React – Applying Redux

1. What is Redux?

=>Redux is a predictable state container for JavaScript applications, primarily used with libraries like React or Angular for managing the application state. It provides a centralized store to manage the state of your entire application in a predictable and consistent way.

Here are some key concepts of Redux:

1. **Store**: Redux maintains the state of an entire application in a single JavaScript object called the store. This store holds the application state and provides methods to access and update it.
2. **Actions**: Actions are plain JavaScript objects that represent an intention to change the state. They are the only source of information for the store. Actions must have a **type** property indicating the type of action being performed. They can also have additional data called payload.
3. **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 a new state based on that action.
4. **Dispatch**: Dispatch is a method of the store object used to dispatch actions to the reducers. When an action is dispatched, the store passes it to the reducer to calculate the new state.
5. **Selectors**: Selectors are functions used to extract specific pieces of data from the store's state. They help in decoupling the components from the structure of the state and make the state access more efficient.
6. What is Redux Thunk used for?

=>Redux Thunk is a middleware for Redux that allows you to write action creators that return a function instead of an action object. This function receives the **dispatch** and **getState** functions as arguments, enabling it to perform asynchronous operations, such as API calls, and dispatch actions based on the results.

const fetchData = () => ({

type: 'FETCH\_DATA',

});

const fetchData = () => {

return (dispatch, getState) => {

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 });

});

};

};

1. What is Pure Component? When to use Pure Component over Component?

=>A Pure Component in React is a class component that extends **React.PureComponent** instead of **React.Component**. The main difference between a Pure Component and a regular Component lies in how they handle shouldComponentUpdate method.

1. **Pure Component**:
   * A Pure Component implements a shallow comparison of props and state in the **shouldComponentUpdate** method provided by React.
2. **Component**:
   * A regular Component doesn't implement a shallow comparison in **shouldComponentUpdate** by default.

When to use Pure Component over Component:

🡺Performance Optimization:

If your component's render method is expensive or if it's rendering frequently even when the props or state haven't changed, using Pure Components can help improve performance.

🡺Simple State or Props:

Pure Components are suitable when the props and state of the component are simple values or immutable data structures.

If the props or state are deeply nested objects or arrays that frequently change, using Pure Components might not provide significant performance benefits, as

🡺Avoid Manual shouldComponentUpdate:

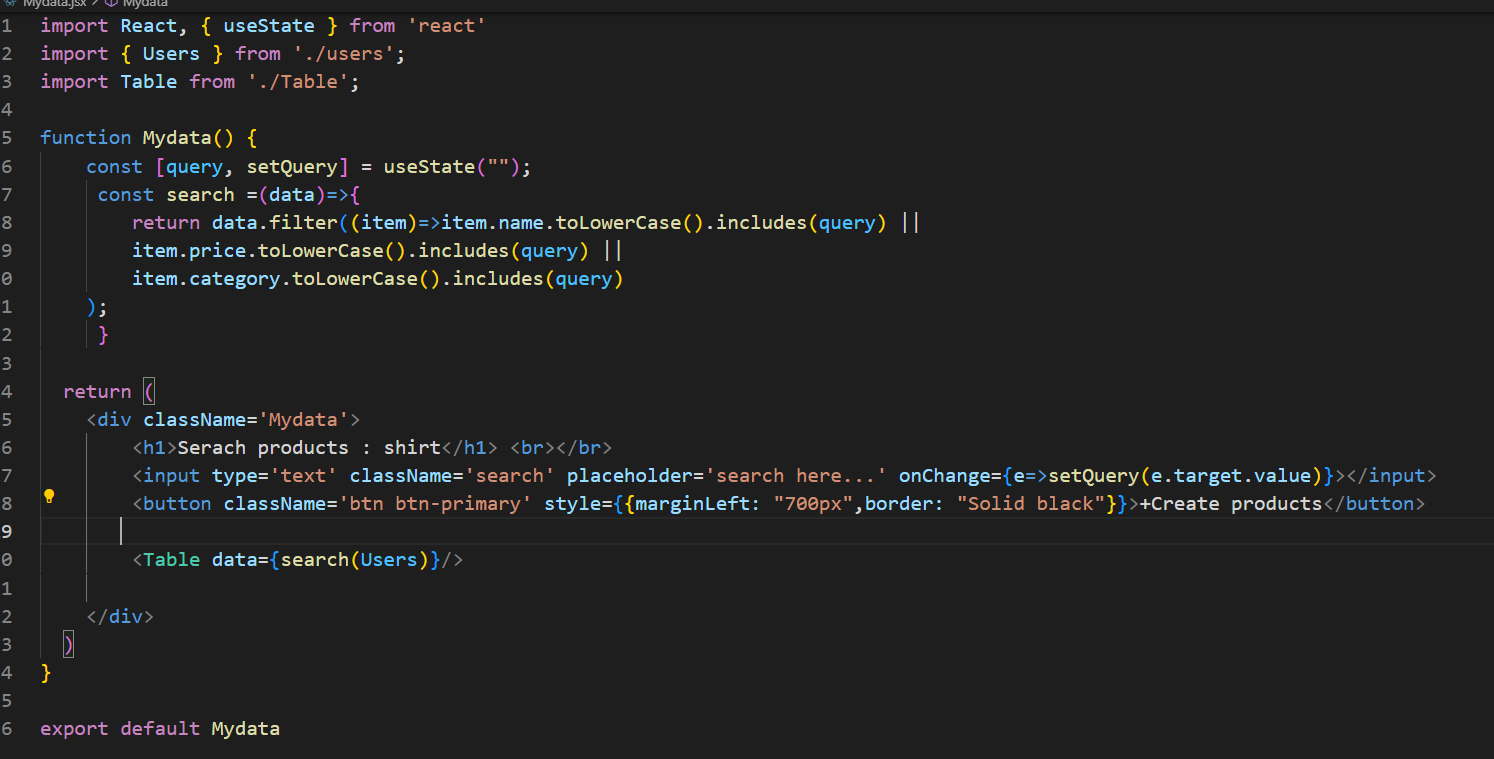
If you find yourself frequently implementing custom shouldComponentUpdate methods in your components to optimize performance, consider using Pure Components instead.

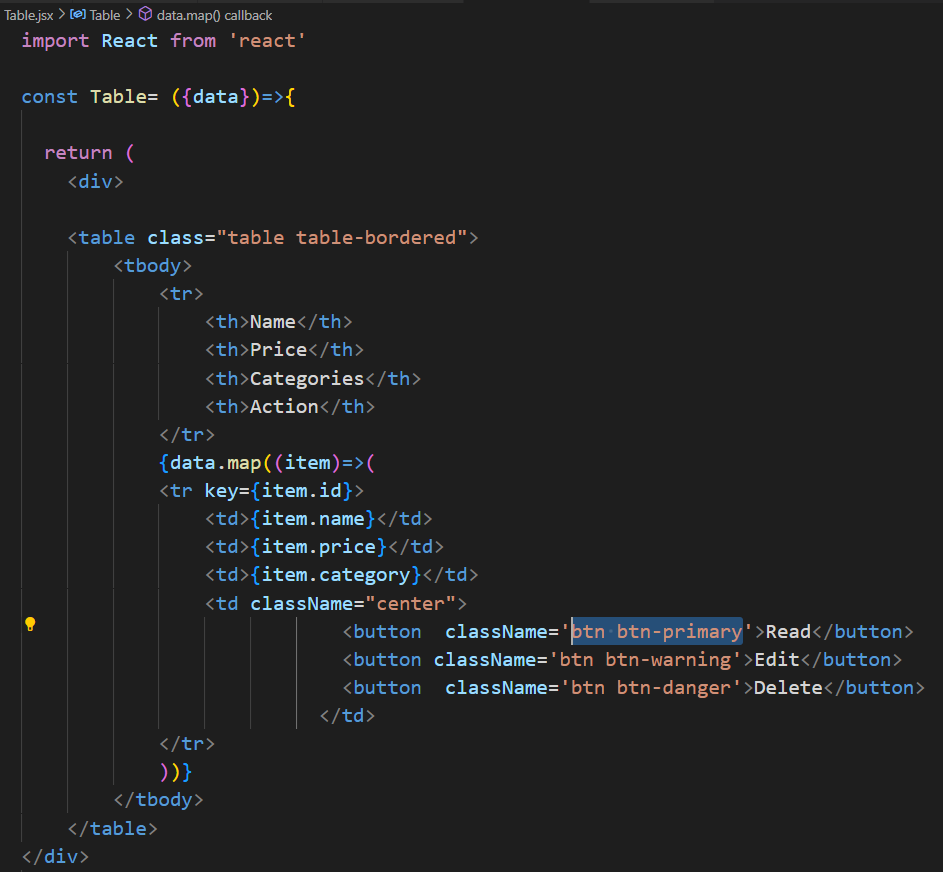
1. What is the second argument that can optionally be passed tosetState and what is its purpose?

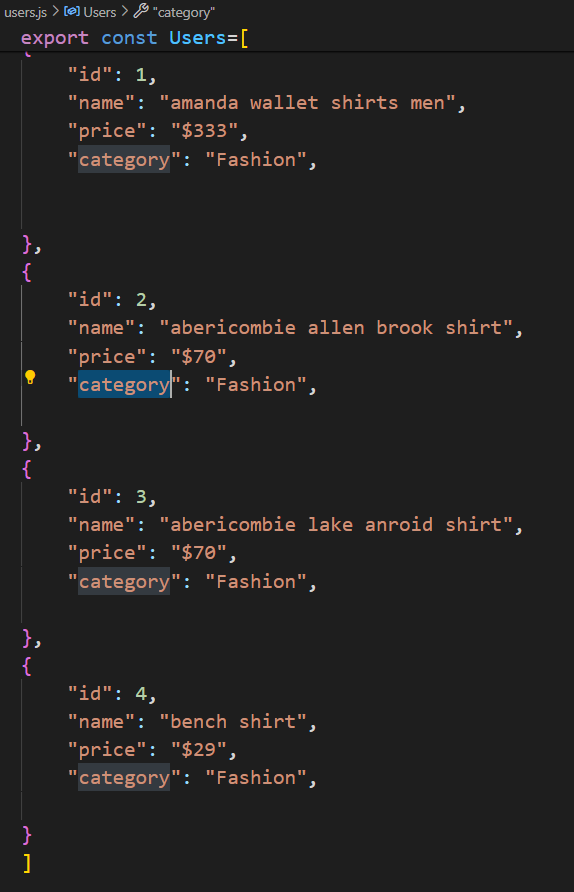
=>In React, the **setState** method can optionally take a second argument, which is a callback function. This callback function is executed once **setState** has been completed and the component has been re-rendered. Its purpose is to perform additional logic after the state has been updated and the component has re-rendered.

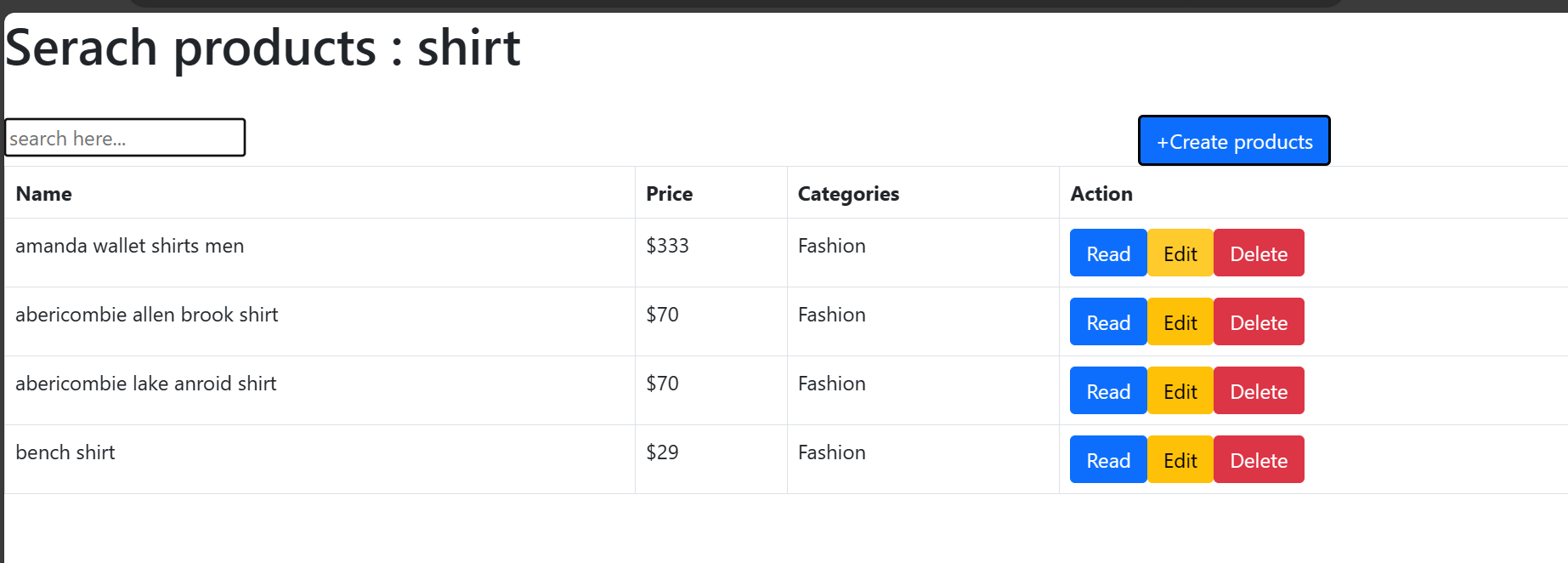
setState(updater, [callback])

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









1. Create Login registration with CRUD Application using API (Redux)

=>Creating a full-fledged Login/Registration application with CRUD functionality using Redux and an API involves several steps, including setting up the Redux store, creating action creators, reducers, API calls, and UI components. Below is a simplified example to guide you through the process. Keep in mind that this example assumes you have some basic knowledge of Redux, React, and RESTful APIs.

1. **Setup Redux Store**:
   * Install Redux and Redux Thunk: **npm install redux react-redux redux-thunk**
2. **Create Action Types**:
   * Define action types for login, registration, and CRUD operations (create, read, update, delete).
3. **Create Action Creators**:
   * Write action creators for each action type. These action creators will dispatch actions to the Redux store.
4. **Create Reducers**:
   * Implement reducers to handle state changes based on dispatched actions.
5. **API Calls**:
   * Write functions to interact with your backend API for authentication (login and registration) and CRUD operations.
6. **UI Components**:
   * Develop React components for login, registration, and CRUD operations.
7. **Connect Redux to Components**:
   * Connect Redux store to your React components using **connect** from **react-redux**.
8. **Dispatch Actions**:
   * Dispatch actions from your components when user interacts with UI elements.

// ActionTypes.js

export const LOGIN\_SUCCESS = 'LOGIN\_SUCCESS';

export const REGISTER\_SUCCESS = 'REGISTER\_SUCCESS';

export const ADD\_ITEM = 'ADD\_ITEM';

export const DELETE\_ITEM = 'DELETE\_ITEM';

// Define other action types as needed...

// ActionCreators.js

import \* as types from './ActionTypes';

export const loginSuccess = (user) => ({

type: types.LOGIN\_SUCCESS,

payload: user

});

export const registerSuccess = (user) => ({

type: types.REGISTER\_SUCCESS,

payload: user

});

export const addItem = (item) => ({

type: types.ADD\_ITEM,

payload: item

});

export const deleteItem = (itemId) => ({

type: types.DELETE\_ITEM,

payload: itemId

});

// Define other action creators as needed...

// Reducers.js

import \* as types from './ActionTypes';

const initialState = {

user: null,

items: []

// Define other initial state properties as needed...

};

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

switch (action.type) {

case types.LOGIN\_SUCCESS:

case types.REGISTER\_SUCCESS:

return {

...state,

user: action.payload

};

case types.ADD\_ITEM:

return {

...state,

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

};

case types.DELETE\_ITEM:

return {

...state,

items: state.items.filter(item => item.id !== action.payload)

};

// Handle other action types...

default:

return state;

}

};

export default reducer;

// API.js

const apiUrl = 'http://api.example.com';

export const loginUser = async (credentials) => {

// Implement login API call

};

export const registerUser = async (userData) => {

// Implement register API call

};

export const addItemToServer = async (itemData) => {

// Implement API call to add item

};

export const deleteItemFromServer = async (itemId) => {

// Implement API call to delete item

};

// Implement other API functions...

// App.js (or any other component)

import React from 'react';

import { Provider } from 'react-redux';

import store from './store';

import Login from './Login';

import Registration from './Registration';

import ItemList from './ItemList';

import ItemForm from './ItemForm';

const App = () => {

return (

<Provider store={store}>

<div>

<h1>Login/Registration with CRUD</h1>

<Login />

<Registration />

<ItemList />

<ItemForm />

</div>

</Provider>

);

};

export default App;