⚙️ Redux Toolkit Setup in React – Step-by-Step 🧠

☑️ Step 1: **Install Redux Toolkit & React-Redux** 📦

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

- `@reduxjs/toolkit` ➤ modern Redux utilities
- `react-redux` ➤ connect Redux with React components

🏗️ Step 2: **Build the Store** (📁 `app/store.js`)

```
import { configureStore } from '@reduxjs/toolkit';
import cartReducer from '../features/cart/cartSlice';

const store = configureStore({
  reducer: {
    cart: cartReducer,  // 👈 You can add multiple slices here
  },
});

export default store;
```

- `configureStore()` combines all your slices and sets up the Redux devtools for free!

🧠 Step 3: **Connect Store to React App** (`index.js` or `main.jsx`) 🎬

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

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

- `Provider` makes the Redux store available to all components.

🍰 Step 4: **Create a Slice** (`features/cart/cartSlice.js`)

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

const initialState = {
  cartItems: [],
  totalQuantity: 0,
};

const cartSlice = createSlice({
  name: 'cart',
  initialState,
  reducers: {
    addItem(state, action) {
      state.cartItems.push(action.payload);
      state.totalQuantity += 1;
    },
    removeItem(state, action) {
      state.cartItems = state.cartItems.filter(
        item => item.id !== action.payload.id
      );
      state.totalQuantity -= 1;
    },
  },
});

// Export actions
export const { addItem, removeItem } = cartSlice.actions;

// Export reducer to be used in store
export default cartSlice.reducer;
```

- `createSlice()` automatically generates actions and reducers! 🪄
- Actions: `addItem`, `removeItem` etc.

🚀 Step 5: **Dispatch an Action** 📨

You dispatch actions using the `useDispatch()` hook.

```
import { useDispatch } from 'react-redux';
import { addItem } from '../features/cart/cartSlice';

function Product({ product }) {
  const dispatch = useDispatch();

  return (
    <button onClick={() => dispatch(addItem(product))}>
      Add to Cart 🛒
    </button>
  );
}
```

🔍 Step 6: **Use Selector to Access Store Data** 📦

You read from the store using the `useSelector()` hook.

```
import { useSelector } from 'react-redux';

function CartSummary() {
  const cartItems = useSelector(state => state.cart.cartItems);
  const totalQuantity = useSelector(state => state.cart.totalQuantity);

  return (
    <div>
      <h3>Total Items: {totalQuantity}</h3>
      <ul>
        {cartItems.map(item => (
          <li key={item.id}>{item.name}</li>
        ))}
      </ul>
    </div>
  );
}
```

🔄 Recap in One Line:

☑ Install → 🏗 Create Store → 🔌 Connect to App → 🛠 Create Slice → 📤 Dispatch → 📥 Select Data

---

⚙ Actions in Redux

🤔 **What are Actions?** Actions are **the only way** your application can interact with the Redux **store**. They **send data** from your app to the store.

☑ Key Characteristics:

- They are **plain JavaScript objects**.
- Must have a `type` property (a string that describes the action).
- Can carry **additional data** (called `payload` or `info`).

📦 Action Example

```
// Action Type (usually defined as a constant)
const BUY_CAKE = 'BUY_CAKE';

// Action Creator Function
function buyCake() {
  return {
```

```
    type: BUY_CAKE,
    info: 'First redux action'
  };
}
```

📌 **Note:** The `type` is like a message label, and `info` or `payload` can contain extra details.

🔧 Reducers in Redux

🧠 **What is a Reducer?** A reducer tells **how the Redux store should change** when an action is dispatched.

☑️ Key Characteristics:

- It's a **pure function**.
- Takes two arguments: `state` and `action`.
- Returns the **new state** of the application.

```
// Initial state
const initialState = {
  numOfCakes: 10
};

// Reducer function
const reducer = (state = initialState, action) => {
  switch (action.type) {
    case BUY_CAKE:
      return {
        ...state,
        numOfCakes: state.numOfCakes - 1
      };
    default:
      return state;
  }
};
```

🧾 **Reducer Logic**

```
(previousState, action) => newState
```

📌 **Note:** Reducers **do not mutate** the state; they return a **new object** representing the updated state.

⚛️ Redux Made Super Simple: Store, Actions, Reducers & Dispatch

---

> Feeling lost in the world of Redux? 😵 Let's break it down with relatable analogies, real-world examples, and emojis so you'll never forget again. 🤓

🏪 1. Store – The Global State Manager

🧠 **Think of the Store as your app's brain.** It holds everything your app knows or needs to remember — like a big global object 📦 that stores all the data.

🔍 In simple terms:

- It's a **central place** where the entire app's **state lives**.
- Your UI (components) gets its data from the **Store**.
- When the **store changes**, components **automatically re-render** to reflect the new data.

> 🛍️ Imagine a giant warehouse (Store) with all your app's data inside neatly packed.

```
const store = createStore(reducer);
```

📝 2. Actions – The "What Happened"

📣 **Actions are messages** 📩 that you send to the Store, saying *"Hey! Something just happened!"*

🧠 Structure:

An action is a plain JS object:

```
{
  type: "ADD_TODO", // what happened
  payload: "Buy Milk" // optional data
}
```

- `type`: describes **what kind of change** to perform.
- `payload`: holds **extra information** (if needed).

> 🎬 Actions are like **event reports**. "Someone clicked a button", "user typed something", etc.

☑️ Example:

```
function addTodo(task) {
  return {
    type: 'ADD_TODO',
    task
  };
}
```

🛠️ 3. Reducers – The "How To Change State"

A **Reducer** is a **pure function** that answers:

> "How should the store change in response to a certain action?"

It takes the **current state** and an **action**, then returns a **new state**. No side effects. No mutations. Just a clean transformation.

🧠 Structure:

```
function todoReducer(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      return [...state, action.task];
    case 'REMOVE_TODO':
      return state.filter(task => task !== action.task);
    default:
      return state;
  }
}
```

> 🧹 Think of it like a **smart janitor**. It knows exactly how to clean or arrange stuff in your store based on what just happened (action).

🚀 4. Dispatch – Triggering the Change

Dispatch is the **method you call to send an action to the reducer**.

```
dispatch({ type: "ADD_TODO", payload: "Learn Redux" });
```

- It **fires** the action.
- Redux catches it and **passes it to the reducer**.
- Reducer **updates the store**.
- Components **re-render** if they depend on the changed data.

> 🎯 Dispatch is like telling the warehouse manager: *"Hey, apply this change using this instruction!"*

🔄 The Flow of Redux — Summed Up in Emojis 🎉

1. 👨‍💻 User interacts with app (click, type, etc.)
2. 📣 You dispatch an **Action**
3. 🧠 **Reducer** receives the action + current state
4. 🔄 Reducer returns new state
5. 🏬 **Store** updates its state

6. 🔁 React components subscribed to this store re-render automatically.

🧩 Real-World Analogy

🎓 Think of a school 🏫:

- 🏬 **Store** is the database of all student records.
- 📝 **Actions** are forms teachers fill like “Add Marks” or “Update Attendance”.
- 🛠️ **Reducers** are the staff who process the forms and update the records.
- 🚀 **Dispatch** is the act of submitting the form to the staff.

✨ BONUS: Action Creators & Pure Reducers

🛠️ Action Creator:

A function that returns an action.

```
const addItem = (item) => ({ type: "ADD_ITEM", payload: item });
```

🧼 Reducers are “Pure”:

- No API calls
- No modifying original state
- Same input always gives same output

🔗 Helpful Resource

For deeper reading, check this out: 🧾 GeeksForGeeks - Redux Intro

---

📌 Final Thoughts

Redux may seem intimidating at first, but once you get the hang of **Store** ➡️ **Action** ➡️ **Reducer** ➡️ **Dispatch**, it becomes second nature!

> 💡 “You describe **what** happened using actions, and let reducers decide **how** the state should change.”

Keep practicing and you’ll master Redux like a pro. 🥷🔥

✍️ Author: Darshan Vasani

🔗 Portfolio | 💼 LinkedIn | 📦 GitHub | 🎙️ Topmate

🆚 Action vs Reducer – 💥 The Core Difference

🔑 Feature	📨 Action	🧠 Reducer
What it is	A plain **JavaScript object** describing **what happened**	A **pure function** that decides **how state should change** based on the action
Role	🚀 Sends info TO the store	🛠️ Handles HOW the store should update
Has a type?	☑️ Always (e.g., `type: "BUY_CAKE"`)	❌ Never (uses `action.type`)
Returns	Just an object	A **new state**
Used by	Dispatched using `dispatch(action)`	Called automatically by Redux store
Example type	`"ADD_TO_CART"`, `"LOGIN"`, etc.	Handles each action type using `switch-case` or `if` logic
Side Effects?	None, but may trigger one indirectly (via middleware)	Never, must be pure!

🎭 Real-Life Analogy: 🍰 Cake Shop Example

- 📝 **Action** = Customer's Order Ticket ➤ "I want to **buy 1 cake**."
- 👨‍🍳 **Reducer** = Shopkeeper ➤ Sees the ticket, updates the number of cakes in the shop accordingly.

🖌️ Code Comparison Example

📨 Action

```
const BUY_CAKE = 'BUY_CAKE';

// Action creator
function buyCake() {
  return {
    type: BUY_CAKE,
    info: 'First redux action'
  };
}
```

🧠 Reducer

```
const initialState = {
  numOfCakes: 10,
};

function cakeReducer(state = initialState, action) {
```

```
  switch (action.type) {
    case BUY_CAKE:
      return {
        ...state,
        numOfCakes: state.numOfCakes - 1
      };
    default:
      return state;
  }
}
```

📌 TL;DR (One-liner Summary):

> 🔁 **Action** = "What happened?" (just data) 🧠 **Reducer** = "What to do with it?" (decides the new state)

Redux analogies and explanations

🔁 1. **Redux in Simple Terms**

> 📦 **Store**: Holds all the data your application uses — the **single source of truth** 📨 **Action**: Tells the store what to do — carries **instructions + data** 🧠 **Reducer**: Actually **manipulates** the store data based on the action 💬 **Switch-case** is commonly used in reducers to choose logic based on action type 🔁 When state changes via the reducer, the UI **re-renders**

🗣️ *Think of it like a request form system:*

- User fills a form (Action) 📝
- Admin reviews it and updates the record (Reducer) 👨‍💼
- All records are stored in a central file cabinet (Store) 🗂️

🎬 2. **Redux Analogy – Event Based**

> 🏪 **Store**: A global state for the entire app 🎯 **Action**: What you **want to do** (e.g., a button click event) 🧠 **Reducer**: How your action transitions **one state → next state** 📮 **Dispatch**: The way to **send actions** to the reducer

🗣️ *Think of it like posting a request:*

- You write an email request (Action)
- You hit "send" (Dispatch)
- Admin reads and updates the database accordingly (Reducer)
- The new database (Store) gets updated 🗄️

📊 3. **Redux Terms as Definitions**

> 🗂️ **Store**: Object holding app's state 🧠 **Reducer**: Function that **returns state** (after being triggered by action) 📨 **Action**: Object with a `type` that **instructs** the reducer how to change state

🗣️ *Like an HR system:*

- Action: "Hire Developer"
- Reducer: Adds person to employee list
- Store: Keeps all employee records

🧱 4. **Building Blocks Explanation**

> 🎬 **Action**: The **only source** of info for the store. 🪄 **Action Creator**: A function that returns an action 🧠 **Reducer**: A pure function — it **knows what to do** with actions 📦 **Store**: Holds your app state

🗣️ *Analogy:*

> Action: "Cook Pasta" 🍲 Reducer: Looks up how to cook pasta, and updates the kitchen state Store: Kitchen inventory that gets updated Action Creator: A recipe instruction creator 📝

🏛️ 5. **Redux vs Props**

> 🔗 **Props**: Used to share data between components ❌ Props become complex to manage in large apps ☑️ **Redux Store**: Central place where components **pull data from** 🔄 Components **auto re-render** when store updates 📨 **Action**: Carrier of data 🧠 **Reducer**: Organizer that updates the store based on incoming actions

🗣️ *Analogy:*

- Store: A shared Google Sheet 📄
- Action: "Add new row" command
- Reducer: Script that adds/modifies the row
- Dispatch: Click to run the script
- Component: Reacts to the new data in the sheet 🎬

📚 6. **Redux Docs Style Summary**

> 📦 **Store**: The **global state container** 📤 **Dispatcher**: Sends actions to the reducer 📨 **Action**: A **plain JS object** with a `type` + optional `payload` 🧠 **Reducer**: A function that decides **how to change state** based on the action

🗣️ *Like a restaurant:*

- Action: "Order Pizza"
- Dispatch: Tells the kitchen
- Reducer: Cooks pizza and updates order list
- Store: Master list of all customer orders 📋

🏁 Final TL;DR Summary

Concept	Analogy	Role
Store	State vault 🏦	Central place where the state lives
Action	Instruction 📨	Tells **what** should happen (with optional data)
Reducer	Decision-maker 🧠	Tells **how** the state should change
Dispatch	Messenger 🚚	Sends action to reducer

🏪 `appStore`: Your Central State Warehouse

🔍 What is `appStore`?

Think of `appStore` as a **giant central warehouse** 🏢 that stores all your app's state. Every section of your app (cart, user, product, etc.) sends updates to and retrieves data from this warehouse.

In Redux, this is the **store** created using `configureStore()` from Redux Toolkit.

☑️ Example:

```
// appStore.js
import { configureStore } from '@reduxjs/toolkit';
import cartSlice from './cartSlice';

const appStore = configureStore({
  reducer: {
    cart: cartSlice,
    // You can add more slices here (like userSlice, productSlice)
  }
});

export default appStore;
```

🧠 Behind the Scenes:

- `configureStore` combines all your slices (state + reducers) into one big `store`.
- The keys in the `reducer` object (like `cart`) determine how you access that slice: `state.cart`.

🛒 `cartSlice`: The Shopping Cart Brain

🦉 What is a Slice?

A **slice** is a modular chunk of Redux state — it includes:

- ☑️ State (`initialState`)
- ⚙️ Reducers (functions that update that state)

- 🚀 Actions (auto-generated from reducers)

☑ Example:

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

const cartSlice = createSlice({
  name: 'cart',
  initialState: {
    items: [],      // array of items added to the cart
    totalAmount: 0  // total price
  },
  reducers: {
    addItem: (state, action) => {
      state.items.push(action.payload);
      state.totalAmount += action.payload.price;
    },
    removeItem: (state, action) => {
      const index = state.items.findIndex(item => item.id === action.payload.id);
      if (index !== -1) {
        state.totalAmount -= state.items[index].price;
        state.items.splice(index, 1);
      }
    },
    clearCart: (state) => {
      state.items = [];
      state.totalAmount = 0;
    }
  }
});

export const { addItem, removeItem, clearCart } = cartSlice.actions;
export default cartSlice.reducer;
```

🔍 Behind-the-Scenes: How Redux Toolkit Works Here

1 `createSlice()`:

- Auto-generates **action creators** like `addItem`, `removeItem`, `clearCart`.
- Auto-creates a **reducer** function under the hood.

2 `addItem()` Logic:

```
addItem: (state, action) => {
  state.items.push(action.payload);
  state.totalAmount += action.payload.price;
}
```

- `action.payload` is the new item added (comes from `dispatch(addItem(item))`).
- Adds it to `items[]` and updates the `totalAmount`.

3 `removeItem()` Logic:

- Finds the item in `items[]` by `id`, removes it, and subtracts its price.

4 `clearCart()` Logic:

- Resets everything — used during checkout or logout.

🔄 Example Flow: What Happens on `dispatch(addItem(item))`

☑ Say your product is:

```
const product = { id: 101, name: "iPhone", price: 799 };
```

🧨 You do:

```
dispatch(addItem(product));
```

⚙ Internally:

- The action `{ type: "cart/addItem", payload: { ...product } }` is dispatched.
- Redux Store finds `cartReducer` based on `type`.
- Runs the logic inside `addItem()` with access to current state.
- Updates the `items` array and `totalAmount`.

🗂 Structure Summary:

```
src/
├── appStore.js       <-- Your central Redux store
└── cartSlice.js      <-- Shopping cart state logic
```

🧠 Analogy Time!

Concept	Analogy
`appStore`	🏢 Central warehouse
`createSlice`	🧠 A department in the warehouse

Concept	Analogy
`initialState`	📦 Initial inventory in that department
`reducers`	🛠️ Set of machines to modify inventory
`actions`	📩 Orders sent to modify inventory
`dispatch(action)`	🚚 Sending a request to the warehouse

☑️ Final Thoughts

- `appStore` = Combines all slices
- `cartSlice` = Manages cart state
- Actions like `addItem`, `removeItem`, `clearCart` allow interaction
- Redux Toolkit simplifies a lot of the boilerplate with `createSlice`

☑️ 1. `onClick={handleAddItem}`

👉 Meaning:

You're **passing a reference** to the function `handleAddItem`. This means React will call the function **only when the user clicks** the element.

☑️ Use this when:

- The function doesn't require any arguments.
- You don't want it to execute on render.

🧠 Example:

```
<button onClick={handleAddItem}>Add</button>
```

☑️ Only runs `handleAddItem` when the button is clicked.

☑️ 2. `onClick={() => handleAddItem(item)}`

👉 Meaning:

You're using an **anonymous arrow function** that **calls `handleAddItem` with an argument** when the click happens.

☑️ Use this when:

- You need to **pass arguments** to the function.
- You want to delay the call until the button is clicked.

🧠 Example:

```
<button onClick={() => handleAddItem(item)}>Add</button>
```

☑ On click, it calls `handleAddItem(item)` with the correct value.

❌ 3. `onClick={handleAddItem(item)}`

⚠ Meaning:

You're **immediately calling** `handleAddItem(item)` during render — not on click!

- The **return value** of that function (which could be `undefined`, a function, or something else) is what gets assigned to `onClick`.

❌ Why it's usually wrong:

This causes the function to run **when the component renders**, not when it's clicked.

🧠 Example:

```
<button onClick={handleAddItem(item)}>Add</button> // ❌ Bad!
```

🚨 This will call `handleAddItem(item)` **as soon as the component renders**, which is not the intended behavior for an `onClick`.

🔁 Summary Table:

Syntax	Calls When Clicked?	Accepts Arguments?	Common Use?
`onClick={handleAddItem}`	☑ Yes	❌ No	☑ Yes
`onClick={() => handleAddItem(item)}`	☑ Yes	☑ Yes	☑ Yes
`onClick={handleAddItem(item)}`	❌ No (runs on render)	☑ Yes	❌ No

🧠 Bonus Tip:

If you want to optimize performance when passing arguments, you can **memoize** the handler using `useCallback` or extract the inline arrow function to a named function inside your component.

State Mutation

```
clearCart: (state) => {
  state.items = [];
  state.totalItems = 0;
  state.totalPrice = 0;
}
```

Let's **break it down** and explain what would happen if you replaced the entire state like `state = []` or `state = ["Darshan"]`.

☑ `state.items = []`

This sets the `items` **property** inside the state object to an **empty array** — i.e., it clears just the `items` list, keeping the rest of the state intact.

🧠 Example:

```
state = {
  items: ["Apple", "Banana"],
  totalItems: 2,
  totalPrice: 100
}
```

After:

```
clearCart: (state) => {
  state.items = [];
  state.totalItems = 0;
  state.totalPrice = 0;
}
```

Now `state` becomes:

```
{
  items: [],
  totalItems: 0,
  totalPrice: 0
}
```

☑ **This is the correct way** when using a state object with multiple keys.

🚫 What If You Write: `state = []` ❌

```
clearCart: (state) => {
  state = []; // 🚫 This does NOT work as expected in Redux Toolkit
}
```

👉 This **replaces the whole state** with a new value — an **array**, instead of the expected **object**.

⚠️ Why it breaks:

Redux Toolkit's `createSlice` manages your state **immutably behind the scenes** using a library called **Immer**.

You **should mutate the state's fields directly**, not reassign `state = []`.

👄 You'd lose:

```
// Original structure:
{
  items: [...],
  totalItems: number,
  totalPrice: number
}
```

➡️ Becomes:

```
[]
```

That's a type mismatch ⚠️ — you're replacing an object with an array.

🔄 What about `state = ["Darshan"]`?

Same issue! You're **replacing the whole state object** with an array.

🟫 Wrong:

```
clearCart: (state) => {
  state = ["Darshan"]; // replaces state entirely — not allowed like this
}
```

Unless your **whole slice's initialState** is an array (e.g. `initialState = []`), this is invalid.

☑️ So When Can I Use `state = []` or `["Darshan"]`?

Only when your **slice's initial state itself is an array**:

```
// ☑ Example
const nameSlice = createSlice({
  name: 'names',
  initialState: ["Darshan"], // initial state is an array
  reducers: {
    clearNames: () => {
      return []; // ☑ you can replace state here!
    }
  }
});
```

✔ Because the slice is **initialized as an array**, replacing it with another array is valid.

🧠 TL;DR

Syntax	Meaning / Use Case	Valid?
`state.items = []`	Clear the `items` array inside an object slice	☑ Yes
`state = []`	Replace entire state with an array	❌ No (unless initialState is an array)
`state = ["Darshan"]`	Replace entire state with an array with 1 item	❌ No (unless initialState is an array)
`return []`	Return a new array if slice state *is* an array	☑ Yes
`return { ... }`	Return a new object state (alternative to mutation)	☑ Yes