Chapter 12 – Redux

Contents

* Redux
* Advantages of Redux
* Disadvantages of Redux
* Redux Toolkit (RTK)
* Redux Store
* Redux Store Slices
* RTK Architecture
* Setting up Redux in a React Application
* How to Create a Redux Store
* How to Provide Redux Store to a React Application
* How to Create Slices
* What are Reducer Functions? How Do They Work?
* How to Add Slices to Redux Store
* How Does a React Application Subscribe to the Redux Store? Importance of the useSelector Hook
* How to Dispatch an Action
* Different Ways to Handle Click Events in React
* Redux DevTools
* RTK Query
* Synthetic Events

What is Redux?

Redux is a state management tool for UI applications, most commonly used with React.js, but can also be used with other libraries and frameworks like Angular and Vue

What are the Advantages of Redux?

Centralized State Management: In React, state/data is stored locally within components. To share this state with other components, props are passed down from parent to child components, and Callbacks are used from child to parent components. In contrast, Redux stores state globally in a central place called the Redux Store, which centralizes all data and makes it accessible across components that need it.

Prevents Prop Drilling: Redux helps avoid prop drilling, similar to the React Context API.

Better for Large Applications: Redux is generally preferred over the React Context API for large-scale applications due to its efficiency in managing complex state across many components.

Performance Optimizations: In React, when a state or prop changes, the components using that state or prop are rerendered. With Redux, when data in the store changes, a shallow copy of that data is created, which can minimize unnecessary rerenders and improve performance.

What are the Disadvantages of Redux?

Steep Learning Curve: Redux can be difficult to learn, especially for newcomers.

Not Suitable for Small Applications: Redux may be overkill for small applications that don't require complex state management.

Complex Configuration: Setting up the Redux store can be complex and time-consuming.

Boilerplate Code: Redux requires a lot of boilerplate code, which can make the codebase messy and harder to read.

Difficult Debugging: Debugging can be challenging due to the complexity of Redux’s flow and actions.

What is Redux Tool Kit (RTK)?

RTK (Redux Toolkit) allows us to write Redux code in a more concise and efficient way. It abstracts the boilerplate code typically required in Redux, enabling developers to write clean and maintainable code with less effort.

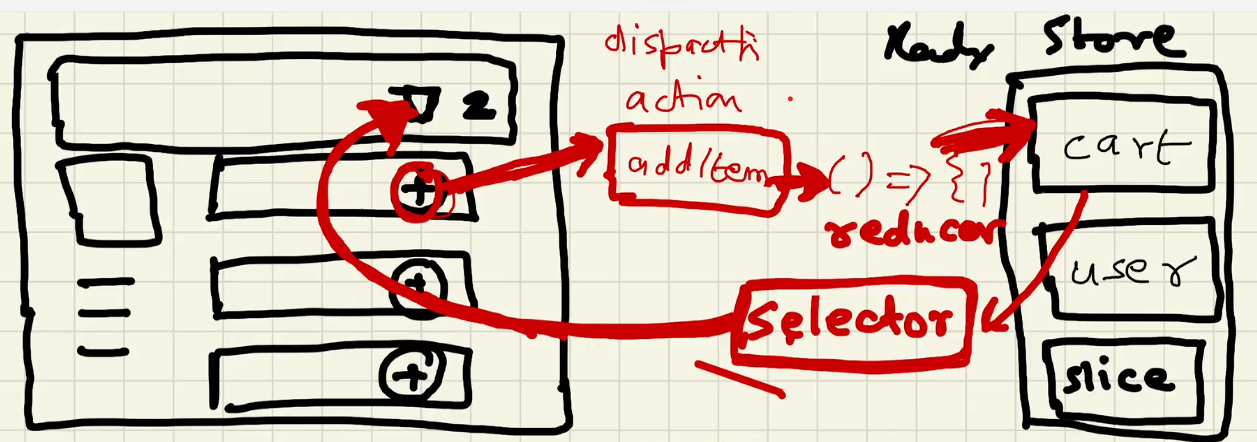
What is Redux Store?

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 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 this 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.

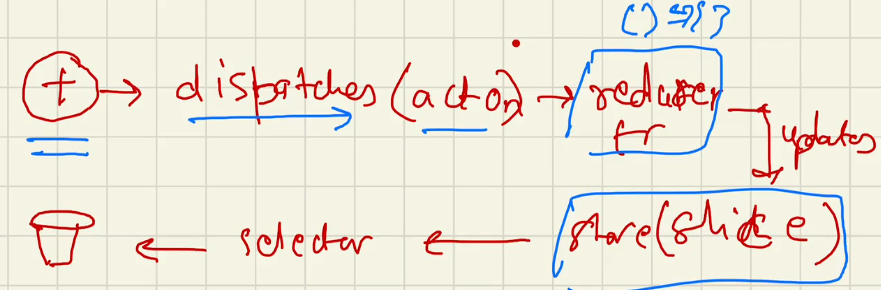
Explain Redux Toolkit Architecture?



In an E-commerce application, the process typically involves adding items to a 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 calls a reducer function that updates the cart slice of the store.

To read the data from the cart slice and update the cart item count in the UI, we use a selector. Essentially, we subscribe to the store through the selector, meaning that the UI will automatically be notified and updated whenever the cart slice data changes. The useSelector hook is used to achieve this. The term selector is used because it refers to selecting a specific portion (or slice) of the 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 ensure consistency and organization, state changes should not happen arbitrarily. Instead, specific functions and actions must be used to update the state. This ensures that Redux can track the changes properly.

**Note:** Whenever a component uses a selector, it means that 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 updated immediately when the relevant slice of the store changes. This is the concept of 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 serves as a bridge between the React app 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.

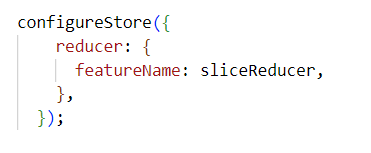


**Empty Configuration**:

Passing an empty object {} to the configureStore function means that no slices (reducers) are defined in the store. The store will still be created successfully, even if no reducers are provided. However, it won’t manage any state or actions. Without slices or reducers, the store cannot -

* Store any state.
* Dispatch actions to update state.
* Allow components to select state.

To make the store functional, you need to define slices (reducers) and pass them under the reducer key in the configureStore argument, like this:



For practical applications, always include at least one reducer to manage state.

Leaving it empty is okay during the initial setup or testing phase. Here, we are performing the initial setup.

How do I provide 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 bridge, we wrap the root component of the application with the Provider component. We then pass the store as a prop to the Provider component to make 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 have wrapped 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 portion with the Provider component, ensuring that the store is available only to that part of the app

How to create Slices inside Redux Store?

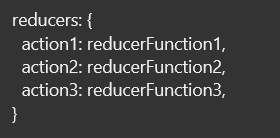
RTK provides an API called createSlice that allows us to create slices. In the example below, we are creating a cart slice.



name represents the name of the slice, which is cart in this case.

initialState represents an object where the initial state of the slice is defined. Here, we set items to an empty array because, initially, the cart will have no items.

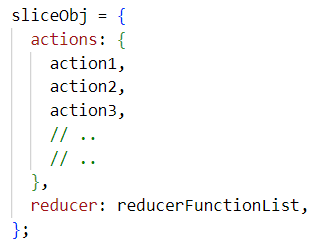
reducers is an object where actions are mapped to their corresponding reducer functions.



Note: A reducer function modifies the data within the slice (a slice is part of the store).

* Each reducer function has a state parameter that provides access to the slice's state object. Initially, this state is {items: []}, meaning it starts with {items: []} as its default values.
* Inside the reducer function, the state is updated based on the action logic and specific requirements.
* The createSlice function returns a slice object that contains both the actions and the reducer.

Internal representation of slice object by redux developers -



* cartSlice.reducer (line 23) refers to a single reducer object that combines all the individual reducer functions defined in the slice. This consolidated reducer can then be used in the store configuration.
* On line 21, we destructure all the actions from cartSlice.actions and export them together, allowing them to be easily used throughout the 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, such as 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.

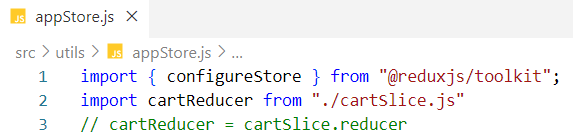
How do we add Slices to Redux Store?

In Redux Toolkit, we use the configureStore method to create a Redux store and manage slices. Slices are added to the store by mapping their reducers inside the reducer property.

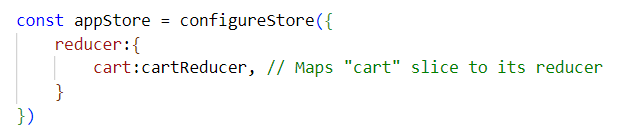
Steps to Add Slices to the Redux Store -

1.Import configureStore and Slice Reducers.

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



2.Map Slice Reducers.

* The configureStore method takes an object as an argument.
* This object includes a reducer property, which is another object.
* Inside the reducer property, each slice is added as a key-value pair.  
  Key: The slice name (used as the state key in the store).  
  Value: The slice's reducer function.
* In this example, the cart slice is added with its reducer (cartReducer), making it part of the store's state.

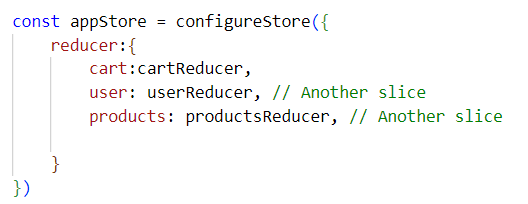
3.Export the Store.

* Export the configured store so it can be used in the application.

Key Notes -

* The reducer property in configureStore acts as the "big reducer" for the entire application.
* Each slice reducer is a "small reducer" managing its respective part of the state.
* Adding more slices is as simple as adding more key-value pairs inside the reducer object.

Example with 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 Redux store, allowing it to read specific slices of the state and automatically re-render whenever that slice of the state changes.

**Purpose**: The useSelector hook is used to access the Redux store's state within a React component.

**How it Works**:

* The hook takes a **selector function** as an argument.
* The selector function is used to retrieve a specific part (or slice) of the state from the Redux store.

**Automatic Updates**: When the slice of state accessed by the selector function changes, the component using useSelector automatically re-renders with the updated data.

**Integration in React Applications**:

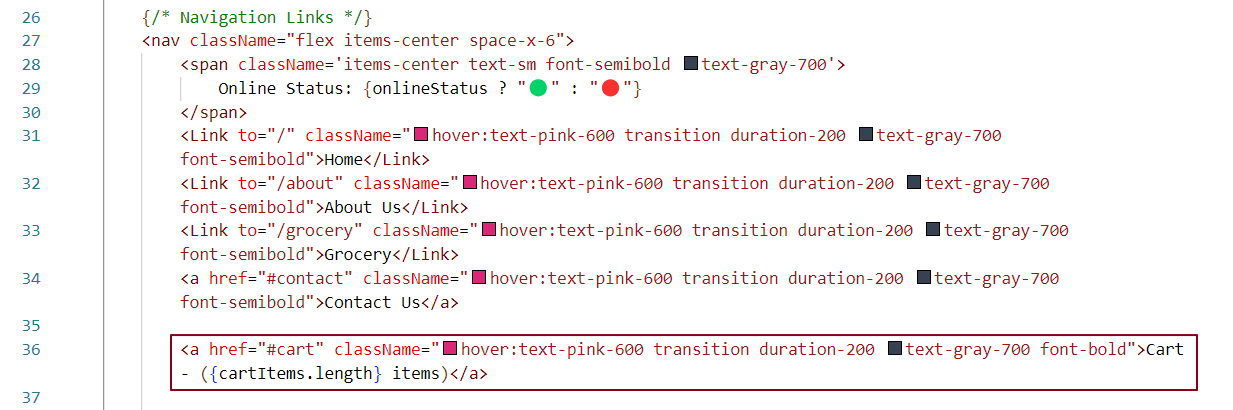
* React applications subscribe to the Redux store using the useSelector hook.
* This hook acts as a bridge between the Redux store and React, ensuring seamless state access and updates.

**Accessing the Store**: By using useSelector, we gain access to the Redux store and can retrieve data as needed for rendering.

**Why the name is called useSelector?**

The name useSelector makes sense because it helps you select a specific part of the Redux store. Instead of subscribing to the whole store, useSelector lets your component get only the part of the state it needs.





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

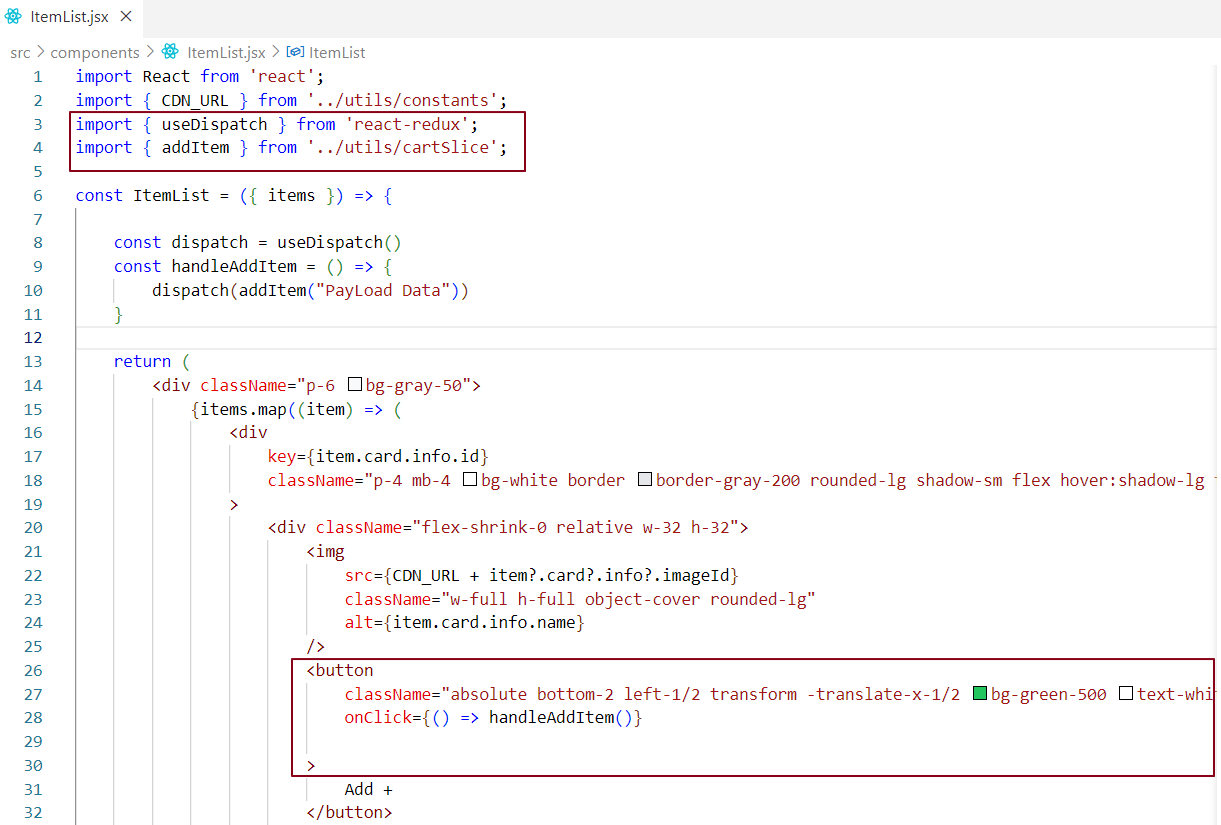
At line 36, we use the subscribed data to display it on the UI. Currently, the cart slice in the Redux store contains zero items, so we expect 0 to be displayed on the UI.

The useSelector hook provides access to the entire Redux store, and we use it to subscribe to a specific portion 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 into redux store?

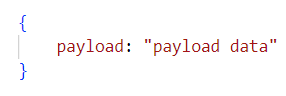
The react-redux library provides a hook called useDispatch, which returns a function. This function accepts an action along with its payload and is responsible for dispatching the action to the Redux store.



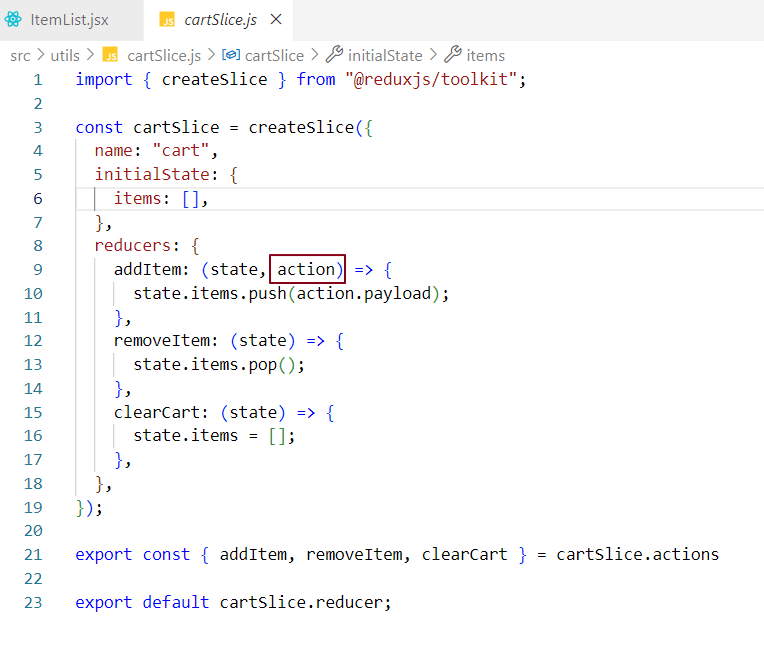
In line 4, we import the addItem action, which is exported by the cartSlice.  
In line 14, we use the useDispatch hook, which returns a function named dispatch.  
In line 27, the handleAddItem function is triggered when the "Add +" button is clicked.  
Within handleAddItem, the dispatch function is called, dispatching the addItem action along with the PayloadData.

The PayloadData is passed to the addItem action, which in turn passes it to the corresponding reducer function. This reducer function 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

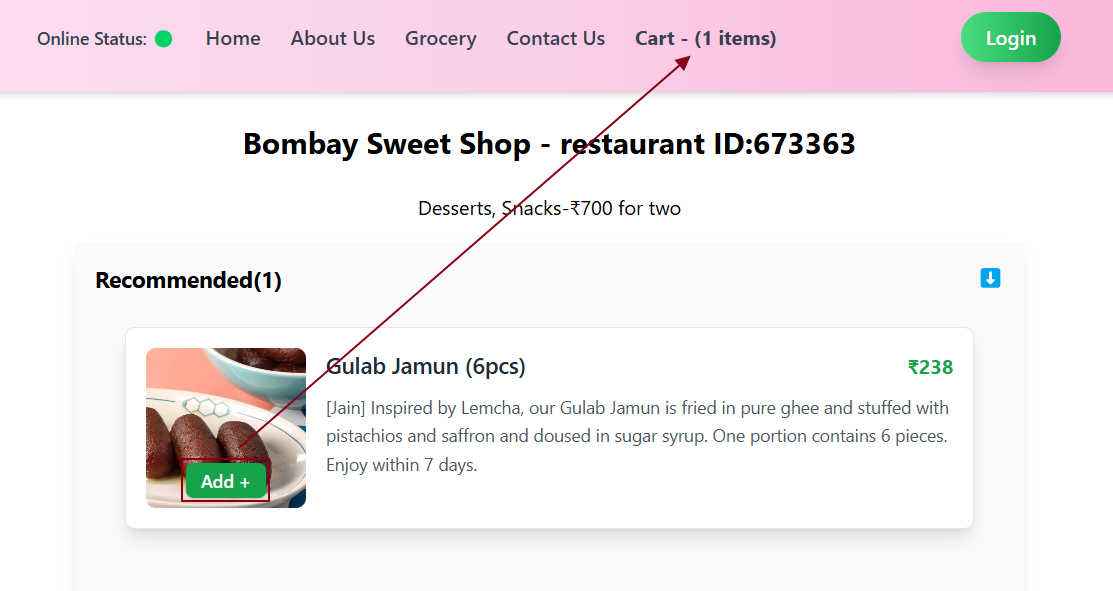


This action object is then passed as the second argument to the reducer function corresponding to the dispatched action.



The addItem reducer function receives this action object and processes it accordingly, updating the state. Since the application is subscribed to the cartSlice, any updates to the items in the store automatically reflect in the UI.

In this case, when the "Add +" button is clicked, the cart slice updates its items with the PayloadData provided by the UI, and the items count in the UI is updated. Meanwhile, the useSelector hook reads the updated value from the store and displays it in the items count section of the UI.

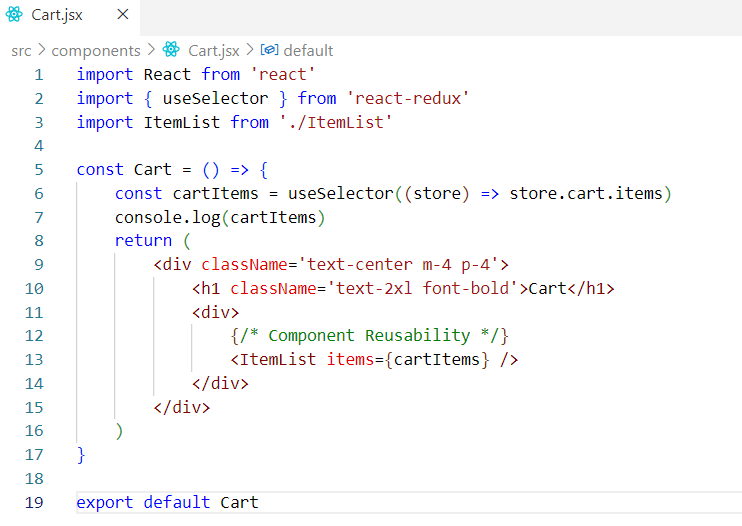


So far, we have been passing hardcoded data to the addItem action, which is dispatched when the "Add+" button is clicked. Let’s pass the card-specific data when the "Add+" button of a particular card is clicked.

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 card count displayed in the UI.

When navigating to the card count section, users should see the list of food cards that have been added to the cart, allowing them to proceed with checkout.

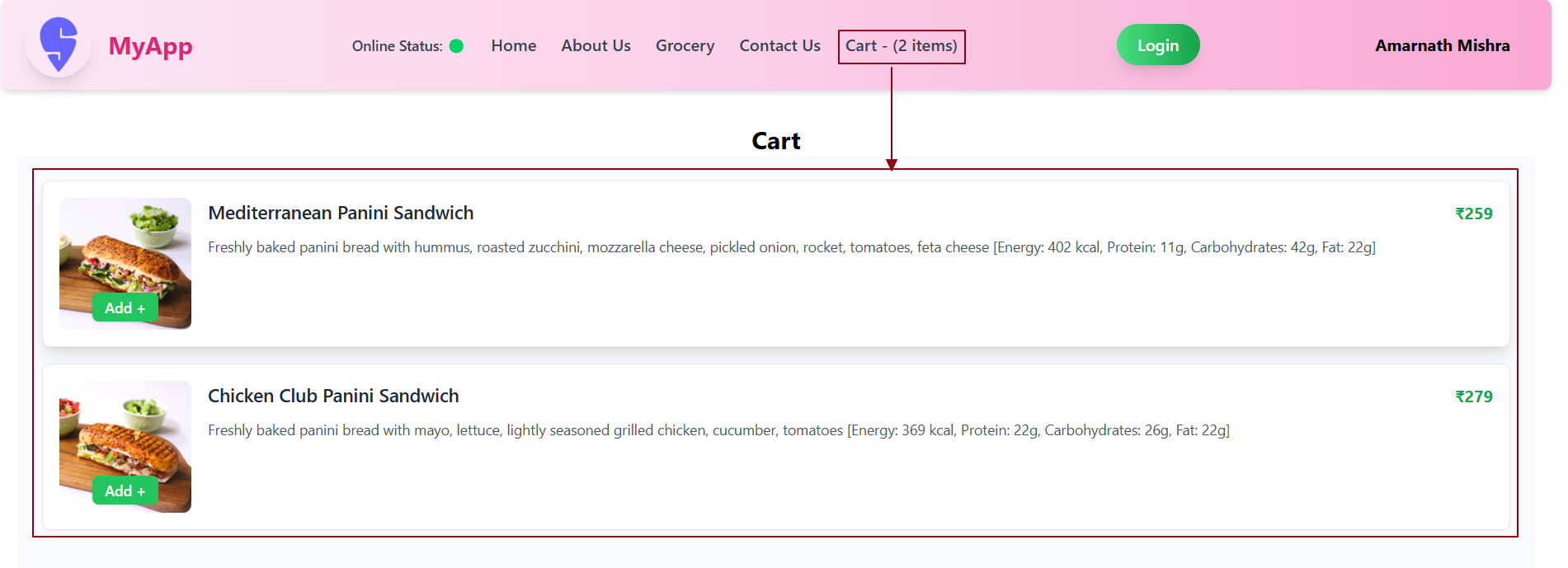
In summary, let’s create a component Cart.jsx that displays the list of food cards added to the cart, enabling users to review their selections and proceed to checkout.



In this component,

The cartItems state will hold the list of cards for which the "Add+" button has been clicked.

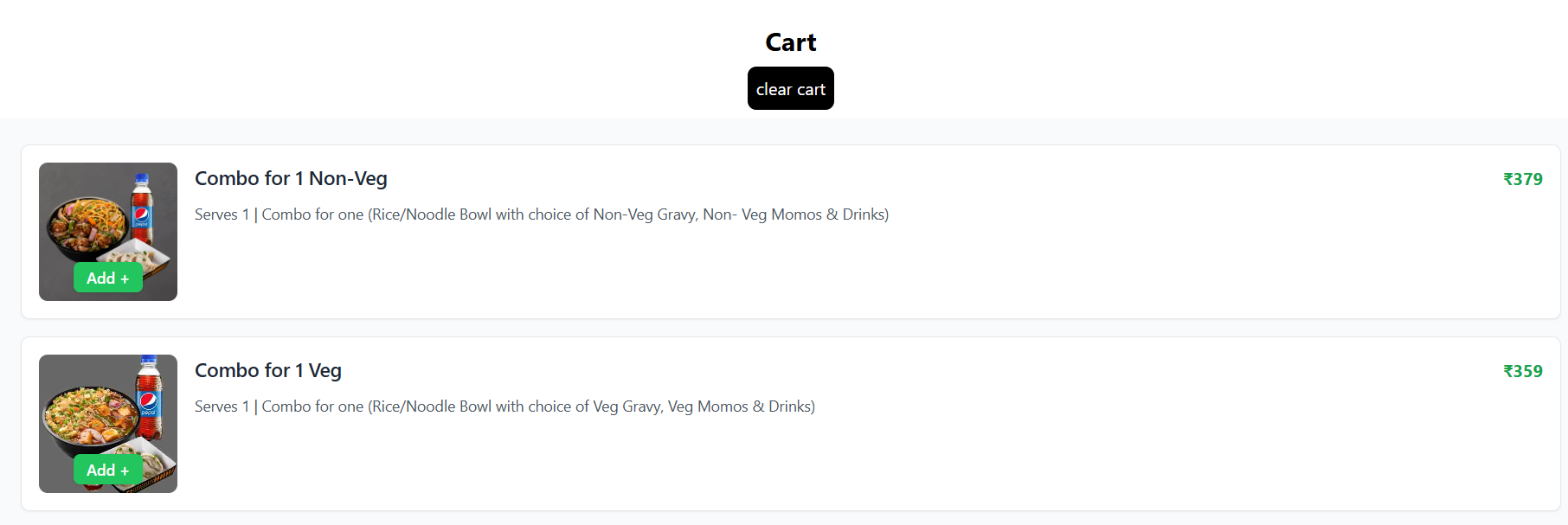
We are reusing the ItemList component to display the list of selected card components on the Cart page.



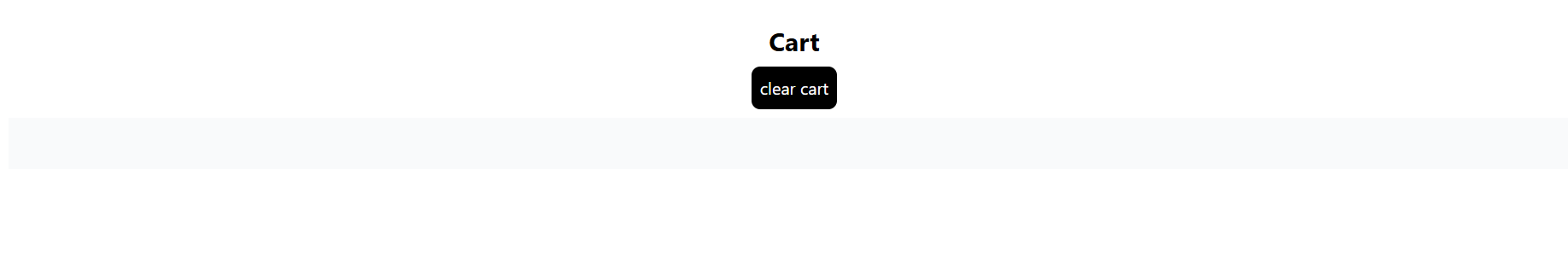
Let’s add a "Clear Cart" functionality to our Cart component.



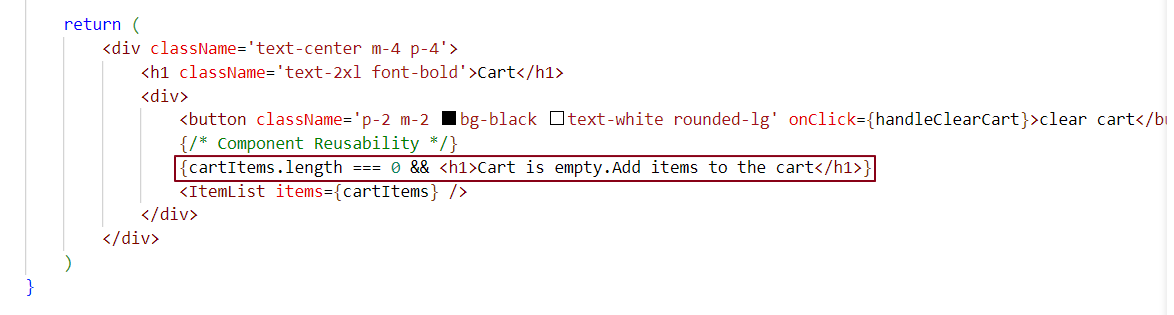
When the "Clear Cart" button is clicked, the handleClearCart function is invoked. This function dispatches the clearCart action, which triggers the associated reducer function to remove all cartItems from the items state in the cart slice.

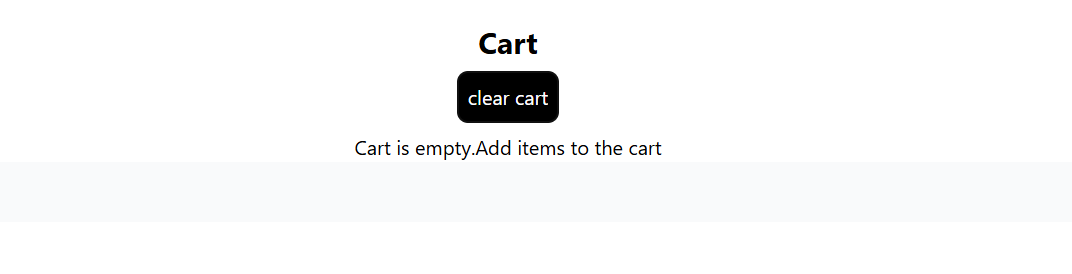


When the "Clear Cart" button is clicked, all the food card components will be removed from the UI.



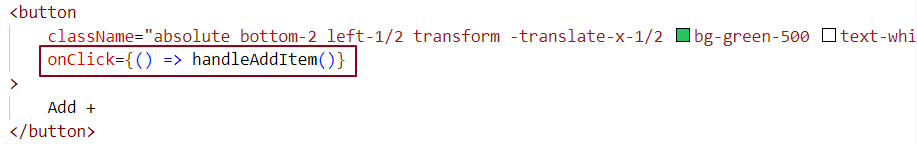
If the cart is empty, I want to display a message in the UI: **"Cart is empty. Add items to the cart"**





Different Ways to Handle Click Events in React -

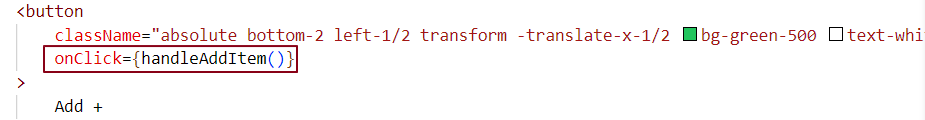
**Case 1 - Using an Arrow Function (Callback) –**



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

Why use this? This approach ensures that handleAddItem () is only called when the click event occurs, not during the render. It allows for more flexibility, such as passing parameters or executing additional logic inside the function.

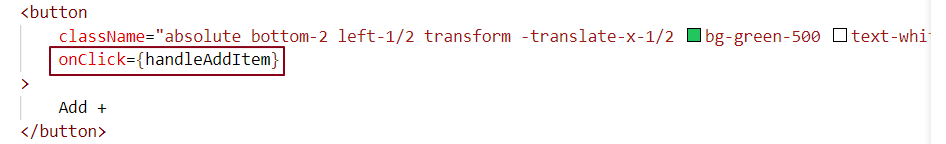
**Case 2 - Calling the Function Immediately (Incorrect Approach)**



What happens here? handleAddItem () is immediately invoked during render, not when the button is clicked. This will execute the function during the component's render cycle and assign its result to onClick, instead of passing the function to be executed later.

Why this is wrong: The function is not waiting for the click event. It runs on every render, which is not the desired behavior for handling events.

**Case 3 - Passing a Function Reference**



When you pass a function reference without parentheses (onClick={handleAddItem}), React will automatically call that function when the event is triggered (in this case, the click event).

This approach does not call the function immediately during render (like in case 2), but instead, it passes a reference to the function. React knows to execute it when the event happens.

**Note -**

Always use **onClick= {() => handleAddItem ()}** for most event handling scenarios in React. This ensures the function is executed **only when the event occurs** and allows you to handle the logic cleanly.

Best Practices

In Redux, it’s better to subscribe to a specific part of the store rather than the entire store to improve performance and reduce re-renders.

**Improved Performance**:

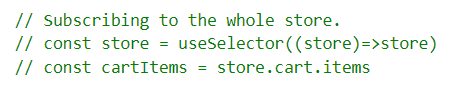
Subscribing to the entire store might lead to unnecessary computations, especially in large applications with a complex state structure. By narrowing down the subscription, react avoids processing irrelevant state updates, resulting in better app performance.

**Reduced Re-Renders**:

When you subscribe to the entire store, any change in the store—whether relevant to your component or not—triggers a re-render. By subscribing only to the part of the store that the component needs, re-renders occur only when the relevant part of the state changes.

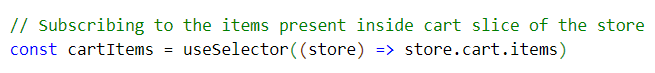
Example -

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

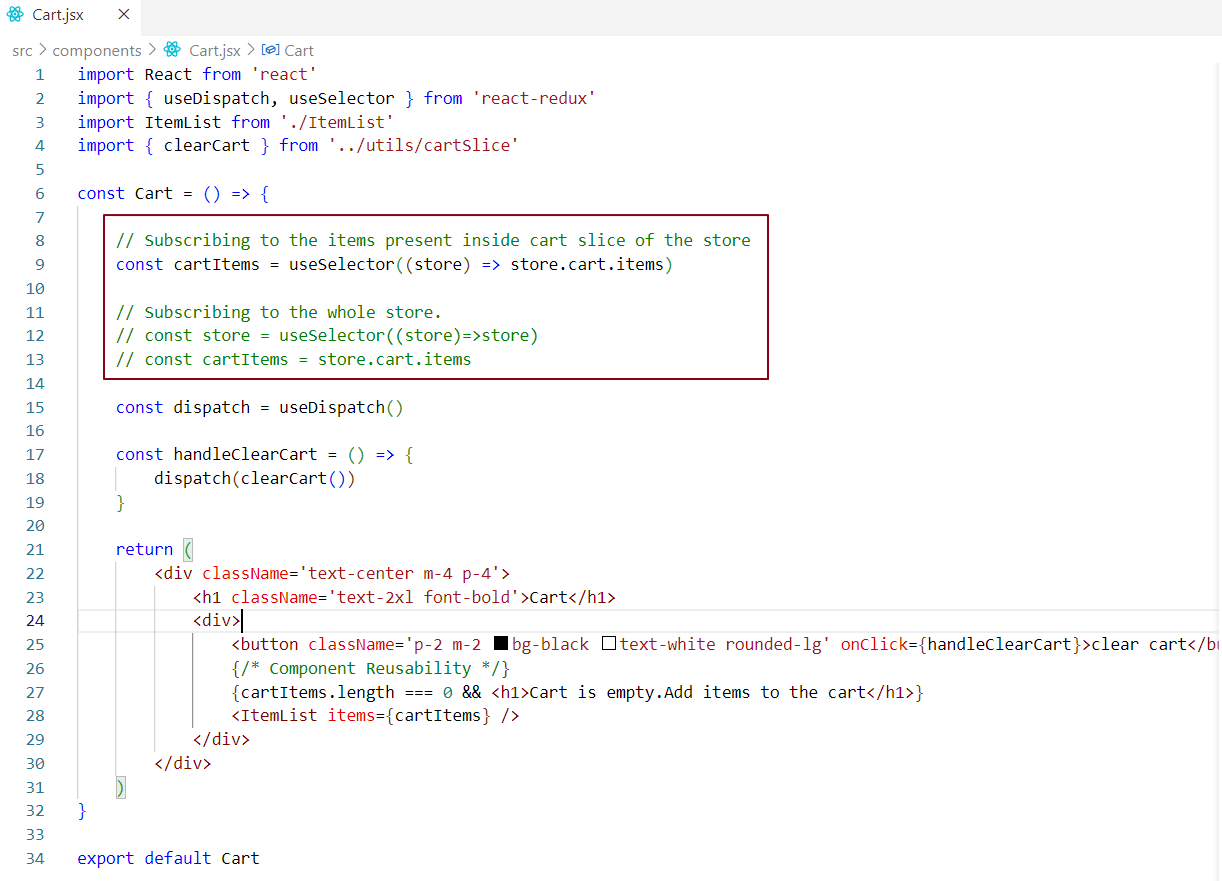


Whenever any part of the store changes, the component re-renders, even if cart.items remains unchanged.

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



The component re-renders only when cart.items changes.



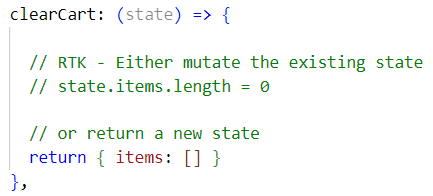
Vanilla redux says don’t mutate state. Earlier, we used to create a copy of the state variable, update it, and then return the updated state from the reducer function.





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 it returns the new, immutable state to Redux.

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

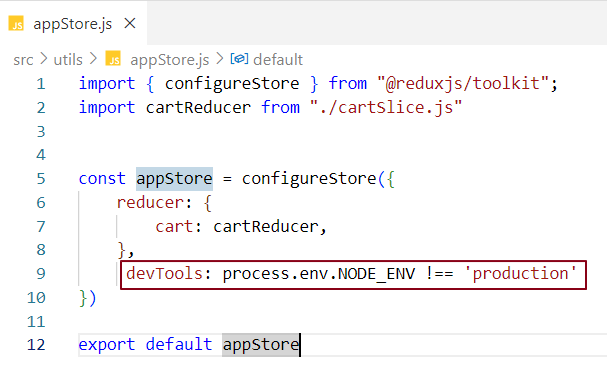


Redux DevTools in Action -

The Redux DevTools allow us to inspect the entire state and the payload of every action.

Redux Dev Tools video guide: <https://www.youtube.com/watch?v=BYpuigD01Ew>

To enable Redux DevTools in non-production environments, we need to add the devTools property inside the object passed to configureStore.



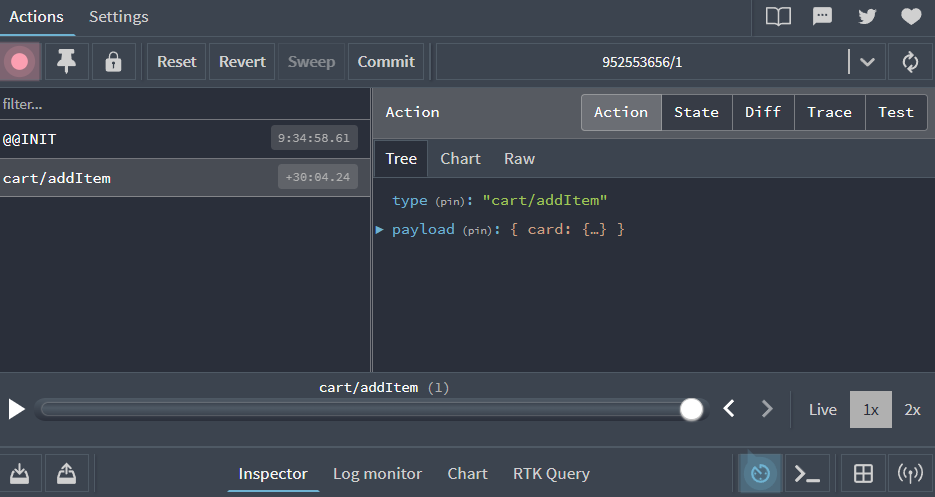
Now we have configured the Redux DevTools extension with our codebase to read and maintain the store data from the Redux store

**Redux DevTools Features:**

* **Action** – Displays all actions that are dispatched to the store.
* **State** – Shows the current state of the Redux store.
* **Diff** – Highlights the changes in state between different actions.
* **Trace** – Helps trace the flow of an action through the application.
* **Replay** – Allows you to replay actions to observe the state changes and test the store.

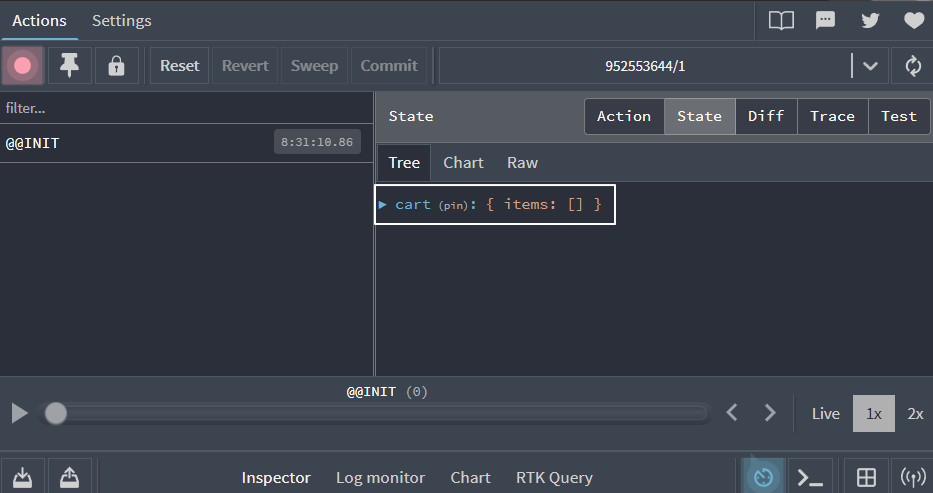
Let's explore each feature in detail.

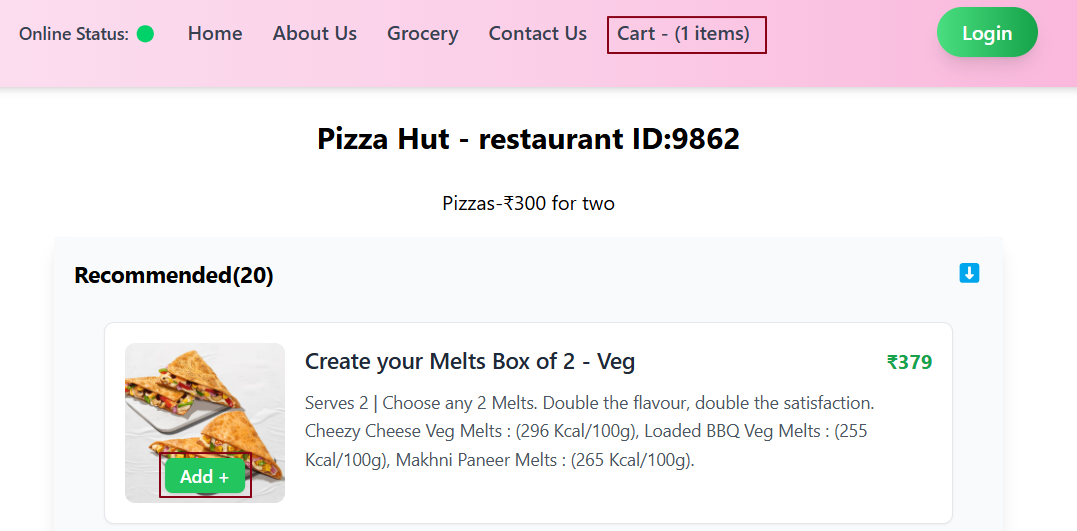
**Action -**  
We can see the name of the action being dispatched. When we click the "Add +" button of a specific food card component, we can see the action name being dispatched along with the payload being passed in the Redux DevTools Action tab.



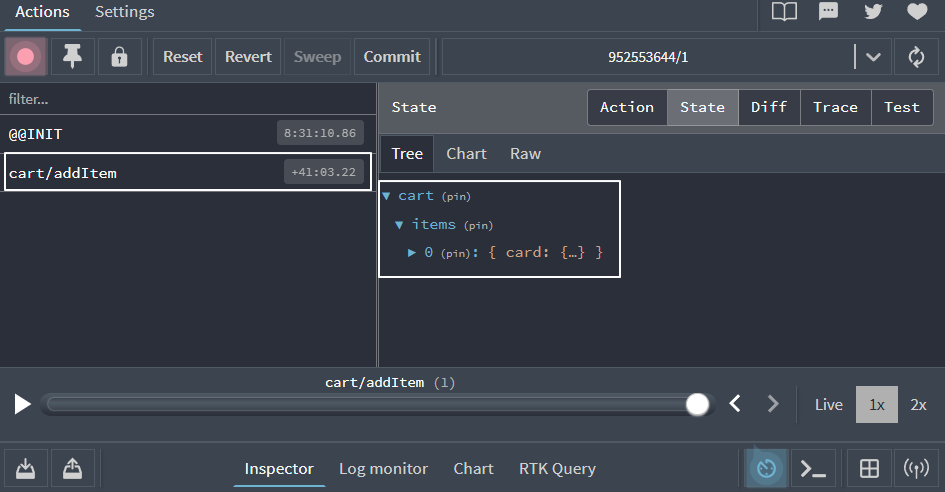
**State –**

In this tab, we can see the current state of the Redux store. On the initial load, the cart slice will contain 0 items.

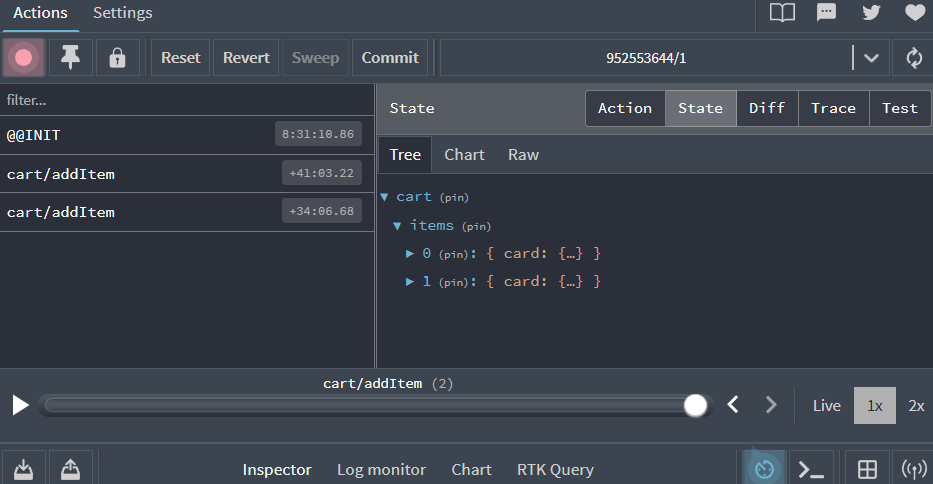


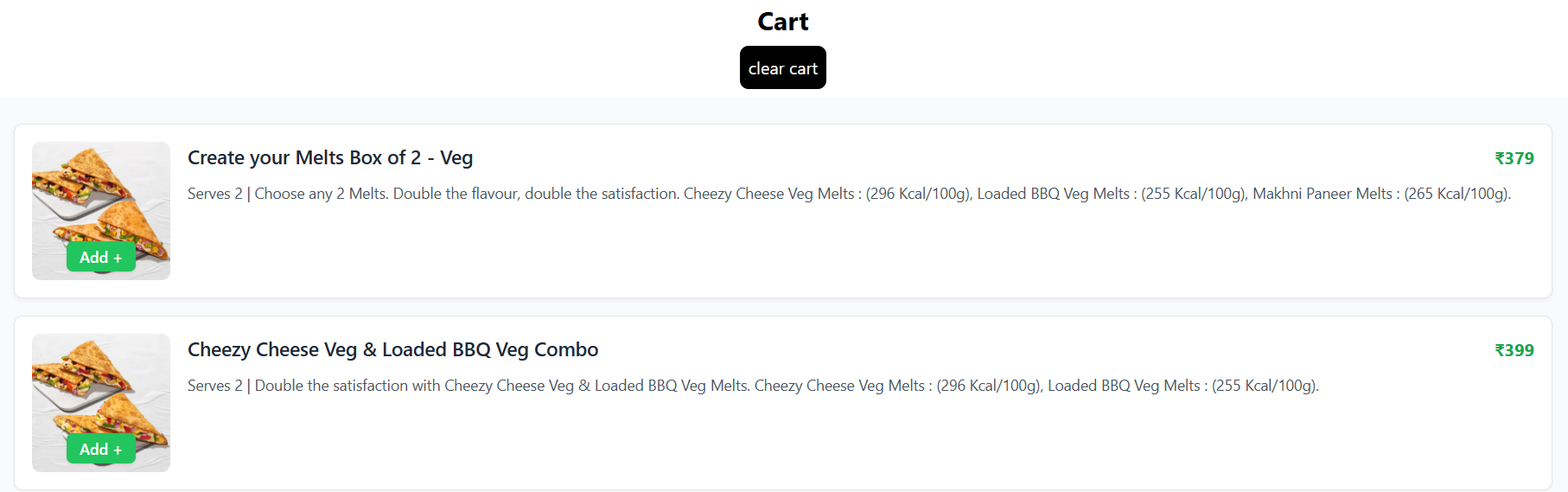


When we click the "Add +" button and open the Redux DevTools, I can see the action that was dispatched and the current state of the cart slice's items.

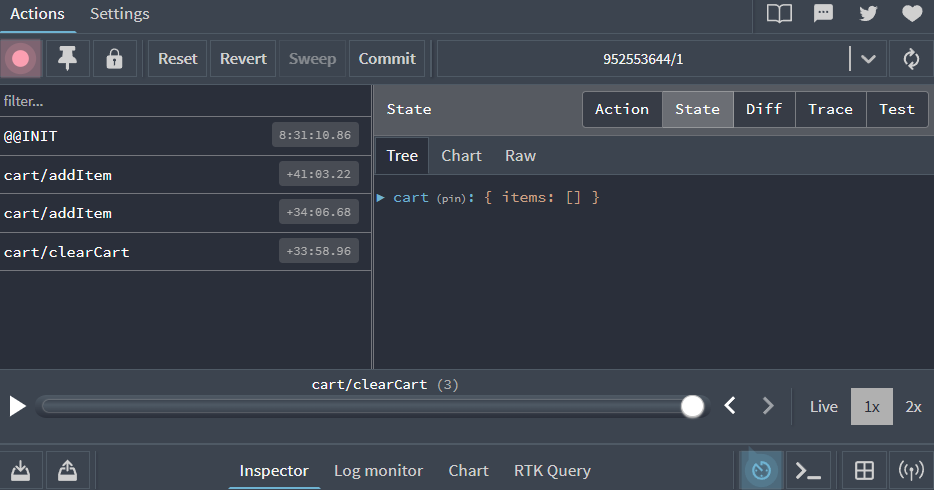


When we click the "Add +" button for another card item, a new addItem action is dispatched, and the cart slice's item count is updated accordingly.



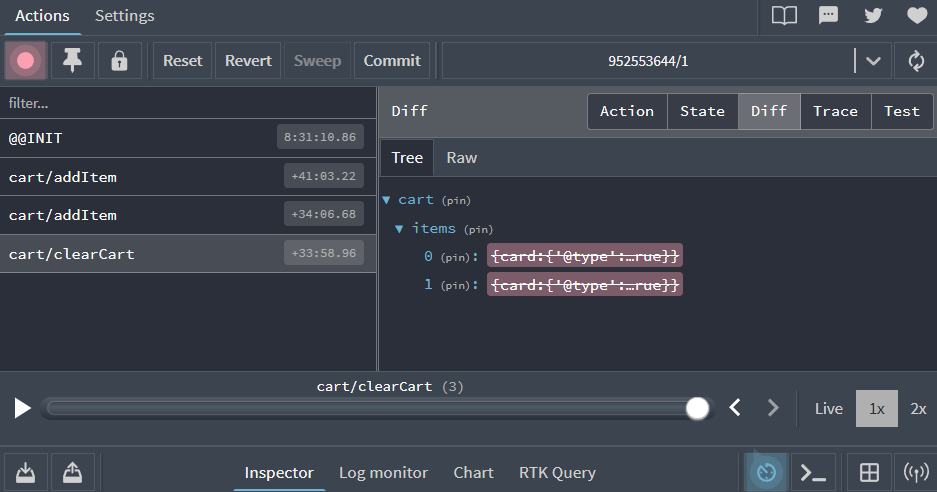


If we click the "Clear Cart" button, a clearCart action is dispatched, and the items state of the cart slice is updated to an empty array ([]).



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

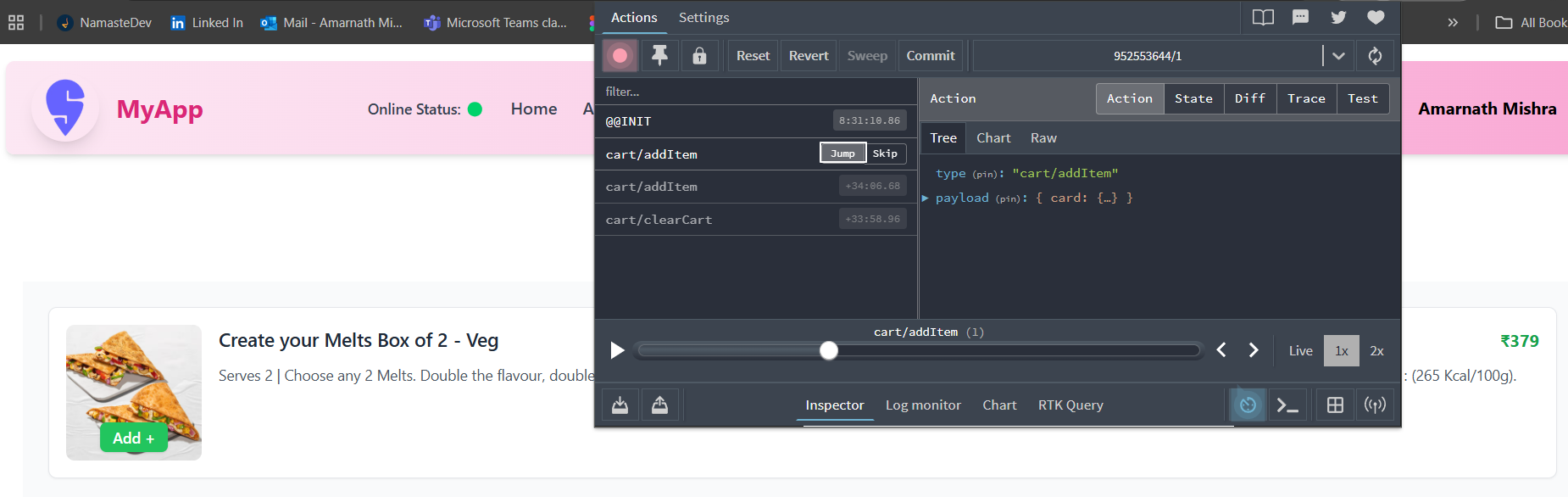
In this tab, we can see the changes in the state between different actions. For example, if the cart initially had two items and the user clicks the **Clear Cart** button, the cart items are updated to 0. This change, showing that 2 items were removed, is visible in the **Diff** section of the **clearCart** action in the Redux DevTools.



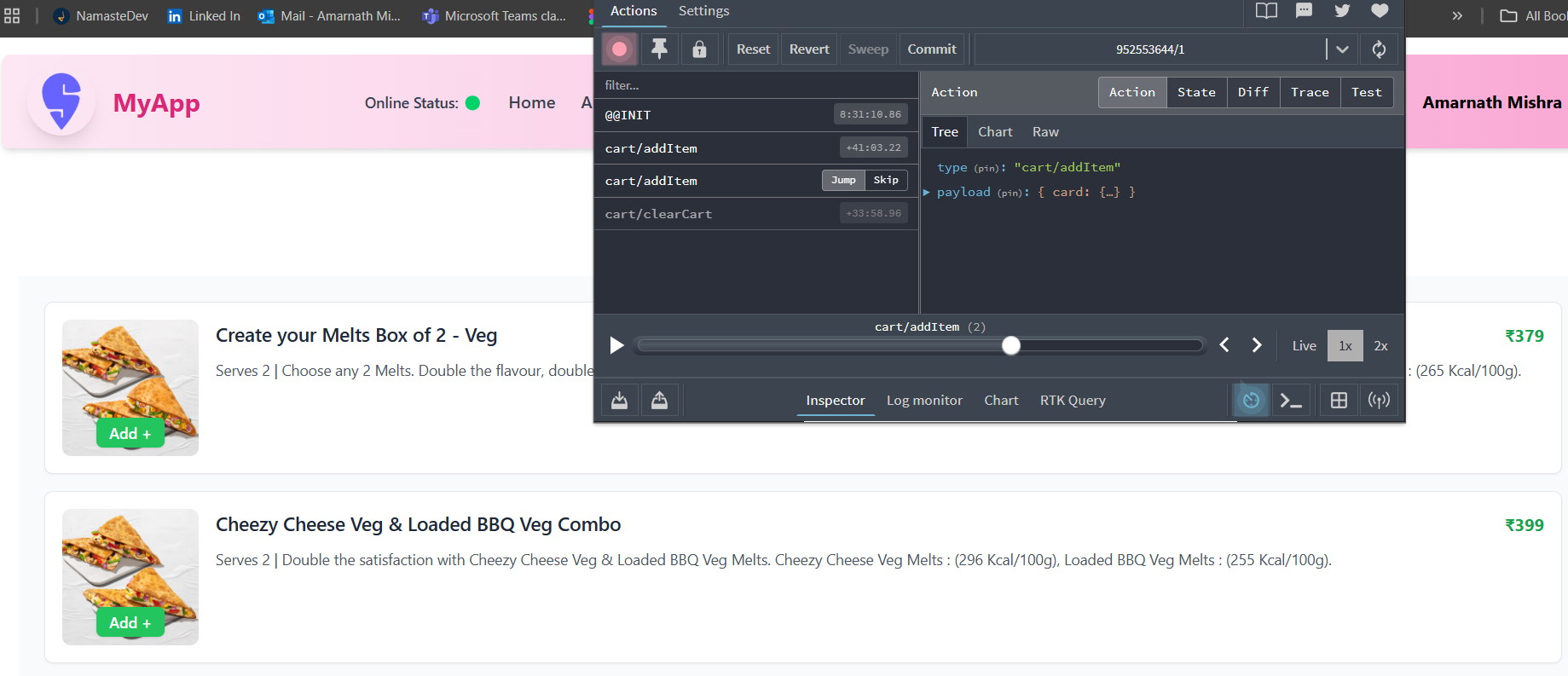
**Note -** We can replay consecutive action triggers and observe the UI updating accordingly as each action is invoked, one after the other.

**Note -** We can also jump to a specific action to view the corresponding UI updates triggered by that action.

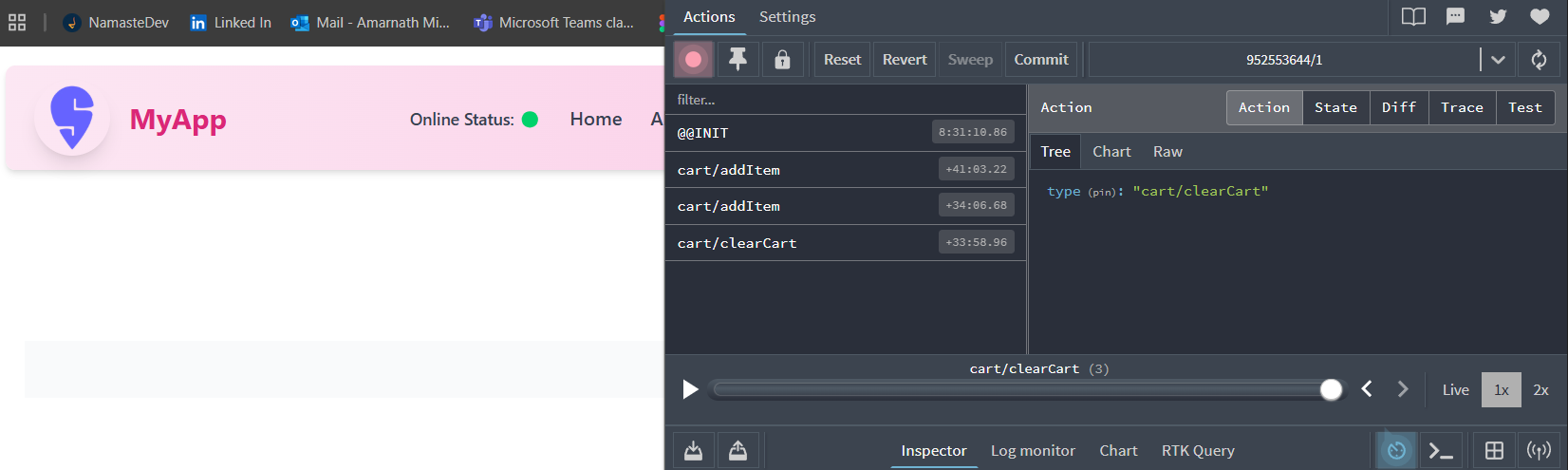
Jumping to the first addItem action -



Jumping to the second addItem action -



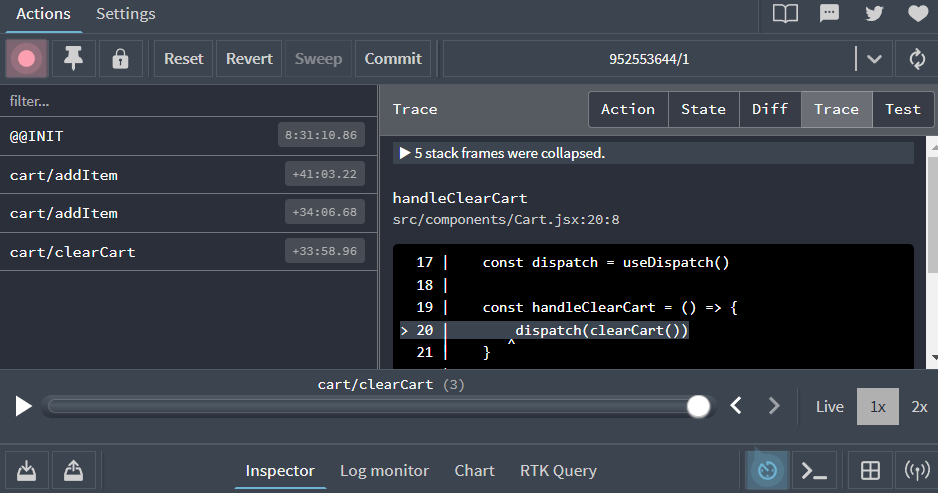
Jumping to clearCart action -



We can skip a specific action and replay the remaining actions to observe the updates in the UI. 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.

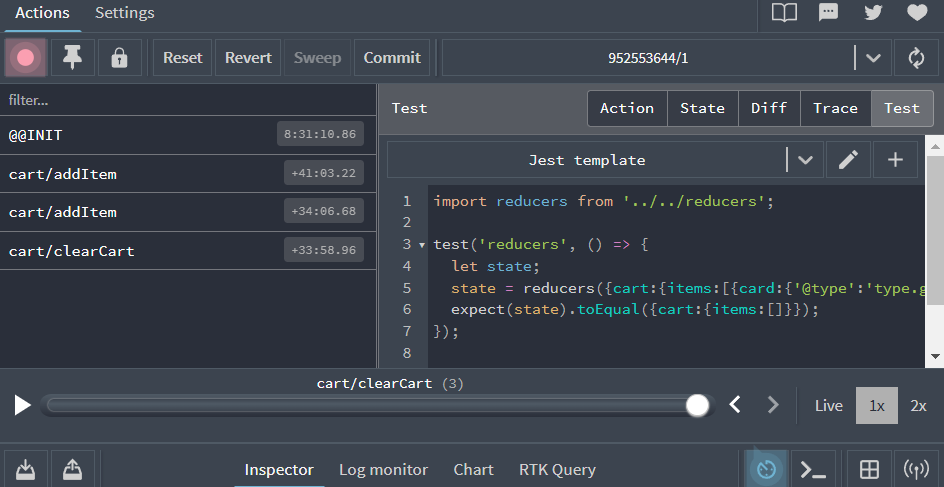
**Trace -**

We can trace a specific action to determine where it is dispatched in the codebase. The trace feature allows developers to pinpoint the exact line 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 handling server-side data in Redux applications by automatically generating actions and reducers for fetching, caching, and updating data. **Explore more online.**

What are Synthetic events ?

React creates its own version of events instead of using the ones directly provided by the browser. These custom events are called "synthetic events." React does this to make sure events work the same way on all browsers, so we don't need to worry about differences between them.

A synthetic event in React is an object that React uses to handle events consistently across all browsers.

React doesn't use the native events from each browser. Instead, it creates its own version of the events, called synthetic events. This ensures that properties like event.target (the element where the event happened) and event.type (the type of event, like 'click' or 'keypress') work the same in every browser.

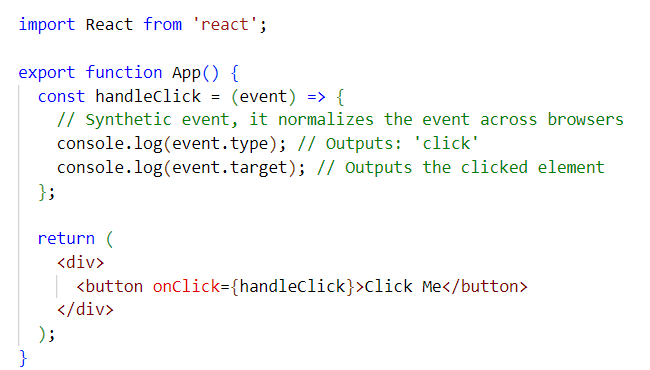
**Why is it useful?**

Cross-browser consistency - React makes sure that events behave consistently, regardless of whether you're using Chrome, Firefox, or any other browser.

Simplifies event handling - React’s synthetic events provide a uniform API, reducing the need for developers to write extra code for different browsers.

**Example -**

When we click a button in a React app, react creates a synthetic event to handle the click. This synthetic event will work the same way, no matter which browser we are using, making it easier to manage events in our app.



In the example above, the event passed to handleClick is a synthetic event created by React. React ensures properties like event.target (the element where the event happened) and event.type (the type of event, like 'click' or 'keypress') work the same in every browser, even if the native event behavior varies.