What is React

1. React is a JavaScript library for building user interfaces. It's like a set of building blocks that developers use to create interactive and dynamic web pages.
2. React introduces the concept of components, which are like reusable Lego blocks. Each block can be developed, maintained, and tested independently. If you need to change something, you just replace or update that specific block (component)
3. Example: A "button" on a website can be a component. You can create one button component and use it everywhere on your site, changing its appearance or behavior as needed without rewriting the button code each time.

Why React ?

Some of the major challenges that React solves

State management

1. We discussed intuition about breaking the entire UI into small reusable components

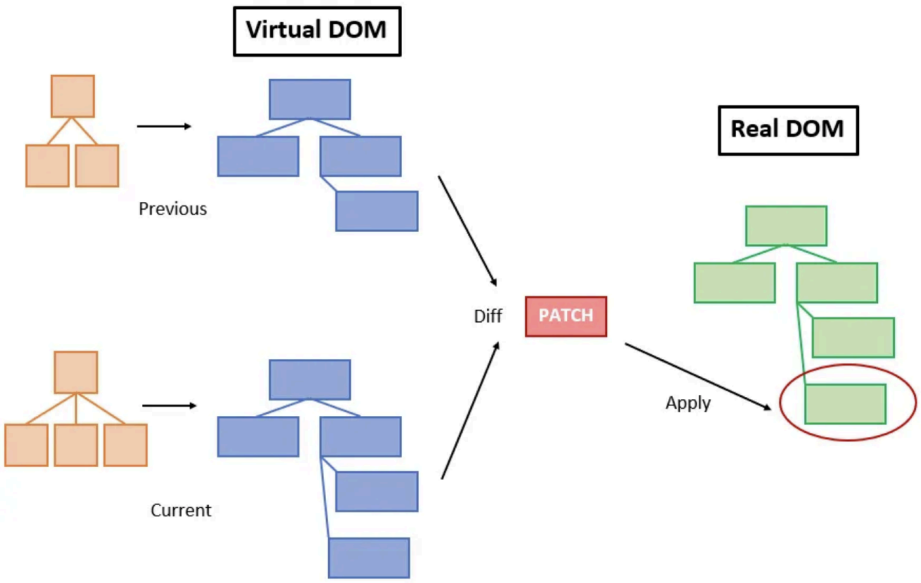
2. Now lets look at one another aspect of web development which is called state updates
3. What is a state ?
    a. let s consider a food ordering app
        i. When you click on a burger to order, the count changes to 1
        ii. If you click again, the count changes to 2
        iii. What is happening?
            1. based on an event something changed on UI
            2. There may be some alert msg on success
            3. You might want to hide / show something based on a button click
        iv. All of this is state management
    b. React does an amazing job in helping us carry out state updates in very efficient manner
    c. You can still do this with vanilla js but with react it is done more efficiently and with lot less code

Efficient DOM Manipulation

1. The DOM is like a tree structure that represents the layout of a webpage. It's similar to a family tree, but for a website, showing how each part of the webpage is connected to others
2. In traditional web development, every time something changes on the webpage, the browser might need to update the Document Object Model (DOM), which can be slow for complex

applications. The slowness happens because browser repaint the whole page.

3. Engineering challenge before react team was that DOM updates are slow.
4. This is solved by React by using the concept of Virtual DOM
    a. Basically React creates a light weight replica of the actual DOM.
    b. All the updates happen on this Virtual DOM
        i. So no re-rendering happens in this process
    c. Finally React compares the two snapshots of Virtual DOM (one before the update and one after the update).



    d. Now only the required differences are applied to DOM
    e. The Virtual DOM doesn't eliminate the need to interact with the real DOM, but it minimizes and optimizes those

interactions, leading to improved performance and a better user experience

f.

Markup reusability

a. This is setup using create-react-app
b. This is a package to create a new react app
    i. npx create-react-app my-app
    ii. cd my-app
    iii. npm start
c. If you want to use vite
    i. npm create vite@latest my-react-app -- --template react
    ii. cd my-react-app
    iii. npm install
    iv. npm run dev

2. index.html - only html file (SPA)
3. Index.js is the starting point for the react app
4. App is the first component
    a. Think of App.jsx as the main entry point or the 'home base' for our React application.
    b. From here, we organize and add other parts of our website, like different sections or features, which are like individual stories in the newspaper or rooms in the house

c. This, like all other react components is a function and it returns something
d. Everything we put inside the 'return' is what gets displayed on our website's main page."
e. And this syntax that you see guys which looks like html but is called JSX
f. JSX
    i. JSX stands for JavaScript XML.
    ii. It's a special way to write code that looks like HTML but works inside your JavaScript files.
    iii. With JSX, you can describe how your website's interface should look in a way that's easy to read and write, almost like you're directly writing the webpage layout.

Markup reusability code

1. We create a component to display the different phones
2. Under components folder, we have a Product folder
    a. Product.js has the layout structure which has the html like syntax and the css
    b. It has an img tag waiting for its src value
    c. It needs some title in h3 and some description
    d. This is also a react component which is nothing but a function that returns jsx
    e. Let us import this component in App.jsx

3. In our code, Product is a component - think of it as a box that displays a product on our website.
4. Now, we want to show specific details about a product, like its title, image, and price. We do this by passing props to our Product component.
5. And inside the jsx whenever you want to write javascript like the iphone obj, we do it inside curly braces
6.
```
<Product title={iphone.title} image={iphone.image}
price={iphone.price}/>
```
7.
8. In the App.js, when we write <Product title={iphone.title} image={iphone.image} price={iphone.price} />, we are giving the Product component the information it needs to display a particular product - in this case, an iPhone.
9. So, props are like parameters we pass to our component to tell it what to display or how to behave. Each prop (title, image, price) carries specific data (iphone.title, iphone.image, iphone.price) that the Product component uses to render the iPhone details on the screen."

```
import "./Product.css";
const Product = ({title, price, image}) => (
 <div className="product-card">
   <div className="product-image">
     <img src={image} />
   </div>
   <div className="product-details">
     <h3>{title}</h3>
```

```
      <span>{price}</span>
    </div>
  </div>
);

export default Product;
```

10.
11. While this is grt but what if we have 1000s of products to display
12. Create a product array
13.
```
const products = [iphone, samsung, nokia]
```

    a. We can loop over this array and use the same component to render different phones
    b. To create a loop , I will take help of javascript and if i need to write JS inside JSX how do i write it

```
{products.map((product) => (
        <Product
          title={product.title}
          image={product.image}
          price={product.price}
        />
      ))}
```

    c. Appreciate that if there were thousands or lacs or products, do I need to write any more code than here
    d. Fix the warning - `Warning: Each child in a list should have a unique "key" prop. In next section`

Understand why this is important: if you were given a task to feed a triplets (same looking babies)

    e. Similarly adding keys here helps in Avoiding unnecessary renders/ repainting using VDOM
14. So this is the markup reusability benefit

Hooks in react

1. Let us understand about the concept of hooks in react
2. Hooks simply put are functions … but special functions
3. So Hooks in React are special functions that let you 'hook into' React features. For example, they allow you to keep track of data in your component (with 'useState') or perform actions when something changes (with 'useEffect').
4. Imagine you have a basic car that drives well, but you want to add some special features to it. In React, your components are like this basic car.
5. And there are hooks like
    a. useState
    b. The useState hook is like adding a digital dashboard to your car that displays and updates information like speed or fuel level. In your React component, useState allows you to keep track of and update information dynamically.
    c. Application can have multiple states like success, error states, hiding a section,etc.
    d. React makes it very easy to create application state data and cause our UI to update when the state data changes

    e. Generally after an event we want to update our UI. To do that we maintain state variables and update them after an event (user action)
    f. Ordering food by clicking on counter
6. useEffect
    a. Lets understand this little intuitively
    b. Basically another challenge that we have while building an application is to sync our application with an external system
    c. External system is something that we are not developing as part of our code.
    d. For eg. our component might need to connect with a server to get some data like a chat application
    e. It tells React to do something after rendering. For example, you might want to fetch data from a database, subscribe to a service, or manually change the DOM in some way after your component shows up on the screen.

Coding time

1. Mock api - https://fakestoreapi.com/docs
2. Comment the Products in App and add a header and product list component

```
{/* {products.map((product) => (
      <Product
        key={product.id}
        title={product.title}
        image={product.image}
```

```
        price={product.price}
      />
    ))}
    */}
    <Header/>
    <ProductList/>
```

3. Import both

```
import Header from "./components/Header";
import ProductList from "./components/ProductList";
```

4. Header component
    a. Add logo

```
import Logo from '../logo.png'
```

    b. Update src

```
c.          <img src={Logo} alt="logo" />
```

    d.
    e. When this component loads, we want to fetch all the categories from an api
    f. This is the code that will fetch (from the fakestore api)

```
fetch('https://fakestoreapi.com/products/categories')
        .then(res=>res.json())
        .then(json=>console.log(json))
```

    g. Where should this code go ?
    h. useEffect

```

```

i. If you are using the fakestore api , use that

```
useEffect(()=>{

fetch('https://fakestoreapi.com/products/categories')
            .then(res=>res.json())
            .then(json=>console.log(json))
 })
```

j. Now we just dont want to console but reflect this data on the ui.

k. How should i store this data - state data using useState

```
const Header = () => {
 const [data, setData] = useState([]);
 useEffect(() => {
   fetch("http://localhost:3300/products/categories")
     .then((res) => res.json())
     .then((json) => setData(json));
 });
 return (
   <div className="header-items">
     <div className="logo">
       <img src={Logo} alt="logo" />
     </div>
     {data.map((category) => (
       <div className="header-item">{category}</div>
     ))}
     <div className="shopping-cart">
```

l. Now what we want is that when we click on a category it should be displayed little differently. Should show that it is selected

   i. What can we use to store selected category so that when we update it, the UI also updates

   ii.
```
 const [selectedCategory, setSelectedCategory] =
useState(null);
```

   iii. Now add a handler on the category div

```
 {data.map((category) => (
       <div
onClick={()=>setSelectedCategory(category)}
className="header-item"
key={category}>{category}</div>
     ))}
```

   iv. Now we need to style the category that is selected a little differently

```
<div
         onClick={() =>
setSelectedCategory(category)}
         className={
           "header-item " +
           (category === selectedCategory ?
"header-item--selected" : "")
         }
         key={category}
       >
```

5. ProductList component
    a. Let us now get some products on our application
    b. `fetch('https://fakestoreapi.com/products/category/jewelery')`
    c. `.then(res=>res.json())`
    d. `.then(json=>console.log(json))`
    e. Now this is a code that causes side effect so this should go in the useEffect. In the product list component

```
const ProductList = () => {
 useEffect(() => {

fetch("http://localhost:3300/products/category/electro
nics")
     .then((res) => res.json())
     .then((json) => console.log(json));
 })
```

    f. Where should we store the products - state variable

```
 const [products, setProducts] = useState([]);
 useEffect(() => {

fetch("http://localhost:3300/products/category/electro
nics")
     .then((res) => res.json())
     .then((json) => setProducts(json));
 })
```

    g. Loop over the products

```
return (
   <div className="products">
```

```
      {products.map((product) => (
        <ProductCard key={product.id} product={product}
/>
      ))}
    </div>
  );
```

h. Import the product card component
i. You can choose to have the product card layout here as well but it helps
    i. With single responsibility
    ii. Less code in one file
    iii. Handover the dev work to some other person
    iv. reusability
j. Update product card component

```
import AddToCart from "./AddToCart";

const ProductCard = ({product}) => {
 const {title, image, price} = product;
 return (
   <div className="product">
     <div className="product-top">
       <img src={image} className="image" />
       <div className="title">{title}</div>
     </div>
     <div className="product-body">
       <span>{price}</span>
       <AddToCart />
     </div>
   </div>
```

```
 );
};

export default ProductCard;
```

k. Now this should load the page. Add a dollar sign to the price

6. Lifting the state up
    a. Notice now friends that no matter what category i select , my list of products is not changing
    b. This is because in product list i am passing a hard coded category
    c. Now understand that both my header component and the product list needs a selected category
    d. Header needs because to style it differently, product list needs it because to get the relevant products
    e. if two or more components needs the same state, then we lift that state to their common parent
    f. Who is the common parent here - app
    g. So cut the selected category from header and move it to app

```
 const [selectedCategory, setSelectedCategory] =
useState('electronics');
```

    h. What do we do if we need to pass them as values to our child components - props

```
<Header selectedCategory={selectedCategory}
setSelectedCategory={setSelectedCategory}/>
      <ProductList
selectedCategory={selectedCategory}/>
```

i. In my header component, i now accept the props

```
const Header = ({selectedCategory,
setSelectedCategory}) => {
```

j. How do we make the url dynamic friends

k.
```
const ProductList = ({selectedCategory}) => {
```

l.
```
    fetch(`http://localhost:3300/products/category/${selectedCategory}`)
```

7.

8. Dependency in useeffect
    a. Add an empty array in header
    b. selectedCategory in product list
        i. Have an empty dependency array in product list to see the diff
9. Add a loader
    a. Let us add a loading information while waiting for products

```
 const [loading, setLoading] = useState(false);
```

Update the loading when the data fetch begins

```
useEffect(() => {
   setLoading(true);

fetch(`http://localhost:3300/products/category/${selec
tedCategory}`)
```

```
      .then((res) => res.json())
      .then((json) => {
        setProducts(json);
        setLoading(false);
      })
      .catch((err) => {
        console.log(err);
        setLoading(false);
      });
```

In the jsx update

```
if(loading){
   return (
     <div className="loading">Loading ...</div>
   )
 }
```

b. understand what happens when the loading state changes
c. The component raises hand , hey something changed plz re-render
d. If true, loading jsx returns else the products

```
if (loading) {
   return (
     // <div className="loading">Loading ...</div>
     <div className="products">
       {Array(6)
         .fill()
         .map((_, index) => (
           <ShimmerCard key={index} />
         ))}
```

```
{/* {Array.from({ length: 6 }, (_, index) => (
        <ShimmerCard key={index} />
      ))} */}

    </div>

  );
}
```

10. Props drilling- Now let us move to adding items in the cart
    a. you need the same state in header and AddToCart component - cart state
    b. concept of props drilling
    c. Soln - context api
    d. context is a data layer around all our components.
    e. Someone top in the hierarchy provides a resource like my grandfather built a house in home town and now my father used it, I am using it and my coming generations will use.
    f. In production apps , we use another library called Redux / MobX but here we will see how react also gives an option help in our cause
    g. Pass the product in AddToCart in the ProductCard
    h.

```
<AddToCart product={product}/>
```

11. Context api
    a. Create context layer for our application

```
import { createContext, useContext } from "react";



const CartContext = createContext();

export const useCartContext = () =>
useContext(CartContext);
```

    b. useCartContext is a reusable function that we will export from here and this will be used in all our components to read from the context layer
    c. This is a custom hook
    d. Now we create a provider component like grandfather which will provide the resource

```
function CartContextProvider({ children }) {
 return (
   <CartContext.Provider value={{ name: "React" }}>
     {children}
   </CartContext.Provider>
 );
}

export const useCartContext = () =>
useContext(CartContext);

export default CartContextProvider;
```

e. Update the return function of App.js to wrap it in Context

```
<CartContextProvider>
     <div className="App">
```

f. Now the understanding is that all the child components should have access to the context

g. Let us test this in header component

```
const [data, setData] = useState([]);
 const cartValue = useCartContext();
 console.log("context in header ", cartValue);
```

h. Import the useCartContext

i.
```
import { useCartContext } from "../context/cart";
```

j.

k. Cart.js update Update cart.js to include global cart state object and functions to update the object

```
function CartContextProvider({ children }) {
 const [cart, setCart] = useState({});
 const addToCart = (product) => {
   setCart((prevCart) => {
       const newCart = {...prevCart}; // clone of prev
cart
       // if the item is alrady in the cart
       if(prevCart[product.id]){
           const newProduct = {...prevCart[product.id]};
           newProduct.quantity++;
           newCart[product.id] = newProduct;
```

```
        } else {
            newCart[product.id] = {
                ...product,
                quantity: 1
            }
        }
        return newCart;
    })
  }
  return (
    <CartContext.Provider value={{ name: "React" }}>
      {children}
    </CartContext.Provider>
  );
}
```

I. Two things to note here
    i. We dont update the state directly because first we need to check what is in the prev state. Whenever our next state depends on the prev state, then we use this way of updating state
    ii. Second is that we are cloning before updating
    iii. "In React, when we update something like our shopping cart, we don't directly change the original cart. Instead, we make a copy or 'clone' of it and then make our changes to this clone. By doing this, we're

changing the object's reference – think of it as giving it a new label.

iv. React is smart and keeps an eye on these labels. When it sees a new label, it knows something has changed. This helps React quickly identify what needs to be updated on the screen. Without this new label (or reference), React might not realize something has changed, and the screen wouldn't update as it should.

v. So, by cloning and changing the object reference, we're helping React notice the changes and update our page faster and more reliably."

```
const removeFromCart = (product) => {
  setCart((prevCart) => {
    const newCart = { ...prevCart };
    // I dont have the product
    // guard clauses
    if (!newCart[product.id]) return prevCart;

    // I have it but the qty is 1 - remove it
    if (newCart[product.id].quantity === 1) {
      delete newCart[product.id];
    } else {
      const newProduct = { ...prevCart[product.id] };
      newProduct.quantity -= 1;
      newCart[product.id] = newProduct;
    }
    return newCart;
  });
};
```

m. Now we pass the state and the methods in the context

```
   <CartContext.Provider value={{ cart, addToCart,
removeFromCart }}>
```

n. AddToCart update - Now let use these in the AddToCart

```
import { useCartContext } from "../context/cart";

const AddToCart = ({ product }) => {
 const { cart, addToCart, removeFromCart } =
useCartContext();

 const productInCart = cart[product.id];

 const handleAdd = () => {
   addToCart(product);
 };
 const handleRemove = () => {
   removeFromCart(product);
 };
```

o. Now we should be able to update items

p. But it does not show in the cart in header

q. So what do we do

r. Update header to read from context

i. Show the console in header the cart values

ii. Now add code to update the cart item count

```
const {cart} = useCartContext();
```

```
console.log("context in header ", cart);
const total = () => {
  let count = 0
  for(let key in cart){
    count += cart[key].quantity
  }
  return count
}
```

iii. Then update this count in jsx

```
      <span
className="cart-length">{total()}</span>
```

Redux

Redux is like the central management system of this restaurant. Here's how:

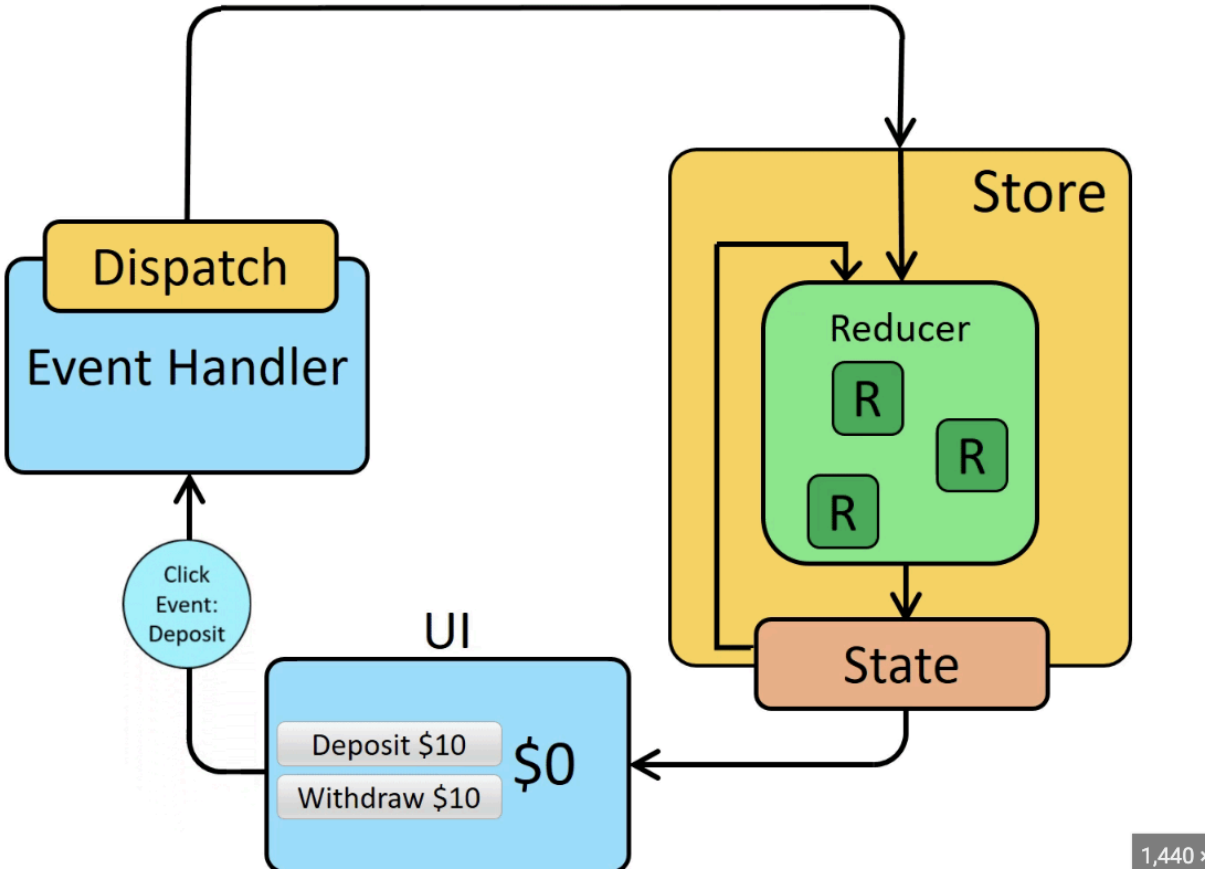
Single Source of Truth (State Store): In Redux, there's a single state store that's like the main order book in the restaurant. It keeps track of everything: what each table ordered, what needs to be cooked next, and which tables are waiting for their meals. This centralization makes it easier to manage and track the state of the entire restaurant.

Actions (Communication): When a customer orders something, a waiter writes down the order and brings it to the kitchen. In Redux,

these are 'Actions'. Actions are like the notes that tell the kitchen what needs to be done, like "cook pasta" or "prepare salad".

Reducers (Kitchen Processors): The kitchen (Reducers) receives these orders and knows exactly how to prepare them. Reducers take the current state of the restaurant (the entire order book) and an action (the new order), and they produce a new state (an updated order book). They follow specific recipes (rules) to ensure that the orders are processed in a consistent way.

Predictable State Management: Just like how a well-run restaurant operates predictably (customers order, the kitchen prepares, and waiters serve), Redux ensures that the state changes in your application are predictable and easy to track. This is great for debugging and understanding how data flows through your app.



1. Index.js
    a. Importing configureStore from Redux Toolkit:
    b. import { configureStore } from "@reduxjs/toolkit"; is where we bring in a function from the Redux Toolkit that helps us create our Redux store.
    c. Importing the cartReducer:
        i. import cartReducer from "./cart"; means we're getting a reducer function related to the cart feature of our application, usually defined in another file.
    d. Setting Up the Reducer:

e. In const reducer = { cart: cartReducer };, we're creating an object that maps the state slice name 'cart' to the cartReducer. This tells the store how to handle updates to the cart part of our state.
f. Creating the Store:
g. const store = configureStore({ reducer, middleware: [] }); creates the Redux store. It's set up with the reducer we defined, and an empty array for middleware (additional tools or enhancements for the store).
h. Exporting the Store:
    i. export default store; makes our configured store available for use in other parts of our application.

2. Cart.js
    a. Importing createSlice:
        i. import { createSlice } from "@reduxjs/toolkit"; imports a function that helps create a slice of the Redux state.
        ii. Creating a Cart Slice:
            1. const cartSlice = createSlice({...}); defines a new slice of the Redux state. This slice is specifically for the cart functionality in the application.
        iii. Slice Configuration:
            1. Inside createSlice, we configure the slice:
            2. name: "cart" sets the name of this slice as 'cart'.
            3. initialState: {} defines the initial state of the cart as an empty object.

4. reducers: {...} contains functions that define how the state can be updated.

iv. Reducers for Cart Operations:
    1. addToCart: Adds an item to the cart or increases its quantity if it's already in the cart.
    2. removeFromCart: Decreases the quantity of an item in the cart or removes it entirely if the quantity becomes zero.

v. Exporting Actions and Reducer:
    1. export const { addToCart, removeFromCart } = cartSlice.actions; exports the actions defined in the cart slice so they can be used elsewhere in the application.

vi. export default cartSlice.reducer; exports the reducer function for the cart slice, which will be used in the store configuration.

Integrating Redux

1. Npm i react-redux
2. In the app.js, import the store and the provider

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

3. Comment the cart context provider and add the PRovider

```
return (
   // <CartContextProvider>
```

```
   <Provider store={store}>
```

4. In the header.js, remove the context and add read from the redux

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

5. Read from cart slice

```
const Header = ({ selectedCategory, setSelectedCategory })
=> {
 const [data, setData] = useState([]);
 // const {cart} = useCartContext();
 const cart = useSelector((state) => state.cart);
```

6. Update add to cart component

```
import { useCartContext } from "../context/cart";
import { useSelector, useDispatch } from "react-redux";
import { addToCart, removeFromCart } from "../store/cart";
```

7. Update cart data store

```
const AddToCart = ({ product }) => {
 // const { cart, addToCart, removeFromCart } =
useCartContext();
 const cart = useSelector((state) => state.cart);
 const dispatch = useDispatch();
```

8. Update handlers

```
const handleAdd = () => {
   // addToCart(product);
   dispatch(addToCart(product));
 };
```

```
const handleRemove = () => {
  // removeFromCart(product);
  dispatch(removeFromCart(product));
};
```