MODULE: 13 React – Applying Redux

1. What is Redux?

Ans :- Redux is a predictable state container for JavaScript applications, primarily used with frameworks like React or Angular for managing application state. It serves as a centralized store to manage the entire state of your application, making state mutations predictable by enforcing strict rules that ensure state can only be modified in a predictable fashion.

Here are the key concepts in Redux:

1. \*\*Store\*\*: The store is a single JavaScript object that holds the entire state of the application.

2. \*\*Actions\*\*: Actions are plain JavaScript objects that represent an intention to change the state. They are dispatched to the store.

3. \*\*Reducers\*\*: Reducers specify how the application's state changes in response to actions. They are pure functions that take the previous state and an action, and return the new state.

4. \*\*Dispatch\*\*: Dispatching an action is the process of sending an action object to the Redux store. This is how you initiate a state change.

5. \*\*Selectors\*\*: Selectors are functions used to extract specific pieces of data from the store state, which can then be used by React components.

6. \*\*Middleware\*\*: Middleware provides a third-party extension point between dispatching an action and the moment it reaches the reducer. It is commonly used for logging, crash reporting, or asynchronous operations like fetching data.

Redux is especially useful in large-scale applications where the state management can become complex. It encourages a unidirectional data flow and helps in debugging and testing applications by providing a predictable way to track changes in the state over time.

While Redux can be initially complex due to its concepts and boilerplate code, it becomes very powerful as applications scale up in size and complexity.

1. 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 then perform asynchronous operations and dispatch actions based on the results of those operations. Here's why Redux Thunk is used and how it works:

* Purpose of Redux Thunk :-

1. \*\*Handling Asynchronous Logic\*\*: Redux by itself only handles synchronous actions, where an action creator returns an action object immediately. However, many real-world applications need to perform asynchronous tasks like API calls or timers before dispatching actions. Redux Thunk enables this asynchronous behavior .

2. \*\*Delayed Actions\*\*: Redux Thunk allows you to dispatch actions with a delay or conditionally dispatch actions based on the result of asynchronous operations.

3. \*\*Complex Side Effects\*\*: Thunks are often used to encapsulate complex side effects like handling network requests, interacting with a browser API, or coordinating multiple synchronous dispatches.

* How Redux Thunk Works :-

- \*\*Thunk Function\*\*: Instead of creating an action object directly, you return a function from your action creator. This function receives `dispatch` and ’get-State’ as arguments, which allows it to dispatch actions itself and to access the current state of the Redux store.

- \*\*Async Logic Inside Thunk\*\*: Inside the thunk function, you can perform asynchronous operations, such as fetching data from an API using `fetch` or `axios`, setting timers using `set-Timeout`, or interacting with browser APIs like `local-Storage`.

- \*\*Dispatching Actions\*\*: Once the asynchronous operation is complete, you can dispatch regular synchronous actions based on the result of the operation. For example, you might dispatch a `FETCH\_SUCCESS` action if the data is fetched successfully, or a `FETCH\_FAILURE` action if an error occurs.

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

Ans :- A Pure Component in React is a class component that extends `React.-Pure-Component` instead of `React.-Component`. The main difference between `Pure-Component` and `Component` lies in how they handle `should-Component-Update`.

* Pure Component :-

1. \*\*Automatic `should-Component-Update`\*\*: `Pure-Component` implements a `should-Component-Update` method with a shallow prop and state comparison. This means it automatically checks if there are changes in props or state by performing a shallow comparison (comparing references).

2. \*\*Performance Optimization\*\*: Since `Pure-Component` implements `should-Component-Update` with a shallow comparison, it prevents unnecessary renders when the props and state of the component haven't changed. This can lead to performance optimizations, especially in large and complex applications where rendering can be expensive.

3. \*\*When to Use\*\*: Use `Pure-Component` when your component's `render` function is expensive and you want to optimize performance by avoiding unnecessary renders. It's particularly useful for components that rely purely on props and state to render and don't have their own internal state or rely on context changes.

* Component

1. \*\*Manual `should-Component-Update`\*\*: Components that extend `React.-Component` require you to explicitly define the `should-Component-Update` method if you want to optimize rendering based on prop or state changes. Without it, the component will re-render whenever `set-State` is called or when parent components re-render.

2. \*\*Flexibility\*\*: `Component` gives you more control over when the component should re-render. You can implement `should-Component-Update` to perform deep comparisons on props and state, or use additional logic to determine if an update is necessary based on other conditions.

3. \*\*Stateful Components\*\*: If your component manages its own internal state using `this.-state`, or relies on other factors beyond props and state to determine rendering behavior, then `Component` might be more appropriate since it allows you to control the update logic explicitly.

* When to Use Pure Component over Component :-

- \*\*Purely Prop-Driven Components\*\*: Use `Pure-Component` when your component is primarily driven by props and its rendering depends only on prop changes. This ensures that the component re-renders only when necessary based on shallow prop comparisons.

- \*\*Performance Critical Applications\*\*: In performance-critical applications where rendering optimizations are crucial, `Pure-Component` can be a good choice to reduce unnecessary renders and improve overall application performance.

- \*\*Functional Components with Hooks\*\*: For functional components that use hooks like `use-State` and `use-Effect`, you cannot use `Pure-Component`. Instead, you should use `React.-memo` to achieve similar performance optimizations by memoizing the functional component based on its props.

In summary, `Pure-Component` is useful when you want to optimize rendering performance by automatically handling prop and state comparisons to prevent unnecessary renders. However, if your component has complex logic for determining when it should update, or if it manages its own state, using `Component` with a manually defined `should-Component-Update` might be more appropriate.

1. What is the second argument that can optionally be passed to-set-State and what is its purpose?

Ans :- The second argument that can optionally be passed to `set-State` in React is a callback function. This callback function is invoked once `set-State` has been completed and the component has been re-rendered. Its primary purpose is to perform any additional logic that needs to be executed after the state has been updated and the component has re-rendered.

- `updater`: This can be an object (when you want to update state based on the previous state) or a function (when you need to access the current state and props to compute the next state).

- `callback`: Optional. This is a function that will be invoked after the state has been updated and the component has been re-rendered.

* Purpose of the callback function:

1. \*\*Execution after State Update\*\*: The callback function passed to `set-State` is called after React has finished applying the state changes and re-rendering the component. This ensures that any code inside the callback runs in the updated state context.

2. \*\*Accessing DOM Elements\*\*: Sometimes, you might need to perform operations that require accessing DOM elements after they have been updated due to state changes. Placing this logic in the callback ensures that it executes at the right time, after the DOM has been updated.

3. \*\*Triggering Side Effects\*\*: If you have side effects that should only occur after state has been updated and rendered (such as logging, analytics tracking, or fetching data based on new state), the callback is an appropriate place to place this code.