React Hooks

State, effects, context and refs in React 19.

01 useState

```
const [count, setCount] = useState(0)
setCount(count + 1)         // set a value
setCount(prev => prev + 1)  // from previous
setUser({ ...user, name: 'Jane' })  // copy
```

Updates are batched. The variable keeps its old value until the next render, so use the function form to build on it.

02 useReducer

```
function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 }
    default:
      throw new Error('Unknown action')
  }
}
const [state, dispatch] =
  useReducer(reducer, { count: 0 })
dispatch({ type: 'increment' })
```

For complex logic: every change is a named action.

03 useEffect

```
useEffect(() => {
  fetchData()
}, [])          // once, after mount
useEffect(() => {
  document.title = `${user.name} - Profile`
}, [user])      // when user changes
useEffect(fn)   // after every render
```

Runs after render. List every value it reads; the ESLint plugin warns about missing ones.

04 Cleanup & fetching

```
useEffect(() => {
  let cancelled = false
  fetch(`/api/users/${userId}`)
    .then(res => res.json())
    .then(data => {
      if (!cancelled) setUser(data)
    })
  return () => { cancelled = true }
}, [userId])
```

Cleanup runs before the effect re-runs and on unmount. Clear timers and listeners there, and ignore stale responses.

05 useLayoutEffect

```
useLayoutEffect(() => {
  const height = ref.current.scrollHeight
  ref.current.style.height = `${height}px`
}, [])
```

Like useEffect, but runs synchronously before the browser paints, so you can measure and adjust the DOM. It blocks painting; use it sparingly.

06 useContext

```
const ThemeContext = createContext(null)
const ThemeProvider = ({ children }) => (
  <ThemeContext value={{ dark: true }}>
    {children}
  </ThemeContext>
)
const theme = useContext(ThemeContext)
```

In React 19 the context is its own provider; .Provider is no longer needed. Readers re-render when the value changes.

07 useRef

```
function Search() {
  const inputRef = useRef(null)
  useEffect(() => {
    inputRef.current.focus()
  }, [])
  return <input ref={inputRef} />
}
```

Changing .current does not re-render, so refs also hold timers, subscriptions and other values that survive renders.

08 useImperativeHandle

```
function Child({ ref }) {
  const inputRef = useRef(null)
  useImperativeHandle(ref, () => ({
    focus: () => inputRef.current.focus(),
  }))
  return <input ref={inputRef} />
}
```

React 19: ref is a normal prop, no forwardRef.

09 useId

```
function EmailField() {
  const id = useId()
  return (
    <>
      <label htmlFor={id}>Email</label>
      <input id={id} type="email" />
    </>
  )
}
```

SSR-safe unique IDs for labels and ARIA attributes.

DevSheets

React Hooks

Performance, Actions and custom hooks.

10 useMemo

```
const total = useMemo(() => {
  return items.reduce(
    (sum, item) => sum + item.value, 0)
}, [items])   // re-runs when items changes
const options = useMemo(
  () => ({ color: 'blue' }), [])  // stable
```

Only for expensive work like sorting, filtering and big calculations, or to keep an object stable. With React Compiler enabled, most manual memoization is unnecessary.

11 useCallback

```
const handleClick = useCallback(() => {
  doSomething(id)
}, [id])        // changes with id
const addTodo = useCallback((text) => {
  setTodos(prev => [...prev, { text }])
}, [])          // never changes
```

Keeps a function the same between renders so a memo() child can skip re-rendering. Worth it for callbacks passed to memoized components, not for every function.

12 useTransition & useDeferredValue

```
const [isPending, startTransition] =
  useTransition()
startTransition(() => {
  setSearchQuery(input)  // not urgent
})
const deferred = useDeferredValue(query)
```

Keeps the UI responsive during heavy updates; React 19 also accepts async functions (Actions). useDeferredValue lets a slow child render with a lagging value.

13 use()

```
function UserName({ userPromise }) {
  const user = use(userPromise) // suspends
  return <p>Hello, {user.name}</p>
}
<Suspense fallback={<p>Loading...</p>}>
  <UserName userPromise={userPromise} />
</Suspense>
```

Reads a promise or a context, even inside if blocks and loops. Pass a cached promise, such as one from a Server Component.

14 useActionState

```
async function save(prevState, formData) {
  const name = formData.get('name')
  if (!name) return { error: 'Required' }
  await saveUser(name)
  return { error: null }
}
const [state, formAction, isPending] =
  useActionState(save, { error: null })
```

Its return value becomes state. Pass formAction to the form action prop. Replaces useFormState.

15 useFormStatus

```
import { useFormStatus } from 'react-dom'
function SubmitButton() {
  const { pending } = useFormStatus()
  return (
    <button disabled={pending}>Save</button>
  )
}
```

Reads the parent <form> status, so render it inside that form.

16 useOptimistic

```
const [optimistic, addOptimistic] =
  useOptimistic(messages, (state, text) =>
    [...state, { text, sending: true }])
async function formAction(formData) {
  addOptimistic(formData.get('text'))
  await sendMessage(formData)
}
```

Shows the change at once until the action ends. Call it in an action or startTransition.

17 Custom hooks

```
function useToggle(initial = false) {
  const [on, setOn] = useState(initial)
  useDebugValue(on ? 'On' : 'Off')
  const toggle = () => setOn(prev => !prev)
  return [on, toggle]
}
const [isOpen, toggleOpen] = useToggle()
```

A use-prefixed function that calls hooks. It shares logic, not state. useDebugValue labels it in DevTools.

18 Rules & gotchas

Top level only	Never inside if blocks or loops
React functions	Components and custom hooks
use() exception	Allowed in if blocks and loops
Effect deps	List every value it reads
After setX()	Old value until next render
ref.current	Changes do not re-render

The React Hooks ESLint plugin enforces these rules.