✅ Chapter 12 – Redux (React State Management Library)

### 📘 **Main Highlights**

1. What is Redux?
2. Advantages of Redux
3. Disadvantages of Redux
4. Redux Toolkit (RTK)
5. Redux Store
6. Redux Slices
7. RTK Architecture
8. Setting Up Redux in a React Application
9. How to Create a Redux Store
10. How to Provide the Redux Store to a React Application
11. How to Create Slices Using createSlice()
12. What Are Reducer Functions and How Do They Work?
13. How to Add Slices to the Redux Store
14. How a React Application Subscribes to the Redux Store (Understanding useSelector)
15. How to Dispatch Actions in Redux (Using useDispatch)
16. Different Ways to Handle Click Events in React
17. Guidelines and Best Practices
18. A closer look at redux dev tools
19. Introduction to RTK Query
20. Understanding Synthetic Events in React

### **What is Redux?**

Redux is a **state management library** used in JavaScript applications to manage and centralize application state.  
It is most commonly used with **React.js**, but it is **library-agnostic**, meaning it can also be integrated with other frameworks like **Angular, Vue,** or even plain JavaScript.

Redux helps you:

* Centralize your app’s state in one place, called the store, making it easier to manage.
* Update state in a predictable way using simple functions called reducers, ensuring consistency.
* Simplify complex state management by making sure data flows in only one direction, which reduces confusion.

### *Simple Analogy -*

Think of Redux as a **global brain** for your app. Instead of each component remembering its own state (like local memory), Redux allows everyone to **talk to one shared memory,** making state easier to control, debug, and test.

***Note -*** Predictable way means that state updates in Redux always follow a consistent pattern, where, when an action is triggered, the respective reducer function is invoked to cause a change or update in the state.

### Advantages of Redux

***1.Centralized State Management****:*

In React, state is often managed locally within components, and to share this state across components, props are passed down from parent to child, and Callbacks are used from child to parent. Redux, however, stores state globally in a central place called the **Redux Store**, making data accessible to all components that need it, without the need for prop drilling.

***2. Prevents Prop Drilling****:*

Redux helps avoid **prop drilling**, which is the process of passing data through multiple layers of components. This can be cumbersome and hard to manage in large applications, but Redux provides a cleaner solution by centralizing the state.

***3. Better for Large Applications****:*

Redux is generally preferred over the React Context API for **large-scale applications** due to its optimized performance and structured approach to managing complex state across many components.

***4. Performance Optimizations****:*

In React, components are rerendered whenever the state or props change. With Redux, when data in the store changes, a shallow copy of that data is created, which helps minimize unnecessary rerenders, leading to improved performance, especially in large apps.

### Disadvantages of Redux

***1. Steep Learning Curve****:* Redux can be challenging to learn, especially for beginners, due to its concepts like actions, reducers, middleware, and store. Understanding the flow of data can take time for those new to state management.

***2. Not Suitable for Small Applications****:* For small-scale applications with simple state needs, Redux may be overkill. The additional complexity of setting up Redux may not justify the benefits for smaller apps.

***3. Complex Configuration****:* Setting up the Redux store and integrating it with React (or any other framework) can be complex and time-consuming, requiring a good understanding of how the Redux flow works.

***4. Boilerplate Code****:* Redux often involves writing a lot of repetitive **boilerplate code**, such as actions, reducers, and action creators, which can make the codebase harder to maintain and read.

***5. Difficult Debugging****:* Debugging in Redux can be tricky because of its complex data flow, especially when handling asynchronous actions. Tracing changes through actions and reducers can become cumbersome.

**What is Redux Toolkit (RTK)?**

**Redux Toolkit (RTK)** allows developers to write Redux code more concisely and efficiently. It abstracts away the boilerplate code typically required in Redux, enabling developers to write clean, maintainable code with less effort.

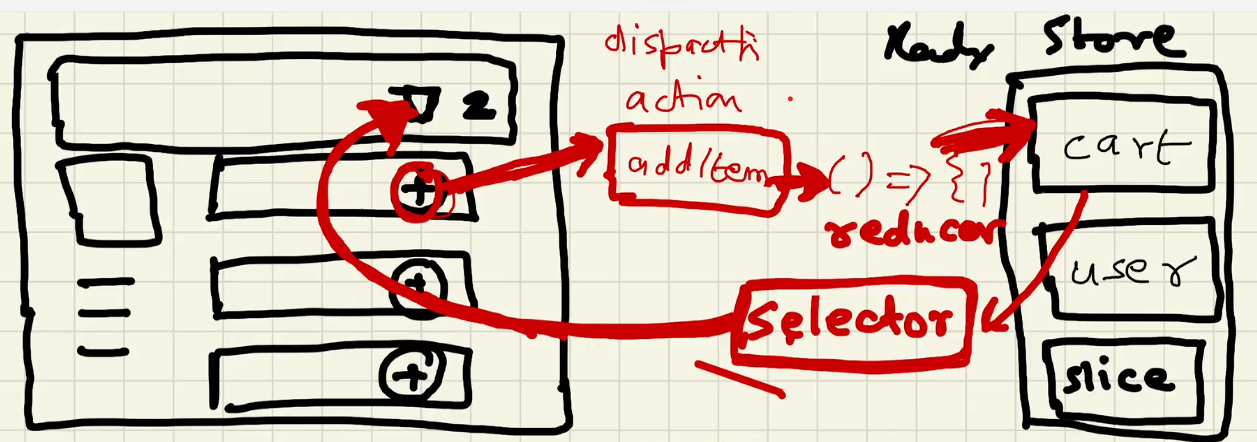
What is Redux Store?

The Redux store is a large JavaScript object that serves as a central hub for storing data needed by any component in a React app. While multiple React contexts can be used to store different pieces of data, Redux uses a single store to manage global data across the entire application. To simplify data management, Redux introduces the concept of **slices**, which group related pieces of state, actions, and reducers together. This approach reduces the setup required compared to manually defining reducers and actions, making the codebase more organized and easier to maintain.

What are Slices?

Slices are logical separations or portions within the Redux store. Each slice maintains a mapping between actions and reducer functions, ensuring that when an action is dispatched, the corresponding reducer is executed. In short, slices not only store data in portions but also manage and update that data. For example, in an e-commerce application, the store can include various slices: a user slice to handle data related to the logged-in user, a theme slice to manage light and dark mode settings for the application's UI, and a cart slice to manage data associated with the shopping cart.

Explain Redux Toolkit Architecture?

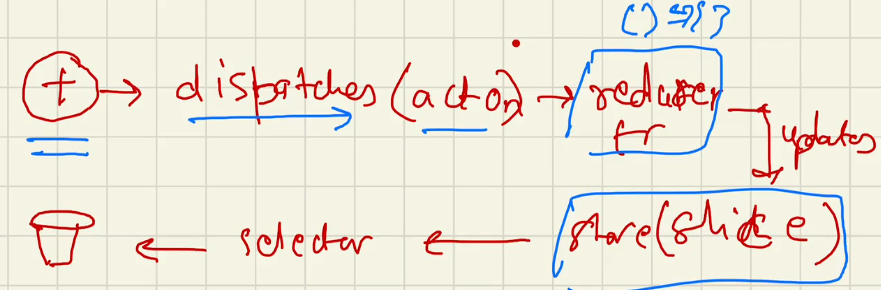


In an e-commerce application, the typical process involves adding items to the cart and then proceeding to checkout.

In the context of Redux, when you click the plus (+) button in the UI, an addItem action is dispatched. This action then triggers a reducer function that updates the **cart slice** of the store.

To read data from the cart slice and update the cart item count in the UI, we use a **selector**. Essentially, a selector subscribes to the store, meaning the UI will automatically be notified and updated whenever the cart slice data changes. This is achieved using the useSelector hook.

The term **selector** is used because it refers to selecting a specific portion (or slice) of the Redux store.



**Note:** When we click the plus button, we can’t directly update the store because we don’t want random components to modify the store without control. Redux Toolkit helps manage the state of a React application in a predictable and organized way. To maintain consistency and structure, state changes should not happen arbitrarily. Instead, specific actions and reducer functions must be used to update the state. This approach ensures that Redux can properly track and manage state changes.

**Note:** Whenever a component uses a selector, it means the component is subscribed to the store. If the data in the store changes, the component’s data will automatically update. Redux ensures that components using useSelector are re-rendered immediately when the relevant slice of the store changes. This behavior is known as **subscribing to the store.**

Setting up Redux in a React Application

How to setup Redux in a React App?

We install two libraries.

1. npm i @reduxjs/toolkit -This library provides functions to create the Redux store and slices. It offers the core functionality of Redux, making state management easier and more efficient.
2. npm i react-redux -This library acts as a bridge between the React application and Redux, allowing React components to interact with the Redux store.

How to create a Redux-Store?

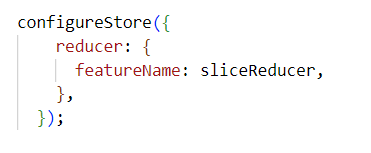
The configureStore function from Redux Toolkit is used to create a Redux store.



***Empty Configuration in Redux Toolkit***

When you pass an empty object {} to the configureStore function, you're telling Redux Toolkit to create a store **without any slices (i.e., no reducers are registered).** The store will still be created successfully. but the store won't manage any state or respond to actions. Without reducers,

* the store cannot **store** any state.
* the store cannot **dispatch actions** to update the state.
* Components **can’t subscribe** to or select state from it.

To make your store useful, you must define one or more **slices** (which include reducers) and register them using the reducer key in the configureStore() function:

Here, featureName is the key (e.g., cart, user) and sliceReducer is the reducer function from createSlice().

Leaving the store empty is fine **during initial setup or for testing purposes,** especially when you’re scaffolding the project structure.

But for real applications, **at least one slice** should be added to manage part of the app’s state.

How do I provide the Redux store to our application?

Our store and application are separate entities. The react-redux library provides a Provider component that acts as a bridge between the Redux store and the React application. To establish this connection, we wrap the root component of the application with the Provider component. We then pass the store as a prop to Provider, making the store available to the entire application.



**Provider** is a named export from react-redux that expects the store as a prop.  
The prop name must be store and should not be changed. We wrap our entire app with the Provider to make the store available throughout the application.

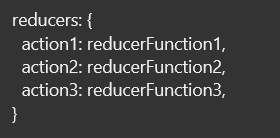
If we want only a specific portion of the app to use Redux, we can wrap just that part with the Provider component, ensuring the store is accessible only to that section.

How to create Slices inside Redux Store?

Redux Toolkit (RTK) provides an API called createSlice that allows us to create slices of the state. In the example below, we are creating a cart slice.



* name represents the name of the slice, which is 'cart' in this case.
* initialState is an object that defines the initial state of the slice. Here, we set items to an empty array because the cart initially has no items.
* reducers is an object where action names are mapped to their corresponding reducerfunctions, which handle how the state should change in response to those actions.



* At line 23, cartSlice.reducer refers to a single reducer function that combines all the individual reducer functions defined in the slice. This consolidated reducer is then used in the store configuration.
* At line 21, we destructure all the actions from cartSlice.actions and export them together, making them easy to use throughout the application.

Note: A **reducer function** modifies the data within a **slice**, which is a **part of the Redux store.**

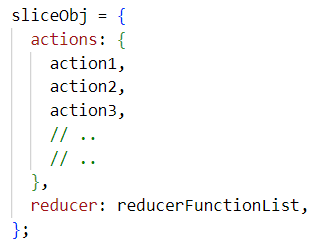
**Each reducer function** receives a state parameter, which provides access to the slice's state object.  
Initially, this state is { items: [] }, meaning the slice starts with an empty item array as its default value.

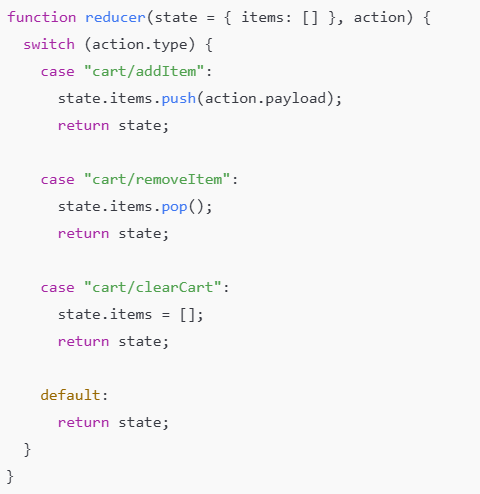
Inside the reducer function, the state is updated based on the logic defined inside the reducer function, which handles the action.

The createSlice function returns a slice object that contains both

1. **reducer -** which combines all the individual case-specific reducer functions, and
2. **actions -** a collection of action creators that are automatically generated based on the names of the reducer functions.

***Internal representation of a slice object -***



***Internal representation of cartSlice.reducer -***

It's a single **reducer function** that looks roughly like this internally.

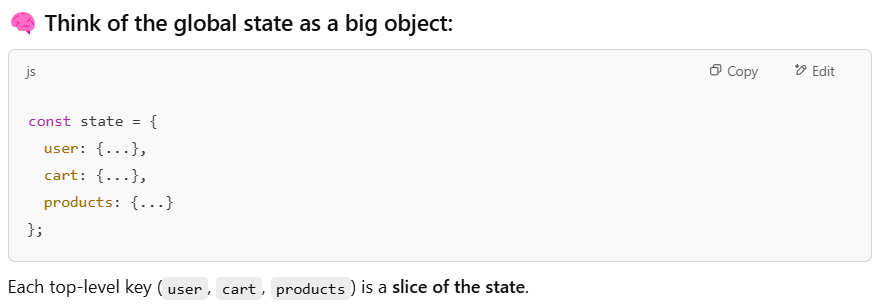
***Internal representation of cartSlice.actions -***

Internally, it's an object that looks like this -



These are just **action creator functions -** when called, they return action objects.

***Note -* Slices of the state** refers to **individual parts or sections of the global Redux state**, each responsible for handling a specific piece of data or functionality in your application.



### What are reducer functions?

Reducer functions are standard JavaScript functions that are invoked whenever an associated action is dispatched. They contain the logic for managing state, enabling updates to the state based on the action that is triggered.

A reducer function takes two parameters: state and action. The state represents the current state, while the action is a plain JavaScript object, often including a payload - i.e., data sent from the UI or other sources.

If no action is dispatched, the state defaults to its initial value (e.g., an empty array items = [] in the case of a cart slice). When an action is triggered, the reducer processes it and updates the state (e.g., the items array) based on the defined logic.

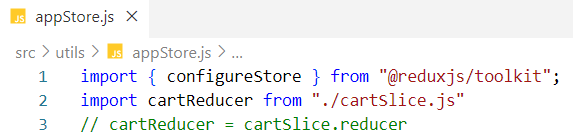
How do we add slices to the Redux store?

In Redux Toolkit, the configureStore method is used to create the Redux store. Slices are added by including their reducers inside the reducer property of the configuration object passed to the configureStore function.

Steps to Add Slices to the Redux Store -

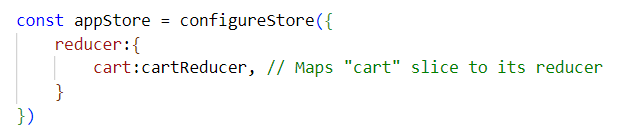
***1. Import configureStore and Slice reducers.***

First, import configureStore from Redux Toolkit, along with the reducers from the slice files.



### *2. Map Slice Reducers*

* The configureStore method takes a configuration object as its argument.
* This object includes a reducer property, which is itself an object.
* Inside the reducer object, each slice is added as a key-value pair
  + **Key**: The slice name (used as the key in the Redux state).
  + **Value**: The corresponding slice's reducer function.



In this example, the cart slice is added to the store by assigning its reducer (cartReducer) as a value, making it part of the store's state.

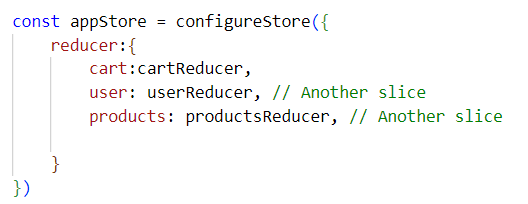
***3. Export the store***

Export the configured store so it can be used throughout the application.

***Key Notes -***

* The reducer property in configureStore serves as the root reducer for the entire application.
* Each slice reducer functions as a sub-reducer, responsible for managing a specific part of the state.
* Adding additional slices is simple - just include more key-value pairs within the reducer object.

Here is an example that demonstrates the use of multiple slices.



Now, the store will have three parts in its state: cart, user, and products.

How to read data from the Redux store in a React application?

The useSelector hook, provided by the react-redux library, allows React components to access data from the Redux store. It enables a component to "subscribe" to the store and read specific slices of the state. The component will automatically re-render whenever the selected part of the state changes.

***Purpose -***

The useSelector hook is used to access the Redux store’s state inside a React component.

***How it Works -***

* The useSelector hook takes a **selector function** as its argument.
* This selector function is used to extract a specific part (or "slice") of the state from the Redux store.

***Automatic Updates -***

When the slice of state selected by useSelector changes, the component automatically re-renders with the updated data from that slice.

### *Integration in React Applications -*

* React applications subscribe to the Redux store using the useSelector hook.
* This hook acts as a bridge between the Redux store and React, ensuring seamless access to the state and automatic updates when the state changes.

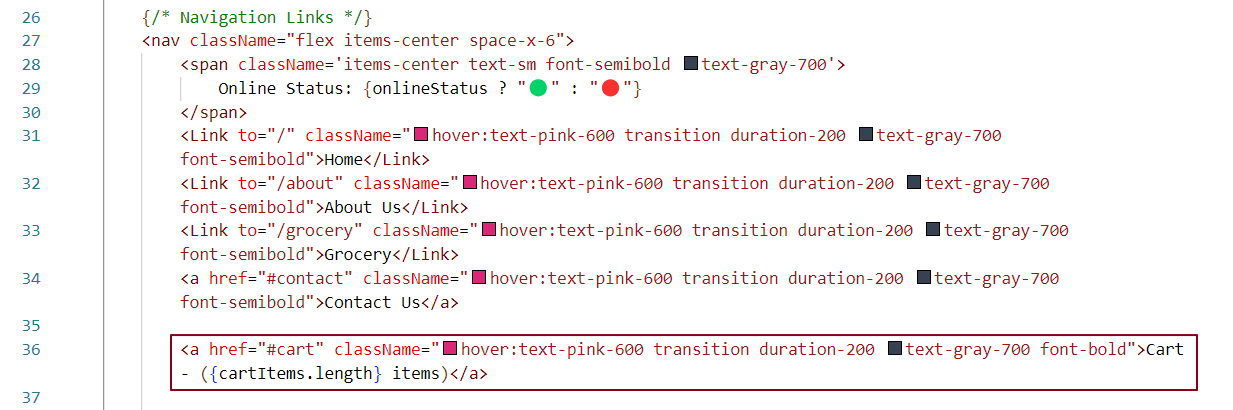
### *Accessing the Store -*

By using useSelector, we gain access to the Redux store and can retrieve the necessary data for rendering.

### *Why is it called useSelector?*

The name useSelector makes sense because it allows you to **select** a specific part of the Redux store. Instead of subscribing to the entire store, useSelector enables your component to access only the slice of state it needs.  
  
Let's use useSelector in our Header component to read the items from the cart slice of the store and display it in the header UI.





At line 16, we use the useSelector hook to subscribe our React application to the items property of the cart slice in the Redux store.

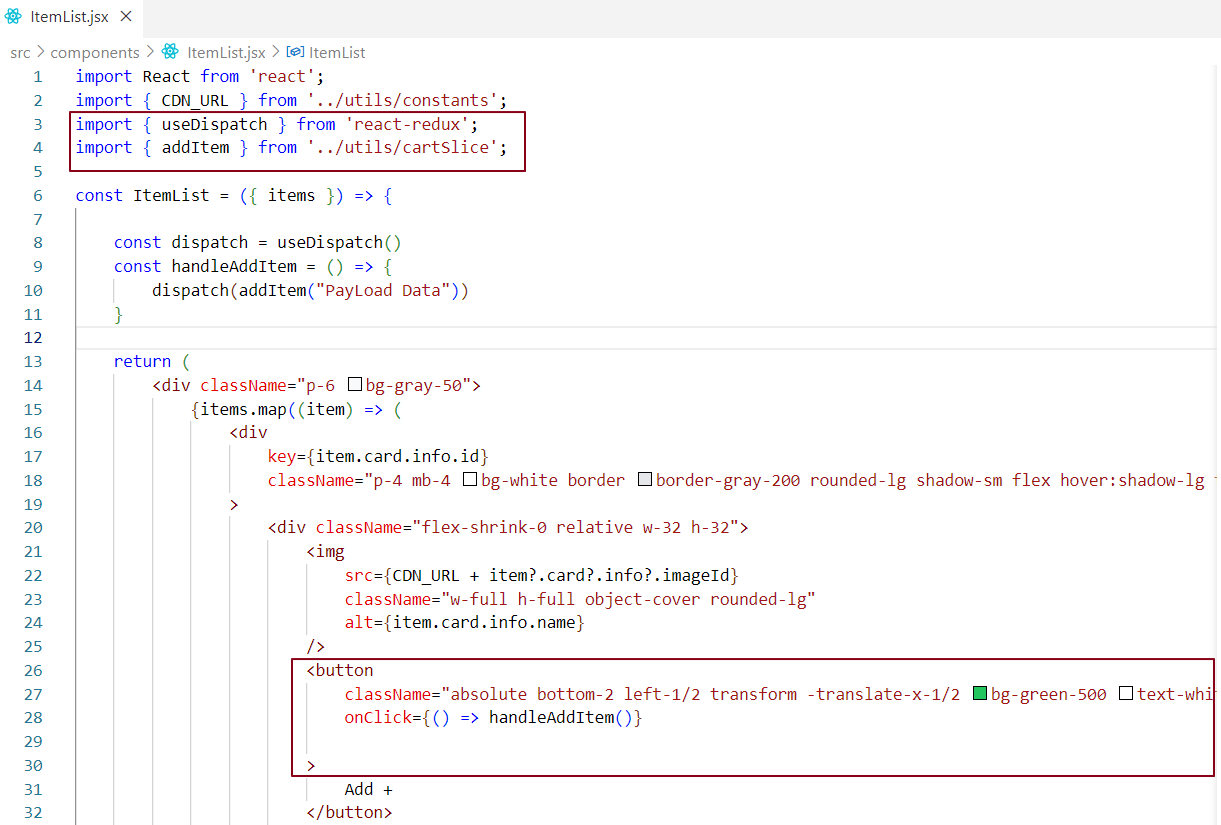
At line 36, we use the subscribed data to display it on the UI. Since the cart slice in the Redux store currently contains zero items, we expect 0 to be shown on the UI.

The useSelector hook provides access to the entire Redux store, but we use it to subscribe to a specific part of the store—in this case, the items property within the cart slice (store.cart.items).



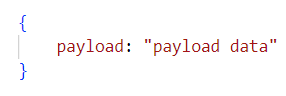
How Do We Dispatch an Action in Redux? ? How to Write Data to the Redux Store?

The react-redux library provides a hook called useDispatch, which returns a function. This function accepts an action and any associated data (payload) and is responsible for dispatching the action to the Redux store.

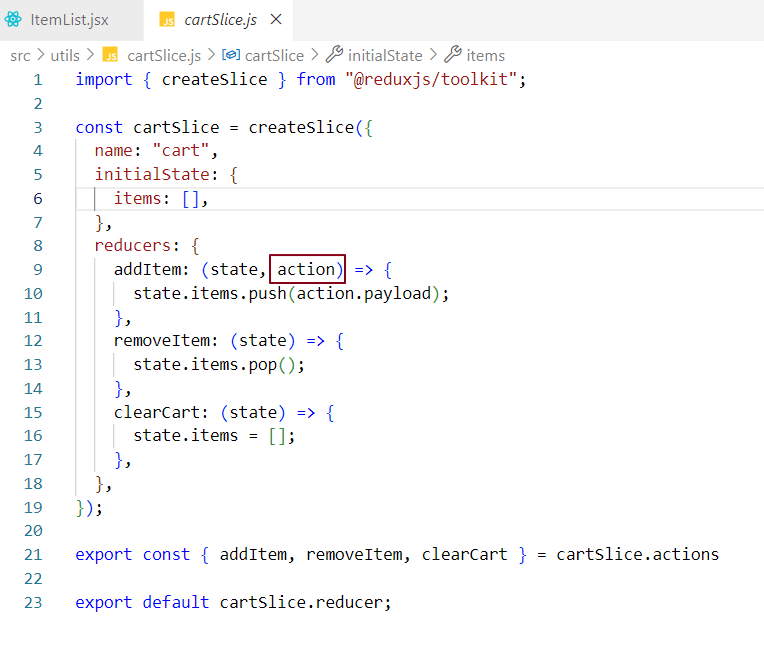


* **At line 4,** we import the addItem action, which is exported from the cartSlice.
* **At line 14,** we use the useDispatch hook, which returns a function named dispatch.
* **At line 27,** the handleAddItem function is triggered when the "Add +" button is clicked.
* Inside handleAddItem, the dispatch function is called, dispatching the addItem action along with PayloadData.
* The PayloadData is passed to the addItem action, which then forwards it to the corresponding reducer function. This reducer updates the items in the Redux store.

***Note****:* When an action is dispatched in Redux, it creates an action object that includes a payload property. The payload contains the data provided during the dispatch.

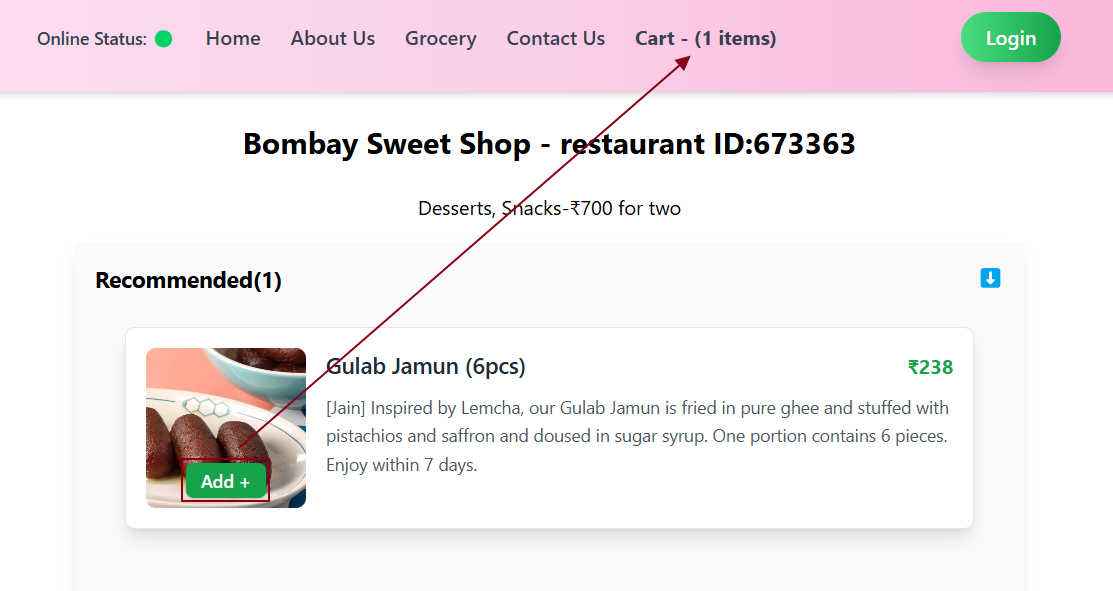


This action object is then passed as the second argument to the reducer function associated with the dispatched action.



The addItem reducer function receives the action object and updates the state based on the reducer logic.

**In this case**, when the "Add +" button is clicked, the cart slice updates its items using the PayloadData provided by the UI. As a result, the item count in the UI is updated. The useSelector hook then reads the updated value from the Redux store and displays it in the item count section of the UI.

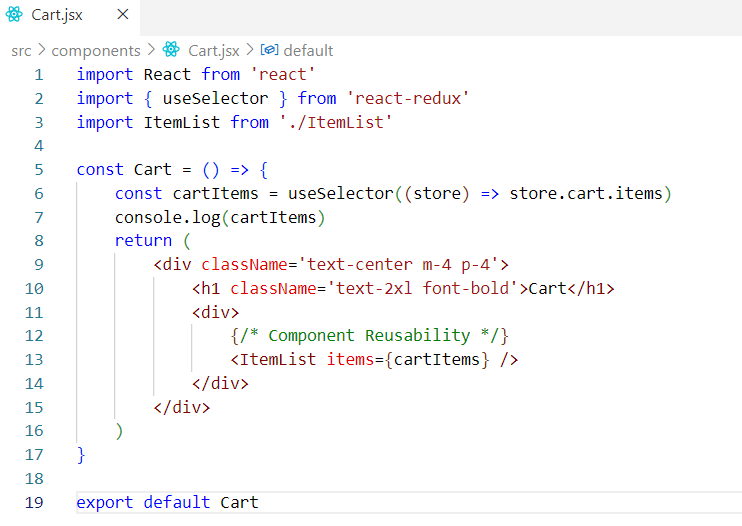


So far, we have been passing hardcoded data to the addItem action, which is dispatched when the Add+ button is clicked. Now, let's pass card-specific data when the Add+ button of a particular card is clicked.

When the Add+ button on an individual card component is clicked, the corresponding card data is added to the items state. Simultaneously the updated state is accessed using the useSelector hook, which updates the item count displayed in the UI.

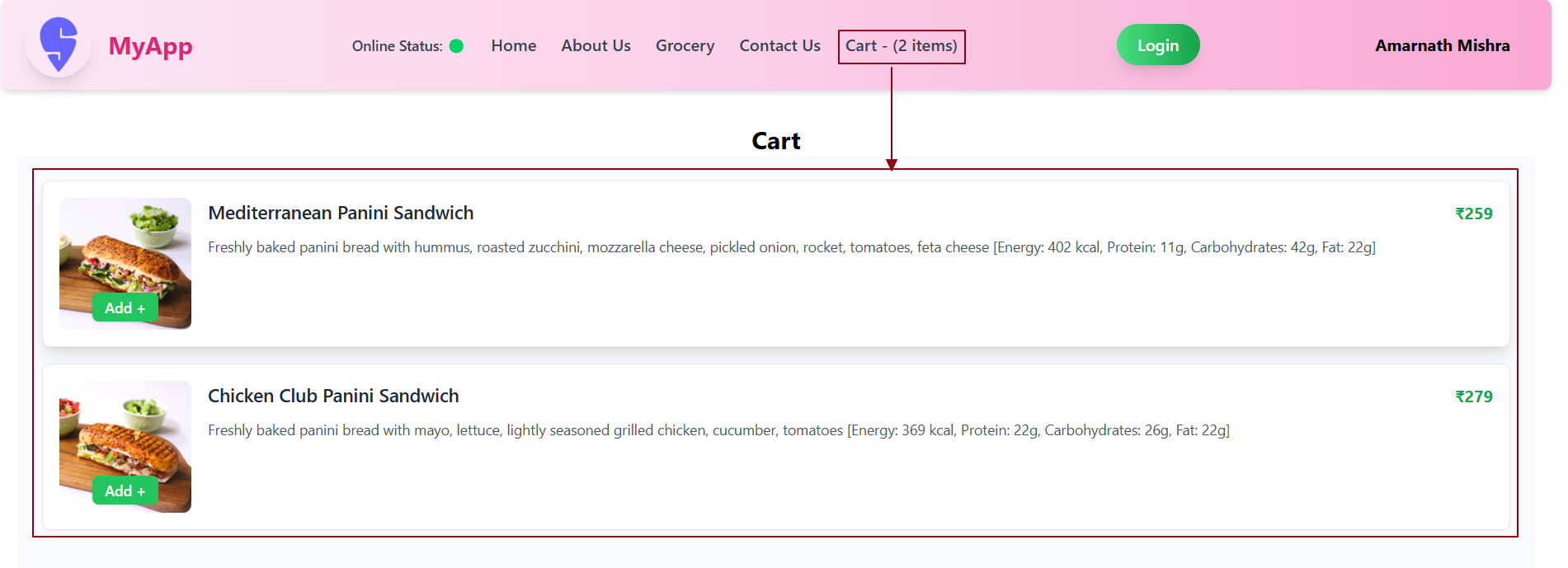
When navigating to the cart section, users should see a list of food items that have been added, allowing them to review their selections and proceed to checkout.

To achieve this, let’s create a Cart.jsx component that displays all the food cards added to the cart, enabling users to review their items and continue with the checkout process.



In this component:

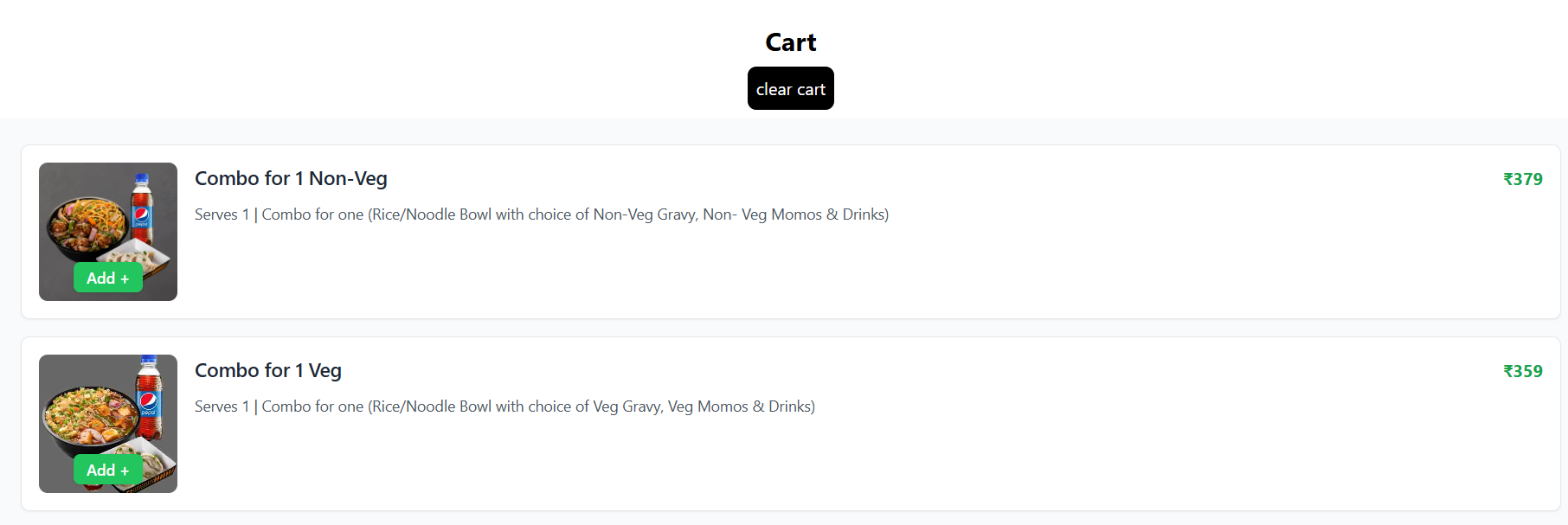
* The cartItems state holds the list of items for which the "Add+" button has been clicked.
* We are reusing the ItemList component to display the selected food cards on the **Cart** page.



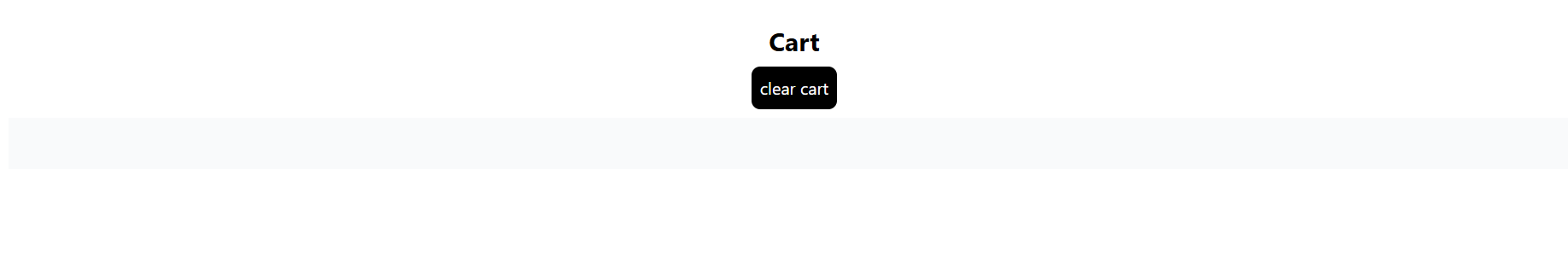
Let’s implement a **Clear Cart** functionality in the Cart component to allow users to remove all items from the cart with a single action.



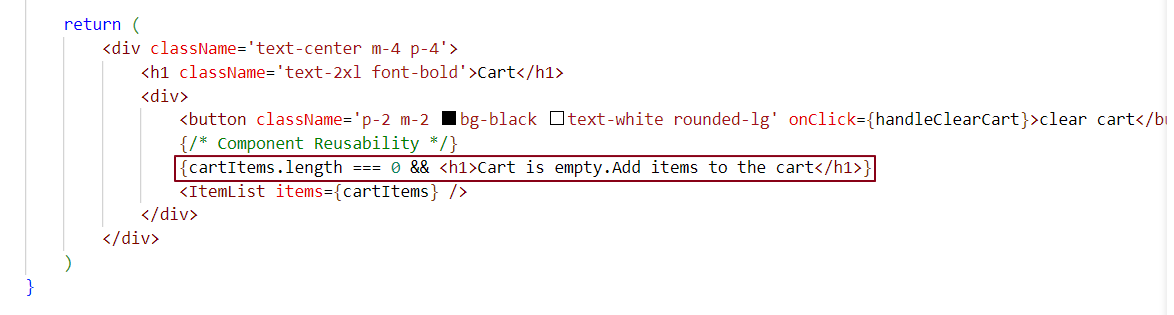
When the **Clear Cart** button is clicked, the handleClearCart function is invoked. This function dispatches the clearCart action, which triggers the corresponding reducer in the cart slice. The reducer then clears all cartItems by resetting the items state.

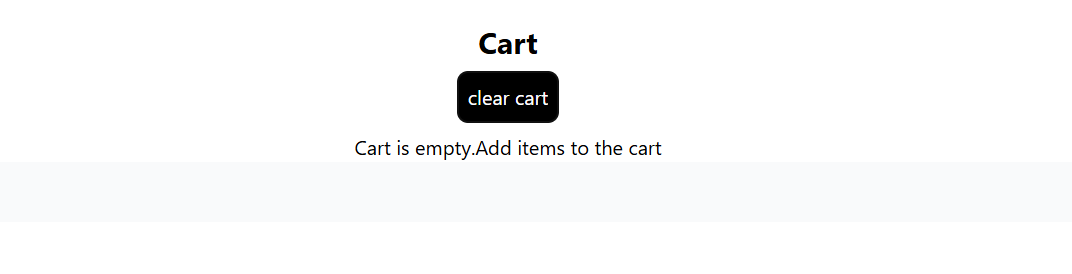


When the Clear Cart button is clicked, all food card components are removed from the UI, reflecting the cleared items state in the Redux store.



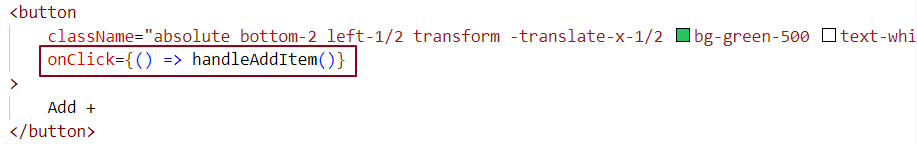
If the cart is empty, display the message: Cart is empty. Add items to the cart.





Different ways to handle click events in React -

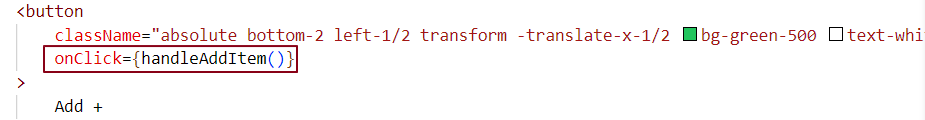
***Case 1 - Using an Arrow Function (Callback) –***



The arrow function () => handleAddItem() is executed only when the button is clicked.

This approach is used to ensure that handleAddItem() is called only during the click event, not during the render. It provides more flexibility, such as passing parameters or executing additional logic inside the function without triggering it prematurely.

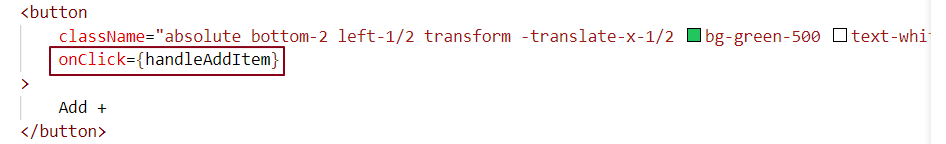
***Case 2 - Calling the Function Immediately (Incorrect Approach)***



When handleAddItem() is called directly in the JSX, it executes right away during the render process, not when the button is clicked. This causes the function to run every time the component re-renders, which is not the intended behavior.

The correct approach is to pass the function reference to onClick so that it only executes when the button is clicked, not during every render.

***Case 3 - Passing a Function Reference***



When you pass a function reference without parentheses (e.g., onClick={handleAddItem}), React stores the function and only calls it when the event (in this case, a click event) occurs.

This method avoids calling the function immediately during the render process, as seen in the previous case, and instead ensures that the function is executed only when the event is triggered.

***Note -***

It’s recommended to use onClick={() => handleAddItem()} for most event handling scenarios in React. This ensures that the function is executed only when the event occurs, and it allows you to include additional logic or parameters if needed, all while keeping the event handling clean and controlled.

Guidelines and Best Practices

In Redux, it's better to subscribe to a specific part of the store rather than the entire store, as this improves performance and reduces unnecessary re-renders.

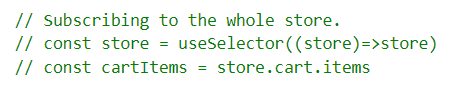
***Improved Performance******-***

Subscribing to the entire store might lead to unnecessary computations, especially in large applications with a complex state structure. By narrowing down the subscription, react avoids processing irrelevant state updates, resulting in better application performance.

***Reduced Re-Renders -***

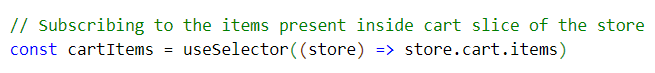
When you subscribe to the whole store, your component re-renders on any state change, even if it doesn’t affect the component. If you subscribe only to the part of the store your component needs, it re-renders only when that part changes.

***Subscribing to the Whole Store (Less Efficient) -***

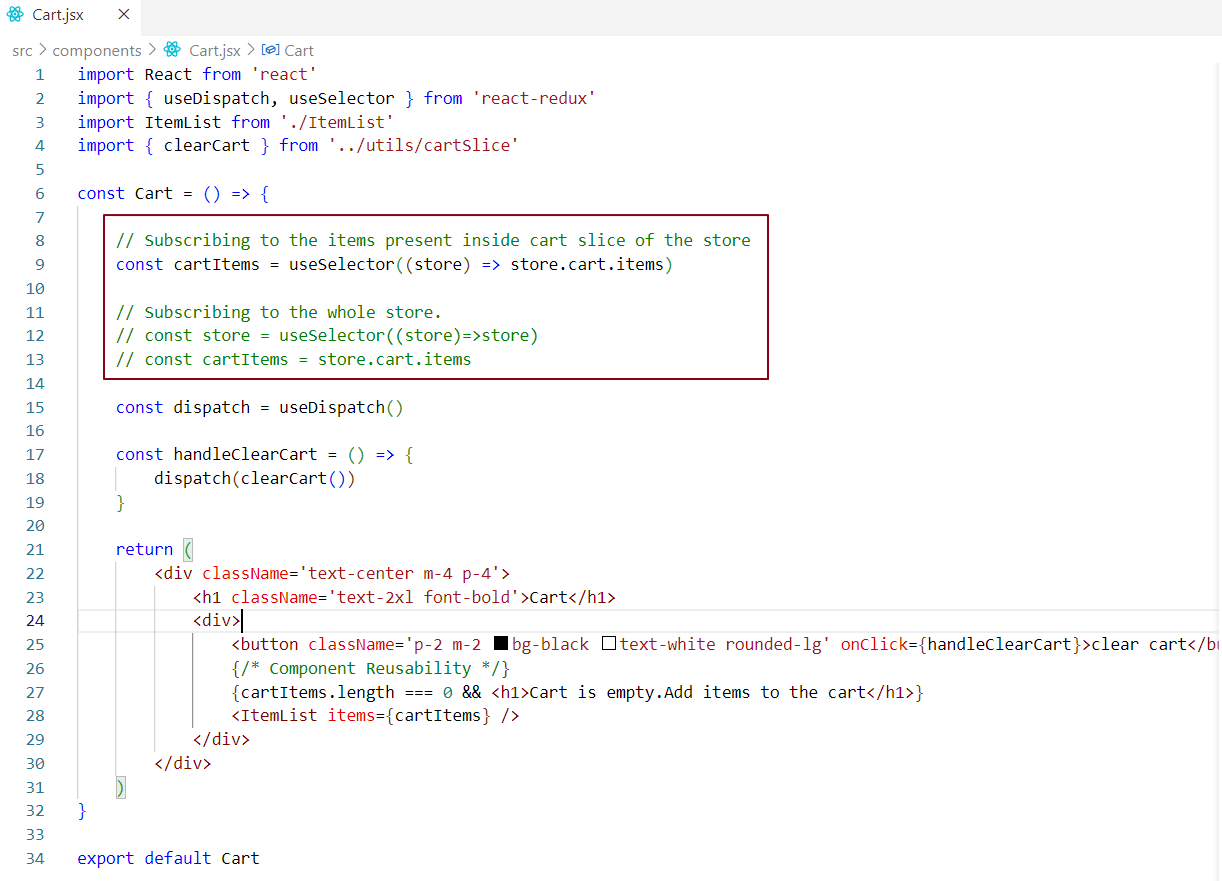


Whenever any part of the store changes, the component re-renders—even if cart.items remains unchanged.

***Subscribing to a Specific Portion (Recommended) -***



The component re-renders only when cart.items changes



Vanilla Redux says not to mutate the state. Earlier, we used to create a copy of the state variable, update it, and then return the updated state from the reducer function.





In Redux Toolkit, we mutate the state directly, and we don’t need to return the state from the reducer function, as was required in the earlier versions of Redux. While it may seem like we are mutating the state, Redux Toolkit actually uses Immer.js behind the scenes to manage immutable state. Immer.js detects the differences between the original and mutated state and returns a new, immutable state to Redux.  
  
***How does Immer.js work behind the scenes?***

➡️ When you write code like state.items.push(action.payload), **it looks like you are directly changing (mutating) the state**, which normally isn’t allowed in Redux

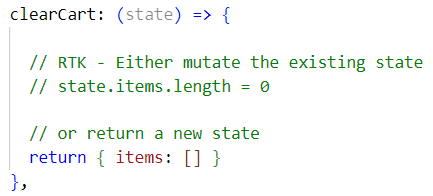
➡️ But **Redux Toolkit uses Immer.js behind the scenes**, which wraps your state in a special object called a **"proxy draft"**.

➡️ **Immer tracks every change** you make on this draft (like adding to items).

➡️ Then, **Immer compares the original state and the draft**, figures out what actually changed (the difference), and creates a **brand-new immutable copy** of the state with those changes applied.

➡️ Finally, Immer returns this **new immutable state** to Redux so that your app state is updated safely without breaking Redux’s immutability rules.

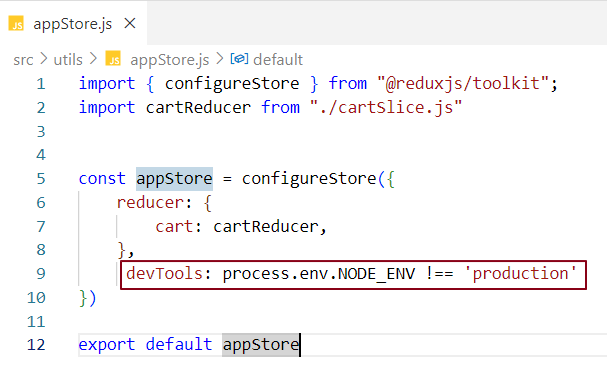
**Note:** RTK allows you to either mutate the existing state or return a new state.



A closer look at redux dev tools

Redux DevTools help us see the full state and the details of every action, which makes debugging much easier.

To use Redux DevTools in non-production environments, we can set the devTools property to true inside the object passed to configureStore.



Redux DevTools is now set up in our codebase, allowing us to view and track the store data easily.

Redux DevTools Features

***Action -*** Displays every action dispatched to the Redux store.

***State -*** Shows the current state of the Redux store at any point in time.

***Diff -*** Highlights what changed in the state as a result of each action.

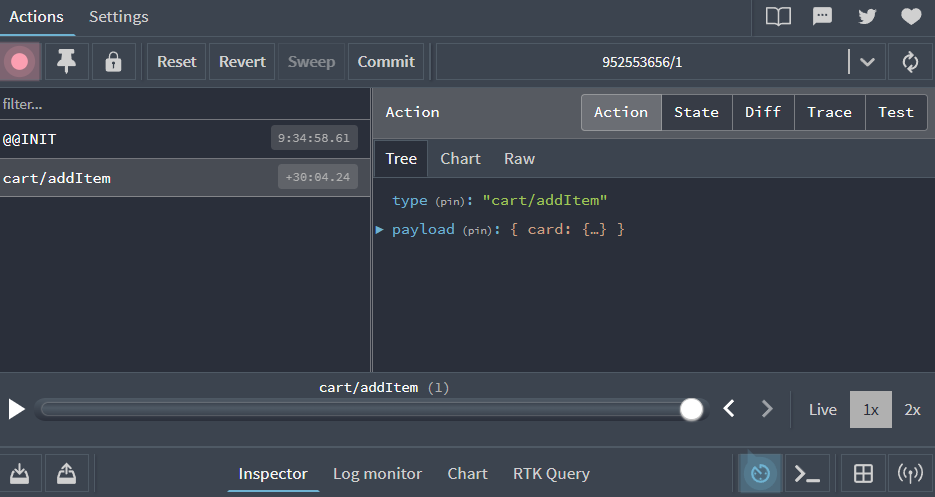
***Trace -*** Helps you see where an action came from and how it moved through the app.

***Replay -*** Lets you replay actions to observe how the state changes, useful for testing and debugging.

Let's explore each feature in detail.

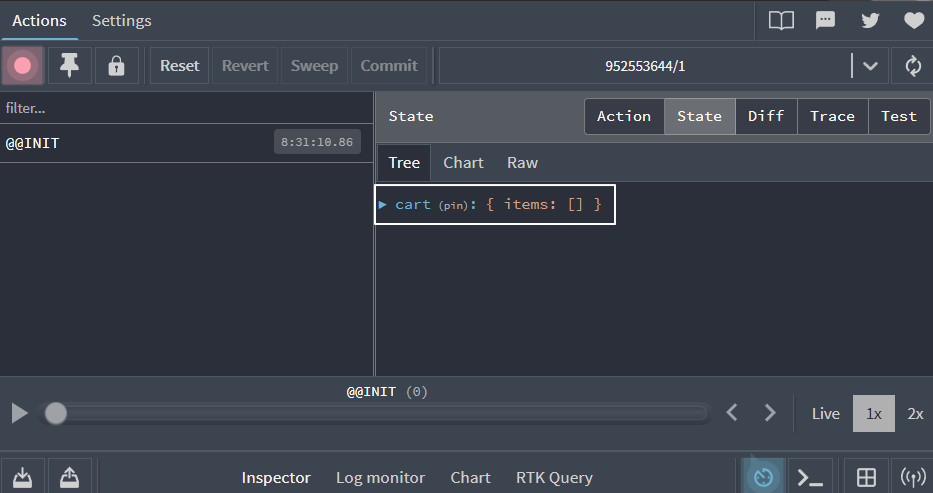
***Action –***

When the Add + button is clicked on a specific food card component, Redux DevTools displays the dispatched action's name along with its payload in the Action tab, allowing us to track the exact data being sent to the store.

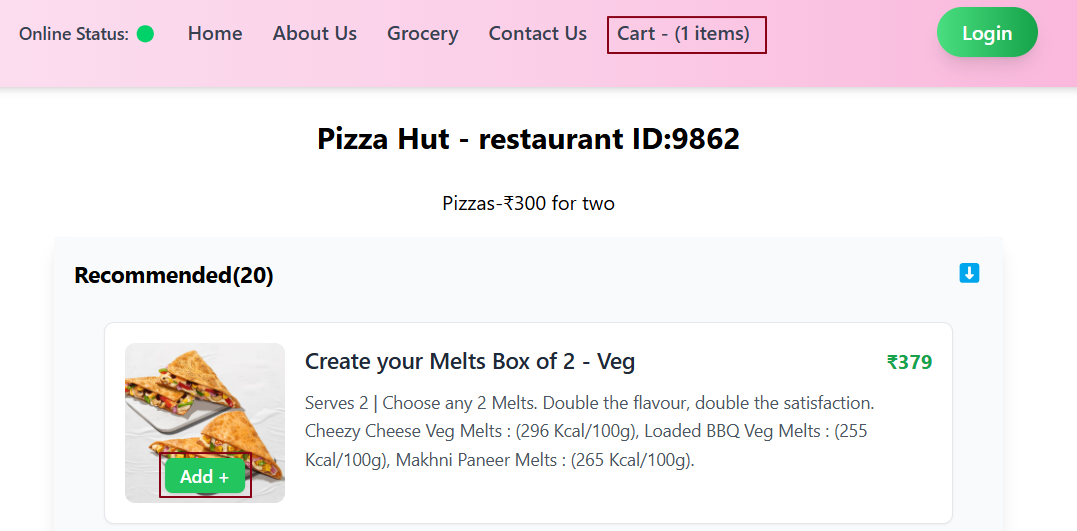


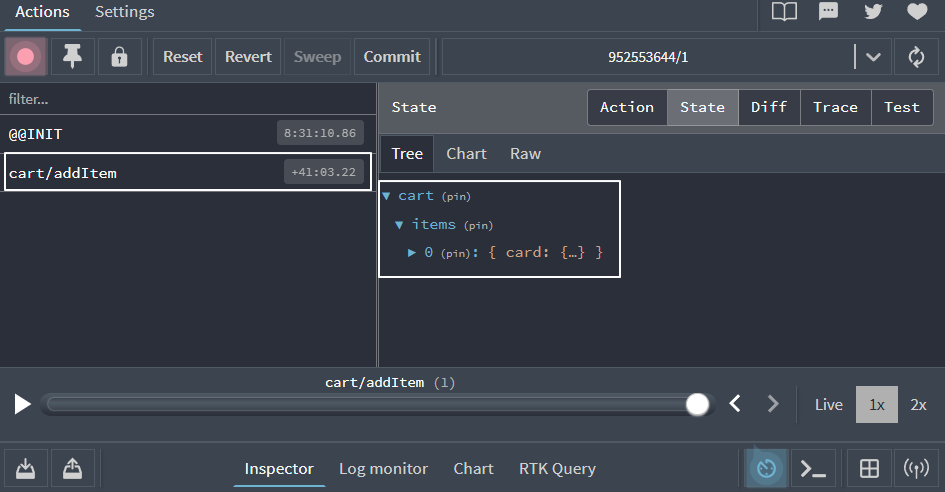
***State -***

This tab shows the current state of the Redux store. At the start, the cart slice has zero items.

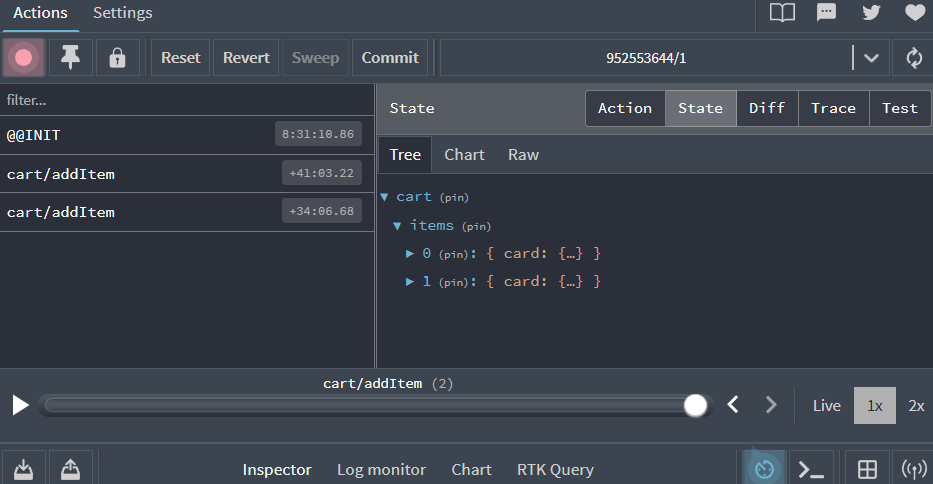


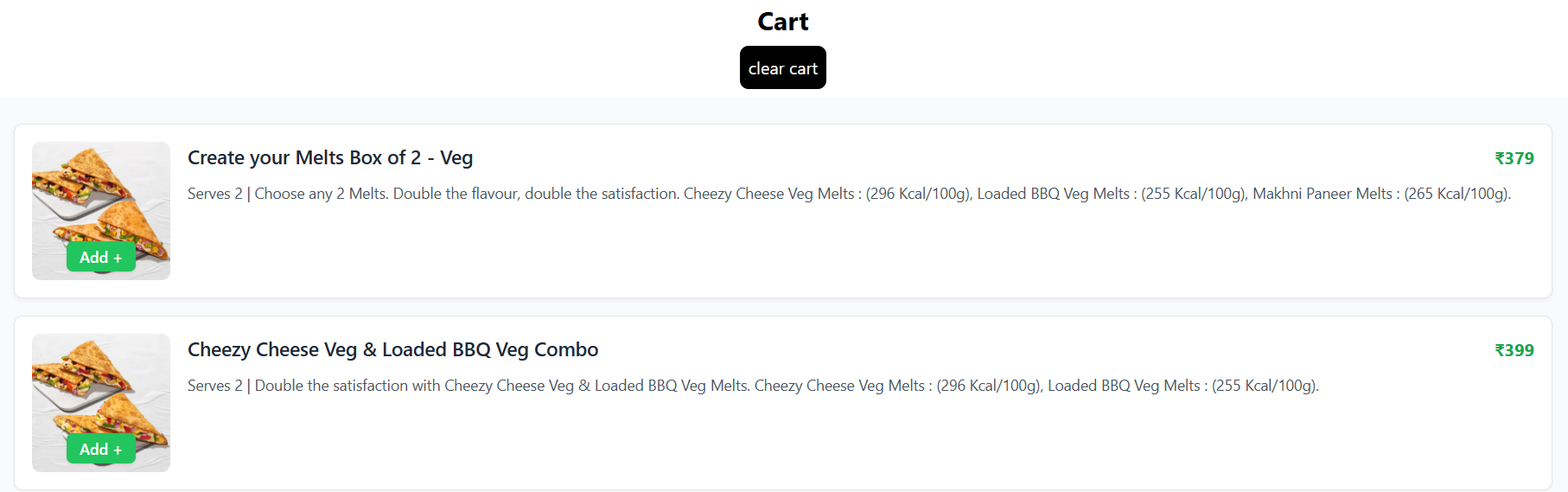
When we click the Add + button and open Redux DevTools, we can see the dispatched action along with the updated state of the cart slice’s items.

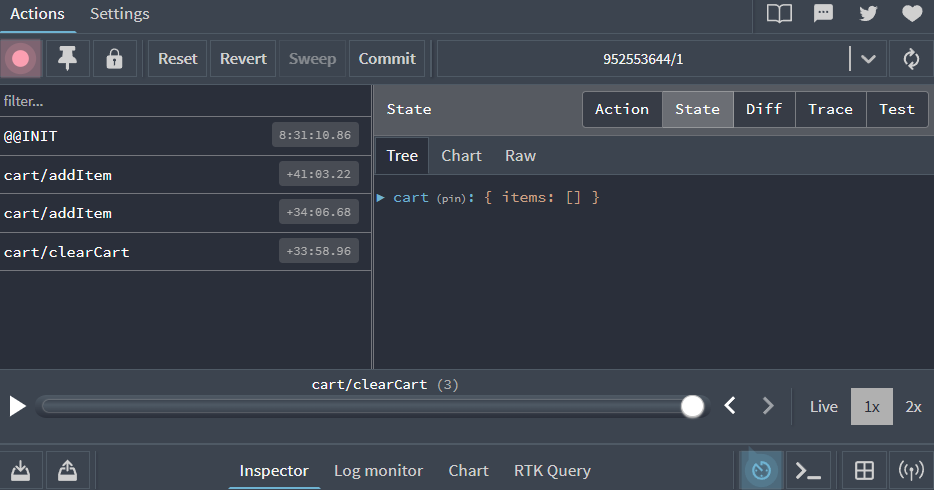




When we click the Add + button for another card item, a new addItem action is dispatched, and the cart slice’s item count updates accordingly.

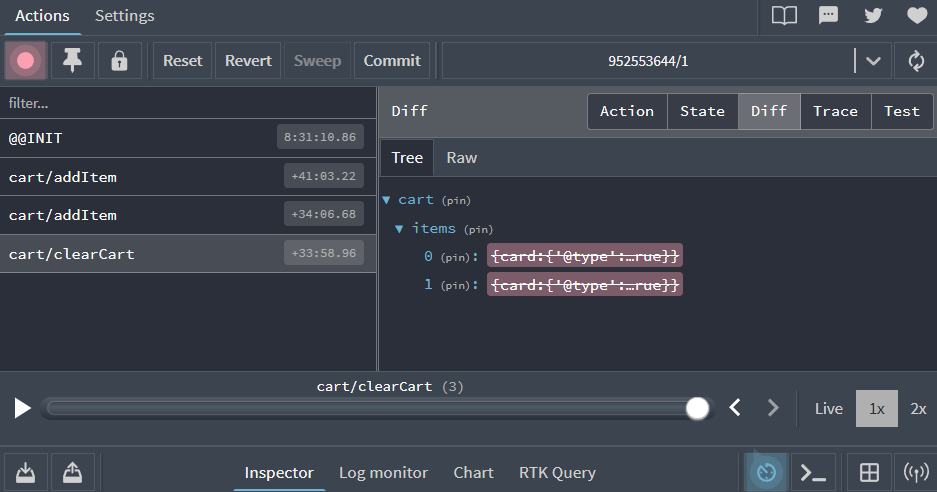


When the Clear Cart button is clicked, the clearCart action is dispatched, and the items state in the cart slice is updated to an empty array ([]).



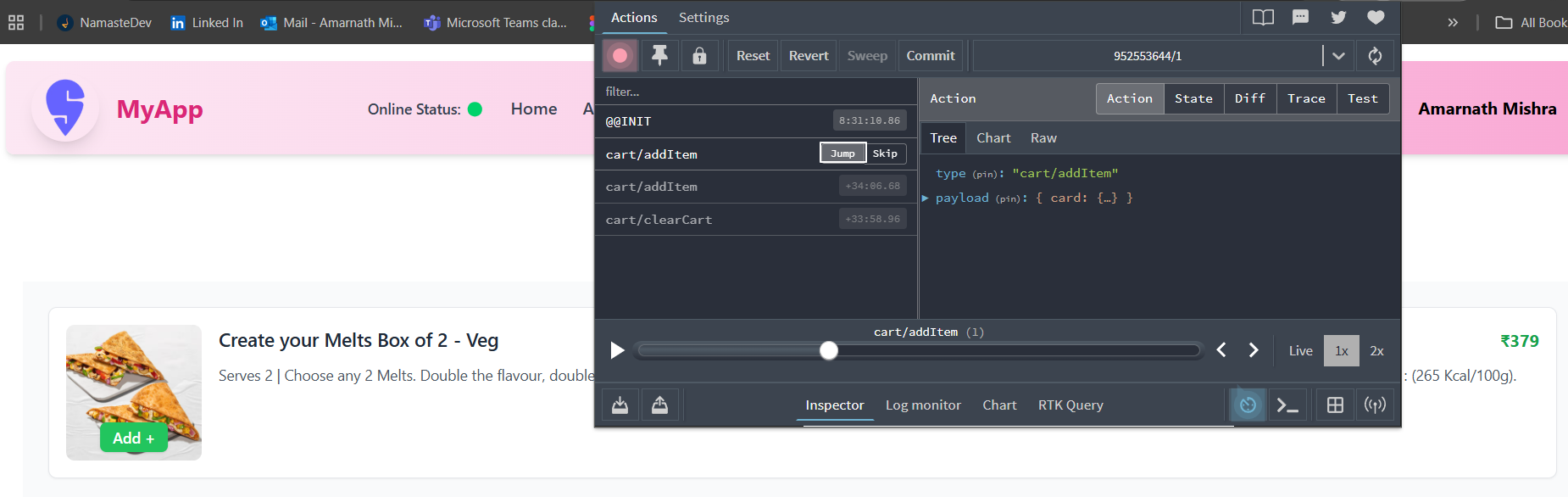
Essentially, we can view a log of every action performed in the Redux DevTools. ***Diff -***

In this tab, we can observe the changes in state between different actions. For example, if the cart initially contains two items and the user clicks the 'Clear Cart' button, the cart items are updated to 0. This change - indicating that two items were removed is visible in the Diff section of the clearCart action within the Redux DevTools

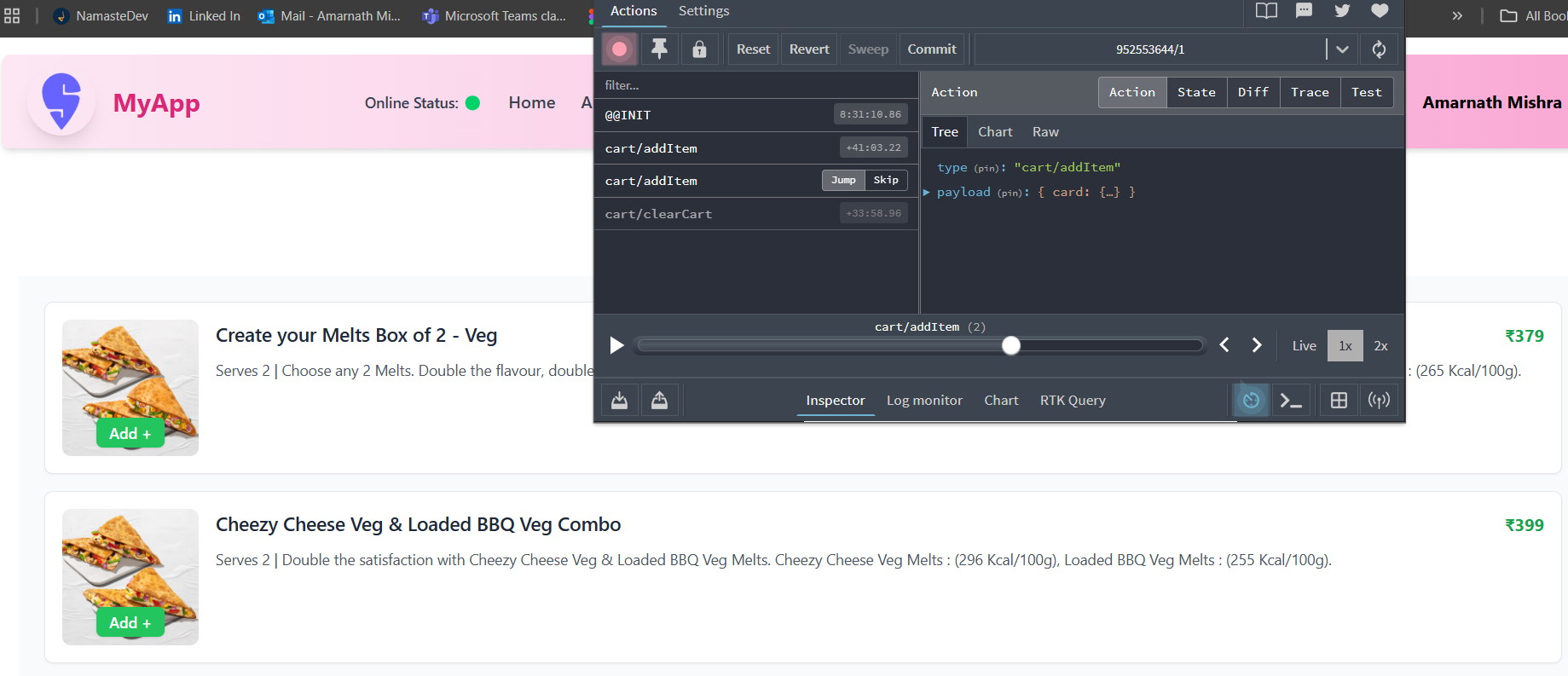


***Note:*** We can replay a sequence of dispatched actions and observe the UI updating step by step as each action is invoked. We can jump to any specific action to see the UI updates caused by that action.

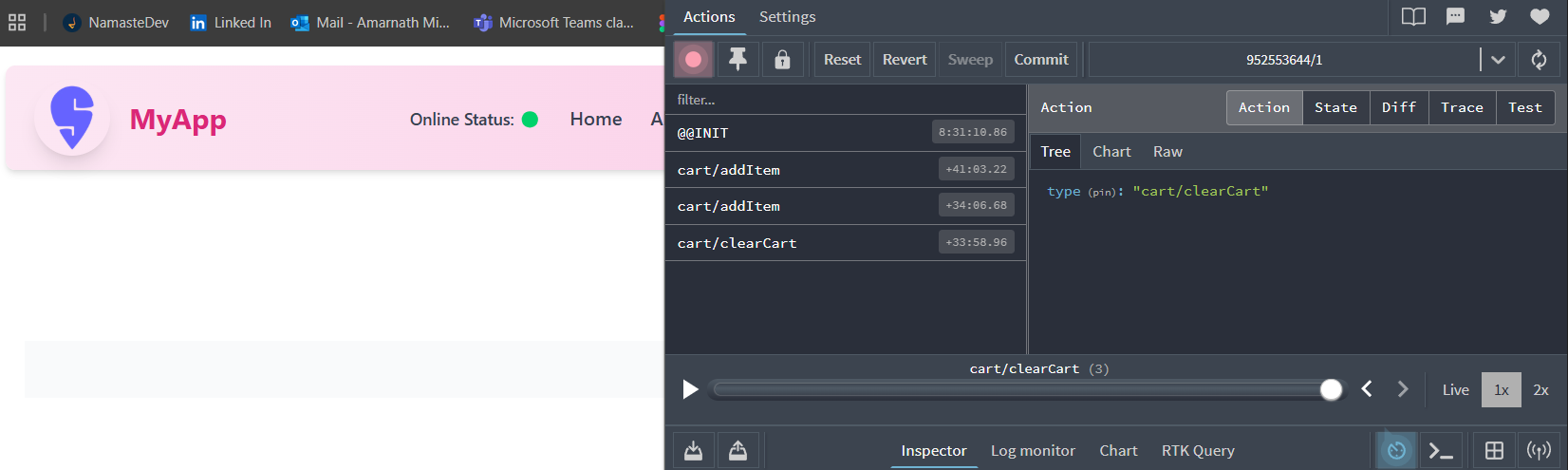
***Jumping to the first addItem action -***



***Jumping to the second addItem action -***



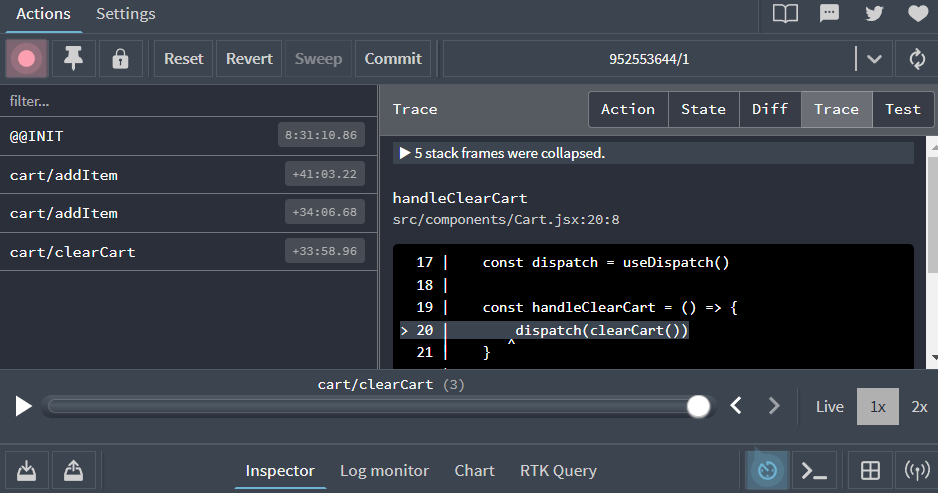
***Jumping to the clearCart action -***



We can skip a specific action and replay the remaining actions to observe the UI updates. This is particularly useful when we suspect that a particular action is causing issues or errors. By skipping that action, we can verify the behavior of the remaining code flow.

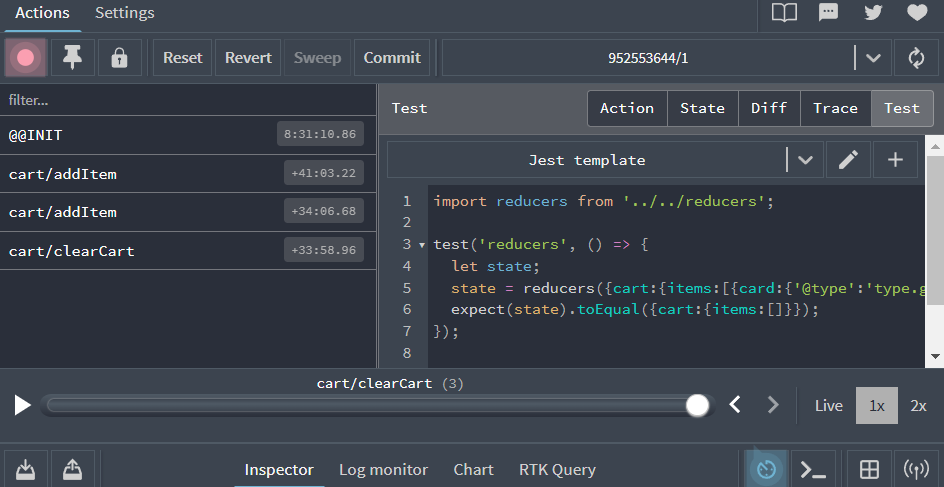
***Trace -***

We can trace a specific action to determine where it is dispatched in the codebase. The trace feature helps developers pinpoint the exact line of code where the action is dispatched.



***Test -***

As developers, we can select a specific action and click on the Test tab to see the test case generated for that action. Essentially, we can view the generated test case for a selected action in the Redux DevTools.



What is RTK Query ?

RTK Query is a data-fetching and caching library built on top of Redux Toolkit. It simplifies the process of handling server-side data in Redux applications by automatically generating the necessary actions and reducers for fetching, caching, and updating data. This reduces boilerplate code and makes working with APIs more efficient within a Redux setup.

### What Are Synthetic Events?

In React, synthetic events are **custom event objects** that are created to handle events in a way that works consistently across all browsers. React doesn't use the native browser events directly. Instead, it creates its own version, called synthetic events, which ensures reliability.

### Why React Uses Synthetic Events ?

***1. Cross-browser consistency -***

React ensures that events behave the same way in all browsers (like Chrome, Firefox, etc.), despite variations in how native events are handled across different browsers.

***2. Simplified event handling -***

React’s synthetic events provide a uniform API, so developers don't need to worry about handling inconsistencies in different browsers.

### *How It Works ?*

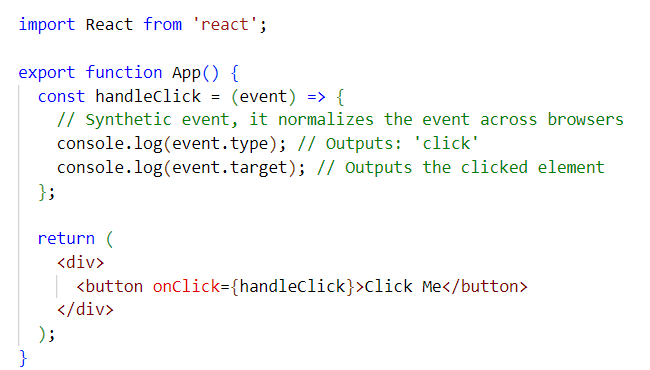
When an event is triggered (like a click), React wraps the native event into a synthetic event object. This object has all the properties you expect (like event.target and event.type), but they work the same way across all browsers.

### *Example -*

When a button is clicked, react ensures uniform behavior across browsers by providing standardized properties in the event object:

* event.target: The element where the event occurred.
* event.type: The type of event (e.g., click, keypress).

Even if browsers have different native event handling, react normalizes the behavior, allowing you to rely on consistent properties



In the example above, the event passed to handleClick is a synthetic event created by React. React ensures that properties like event.target (the element where the event occurred) and event.type (the type of event, such as 'click' or 'keypress') work the same across all browsers, even when the native event behavior may differ between browsers.

***Perks -***

* No Browser Compatibility Issues: React handles the differences, so developers don’t need to write extra code to support each browser’s unique event behavior. It abstracts this code away.
* Cleaner Code: You don't need to manually check for event handling differences between browsers.