**MODULE: 13 React – Applying Redux**

**1.** **What is Redux?**

**Ans.** Redux is a popular JavaScript library used for managing the state of an application in a predictable and centralized manner. It is often used with frameworks such as React but can be used with any JavaScript framework or library.

The core concept of Redux revolves around a single immutable state tree. This means that the entire state of an application is stored in a single JavaScript object called the "store." The store represents the single source of truth for the entire application.

To modify the state, Redux relies on pure functions called "reducers." Reducers take the current state and an action as input and return a new state as output. Actions are plain JavaScript objects that describe the type of operation to be performed on the state.

Redux also introduces the concept of "actions" and "action creators." Actions are objects that represent an intention to change the state, while action creators are functions that create and return these action objects.

The state in Redux is not mutated directly; instead, a new state object is created every time an action is dispatched, and reducers produce a new state based on the current state and the dispatched action. This immutability and purity of functions make the state management predictable and helps with debugging and testing.

To connect Redux with a user interface, typically a React component, a binding library like "react-redux" is used. It provides a way to connect the components to the Redux store and access the state and dispatch actions as needed.

**2. What is Redux Thunk used for?**

**Ans.** Redux Thunk is a middleware for Redux that allows you to write action creators that return functions instead of plain action objects. It extends the capabilities of Redux by enabling the dispatch of asynchronous actions and handling side effects.

Redux Thunk enables you to write action creators that return functions instead of plain objects. These functions can have side effects, perform asynchronous operations, and dispatch multiple actions based on the result of those operations.

When Redux Thunk middleware is applied to the Redux store, it intercepts actions that are functions instead of objects. It passes the dispatch and getState functions of the Redux store as arguments to the function. This allows the function to dispatch actions asynchronously or conditionally based on the current state of the application.

**Ex:**

import { createStore, applyMiddleware } from 'redux';

import thunk from 'redux-thunk';

// Redux Thunk middleware is applied during the store creation

const store = createStore(

  rootReducer,

  applyMiddleware(thunk)

);

// An action creator that returns a function

const fetchData = () => {

  return (dispatch, getState) => {

    dispatch({ type: 'FETCH\_DATA\_REQUEST' });

    // Asynchronous operation, such as making an API call

    fetch('https://api.example.com/data')

      .then(response => response.json())

      .then(data => {

        dispatch({ type: 'FETCH\_DATA\_SUCCESS', payload: data });

      })

      .catch(error => {

        dispatch({ type: 'FETCH\_DATA\_FAILURE', payload: error });

      });

  };

};

// Dispatching the fetchData action

store.dispatch(fetchData());

In the example above, the fetchData action creator returns a function instead of an object. Within the function, an asynchronous operation (API call) is performed, and based on the result, different actions (FETCH\_DATA\_REQUEST, FETCH\_DATA\_SUCCESS, or FETCH\_DATA\_FAILURE) are dispatched.

**3. What is Pure Component? When to use Pure Component over Component?**

**Ans.** A Pure Component automatically implements a **shouldComponentUpdate** method, which performs a shallow comparison of the previous and current props and state. If there are no changes detected in the props or state, the component prevents unnecessary re-rendering, optimizing performance by avoiding unnecessary updates to the component tree.

When to use Pure Component over regular Component:

1. When the component's props and state are immutable: Pure Components are most effective when the props and state are immutable or don't change frequently. Since the shallow comparison only checks for changes in the references of props and state, mutating objects or arrays within props or state may not trigger a re-render.
2. When the component relies on pure functions or derives data from props: If a component only relies on pure functions or derives its rendered output solely from its props, using Pure Component can help eliminate unnecessary re-renders. Pure Components are particularly useful when the component's rendering depends on large data sets or complex calculations derived from props.
3. When performance optimization is crucial: Pure Components are especially beneficial in scenarios where performance optimization is crucial, such as large-scale applications or components that render frequently. By reducing unnecessary re-renders, Pure Components can improve the overall performance of the application.

**4. What is the second argument that can optionally be passed tosetState and what is its purpose?**

**Ans.** The purpose of the callback function is to perform additional operations or execute code that relies on the updated state. By using the callback, you can ensure that the code is executed at the right time, after the state has been successfully updated.

Here's an example to illustrate the usage of the callback function in **setState**:

**Ex:**

class MyComponent extends React.Component {

    constructor(props) {

      super(props);

      this.state = {

        count: 0

      };

    }

    handleClick = () => {

      this.setState({ count: this.state.count + 1 }, () => {

        console.log('State updated:', this.state.count);

        // Additional operations or code relying on the updated state can be executed here

      });

    };

    render() {

      return (

        <div>

          <p>Count: {this.state.count}</p>

          <button onClick={this.handleClick}>Increment</button>

        </div>

      );

    }

  }

In the example above, when the button is clicked, the **handleClick** method is called, which updates the state using **setState**. The second argument to **setState** is a callback function that logs the updated state to the console.

**5. Create a Table and Search data from table using React Js?**

**Ans.**

**Table.js**

import React, { useState } from 'react';

const Table = () => {

  const [data, setData] = useState([

    { id: 1, name: 'Roky', age: 19 },

    { id: 2, name: 'Jainam', age: 25 },

    { id: 3, name: 'preet', age: 27 },

    { id: 4, name: 'Alpesh', age: 39 }

  ]);

  const [searchTerm, setSearchTerm] = useState('');

  const [editData, setEditData] = useState(null);

  const handleSearch = (e) => {

    setSearchTerm(e.target.value);

  };

  const handleSave = (id, name, age) => {

    if (id) {

      // Edit existing data

      const updatedData = data.map((item) =>

        item.id === id ? { id, name, age } : item

      );

      setData(updatedData);

      setEditData(null);

    } else {

      // Add new data

      const newId = data.length + 1;

      const newData = { id: newId, name, age };

      setData([...data, newData]);

    }

  };

  const handleEdit = (id) => {

    const itemToEdit = data.find((item) => item.id === id);

    setEditData(itemToEdit);

  };

  const handleDelete = (id) => {

    const updatedData = data.filter((item) => item.id !== id);

    setData(updatedData);

  };

  const filteredData = data.filter((item) =>

    item.name.toLowerCase().includes(searchTerm.toLowerCase())

  );

  return (

    <div>

      <input

        type="text"

        placeholder="Search by name"

        value={searchTerm}

        onChange={handleSearch}

      />

      <table>

        <thead>

          <tr>

            <th>ID</th>

            <th>Name</th>

            <th>Age</th>

            <th>Actions</th>

          </tr>

        </thead>

        <tbody>

          {filteredData.map((item) => (

            <tr key={item.id}>

              <td>{item.id}</td>

              <td>{item.name}</td>

              <td>{item.age}</td>

              <td>

                <button onClick={() => handleEdit(item.id)}>Edit</button>

                <button onClick={() => handleDelete(item.id)}>Delete</button>

              </td>

            </tr>

          ))}

        </tbody>

      </table>

      <div>

        <h2>{editData ? 'Edit Data' : 'Add Data'}</h2>

        <input

          type="text"

          placeholder="Name"

          value={editData ? editData.name : ''}

          onChange={(e) => setEditData({ ...editData, name: e.target.value })}

        />

        <input

          type="number"

          placeholder="Age"

          value={editData ? editData.age : ''}

          onChange={(e) =>

            setEditData({ ...editData, age: parseInt(e.target.value) })

          }

        />

        <button

          onClick={() =>

            handleSave(editData ? editData.id : null, editData.name, editData.age)

          }

        >

          Save

        </button>

      </div>

    </div>

  );

};

export default Table;

**App.js**

import React from 'react';

import Table from './Table';

const App = () => {

  return (

    <div>

      <h1>Redux Table</h1>

      <Table />

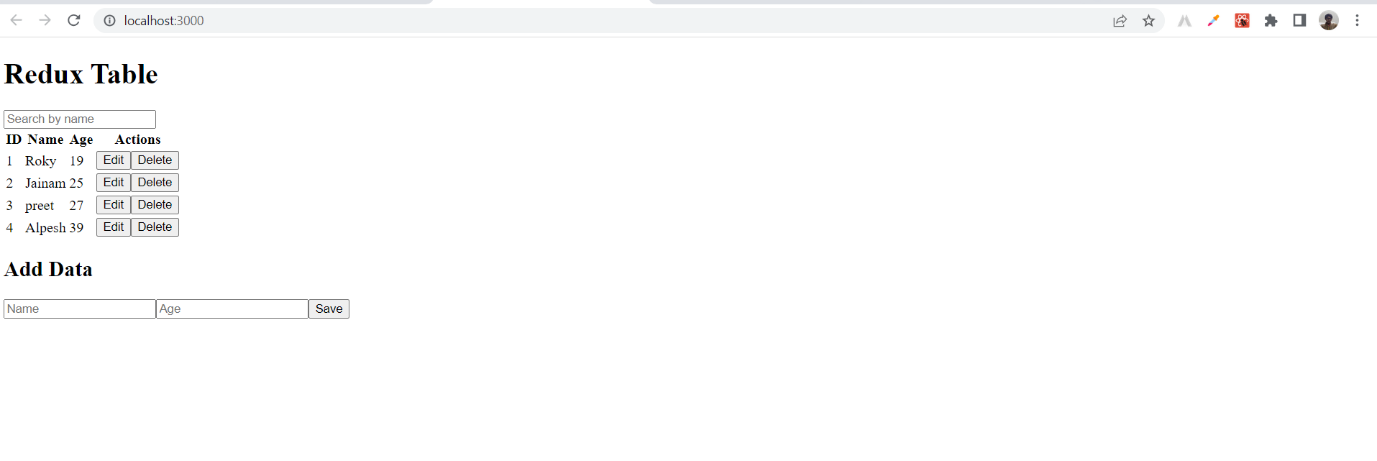
    </div>

  );

};

export default App;

**Output:**

****

);