React – Applying Redux

Q1: What is Redux?

Answer:-
Redux is an open-source JavaScript library used for managing the state of an application in a predictable way. It is commonly used with React, a popular JavaScript library for building user interfaces. Redux helps you manage the state of your application by providing a centralized store that holds the entire state of your application.

In Redux, the state of your application is kept in a single store, which is a plain JavaScript object. The only way to change the state is by dispatching actions. Actions are plain JavaScript objects that describe what happened in your application. Reducers are functions that specify how the state changes in response to actions. Reducers calculate the new state based on the previous state and the action being dispatched.

One of the key benefits of Redux is its ability to manage complex state logic in large applications. It provides a clear and predictable way to manage the state, making it easier to debug and test your application. Redux also enables powerful features like time-travel debugging, where you can move backward and forward through the state of your application to understand what happened at different points in time.

Q2: What is Redux Thunk used for?

Answer:-
Redux Thunk is a middleware for Redux that allows you to write asynchronous logic that interacts with the Redux store. In Redux, actions are typically plain JavaScript objects that describe events that have occurred in the application. These actions are processed synchronously by reducers, which can update the state accordingly.

However, in real-world applications, you often need to deal with asynchronous operations, such as making API requests, reading from a database, or performing a timeout. Redux Thunk comes into play in such scenarios. It enables you to write action creators that return functions instead of plain action objects. These functions can perform asynchronous operations and dispatch actions based on the results of those operations.

Q3: What is Pure Component? When to use Pure Component over Component?

Answer:-
In React, a `PureComponent` is a class component that automatically implements the `shouldComponentUpdate` lifecycle method with a shallow prop and state comparison. This means that a `PureComponent` performs a shallow comparison of its current and next props and state to determine if it should re-render. If the props and state have not changed shallowly, the component will not re-render, thus optimizing performance by preventing unnecessary renders.

When to use `PureComponent` over a regular `Component` class depends on the specific use case. Here are some guidelines to help you decide:

Use `PureComponent` When:

1. Simple Props and State: If your component's props and state are simple (e.g., primitives or immutable data structures), using `PureComponent` can be beneficial. It works well when your props and state are deeply nested objects or arrays, as long as their references do not change when the data inside them changes.

2. Performance Optimization: When you want to optimize performance and prevent unnecessary renders, especially in components that render frequently or are part of long lists. By avoiding re-renders when nothing has changed, you save CPU and memory resources.

3. Functional Components with Hooks: In modern React applications, functional components with Hooks are becoming more prevalent. You can achieve similar performance optimizations in functional components by using the `React.memo` higher-order component or the `useMemo` Hook for memoization. However, if you prefer class components or are working with class-based codebases, `PureComponent` is a good option.

Use `Component` When:

1. Complex State or Props Logic: If your component relies on complex logic inside the `shouldComponentUpdate` method (beyond simple prop and state comparisons), or if you're dealing with mutable data structures and need more control over when the component should re-render, you might want to use a regular `Component`.

2. Force Re-renders: Sometimes, you might want to force a re-render of the component even if its props and state haven't changed. In such cases, using a regular `Component` and manually calling `this.forceUpdate()` can be appropriate. This approach is rare and should be used with caution, as it can lead to performance issues if overused.

Q4: What is the second argument that can optionally be passed tosetState and what is its purpose?

Answer:-
The setState function can take an optional second argument, which is a callback function that will be executed after the setState operation is completed and the component is re-rendered. This callback function allows you to perform actions or trigger code that relies on the updated state, ensuring that you are working with the most recent state data.

Syntax of setState:- this.setState(newState, () => {}

The primary purpose of this callback function is to handle side effects or perform actions that depend on the updated state. For example, you might want to make an API request after updating the state to fetch additional data, or you might want to perform certain calculations based on the updated state values.

Using the callback function ensures that you are working with the latest state data, as React batches state updates for performance reasons. State updates are asynchronous, so if you rely on this.state immediately after calling setState, it might not reflect the updated state value. By placing the code inside the callback function, you ensure that it runs after the state is guaranteed to be updated and the component is re-rendered.

Q5:- Create a Table and Search data from table using React Js?

Answer:-
```
import React, { useState } from 'react';

const DataTable = ({ data }) => {
  return (
    <table>
      <thead>
        <tr>
          <th>Name</th>
          <th>Age</th>
          <th>Role</th>
        </tr>
      </thead>
      <tbody>
        {data.map((item, index) => (
          <tr key={index}>
            <td>{item.name}</td>
            <td>{item.age}</td>
            <td>{item.role}</td>
          </tr>
        ))}
      </tbody>
    </table>
  );
};
```

```
export default DataTable;


import React from 'react';

const Search = ({ onSearch }) => {
  return (
    <div>
      <input type="text" placeholder="Search by name" onChange={(e) =>
onSearch(e.target.value)} />
    </div>
  );
};

export default Search;


import React, { useState } from 'react';
import DataTable from './DataTable';
import Search from './Search';

const App = () => {
  const initialData = [
    { name: 'Alice', age: 25, role: 'Developer' },
    { name: 'Bob', age: 30, role: 'Designer' },
    { name: 'Charlie', age: 28, role: 'Manager' },
    // ...more data items
  ];

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

  const handleSearch = (term) => {
    setSearchTerm(term);
    const filteredData = initialData.filter((item) =>
      item.name.toLowerCase().includes(term.toLowerCase())
    );
    setData(filteredData);
  };

  return (
    <div>
      <h1>Table with Search</h1>
      <Search onSearch={handleSearch} />
      <DataTable data={data} />
    </div>
  );
};

export default App;
```

Q6: Create Login registration with CRUD Application using API (Redux)?

Answer:-

```
import axios from 'axios';

export const loginUser = (userData) => (dispatch) => {
};

export const registerUser = (userData) => (dispatch) => {
};

export const logoutUser = () => (dispatch) => {
};


const initialState = {
  isAuthenticated: false,
  user: {},
  token: null,
};

const authReducer = (state = initialState, action) => {
  switch (action.type) {
    default:
      return state;
  }
};

export default authReducer;


import axios from 'axios';

export const fetchItems = () => (dispatch) => {
};

export const createItem = (itemData) => (dispatch) => {
};

export const updateItem = (itemId, itemData) => (dispatch) => {
};

export const deleteItem = (itemId) => (dispatch) => {
};



const initialState = {
  items: [],
};
```

```
const crudReducer = (state = initialState, action) => {
  switch (action.type) {
      default:
      return state;
  }
};

export default crudReducer;
```

```
import { combineReducers } from 'redux';
import authReducer from './authReducer';
import crudReducer from './crudReducer';

const rootReducer = combineReducers({
  auth: authReducer,
  crud: crudReducer,
});

export default rootReducer;
```

```
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import rootReducer from './reducers/rootReducer';

const store = createStore(rootReducer, applyMiddleware(thunk));

export default store;
```