State Management (Redux, Redux-Toolkit or Recoil)

Question 1: What is Redux, and why is it used in React applications? Explain the core concepts of actions, reducers, and the store.

Ans.

Redux is a state management library used mainly with React (but it works with any JS framework).
It helps you manage and share application state in a predictable, centralized way, especially in large or complex applications.

Instead of passing data through many components (prop drilling), Redux stores the state in one global store that any component can access.

Why is Redux used in React applications?

Redux is useful when:

- **Many components need the same data**
- State becomes **too complex** for useState / useContext
- You want **predictable state updates**
- You need **easy debugging & time-travel debugging**
- Large-scale apps (e.g., dashboards, e-commerce, admin panels)

Problems Redux solves:

- Prop drilling
- Uncontrolled state changes
- Difficult debugging
- Inconsistent app behavior

Core Concepts of Redux

Redux is based on **three main concepts**:

1. Actions

What is an Action?

An action is a plain JavaScript object that describes what happened in the application.

It does not change the state, it only tells Redux what to do.

Structure of an Action:

```
{
  type: "INCREMENT",
  payload: 1
}
```

Example:

```
const increment = () => {
  return {
    type: "INCREMENT"
  };
};
```

Key points:

- Must have a type

- type is usually a string
- payload is optional (data sent to reducer)

2. Reducers

What is a Reducer?

A reducer is a pure function that:

- Takes the current state
- Takes an action
- Returns a new state

Reducers are the only place where state changes happen.

Syntax:

```
const reducer = (state, action) => {

  return newState;

};
```

Example:

```
const initialState = { count: 0 };


const counterReducer = (state = initialState, action) => {

  switch (action.type) {

    case "INCREMENT":

      return { count: state.count + 1 };
```

```
    case "DECREMENT":
      return { count: state.count - 1 };

    default:
      return state;
  }
};
```

Rules of reducers:

- Must be **pure**
- Must not mutate state
- Always return a new state object

3. Store

What is the Store?

The store is the single source of truth that:

- Holds the entire application state
- Allows state access via getState()
- Allows updates via dispatch(action)
- Registers reducers

Example:

```
import { createStore } from "redux";

const store = createStore(counterReducer);
```

How Redux Works (Flow) :

UI → dispatch(action)

↓

Reducer

↓

New State

↓

Store

↓

UI updates

Redux in React (Brief)

Common libraries used:

- **redux**
- **react-redux**

Key hooks:

useSelector() // read state

useDispatch() // send actions

Example:

```
const count = useSelector(state => state.count);

const dispatch = useDispatch();
```

dispatch({ type: "INCREMENT" });

Advantages of Redux

Centralized state
Predictable updates
Easy debugging (Redux DevTools)
Scales well for large apps

When NOT to use Redux?

Small apps
Simple state management
Local component state only

In such cases, useState or useContext is enough.

Question 2: How does Recoil simplify state management in React compared to Redux?

Ans.

Recoil is a React-first state management library that aims to make global state feel as simple as local state, especially when compared to Redux.

What is Recoil?

Recoil is a state management library for React developed by Meta (Facebook).
It works naturally with React hooks and is designed to handle shared and derived state with minimal boilerplate.

How Recoil Simplifies State Management Compared to Redux

1.No Boilerplate (Biggest Advantage)

Redux

To update state, you need:

- Action
- Action type
- Reducer
- Store
- Dispatch

```
dispatch({ type: "INCREMENT" });
```

Recoil

Just update state like useState:

```
const [count, setCount] = useRecoilState(countState);

setCount(count + 1);
```

2. **Atom-Based State (Instead of One Big Store)**

Redux

- Single **global store**
- All state lives in one object
- Large reducers over time

Recoil

- State is split into **atoms**

- Each atom is an independent piece of state

```
import { atom } from "recoil";

export const countState = atom({
  key: "countState",
  default: 0,
});
```

3. **Derived State with Selectors (Simpler than Redux)**

Redux

- Derived data handled using:
    - Reselect
    - Complex logic inside reducers

Recoil

- **Selectors** are built-in
- Automatically recompute when dependencies change

```
import { selector } from "recoil";

export const doubleCount = selector({
  key: "doubleCount",
  get: ({ get }) => {
```

```
  return get(countState) * 2;
 },
});
```

4. **React Hook–Friendly**

Redux

- Requires react-redux
- Concepts like connect, mapStateToProps
- Extra abstraction

Recoil

- Uses **native React hooks**
- Feels like useState / useEffect

```
const value = useRecoilValue(countState);
```

5. **No Prop Drilling (Same Benefit, Less Work)**

Both Redux and Recoil solve prop drilling, but:

- Redux → global store + dispatch
- Recoil → atoms accessible anywhere

Recoil is simpler for **component-level shared state**.

6. **Asynchronous State Is Easier**

Redux

Needs middleware:

- Redux Thunk
- Redux Saga

Recoil

Async state handled via **async selectors**

```
const userData = selector({
  key: "userData",
  get: async () => {
    const res = await fetch("/api/user");
    return res.json();
  },
});
```

7. **Better Performance by Default**

Redux → entire connected tree may re-render

Recoil → **fine-grained subscriptions**

Redux vs Recoil (Quick Comparison)

Feature	Redux	Recoil
Boilerplate	High	Very Low
Learning Curve	Steep	Easy
Store	Single global store	Multiple atoms
Async Handling	Middleware needed	Built-in
Derived State	Reselect	Selectors
React Integration	External	Native
Best for	Large enterprise apps	Medium apps & UI state

When to Use Recoil Instead of Redux?

Use **Recoil** when:

- You want **simple global state**
- UI-focused state
- Less boilerplate
- React-only apps

Avoid Recoil when:

- App needs strict architecture
- Heavy business logic
- Large team standardizing on Redux
