Hooks

Problems

- It’s hard to reuse stateful logic between components
- Complex components become hard to understand
- Classes confuse both people and machines

Hooks to the rescue!

- you can extract stateful logic from a component so it can be tested independently and reused
- allow you to reuse stateful logic without changing your component hierarchy
- let you split one component into smaller functions based on what pieces are related, rather than forcing a split based on lifecycle methods
- Hooks let you use more of React’s features without classes
- Completely opt-in and 100% backwards-compatible

Rules of Hooks

- Only call Hooks at the top level. Don’t call Hooks inside loops, conditions, or nested functions
- Only call Hooks from React function components. Don’t call Hooks from regular JavaScript functions
- Call hooks on your custom hooks

Let’s play with it!

React Hooks

useState(initialState)

- Returns array [value, setValue]
- Same value will not trigger re-render
- Can accept function to Lazy Initial State
- Multiple updates will be merge in 1
- Can hold object/arrays if you need a lot of data in 1 useState
- No callback function
- setState identity is stable

useEffect(effect, deps)

- 1st parameter is a function that will be executed after React paints the UI in the DOM
- 1st parameter can return a function for cleanup (like componentWillUnmount)
- 2nd parameter are dependencies to check whether to execute the effect or not

useLayoutEffect(effect, deps)

- useLayoutEffect runs in sync before browser has painted

useCallback + useMemo

- Both use for memoization
- 1st parameter is the function to be memoized
- 2nd parameter are dependencies
- useCallback for callback/function
- useMemo to memoize computation of value but can be used for function as well

useRef

- Reference elements/components
- Use only in Effects or events otherwise it will lead to suprising behaviours
- Can still used callbackRef if you need to listen when ref was attached to a different node
- Can be used as instance fields so that if component rerenders, value will not changed

useImperativeHandle

- Override ref Implementation
- Should be used with forwardRef
- Deps are the same with useEffect
- imperative code using refs should be avoided in most cases

Custom Hooks

- Create your own hooks
- Function that contains 1 or more hooks to create your own hook and logic
- Name should start with “use” as a convention

useContext(context)

- read the context and subscribe to its changes

useReducer(reducer, initialValue, init)

- Reducer parameter is same with Redux
- 2nd parameter is the initialValue
- Use 3rd parameter for Lazy initialization

- Like in useState, if value didn’t change it will not rerender
- Dispatch identity is stable

- usually preferable to useState when you have complex state logic that involves multiple sub-values or when the next state depends on the previous one

Hooks in React Redux

- **useSelector(selector: Function, equalityFn?: Function)** – select data from store
- **useDispatch –** get dispatch function
- **useStore –** get store object

useDebugValue(value, format?)

- can be used to display a label for custom hooks in React DevTools.
- 2nd parameter is a function to format value, only called when you inspect the hook
- **We don’t recommend adding debug values to every custom Hook. It’s most valuable for custom Hooks that are part of shared libraries.**

Additional Information

- no Hook equivalents to the uncommon **getSnapshotBeforeUpdate**, **getDerivedStateFromError** and **componentDidCatch** lifecycles yet

- **shouldComponentUpdate** – use React.Memo for props and useMemo to render components

- raw performance of closures compared to classes doesn’t differ significantly except in extreme scenarios.

BECOME A HOOKS MASTER