MODULE: 13 React – Applying Redux

Que - What is Redux?

Sol –

Redux is an open-source JavaScript library used to manage application state. React uses Redux for building the user interface. It was first introduced by Dan Abramov and Andrew Clark in 2015.

React Redux is the official React binding for Redux. It allows React components to read data from a Redux Store, and dispatch Actions to the Store to update data. Redux helps apps to scale by providing a sensible way to manage state through a unidirectional data flow model. React Redux is conceptually simple. It subscribes to the Redux store, checks to see if the data which your component wants have changed, and re-renders your component.

Redux was inspired by Flux. Redux studied the Flux architecture and omitted unnecessary complexity.

- Redux does not have Dispatcher concept.

- Redux has an only Store whereas Flux has many Stores.

- The Action objects will be received and handled directly by Store.

Why use React Redux?

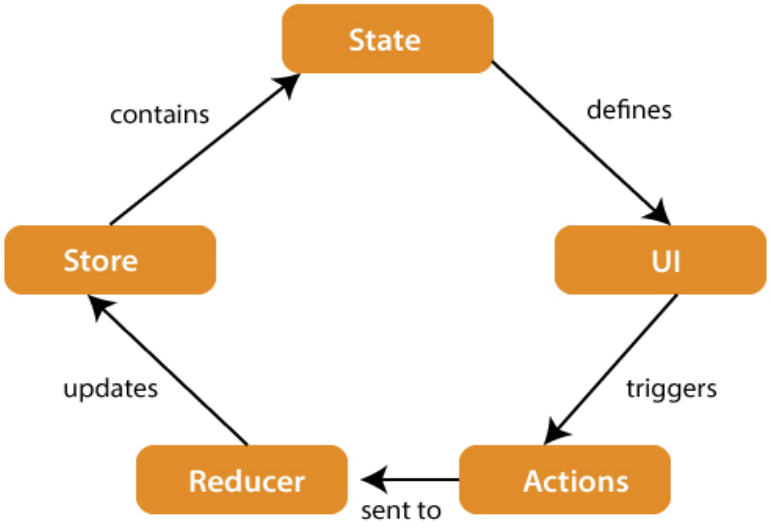
The main reason to use React Redux are:

- React Redux is the official UI bindings for react Application. It is kept up-to-date with any API changes to ensure that your React components behave as expected.

- It encourages good 'React' architecture.

- It implements many performance optimizations internally, which allows to components re-render only when it actually needs.

Redux Architecture :



The components of Redux architecture are explained below.

STORE: A Store is a place where the entire state of your application lists. It manages the status of the application and has a dispatch(action) function. It is like a brain responsible for all moving parts in Redux.

ACTION: Action is sent or dispatched from the view which are payloads that can be read by Reducers. It is a pure object created to store the information of the user's event. It includes information such as type of action, time of occurrence, location of occurrence, its coordinates, and which state it aims to change.

REDUCER: Reducer read the payloads from the actions and then updates the store via the state accordingly. It is a pure function to return a new state from the initial state.

Que - What is Redux Thunk used for?

Sol –

Redux Thunk is a middleware for Redux, a library commonly used with React, that allows you to write asynchronous logic in your Redux actions. In Redux, actions are typically synchronous and represent simple state changes. However, in real-world applications, you often need to deal with asynchronous operations like fetching data from a server or interacting with APIs. This is where Redux Thunk comes into play.

Redux Thunk enables you to dispatch functions as actions instead of just plain objects. These functions are referred to as "thunks". Thunks have access to the dispatch and getState functions, which allows them to perform asynchronous operations and dispatch regular actions as needed. This is particularly useful for managing asynchronous operations in a more organized and centralized manner.

Here's how Redux Thunk works:

1. Dispatching Thunks: Instead of dispatching a regular action object, you dispatch a thunk function. This thunk function can contain asynchronous code, such as making API requests using libraries like `axios` or `fetch`.

2. Asynchronous Logic: Within the thunk function, you can perform asynchronous operations. These operations could involve fetching data from an API, posting data, or any other asynchronous task that requires time to complete.

3. Dispatching Regular Actions: As the asynchronous operations complete, you can dispatch regular actions based on the results. These actions represent the different states that your asynchronous operation can be in, such as success, failure, or loading.

By using Redux Thunk, you can keep your asynchronous logic separate from your components, making them more focused on rendering UI and interacting with the state. This separation of concerns helps in keeping your codebase organized and easier to maintain. Redux Thunk also integrates well with the rest of the Redux ecosystem, making it a popular choice for handling asynchronous operations in Redux applications.

To use Redux Thunk in your Redux store, you need to apply it as middleware during store setup. This middleware intercepts actions before they reach the reducers, allowing you to handle thunks and asynchronous logic before eventually dispatching regular actions that update the state.

Que -What is Pure Component? When to use Pure Component over Component?

Sol –

In React, both `Component` and `PureComponent` are base classes that you can use to create your own custom components. The key difference between them lies in how they handle component updates and re-rendering.

1. Component:

The `Component` class is the base class for creating React components. When you create a component by extending `Component`, it implements a default behavior for the `shouldComponentUpdate` method. This method is responsible for determining whether a component should re-render when its props or state change.

By default, the `shouldComponentUpdate` method in a regular `Component` performs a shallow comparison of the new props and state with the previous props and state. If there is any difference, the component re-renders, which can sometimes lead to unnecessary re-renders if the props or state contain complex data structures.

2. PureComponent:

The `PureComponent` class, on the other hand, is an extension of the `Component` class that provides a performance optimization out of the box. When you create a component by extending `PureComponent`, it automatically implements a `shouldComponentUpdate` method that performs a shallow comparison of both props and state.

The key benefit of `PureComponent` is that it can prevent unnecessary re-renders by automatically comparing the current props and state with the previous ones. If no changes are detected in these shallow comparisons, the component won't re-render. This can improve the performance of your application by reducing the number of rendering operations.

When to use `PureComponent` over `Component`:

Use `PureComponent` when:

1. You want to optimize performance: If you know that your component's re-rendering is expensive and you want to minimize unnecessary re-renders, `PureComponent` can be a good choice.

2. Your component's props and state are mostly primitive values or simple data structures: Since `PureComponent` performs shallow comparisons, it works best when props and state can be compared easily using JavaScript's equality operators.

Use `Component` when:

1. You need custom control over rendering logic: If you need to perform custom checks beyond shallow comparisons to determine whether a component should re-render, you should use a regular `Component` and implement your own `shouldComponentUpdate` logic.

2. Your component's rendering logic is not impacted by frequent re-renders: In some cases, the performance gain from using `PureComponent` might be negligible, especially if your component's rendering is lightweight and not noticeably impacted by re-rendering.

In summary, use `PureComponent` when you want a performance optimization out of the box and your component's props and state can be effectively compared using shallow comparisons. Use `Component` when you need more control over rendering logic or when the performance gain from `PureComponent` is not significant for your specific use case.

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

Sol –

In React, the `setState` function is used to update the state of a component. It can take an optional second argument, which is a callback function that gets executed after the state has been updated and the component has re-rendered.

Here's the syntax for using `setState` with the optional callback:

this.setState(newState, callback);

The purpose of the optional callback is to perform actions that need to be executed after the state update is complete and the component has been re-rendered. This can be useful in scenarios where you need to perform some action that relies on the updated state or the component's updated UI.

For example, consider the following scenario where you want to log a message after updating the state:

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0,
    };
  }

  incrementCount = () => {
    this.setState({ count: this.state.count + 1 }, () => {
      console.log("State updated:", this.state.count);
    });
```

```
  };

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={this.incrementCount}>Increment</button>
      </div>
    );
  }
}
```

In this example, the callback function provided to `setState` is executed after the state has been updated and the component has re-rendered. It logs the updated count value to the console.

Using the callback can be particularly useful when you need to ensure that your action is performed with the updated state, as state updates in React are asynchronous and might not be immediately reflected in subsequent lines of code.

Keep in mind that while the callback can be helpful in certain situations, you should be cautious not to perform heavy computations or side effects within it, as it will be executed on every state update and could potentially affect the performance of your application.

Que - Create a Table and Search data from table using React Js?

Sol -