  if (isError) return <p style={{ padding: 16, color: "crimson" }}>Error loading posts.</p>;

  return (
    <div style={{ padding: 16, fontFamily: "system-ui, sans-serif" }}>
      <h1>Posts</h1>
      <button onClick={refetch} style={{ marginBottom: 12 }}>Refetch</button>
      <ul>
        {posts?.map((p) => (
          <li key={p.id} style={{ margin: "6px 0" }}>{p.title}</li>
        ))}
      </ul>

      <div style={{ marginTop: 16 }}>
        <input
          value={title}
          onChange={(e) => setTitle(e.target.value)}
          placeholder="New post title"
          style={{ padding: 8, marginRight: 8 }}
        />
        <button onClick={onAdd} disabled={adding}>
          {adding ? "Adding…" : "Add Post"}
        </button>
      </div>
    </div>
  );
}
```

Fig 1.3

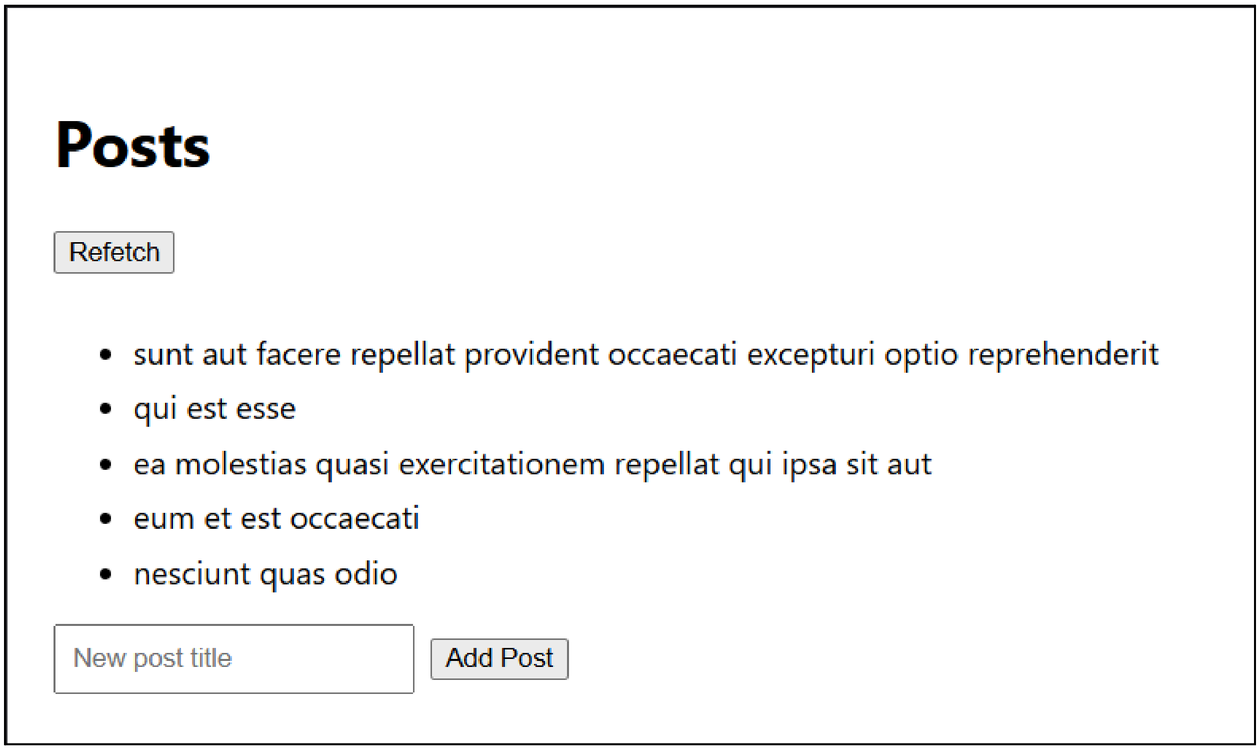


src > main.jsx

```jsx
import React from "react";
import ReactDOM from "react-dom/client";
import App from "./App.jsx";
import { Provider } from "react-redux";
import { store } from "./app/store.js";

ReactDOM.createRoot(document.getElementById("root")).render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>
);

```

Fig 1.4

Output



Posts

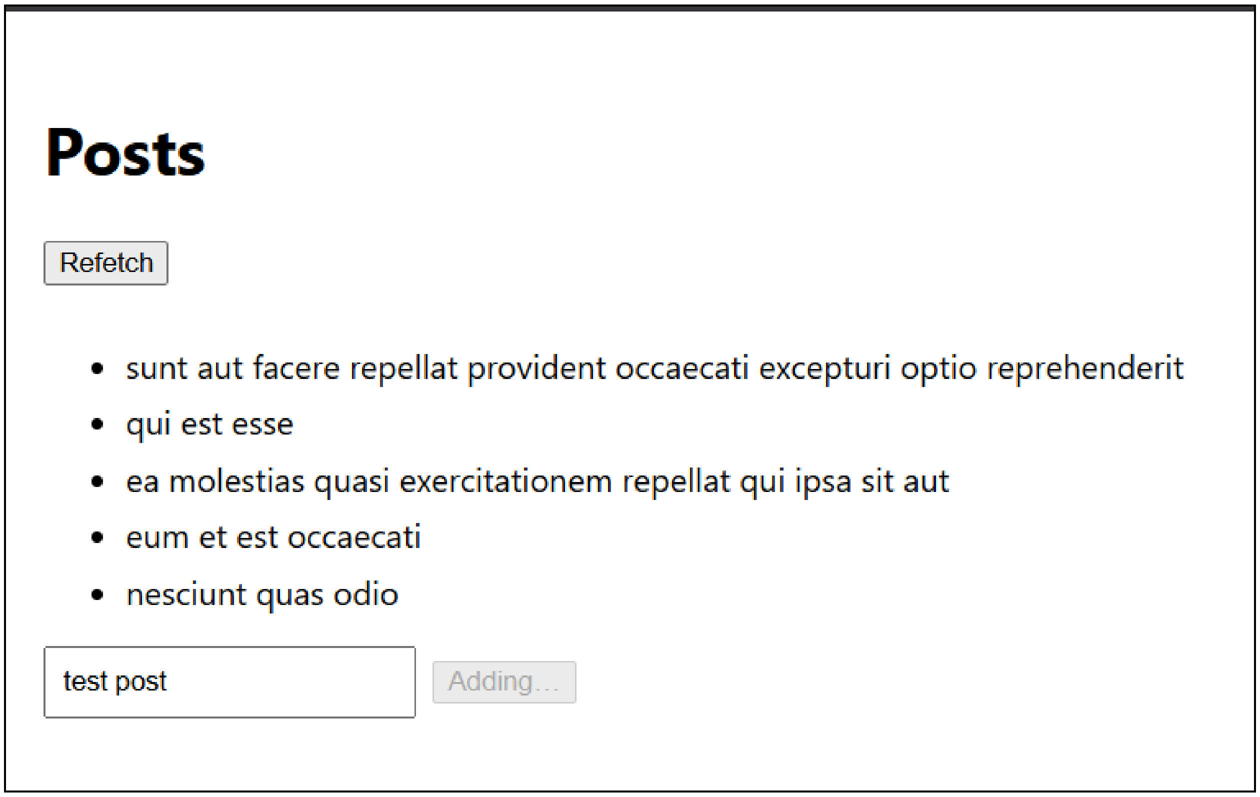
Refetch

- sunt aut facere repellat provident occaecati excepturi optio reprehenderit
- qui est esse
- ea molestias quasi exercitationem repellat qui ipsa sit aut
- eum et est occaecati
- nesciunt quas odio

New post title

Add Post

Fig 2.1

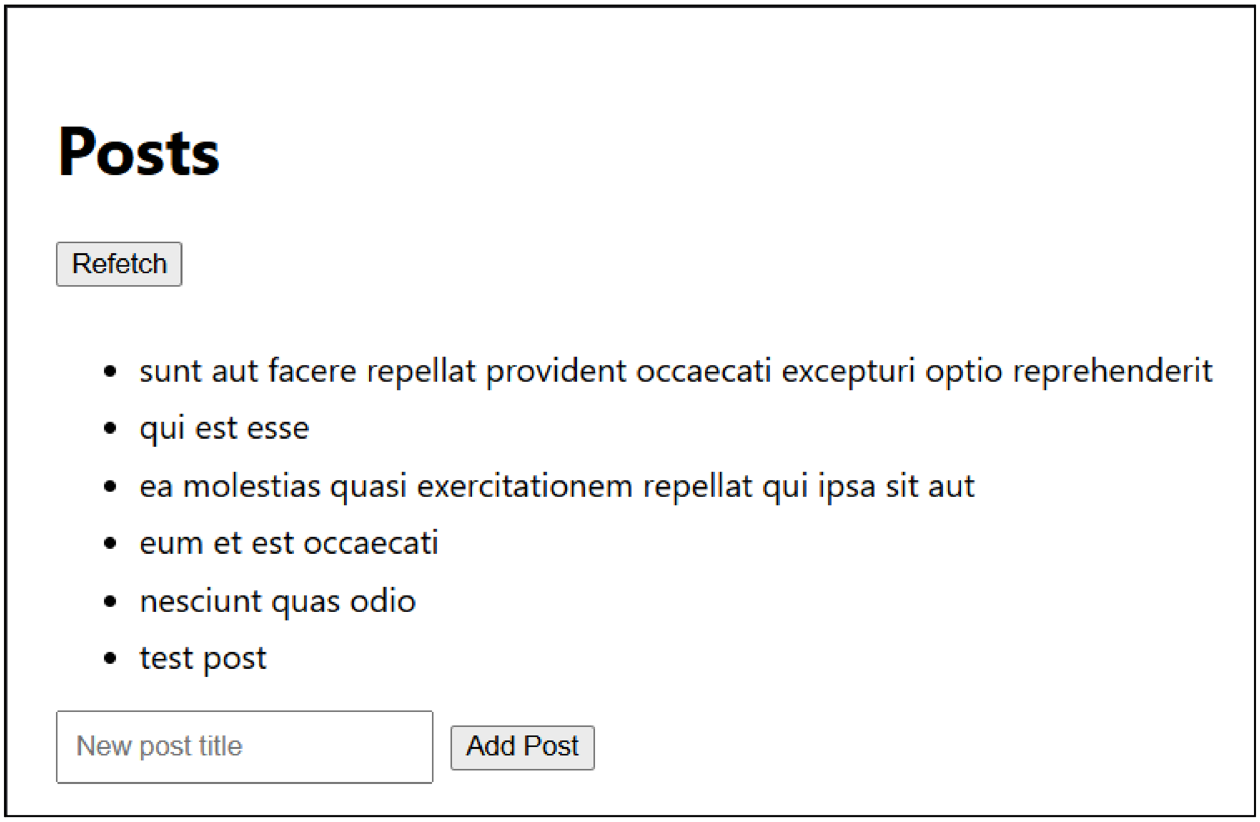

Posts

Refetch

- sunt aut facere repellat provident occaecati excepturi optio reprehenderit
- qui est esse
- ea molestias quasi exercitationem repellat qui ipsa sit aut
- eum et est occaecati
- nesciunt quas odio

test post

Fig 2.2


Posts

Refetch

- sunt aut facere repellat provident occaecati excepturi optio reprehenderit
- qui est esse
- ea molestias quasi exercitationem repellat qui ipsa sit aut
- eum et est occaecati
- nesciunt quas odio
- test post

Fig 2.3

Conclusion

Managing complex state with Redux and Context shows two distinct approaches: Context works best for stable, low-frequency shared values, while Redux (via Redux Toolkit) provides a structured, scalable solution for frequently updated, cross-cutting state. Extending Redux with **RTK Query** further streamlines server-state management by handling fetching, caching, and invalidation automatically, letting components remain focused on rendering. Together, these tools demonstrate how modern React apps can balance simplicity and scalability when state grows complex.