**React - Assignment**

**Module 7 : React - Applying Redux**

**Qs 1 :- What is Redux?**

**Ans :**

Redux is a predictable state container for JavaScript apps, most commonly used with libraries like React or Angular for managing application state. It helps you write applications that behave consistently, run in different environments (client, server, and native), and are easy to test.

At its core, Redux maintains the entire state of your application in a single immutable state tree (object), which can't be changed directly. When something changes, a new object is created (using actions and reducers) instead of mutating the existing state. This makes the state changes predictable and easier to understand and debug.

Redux consists of three main parts:

**1. Actions :** These are payloads of information that send data from your application to your Redux store. They are plain JavaScript objects that must have a `type` property indicating the type of action being performed. Actions are typically created using action creator functions.

**2. Reducers :** Reducers specify how the application's state changes in response to actions sent to the store. A reducer is a pure function that takes the previous state and an action, and returns the next state. It should never modify the previous state, but instead return a new state object.

**3. Store :** The store is the object that brings actions and reducers together. The store holds the application state, allows access to the state via `getState()`, allows state to be updated via `dispatch(action)`, and registers listeners via `subscribe(listener)`.

Redux is commonly used in React applications to manage the application's state in a more organized and scalable way, especially in large applications where managing state can become complex. However, it can be used with other frameworks or even vanilla JavaScript.

**Qs 2 :- What is Redux Thunk used for?**

**Ans :**

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 can perform asynchronous operations and then dispatch actions based on the result of those operations. This is particularly useful for handling asynchronous logic such as making AJAX requests, timeouts, or conditional dispatching of actions.

**Here's a basic example of how Redux Thunk works:**

// This is a thunk

const fetchData = () => {

return (dispatch, getState) => {

// You can perform asynchronous operations here

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

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

.then(data => {

// Dispatch an action with the fetched data

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

})

.catch(error => {

// Dispatch an action if there's an error

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

});

};

};

// This thunk can be dispatched like a regular action

store.dispatch(fetchData());

In this example, fetchData is a thunk that returns a function. When this thunk is dispatched, Redux Thunk intercepts it and invokes the inner function with dispatch and getState as arguments. This allows you to dispatch multiple actions, perform conditional dispatching, or perform other side effects (like logging) before or after dispatching actions.

**Qs 3 :- What is Pure Component? When to use Pure Component over Component?**

**Ans :**

In React, a `PureComponent` is a class component that extends `React.PureComponent` instead of `React.Component`. The main difference between `PureComponent` and `Component` is how they handle the `shouldComponentUpdate` method.

**1. PureComponent :** When a `PureComponent` is used, it performs a shallow comparison of the current props and state with the next props and state in the `shouldComponentUpdate` method. If the props and state have not changed, `shouldComponentUpdate` returns `false`, preventing the component from re-rendering. This can improve performance by avoiding unnecessary re-renders.

**2. Component :** On the other hand, a regular `Component` always returns `true` from the `shouldComponentUpdate` method, which means that the component will re-render whenever `setState` is called, even if the props and state have not changed. This can lead to performance issues in some cases, especially for deeply nested components or components with complex rendering logic.

**When to use `PureComponent` over `Component` :**

Use `PureComponent` when :

You want to optimize performance by avoiding unnecessary re-renders.

Your component's `render` method depends only on the props and state that are primitive values or immutable objects.

You don't need to implement a custom `shouldComponentUpdate` method.

**Use `Component` when :**

You need to implement a custom `shouldComponentUpdate` method to perform a deep comparison of props and state.

You want more control over when the component should re-render, even if the props and state have not changed.

You are working with mutable data structures in props or state that cannot be compared with a shallow comparison.

In general, if your component meets the criteria for using `PureComponent` and you want to optimize performance, it's a good idea to use `PureComponent` instead of `Component`. However, if you need more control over the re-rendering behavior or if you need to implement a custom `shouldComponentUpdate` method, then `Component` may be more suitable.

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

**Ans :**

In React, the second argument that can optionally be passed to setState is a callback function that will be executed after the setState operation has been completed and the component has been re-rendered. This callback is commonly used to perform actions that depend on the updated state or to execute code after the component has been re-rendered.

Here's an example of how the second argument callback can be used:

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

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

});

In this example, the setState method is used to increment the count state by 1. The second argument is a callback function that logs the updated count to the console after the component has been re-rendered with the new state.

Using the callback can be useful in scenarios where you need to perform additional operations after updating the state, such as updating the DOM, making API calls, or triggering animations based on the updated state.