React Hooks cheat sheet

useEffect.dev

Local state: `useState`

- Declare the state

```
const [name, setName] = useState('initial value')
```

- Update the state

```
setName('new value')
// or
setName((value) => 'new ' + value)
```

- Lazy-initialize the state

```
const [value, setValue] = useState(
  // Evaluated only at first rendering
  () => computeValue()
)
```

Side effects: `useEffect`

- Trigger side effects when the component is mounted

```
useEffect(() => {
  // HTTP request, setTimeout, etc.
  doSomething()
}, [])
```

- Trigger side effects each time a prop or state is updated

```
useEffect(() => {
  doSomethingWith(value)
}, [value])
```

- Clean something when the component is unmounted:

```
useEffect(() => {
  let timeout = setTimeout(doSomething, 5000)
  return () => clearTimeout(timeout)
}, [])
```

Rules when using hooks

- Must only be used in function components
- Only at component top-level (not in `if`)
- No `return` before any hook

Custom hooks

- Must start with `use`
- Used to extract common behavior of components, like async requests:

```
const useApiResult = (param) => {
  const [result, setResult] = useState(null)

  useEffect(() => {
    fetch('http://your.api?param=' + param)
      .then((res) => res.json())
      .then((result) => setResult(result))
  }, [])

  return { result }
}
// To use it in a component:
const { result } = useApiResult('some-param')
```

Getting the current state in async code

- Problem: in a `useEffect`, you want to get the value of the current state to update it (e.g. increment a number)

```
const [val, setVal] = useState(0)
useEffect(() => {
  setInterval(() => {
    setVal(val + 1)
  }, 1000)
}, [])
console.log(val) // always logs 0 :(
```

- Solution 1: use the other syntax of `setValue`:

```
const [val, setVal] = useState(0)
useEffect(() => {
  setInterval(() => {
    // val always contain the current value
    setVal((val) => val + 1)
  }, 1000)
}, [])
console.log(val) // logs 0, 1, 2, 3... :)
```

- Solution 2: use a ref containing the current value:

```
const [val, setVal] = useState(0)

const valRef = useRef(val)
useEffect(() => (valRef.current = val), [val])

useEffect(() => {
  setInterval(() => {
    // valRef.current contains the current value
    setVal(valRef.current + 1)
  }, 1000)
}, [])
console.log(val) // logs 0, 1, 2, 3... :)
```

Solving infinite loops with `useEffect`

- Cause 1: no dependencies array:

```
useEffect(() => {
  asyncFunction().then((res) => setValue(res))
})
```

- Solution: always pass an array as second parameter to `useEffect`:

```
useEffect(() => {
  // ...
}, [])
```

- Cause 2: states updating each other

```
useEffect(() => {
  setSecond(first * 3)
}, [first])

useEffect(() => {
  setFirst(second / 2)
}, [second])

// Triggers an infinite update loop :(
<button onClick={() => setFirst(5)}/>
```

- Solution 2: store in a state which value was updated by the user:

```
const [updating, setUpdating] = useState(null)

useEffect(() => {
  if (updating === 'first') setSecond(first * 3)
}, [first, updating])

useEffect(() => {
  if (updating === 'second') setFirst(second / 2)
}, [second, updating])

// No infinite update loop :)
<button onClick={() => {
  setUpdating('first')
  setFirst(5)
} />
```

- Cause 3: using an object state

```
const [object, setObject] = useState({ value: 'aaa', changes: 0 })

useEffect(() => {
  setObject({ ...object, changes: object.changes + 1 })
}, [object])

<button onClick={() => {
  // Will trigger an infinit loop :(
  setObject({ ...object, value: 'bbb' })
}} />
```

- Solution 3: watch only some of the object's attributes

```
const [object, setObject] = useState({ value: 'aaa', changes: 0 })

useEffect(() => {
  setObject({ ...object, changes: object.changes + 1 })
}, [object.value]) // watch only the `value` attribute

<button onClick={() => {
  // No infinit loop :)
  setObject({ ...object, value: 'bbb' })
}} />
```

Memoize a value with `useMemo`

```
const value = useMemo(() => {
  // Will be evalutated only when param1 or param2 change
  return expensiveOperation(param1, param2)
}, [param1, param2])
```

Memoize a callback with `useCallback`

```
// Will return a new function only when param1 or param2 change
const handleClick = useCallback(() => {
  doSomethingWith(param1, param2)
}, [param1, param2])
```

- Memoize callback for a dynamic list of elemments:

```
// The same function for all the buttons created dynamically
const handleClick = useCallback((event) => {
  const button = event.target
  const value = button.getAttribute('data-value')
  doSomethingWith(value)
}, [])

<ul>
  {objects.map((obj) => (
    <li key={obj.id}>
      <button data-value={obj.value} onClick={handleClick}>
        {obj.value}
      </button>
    </li>
  ))}
</ul>
```

Contexts & provider/consumer with `useContext`

- Create the context:

```
const themeContext = createContext()
```

- Create a specific provider for the context:

```
const ThemeProvider = ({ children, initialTheme = 'light' }) => {
  const [theme, setTheme] = useState(initialTheme)
  return (
    <themeContext.Provider value={[theme, setTheme]}>
      {children}
    </themeContext.Provider>
  )
}

// Usage
;<ThemeProvider initialTheme="dark">
  <Label>Hello</Label>
</ThemeProvider>
```

- Create a custom hook to consume the context:

```
const useTheme = () => {
  const [theme, setTheme] = useContext(themeContext)
  // Add here additional logic if necessary...
  return [theme, setTheme]
}

// Usage
const [theme, setTheme] = useTheme()
```

Reducers to manage state with `useReducer`

- Initialize a local state:

```
const initialState = {
  value: 0,
}
```

- Create the reducer:

```
const reducer = (state, action) => {
  switch (action.type) {
    case 'increment':
      // Must create a new state, not modify the current one!
      return { ...state, value: state.value + 1 }
    case 'set_to':
      return { ...state, value: action.value }
    default:
      throw new Error('Unhandled action')
  }
}
```

- Create a local state using `useReducer`:

```
const [state, dispatch] = useReducer(reducer, initialState)

<span>{state.value}</span>
```

- Dispatch actions to update the state:

```
<button onClick={() => {
  dispatch({ type: 'increment' })
}} />
<button onClick={() => {
  dispatch({ type: 'set_to', value: 42 })
}} />
```

Examples of hooks to access the browser API

- Persist a state in the local storage

```
const usePersistedState = (key, initialValue) => {
  const [value, setValue] = useState(initialValue)

  useEffect(() => {
    const existingValue = localStorage.getItem(key)
    if (existingValue !== null) {
      setValue(existingValue)
    }
  }, [key])

  const setAndPersistValue = (newValue) => {
    setValue(newValue)
    localStorage.setItem(key, newValue)
  }

  return [value, setAndPersistValue]
}

// Usage
const [name, setName] = usePersistedState('name', 'John Doe')
```

- Get an element's size

```
const useElementSize = (elementRef) => {
  const [width, setWidth] = useState(undefined)
  const [height, setHeight] = useState(undefined)

  useEffect(() => {
    const resizeObserver = new ResizeObserver((entries) => {
      for (let entry of entries) {
        if (entry.contentRect) {
          setWidth(entry.contentRect.width)
          setHeight(entry.contentRect.height)
        }
      }
    })
    resizeObserver.observe(elementRef.current)

    return () => {
      resizeObserver.disconnect()
    }
  }, [elementRef])

  return [width, height]
}

// Usage
const div = useRef()
const [width, height] = useElementSize(div)
<div style={{ resize: 'both' }} ref={div} />
```

- Get the user's geolocation

```
const useGeolocation = () => {
  const [status, setStatus] = useState('pending')
  const [latitude, setLatitude] = useState(undefined)
  const [longitude, setLongitude] = useState(undefined)

  useEffect(() => {
    navigator.geolocation.getCurrentPosition(
      (res) => {
        setStatus('success')
        setLatitude(res.coords.latitude)
        setLongitude(res.coords.longitude)
      },
      (err) => {
        console.log(err)
        setStatus('error')
      }
    )
  }, [])

  return { status, latitude, longitude }
}

// Usage
const { status, latitude, longitude } = useGeolocation()
```