1. **useContext vs Redux**

Both useContext and Redux are widely used tools for managing global state in React applications, but they serve different purposes and are suited for different levels of complexity in an application's state management needs.

useContext

- useContext is a **built-in React hook** introduced with React 16.3, designed to avoid "prop drilling" — the process of passing props through many nested components when only a few components need access to the data.
- It allows developers to **share global state across the component tree** by creating a Context object using React.createContext() and wrapping parts of the app with a corresponding provider.
- **Example use cases** include sharing themes, authentication states, locale settings, or small user preferences — typically where the state is not too large or doesn't require advanced updates.
- You can create **multiple contexts** depending on your application's architecture. This provides flexibility and semantic clarity, though managing multiple contexts can become cumbersome as the app grows.
- However, useContext is **not optimized for complex state updates** or high-frequency state changes. Frequent re-renders can occur if not handled carefully, which may affect performance in larger applications.

Redux

- Redux is a **predictable state container** for JavaScript applications. Although it's often used with React, it's framework-agnostic and can be used with Vue, Angular, or even plain JavaScript.
- Redux enforces a **unidirectional data flow** and uses a centralized **store** that holds the entire application's state in a single immutable object.
- In Redux, state changes are triggered by **dispatching actions**, which are plain JavaScript objects describing an event. These actions are handled by **reducers**, which are pure functions that update the state based on the action type.
- Using tools like **Redux Toolkit**, we can organize the store into **slices**, where each slice manages a specific part of the state. This makes large-scale state logic modular and easier to manage.
- Redux is well-suited for applications with **complex state logic**, **cross-component interactions**, or where **debugging and time-travel features** are essential. It provides

advanced features like middleware support (e.g., for handling async logic using Redux Thunk or Redux Saga), logging, and development tools.

Feature	useContext	Redux
Type	React built-in hook	External library
Setup	Minimal	Requires installation and setup
Use Case	Simple to moderate state sharing	Complex state logic and large-scale apps
Data Flow	Context Provider → Consumer	Action → Reducer → Store
Performance	May cause re-renders on update	Optimized with selectors and middleware
Dev Tools	Limited	Extensive Redux DevTools
Async Support	Manual (via hooks or context)	Native (with middleware)
Learning Curve	Low	Medium to High

2. Advantages of Using Redux Toolkit (RTK) Over Traditional Redux

Redux Toolkit (RTK) is the officially recommended way to write Redux logic. It simplifies Redux usage by reducing boilerplate, improving developer experience (DX), and making complex patterns more approachable.

a. Simplified Setup

Traditional Redux requires manual setup of actions, reducers, and middleware. RTK's configureStore simplifies this by providing sensible defaults, built-in middleware like redux-thunk, and Redux DevTools support out of the box.

b. Less Boilerplate

RTK's createSlice combines action creators and reducers into a single function, eliminating the need for separate action types and reducing repetitive code.

c. Cleaner State Updates with Immer

Traditional Redux requires immutable updates using spread operators or utility functions. RTK uses **Immer** under the hood, allowing developers to write "mutating" code while preserving immutability automatically.

Example:

Traditional Redux:

return { ...state, value: state.value + 1 };

RTK:

state.value += 1;

d. Improved Developer Experience

RTK:

- Reduces the number of files and code required.
- Enforces best practices.
- Works seamlessly with TypeScript.
- Makes state logic easier to understand and maintain.

5. Built-in Support for Async Operations

RTK includes createAsyncThunk to handle asynchronous logic like API calls in a structured and efficient way—removing the need to manually handle loading, success, and error states.

Conclusion

Redux Toolkit makes working with Redux more efficient, less error-prone, and better suited for modern application development. It abstracts complex patterns while offering powerful tools to manage both synchronous and asynchronous state logic with ease.

3. Explain Dispatcher

In Redux, a **dispatcher** is the mechanism used to send **actions** to the Redux store. Whenever we want to update the state, we **dispatch** an action using the dispatch() function.

This dispatched action is then passed to the **reducer**, a pure function responsible for determining how the current state should change based on the action's type and payload.

The flow looks like this:

Component → dispatch(action) → Reducer → Store (updated state)

Key Points:

- dispatch() is the only way to trigger a state update in Redux.
- Actions are plain JavaScript objects describing *what happened*.

- The reducer listens for these actions and returns a new updated state based on the action type.

In Redux Toolkit, dispatching becomes even simpler, as action creators are auto-generated when using createSlice.

Example:

```
js

dispatch(increment()); // calls the reducer tied to the 'increment' action
```

In summary, the dispatcher acts as the bridge between your components and the Redux store, ensuring that state updates happen in a predictable and controlled way.

4. Explain Reducer

In Redux, a **reducer** is a pure function that determines how the application's state should change in response to an **action**.

It takes two arguments:

1. The **current state**.
2. The **action** dispatched.

In Redux Toolkit, reducers are defined inside createSlice, which simplifies syntax and uses Immer to handle immutability internally.

In short, reducers are the core of Redux's state update mechanism, deciding *how* state changes in response to actions.

5. **Explain Slice**

In Redux Toolkit (RTK), a **slice** represents a logical portion of the Redux store. It is a self-contained module that includes:

- A name (namespace for action types)
- An initial state
- One or more **reducer functions**
- Automatically generated **action creators**

Slices help in organizing the Redux store based on features or application domains (e.g., userSlice, cartSlice, authSlice), making state management modular and scalable.

Each slice is created using the createSlice() function, which returns a reducer and corresponding action creators. These reducers are then combined in the store configuration under the reducer field.

```js
const counterSlice = createSlice({
  name: 'counter',
  initialState: { value: 0 },
  reducers: {
    increment: (state) => { state.value += 1 },
    decrement: (state) => { state.value -= 1 }
  }
});

// Export actions and reducer
export const { increment, decrement } = counterSlice.actions;
export default counterSlice.reducer;
```

Usage in store

```js
import counterReducer from './features/counter/counterSlice';

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

Summary:

- Slices break down the global state into manageable pieces.
- They encapsulate logic and actions related to a specific part of the state.
- createSlice simplifies Redux by reducing boilerplate and improving maintainability.

Explain Selector

In Redux (especially with Redux Toolkit), a **selector** is a function used to **read and access data** from the Redux store.

Components **cannot directly access** the store; instead, they **subscribe** to specific parts of the store using the useSelector hook from the react-redux library.

In the example above:

```
import { useSelector } from 'react-redux';


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

- useSelector connects the component to the Redux store.
- The selector function (state) => state.counter.value retrieves the specific piece of state the component needs.