Problem:

React components often need to share data like user info, theme, or authentication status. Passing props down through many nested components becomes tedious and error-prone — this is called prop drilling.

useContext Hook in React

Definition:

useContext is a **function** that helps you **share data (like state or functions)** across multiple components **without passing props manually** at every level.
It’s especially useful for managing global data such as:

- 🌗 Themes (light/dark)
- 🔐 User authentication
- 🧠 Application-wide settings or preferences

How to Resolve:

We create a Context using React.createContext(), provide the data at a high level using a Provider, and then consume it using useContext in any child component.

Step-by-Step Process:

1. Create a Context

```
const AppContext = React.createContext(defaultValue);
```

- defaultValue: The initial value that will be used when no Provider wraps the component.

Analogy:

Creating an **empty box** that will eventually hold your shared data.

2. Provide the Context

Wrap your component tree with a Provider to make the context available to all its children.

```
<AppContext.Provider value={someValue}>
  <ChildComponent />
</AppContext.Provider>
```

- value: The data (could be state, object, or functions) you want to share globally.

Analogy:

This is like **filling the box with data** and giving access to all components inside the tree.

3. Consume the Context

Use useContext in any child component to access the value from the context.

```
const value = useContext(AppContext);
```

Analogy:

This is like **opening the box** in any component and using the data inside it.

Simple Example:

```
// 1. Create context
const AppContext = React.createContext();

// 2. Provide context
function App() {
  const user = { name: "Venky", isLoggedIn: true };
  return (
    <AppContext.Provider value={user}>
      <Profile />
    </AppContext.Provider>
  );
}
// 3. Consume context
function Profile() {
  const user = useContext(AppContext);
  return <h1>Welcome, {user.name}</h1>;
}
```

3. Theme & Auth Context

Problem:

We often need **theme switching** (light/dark) and **authentication status** (isUserLoggedIn) across the entire app.

Basic Definition:

Theme and Auth Contexts are specialized Context instances used to manage **global UI and auth state**.

How to Resolve:

1. Create separate contexts for Theme and Auth.
2. Provide respective values (e.g., dark/light, isAuthenticated).
3. Access them using custom hooks.

Syntax:

```
const AuthContext = createContext();
const ThemeContext = createContext();
```

Example:

AuthContext.jsx

```
import { createContext, useContext, useState } from 'react';

const AuthContext = createContext();

export function AuthProvider({ children }) {
  const [isLoggedIn, setLoggedIn] = useState(false);
  return (
    <AuthContext.Provider value={{ isLoggedIn, setLoggedIn }}>
      {children}
    </AuthContext.Provider>
  );
}

export function useAuth() {
  return useContext(AuthContext);
}
```

// App.jsx

```
import { AuthProvider, useAuth } from './AuthContext';

function LoginButton() {
  const { isLoggedIn, setLoggedIn } = useAuth();
  return (
    <button onClick={() => setLoggedIn(!isLoggedIn)}>
      {isLoggedIn ? "Logout" : "Login"}
    </button>
  );
}
```

```
function App() {
  return (
    <AuthProvider>
      <LoginButton />
    </AuthProvider>
  );
}
```

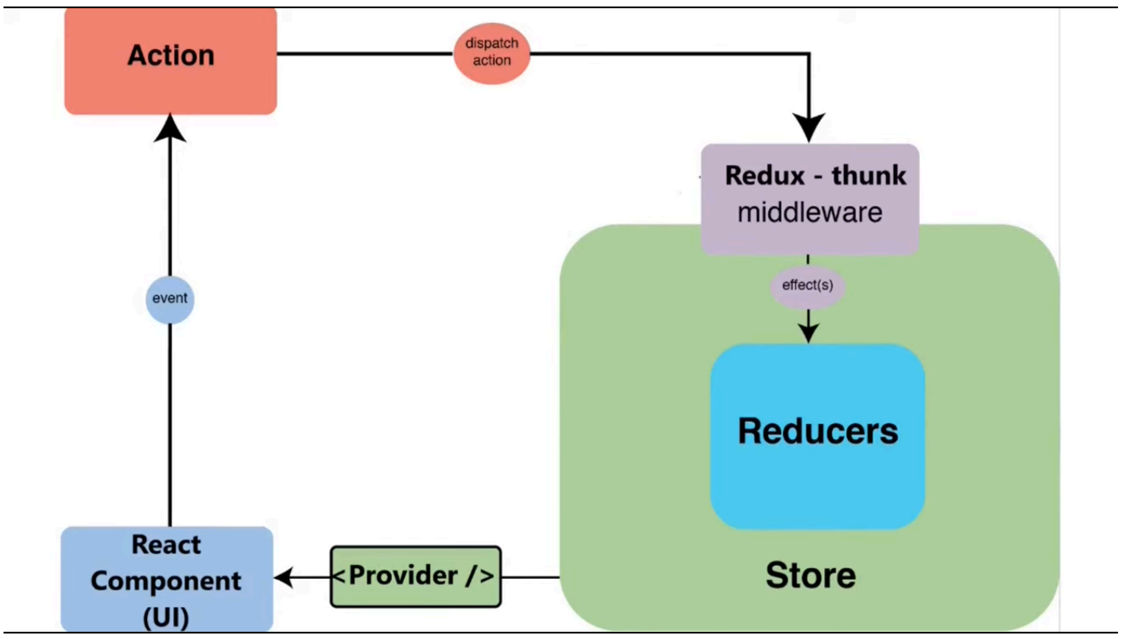
4. Intro to useReducer

Problem:

useState is not efficient when managing **complex state logic**, especially involving multiple state transitions or multiple pieces of related state.

Basic Definition:

useReducer is a React hook used for managing **complex state** using a reducer function, similar to Redux reducers.



How to Resolve:

Define a reducer function with state logic, pass it to useReducer, and dispatch actions to update the state.

Syntax:

```
const [state, dispatch] = useReducer(reducerFn, initialState);

function reducerFn(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    default:
      return state;
  }
}
```

Example:

```
import { useReducer } from 'react';

const initialState = { count: 0 };
function reducer(state, action) {
  switch (action) {
    case 'inc': return { count: state.count + 1 };
    case 'dec': return { count: state.count - 1 };
    default: return state;
  }
}
export default function CounterReducer() {
  const [state, dispatch] = useReducer(reducer, initialState);

  return (
    <>
      <p>Count: {state.count}</p>
      <button onClick={() => dispatch('inc' )}>+</button>
      <button onClick={() => dispatch('dec')}>-</button>
    </>
  );}
```

5. Custom Hooks

Problem:

When logic (like fetching data, toggling UI, managing timers) is repeated across components, it creates **duplicate code** and clutter.

Basic Definition:

A **custom hook** is a reusable function that uses built-in React hooks and can be shared across components.

How to Resolve:

Move the repeated logic into a function that starts with `use`, return the needed values, and use that hook in multiple places.

Syntax:

```
function useCustomHook() {
  // useState / useEffect logic
  return { /* data */ };
}
```

Example:

useToggle.jsx

```
import { useState } from 'react';

export function useToggle(initial = false) {
  const [value, setValue] = useState(initial);
  const toggle = () => setValue((v) => !v);
  return [value, toggle];
}
```

// App.jsx

```jsx
import { useToggle } from './useToggle';

export default function ToggleExample() {
  const [isVisible, toggleVisibility] = useToggle();

  return (
    <>
      <button onClick={toggleVisibility}>
        {isVisible ? "Hide" : "Show"} Text
      </button>
      {isVisible && <p>Hello! This is visible.</p>}
    </>
  );
}
```