C# Corner

REDUX SIMPLIFIED

Learn **Redux** by Building a Real Car Parts E-commerce Application

RTK REDUX TOOLKIT | REACT | TS TYPESCRIPT | REAL-WORLD APPLICATION

RIKAM PALKAR

C# Corner

Redux Simplified

Rikam Palkar

Author – Rikam Palkar
Publisher – C# Corner
Editorial Team – Deepak Tewatia, Baibhav Kumar
Publishing Team – Praveen Kumar
Promotional & Media – Rohit Tomar

About the Book

Redux Simplified is a practical, hands-on guide that teaches Redux by building a real application from start to finish. This concise four-chapter guide is designed to give you enough practical knowledge to understand and start using Redux confidently.

Instead of focusing on theory and boilerplate, it walks through slices, stores, hooks, actions, selectors, and state flow in a simple and structured way, helping you understand how Redux works in real React applications.

It teaches Redux by building a real car parts e-commerce application using React, TypeScript, and Redux Toolkit. You will write code while implementing features such as product listings, cart management, quantity updates, and synchronized state across multiple components.

The application includes:

- Product catalog
- Cart panel
- Item quantity management
- Dynamic cart totals

Throughout the journey, you will build the application step by step while learning how slices, stores, actions, reducers, hooks, selectors, and Redux data flow work together in a real-world application.

The complete source code for the application built in this book is available in the repository.

About the Author

Rikam believes that the best way to truly understand technology is by building real applications with it. Over the past seven years, he has designed and delivered enterprise systems serving millions of users, gaining practical experience in architecture, scalability, and software engineering. As a Senior Software Engineer and Software Architect, he focuses on solving real-world engineering problems through hands-on development rather than theory alone.

His work and contributions have earned him recognition as a Microsoft Most Valuable Professional (MVP), and he is also certified as an Amazon Solutions Architect. Across cloud computing, AI/ML, AWS, Azure, and software architecture, he holds 24 professional certifications that reflect his deep commitment to understanding systems before applying them in production environments.

Rikam completed his Master of Computer Applications (MCA) from Veermata Jijabai Technological Institute and has authored multiple practical technology books, including Redux Simplified, Blazor Simplified, and WPF Simplified. His research work has been published in peer-reviewed technical journals, and he has written more than 230 technical articles for platforms such as C# Corner, Medium, and the AWS Community Blog. He has also contributed open-source learning resources covering data structures, architecture patterns, and full-stack development.

Redux Simplified is built on the same philosophy that defines his engineering journey: learning through practical implementation. Rather than being a theoretical reference manual, the book focuses on building real applications and understanding how Redux works in production-ready React projects. Through simple explanations and hands-on examples, Rikam aims to make complex concepts approachable and easy to understand for developers at every level.

Portfolio Website · GitHub Profile · LinkedIn Profile

Mahesh Chand from C# Corner

Your support and encouragement have been instrumental in making this book a reality. The author of this book is sincerely thankful to each one of you.

— Rikam Palkar

Table of Contents:

Fundamentals 6

Slice and Store 11

Custom Hooks and UI Shell 16

Writing and Reading State 21

1

Fundamentals

Overview

In this chapter, we will explore the fundamentals of Redux and understand how centralized state management works in React applications. You will learn about actions, reducers, selectors, Redux data flow, and Redux Toolkit while building the foundation of a real car parts e-commerce application.

You Want to Learn Redux

Good. And the best way to learn Redux is to build something real with it something with moving parts. Products, a cart, quantities. So that is exactly what we are going to do. We are building a store for car parts.

This four-part guide takes you through state management from the ground up. By the end, you will not just have a functional shopping cart you will truly understand the mechanics behind it.

Here is what the finished app looks like:

There is a header that shows a cart icon with an item count. Below it sits a product grid where each product has an Add to Cart button. Off to the side there is a cart panel that shows items, quantities, and a running total. Three completely separate components and yet they are perfectly in sync.

Who Owns the Data?

This is the question that eventually breaks every React beginner's approach. If the header holds the cart count, how does the product grid update it? If the product grid holds it, how does the cart panel read it? You start by centralising the logic, passing props down, and sending callbacks back up. It works for a while. Then you add one more component and the whole house of cards collapse.

This is where Redux comes in.

What Redux Is

Redux is a state management library, but that definition is a bit dry. Think of it as the Centralized Store for your entire application. In a typical app, components are constantly gossiping: the Header asks the Cart for a count, or the Cart asks the Product Grid what was added. With Redux, that whisper game ends. Every component that needs data reads from the Store, and every component that changes data writes to the Store. Everything flows through one place.

To keep this flow predictable, Redux relies on three main players:

- Actions are plain objects that describe events saving, removing, and updating.
- Reducers are pure functions that take the current state and an action, then calculate the new state.
- Selectors are functions that extract and format specific pieces of state for your components.

The Redux Data Flow

Picture a loop. It starts with the UI. The user does something, clicks a button, types a value. The UI dispatches an Action, which is a simple object describing what happened (for example, TYPE: 'ADD_ITEM'). The Store receives that action and passes it to the Reducer. The Reducer looks at the current application State and the incoming Action and calculates the new State. The Store updates itself with what the Reducer returned. Because the State changed, the UI, which is subscribed to the Store, gets notified, receives fresh data, and re-renders.

One click. One predictable chain. No component talks to another directly.

What We Are Building

A car parts e-commerce store. The folder structure we will end up with:

```
src/
├── App.tsx
├── dummy-products.ts
├── components/
│   ├── Header.tsx
│   ├── Product.tsx
│   ├── Cart.tsx
│   ├── CartItems.tsx
│   └── Shop.tsx
└── store/
    ├── store.ts
    ├── cart-slice.ts
    └── hooks.ts
```

Notice the split. The components/ folder is pure UI it displays things and fires actions. It does not care how state is managed. The store/ folder is pure logic it manages state. No JSX. No styling. Just state. This separation is what makes the app maintainable. Components are displays. The store is the brain.

We Are Using Redux Toolkit

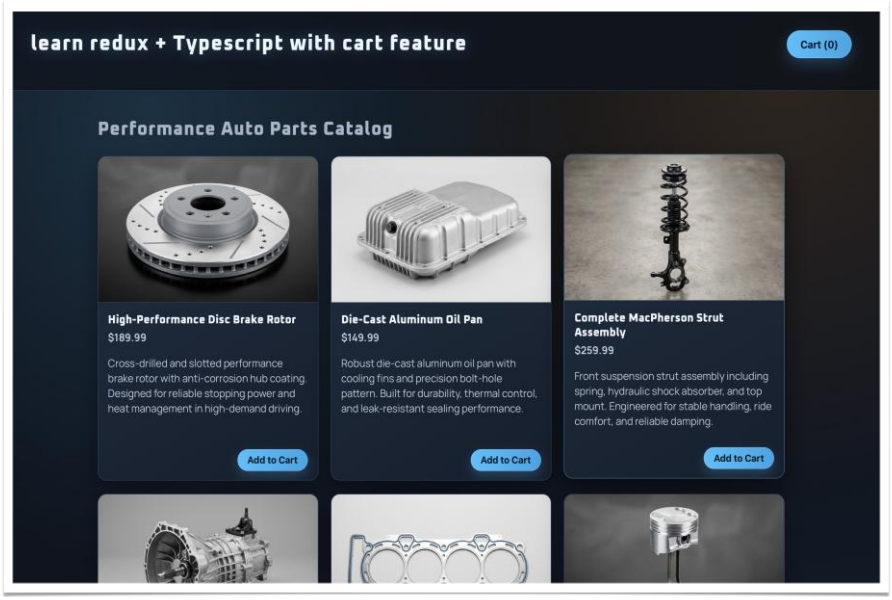
Classic Redux had a reputation. Verbose. Too much boilerplate. People spent more time setting up than building. Redux Toolkit RTK fixed that. It ships with createSlice, which lets you write your actions and reducers together in one place. It handles immutability for you. It sets up the store with sensible defaults. Less ceremony, same power. That is why we are using RTK throughout this guide.

Why TypeScript Makes Redux Much Better

Redux plus TypeScript gives practical safety exactly where beginners usually struggle: state shape safety, payload safety, and dispatch and selector safety. You get clearer autocomplete, safer refactors, and far fewer runtime surprises. It is not just nice to have it, but it genuinely changes how confidently you can work.

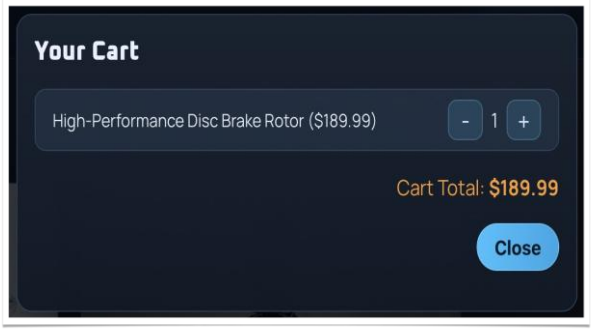
The Demo

You have a header. It shows a cart icon with a count. You have a product grid. Each product has an "Add to Cart" button. You have a cart panel. It shows items, quantities, and totals. Three different components but somehow, they seem to be in sync with the data.

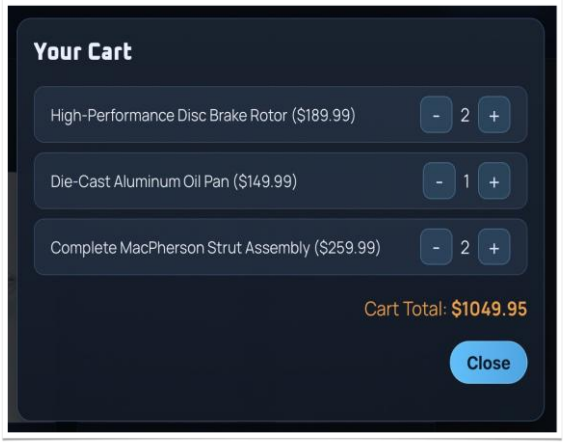


Now let's say you click on the "Add to Cart" button for the disc brake rotor. It will then be automatically added to the cart.

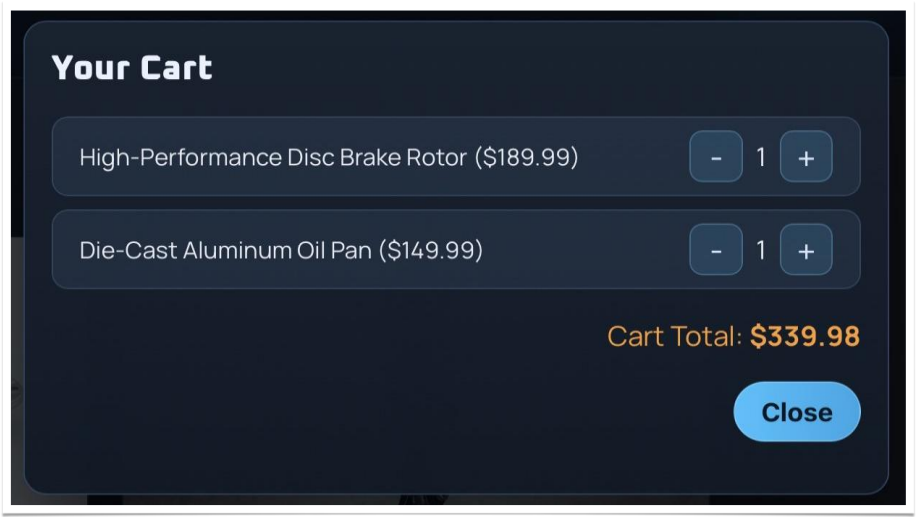
Now when you open the cart, you'll see the following entry for the disc brake rotor, along with plus and minus buttons to increase or decrease the quantity.



If you go back and add a few more items, you'll see that the cart is automatically updated with the new items.



As we saw, we can manage item quantities directly in the cart itself. I removed some items that I didn't need.



Source Code Repository

You can access the complete source code for this project from the official GitHub repository: [redux-simplified GitHub Repository](https://github.com/RikamPalkar/redux-simplified)

https://github.com/RikamPalkar/redux-simplified

2

Slice and Store

Overview

In this chapter, we will explore how to create Redux slices and configure the global store using Redux Toolkit. You will learn how reducers, actions, initial state, and TypeScript types work together to manage application state in a clean and scalable way.

Before You Write a Single Component

We build the store first. Go ahead and create a folder named store under src.

Why a Folder Called store?

When a new developer joins your project, they need to find things fast. The folder name store is a signal. It says: everything inside here is Redux. State lives here. Logic lives here. UI does not. You will never find JSX in store/. You will never find API calls or styling. Just state shape, transitions, and types.

That discipline, keeping concerns separated, is what makes large codebases survivable.

Two files live here right now. Create them:

```
store/
├── store.ts
└── cart-slice.ts
```

They will grow across this guide, but the boundary never moves.

Start With the Shape: What Is a Slice?

A slice is a self-contained piece of your Redux state. It holds three things together in one file: the state shape, the initial state, and the reducers that change it. We named it cart-slice.ts because it owns exactly one slice of the global store, the cart. Nothing else. The naming convention [feature]-slice.ts is intentional. When your app grows, and you have user-slice.ts, orders-slice.ts, and payments-slice.ts, you know exactly what each file owns just by reading the name.

Open cart-slice.ts

The Shape

Before you write any logic, answer one question: what does the cart actually look like?

```
export type CartItem = { id: string;
title: string; price: number; quantity: number;
}

type CartState = { items: CartItem[];
}
```

This is your contract. Every item in the cart must have those four fields. CartItem is exported; CartState is not. CartItem will be used by components later they need to know the shape of what they are rendering. CartState is internal. It is the store's concern, not the UIs.

The Empty Cart

```
const initialState: CartState = { items: []
};
```

Ask yourself: what does the app look like the very first time it loads? Redux always needs an initial state because on first load, there is nothing in the store yet. The reducer fires with undefined as the current state, and initialState is what it falls back to.

createSlice

createSlice from Redux Toolkit bundles three things together the name, the initial state, and the reducers.

```
export const cartSlice = createSlice({ name: "cart",
initialState,

reducers: { addToCart(...), removeFromCart(...)
}
});
```

The name field matters more than it looks. Redux Toolkit uses it to automatically generate action type strings. So, when addToCart fires, the action type becomes "cart/addToCart".

Reducers

Every reducer function receives two parameters automatically. state is the current value of the cart Redux passes it in for you. action is the object that was dispatched. It always has a type and a payload. When a component calls dispatch(addToCart(product)), that product becomes action.payload.

addToCart

When someone clicks Add to Cart, one of two things is true: either the product is not in the cart yet so you add it with quantity 1 or it is already there, in which case you just increment the quantity.

```
addToCart(state, action: PayloadAction<CartItem>) { const
existingIndex = state.items.findIndex(
(item) => item.id === action.payload.id
);

if (existingIndex >= 0) { state.items[existingIndex].quantity++;
} else {
state.items.push({ ...action.payload, quantity: 1 });
}
}
```

That state.items.push(...) and quantity++ look like mutation. In plain JavaScript that would be illegal in a Redux reducer. But Redux Toolkit uses Immer under the hood. Immer watches your mutations, intercepts them, and produces a new immutable state object behind the scenes. So you write readable code and still get pure functional behaviour.

removeFromCart

Remove works like the mirror image of add:

```
removeFromCart(state, action: PayloadAction<{ id: string }>) { const
existingIndex = state.items.findIndex(
(item) => item.id === action.payload.id
);

if (existingIndex >= 0) {
if (state.items[existingIndex].quantity === 1) {
state.items.splice(existingIndex, 1);
```

```
} else { state.items[existingIndex].quantity--;
}
}
}
```

If quantity is 1 and the user clicks minus, remove the item entirely. If it is more than 1, just decrement.

Export the Actions

```
export const { addToCart, removeFromCart } = cartSlice.actions;
```

Redux Toolkit generated these action creators automatically when you defined the reducers. This line just pulls them out so components can import and dispatch them.

The Complete cart-slice.ts

```
import { createSlice, PayloadAction } from "@reduxjs/toolkit";
export type CartItem = {
id: string; title: string; price: number; quantity: number;
}

type CartState = { items: CartItem[];
}
const initialState: CartState = { items: [] }; export const
cartSlice = createSlice({
name: "cart", initialState,
```

```
reducers: {
addToCart(state, action: PayloadAction<{id: string, title: string,
price: number}>) {
const itemIndex = state.items.findIndex((item) => item.id ===
action.payload.id);
if (itemIndex >= 0) { state.items[itemIndex].quantity++;
} else {
state.items.push({ ...action.payload, quantity: 1 });
}
},
removeFromCart(state, action: PayloadAction<string>) {
const itemIndex = state.items.findIndex((item) => item.id ===
action.payload);
if (state.items[itemIndex].quantity === 1) {
state.items.splice(itemIndex, 1);
} else { state.items[itemIndex].quantity--;
}
},
},
});

export const { addToCart, removeFromCart } = cartSlice.actions;
```

Now, build the Store

store.ts has one job: to take all your slices and combine them into a single global store.

```
export const store = configureStore({ reducer: {
cart: cartSlice.reducer
},
});
```

Right now there is only one slice. That is fine. As the app grows, the user slice and the orders slice each get a key here. The store is in one place.

The Two Types You Will Use Everywhere

```
export type RootState = ReturnType<typeof store.getState>; export
type AppDispatch = typeof store.dispatch;
```

RootState is for reading: every component that reads from the store needs to know what is in it. AppDispatch is for writing: every component that sends actions needs this to stay type safe.

The Complete store.ts

```
import { configureStore } from "@reduxjs/toolkit"; import {
cartSlice } from "./cart-slice";

export const store = configureStore({ reducer: { cart:
cartSlice.reducer },
});

export type RootState = ReturnType<typeof store.getState>; export
type AppDispatch = typeof store.dispatch;
```

3

Custom Hooks and UI Shell

Overview

In this chapter, we will explore how to connect Redux with React components using Provider, custom hooks, and typed selectors. You will also build the UI structure of the application, including the cart modal, product components, and overall application layout.

Connecting the Store to React

In the last chapter we built the Store. But your React components have no idea it exists. A component can't just reach into the store directly React and Redux are two separate worlds. Something has to connect them. That is what this chapter is about: the bridge.

Two things need to happen. First, the store needs to be injected into the React tree. Second, every component that talks to Redux needs a safe, typed way to do it.

Why Not Just Use the Raw Hooks?

React-Redux ships with useSelector and useDispatch. They work out of the box. But they are untyped. useDispatch() returns a generic dispatch function. useSelector does not know what your state looks like. Every component would need to import RootState and AppDispatch and wire the types manually every single time. That is repetition. Instead, we create typed wrappers once in hooks.ts, and then every component imports from there.

Create hooks.ts

```
import { useDispatch, useSelector, type TypedUseSelectorHook } from
"react-redux"; import { AppDispatch, RootState } from "./store";

type DispatchFunc = () => AppDispatch;
export const useCartDispatch: DispatchFunc = useDispatch;
export const useCartSelector: TypedUseSelectorHook<RootState> =
useSelector;
```

We are simply wrapping the raw hooks with our own types so every component gets type safety for free. useCartDispatch is for writing. useCartSelector is for reading. TypedUseSelectorHook comes from React-Redux, a generic that wraps useSelector with your specific state type.

The UI Shell

The store is wired, and the hooks exist. Now we build the UI shell, the structure that Redux will eventually power. Create a folder named components under src.

index.html

```
<!doctype html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>Redux + TypeScript</title>
</head>
<body>
<div id="modal"></div>
<div id="root"></div>
<script type="module" src="/src/main.tsx"></script>
</body>
</html>
```

There are two divs. root is standard React mounts your entire app here. But the modal is separate, sitting above it in the DOM. This is where the cart dialog will be rendered. Not inside the component tree. Above it.

Cart.tsx The Portal

```
import { createPortal } from 'react-dom'; import CartItems from
'./CartItems.tsx';

type CartProps = { onClose: () => void; };

export default function Cart({ onClose }: CartProps) { return
createPortal(
<>
<div className="cart-backdrop" />
<dialog id="cart-modal" open>
<h2>Your Cart</h2>
<CartItems />
<p id="cart-actions">
<button onClick={onClose}>Close</button>
</p>
</>,
document.getElementById('modal')!
);
}
```

The cart renders as a dialog a proper HTML modal element. Notice the second argument to createPortal: document.getElementById('modal'). By portaling to #modal which sits outside #root entirely, the cart floats above everything, without any CSS battles.

Header.tsx The Cart Button That Does Nothing Yet

```
export default function Header() { function handleOpenCartClick() {}

return (
<header id="main-header">
<h1>learn redux + TypeScript with cart feature</h1>
<p>
<button onClick={handleOpenCartClick}>Cart ({0})</button>
</p>
</header>
);
}
```

handleOpenCartClick is empty. The button renders. Nothing opens. That is temporary. Cart ({0}) has a hardcoded zero for the count. That zero will be replaced by a selector reading from the store in Chapter 4.

Product.tsx The Add Button That Does Nothing Yet

```
type ProductProps = {
id: string; image: string; title: string; price: number;
description: string;
};

export default function Product({ id, image, title, price,
description }: ProductProps) {
function handleAddToCart() {}

return (
<article className="product">
<img src={image} alt={title} />
<h3>{title}</h3>
<p>${price}</p>
<button onClick={handleAddToCart}>Add to Cart</button>
</article>
);
}
```

handleAddToCart is empty. The only missing piece is one line: dispatch(addToCart(...)). That line arrives in Chapter 4.

Shop.tsx The Container

```
import { type ReactNode } from 'react'; type ShopProps = { children:
ReactNode; };
export default function Shop({ children }: ShopProps) { return (
<section id="shop">
<h2>Performance Auto Parts Catalog</h2>
<ul id="products">{children}</ul>
</section>
);
}
```

Shop is a layout wrapper. It does not know what products exist. It just renders whatever is passed as children inside a ul. ReactNode is the TypeScript type for anything React can render.

App.tsx Where It All Comes Together

```
function App() { return (
<>
<Header />
<Shop>
{DUMMY_PRODUCTS.map((product) => (
<li key={product.id}>
<Product {...product} />
</li>
))}
</Shop>
```

```
</>
);
}
```

App pulls in the dummy product data and maps over it one Product per item. The spread {...product} passes every field as individual props, matching ProductProps exactly. This is the whole app at this stage. It renders. It looks like a store. Nothing interactive works yet.

Provider: Injecting the Store

Provider makes the store available to every component in the app. Without Provider, any component that calls useCartSelector or useCartDispatch will throw it can't find the store.

```
import { Provider } from 'react-redux'; import { store } from
'./store/store';

function App() { return (
<Provider store={store}>
<Header />
<Shop>...</Shop>
</Provider>
);
}
```

Provider wraps everything. Every component in this tree can now reach the store through hooks. This is the only place the store gets imported into the UI layer. Components never import the store directly; they use hooks.

The foundation is complete. Every piece is in place. What is missing is the moment a user clicks Add to Cart, and the store responds. That is next. 21

4

Writing and Reading State

Overview

In this chapter, we will explore how to connect Redux with React components using Provider, custom hooks, and typed selectors. You will also build the UI structure of the application, including the cart modal, product components, and overall application layout.

Now We Use It

In the past three chapters, we built the foundation of Store, Slice, Hooks, and Provider. Now we use it. This chapter is about two things: writing to the store when a user clicks a button and reading from the store to update the UI. By the end, the cart works. Completely. End-to-end.

Writing: Dispatching Actions

Product.tsx The Add to Cart Button

The handleAddToCart stub in Product.tsx has been empty since Chapter 3. Now it does something.

```
const dispatch = useCartDispatch(); function handleAddToCart() {
dispatch(addToCart({ id, title, price }));
}
```

useCartDispatch() gives you the dispatch function. addToCart({ id, title, price }) builds the action. Notice what is in the payload id, title, price. Not quantity. The reducer sets quantity: 1 on first add, and increments on subsequent adds. The component does not decide that. The reducer does.

Components describe what happened. Reducers decide what changes.

When the user clicks Add to Cart: handleAddToCart fires → dispatch sends the action to the store → the addToCart reducer runs → state updates → every component reading that state re-renders.

That is the full write cycle.

CartItems.tsx Plus and Minus Buttons

The cart panel has two buttons per item. Plus adds one more. Minus removes one, or removes the item entirely if quantity is already 1.

```
const dispatch = useCartDispatch(); function handleAddToCart(item:
CartItem) {
dispatch(addToCart(item));
}

function handleRemoveFromCart(id: string) {
dispatch(removeFromCart(id));
}
```

handleAddToCart here passes the full CartItem because the item already exists in the cart. handleRemoveFromCart only needs the id. The component does not decide what happens it just says: this item should be removed. The reducer handles the rest.

Reading: Selectors

Writing pushes data into the store. Reading pulls it out. Two components need to read Header for the cart count badge, and CartItems for the full item list and total price.

Header.tsx The Cart Count

```
const cartQuantity = useCartSelector( (state) =>
state.cart.items.reduce(
(total, item) => total + item.quantity, 0
)
);
```

useCartSelector reads from the store. The callback receives the full state. state.cart.items is the array of cart items from cart-slice.ts. reduce walks through every item and adds up the quantities. If the cart has 3 rotors and 2 pistons, cartQuantity is 5.

```
<button onClick={handleOpenCartClick}>Cart ({cartQuantity})</button>
```

That hardcoded {0} from Chapter 3 is gone.

CartItems.tsx Items and Total Price

CartItems is the only component in this app that both reads and writes at the same time.

```
const cartItems = useCartSelector((state) => state.cart.items);
const totalPrice = cartItems.reduce(
(total, item) => total + item.price * item.quantity, 0
);

const formattedTotalPrice = `$${totalPrice.toFixed(2)}`;
```

cartItems is the raw array straight from the store that is all the store holds. totalPrice is derived at read time: price multiplied by quantity, summed across all items. The store never holds a total. Totals are calculated when needed, from the source data.

```
{cartItems.length === 0 && <p>No items in cart!</p>}
{cartItems.length > 0 && (
<ul id="cart-items">
{cartItems.map((item) => (
<li key={item.id}>
<span>{item.title}</span>
<button onClick={() => handleRemoveFromCart(item.id)}>-</button>
<span>{item.quantity}</span>
<button onClick={() => handleAddToCart(item)}>+</button>
</li>
))}
</ul>
)}
<p>Cart Total: <strong>{formattedTotalPrice}</strong></p>
```

Empty cart shows the message. Items in cart renders the list. cartItems from the selector drives both. The component just reacts.

The Full Picture

One click at the top. Automatic update at the bottom. Nothing in between needs to be managed manually.

This is why Redux scales. The component does not coordinate with other components. It talks to the store. The store broadcasts the change. Everyone who is listening updates.

What You Built Across This Series

File	Responsibility
`cart-slice.ts`	State shape, initial state, add/remove reducers, exported actions
`store.ts`	Configured global store, RootState and AppDispatch types
`hooks.ts`	useCartDispatch for writing, useCartSelector for reading
`App.tsx`	Provider wrapping the entire component tree
`Product.tsx`	Dispatches addToCart on button click
`CartItems.tsx`	Reads items, derives total, dispatches add and remove
`Header.tsx`	Reads cart quantity always accurate

Where to Go Next

You now know the full Redux pattern: Slice → Store → Hooks → Dispatch → Select.

The best way to solidify it is to add one more slice yourself. A wishlist. A recently viewed list. A product comparison panel. Anything with state that multiple components need to share.

Follow the same sequence every time: define the shape, write the reducers, export the actions, wire the hooks, dispatch from components, select in the UI.

Final Summary

Redux is often introduced as a state management library, but throughout this guide, you saw that it is really a system for keeping applications predictable as they grow.

We began with the fundamentals understanding why shared state becomes difficult once multiple components need the same data. Instead of passing props endlessly between components, Redux introduced a centralized Store where all state changes flow through a single predictable path.

From there, we built the foundation using Redux Toolkit:

- Creating slices
- Defining reducers
- Configuring the store
- Exporting actions
- Typing everything safely with TypeScript

Once the foundation was ready, we connected Redux to React using Provider, custom hooks, selectors, and dispatch functions. Then finally, we made the application fully interactive:

- Adding products to the cart
- Updating quantities
- Removing items
- Calculating totals

Synchronizing multiple UI components automatically Most importantly, you learned the actual Redux workflow:

UI → Dispatch Action → Reducer Updates State → Store Changes → UI Re-renders

That single unidirectional flow is the reason Redux remains reliable even in large applications.

The cart application you built may look simple on the surface, but it demonstrates the exact same architectural principles used in production-scale React systems.

At this point, you are no longer just reading Redux code. You can now structure slices, manage global state, connect components properly, and reason about application data flow with confidence.

The next step is practice. Build another feature. Add another slice. Extend the application. Redux becomes truly intuitive once you begin designing your own state flows.

C# Corner

OUR MISSION

Free Education is Our Basic Need! Our mission is to empower millions of developers worldwide by providing the latest unbiased news, advice, and tools for learning, sharing, and career growth. We're passionate about nurturing the next young generation and help them not only to become great programmers, but also exceptional human beings.

ABOUT US

CSharp Inc, headquartered in Philadelphia, PA, is an online global community of software developers. C# Corner served 29.4 million visitors in year 2022. We publish the latest news and articles on cutting-edge software development topics. Developers share their knowledge and connect via content, forums, and chapters. Thousands of members benefit from our monthly events, webinars, and conferences. All conferences are managed under Global Tech Conferences, a CSharp Inc sister company. We also provide tools for career growth such as career advice, resume writing, training, certifications, books and white-papers, and videos. We also connect developers with their potential employers via our Job board. Visit C# Corner

MORE BOOKS