Packages installed using npm get installed in node modules directory, also –g is used for specifying global installation

npx create-react-app client

this creates the react app in a folder called client]]

Command:

npm i react-redux @reduxjs/toolkit redux-persist react-dropzone dotenv formik yup react-router-dom@6 @mui/material @emotion/react @emotion/styled @mui/icons-material

@ means that the related packages fall under a common group

**1. State Management:**

**State** in programming refers to the data that describes the current condition of a system. In React applications, state typically represents the data that the UI is based on, such as user inputs, API responses, or the application's current mode.

**State Management** is the process of managing and updating this data (state) in a consistent and predictable way throughout the application. It involves:

* Storing the application's state in a centralized location.
* Providing mechanisms for components to read and update the state.
* Ensuring that changes to the state trigger appropriate updates to the UI.

**2. Redux:**

**Redux** is a popular library for managing application state in JavaScript applications, particularly those built with React. It provides a predictable and centralized way to manage state, making it easier to understand how data flows through the application.

**Key Concepts**:

* **Store**: Redux uses a single **store** to hold the entire state tree of the application. This makes it easy to access and modify the state from any part of the application.

**Actions**

* **What they are:** Actions are plain JavaScript objects.
* **Purpose:** They describe what happened in the application. They are essentially "messages" that are sent to the Redux store to tell it what should change.
* **Structure:** An action must have a **type** property (a string that describes the type of action) and can have additional data (called **payload**) to provide more information about the change.
* **Reducers**: Reducers specify how the application's state changes in response to actions. They are pure functions that take the current state and an action as arguments, and return the new state.
* **Immutable State**: Redux encourages immutability, meaning that state should not be directly mutated. Instead, new state objects are created based on the previous state and the changes specified by actions.

**3. Redux Persist:**

**Redux Persist** is an extension for Redux that allows you to persist the Redux state to storage, such as local storage or AsyncStorage (in React Native), and rehydrate it on application startup. This ensures that the state persists across page reloads or app restarts, providing a seamless user experience.

**Key Features**:

* **Automatic Serialization**: Redux Persist automatically serializes the Redux state to a string representation that can be stored in storage.
* **Configurable Storage**: You can configure Redux Persist to use different storage engines, such as local storage, AsyncStorage, or custom storage solutions.
* **Rehydration**: Redux Persist automatically rehydrates the Redux state from storage when the application starts, restoring the state to its previous state.
* **Configurable Subset**: You can specify a subset of the state to persist, allowing you to exclude sensitive or unnecessary data from being persisted.

**In Summary:**

* **State Management** involves managing and updating the data (state) of an application in a consistent and predictable way.
* **Redux** is a library that provides a predictable and centralized way to manage application state in JavaScript applications, particularly those built with React.
* **Redux Persist** is an extension for Redux that allows you to persist the Redux state to storage and rehydrate it on application startup, providing a seamless user experience.

Scenes refers to the different windows of the UI.

Components allows us to reuse stuff.

BASIC JAVASCRIPT INFO

**Default Values in JavaScript**

* **Undefined:** If a variable is declared but not initialized, its value is **undefined**.

javascript

Copy code

let myString; console.log(myString); // Output: undefined

* **Null:** **null** is a special value in JavaScript that represents the intentional absence of any object value. It has to be explicitly assigned.

javascript

Copy code

let myString = null; console.log(myString); // Output: null

FOR STATES IN REDUX:

const initialState ={

    mode: "light",

    user: null,//null reps intentional absence

    token: null, //authentication token

    posts:[]

};

In the context of Redux and frontend state management, the **posts** array will hold the data fetched from the backend, but it won't directly hold MongoDB models. Here's a breakdown of how it works:

**Frontend vs. Backend Models**

1. **Backend (MongoDB Models):**
   * MongoDB models are schema-based structures used to interact with a MongoDB database.
   * They define the structure of documents in the database, including fields and data types.
2. **Frontend (Redux State):**
   * The frontend state, managed by Redux, typically holds plain JavaScript objects or arrays.
   * These objects represent the data needed by the frontend, which is often fetched from a backend database like MongoDB via an API.

**How it Works in Practice**

1. **Fetching Data:**
   * The frontend makes an API call to fetch data from the backend. This data is often in JSON format.
   * The backend (Node.js/Express) retrieves data from MongoDB using Mongoose models, converts it to a JSON format, and sends it to the frontend.
2. **Storing Data in Redux:**
   * The fetched data is then stored in the Redux state. In this case, the **posts** array will be populated with plain JavaScript objects representing the posts.

INFORMATION ON STATES:

State can be represented as either an array or an object depending on scenario. New state is created with updated properties everytime. Actions contain the new parameter values that need to be reflected in the new state.

Example:  
const addPostAction = {

type: 'ADD\_POST',

payload: {

id: 1,

title: 'New Post',

content: 'This is a new post.'

}

};

const postsReducer = (state = [], action) => {

switch (action.type) {

case 'ADD\_POST':

return [...state, action.payload];

default:

return state;

}

};

This creates a new state (…state means new state contains all properties of old state, action.payload updates the new parameters as per the ones specified in the payload )

EXAMPLE 2:

const initialState = {

mode: "light",

user: null,

token: null,

posts: []

};

const addPostAction = {

type: 'ADD\_POST',

payload: {

id: 1,

title: 'New Post',

content: 'This is a new post.'

}

};

const rootReducer = (state = initialState, action) => {

switch (action.type) {

case 'SET\_MODE':

return {

...state,

mode: action.payload

};

case 'SET\_USER':

return {

...state,

user: action.payload

};

case 'SET\_TOKEN':

return {

...state,

token: action.payload

};

case 'ADD\_POST':

return {

...state,

posts: [...state.posts, action.payload]

};

default:

return state;

}

};

**CREATE SLICE**

* **createSlice Function**: It's a utility provided by Redux Toolkit to make Redux development easier.
* **Purpose**: It helps create Redux slices, which are bundles of Redux-related logic (reducers, action creators, initial state) for managing a specific part of your application state.
* **Simplification**: **createSlice** simplifies the process of defining Redux logic by handling the boilerplate code for reducers and action creators.
* **Usage**: You use **createSlice** to define a slice of your application state, specifying its initial state and defining reducer functions for handling actions that update that state.
* **Convenience**: It's convenient because you write less code compared to traditional Redux setup, making your codebase cleaner and more maintainable.

When you define a slice using Redux Toolkit's **createSlice** function, it returns an object with two properties: **reducer** and **actions**. The **actions** property contains all the action creators automatically generated by **createSlice**.

ACTION CREATORS:

export const authSlice= createSlice({

    name: "auth",

    initialState,

    reducers: {

        setMode: (state) =>{

            if(state.mode=="light")

                state.mode="dark";

            else

                state.mode="light";

        },

        setLogin : (state,action) =>{

            state.user=action.payload.user;

            state.token=action.payload.token;

        },

        setLogout: (state)=>{

            state.user=null;

            state.token=null;

        },

        setFriends: (state,action)=>{

            if(state.user){

                state.user.friends=action.payload.friends;

            }

            else{

                console.error("User non existent so can't update friends");

            }

        },

        setPosts:  (state,action)=>{

            state.posts=action.payload.posts;

        },

        setPost: (state,action)=>{

            const updatedPosts= state.posts.map((post)=>{

                if(post.\_id==action.payload.post\_id) return action.payload.post;

                return post;

                //maps through each post and if post id matches we update it with new post

                //if post is not the o ne we;re searching for we return old post

            });

            state.posts=updatedPosts;//update state with list of updated posts

        }

    }

});

In the context of Redux Toolkit's **createSlice**, action creators are automatically generated for each reducer function defined within the **reducers** object. These action creators are named after the keys you provide in the **reducers** object and are automatically bound to dispatch.

Here's how the action creators are generated and used in your code:

1. **Action Creators:**
   * For each reducer function (**setMode**, **setLogin**, etc.), an action creator with the same name is automatically generated.
   * These action creators are functions that accept any necessary payload data and return action objects.

export const setMode = createAction('auth/setMode');

export const setLogin = createAction('auth/setLogin');

export const setLogout = createAction('auth/setLogout');

export const setFriends = createAction('auth/setFriends');

export const setPosts = createAction('auth/setPosts');

export const setPost = createAction('auth/setPost');

**Usage of Action Creators:**

* These action creators can be imported and used elsewhere in your application to dispatch actions to the Redux store.

import { setMode, setLogin } from './authSlice';

// Dispatching actions

dispatch(setMode('dark'));

dispatch(setLogin({ user: 'username', token: 'token123' }));

IMP:

Let's break it down step by step:

1. **Action Creators:**
   * **authSlice.actions** refers to the object containing all the action creators automatically generated by Redux Toolkit's **createSlice** function. Each key in this object corresponds to a reducer function defined in your **createSlice** configuration.
2. **Destructuring:**
   * **export const { setMode, setFriends, setLogin, setLogout, setPost, setPosts } = authSlice.actions;**
   * This line uses object destructuring to extract individual action creators (**setMode**, **setFriends**, etc.) from the **authSlice.actions** object and export them as named exports.

INDEX.JS

WE IMPORT A BUNCH OF STUFF FROM REDUX PERSIST, REACT-REDUX YA GOTTA READ EM UP BRO

**REACT:**

React is a JavaScript library for building user interfaces. It allows you to create reusable UI components that manage their own state.

IN JSX WE ENCLOSE JAVASCRIPT COMPONENTS WITHIN CURLY BRACES