

REACT .T .K

What is react? is it library or just Framework

Library vs. Framework: Key Differences

Feature	Library 🏛	Framework 🏭
Definition	A collection of reusable functions and tools	A complete structure that dictates how to build an application
Control	You control how and when to use it	The framework controls the flow, and you follow its rules
Flexibility	Highly flexible, use as needed	Less flexible, enforces a structure
Code Inversion	You call the library functions	The framework calls your code (Inversion of Control)
Examples	React, Lodash, jQuery, D3.js	Angular, Next.js, Django, Ruby on Rails

React is a **JavaScript library**, not a framework.

Why People Think It's a Framework

- React alone is a UI library, but when combined with tools like **React Router**, **Redux**, and **Next.js**, it behaves like a full-fledged framework.

The Key Concept: "Inversion of Control"

- In a **library**, you are in control. You decide when and where to use the library functions.
- In a **framework**, the framework is in control. It provides structure and calls your code at the right time.

Flux

Flux is an **architecture pattern** introduced by Facebook in **2014**, designed to manage how **data flows in a frontend application** — especially in **React apps**.

It was Facebook's answer to the confusion created by older patterns like:

- **MVC** (Model-View-Controller)
- **MVVM** (Model-View-ViewModel)

These patterns had a problem:

They often allowed **two-way data binding**, which made data flow confusing and unpredictable.

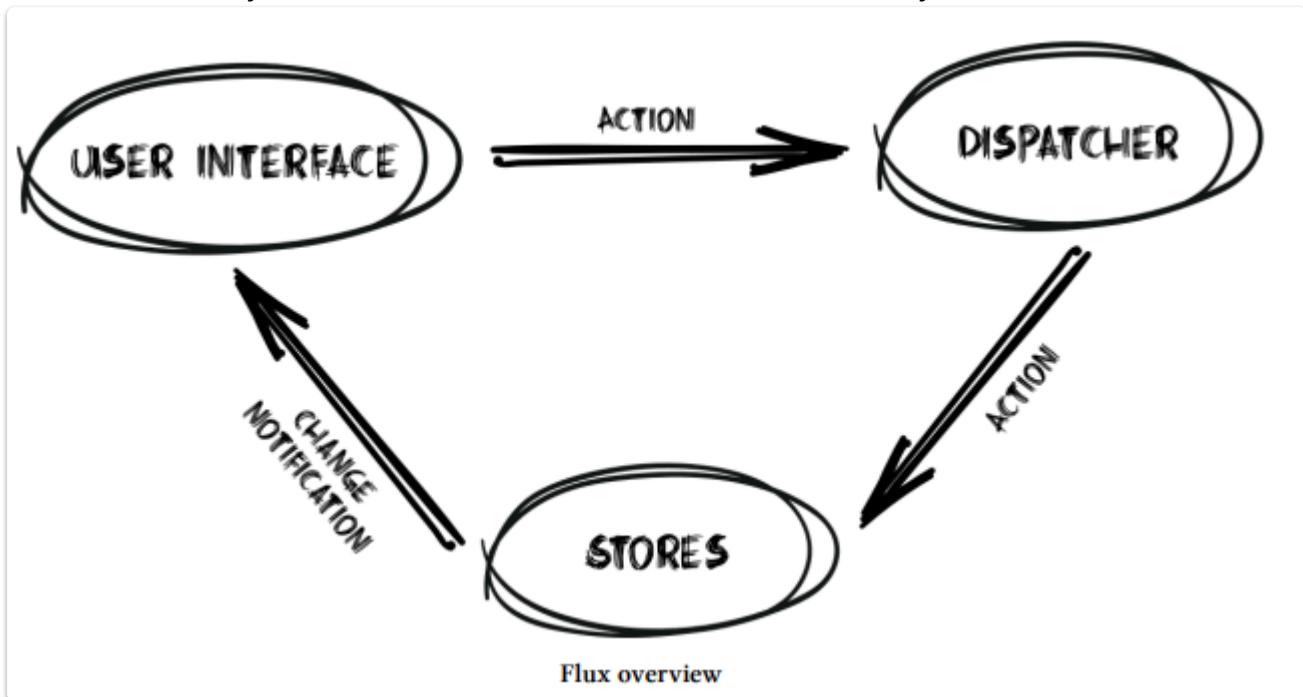
Flux Introduced a Better Idea: Unidirectional Data Flow

Instead of data going **back and forth** between view and model, Flux made it **flow in one direction only**.

Part	Description
Action	A plain object describing what happened (e.g., user clicked a button)
Dispatcher	A central hub that sends actions to stores
Store	Holds the application state and logic (like a database in memory)
View (React)	Listens to the store and renders the UI

```
User Action (click, API, etc.)  
↓  
Action Created  
↓  
Dispatcher sends it  
↓  
Store updates state  
↓  
View (React) re-renders  
↓  
User sees update, may act again ...
```

This is what they call the "**unidirectional data flow**". One-way. Clean. Predictable.



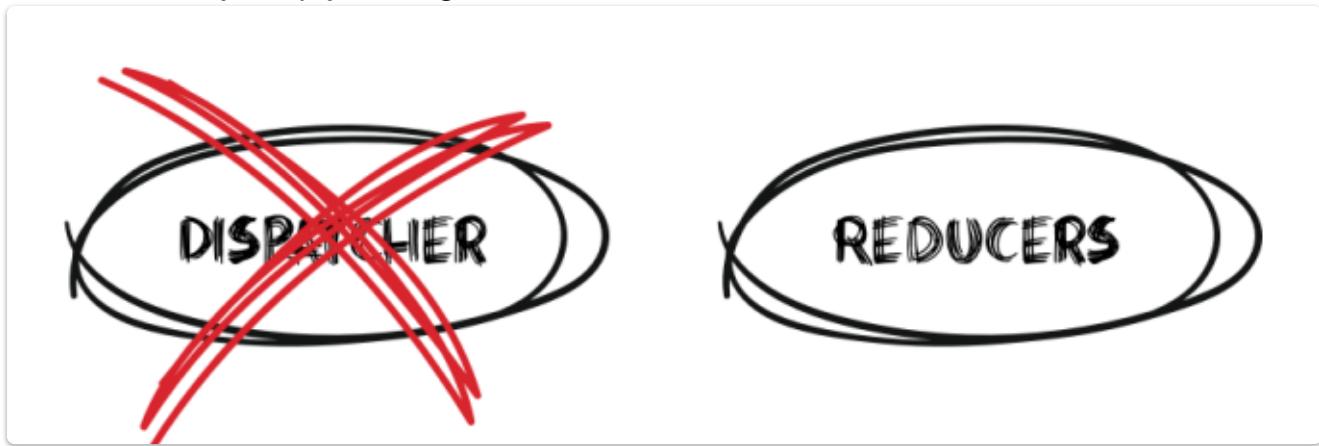
To solve this, **Flux** introduces a unidirectional data flow, which ensures a much more predictable way of handling updates. In Flux, when a user performs an action — like clicking a button — it triggers a single, well-defined **action** object. This action is sent to a **central dispatcher**, which then passes it to relevant **stores** that are responsible for updating specific parts of the application state. Once the store updates the state, the **view (UI)** reacts to those changes and re-renders accordingly. If any new actions are triggered during this process, Flux queues them and makes sure they are executed **after** the first action has finished. This approach keeps the entire data flow linear and easy to understand.

By enforcing this strict order of operations and keeping all updates within a clearly defined *loop* — **action** → **dispatcher** → **store** → **view** — Flux helps developers build applications where it's easy to reason about state changes. You always know what caused a change, where it happened, and what the outcome will be. This clarity is especially valuable in large-scale applications where managing complexity is key.

While **Redux** is inspired by **Flux**, it introduces a few key differences to simplify and streamline the architecture. One major distinction is that Redux uses a **single store** to hold the entire application's state, unlike Flux, which may have multiple stores for different parts of the app. In Redux, the store itself doesn't contain logic for how to change the state. Instead, it simply receives **actions** — plain JavaScript objects that describe what happened — and passes them to **reducers**, which are pure functions responsible for determining how the state should change. This eliminates the need for a separate **dispatcher** component, which Flux requires. In short, Redux removes the dispatcher and replaces it with **reducers**, making the data flow even more straightforward.

To make this idea easier to understand, imagine you're building a **recipe book app** using Redux. The **store** acts like the actual recipe book — it contains all the recipes and their ingredients in a structured format. When users interact with the app, such as adding a new recipe, editing an ingredient, or changing the quantity of an item, Redux dispatches a specific **action** for that interaction. These actions are then handled by different **reducers**, each responsible for a specific part of the state. For example, a `bookReducer` could handle adding or removing recipes, a `recipeReducer` might manage recipe names or steps, and a `ingredientsReducer` would handle ingredient details. This modular approach makes it easy to organize your logic and scale the app

in the future by simply adding new actions and reducers as needed.



In summary, Redux simplifies the Flux pattern by removing unnecessary parts like the dispatcher and combining state management into a single store. It introduces **reducers** to handle logic in a predictable, organized way, making it easier to manage and debug even in large applications.

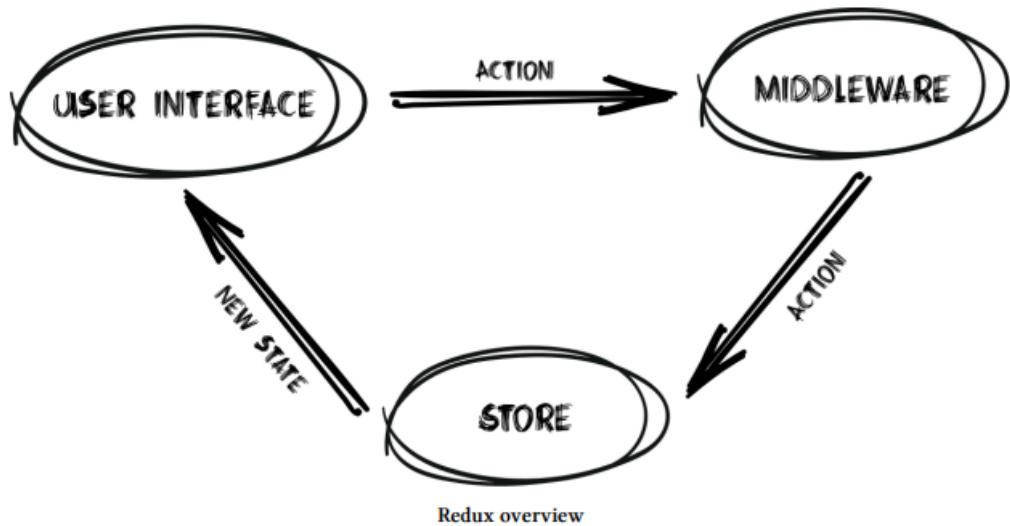
In a Redux-based application, each part of the logic has a clear and specific role. Let's go back to the **recipe book example**. Suppose the user decides to change the ingredient measurements from **grams to ounces**. In this case, the part of the system responsible for handling ingredient details — which we can call the **ingredients service** — will need to **recalculate the quantities**. At the same time, the **recipe service** may also need to update some metadata to mark the recipe as now using **imperial units** instead of metric. In Redux, these "services" are implemented as **reducers** — each reducer handles a specific section of the app's state and updates it according to the action it receives.

Apart from reducers, Redux also allows you to plug in something called **middleware**. Middleware sits between the action being dispatched and the moment it reaches the reducer. This allows you to run some logic **before** the action is handled. For example, you can use middleware to **log actions**, **check user permissions**, or even **send API requests**. Unlike reducers, middleware has more freedom — it can **stop an action**, **change it**, or even **dispatch other actions**. It's like a helpful assistant that can take care of background tasks, like checking rules or syncing with a server, without cluttering your main application logic.

One of the most important things to understand about Redux is that it uses **immutable state**. This means reducers never directly change the existing state. Instead, they **create a new copy** of the state (like a new version of the recipe book), apply the necessary changes, and return the new state. This behavior is essential for performance and clarity. It helps the UI know exactly what changed and when, which makes things like debugging, time-traveling through state history, or tracking down bugs much easier.

Lastly, Redux stores the entire application state in **a single place**, known as the **store**. This single source of truth makes it much easier to manage data, especially in larger

applications. With everything centralized, you always know where the current state of the app lives, which helps a lot during development and debugging.



Redux Terminology

Actions and Action Creators

In Redux, **the only way to change the state** (the data of your app) is by using something called an **action**.

An **action** is just a **plain JavaScript object** that describes **what happened** — for example: "increase a counter", "delete a user", "fetch data", etc.

```
{  
  type: 'INCREMENT',  
  payload: {  
    counterId: 'main',  
    amount: -10  
  }  
}
```

- `type` : A required string that describes **what kind of action** this is.
- `payload` : An optional object that carries the **data** needed to perform that action. But instead of writing this object every time, we usually create a **function** that builds this object for us — that function is called an **action creator**.

```
function incrementAction(counterId, amount) {  
  return {  
    type: 'INCREMENT',  
    payload: {  
      counterId,  
      amount,  
    },  
  },
```

```
};  
}
```

Reducers

Once an action is sent to the store, the store needs to figure out how to change the state accordingly. To do so, it calls a function, passing it the current state and the received action:

```
function calculateNextState(currentState, action)  
{ return nextState; }
```

This function is called a reducer. In real Redux applications, there will be one root reducer function that will call additional reducer functions to calculate the nested state.

In Redux, the **reducer** is a function that **decides how the state should change** when an action is dispatched.

You can think of it like this:

- **Action says:** "Hey! I want to add 1 to the counter."
- **Reducer says:** "Okay, let me update the state accordingly."

A reducer is a **pure function** — meaning:

- It **only uses the input** (state and action),
- It **returns a new state**,
- It does **not modify** the original state.



Tip to Remember

Reducers are pure. They clone, not mutate.

middleware

In Redux, **middleware** works like a **gatekeeper** for every action that's sent to the store. Before any action reaches the **reducers** (which actually update the state), it first **passes through the middleware**.

You can think of middleware as **interceptors** or **middlemen** that sit between the action being **dispatched** and the store **handling** it.

Middleware can:

- **Log** every action (for debugging),
- **Check user permissions** before allowing an action,
- **Trigger API calls**,
- **Stop or change an action** before it reaches the reducer,

- Or even **dispatch new actions**.
This makes middleware **super powerful and flexible** — it has access to:
 - The current action,
 - The `dispatch()` function (to send more actions),
 - And the `getState()` function (to read current state).

The **Redux store** is the central place where all the data (state) of your application is kept. It doesn't contain any logic or calculations itself — its job is to receive actions, send them through any middleware (like logging or async tasks), and then pass them to the reducers. The reducers decide how to update the state. After the state is updated, the store tells all parts of the app that are listening (like your UI) that the data has changed, so they can re-render or respond accordingly. There is only **one store** in Redux, which helps keep everything in one place and makes debugging and tracking state easier.

Tip to Remember

understanding **pure functions** and **Mutable** is **super important** for working with **Redux**

pure function is a function that:

1. **Always returns the same output for the same input**, and
2. **Does not change (or depend on) anything outside of itself** — no side effects.
what is not ***pure function

```
let total = 0;

function addToTotal(value) {
  total += value;
  return total;
}
```

What is Mutation?

Mutation means **changing data directly**— for example, modifying an object or array **without creating a copy**.

```
const user = { name: "Alex" };
user.name = "Sam"; // this changes the original object directly
```

please don't Mutate Directly

In stated Use this

```
const newUser = { ...user, name: "Sam" }; // create a copy and update it
```

Deep Pure and Impure Functions

A pure function returns values by using only its arguments: it uses no *additional data* and changes no data structures, **touches no storage**, and emits no external events (like network calls). This means that you can be completely sure that every time you call the function with the same arguments, you will always get the same result

If a function uses **any variables not passed in as arguments or creates side effects**, the function is impure. When a function depends on variables or functions outside of **its lexical scope**, you can never be sure that the function will behave the same every time it's called.

Mutating Objects

Immutability means that something can't be changed, guaranteeing developers that if you create an object, it will have the same properties and values forever.

```
const colors = {
  red: '#FF0000',
  green: '#00FF00',
  blue: '#0000FF'
};
// You can't assign new value to color but
colors = {};
console.log(colors);
colors.red = '#FFFFFF';
console.log(colors.red);
```

Even though the `colors` object is a constant, we can still change its content, as `const` will only check if the reference to the object is changed

Try writing this in the developer console. You will see that you can't reassign an empty object to `colors`, but you can change its internal value. To make the `colors` object appear immutable, we can use the **Object.freeze()** method.

```
Object.freeze(colors);
colors.red = '#000000';
console.log(colors.red);
```

Here, once we used **Object.freeze()**, the `colors` object became immutable. In practice things are often more complicated, though. JavaScript does not provide good native ways to make data structures fully immutable.

⚡ Object.freeze() won't freeze nested objects

```
const orders = {  
  bread: {  
    price: 10  
  },  
  milk: {  
    price: 20  
  }  
};  
  
Object.freeze(orders); // Only the 'orders' object is frozen  
  
orders.milk.price -= 15; // ⚠️ This still works!  
  
console.log(orders.milk.price); // Output: 5
```



`Object.freeze(obj)` is **shallow** — only top-level is frozen.

Use a **recursive deep freeze** to make nested objects truly immutable.

⚠️ **Redux principles, the application will keep all its state in our global store, including some parts of the UI**

The first step with any Redux-based app is to plan how data will be **arranged in the store**

RTK Query + Redux Setup

Why Use RTK Query?

- **Automatic Caching & Re-fetching** → No need to manually store API responses.
- **Efficient API Calls** → Avoid unnecessary requests with built-in caching.
- **Optimistic Updates** → Improves UI responsiveness by assuming API success.
- **Auto Data Synchronization** → Keeps UI up-to-date with minimal effort.
- **Simplifies API Handling** → Reduces boilerplate code compared to Redux Thunk/Saga.

RTK Query is a tool in Redux Toolkit that makes API fetching easier by handling caching, state management, and re-fetching automatically. Instead of manually writing reducers, actions, and thunks for API calls, RTK Query generates hooks like `useGetPostsQuery()` that manage everything—loading state, error handling, and caching—out of the box. It's inspired by React Query and Apollo but is built specifically for Redux' users.

Install Redux Toolkit and React-Redux

Add the Redux Toolkit and React-Redux packages to your project:

```
npm install @reduxjs/toolkit react-redux
```

Create a Redux Store

Create a file named `src/app/store.js`. Import the `configureStore` API from Redux Toolkit. We'll start by creating an empty Redux store, and exporting it:

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

export const store = configureStore({
  reducer: {
    [apiSlice.reducerPath]: apiSlice.reducer,
  },
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware().concat(apiSlice.middleware)
      .concat(rtkQueryErrorLogger),
})

```

What is a Redux Store?

Think of the **Redux store** like a **big JavaScript object** (a container) that holds the **entire state** of your application in one place.

"Redux Store is the single source of truth for all the data (state) in a React app."

Why Do We Create a Store?

We create a Redux store so that:

1. **All the state is managed in one place.**
2. **Components can access and update state easily.**
3. **State stays predictable** (using strict rules to update).
4. **Debugging is easier** (with tools like Redux DevTools).
5. **Middleware can be used** (like RTK Query, logging, etc.)

How Do We Manage the Store?

1. **Define "slices"** using `createSlice()` (each handles part of the state).
2. **Create the store** using `configureStore()`.
3. **Provide** the store to React using `<Provider>` from `react-redux`.
4. Use hooks like:
 - `useSelector()` to **read** data from the store.

- `useDispatch()` to **update** the store using actions.

What is `configureStore` ?

`configureStore` is a function from Redux Toolkit that helps you **create and set up a Redux store easily**.

Normally, setting up **Redux** is complex (you have to use `createStore`, `applyMiddleware`, etc.), but `configureStore` simplifies everything in one place.

- Automatically sets up **Redux DevTools**
- Adds default **middleware** (like `redux-thunk`)
- Allows you to combine multiple reducers
- Works smoothly with RTK Query

Basic Boilerplate code (**Create Slice**)

```
import { createSlice, nanoid } from "@reduxjs/toolkit";
const initialState = {
  todos: [{id: 1, text: 'Hello Redux ToolKit'}]
}

export const todoSlice = createSlice({
  name: "auth",
  initialState,
  reducers: {
    addTodo: (state, action) => {
      // here state gives you intial state what have until now.
      //for perform action you need something here action comes in play
      const todo = {
        id: nanoid(),
        text: action.payload?.text
      }
      state.todos.push(todo)
    },
    removeTodo: (state, action) => {},
    updateTodo: (state, action) => {}
  }
})
export const { addTodo, removeTodo, updateTodo } = todoSlice.actions;
export default todoSlice.reducer;
```

Create a Slice For Stores Used in our code

```
import { createSlice } from "@reduxjs/toolkit";
import { formatPermissions } from "../components/utils/format";
```

```

import { decryptData } from "../../components/utils/decryptData";
const user = localStorage.getItem("userInfo")
? decryptData(localStorage.getItem("userInfo")).userInfo: null;
const permissions = localStorage.getItem("userInfo")
? decryptData(localStorage.getItem("userInfo")).userPermissions
: null;
const initialState = {
userInfo: localStorage.getItem("userInfo")
? localStorage.getItem("userInfo")
: null,
user,
permissions,
isAuthenticated: localStorage.getItem("jwtToken") ? true : false,
};

export const authSlice = createSlice({
name: "auth",
initialState,
reducers: {
setCredentials: (state, action) => {
localStorage.setItem("userInfo", action.payload);
const permission = decryptData(action.payload);
state.permissions = formatePermissions(permission.userPermissions);
state.user = permission.userInfo;
state.userInfo = action.payload;
state.isAuthenticated = true;
},
logout: (state, action) => {
state.isAuthenticated = false;
state.userInfo = null;
state.user = null;
state.permissions = null;
localStorage.removeItem("jwtToken");
localStorage.removeItem("userInfo");
},
},
});

export const { setCredentials, logout } = authSlice.actions;

export default authSlice.reducer;

```

createSlice क्या करता है?

- Reducer बनाता है
- Action creators देता है
- Slice = state + reducer + actions

Store को क्या चाहिए?

- Store को केवल reducers चाहिए
- Reducers manually या slice से आ सकते हैं

Slice बनाओ या नहीं?

- Slice ज़रूरी नहीं, लेकिन बहुत convenient है
- Redux Toolkit recommend करता है createSlice का use

Shortcut:

createSlice() = reducer + actions + state (all-in-one)

Typical Reducer Function Looks Like:

```
const reducer = (state, action) => {
  // logic to return new state
}
const initialState = { count: 0 };
const counterReducer = (state = initialState, action) => {
  // state = { count: 0 } initially
}
```

What is state ?

- It's the **current state** of the store (or a slice of the store).
- It is **read-only** inside the reducer.
- You should **never mutate** it directly — always return a new copy or update it immutably (RTK uses *Immer*, so you **can** write it like you're mutating).

What is action ?

- It's an object describing what happened (the event).
- It must have a type field (like a label).
- It can also carry data using payload.

When do we need action?

If we want to use a dynamic value (like adding a custom number), we must accept the action argument and use action.payload.

```
addByAmount: (state, action) => {
  state.value += action.payload;
}
dispatch(addByAmount(10));
```

Provide the Redux Store to React

Once the store is created, we can make it available to our React components by putting a React-Redux `<Provider>` around our application in `src/index.js`. Import the Redux store we just created, put a `<Provider>` around your `<App>`, and pass the store as a prop:

```
import React from 'react'
import { createRoot } from 'react-dom/client'
import './index.css'
import App from './App'
import { store } from './app/store'
import { Provider } from 'react-redux'

const container = document.getElementById('root')

if (container) {
  const root = createRoot(container)

  root.render(
    <Provider store={store}>
      <App />
    </Provider>,
  )
} else {
  throw new Error(
    "Root element with ID 'root' was not found in the document. Ensure there is a corresponding HTML element with the ID 'root' in your HTML file."
  )
}
```

💡 What is `Provider` in React-Redux?

`Provider` is a **React component** that comes from the `react-redux` library. It's like a "**bridge**" between **Redux** and **React**.

|  *It gives your entire React app access to the Redux store.*****

Imagine Your App as a Big Building

-  **The Redux Store** = A **Power Generator Room** (holds electricity = state)
-  **State (Power)** = Flows through the building
-  **React Components** = Rooms in the building (like Kitchen, Office, etc.)
-  **Provider** = The **Main Electric Wiring Panel** that connects the power generator to all rooms

What is `reducer` ?

A **reducer** is a pure function in Redux that **controls how your state changes** when an action is dispatched.

Imagine a diary 📝 of your expenses:

- Every time you **spend money**, you write it down = dispatch an action.
- At any point, you can **calculate the total** = reducer processes all entries and gives final state.

What is `createApi` in RTK Query?

RTK Query is a powerful data fetching and caching tool built into Redux Toolkit.

`createApi` helps you define your API **endpoints**, how to fetch them, cache them, and even **invalidate** them when something changes.

```
export const store = configureStore({
  reducer: {
    [apiSlice.reducerPath]: apiSlice.reducer,
    [myReducer.name]: myReducer.reducer,
    auth: authReducer,
  },
});
```

What is `fetchBaseQuery` ?

`fetchBaseQuery` is a **built-in fetch wrapper** provided by RTK Query that helps you make **API calls easily**.

It's like a **helper function** that uses `fetch()` under the hood, but with extra features like:

- Adding headers (e.g. tokens)
- Setting a base URL
- Handling errors more gracefully

Why we use `fetchBaseQuery` :

- To avoid writing raw `fetch()` or `axios` calls everywhere
- To **set common settings** like `baseUrl` or `headers` **once** for all API calls
- So we can integrate it smoothly with RTK Query and its caching system

```
const baseQuery = fetchBaseQuery({
  baseUrl: "", // Common part of all your API URLs
  prepareHeaders: (headers) => {
    const token = localStorage.getItem("jwtToken");
    if (token) {
      headers.set("Authorization", `Bearer ${token}`);
    }
    return headers;
});
```

```
  },
});
```

For every API call, add this base URL and attach the **JWT** token from `localStorage` in the `Authorization` header.

What is `tagTypes` and why do we define it?

◆ Simple Definition:

`tagTypes` are **labels** (or "tags") you give to certain API data to help RTK Query know when to refetch or invalidate cache.

Simple Definition:

`tagTypes` are **labels** (or "tags") you give to certain API data to help RTK Query know when to refetch or invalidate cache.

Why use `tagTypes` ?

In real apps, after you **create**, **update**, or **delete** something via an API, you want to **refetch the latest data**.

RTK Query uses `tagTypes` + `providesTags` + `invalidatesTags` to know **what data should be refetched**.

Think of it as telling Redux: "This data belongs to the tag 'crypto' — if something changes with crypto, please refresh this."

```
tagTypes: ["broker", "admin", "commission", "assetCommission"]
```

We are working with these types of data in the app. Later, when we define API endpoints, we will use these tags to **control caching and refetching**.

2. now create all slicer where you want

```
import { fetchBaseQuery, createApi } from "@reduxjs/toolkit/query/react";
const baseQuery = fetchBaseQuery({
  baseUrl: "",
  prepareHeaders: (headers) => {
    // Get the JWT token and include it in the headers
    const token = localStorage.getItem("jwtToken");
    if (token) {
      headers.set("Authorization", `Bearer ${token}`);
    }
    return headers;
  },
});
```

```
});

export const apiSlice = createApi({
baseQuery,
tagTypes: [
"broker",
"admin",
"commission",
"assetCommission",
"crypto",
"cryptoPairs",
"ExchangeCredentials",
"ib",
"riskInfo",
"subib",
"support",
],
endpoints: (builder) => ({}) ,
```

`createApi` is a function from [Redux Toolkit Query \(RTK Query\)](#) that:

- Creates an `apiSlice` to manage **API requests** in Redux
- Automatically gives you:
 - Hooks like `useGetXQuery`, `usePostYMutation`
 - Caching
 - Auto-fetching, refetching, invalidation
 - Loading/error states

This is **Redux's** smart way of handling API in React apps.

What is `baseQuery` ?

```
const baseQuery = fetchBaseQuery({
  baseUrl: "",
  prepareHeaders: ...
})
```

This is the default function to handle API requests:

- You set your **base URL** here (e.g., `https://api.example.com`)
- You can also add things like:
 - **JWT tokens** in headers
 - Custom error handling
 - CORS, etc.

What is `endpoints: (builder) => ({})` ?

This is where you **define each API call** – GET, POST, PUT, DELETE etc.

```
endpoints: (builder) => ({  
  getUsers: builder.query({ ... }),  
  addUser: builder.mutation({ ... }),  
})
```

What is `tagTypes` ?

```
tagTypes: ["users", "products"]
```

Tag types help with **cache management**.

` GET

```
getUsers: builder.query({  
  providesTags: ["users"]  
})
```

Why do we use `providesTags: ["users"]` in a GET (query) endpoint in RTK Query?

```
getUsers: builder.query({  
  query: () => "/users",  
  providesTags: ["users"],  
})
```

You're telling RTK Query:

"Hey, this GET request is **linked to the tag `users` in the cache.**"

This helps RTK Query **know which data to refresh later**.

When you call a **mutation** like `addUser`:

```
addUser: builder.mutation({  
  query: (user) => ({  
    url: "/users",  
    method: "POST",  
    body: user,  
  }),  
  invalidatesTags: ["users"], // ⚡ This tells RTK to refetch getUsers  
})
```

`invalidate` ka matlab hota hai:

"Cache purana ho gaya hai, dobara fresh data fetch karo."

RTK Query will:

Run the `POST /users` call

See that it invalidated the `"users"` tag

Automatically refetch the `getUsers` query because it had `providesTags: ["users"]`

What is `useSelector`?

`useSelector` is a **React-Redux hook** used to **access the Redux store's state** inside a React component.

Why use `useSelector` in RTK (Redux Toolkit)?

Even when you're using **Redux Toolkit (RTK)** (which simplifies Redux setup), the core idea remains: **components need to read state from the store**.

- `useSelector` is how your component "subscribes" to the Redux store.
 - It allows your component to **re-render only when the selected state changes**.
 - RTK doesn't change how you read state; it just makes it easier to write actions, reducers, and setup.
- Use it **inside React components** when you want to **read data** from the Redux store.

```
import { useSelector } from 'react-redux';
const MyComponent = () => {
  const user = useSelector((state) => state.auth.user);
  return <div>Hello, {user.name}!</div>;
};
```

- `state.auth.user` is coming from the Redux store.
- The component **re-renders** when `state.auth.user` changes

RTK Query Overview

RTK Query is a powerful data fetching and caching tool. It is designed to simplify common cases for loading data in a web application, eliminating the need to hand-write data fetching & caching logic yourself.

RTK Query is an optional addon included in the Redux Toolkit package, and its functionality is built on top of the other APIs in Redux Toolkit.

Motivation

Web applications normally need to fetch data from a server in order to display it. They also usually need to make updates to that data, send those updates to the server, and keep the cached data on the client in sync with the data on the server.

- Tracking loading state in order to show UI spinners
 - Avoiding duplicate requests for the same data
 - Optimistic updates to make the UI feel faster
 - Managing cache lifetimes as the user interacts with the UI
- "data fetching and caching" is really a different set of concerns than "state management".**

APIs

```
import { createApi } from '@reduxjs/toolkit/query'
/* React-specific entry point that automatically generates
   hooks corresponding to the defined endpoints */
import { createApi } from '@reduxjs/toolkit/query/react'
```

let's quick setup for RTK and Query

1. same as above Index.js

```
import { Provider } from 'react-redux';
import { store } from './redux/store';
ReactDOM.createRoot(document.getElementById('root')).render(
<Provider store={store}>
<App />

</Provider>

);
```

Build store `store.js`

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

// You can import other feature reducers like authReducer, etc.
import authReducer from "../features/auth/authSlice"; // optional
// import any other reducers as needed

export const store = configureStore({
  reducer: {
    [apiSlice.reducerPath]: apiSlice.reducer, // RTK Query auto reducer
    auth: authReducer, // your custom reducers
    // add more reducers here
  },
  middleware: (getDefaultMiddleware) =>
```

```
    getDefaultMiddleware().concat(apiSlice.middleware),  
  
    devTools: process.env.NODE_ENV !== "production",  
});
```

RTK Query API Template

```
// src/features/api/apiSlice.js  
import { createApi, fetchBaseQuery } from "@reduxjs/toolkit/query/react";  
  
export const apiSlice = createApi({  
  reducerPath: "api",  
  baseQuery: fetchBaseQuery({  
    baseUrl: "https://your-api-url.com/api", // <- change this  
    prepareHeaders: (headers) => {  
      const token = localStorage.getItem("authToken");  
      if (token) {  
        headers.set("Authorization", `Bearer ${token}`);  
      }  
      return headers;  
    },  
  }),  
  tagTypes: ["Product", "users"],  
  endpoints: () => ({})  
});
```

In **RTK Query**, `tagTypes` are just **labels** you assign to pieces of data you fetch. They help RTK Query **know which data to refetch or invalidate automatically**.

4 authSlice.js

```
// src/store/slices/authSlice.js  
import { createSlice } from "@reduxjs/toolkit";  
  
const initialState = {  
  isAuthenticated: false,  
  user: null,  
  token: null,  
};  
  
const authSlice = createSlice({  
  name: "auth",  
  initialState,  
  reducers: {  
    setCredentials: (state, action) => {
```

```

    const { user, token } = action.payload;

    state.user = user;
    state.token = token;
    state.isAuthenticated = true;

    // Optional: Persist to localStorage
    localStorage.setItem("authToken", token);
    localStorage.setItem("authUser", JSON.stringify(user));
  },

  logout: (state) => {
    state.user = null;
    state.token = null;
    state.isAuthenticated = false;

    localStorage.removeItem("authToken");
    localStorage.removeItem("authUser");
  },
}

restoreSession: (state) => {
  const token = localStorage.getItem("authToken");
  const user = localStorage.getItem("authUser");

  if (token && user) {
    state.token = token;
    state.user = JSON.parse(user);
    state.isAuthenticated = true;
  }
},
},
});

export const { setCredentials, logout, restoreSession } =
authSlice.actions;
export default authSlice.reducer;

```

Product API (boilerplate)

```

// src/features/products/productApi.js
import { apiSlice } from "../api/apiSlice";
import {getproducts,updateproducts,} from "../components/constant/Api";
export const productApi = apiSlice.injectEndpoints({
  endpoints: (builder) => ({
    // ✅ GET products
    getProducts: builder.query({
      query: () => ({
        url: getproducts,

```

```

        method: "GET",
    }),
    transformResponse: (response, meta, arg) => {
        return response.status ? response?.data ?? [] : [];
    },
    providesTags: ["Product"], // <-- tells RTK when to refetch
),

// ✅ POST a new product
addProduct: builder.mutation({
    query: (newProduct) => ({
        url: "/products",
        method: "POST",
        body: newProduct,
    }),
    invalidatesTags: ["Product"], // <-- triggers getProducts again
}),
),
);
}

export const {
    useGetProductsQuery,
    useAddProductMutation,
} = productApi;

```

Component Usage

```

import { useGetProductsQuery, useAddProductMutation } from
"../features/products/productApi";

const ProductComponent = () => {
    const { data: products = [], isLoading, refetch } =
useGetProductsQuery();
    const [addProduct] = useAddProductMutation();

    const handleAddProduct = async () => {
        await addProduct({ name: "New Product", price: 99 });
        // refetch(); // optional, but not needed if `invalidatesTags` is used
    };

    if (isLoading) return <div>Loading...</div>

    return (
        <div>
            <button onClick={handleAddProduct}>Add Product</button>
            <ul>
                {products.map((item) => (
                    <li key={item.id}>{item.name} - ${item.price}</li>

```

```

        ))}
      </ul>
    </div>
  );
}

```

Folder Structure Suggestion

```

src/
└── app/
    └── store.js
└── features/
    ├── api/
    │   └── apiSlice.js
    ├── auth/
    │   └── authSlice.js
    └── products/
        └── productApi.js

```

Purpose of `skip` in RTK Query:

The `skip` option is a special flag provided by **RTK Query** (Redux Toolkit Query) to **conditionally prevent an API call** from happening.

```
useGetAdminsQuery({}, { skip: user?.role === "Admin" })
```

Means:

- "Run this query to fetch **admins data**, but..."
- **Only if** the current user is **not** an Admin.

Structure of `useAddBrokerMutation()`

```
const [addBroker, { data, error, isLoading, isSuccess, isError }] =
  useAddBrokerMutation();
```

Returned value	Description
<code>addBroker</code>	The function you call to trigger the mutation (e.g., sending the API request).
<code>data</code>	The response data returned from the API after a successful request.
<code>error</code>	The error object if the mutation fails .
<code>isLoading</code>	<code>true</code> while the mutation is in progress.
<code>isSuccess</code>	<code>true</code> if the mutation completed successfully .

Returned value	Description
isError	true if the mutation failed.

Structure of `useGetAdminsQuery()`

```
const { data, error, isLoading, isSuccess, isError, refetch } = useGetAdminsQuery();
```

Property	Description
data	The actual fetched data from the API.
error	Error info if the request fails.
isLoading	true while the request is in progress (first load).
isSuccess	true when the data is successfully fetched.
isError	true if the request fails.
refetch	A function to manually re-trigger the API call.

The empty `{}` in `useGetAdminsQuery({}, { skip: user?.role === "Admin" })` is the **query argument** — it's the first parameter of the hook and it's empty here because If your getAdmins endpoint is defined like this:

```
endpoints: (builder) => ({
  getAdmins: builder.query({
    query: () => "/admins", // No params needed
  }),
})
```

Then it **doesn't expect any arguments**, so you just pass `{}` or `undefined`.

So Why Pass `{}`?

Passing `{}` helps in two cases:

1. **Satisfy the expected argument** — if your hook is typed to expect an object.
2. To **match structure**, even if not needed now (e.g., you might add filters like `role`, `search`, or `limit` later).