React Hooks

Why React Hooks?

Reasons to choose react hooks over react classes:

- **No need for conversion from functional to component class:** In React app there is a need to change functional class with props to component class with the state as the application grows. This process becomes sometimes difficult due to large complex classes. React hooks save us from this scenario as it has the capability to pass state in the functional component.

- **The significance of “this” becomes less:** “this” keyword always confuses developers with its functionality. We can avoid the use of “this” keyword with React hooks which is beneficial for new developers.

- **No need for binding methods:** In React applications we have to bind functions together to make them functional and that makes things confusing. With React hooks, this thing becomes easier as it has inbuilt functions for this.

- **Logic And UI are decoupled:** Using hooks, logic and UI is easy to distinguish. No need for HOC or render props. Hooks do it beautifully with a small boilerplate and a precise design of the UI and logic. This makes code more reusable.

- **Related Logic at Same Place:** React Hooks keeps all related logic at one place unlike life cycle methods in classes. We have to differently define things in componentDidMount or componentDidUpdate which makes things confusing when they are filled with many logics.

- **Helps to share stateful logic between components:** Unlike classes React hooks helps to share the same logic between different components which is not possible with classes as there each component has its own state and sources.

What are hooks?

Hooks provide us with a feature to use state and other React features without using a class component. Hooks are the function which **"hooks into"** state and lifecycle features with the functional component. It does not work with the class component.

useEffect Hook

useEffect is a substitution to class component lifecycle methods. We can use it to copy the functionality of componentDidMount, componentDidUpdate etc. It helps to tell the component to execute some logic after rendering the component.

Why it is used inside the component?
"useEffect" function is defined in the component so that the variables and functions defined within the components are directly accessible. It uses the concept of closure to provide access to local functions and the variables defined within the function.

Does it run after every render?
Yes, it runs before the first render and after every update. Though we can customise this using by passing some additional arrays. useEffect runs after every update because every action requires newly updated data.

Note: Unlike componentDidMount or componentDidUpdate, the results set for useEffect do not prevent the browser from updating the screen. This makes your app feel more responsive. Most effects do not happen simultaneously.

Rules for Using Hooks

Always use hooks at top-level:
Do not call Hooks within loops, conditions, or integrated functions. Use Hooks at the top level of your React function. By following this rule, you ensure that Hooks are called in sequence each time a component renders. This is what allows React to properly maintain the state of Hooks between multiple useState and useEffect calls.

Call hooks from React Components:
Always call hooks from React components, not from a plain javascript function.

Call hooks from custom hooks:
Hooks can be called from custom hooks whose names start with “use” and they have useState and useEffect hooks used inside them.

Miscellaneous Hooks

- **useContext:** useContext makes it easy to pass data down the tree without using props. We can directly pass data from the Provider to the component. It is useful only when the app is small and the data is simple. It replaces the consumer component and helps to avoid nesting.

- **useRef:** UseRef Hook is a function that returns a fixed object with its .current property initialised with the argument (initialValue). The returned object will remain for the full lifetime of the component.

- **useReducer:** It is an alternative to state. Accept reducer of type (status, action) => newState, and restore the current state paired with the dispatch method.
  useReducer is usually preferred to useState where you have a complex idea that involves a lot of sub-values or where the next state relies on the previous one. useReducer also allows you to maximize the functionality of the elements that cause deep updates because you can dispatch down rather than passing callbacks.

Summarising It

Let's summarise what we have learnt in this module:

- Learned about why hooks are used instead of classes.
- Learned about hooks in detail and rules to use hooks.
- Learned about useEffect and other miscellaneous hooks.

Some References:

● useState
https://reactjs.org/docs/hooks-state.html
● Hooks in detail
https://reactjs.org/docs/hooks-reference.html