Day 26 — Redux Toolkit (RTK) + RTK Query (Modern 2025)

Week 4 — State Libraries & Backend Integration

Goal: Learn the modern, recommended way to manage app state with **Redux Toolkit** and to handle server state & data-fetching with **RTK Query**. Small bite-sized examples, best practices (2025), and exercises included.

Why Redux Toolkit (RTK)? — Short answer

- RTK is the *official*, modern, batteries-included way to use Redux.
- It removes boilerplate (no manual action types, reducers wiring).
- Works well with large apps where global state, caching, and predictable flows are required.

Use RTK when: - You need predictable global state shared across many components. - You want normalized state and time-travel debugging (rare now but useful). - Your app needs advanced patterns (undo, complex selectors).

For most server-driven UI, pair RTK with **RTK Query** — it handles caching, deduping, polling, and mutations out of the box.

Quick install

```
npm install @reduxjs/toolkit react-redux
# RTK Query comes built into @reduxjs/toolkit
```

If you prefer TypeScript, install types for react-redux: `npm i -D @types/react-redux` (but RTK is TS-first-ready).

Part A — Redux Toolkit Basics (tiny and modern)

1) Create the store

📌 `src/app/store.js`

```
import { configureStore } from '@reduxjs/toolkit';
import counterReducer from '../features/counter/counterSlice';

export const store = configureStore({
  reducer: {
    counter: counterReducer,
    // add other slice reducers here
  },
});
```

2) Create a slice (feature)

📌 `src/features/counter/counterSlice.js`

```
import { createSlice } from '@reduxjs/toolkit';

const initialState = { value: 0 };

const counterSlice = createSlice({
  name: 'counter',
  initialState,
  reducers: {
    increment(state) { state.value += 1 },
    decrement(state) { state.value -= 1 },
    incrementByAmount(state, action) { state.value += action.payload }
  }
});

export const { increment, decrement, incrementByAmount } =
counterSlice.actions;
export default counterSlice.reducer;
```

3) Provide the store to React

📌 `src/index.js`

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

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

4) Use hooks in components

```
import { useSelector, useDispatch } from 'react-redux';
import { increment, decrement } from '../features/counter/counterSlice';

function Counter() {
  const value = useSelector(state => state.counter.value);
  const dispatch = useDispatch();

  return (
    <div>
      <button onClick={() => dispatch(decrement())}>-</button>
      <span>{value}</span>
      <button onClick={() => dispatch(increment())}>+</button>
    </div>
  );
}
```

🔗 **Small problems / practice steps** 1. Create a `todoSlice` with `addTodo`, `toggleTodo`, `removeTodo` — keep todos normalized (array of ids + entities or simple array for now).
2. Create selectors: `selectTodos`, `selectCompletedCount` (use `createSelector` if heavy computation).

Part B — RTK Query (modern server-state management)

Why RTK Query? - Auto-generated hooks for endpoints (`useGetTasksQuery`, `useAddTaskMutation`) — no more manual `useEffect` fetches. - Built-in caching, deduping, invalidation, optimistic updates, polling. - Integrates cleanly with the RTK store.

1) Add API slice

📌 `src/app/api.js`

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

export const api = createApi({
  reducerPath: 'api',
  baseQuery: fetchBaseQuery({ baseUrl: 'http://localhost:8000' }), // your
FastAPI URL
  tagTypes: ['Tasks'],
  endpoints: (builder) => ({
    getTasks: builder.query({
      query: () => '/gettasks',
      providesTags: (result = []) =>
        result
```

```
          ? [
              ...result.map(({ id }) => ({ type: 'Tasks', id })),
              { type: 'Tasks', id: 'LIST' },
            ]
          : [{ type: 'Tasks', id: 'LIST' }],
    }),
    addTask: builder.mutation({
      query: (task) => ({ url: '/addtask', method: 'POST', body: task }),
      invalidatesTags: [{ type: 'Tasks', id: 'LIST' }],
    }),
    updateTask: builder.mutation({
      query: ({ id, ...patch }) => ({ url: `/updatetask/${id}`, method:
'PUT', body: patch }),
      invalidatesTags: (result, error, { id }) => [{ type: 'Tasks', id }],
    }),
    deleteTask: builder.mutation({
      query: (id) => ({ url: `/deltask/${id}`, method: 'DELETE' }),
      invalidatesTags: (result, error, id) => [{ type: 'Tasks', id }, {
type: 'Tasks', id: 'LIST' }],
    }),
  }),
});

export const { useGetTasksQuery, useAddTaskMutation, useUpdateTaskMutation,
useDeleteTaskMutation } = api;
```

Note: I matched the endpoints to your FastAPI routes (`/gettasks`, `/addtask`, `/updatetask/{id}`, `/deltask/{id}`) — change if your backend differs.

2) Add RTK Query reducer & middleware to store

📌 `src/app/store.js`

```
import { configureStore } from '@reduxjs/toolkit';
import { api } from './api';
import counterReducer from '../features/counter/counterSlice';

export const store = configureStore({
  reducer: {
    counter: counterReducer,
    [api.reducerPath]: api.reducer,
  },
  middleware: (getDefaultMiddleware) =>
getDefaultMiddleware().concat(api.middleware),
});
```

3) Use autogenerated hooks in components — no useEffect required

📌 `src/features/tasks/TasksRTK.jsx`

```
import React from 'react';
import { useGetTasksQuery, useAddTaskMutation, useDeleteTaskMutation,
useUpdateTaskMutation } from '../../app/api';

export default function TasksRTK() {
  const { data: tasks = [], error, isLoading } = useGetTasksQuery();
  const [addTask] = useAddTaskMutation();
  const [deleteTask] = useDeleteTaskMutation();
  const [updateTask] = useUpdateTaskMutation();

  if (isLoading) return <div>Loading...</div>;
  if (error) return <div>Error loading tasks</div>;

  return (
    <div>
      <button onClick={() => addTask({ title: 'New Task' })}>Add Task</
button>
      <ul>
        {tasks.map(t => (
          <li key={t.id}>
            <h3>{t.title}</h3>
            <button onClick={() => deleteTask(t.id)}>Delete</button>
            <button onClick={() => updateTask({ id: t.id, title: 'Updated
Title' })}>Update</button>
          </li>
        ))}
      </ul>
    </div>
  );
}
```

4) Optimistic Updates (example)

RTK Query supports optimistic updates via `onQueryStarted`.

Example mutation with optimistic update (inside `api` endpoints):

```
addTask: builder.mutation({
  query: (task) => ({ url: '/addtask', method: 'POST', body: task }),
  async onQueryStarted(arg, { dispatch, queryFulfilled }) {
    // Optimistically update LIST
    const patchResult = dispatch(
      api.util.updateQueryData('getTasks', undefined, (draft) => {
        draft.push({ id: Date.now(), ...arg });
      })
    );
    try {
      await queryFulfilled;
    } catch {
      patchResult.undo();
    }
  },
  invalidatesTags: [{ type: 'Tasks', id: 'LIST' }],
}),
```

This makes the UI feel immediately responsive and rolls back if the server fails.

Best Practices (2025)

- **Use RTK Query for server state** (fetching, caching, mutations). Use slices for client-only state (UI toggles, forms cache).
- **Keep API endpoints thin** — transform responses in `transformResponse` when necessary.
- **Use tags** to finely control cache invalidation (`LIST`, individual ids).
- **Use optimistic updates** for fast UX on create/update/delete.
- **Avoid storing large server data in slices** — prefer RTK Query cache.

Small Exercises (step-by-step)

Exercise 1 — Redux Toolkit slice (10–15 min)

- Create `counterSlice` as shown above. Add buttons to increment, decrement, and increment by amount. Use `useSelector` and `useDispatch`.

Exercise 2 — RTK Query basic (20–30 min)

- Create the `api` slice matching your FastAPI tasks endpoints.
- Replace your `fetch/useEffect` Tasks component with `TasksRTK` using `useGetTasksQuery`.
- Implement `useAddTaskMutation`, `useDeleteTaskMutation`, `useUpdateTaskMutation`.

Exercise 3 — Optimistic UI (optional, 20–30 min)

- Implement optimistic update for `deleteTask` or `addTask` using `onQueryStarted` and `api.util.updateQueryData`.

Extra Tips

- For complex selectors, use `createSelector` from `reselect` (bundled with RTK) to memoize.
- Use `transformResponse` to normalize server responses.
- When debugging, enable Redux DevTools — RTK config includes them by default in dev.

Deliverable for this Day

- `src/app/store.js` with RTK and RTK Query configured.
- `src/app/api.js` defining endpoints for tasks.
- `src/features/tasks/TasksRTK.jsx` that lists tasks and supports add/update/delete using RTK Query hooks.

Happy? When you want, I will create a **separate canvas** focusing only on **RTK Query deep-dive** (caching strategies, pagination, polling, streaming, error handling patterns).