Week 7: React Hooks and Functional State Management

Day 1: The Transition to Hooks and Rules of Engagement

Functional Component Shortcomings

Before the introduction of Hooks in React 16.8, functional components were strictly "stateless." They could not hold their own data or tap into the component lifecycle. To manage state, developers were forced to write complex class components. Hooks were introduced to allow functional components to use state and other React features without writing a class.

The Fundamental Rules of Hooks

To ensure React correctly associates state with the right component, developers must follow two strict rules:

1. **Only Call Hooks at the Top Level**: Do not call Hooks inside loops, conditions, or nested functions. This ensures that Hooks are called in the same order each time a component renders.
2. **Only Call Hooks from React Functions**: Hooks must be called from React functional components or custom Hooks—never from regular JavaScript functions.

Day 2: The useState Hook

Managing Local State

The useState hook is the primary tool for adding local state to functional components. When you call useState, it returns a pair: the current state value and a function that lets you update it.

- **Initialization**: State can be initialized with strings, numbers, booleans, or objects.

- **Multiple Variables**: A single component can use multiple useState hooks to track different pieces of data independently.
- **Asynchronous Updates**: State updates are not immediate; React schedules the update and re-renders the component to reflect the new value.

Day 3: The useEffect Hook and Lifecycle Management

Handling Side Effects

The useEffect hook allows you to perform side effects in functional components, such as data fetching, subscriptions, or manually changing the DOM. It serves the same purpose as componentDidMount, componentDidUpdate, and componentWillUnmount in class components.

- **Data Loading**: It is frequently used to load data from an external API (like your **Gadget API**) when a component first loads.
- **Dependency Array**: By providing an array of dependencies, you can restrict when the effect is called, ensuring it only runs when specific values change.
- **Cleanup Logic**: If an effect returns a function, React will execute it when the component unmounts, preventing memory leaks.

Day 4: Performance Optimization: useMemo and useCallback

The useMemo Hook

useMemo is used to memoize expensive calculations. It ensures that a computation is only re-run when one of its dependencies changes, rather than on every render.

The useCallback Hook

useCallback returns a memoized version of a callback function that only changes if one of the dependencies has changed. This is useful when passing callbacks to optimized child components that rely on reference equality to prevent unnecessary renders.

Day 5: Advanced Hooks and State Hierarchy

The useReducer Hook

For complex state logic that involves multiple sub-values or when the next state depends on the previous one, useReducer is preferred over useState. It uses a "reducer" function and "actions" to transition the state, similar to how Redux operates.

Lifting State Up and Component Communication

When multiple components need to share the same data, React encourages "Lifting State Up".

- **State Hierarchy**: State is moved to the closest common ancestor of the components that need it.
- **Passing down Functions**: The parent component passes a function down to the child as a prop, allowing the child to trigger a state update in the parent.
- **Props vs. State**: Props are read-only data passed from a parent, while state is local, mutable data managed within the component itself.

The useRef Hook

The useRef hook returns a mutable ref object whose .current property is initialized to the passed argument. It is used to persist values between renders without triggering a re-render or to access a DOM element directly.