

# Episode-12 | Let's Build Our Store



Please make sure to follow along with the whole "Namaste React"

series, starting from Episode-1 and continuing through each subsequent episode. The notes are designed to provide detailed explanations of each concept along with examples to ensure thorough understanding. Each episode builds upon the knowledge gained from the previous ones, so starting from the beginning will give you a comprehensive understanding of React development.



I've got a quick tip for you. To get the most out of these notes, it's a good idea to watch Episode-12 first. Understanding what "Akshay" shares in the video will make these notes way easier to understand.

Q) `useContext` VS `Redux`

`useContext` and `Redux` are both tools used for state management in React applications, but they serve different purposes and have different use cases. Let's explore the key differences between `useContext` and `Redux`:



`useContext` t:

**Scope:** `useContext` is part of the React core and is used for managing state within the component tree. It provides a way to access the value of a context directly within a component and its descendants. It's typically used for smaller-scale state management needs within a component or a small section of the application.

**Complexity:** `useContext` is simpler and more lightweight compared to `Redux`. It's a part of the React library and doesn't introduce additional concepts or boilerplate code.

**Component Coupling** State managed with `useContext` is local to the component or a subtree of components where the context is provided. This can lead to more isolated and less globally shared state.

**Integration** It's seamlessly integrated into React and works well with the component lifecycle. You can create and consume contexts within functional components using the `useContext` hook.



`Redux`:

**Scope** `Redux` is a state management library that provides a global state container for the entire application. It allows you to manage the application state in a predictable and centralized manner.

**Complexity** `Redux` introduces a set of concepts, such as actions, reducers, and a store. This can make it more complex compared to using `useContext` for local state management. However, it becomes valuable in larger and more complex applications.

**Component Coupling** □ State managed with Redux is global, which means any component can connect to and access the state. This can be advantageous for sharing state across different parts of the application.

**Integration** □ Redux needs to be integrated separately into a React application. You need to create actions, reducers, and a store. Components interact with the global state using the connect function or hooks like useSelector and useDispatch.



#### Use Cases:

**Use useContext When** □ We have smaller-scale state management needs within a component or a local subtree. We want a lightweight solution without introducing additional complexity. Our state doesn't need to be shared extensively across different parts of the application.

**Use Redux When** □ We have a complex application with a large state that needs to be shared across many components. We want a predictable state management pattern with a unidirectional data flow. We need middleware for advanced features like asynchronous actions.

Choose **useContext** for simpler and local state management within component sections of our application. Choose **Redux** for more complex applications where global state that can be easily shared across different components.

## Q ) Advantages of using **Redux Toolkit over Redux** ?

**Redux Toolkit** is a set of utility functions and abstractions that simplifies and streamlines the process of working with Redux. It is designed to address some of the common pain points and boilerplate associated with using plain Redux. Here are some advantages of using Redux Toolkit over plain Redux:

**Less Boilerplate Code** □ Redux Toolkit helps us write less code. It provides shortcuts that save us from typing a lot of repetitive and verbose code, making our Redux logic cleaner and more concise.

**Easier Async Operations** □ If our app deals with things like fetching data from a server, Redux Toolkit makes it simpler. It has a tool called createAsyncThunk that handles async actions in a way that's easy to understand and use.

**Simpler Store Setup** □ Setting up your Redux store is easier with Redux Toolkit. It has a function called `configureStore` that simplifies the process, and it comes with sensible defaults, so you don't have to configure everything from scratch.

**Built-in DevTools Support** □ If you use Redux DevTools for debugging, Redux Toolkit has built-in support. Enabling it is as easy as adding one line of code when setting up your store.

**Encourages Best Practices** □ Redux Toolkit is recommended by the official Redux documentation. It encourages you to follow best practices in Redux development, making sure your code is more maintainable and aligns with industry standards.

**Handles Immutability for You** □ Working with immutable data (making sure you don't accidentally change your data) is usually a bit tricky. Redux Toolkit uses a library called Immer to handle this behind the scenes, so you can write more straightforward and readable code.

**Backward Compatibility** □ If you already have a Redux app, you can slowly transition to Redux Toolkit without rewriting everything. It's designed to be compatible with your existing Redux code.

**Faster Development** □ With Redux Toolkit, you can get things done more quickly. You spend less time setting up and configuring Redux and more time focusing on building features for your app.

In simple terms, Redux Toolkit is like a set of tools that makes working with Redux easier. It simplifies common tasks, reduces the amount of code you need to write, and encourages good coding practices. If you're starting a new project or thinking about improving an existing one, Redux Toolkit can save you time and make your life as a developer more enjoyable.

## Q ) Explain **Dispatcher** ?

In Redux, a **dispatcher** is not a standalone concept; instead, it's a term often used to refer to a function called `dispatch`. The `dispatch` function is a key part of the Redux store, and it plays a crucial role in the Redux data flow.



## Here's a breakdown of the dispatch function and its role in Redux

1 **Dispatch Function** The dispatch function is provided by the Redux store. We use it to send actions to the store. An action is a plain JavaScript object that describes what should change in the application's state.

2 **Usage** When we want to update the state in our Redux store, we create an action and dispatch it using the dispatch function.

```
const myAction = { type: 'INCREMENT' };  
store.dispatch(myAction);
```

- Here, the **INCREMENT** action is an example. The dispatch function is responsible for sending this action to the Redux store.

3 **Middleware** The dispatch function is also a crucial point in the Redux middleware chain. Middleware can intercept actions before they reach the reducer or modify actions on the way out. Middleware functions receive the dispatch function, allowing them to either pass the action along or stop it.

4 **Redux Store** The dispatch function is a core method provided by the Redux store. When an action is dispatched, the store passes the action through its reducer, which is a function that specifies how the state should change in response to the action.

5 **Asynchronous Actions** Redux supports asynchronous actions using middleware like **redux-thunk** or **redux-saga**. The dispatch function allows you to handle asynchronous operations by dispatching actions inside functions (thunks) and handling those actions asynchronously.



Here's an example of how you might use dispatch in a React component:

```
import { useDispatch } from 'react-redux';  
  
const MyComponent = () => {
```

```

const dispatch = useDispatch();

const handleButtonClick = () => {
  // Dispatching an action to increment the count
  dispatch({ type: 'INCREMENT' });
};

return (
  <button onClick={handleButtonClick}>
    Increment Count
  </button>
);
};

```

In this example, the `useDispatch` hook from react-redux gives us access to the dispatch function, which we then use to send an action to the Redux store when the button is clicked. This action will be processed by the reducer, updating the state accordingly.

Q ) Explain

Reducer?

In Redux Toolkit, the `createSlice` function is commonly used to create reducers. It simplifies the process of defining actions and the corresponding reducer logic, reducing boilerplate code. Let's break down the key concepts related to creating reducers with `createSlice` in Redux Toolkit :

**Creating a Slice** □ Instead of creating a standalone reducer function, you use `createSlice` to define a "slice" of your Redux store. A slice includes actions, a reducer, and the initial state.

```

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

const counterSlice = createSlice({

```

```

name: 'counter',
initialState: { value: 0 },
reducers: {
  increment: (state) => {
    state.value += 1;
  },
  decrement: (state) => {
    state.value -= 1;
  },
},
});

// Extracting actions and reducer from the slice
export const { increment, decrement } = counterSlice.actions;
export default counterSlice.reducer;

```

2 **Automatically Generated Action Creators**: `createSlice` automatically generate creators for each reducer function. In the example above, `increment` and `decrement` are automatically created and exported for use in your components.

3 **Immutability with Immer** Redux Toolkit uses the `immer` library internally to handle immutability. This means you can write reducer logic that appears to directly modify the state, but `immer` ensures it produces a new state without mutating the original.

```

reducers: {
  increment: (state) => {
    state.value += 1;    // Immer takes care of creating a new
    state               state
  },
},
},

```

4 **Reducer Function** The `createSlice` function returns an object that includes a `reducer` property. This reducer is a function that you can use in your store's configuration.

```
const rootReducer = combineReducers({
  counter: counterSlice.reducer,
  // ... other reducers
});
```

5 **Initial State** The `initialState` property in `createSlice` defines the initial state of your slice. This is the starting point for your state before any actions are dispatched.

6 **Reducer Logic** The logic inside each reducer function specifies how the state should change in response to the associated action. In the example, the increment and decrement reducers modify the `value` property of the state.

7 **Simplifying Reducer Composition** With `createSlice`, we can easily compose reducers using `combineReducers` or by directly adding the slice's reducer to the root reducer. This simplifies the overall reducer composition in our application.

Using `createSlice` in Redux Toolkit streamlines the process of defining reducers, actions, and initial states, making your Redux code more concise and readable. It encourages best practices, such as immutability and simplicity, while reducing the boilerplate traditionally associated with Redux.

## Q ) Explain **Slice** ?

---

In Redux Toolkit, a **slice** is a collection of Redux-related code, including reducer logic and actions, that corresponds to a specific piece of the application state. Slices are created using the `createSlice` utility function provided by Redux Toolkit. The primary purpose of slices is to encapsulate the logic related to a specific part of the state, making the code more modular and easier to manage.



Here's a breakdown of key concepts related to slices in Redux Toolkit:



Creating a Slice: The `createSlice` function takes an options object with the following properties:

1 **name (string)** □ A string that identifies the slice. This is used as the prefix for the generated action types.

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

const mySlice = createSlice({
  name: 'mySlice',
  initialState: { /* ... */ },
  reducers: {
    // ...reducers
  },
});
```

2 **initialState (any)** □ The initial state value for the slice. This is the starting point for your state before any actions are dispatched. 3 **reducers (object)** □ An object where each key-value pair represents a reducer function. The keys are the names of the actions, and the values are the corresponding reducer logic.

```
const mySlice = createSlice({
  initialState: { /* ... */ },
  reducers: {
    increment: (state) => {
      state.value += 1;
    },
    decrement: (state) => {
      state.value -= 1;
    },
  },
});
```

Output: The `createSlice` function returns an object with the following properties:

**name (string)** The name of the slice. **reducer (function)** The reducer function generated based on the provided reducers. This is the function you use in your store configuration. **actions (object)** An object containing the action creators for each defined reducer. These action creators can be directly used to dispatch actions.

```
const { increment, decrement } = mySlice.actions;
```

Using a Slice: Once we've created a slice, you can use its reducer and actions in your Redux store configuration and in your React components.

1 **Configuring the Store** We can include the generated reducer in your store configuration.

```
import { configureStore } from '@reduxjs/toolkit';
import mySliceReducer from './path/to/mySlice';

const store = configureStore({
  reducer: {
    mySlice: mySliceReducer,
    // ...other reducers
  },
});
```

2 **Dispatching Actions** In our React components, we can use the generated action creators to dispatch actions.

```
import { useDispatch } from 'react-redux';
import { increment } from './path/to/mySlice';

const MyComponent = () => {

  const dispatch = useDispatch();
  const handleIncrement = () => {

    dispatch(increment());
  };
};
```

```
// ... rest of the component logic  
};
```

Using **slices in Redux Toolkit** promotes a modular and organized approach to state management. Each slice encapsulates the logic related to a specific part of the state, making it easier to understand, maintain, and scale your Redux code.

## Q ) Explain **Selector** ?

In Redux Toolkit, a **selector** is a function that extracts specific pieces of data from the Redux store. It allows you to compute derived data from the store state and efficiently access specific parts of the state tree. Selectors play a crucial role in managing the state in a clean and efficient way.

Redux Toolkit provides the `createSlice` and `createAsyncThunk` utilities along with the `createSelector` function from the `reselect` library to help manage selectors easily.



Here's an explanation of how selectors work in Redux Toolkit

1 **Defining Selectors with `createSlice`** When we create a slice using `createSlice`, we can include selectors in the `extraReducers` field. These selectors can compute and return specific pieces of data from the state.

```
import { createSlice } from '@reduxjs/toolkit';  
  
const mySlice = createSlice({  
  name: 'mySlice',  
  initialState: { data: [] },  
  reducers: {  
    // ...reducers
```

```

},
extraReducers: (builder) => {
  builder
    .addCase(otherSliceAction, (state, action) => {
      // logic for handling other slice's action
    })
    .addDefaultCase((state, action) => {
      // default logic for handling actions not handled by
      this slice
    });
},
selectors: (state) => ({
  // selector functions here
  selectData: () => state.data,
  selectFilteredData: (filter) => state.data.filter(item =>
item.includes(filter)),
}),
});

export const { selectData, selectFilteredData } = mySlice.selectors;

```

2 **Using Reselect with createSlice** If we need more advanced memoization and composition of selectors, you can use the createSlice function along with the reselect library.

```

import { createSlice, createSelector } from '@reduxjs/toolkit';

const mySlice = createSlice({
  // ... other options
  selectors: {
    selectData: (state) => state.data,
    selectFilteredData: createSelector(
      (state) => state.data,
      (_, filter) => filter,

```

```

        (data, filter) => data.filter(item => item.includes(fil
ter))
    ),
    },
  });

export const { selectData, selectFilteredData } = mySlice.sel
ectors;

```

3 **Using Selectors in Components** Once we've defined selectors, we can use them in your React components using the useSelector hook from the react-redux library. This hook allows you to efficiently extract and subscribe to parts of the Redux store.

```

import { useSelector } from 'react-redux';
import { selectData, selectFilteredData } from './mySlice';

const MyComponent = () => {

  const data = useSelector(selectData);
  const filteredData = useSelector(state => selectFilteredDat
a(state, 'someFilter'));

  // ... rest of the component logic
};

```


Selectors help keep your state management logic clean and efficient by allowing us to centralize the computation of derived data from the Redux store. They contribute to better organization, improved performance, and easier maintenance of our Redux code.

Q ) Explain **createSlice** and the configuration it takes?


---

`createSlice` is a utility function provided by Redux Toolkit that simplifies the `process of creating Redux slices`. A `Redux slice is a piece of the Redux store that includes a set of actions, a reducer, and an initial state`. The `createSlice` function helps reduce boilerplate code associated with defining actions and the reducer for a specific slice of your Redux store.

Here's an explanation of the configuration options that `createSlice` takes:

 Syntax:

```
createSlice(options)
```

 Configuration Options:

1 `name (string)` A string that identifies the slice. This is used as the prefix for the generated action types.

```
const mySlice = createSlice({  
  name: 'mySlice',  
  // ... other options  
});
```

2 `initialState (any)` The initial state value for the slice. This is the starting point for your state before any actions are dispatched.

```
const mySlice = createSlice({  
  initialState: { value: 0 }, // ...  
  other options  
});
```

3 **reducers (object)** An object where each key-value pair represents a reducer function. The keys are the names of the actions, and the values are the corresponding reducer logic.

```
const mySlice = createSlice({
  reducers: {
    increment: (state) => {
      state.value += 1;
    },
    decrement: (state) => {
      state.value -= 1;
    },
  },
  // ... other options
});
```

4 **extraReducers (builder callback)** A callback function that allows you to define additional reducers outside of the reducers field. It is called with a builder object that provides methods for adding reducers based on other action types.

```
const mySlice = createSlice({
  extraReducers: (builder) => {
    builder
      .addCase(otherSliceAction, (state, action) => {
        // logic for handling other slice's action
      })
      .addDefaultCase((state, action) => {
        // default logic for handling actions not handled by
        this slice
      });
  },
  // ... other options
});
```


5 **slice (string)** An optional string that specifies a slice of the state to be used with the createAsyncThunk utility. This is useful when working with asynchronous

actions.

```
const mySlice = createSlice({
  slice: 'myAsyncSlice',
  // ... other options
});
```

6 **extraReducers (object)** An alternative way to define extra reducers using an object directly. Each key represents an action type, and the value is the corresponding reducer function.

```
const mySlice = createSlice({
  extraReducers: {
    [otherSliceAction.type]: (state, action) => {
      // logic for handling other slice's action
    },
    // ... additional action types
  },
  // ... other options
});
```

 **Output :**

The createSlice function returns an object with the following properties:

**name (string)** The name of the slice. **reducer (function)** The reducer function generated based on the provided reducers and extraReducers. This is the function you use in your store configuration. **actions (object)** An object containing the action creators for each defined reducer. These action creators can be directly used to dispatch actions.

```
const { increment, decrement } = mySlice.actions;
```



These are the main configuration options for createSlice in Redux Toolkit. It provides a convenient way to define actions, reducers, and initial states for slices of our Redux store, reducing the amount of boilerplate code and promoting best practices.

## Self- Notes –

React is not mandatory, use it when its required

When you are building the small application or mid-size application you don't need redux but when we build the large skill application where the data heavenly, where read and write operation happening in the react application , lot of components , lot of data transfer between these components , allocation grow huge , we are redux.

React and redux are not the same things, redux is diff library and react is diff library

Ther is multiple libraries for the state management Zustand (lot of companies and developer using this)

### **Why do we use redux-**

First of all when you building the large skill application redux offers you the great solution for it handling data, managing your store but the other libraries also doing the same things.

When we use redux our application becomes easier to debug, there is the chrome extension (using redux dev tools)

<https://redux.js.org/>

### **Redux**

A JS library for predictable and maintainable global state management. It is not just tide to react , It works with other library and frameworks as well but its heavily used with react

### **Other Libraries from the Redux Team offers**

**React-Redux** (bridge between react and redux)

**Redux Toolkit** (newer way of written redux)

Why we are using these libraries –

In the older days there is the diff ways to written redux but now with the modern web development is coming up so redux is simplify itself a lot

<https://redux-toolkit.js.org/introduction/getting-started>

The **Redux Toolkit** package is intended to be the standard way to write [Redux](#) logic. It was originally created to help address three common concerns about Redux:

- ☐ "Configuring a Redux store is too complicated"
- ☐ "I have to add a lot of packages to get Redux to do anything useful"

"Redux requires too much boilerplate code" so we are using Redux tool kit (RTK)

Let us build the card flow, If I click on this Add the item go inside this card (header) and when we click on it will go to the card page where we can see what all the item are in my card. We can build this kind of feature and to store all the card information we will we using redux **Architecture of Redux Toolkit -**

Redux store - It's the big whole object. You can assume it's a big object and lot of data inside of it and it is kept in the single central global place. Any component access it inside our application, in the react application any component access the store, It can read data , write data from the store and it's a very big object and its kept in the central place and we keep all the data of our application into this redux store.

Que- Is it a good practice to keep all the data inside the whole big object

Yes , It is absolutely fine there is no problem of keep data on the very big object  
So that the redux store not very clumsy we have something known as slices inside the redux store.  
You can assume Slice is to be a small portion of the redux store and we can created the multiple slices inside the redux store.

Why do we need slices and what are the slices actually?

Whatever you need to create, you create the logical separation and will make the slices inside the redux store  
There can be a card slice, user slice, theme slice . The card slice will hold the all the data of the card.  
Initially the card can we empty array and later on as we put data inside of it , it just modify the data of the card slice

Use case-

When I click on the add button how does the data go inside the card slice  
We cannot directly add data to the card slice , redux say that you can directly modify the card slice  
There is a way we can do that –  
Suppose I click on this add button It will **dispatch** an **action**, what will happen after dispatching an action , it calls a function and the function(**reducer**) internally modifies the card.  
Now what is the function actually – This function known as the **reducer**

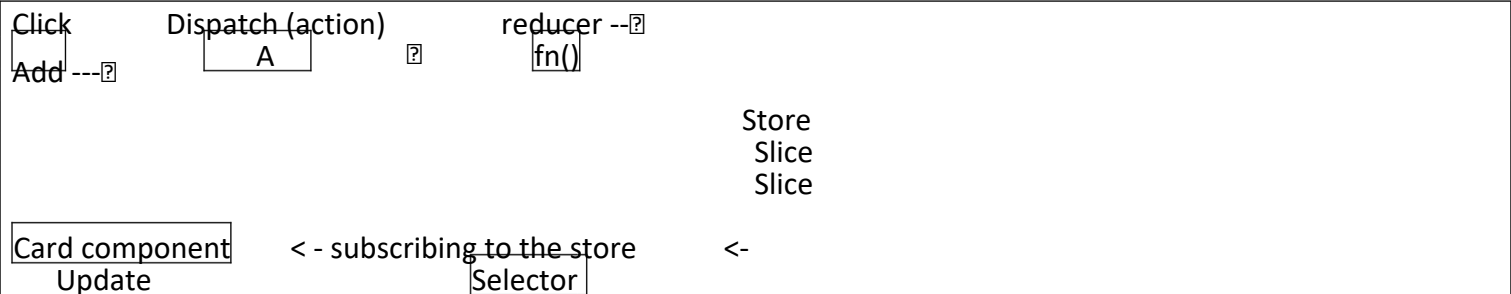
Once again when I click on the add button it dispatches an action which calls the reducer function which modifies /updates the slice of redux store. Now the card slice has some data inside of it. **This is how to write the data**

Suppose I want to read data. Now I want to get this data to some other part. **How can I read data**

For that we use **selector** to read the data from our store and the selector will modify are react component . This is how we read data. Selector will give you data over here.

When we use selector, this phenomenon known as **subscribing to the store**

We say that header component subscribe to a store and when I say subscribe the store basically we shink with the store. If the data inside the store changes my header component will update automatically.  
How do you subscribe – will subscribe using the selector



## # Redux Toolkit

- Install @reduxjs/toolkit and react-redux libraries
- Build our store
- Connect our store to our app
- Slice (cartSlice)
- dispatch(action)
- Selector

### Install @reduxjs/toolkit and react-redux libraries-

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

### Build our store –

#### appStore.js-

```
import { configureStore } from "@reduxjs/toolkit";  
import cartReducer from "./cartSlice";  
  
const appStore = configureStore({  
  reducer: {  
    cart: cartReducer,  
  },  
});  
  
export default appStore;
```

let us create the appStore and we will use a function which is known as configureStore to create the own store

So, basically we will use something configureStore and from where we get this function it will come from @reduxjs/toolkit and this configureStore will give us the store of the react application. This is how we build the appStore. We will add slices to it , Inside the store we will add slices.

### Now we will add the store to our application. How will be do that

So basically we need to add the store to our application. How do we provide it  
Will be importing the provider from react-redux

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

There is the difference between two libraries this reactjs/toolkit RTK library has things to do with redux so creating the store is redux things but now we have to provide store to react application its kind of bridge between there act application and redux so the provider comes from react-redux.

### Why configure store comes from reduxjs/toolkit?

Because configureStore it's the redux job, providing it to react application is the react-redux job, we are providing inside the react application that is why this provider coming from react-redux

**How do we proving in our app –** so basically wrap our whole application in the provider and pass the store as the props and it will take appStore here and it comes from utils

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

```

<Provider store={appStore}>
  <UserContext.Provider value={{ loggedInUser: userName, setUsername }}>
    <div className="app">
      <Header />
      <Outlet />
    </div>
  </UserContext.Provider>
</Provider>

```

This store we have providing to our application now

Suppose if you don't want to whole app to have redux, you want small portion to use redux so you just provide the provider in that portion to the app but whenever you need to use app, you need to store wrap inside the provider just like the context provider

### Now creating slices, how do you create slice—

It created by the function createSlice and it comes from @ reactjs/toolkit  
createSlice is the function and it takes the configuration to create the slice

First thing we have to given is the name of the slice, let gave name: 'card' and the second configuration it takes the initialState, what basically initially the card slice will be , what will be the card items it takes the initially state  
So now give the initial state items and the items is the empty object, So this is the initial state of the slice and then something known as reducers here we will write the reducer functions, **we will write reducer function correspond to those actions** so we will create here reducer and actions over here and reducer again an object

What is action?

Action is something known as what type of actions inside our cards, you can add the item , you can remove the items, clear the cards so these things are actions, Actions basically its kinds of small Apis you can think of it apis to communicate to redux store. So, if you want to add the item so I will dispatch the add items

addItem is the action an there is the reducer function which is map to this so reducer function actually the

modifies the data inside the slices. How do you modify it –

It get access to the state initially state is the empty array, this get access to the state and now it also get access to this action so this is the reducer function-

```

reducers: {
  //action //reducer function
  addItem: (state, action) => {
  },
}

```

It gets two paraments state and action and now it will modify our state based on the action

State is basically initially state we get access to the state of our store. How do we modify it – so suppose I will get item so I will do state.items and item is the array .push and I will push an action.payload inside it

```

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

```

When we will dispatch and add item we will get this payload when we will call the add item, dispatch the function

### Write the reducer for remove item-

removeItems again we have reducer and its take state and action and now over here again do the state.items and

I can remove my items from this array. I am removing one item form the end but what can we do is just find out action.payload

```
removeItem: (state, action) => {  
  state.items.pop();  
},
```

### Write the reducer for clear card-

Let write the reducer function and its don't need an action because we want to clear the card. How do I clear a card state.items and length so basically I will do a empty array is 0 , Suppose if there are 10 items in the card it will make it 0

```
clearCart: (state, action) => {  
  state.items.length = 0; // originalState = []  
},
```

Now we will export 2 things – actions and reducers

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

Now I want to add the slice to the store, how do you add it you create the reducer

If you modify the store it also has the reducer for itself and the reducer combine the reduce of slices so this reducer responses to modifies the appStore and this reducer basically the combination of different small stores For each slice we will have the different reducer so that is why we will import the reducer

```
import { configureStore } from "@reduxjs/toolkit";  
import cartReducer from "./cartSlice";  
  
const appStore = configureStore(  
  {  
    reducer: {  
      cart: cartReducer,  
    },  
  }  
);  
  
export default appStore;
```

Now just see can we read data from here. How will I read; I want to show how many items are in the card so we have to subscribe to the store using the selector

Header.js has the card

Lets just added pizza and burger to an card (dummy values) and when I am read it will show me 2 items So how I will read it so basically, we will use the selector

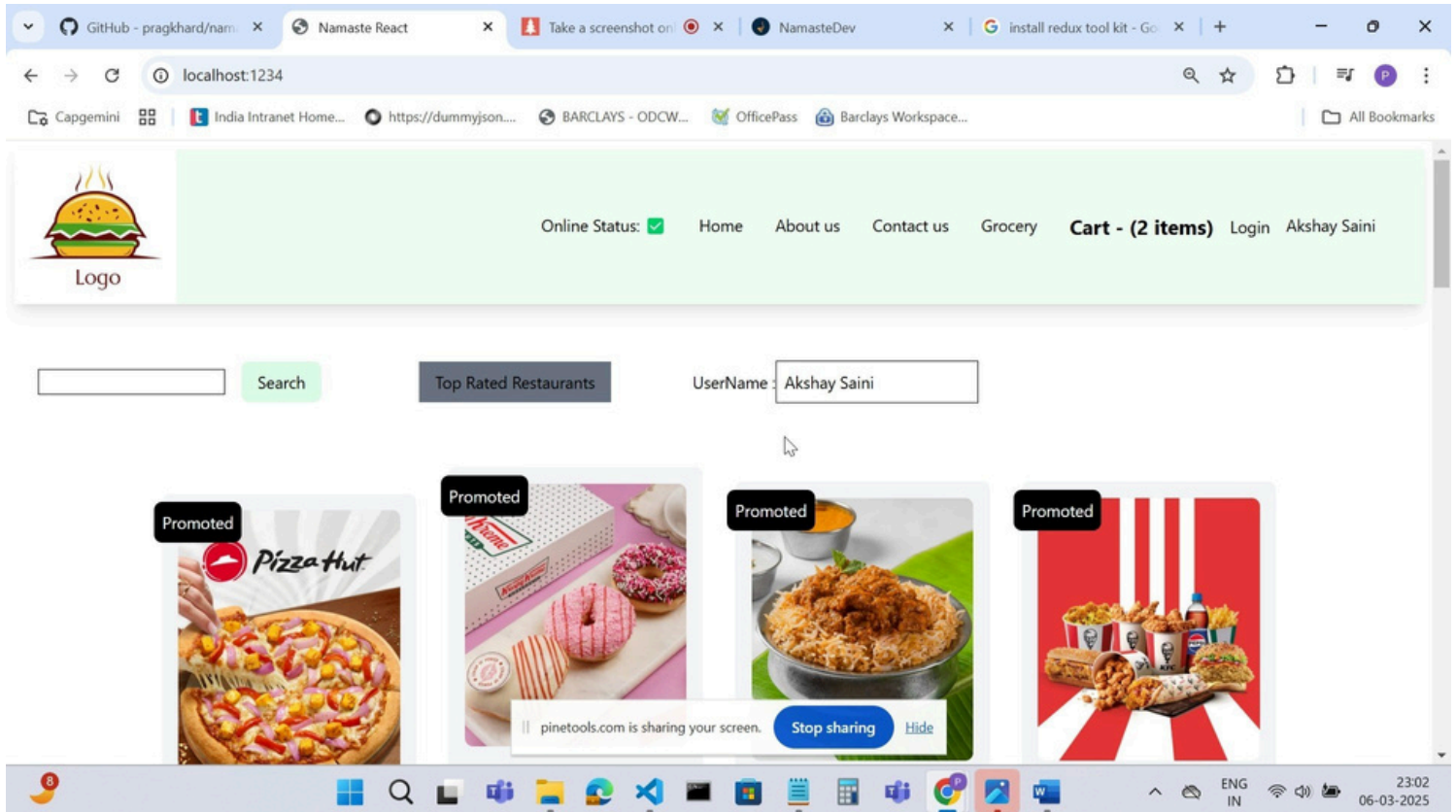
What is the selector?

Selector is hook inside in react and it comes from react-redux library. Basically, the hook gives us access to the store. So now we are subscribing to the store using a selector. It basically gives the access to the store but we will now tell them what portion of the store we need access. Suppose I need access to store.card.item

Now the card will get the data of this items

```
// Subscribing to the store using a Selector
const cartItems = useSelector((store) => store.cart.items);
//console.log(cartItems);
```

```
<li className='px-4 font-bold text-xl'>Cart - ({cartItems.length} items)</li>
```



This is all how to read the store

Will get (2) ['burger', 'pizza'] on console

Let us use the real items and remove the dummy items

### Write the logic on add button

On click of add button we have to dispatch an action

Will write the function on top

How do I dispatch an action. First, I need to access an dispatch

Dispatch basically an function that we get from hook known as useDispatch and its given to us by react-redux

Now, I want to dispatch an action, we have export some actions in slice(cardSlice)

So first I will import the action and after that dispatch the action.

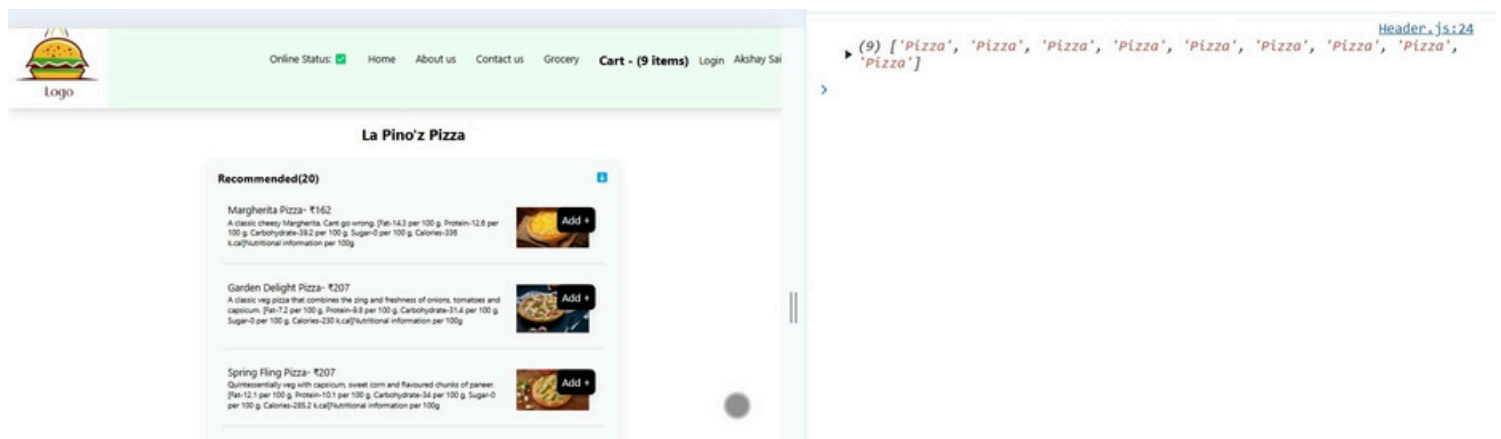
```
import { useDispatch } from "react-redux";
import { addItem } from "../utils/cartSlice";
```

```
const dispatch = useDispatch();
```

```
const handleAddItem = (item) => {
  // Dispatch an action
  dispatch(addItem(item));
};
```

and now whatever I will pass inside it will go inside my card suppose I will add pizza over here that is my action.payload. So whatever I pass here it will go to the reducer function action and that too inside the payload so now action.payload is pizza. Whenever we dispatch an action redux will create an payload inside the obj and it will add whatever data added to an object and it will take an object and pass it to the second argument so will get pizza overhere

```
const handleAddItem = (item) => {
  // Dispatch an action
  dispatch(addItem("Pizza"));
};
```



```
<button
  className="p-2 mx-16 rounded-lg bg-black text-white shadow-lg"
  onClick={() => handleAddItem(item)}
  Add +
</button>
```

So whenever I am click on the add button action will dispatch and calls the reducer function which updates the slice / modify the store because my header subscribe the store using the selector everything working seamlessly

Rightnow I am passing pizza over here instead of I want to pass the specific items on click  
 How do I pass the specific items over here I am iterating items, so I want to pass the specific items over here  
 So lets pass this items into my handle item so let me add the function and don't do this mistake  
 onClick={handleAddItem(item)} that means you have called the function already so do something like this  
 onClick={() => handleAddItem(item)}

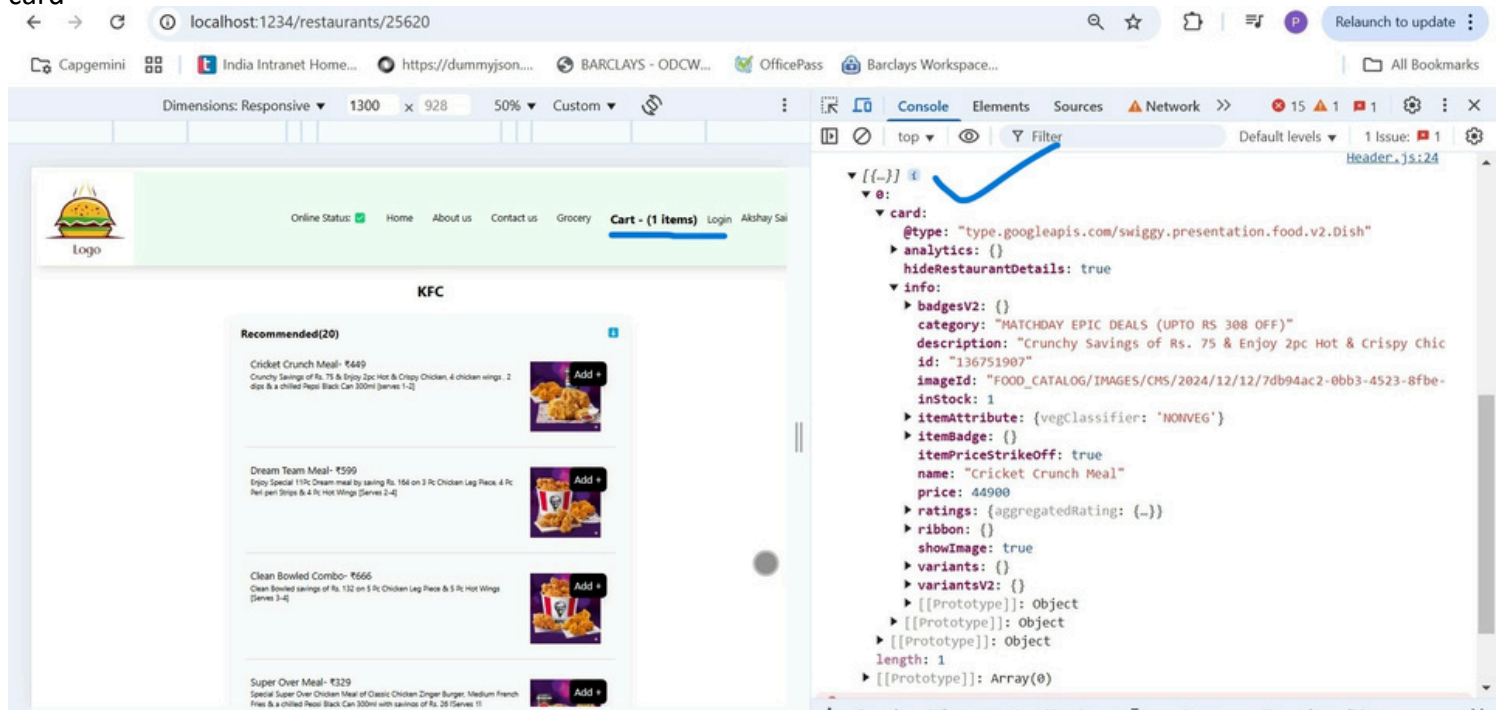
**Que- What is the difference between**

onClick={handleAddItem}  
 onClick={() => handleAddItem(item)} //want to pass the callback function so using this  
 onClick={handleAddItem(item)} // calling the function rightaway

Now pass the items in handleAddItem like this , and add the items inside my card now the items will be big object

```
const handleAddItem = (item) => {
  dispatch(addItem(item));
};
```

Now I click on the add button see this we have updated the card and we got this one card and one item into the card



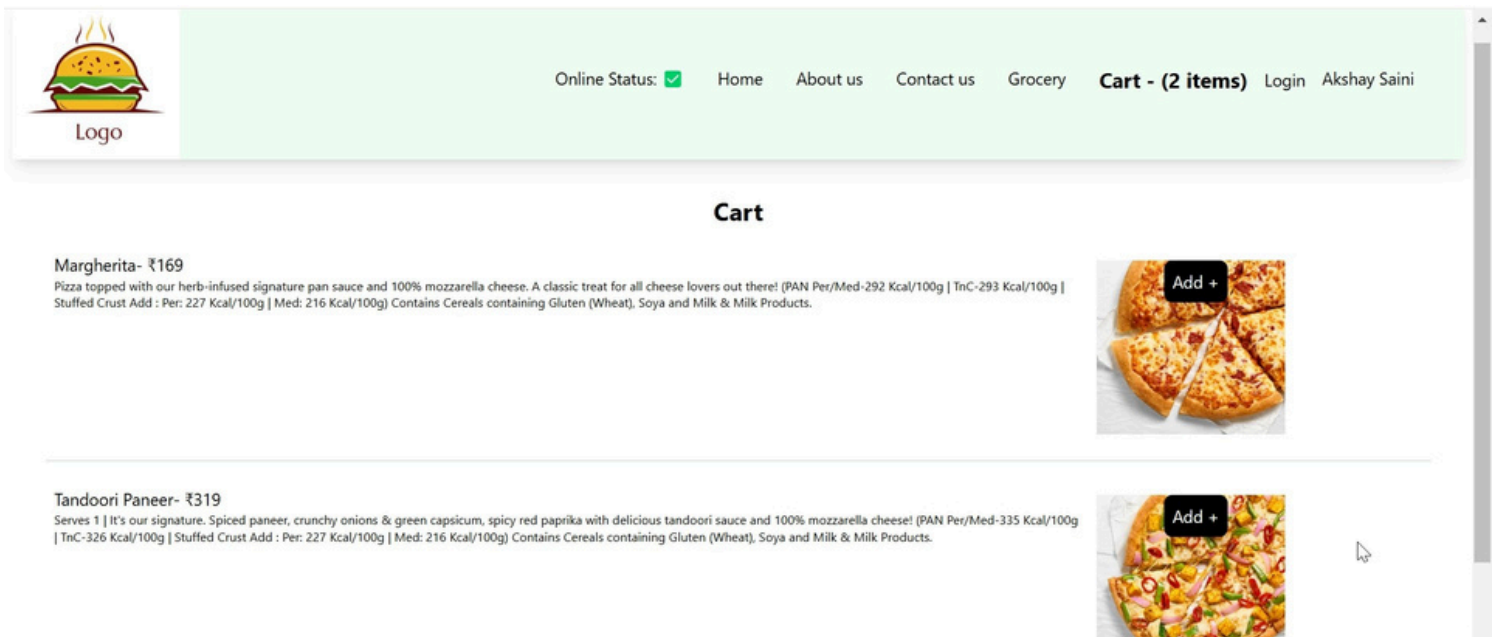
**Let's build the card page where we see the card info-**  
 Now create the card component  
 Let add one more route for card component in App.js



**How will I read my card / read the card info./ How will I read card from the store?** We have to subscribe the store using useSelector, it will access to whole store but we want access to only the small part of store so we do (store.cart.items)

```
import { useSelector } from "react-redux"; const cartItems = useSelector((store) => store.cart.items); so I will  
get my cart items here let show the cart items below-  
use the itemList ui in the cart components. It's the good reuse of the component  
import ItemList from "./ItemList"; <div className="text-center m-4 p-4">
```

```
<h1 className="text-2xl font-bold">Cart</h1>  
<div>  
  <ItemList items={cartItems} />  
</div>  
</div>
```



Let's build the feature of clear cart here-

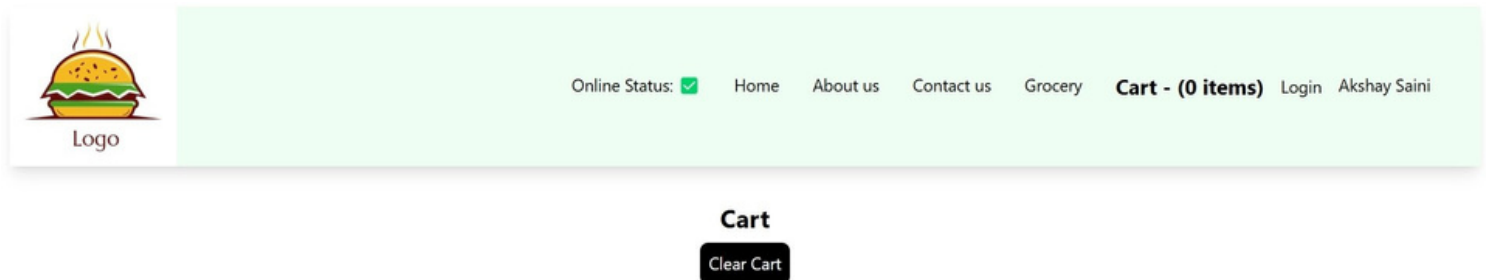
```
import { useDispatch, useSelector } from "react-redux";
import { clearCart } from "../utils/cartSlice";
```

```
const dispatch = useDispatch();
```

```
const handleClearCart = () => {
  dispatch(clearCart());
};
```

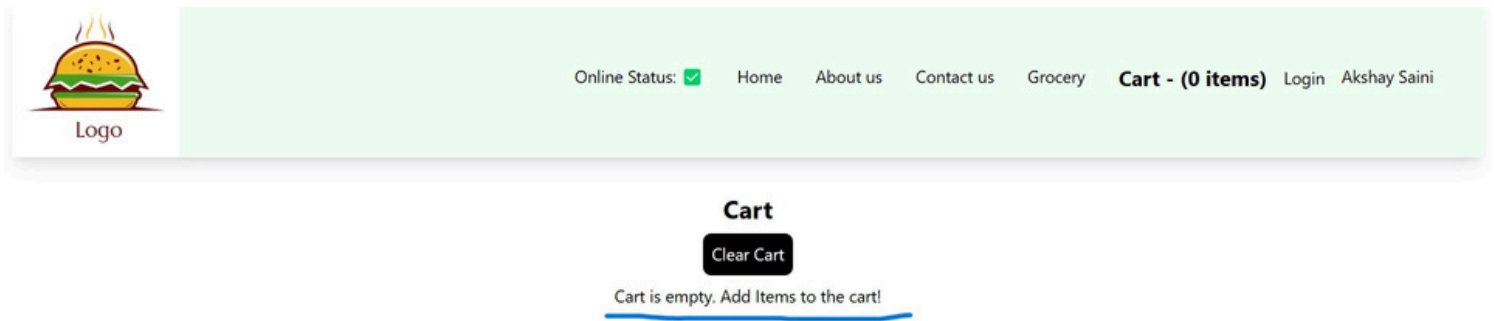
```
<button
  className="p-2 m-2 bg-black text-white rounded-lg"
  onClick={handleClearCart}
>
  Clear Cart
</button>
```

If I click on the clear cart it should clear my cart , it should dispatch an action so create the handleClearCart function, It basically dispatch an action, How do you get the dispatch function , using useDispatch hook, What action should I dispatch “clear cart” it will come from cartSlice and import and call it



If the items is 0 you can just show /display a message add items to the cart

```
{cartItems?.length === 0 && (
  <h1> Cart is empty. Add Items to the cart!</h1>
)}
```



**When we click on the button it dispatch the action which calls the reducer function which updates the slices of the store and because this header component subscribes to the store using the selector it automatically updated.**

### Interview questions-

1. When you are using selector make sure you are subscribing to the right portion of the store. Here you are optimizing the performance. If you don't subscribe the right portion of store, it will be the big performance loss

We can do something like this:

I can do the store and can access to the whole store and extract my cart items

```
const cartItems = useSelector((store) => store.cart.items);
```

here I am subscribing this portion of the store

```
const store = useSelector((store) => store);
```

```
const cartItems = store.cart.items;
```

here I am subscribing to the whole store and extracting my items. It will perfectly fine but this very less efficient what happen is when you write like this so basically this store variable is subscribed to the redux store whenever anything changes inside the store your cat component get to know basically your store variable will be updated whenever anything changes to the store so you don't want to update, you don't want to subscribe the whole store. Suppose something happens in another slice it is nothing to do with cart so why this store variable subscribes to that store.

When the application grow huge are store is very big so anything randomly changes in our store I don't want be this component affected by I don't want to subscribe my whole store it is foolish better way is to only subscribe to the specific portion of the store. Never do this only subscribe small portion of the store.

Why the name is selector- > because you are selecting the selected portion of the store.

2. There is the confusion between reducer and reducers

When you are creating the appStore here the keyword is reducer because it's the one big reducer for appStore and this reducer can have multiple small reducer. **Reducer is nothing but the combination of small reducer**

But when we are writing slice, we create multiple reducers so this word reducers and when you are exporting it you are just exporting the one reducer **and reducers is the combination of small reducers functions**

### Reducer-

```
import { configureStore } from "@reduxjs/toolkit";
import cartReducer from "./cartSlice";

const appStore = configureStore(
  {
    reducer: {
      cart: cartReducer,
    },
  }
);

export default appStore;
```

### reducers-

```
import { createSlice, current } from "@reduxjs/toolkit";

const cartSlice = createSlice({
  name: "cart",
  initialState: {
    // items: ["burger", "pizza"],
    items: [],
  },
  reducers: {
    //action      //reducer function
    addItem: (state, action) => {
      // We are mutating the state over here
      // Redux Toolkit uses immer BTS
      state.items.push(action.payload);
    },
    removeItem: (state, action) => {
      state.items.pop();
    },
    //originalState = {items: ["pizza"]}
    clearCart: (state, action) => {
      //RTK - either Mutate the existing state or return a new State
      // state.items.length = 0; // originalState = []
      return { items: [] }; // this new object will be replaced inside originalState = { items: [] }
    },
  },
});
```

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

3. Earlier when we used to write vanilla redux (older version) of redux there was a big problem with state here, when we used to write the reducer function in vanilla redux, it gives us the warning – Don't mutate state. Right now, we are mutating the state but with the older version I was prohibited, it is like the impure function. How we used to do – basically we used to create the copy of state variable and then we used to push  
Create the new state from my state and will modify the new state and then return to the new state. This is how we used to earlier. Because redux say you don't modify (mutate) the state

```
//Vanilla(older) Redux => DON'T MUTATE STATE  
  
const newState = [...state];  
newState.items.push(action.payload)  
return newState
```

Redux Toolkit give us the option we have to mutate the state

```
// We are mutating the state over here  
state.items.push(action.payload);
```

earlier returning was mandatory and now returning is not mandatory redux take care of it automatically

Redux uses something known as immer library

It is kind of like to finding the difference between original state, the mutated state and gives you back the new state which is immutable state (new copy ) of the state so whatever I was writing the logic all that logic has been abstracted and we as the developer don't take care of it and immers take care of it.

<https://immerjs.github.io/immer/>

### Redux DevTools

It's an extension and it will activate once you have the redux app with you and it helps you for so much debugging . It basically see you the log of everything


Capgemini

India Intranet Home...

https://dummyjson...

BARCLAYS - ODCW...

OfficePass



Online Status: ✓ Home About

# KFC

## Recommended(20)

### LATE NIGHT SPECIALS (STARTING AT 199)(14)

Craving Saving meal- ₹779

Crave & Save whooping Rs. 213 on 4 Pc Hot & Crispy, 4 Pc Peri Peri Strips, Medium Popcorn, 2 Dips & a Pepsi PET 475ml [Serves 2-3]

#### Extensions

Full access  
These extensions can see and change information on this site.

- Adobe Acrobat: PDF edi...
- Allow CORS: Access-Co...
- Careerflow AI Job Appli...
- Nextthink
- React Developer Tools
- Redux DevTools
- ScreenClip - Screenshot...

No access needed  
These extensions don't need to see and change information on this site.

Manage extensions

ms) Login Akshay Saini


Capgemini

Ind

Actions Settings

811020422/1

All Bookmarks



rt - (1 items) Login Akshay Saini

filter...

@@INIT 12:29:44.04

cart/addItem +38:08.21

Diff

Tree Raw

```
▼ cart (pin)  
  ► items (pin): { 0: [...] }
```

Inspector Log monitor Chart RTK Query

Crave & Save whooping Rs. 213 on 4 Pc Hot & Crispy, 4 Pc Peri Peri Strips, Medium Popcorn, 2 Dips & a Pepsi PET 475ml [Serves 2-3]

Add +