CipherSchools

Class 9 ReactJS

Redux-toolkit

What is Redux?

Redux is a state management library for JavaScript applications. It helps manage and centralize application state in a predictable way.

- Originally designed for React, but can be used with any JS framework.
- It uses a single source of truth (store).
- Data flow is unidirectional (one way).

Why Redux?

React already has state management using useState, useContext, and useReducer. So why Redux?

When useContext + useReducer is enough:

- Small to medium apps.
- Shared state is limited to a few components.
- State updates are not complex.
- No need for middleware, devtools, or persistence.

When Redux is a better choice:

You need:

- Global shared state across large components.
- Devtools support (time travel debugging).
- Middleware (e.g. for logging, async logic).

- State persistence across refreshes.
- Better state architecture for teams.

Example Use Cases:

- Large forms
- Authentication tokens
- Shopping cart in e-commerce
- Video player states
- Real-time chat states

What is Redux Toolkit?

Redux Toolkit (RTK) is the official, recommended way to use Redux.

Benefits:

- Reduces boilerplate
- Built-in immer for immutability
- Easy setup with configureStore
- Includes utilities like createSlice, createAsyncThunk
- Good DX (Developer Experience)

  immer: powerful JavaScript library that lets you work with immutable state in a simple, mutable-looking way.

Key Redux Concepts

Term	Meaning
Store	The global state container. Created using configureStore.

Slice	A portion of the state with its own reducers and actions. Created using createSlice.
useSelector	React hook to access data from the Redux store.
useDispatch	React hook to dispatch actions to update the state.

Redux Counter Example using Redux Toolkit

Step 1: Install Redux dependencies

npm install @reduxjs/toolkit react-redux

Step 2: Create store.js

```
// store.js
import { configureStore } from '@reduxjs/toolkit';
import counterReducer from './counterSlice';

const store = configureStore({
  reducer: {
    counter: counterReducer,
  },
});

export default store;
```

Step 3: Create a counterSlice.js

```
// counterSlice.js
import { createSlice } from '@reduxjs/toolkit';

const counterSlice = createSlice({
```

```
  name: 'counter',
  initialState: { value: 0 },
  reducers: {
    increment: (state) => {
      state.value += 1;
    },
    decrement: (state) => {
      state.value -= 1;
    },
    incrementByAmount: (state, action) => {
      state.value += action.payload;
    },
  },
});

export const { increment, decrement, incrementByAmount } =
counterSlice.actions;
export default counterSlice.reducer;
```

Step 4: Provide Redux store to React App

```
// index.js or main.jsx
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import store from './store';
import App from './App';

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
);
```

Step 5: Create Counter Component

```
// Counter.js
```

```
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { increment, decrement, incrementByAmount } from './counterSlice';

const Counter = () => {
  const count = useSelector((state) => state.counter.value);
  const dispatch = useDispatch();

  return (
    <div>
      <h1>Counter: {count}</h1>
      <button onClick={() => dispatch(increment())}>+1</button>
      <button onClick={() => dispatch(decrement())}>-1</button>
      <button onClick={() => dispatch(incrementByAmount(5))}>+5</button>
    </div>
  );
};

export default Counter;
```

Summary of Important Redux Toolkit Terms

Term	Explanation
configureStore()	Creates the Redux store with good defaults.
createSlice()	Combines reducers + actions for a feature.
useSelector()	Read data from the store.
useDispatch()	Sends actions to update state.