React - 8 Context API and Redux-1

Agenda
Context Api
Integrate Context API in our app
Redux
Redux toolkit

Context API in React: The Central Library of Data

In React applications, we often have data that needs to be accessed by many components, not just the component where the data is created or managed. Traditionally, passing data from parent to child components (prop drilling) can become cumbersome and complex, especially when the data is needed by deeply nested components.

The Context API acts like the central library. It allows you to create a central place to store data (context) that can be easily accessed by any component, without the need to pass props through every level of the component tree.

How to Use Context API

Create a Context: First, you create a context using React.createContext(). This sets up a central place to store data.

Provide the Context: Use a Provider component to wrap the part of your component tree that needs access to the context. The Provider makes the context data available to all the components inside it.

Consume the Context: Components that need the data from the context can use the useContext hook or Consumer component to access it.

title: adding context API to our Application

What are the data and methods that were needed by different components for which we had to l lift state up and ended up with props drilling

With the help of context API we can keep watchList , add movieToWatchList and removeFromWatchList at one place. Let's see how it's done

1. create a directory inside src and name it context . create a file WatvhListContext.jsx.
2. What is the first step now to use the context
    a. Create the context

```
import {createContext} from 'React'


const WatchListContext = createContext()
```

3. Now we create a wrapper component that becomes the provider of this data (context)
    a. Any child component no matter how deeply nested they are will have access to this context

```
import { createContext } from "React";

const WatchListContext = createContext();

export default function WatchListContextWrapper({ children
}) {
 return <WatchListContext.Provider>{children}
</WatchListContext.Provider>;
}
```

4. Now we move all the handlers and effects from Movies.jsx to this component

```
import { createContext } from "React";

const WatchListContext = createContext();

export default function WatchListContextWrapper({ children }) {
 const [watchList, setWatchList] = useState([]);

 const addToWatchList = (movieObj) => {
   const updatedWatchlist = [...watchList, movieObj];
   setWatchList(updatedWatchlist);
   localStorage.setItem("movies", JSON.stringify(updatedWatchlist));
 };

 const removeFromWatchList = (movieObj) => {
   let filtredMovies = watchList.filter((movie) => {
     return movie.id != movieObj.id;
   });
   setWatchList(filtredMovies);
```

```
    localStorage.setItem("movies", JSON.stringify(filtredMovies));
  };

  useEffect(() => {
    let moviesFromLocalStorage = localStorage.getItem("movies");
    if (moviesFromLocalStorage) {
      setWatchList(JSON.parse(moviesFromLocalStorage));
    }
  }, []);

  return <WatchListContext.Provider>{children} </WatchListContext.Provider>;
}
```

5. Pass all the state and handler in value prop.

```
  <WatchListContext.Provider
    value={{ addToWatchList, removeFromWatchList, watchList,
setWatchList }}
  >
    {children}{" "}
  </WatchListContext.Provider>
```

6. we have wrapped the watchList and it's method into one Wrapper and when we wrap it on any Component, it can access all the things related to context API
7. Wrapping our components with this provider
8. In App.jsx , add the wrapper component

```
import "./App.css";
import Home from "./components/Home";
import WatchList from "./components/WatchList";
import NavBar from "./components/Navbar";
import { Routes, Route } from "react-router-dom";
import WatchListContextWrapper from "./context/WatchListContext";

function App() {
  return (
    <>
```

```
      <NavBar />
      <WatchListContextWrapper>
        <Routes>
          <Route path="/" element={<Home />} />
          <Route path="/watchlist" element={<WatchList />} />
        </Routes>
      </WatchListContextWrapper>
    </>
  );
}

export default App;
```

Utilizing context

In movie.jsx now you can access all the things through context . Let's see th syntax how it's done

In the Movies.jsx, update the imports

```
import axios from "axios";
import { useState, useEffect, useContext } from "React";
import Pagination from "./Pagination";
import MovieCard from "./MovieCard";
import { WatchListContext } from "../context/WatchListContext";
function Movies() {
```

```
 const { watchList, addToWatchList, removeFromWatchList } =
useContext(WatchListContext)
```

Just these three above steps now you have removed all the watchList logic from Movies component

Let's do the same for Watch list as well also remove removeFromwatchList Implementaion beacuse that is now coming from context API

Remove the state variable and now read it from context

```
function WatchList() {
 const { watchList, setWatchList } = useContext(WatchListContext)
```

Redux

In a React application, managing state can become complex as the application grows. Here are some reasons why we might need help from a state management library

1. Global State Management: When state needs to be shared across many components, managing it locally can become cumbersome. We need a centralized store where all the state lives, making it easier to share and manage state across the application.
2. Prop Drilling: Passing state and functions down through multiple levels of components (prop drilling) can make the code hard to manage and understand.
3. Performance Issues: When using Context API, any change in context value causes all the components that consume that

context to re-render. This can lead to performance issues in larger applications.

There are some additional benefits that we get when using libraries like Redux which we will discuss

We will be using reduxtoolkit as complete solution for as our state managment . It is the recommended way to use redux .

Let's see all the important facts you need to know about redux

1. Redux is a third-party javascript library for state management.
2. It can be integrated with every front-end framework of JavaScript.
3. We just need to install it.
4. It gives a feature known as store where all the states are stored.
5. It also provides a centralised state management feature with the help of a feature known as slice.

Analogy

Let us assume a supermarket, let us assume this shop has different sections: It sells different items like kitchen items, electronics , clothes and food items. and we have only one store owner

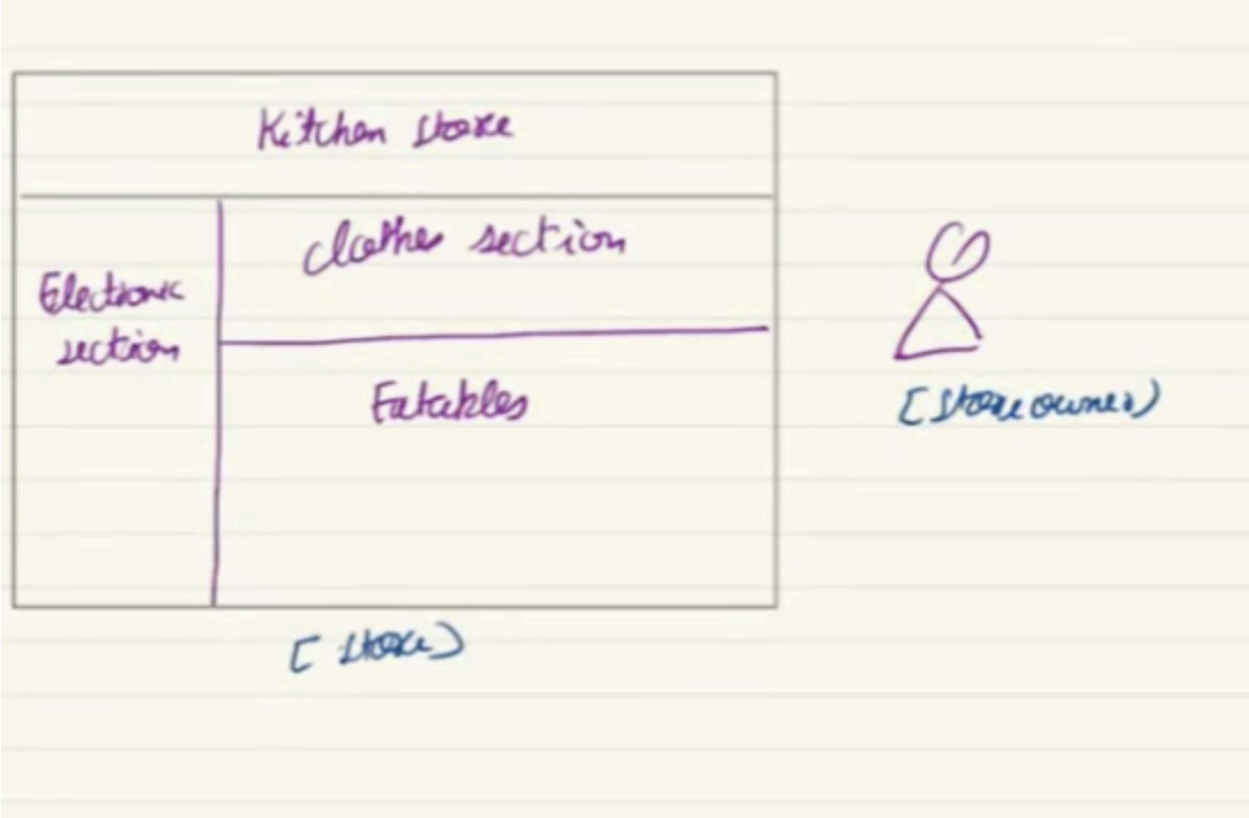
Usually you may have observed similar items are kept together

Kitchen section: For storing all the kitchen-related items.

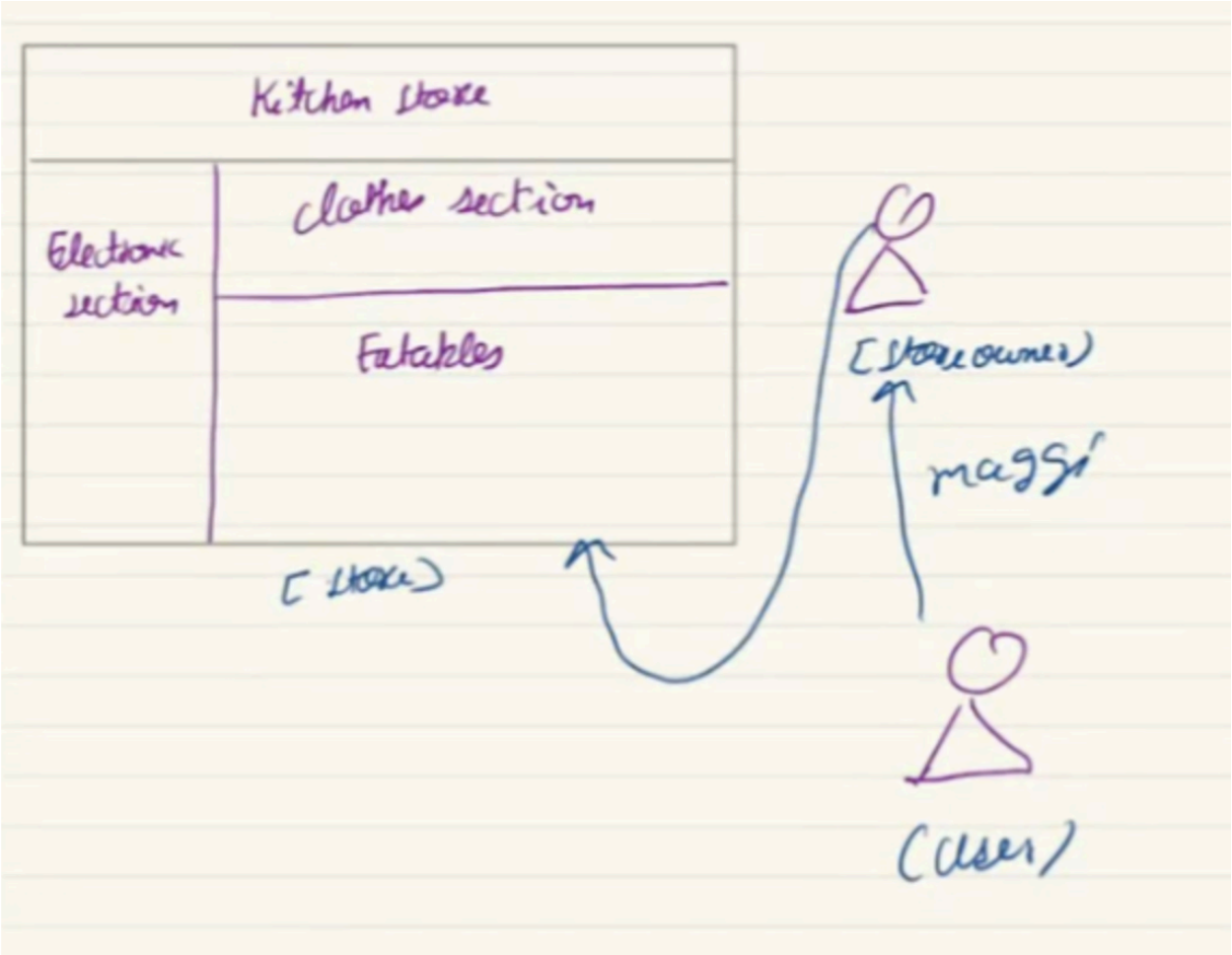
Electronics section

Clothes section

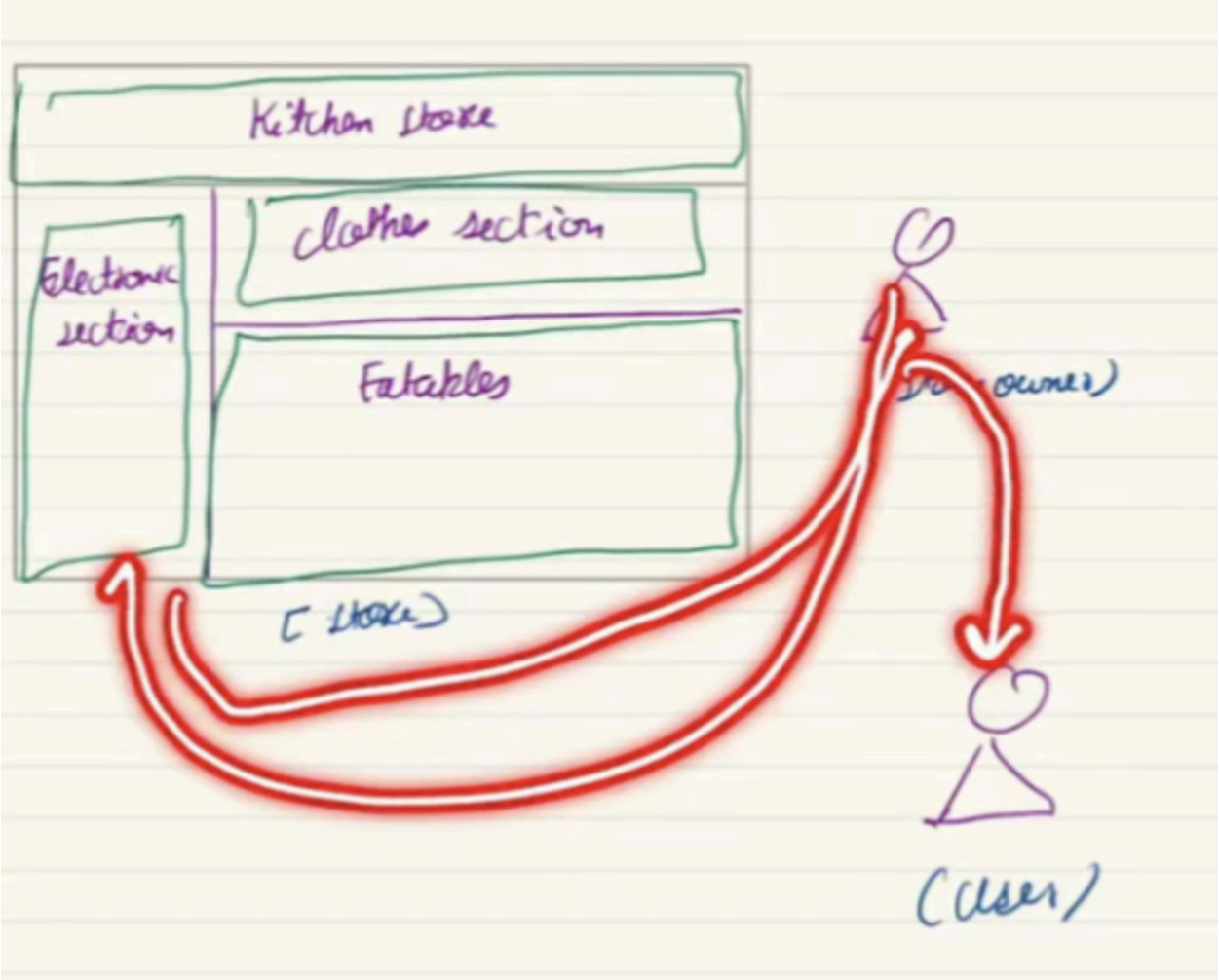
Food items



now a user arrives and asks for Maggie, then the owner will go to eatables and give you Maggie.



If a user asks for headphones, the owner will directly go to the electronics bring them and give them to the user.



The job of the store owner is to go to a particular section, bring material and give it to the user.

Take away

1. Redux store is similar to supermarket
2. sections are similar to slices that store the state for a particular feature
3. In redux, all the slices are combined to make the store.

4. user are the components that will use this store but and this but is most important -> store owner will lay ground rules how a component can set or get some value from the store
5. Again the user will never directly go to the section to fetch the items, it is the responsibility of the owner to go to the section bring the material and come back to the counter to give the item to the user.

Analogy -2

Imagine you're in charge of a large spaceship. This spaceship has many sections: the engine room, the navigation room, the crew quarters, and more.

Each section has its own controls and instruments. Now, if you need to change the ship's course, adjust the engine power, and update the crew's schedule, you have to go to each section and make the changes individually. This process is time-consuming and can lead to mistakes or miscommunications.

To solve this, you set up a central command center.

From this command center, you can control every section of the ship. All the data about the ship's status flows into the command center, and all the orders go out from there. This makes managing the spaceship much easier, faster, and more efficient.

Introduction to Redux

Store: The store is the command center where the state of your entire application lives. It holds the state and allows access to it, enables dispatching actions, and registers listeners.

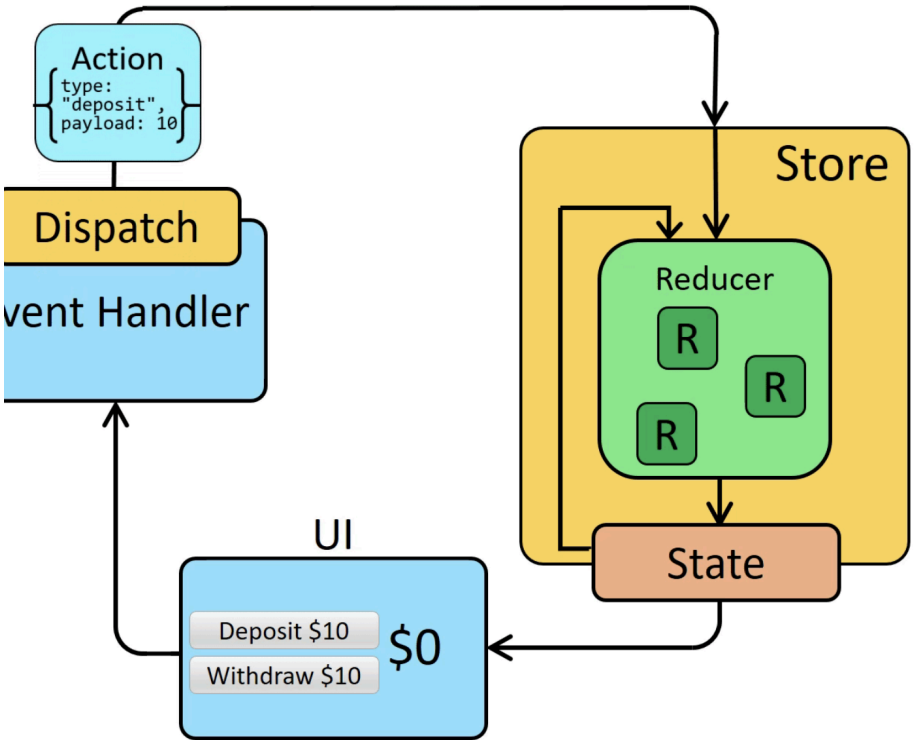
Actions: Actions are plain JavaScript objects that describe what happened. They are the only source of information for the store.

Reducers: Reducers specify how the application's state changes in response to actions. They are pure functions that take the previous state and an action, and return the next state.

Dispatch: Dispatch is the method used to send actions to the store. It's how we communicate with the store to update the state.

Go to this URL and see the flow. Scroll down where you get this image -

https://redux.js.org/tutorials/fundamentals/part-2-concepts-data-flow



Main Parts of Redux Toolkit

1. Slices:
    a. A slice is like a part of your application's state.
    b. It contains the state, actions, and reducers for a specific part of your application.
    c. Think of a slice as a "slice of the pie" where each slice represents a piece of your overall state.
2. Actions:
    a. Actions are objects that describe what happened in your app. For example, clicking a button might generate an action like { type: 'counter/increment' }.

    b. Actions tell Redux what to do but don't do the work themselves.
3. Reducers:
    a. Reducers are functions that take the current state and an action, and return a new state. They handle the actual state updates based on the action received.
    b. Think of reducers as the part of the app that knows how to do the work when an action says what happened.
4. Store:
    a. The store is the place where your app's state lives. It brings together the actions and reducers to manage the state.
    b. The store is like the central command center for your app's state.
5. Dispatch:
    a. Dispatch is the method used to send actions to the store. When you dispatch an action, you are telling Redux to update the state.

The Flow of Actions in Redux Toolkit

1. User Interaction: The user interacts with the UI (e.g., clicks a button).
2. Dispatch an Action: The UI component dispatches an action to the store.
3. Reducer Processes Action: The store sends the action to the appropriate reducer.

4. State Update: The reducer updates the state based on the action.
5. UI Update: The updated state causes the UI to re-render with the new state.

title: installation and integration

Open the terminal inside your application folder, and write the below commands:

npm install @reduxjs/toolkit react-redux

Redux Toolkit which is a set of tools and best practices designed to simplify Redux development. It includes utilities to help write Redux logic that is more concise and less error-prone

Implementation

1. Create a folder named redux
2. Setting Up the Store

```
// store.js
import { configureStore } from '@reduxjs/toolkit';
import counterReducer from './counterSlice';

const store = configureStore({
 reducer: {
   counter: counterReducer,
 },
});

export default store;
```

3. Creating the slice

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

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

export const { increment, decrement } = counterSlice.actions;
export default counterSlice.reducer;
```

Breaking Down the Code

1. *import { createSlice } from '@reduxjs/toolkit';*
    a. This line imports the createSlice function from Redux Toolkit. createSlice is a helper function that simplifies creating Redux slices.
    b. createSlice helps us create actions and reducers more easily and reduces boilerplate code.
2. Then we create a slice called counterSlice using the createSlice function.
    a. name: 'counter'

    i. This gives the slice a name. The name is used to identify the slice and its actions.
    ii. In this case, the slice is named 'counter'.
  b. initialState: { value: 0 }
    i. This defines the initial state of the slice. The state is an object with a single property value set to 0.
    ii. This means when the application starts, the counter value is 0.
  c. reducers
    i. This is an object that contains reducer functions. Reducers define how the state should change in response to actions.
  d. increment(state)
    i. This is a reducer function that increases the value in the state by 1.
  e. decrement(state)
    i. This is a reducer function that decreases the value in the state by 1.
3. *export const { increment, decrement } = counterSlice.actions;*
  a. This line exports the action creators increment and decrement that were generated by createSlice.
  b. By exporting these actions, other parts of your application can dispatch these actions to change the state.
  c. For example, you can dispatch increment to increase the counter value and decrement to decrease it.
4. *export default counterSlice.reducer;*

a. This line exports the reducer function created by createSlice.
b. The reducer is what actually updates the state when actions are dispatched.
c. By exporting the reducer, it can be used in your Redux store configuration.

4. Providing the Store to the App
    a. Update main.jsx

```
import React from "React";
import ReactDOM from "react-dom/client";
import App from "./App.jsx";
import "./index.css";
import { BrowserRouter } from "react-router-dom";
import { Provider } from "react-redux";
import store from "./redux/store.js";

ReactDOM.createRoot(document.getElementById("root")).render(
 <React.StrictMode>
   <Provider store={store}>
     <BrowserRouter>
       <App />
     </BrowserRouter>
   </Provider>
 </React.StrictMode>
);
```

5. Create a component and connect it to the store

```
// Counter.js
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { increment, decrement } from '../redux/counterSlice';
```

```
const Counter = () => {
 const count = useSelector((state) => state.counter.value);
 const dispatch = useDispatch();

 return (
   <div>
     <h1>Count: {count}</h1>
     <button className='p-3 m-3 border border-gray-400' onClick={() =>
dispatch(increment())}>Increment</button>
     <button className="p-3 m-3 border border-gray-400" onClick={() =>
dispatch(decrement())}>Decrement</button>
   </div>
 );
};

export default Counter;
```

title: Adding a Todo App in current App

1. Create TodoRedux.jsx

```
function TodoRedux() {
 const list = [];
 return (
```

```
    <>
      <h2>Todo</h2>
      <div style={{ display: "flex" }}>
        <div className="inputBox">
          <input type="text" />
          <button></button>
        </div>
        <div className="list">
          <ul>
            {list.map((task, idx) => {
              return <li key={idx}>{task}</li>;
            })}
          </ul>
        </div>
      </div>
    </>
  );
}

export default TodoRedux;
```

2. The first step for redux is creating the slice, creating todoSlice.js inside the redux folder, we will simply put value: "" and a list as initial state.

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


const todoSlice = createSlice({
    name:"toolbox",
    initialState:{
        value:"",
        todoList:["task 1", "taks 2"]
    },

});

export default todoSlice;
```

3. Now we add reducers

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



const todoSlice = createSlice({
   name:"toolbox",
   initialState:{
       value:"",
       todoList:["task 1", "taks 2"]
   },
   reducers:{
       setValue(state, action){
           console.log(action)
           state.value = action.payload;
       },
       addTask(state){
           state.todoList.push(state.value);
           state.value = "";
       },
   }

});

export default todoSlice.reducer;
export const {setValue, addTask} = todoSlice.actions;
```

4. Update the store.js

```
// store.js
import { configureStore } from '@reduxjs/toolkit';
import counterReducer from './counterSlice';
import todoReducer from './todoSlice';

const store = configureStore({
 reducer: {
   counter: counterReducer,
   todo: todoReducer,

 },
```

```
});

export default store;
```

5. Update the Todo.jsx to read from state

```
import {useSelector, useDispatch} from 'react-redux';
import {addTask, setValue} from "../redux/todoSlice"

function TodoRedux() {
 const dispatch = useDispatch();
 const handleChange = (e) => {
   dispatch(setValue(e.target.value));
 }
 const {value, todoList} = useSelector((state) => state.todo);

   const handleAdd = () => {
       dispatch(addTask(value));
   }
 return (
   <>
     <h2>Todo</h2>
     <div>
       <div className="inputBox">
         <input className='border border-gray-400' onChange={handleChange}
type="text" value={value}/>
         <button onClick={handleAdd}>Add Task</button>
       </div>
       <div className="list">
         <ul>
           {todoList.map((task, idx) => {
             return <li key={idx}>{task}</li>;
           })}
         </ul>
       </div>
     </div>
   </>
 );
}
```

```
export default TodoRedux;
```