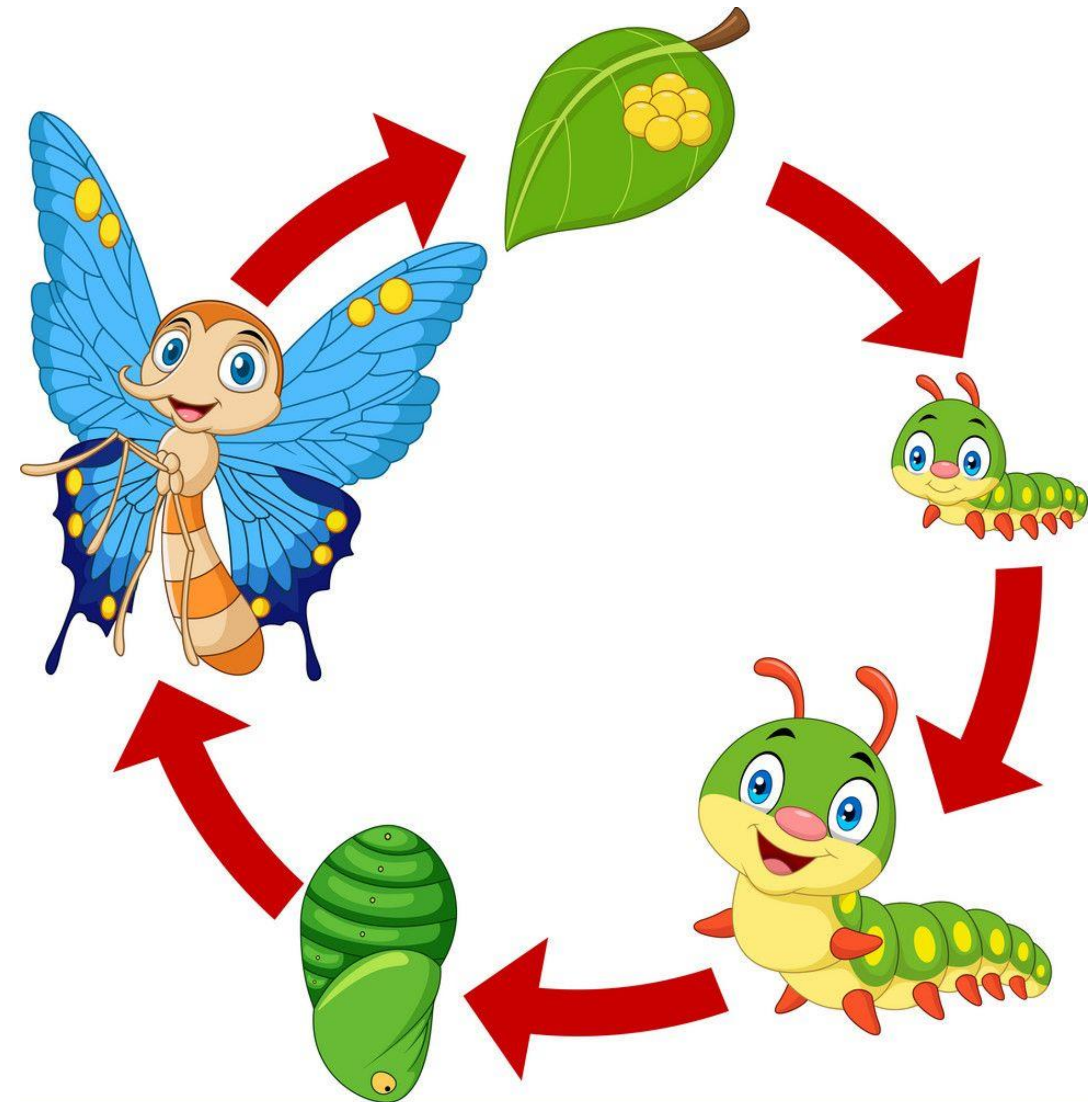
<WA1/>
<AW1/>
2021

React Life Cycle

Making React Components Alive

Fulvio Corno
Luigi De Russis
Enrico Masala



https://reactjs.org/docs/state-and-lifecycle.html

https://reactjs.org/docs/react-component.html

https://github.com/Wavez/react-hooks-lifecycle

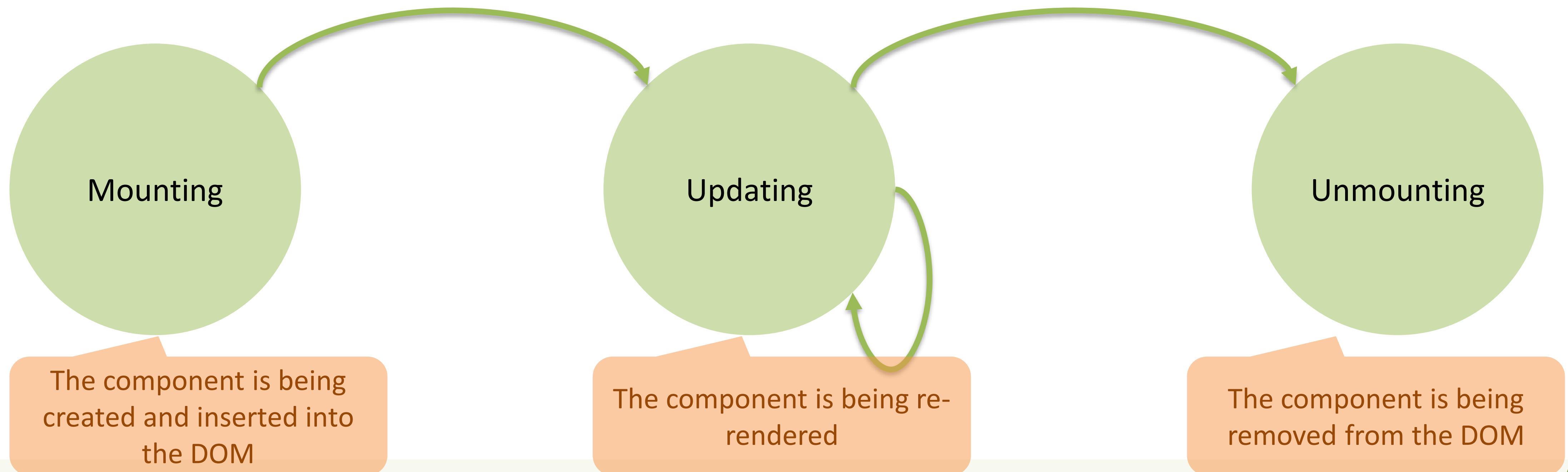
Full Stack React, Chapter “Advanced Component Configuration with props, state, and children”

There’s life before and after `return<JSX>`

COMPONENTS’ LIFECYCLE

Lifecycle Events

- The render action is the most important one for a component
- However, it is also useful to customize what happens at different moments in the evolution of the component



React Hooks Lifecycle

Mounting

Updating

Unmounting

"Render phase"

Pure and has no side effects. May be paused, aborted or restarted by React

`function () {}`

`useMemo()`

`return ()`

`useState()` `useReducer()` `useContext()`

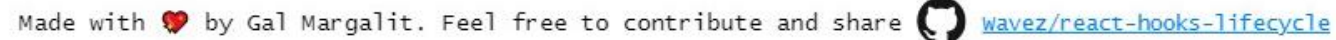
`useCallback()`

"Commit phase"

Can work with DOM, run side effects, schedule updates.

React updates DOM and refs

`useEffect()` `useLayoutEffect()`



Side Effects in Function Components

- A functional React component uses props and state to calculate its output
- **Side effect**: any calculation that do not target the output values, anything that affects something *outside the scope of the function component* being executed
- Examples of side effects:
  - **Data fetching**
  - Log recording
  - Setting up a subscriptions (handlers, etc.), or removing them
  - Scheduling additional actions when some state values change
  - Manually changing the DOM in React components
  - Managing timeouts and interval timers
  - …

Side Effects in Function Components

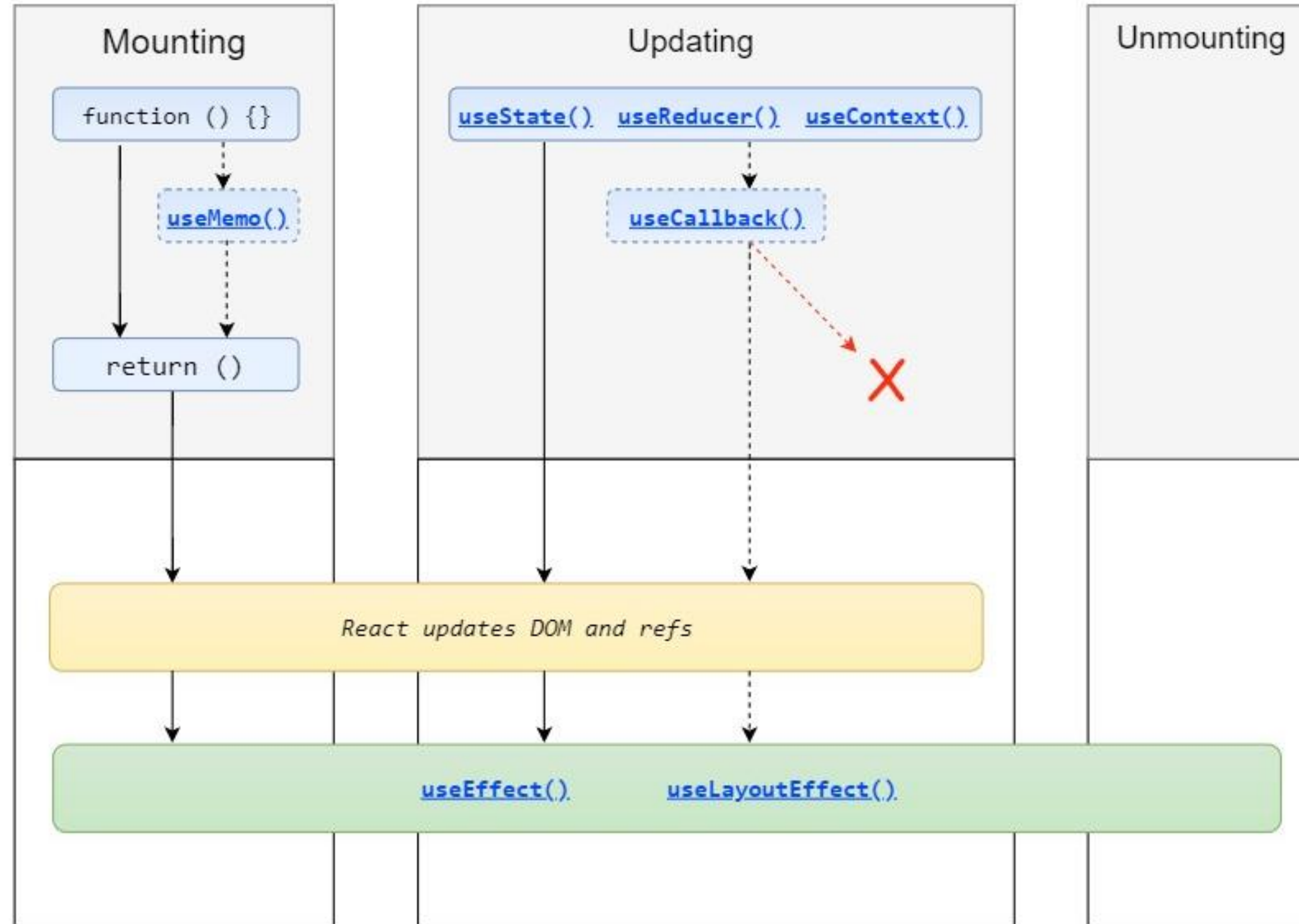
- A functional React component uses props and state to calculate its output
- **Side effect**: any calculation that do not target the output values, anything that affects something *outside the scope of the function component* being executed
- Examples of side effects:
  - **Data fetching**
  - Log recording
  - Setting up a subscriptions (handlers, etc.), or r
  - Scheduling additional actions when some stat
  - Manually changing the DOM in React compon
  - Managing timeouts and interval timers
  - …

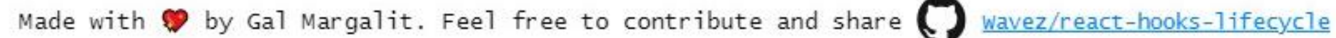
The component rendering and side-effect logic are *independent*.

It would be a mistake to perform side-effects directly in the body of the component.

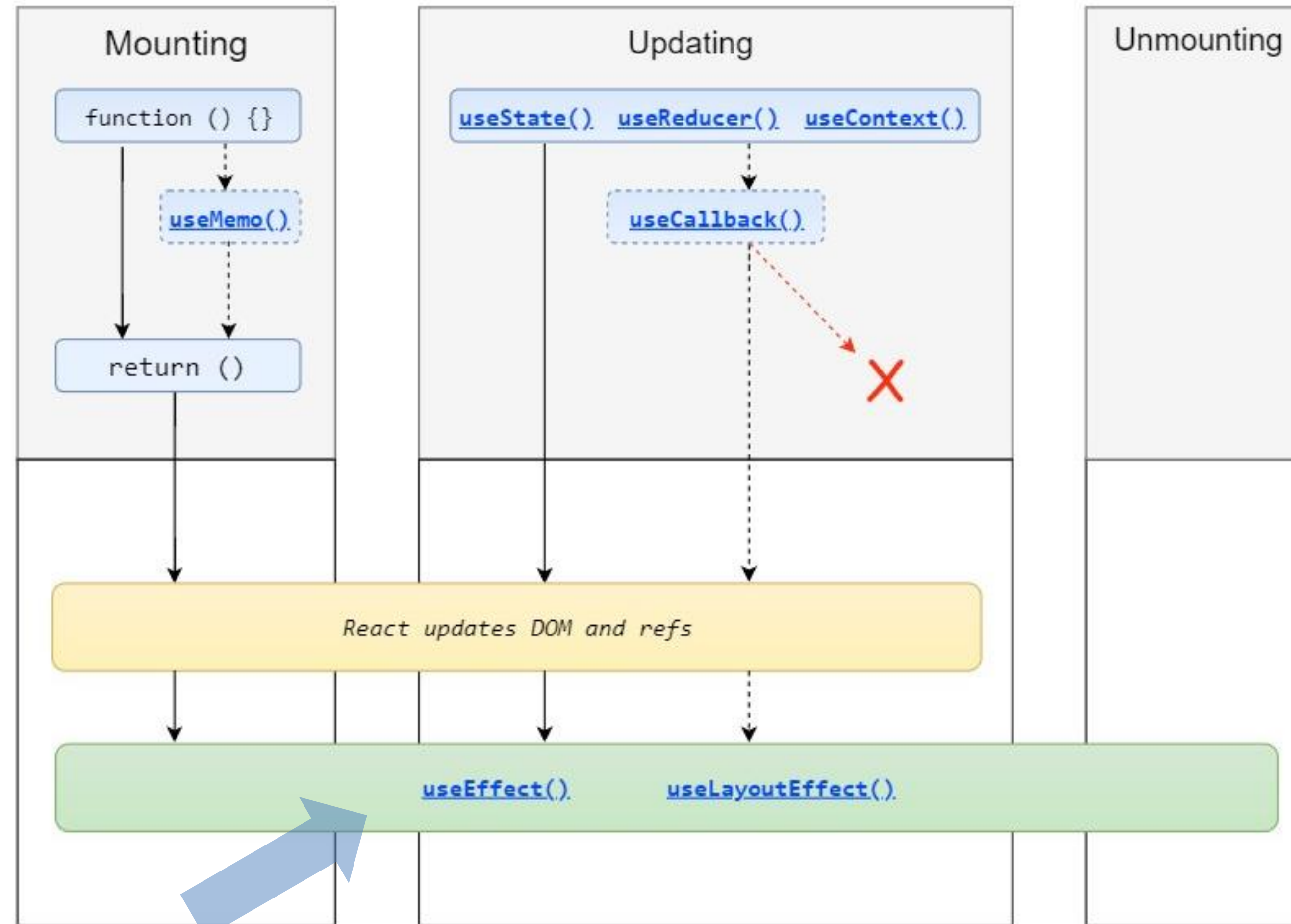
React Hooks Lifecycle

No side effects in the render phase





React Hooks Lifecycle



No side effects in the render phase

Side effects run after rendering and DOM updating

Full Stack React, “Appendix C: React Hooks”

React Handbook, chapter “Hooks”

https://reactjs.org/docs/hooks-effect.html

https://dmitripavlutin.com/react-useeffect-explanation/ (source for many examples)

Side-effects and Life Cycle in Functional Components

USEEFFECT HOOK

No Side Effects in Render Function

Side effects are confined within a useEffect hook.

The hook controls their execution

```
function GreetBAD(props) {
    const message = `Hello, ${props.name}!`;
    // Calculates output

    // Bad!
    console.log(`Greetings: ${message}`); // Side-effect!

    return <div>{message}</div>;         // Calculates output
}
```

```
import {useEffect} from "react";

function Greet(props) {
    const message = `Hello, ${props.name}!`;
    // Calculates output

    useEffect(() => {
        // Good!
        console.log(`Greetings: ${message}`); // Side-effect!
    }, []);

    return <div>{message}</div>;        // Calculates output
}
```

The side effect will be executed when React decides to [re-]render.

Never? Once? Twice? When?

Rendering is under React control

How To `useEffect`

Very "dense" API

- `useEffect(callback, [dependencies])`

  What to execute — When to execute it

  - `callback`: function containing side-effect logic
  - `useEffect` executes the `callback` function after React has committed the changes to the screen

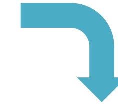
  - `[dependencies]`: an *optional* array of dependencies
  - `useEffect` executes `callback` only if at least one of the `dependencies` have changed between renderings

`useEffect`'s Dependency Array

- **Not provided**: the side-effect runs after *every* rendering
- **An empty array** `[]`: the side-effect runs *once* after the initial rendering
- **Has props or state values** `[prop1, prop2, ..., state1, state2]`: the side-effect runs only *when any dependency value changes*

`useEffect`'s Dependency Array

- **Not provided**: the side-effect runs after *every* rendering



```
import { useEffect } from 'react';

function MyComponent() {
  useEffect(() => {
    // Runs after EVERY rendering
  });
}
```

- **An empty array** [ ]: the

- **Has props or state value**
  `state2]`: the side-effect runs only *when any dependency value changes*

`useEffect`'s Dependency Array

- **Not provided**: the side-effect runs after *every* rendering
- **An empty array `[]`**: the side-effect runs *once* after the initial rendering

- **Has props or state v** `state2]`: the side-e ges

useEffect's Dependency Array

- **Not provided**: the side

- **An empty array [ ]**: th g

```
import { useEffect, useState } from 'react';

function MyComponent({ prop }) {
  const [state, setState] = useState('');
  useEffect(() => {
    // Runs ONCE after initial rendering
    // and after every rendering ONLY IF `prop` or `state` changes
  }, [prop, state]);
}
```

- **Has props or state values** `[prop1, prop2, ..., state1, state2]`: the side-effect runs only *when any dependency value changes*

Side Effects At Mount Time / Update Time

```
<Count num={num}/> <button onClick={()=>setNum(i=>i+1)}>+</button>
```

```
function Count(props) {

    useEffect( ()=>{ console.log(`My static number is ${props.num}`)},  [] ) ;
    // run only once

    useEffect( ()=>{ console.log(`My dynamic number is ${props.num}`)},  [props.num] ) ;
    // run at every change

    return <div>{props.num}</div> ;
}
```

Only when the component is *mounted*.

Will print the *initial value* of the num, only.

At mount time, *plus* every time the num changes.

Will print all the values.

```
My static number is 3          Count.js:5
My dynamic number is 3         Count.js:8
My dynamic number is 4         Count.js:8
My dynamic number is 5         Count.js:8
My dynamic number is 6         Count.js:8
My dynamic number is 7         Count.js:8
My dynamic number is 8         Count.js:8
My dynamic number is 9         Count.js:8
My dynamic number is 10        Count.js:8
```

Side Effects At Mount Time / Update Time

```
<Count num={num}/> <button onClick={()=>setNum(i=>i+1)}>+</button>
```

```
function Count(props) {

    useEffect( ()=>{ console.log(`My static number is ${props.num}`)},  [] ) ;
    // run only once

    useEffect( ()=>{ console.log(`My dynamic number is ${props.num}`)},  [props.num] ) ;
    // run at every change

    return <div>{props.num}</div> ;
}
```

```
My static number is 3          Count.js:5
My dynamic number is 3         Count.js:8
My dynamic number is 4         Count.js:8
My dynamic number is 5         Count.js:8
My dynamic number is 6         Count.js:8
My dynamic number is 7         Count.js:8
My dynamic number is 8         Count.js:8
My dynamic number is 9         Count.js:8
My dynamic number is 10        Count.js:8
```

TIMELINE

- Component Count is created (num=3) and mounted in App
- Function Count is called
- useEffects are registered (not executed)
- The JSX is returned (with 3)
- Component just mounted => run 1st effect
- Component just mounted => run 2nd effect
- …
- User clicks, App updates state, num changes to 4
- Function Count is called for re-rendering (num=4)
- The JSX is returned (4)
- `props.num` changed (prev=3, curr=4) => run 2nd effect
- …

`useState` Meets `useEffect`

- A state variable may be listed as a dependency in an effect
  - When the state changes, the effect is run
  - If the state is updated, but the value does not change, the effect is not run
- Inside a `useEffect` function, you may schedule a state update
  - The state will be updated after the effect is finished (*asynchronously*)
  - If the state value changes, the component is re-rendered

useState Meets useEffect

```
function QuickGate(props) {
    const [open, setOpen] = useState(false) ;

    useEffect(()=>{
        setTimeout(()=>setOpen(false), 500)
    }, [open]) ;

    const openMe = () => {
        setOpen(true) ;
    } ;

    return <div onClick={openMe}>
        {open ? <span>GO</span> : <span>STOP</span>}
    </div> ;
}
```

TIMELINE

- Component `QuickGate` is created and mounted in App
- Function `QuickGate` is called
- `useState` creates state open with default value
- `useEffect` is registered (not executed)
- The JSX is returned (STOP)
- Component just mounted => run effect
    - `setTimeout` is executed: Timeout is set
- Timeout expires
- `setOpen` is executed
- State open becomes `false` => no change
- …
- User clicks
- openMe callback is called
    - `setOpen(true)` executed
- State open becomes `true`
- Component re-renders
- The JSX is returned (GO)
- `useEffect` finds open changed (from `false` to `true`)
    - `setTimeout` is executed: Timeout is set
- …
- Timeout expires
    - `setOpen` is executed
- State open becomes `false`
- Component re-renders
- `useEffect` finds open changed (from `true` to `false`)
- …

`useEffect` Optional Array Caveats

- Make sure the array includes **all** values from the component scope (such as props and state) that change over time and that are used by the effect
- Otherwise, your code will reference stale values from previous renders
  - **Rule**: every value *referenced inside the effect* function should also appear in the dependencies array
    - *arguments* of the functions
    - variables (and functions) accessed through *closure*
- If the array includes variables that *always change* when executing the effect, you risk having an infinite loop

useState & useEffect Meet fetch

```
import { useEffect, useState } from 'react';

function FetchEmployeesByQuery({ query }) {
  const [employees, setEmployees] = useState([]);

  useEffect(() => {
    async function fetchEmployees() {
      const response = await fetch(
        `/employees?q=${encodeURIComponent(query)}`
      );
      const fetchedEmployees = await response.json(response);
      setEmployees(fetchedEmployees);
    }
    fetchEmployees();
  }, [query]);

  return (
    <div>
      {employees.map(name => <div>{name}</div>)}
    </div>
  );
}
```

- useEffect() can perform data fetching side-effect
- When props.query changes, the effect is run
  - Also at the first component mount
- fetchEmployees fetches data from the server
- When the response is available, the employees state is updated
  - Component will re-render

⚠ Note ⚠

- The `callback` argument of useEffect(callback) **cannot** be an async function.
- But you can always define and then invoke an `async` function inside the callback itself
  - Inside the function, you may then use `await`

```
function FetchEmployeesByQuery({ query }) {
  const [employees, setEmployees] = useState([]);

  useEffect(() => {  // <--- CANNOT be an async function
    async function fetchEmployees() {
      // ...
    }
    fetchEmployees(); // <--- But CAN invoke async functions
  }, [query]);

  // ...
}
```

Example

Text: Hello, world

Flipped: plɹoʍ 'olləH

```
import {useEffect, useState} from "react";

function TextFlipper(props) {
    const [text, setText] = useState('') ;
    const [flipped, setFlipped] = useState('') ;

    useEffect( ()=>{
        const fetchFlipped = async (t) => {
            const response = await fetch('/flip?text='+text) ;
            const responseBody = await response.json() ;
            setFlipped( responseBody.text ) ;
        };
        fetchFlipped(text) ;
    }, [text] ) ;

    const handleChange = (ev) => {
       setText(ev.target.value) ;
    } ;

    return <div>
        Text: <input type='text' value={text} onChange={handleChange}/><br/>
        Flipped: {flipped}
    </div> ;
}
```

```
const express = require('express') ;
const flip = require('flip-text') ;

const app = express() ;

app.get('/flip', (req, res) => {
   const text = req.query.text ;
   const flipped = flip(text) ;
   res.json({text: flipped}) ;
});

app.listen(3001, ()=>{console.log('running')})
```

Handling Slow Responses

```
function TextFlipper(props) {
    const [text, setText] = useState('') ;
    const [flipped, setFlipped] = useState('') ;
    const [waiting, setWaiting] = useState(true) ;

    useEffect( ()=>{
        const fetchFlipped = async (t) => {
            const response = await fetch('/flip?text='+text) ;
            const responseBody = await response.json() ;
            setFlipped( responseBody.text ) ;
            setWaiting(false);
        };
        setWaiting(true) ;
        fetchFlipped(text) ;
    }, [text] ) ;

    const handleChange = (ev) => {
       setText(ev.target.value) ;
    } ;

    return <div>
        Text: <input type='text' value={text} onChange={handleChange}/><br/>
        Flipped: {waiting && <span></span>}{flipped}
    </div> ;
}
```

- If HTTP API calls are slow, you can use an extra state to remember whether a call is still ongoing (or if it is been answered)
- The Effect will initially set it to ‘waiting’, and when the response is back, it may be reset to ‘not waiting’
- The component rendering will show in some way that the result is still temporary

Clean-up After Side Effects

- Some side-effects need cleanup: close a socket, clear timers

- If the callback returns a function, then useEffect() considers this as an effect cleanup:

```
useEffect(() => {
  // Side-effect...

  return function cleanup() {
    // Side-effect cleanup...
  };
}, dependencies);
```

- Cleanup works in the following way:
  - After initial rendering, `useEffect()` invokes the callback having the side-effect. `cleanup` function is **not** invoked.
  - On later renderings, before invoking the next side-effect callback, useEffect() invokes the `cleanup` function from the previous side-effect execution (to clean up everything after the previous side-effect), *then* runs the current side-effect.
  - Finally, after unmounting the component, useEffect() invokes the cleanup function from the latest side-effect.

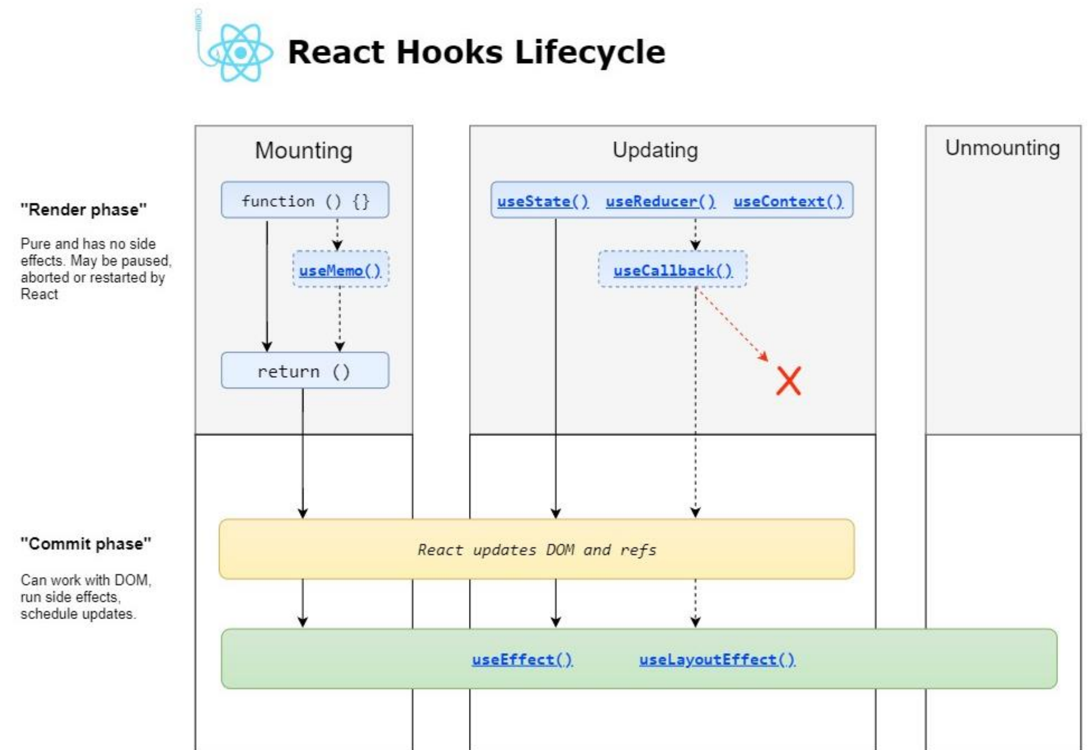
Summary: Four Ways To Call `useEffect`

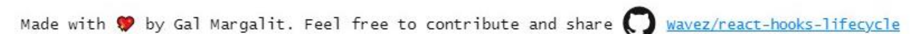
- Once, when the component mounts
  - `useEffect( () => callOnce(), [] )  // empty 2nd arg`
- On every component render
  - `useEffect( () => callAtEveryRender() ) // missing 2nd arg`
- On every component render, **if** some values changed
  - `useEffect( () => callIfAnyDepChange(dep1,dep2), [dep1,dep2])`
- When component unmounts
  - ```
    useEffect( () => { doSomething();
                       return ()=>cleanupFunction(); }, [] )
    ```

https://dev.to/spukas/4-ways-to-useeffect-pf6
    ```

How To Handle Other Lifecycle Situations

- Full lifecycle is more complex
- Other hooks available for particular situations
  - `useLayoutEffect`: it fires *synchronously* after all DOM mutations
  - `useMemo`: returns a *memoized* value (re-computed by a pure function when its parameters change)
  - `useCallback`: returns a *memoized* callback function
- *Not recommended in general*





https://www.robinwieruch.de/react-fetching-data

The Road to Learn React, Chapter “Getting Real with APIs”

Taming the State in React, Chapter “Local State Management”

React as an API Client

HANDLING API CALLS IN REACT

Different Kinds Of State

Application State (or Entity State)

- Retrieved from the back-end
- Should update the back-end
  - on user-initiated CRUD actions
- Should "periodically" check for updates
  - caused by other users, by other open sessions, or by connected systems
- Globally managed, accessible by various components

Presentation State (or View State)

- Not stored in the back-end
  - only in React
- Does not need to persist
- Lives and dies within the controlling Component
- Implemented as **Local State**
  - by using useState

Frequent Use Cases

- How to integrate remote HTTP APIs
- Where/when to load data from remote APIs?
- Delays and “loading…”
- Updating remote data

API Client Classes

- *Recommendation*: keep your fetch methods in a separate JS module (e.g., `API.js`)
- Keeps details of HTTP methods inside the API module
  - API should not depend on React or application state/props
  - Application code should not call fetch or have any HTTP information
- Allows easy swapping with “stub” methods for testing

Conceptual Architecture