{isOpen && <Component /> } **= Short Circuiting(&&)** {isOpen ? <Component /> : null} = **Ternary Operator (? :)**

**Components** are independent and reusable part of React. They serve the same purpose as JavaScript functions, but work in isolation and return HTML. Components come in two types, Class components and Function components.

**The Props** is a special keyword in React that stands for properties and is used for passing data **from** **parent** component **to** **child** components. Props are read-only (Only can change from parent component). And you can’t use props between sibling components. Only down the tree (parent → child).

**The State** is a component's memory in React that is used to contain data or information about the component. A component's state can change over time; whenever it changes, the component re-renders.

**Rendering** is React's process of describing a User Interface (UI) based on the component's current state and props. Rendering happens 2 times.

a). **Initial Render**: when first page upload

b). **Re-render:** when state updates

A **Key** is a special **unique** string attribute you need to include when creating multiple instance child components in React. There is no any rendering(changing) in child components if we did not include Key property.

**State Management**

**1. useState**

**2. useReducer**

**3. useContext (Context API)**

**4, Redux / Redux Toolkit** (Global State Management)

**5. React Query** (Remote State Management)

**The useState** is React Hook that lets you add state variable to your component. *const [state, setState] = useState(initialState)*

**The useEffect** hook allows you to perform **side effects** in your components. Some examples of side effects are: fetching data, directly updating the DOM, and timers. useEffect does not return anything and works **Asynchronously(executed after browser render).**

useEffect accepts two arguments. useEffect(function(){}, []) Function, Depedency Array

**Lifting Up** to place some state in a component that is parent of both components.

**The Prop Drilling** is the act of passing data(props), through several nested layers of components.

**Custom Hooks** just like normal JavaScript Function but it must use at least one or more React Hooks. Try to **export** it not export default and in order to Recognize this function as Custom Hooks we always start names with **useFunctionName**. At the end it is mandatory **to return states**, otherwise it is gonna be a regular function in the eyes of React. In other words, Custom Hooks are a powerful feature of React that allows you to encapsulate common functionality into reusable functions. They are a great way to make your code more organized, reusable, and maintainable. Context API-de biz ona gore custom hooks yaradiriqki (onun icine useContext daxil edirik stateleri oxumaq ucun) diger componentlerden el catan olmaqi ucun

**The useRef Hook** allows you to persist values between renders. It can be used to store a mutable value that **does not cause a re-render when updated.** We can only update useRef in useEffect (Updating input element).

It can be used to access a DOM element directly, Local/Sessions storage, APIs and Timer. const ref = useRef(initialValue)

**The useReducer** is very similar to useState , but it lets you move the state update logic from event handlers into a single function outside of your component. All **states centralized** in Central Place (Reducer Function). const [state, dispatch] = useReducer(reducer, initialState).

**Dispatch:** Function to trigger **state updates**, by sending **action** from event handler to the **reducer** Function.

***HINT!*** *useReducer Function* ***does note mutate*** *current state, instead COPY it and create new one*

**Link** is an element that lets the user navigate to another page by clicking or tapping on it.

**NavLink** is a special kind of Link that knows whether or not it is "active", "pending", or "transitioning".

www.example.com/path/params?query=abc&string=123

**Index** determines if the route is an index route. By default open component which has index route.

**Index Route** aid olduqu scopede(<Route></Route>) eger nese tapmasa burani default olaraq gotursun.

**The <Outlet/>** component is used within <Root/> to specify where in the view the child element. When Route has child route (Nested Route) like children props. Outlet ancaq lazim olan URL yazilanda hemen pageni gosterir.

**<Navigate replace to="location"/>** element changes the current location when it is rendered. Replace geriye qayitmaq ucun istifade olunur.

**The useNavigate** navigates to the other routers using the push or replace methods navigate(‘location’)

**The useParams** hook is a React router hook that is used to create and access multiple parameters of a website and main Routedeki path="location/:id" gelen linkin {id: '73930385'} hissesidir

**The useSearchParams** provides a simple API to work with URL search parameters in React.

**Context API** basically allows components everywhere in the tree to read state that a context shares.

**1. Create new content** const Context = createContext()

**2. Provider** gives all child components access to value <Context.Provider value={values}></Context.Provider>

**3. Value** data that we want to make available (usually state and functions) values = { name, fullName, age: birthYear}

**4. Consumers** all components that read the provided context value const states = useContext(Context)

**The useSelector** for **read** **data** from Redux Store const customer = useSelector((store)=>store.customer)

**The useDispatch** for **call** **dispatch method** from Redux Store const dispatch = useDispatch() => dispatch(function())

***HINT!*** *Context API Function* ***does note mutate*** *current state, instead COPY it and create new one*

**The createAsyncThunk** we can’t call **async** functions directly inside Redux Reducer (because Redux is by nature completely **synchronous**). Thunk is **Middleware** that sits between the dispatching and reducer. In Redux Toolkit we use createAsyncThunk. And receives 2 parametr (actionName, async function) and dont use **get** keyword in the name calling. (it is for only useSelector). createAsyncThunk produce 3 action type: Depending promise state, fulfilled, rejected. We handle(connect) them into our reducer with **extraReducers**.

**The useFetcher** fetchs some data from another route (data is not associated with current page) without causing a navigation. (Yeni basqa sehifeye kecmeden hemin pagedeki datani alir)

**Performance Optimizations (Memoization)** optimization technique that executes a pure function once, and saves the result in memory. If we try to execute the function again with the same arguments as before, the previously saved result will be returned, without executing the function again.

**The memo Function** used to create a component that **will not re-render when its parent re-renders, as long as the props stay the same** between renders (**if prop change it will re-render**). Only affects props! A memoized component **will still re-render when its own state changes** or when a context that it's subscribed to changes. Only makes sense when the component is heavy (slow rendering), re-renders often, and does so with the same props. If **objects** or **functions** are passed as props, the child component will always see them **as new props on each re-render**, as result perfomance go slow. For preventing it **useMemo/Callback** come into play. **useMemo** and **useCallback** have a **dependency array** (like useEffect): whenever one dependency changes, the value will be re-created.

***HINT!******useMemo*** *accept function while* ***useCallback*** *don’t.*

**memo** memoize Components const Name = memo(function Name() {})

**useMemo** memoize Objects const Name = useMemo(()=> {return {Object}}, [])

**useCallBack** memoize Functions const Name = useCallback(function Name() {}, [])