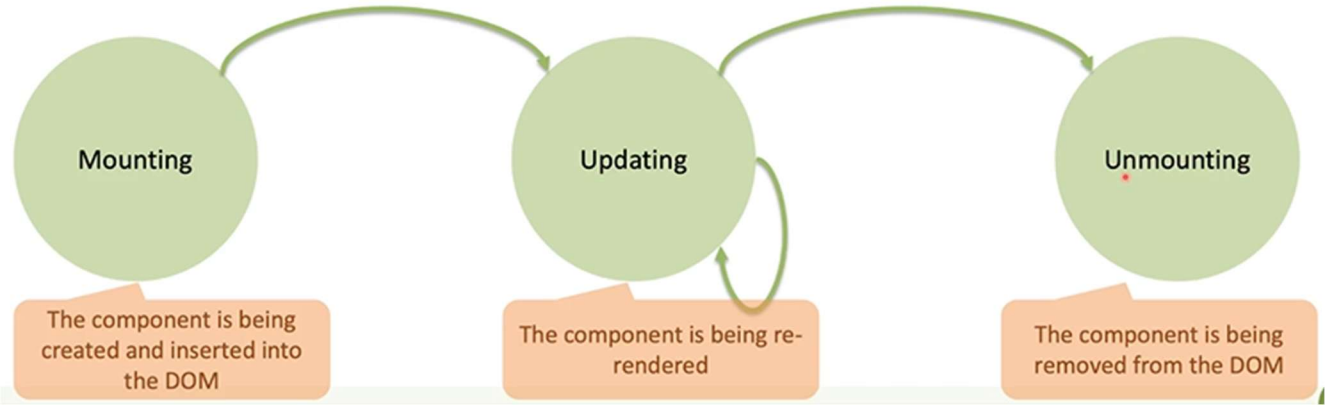
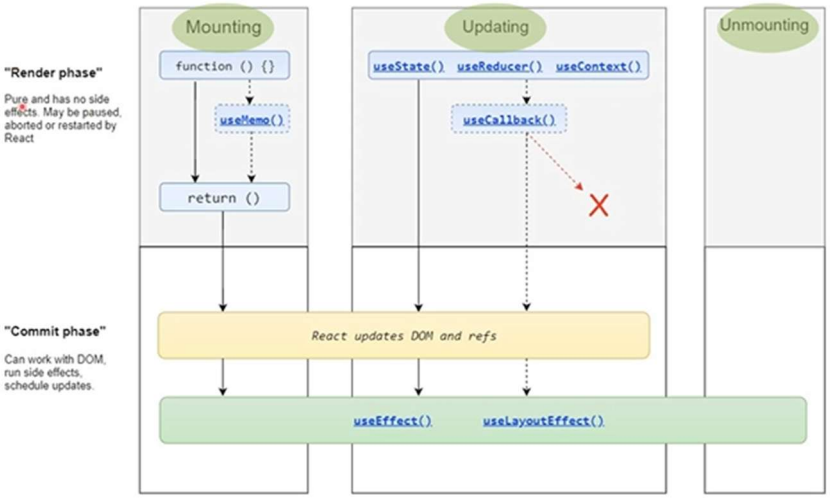
04_08 REACT LIFE-CYCLE



Ci sono diversi Hook che possono agire durante queste fasi:, in particolare ci concentriamo su:

- useState
- useContext

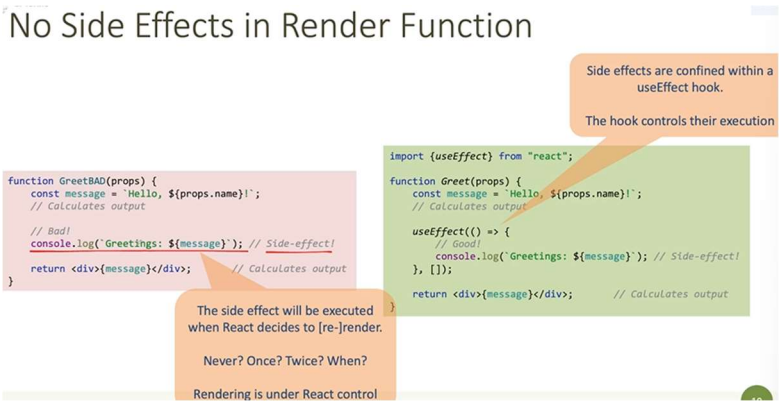


Nella fase di rendering, react agisce sul
virtual DOM e non su DOM effettivo, nella fase di Commit ci sono degli Hook ad hoc:

- useEffect()
- useLayoutEffect()

Nota: nella **fase di commit**, a differebza della fase di rendering, ci possono essere side-effect (calcolo che non ha come obiettivo l'output finale del componente ma influisce qualcos'altro al di fuori del componente funzionale che viene eseguito... es: Console.log)

- data fetching
- log recording
- subscription
- cambiare manualmente il DOM
- Gestire time-out e intervalli di tempo



Come usare useEffect

```
useEffect(callback, [dependencies])
```
What to execute — When to execute it

Callback: indica cosa eseguire

- Funzione contenente tutte le logiche di side-effect
- useEffect esegue la callback dopo che react ha fatto il committ dei cambiamenti sullo schermo (dopoc he il DOM del browser è stato aggiornato).

Array di dipendenze: indicano quando eseguire

- Array di dipendenze opzionale
- Serve per fare in modo che la callback venga eseguita se almeno una delle depencies dell'arrat è cambiata tra due rendering
- Può avere tre stati:
  - **Not provided** → array non c'è
    - Side-effect eseguito ad ogni rendering, comodo se c'è operazione da fare
  - **Empty** array []
    - C'è rendering solo quando il componente viene creato la prima volta o viene fatto il refresh di quel componente
  - **Con props o stati**: side-effect eseguito una volta dopo rendering inziiali e ogni volta ch euno di questi valori di dipendenza cambia

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

```
function MyComponent() {
  useEffect(() => {
    // Runs ONCE after initial rendering
  }, []);
```

```
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]);
}
```

```
<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
```

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.

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

Quando una variabile di stato cambia, effetto eseguito:

- Se stato aggiornato e valore non cambia → effetto non eseguito.

Dentro uno useEfffect si può **schedulare un aggiornamento di stato** → stato viene aggiornato in maniera asincrona:

```
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)

Nota: array dipendenze useEffect:

- Deve includere tutti i valori (props stati) del componente che possono cambiare nel tempo e che sono usati dall'effetto stesso
- Se l'array include variabili che cambiano sempre, rischio loop infinito

Se si considera anche la 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 non prevede che la callback possa avere una async davanti ma va inserita all'interno della callback.

Example

Text: Hello, world
Flipped: plɹoʍ 'ollǝH

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

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

    useEffect( ()=>{
        const fetchFlipped = async () => {
            const response = await fetch('/flip?text='+text) ;
            const responseBody = await response.json() ;
            setFlipped( responseBody.text ) ;
        };
        fetchFlipped(text) ;
    }, [text] ) ;     //chiamata dopo il primo rendering e ogni volta che il campo text cambia

    const handleChange = (ev) => {      //quando c'è un evento, prende un valore passato nel
       setText(ev.target.value) ;       //campo di input e imposta il testo con quel valore
    } ;

    return <div>    //ritorna un div che ha valore text e un onChange e che mostra lo stato Flipped
        Text: <input type='text' value={text} onChange={handleChange}/><br/>
        Flipped: {flipped}
    </div> ;
}
```

server express

```
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')})
```

Per **gestire eventuali risposte lente**, si può aggiungere uno stato aggiuntivo che segnala waiting

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

    useEffect( ()=>{
        const fetchFlipped = async () => {
            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> ;
}
```

nota: di default, mostra orologio. Quando la risposta viene ricevuta e lo stato viene aggiornato, bisogna aggiornare anche lo stato del waiting a false.

Alcuni side-effect necessitano di **clean-up**, ad esempio per chiudere un socket o pulire timer.

- Se la side-effect ritorna una funzione, il side-effect considera quella funzione come una **funzione di pulizia/clean-up**:
    - Dopo il rendering iniziale, viene chiamato useEffect normale, senza clean-up
    - Ad un rendering successivo, prima di invocare l'use-effect, viene chiamata la funzione di clean-up

Nota: non c'è biosgno di use-effect quando si vogliono trasformare i dati per il rendering

- **Esempio:** per filtrare una lista prima di mostrarla, possiamo trasformare uttti i dati a livello alto del componenete senza usare use-effect

Nota: non c'è bisogno di use-effect per gestire eventi che dipendono dagli utenti → vanno gestiti nell'event handler

- **Esempio:** submit su form, click sul bottone

Dunque, sommario 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
  - in addition, it is called when the component mounts
  - `useEffect( () => callIfAnyDepChange(dep1,dep2), [dep1,dep2])`
- When component unmounts
  - `useEffect( () => { doSomething(); return ()=>cleanupFunction(); }, [] )`

Nota: i *metodi di fetch non devono essere mantenuti all'interno dell'use effect ma in un modulo javascript separato (API.js)*.

In un'applicazione ci possono essere 2 tipi di stato:

- **Application state**: quello che recuperiamo dal backend e in caso di aggiornamento, dovrebbe aggiornare il backend
    - Dovrebbe, periodicamente , controllare se ci sono degli aggiornamenti
- **Presentation state**: memorizzato solo in react, non c'è bisogno che persista, vive e muore nel componente che lo controlla

Reidratazione di un application state: prenderlo dalle API

- Deve avvenire al primo rendering (attraverso use effect con array delle dipendenze presente, vuoto)

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

function ShoppingList() {
  const [list, setList] = useState([]);
  const [loading, setLoading] = useState(true) ;

  useEffect(()=> {
    const getItems = async () => {
      const response = await fetch('/api/items');
      const items = await response.json();
      setList(items);
      setLoading(false);
    };
    getItems();
  }, []);

return (<>
  {loading && <span></span>}
  <ul>{list.map((item, i) =>
    <li key={i}>{item}</li>)}</ul>
  </>);
```

Si può anche effettuare una reidratazione per rinfrescare lo stato → aggiornarsi quando avviene un aggiornamento.

Nel caso in cui ci siano tanti browser e un solo server, nel caso in cui il browser 3 aggiorna qualcosa e vogliamo fare in modo che le altre lo sappiano → server deve comunicare i cambiamenti ai vari browser che ha a disposizione.

- *Noi usiamo un modo semplificato che minimizza il problema usando http standard:*
    - *Applicazione web chiede aggiornamenti il più frequentemente possibile*

Deidratazione: estrare application state da react e inviarlo al back-end

- Può avvenire diverse volte

```
const addItem = async () => {
  setList(items => [...items, element]);

  const response = await fetch('/api/items', {
    method: 'POST',
    body: element,
  });
  ...
};

return (...
    <input type="text" value={element} ... }></input>
    <button onClick={addItem}>Add</button>
  ...);
```

→ due aggiornamenti ch efunzionano in parallelo:

- Lista della pagina
- Server in parallelo

Nota: bisogna sperare che tutto vada bene, potrebbero esserci degli errori

```
const addItem = async () => {

  const response = await fetch('/api/items', {
    method: 'POST',
    body: element,
  });
  if (response.ok)
    setList(items => [...items, element]);
  ...
};

return (...
    <input type="text" value={element} ... }></input>
    <button onClick={addItem}>Add</button>
  ...);
```

The state is updated only **after** checking that the request is successfully
=> **No parallel updates!**

Issue: the user of our app will not see the just added item for a while...

Faccio richiesta al server, se la risposta è positiva aggiorno pagina, altrimenti no

➔ *Stato aggiornato solo dopo aver controllato che la richiesta si server sia andata a buon fine. Problema: l'utente della nostra applicazione deve attendere che il tempa risponda, altrimenti non vede subito la risposta*

```
function ShoppingList() {
  const [list, setList] = useState([]);
  const [element, setElement] = useState('');

  useEffect(()=> {
    getItems();
  }, []);

  const getItems = async() => {
    const response = await fetch('/api/items');
    const items = await response.json();
    setList(items);
  };

  const addItem = async () => {
    setElement('');
    setList(items => [...items, `${element} (temp)`]);

    const response = await fetch('/api/items', {
      method: 'POST',
      body: element,
    });
    if (response.ok)
      getItems();
  };

  return (<>
      <ul>{list.map((item, i) => <li key={i}>{item}</li>)}</ul>
      <input type="text" value={element}
onChange={(ev)=>setElement(ev.target.value)}></input>
      <button onClick={addItem}>Add</button>
    </>
  );
}
```

1. **Update** the state in parallel so that the user can see that the operation is *completed*
2. **Mark** the just updated item as *temporary*
   – e.g., by using a different background color, label, ... than the others
3. **Refresh** the *entire* component as soon as the server completes the update operation

Per risolvere, si fa reidratazione+deidratazione: aggiorno stato in parallelo così utente può vedere che l'oggetto è stato aggiunto ma lo informo che è provvisorio

Loop infiniti con useEffect

Usando use effect si possono creare dei loop infiniti a causa di:

- Array delle dipendenze mancante
- Nell'array delle dipendenze si usa un oggetto o un array

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

function CountInputChanges() {
  const [value, setValue] = useState('');
  const [count, setCount] = useState(-1);

  useEffect( () => setCount((c) => (c + 1)) );

  const handleChange = (ev) => setValue(ev.target.value);

  return (
    <div>
      <input type="text" value={value} onChange={handleChange} />
      <div>Number of changes: {count}</div>
    </div> );
}
```

→ **dipendenza mancante** e incremento lo stato count che causa re-rendering infinito.

Modo corretto ←

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

function CountInputChanges() {
  const [value, setValue] = useState('');
  const [count, setCount] = useState(-1);

  useEffect( () => setCount((c) => (c + 1)), [value] );

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

  return (
    <div>
      <input type="text" value={value}
onChange={handleChange} />
      <div>Number of changes: {count}</div>
    </div> );
```

```
function CountSecrets() {
  const [secret, setSecret] = useState({ value: "", countSecrets: 0 });

  useEffect(() => {
    if (secret.value === 'secret')
      setSecret(s => ({...s, countSecrets: s.countSecrets + 1}));
  }, [secret]);

const onChange = (ev) => { setSecret(s => ({ ...s, value: ev.target.value })); };

return ( <div>
    <input type="text" value={secret.value} onChange={onChange} />
    <div>Number of secrets: {secret.countSecrets}</div>
  </div>
);
```

→ questo useState, **crea un nuovo oggetto** ogni volta che viene chiamato (Anche se contenuto non chiamato → quindi useEffect chiamato ogni volta)

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

function CountSecrets() {
  const [secret, setSecret] = useState({ value: "",
countSecrets: 0 });

  useEffect(() => {
    if (secret.value === 'secret')
      setSecret(s => ({...s, countSecrets: s.countSecrets
+ 1}));
  }, [secret.value]);

const handleChange = ({ target }) => { setSecret(s => ({
...s, value: target.value })); };

return ( <div>
    <input type="text" value={secret.value}
onChange={handleChange} />
    <div>Number of secrets: {secret.countSecrets}</div>
  </div>
```

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

function ShoppingList() {
  const [list, setList] = useState([]);

  useEffect(()=> {
    const getItems = async () => {
      const response = await fetch('/api/items');
      const items = await response.json();
      setList(items);
    };
    getItems();
  }, []); // don't use: [list]

return (
  <ul>{list.map((item, i) => <li
key={i}>{item}</li>)}</ul>
  );
}
```

→ nell'array di dipendenze, non usare un array di dipendenze, al massimo si può usare la lunghezza di un array o cose simili ma non l'array stesso.