

<WA/>

2024

# React Life Cycle

## Making React Components Alive

Fulvio Corno  
Luigi De Russis  
Enrico Masala



VectorStock®

VectorStock.com/22718058



<https://react.dev/learn/lifecycle-of-reactive-effects>

<https://react.dev/learn/synchronizing-with-effects>

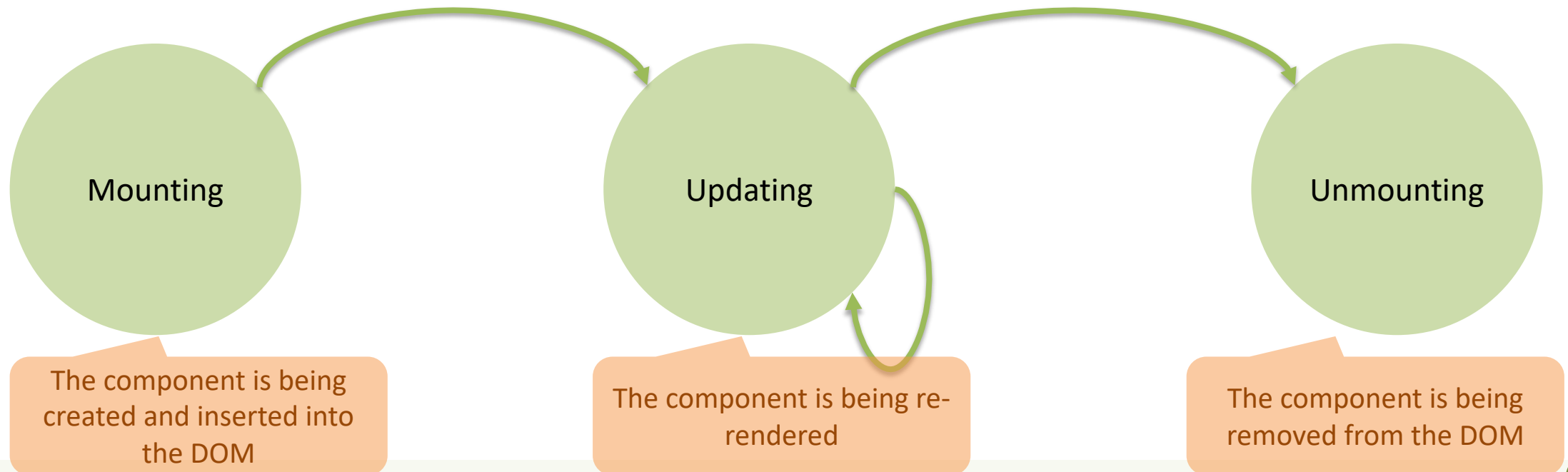
Full Stack React, Chapter “Advanced Component Configuration with props, state, and children”

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

## "Render phase"

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

### Mounting

```
function () {}
```

```
useMemo()
```

```
return ()
```

### Updating

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

```
useCallback()
```



### Unmounting


## "Commit phase"

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

React updates DOM and refs

```
useEffect()
```

```
useLayoutEffect()
```

Made with ❤️ by Gal Margalit. Feel free to contribute and share  [wavez/react-hooks-lifecycle](https://github.com/wavez/react-hooks-lifecycle)

# 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 ...
  - Scheduling additional actions when some state changes
  - Manually changing the DOM in React component
  - 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.



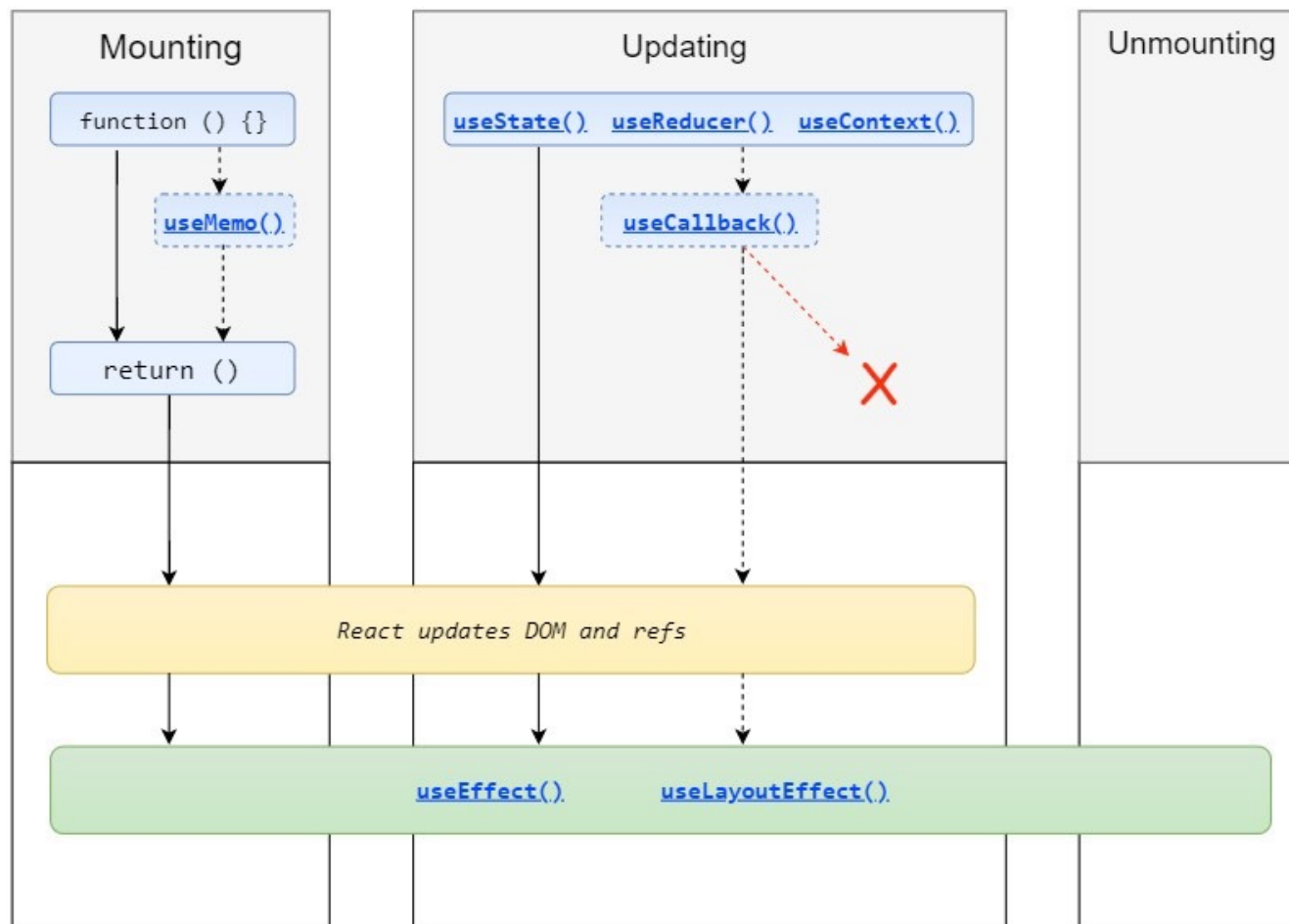
# React Hooks Lifecycle


➔

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

No side effects in the render phase ➔

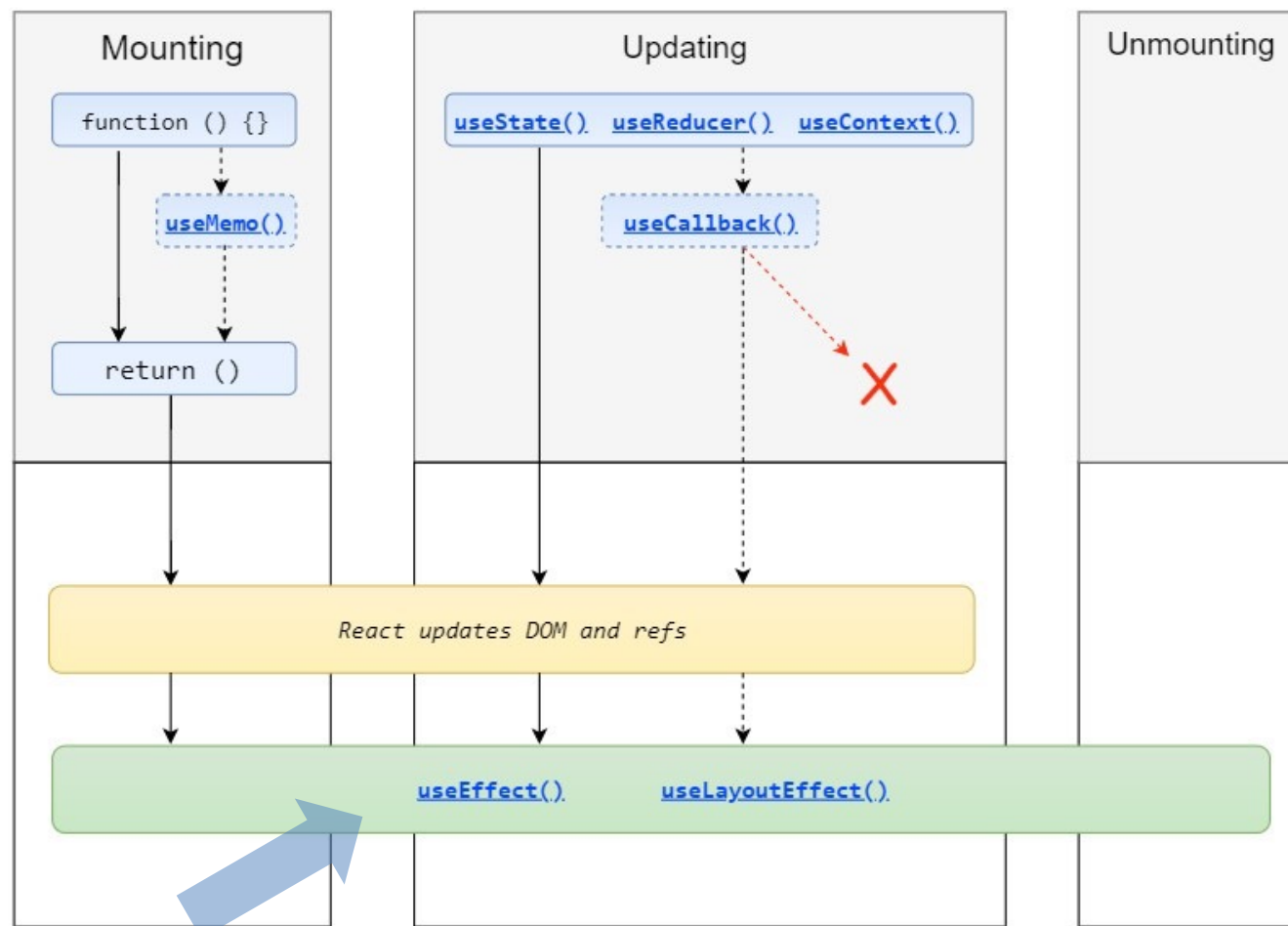
**"Commit phase"**  
Can work with DOM, run side effects, schedule updates.



Made with ❤️ by Gal Margalit. Feel free to contribute and share  [wavez/react-hooks-lifecycle](https://github.com/wavez/react-hooks-lifecycle)



# React Hooks Lifecycle



No side effects  
in the render phase

Side effects run after  
rendering and DOM  
updating

Made with ❤️ by Gal Margalit. Feel free to contribute and share [wavez/react-hooks-lifecycle](https://github.com/wavez/react-hooks-lifecycle)





Full Stack React, “Appendix C: React Hooks”

React Handbook, chapter “Hooks”

<https://react.dev/reference/react/useState>

<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

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

Side effects are confined within a `useEffect` hook.

The hook controls their execution

```
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  
}
```

# 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
- the callback may return a “cleanup function” (see later...)
- `[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 *once* after the initial rendering, then 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 values:** the side-effect runs *whenever* the props or state values change



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

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

# useEffect's Dependency Array

- Not provided: the side effect runs every time the component renders
- An empty array `[]`: the side effect runs only once after the initial render

```
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 1<sup>st</sup> effect
- Component just mounted => run 2<sup>nd</sup> 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 2<sup>nd</sup> 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 Dependency 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: pluoM 'olləH


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

function TextFlipper(props) {
  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] ) ;

  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 () => {
      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><img alt="loading spinner" data-bbox="258 755 275 775"/></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**:
- 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.

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

# You Might Not Need an Effect

- Two common cases where you do not need effects:
  1. **To transform data for rendering.** For example, if you want to filter a list before displaying it, you can transform all the data at the top level of your components.
  2. **To handle user events.** Handle user events in the corresponding event handler.
- Instead, use effects when you want to synchronize with external systems.
- Further examples:
  - <https://react.dev/learn/you-might-not-need-an-effect>

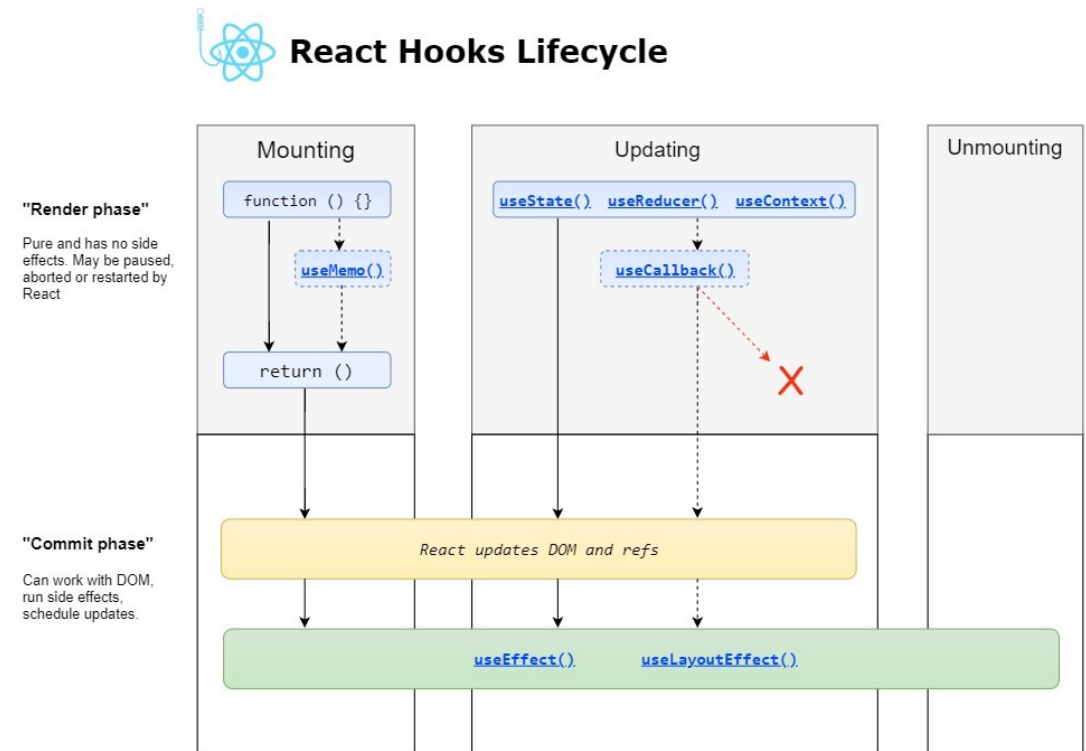
# Summary: Four Ways To Call useEffect


- 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
  - in addition, it is called when the component mounts
  - `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*



Made with ❤️ by Gal Margalit. Feel free to contribute and share  [wavez/react-hooks-lifecycle](https://github.com/wavez/react-hooks-lifecycle)



<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

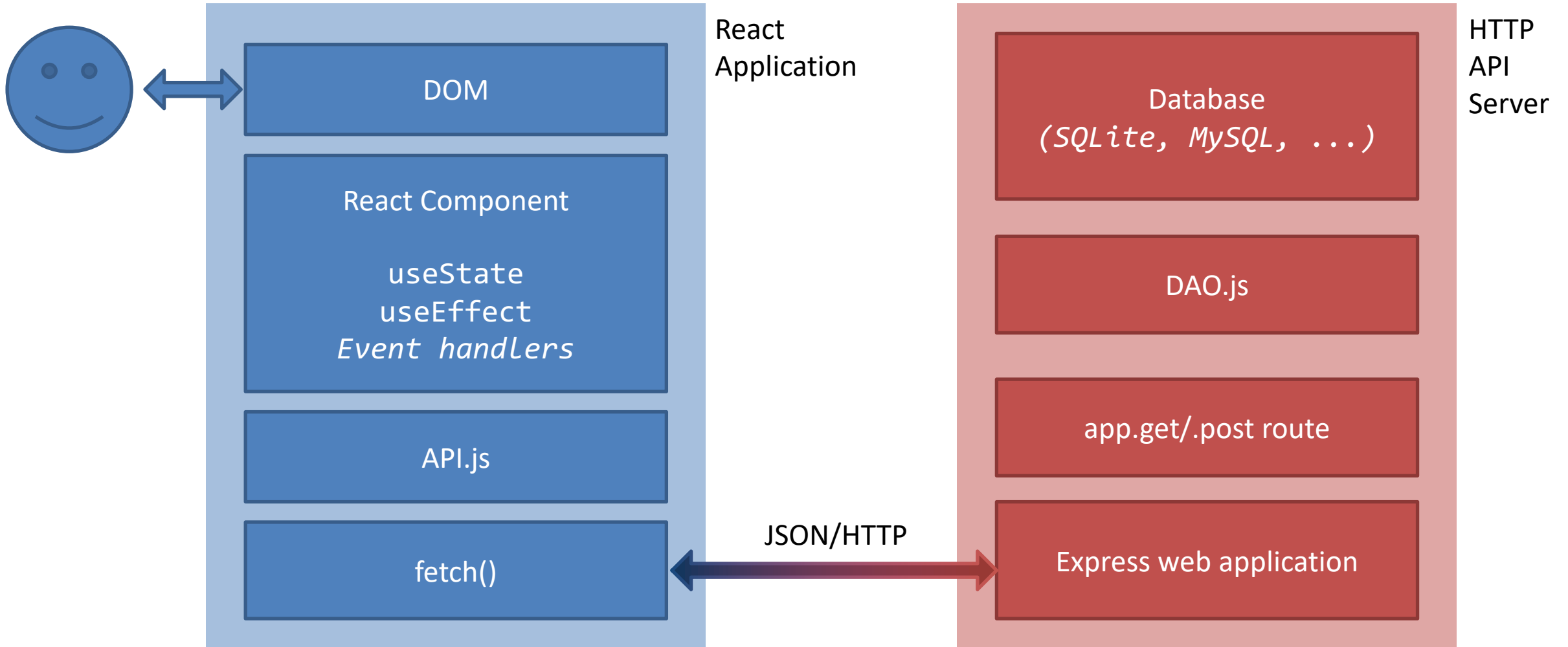
- How to integrate remote HTTP APIs
- Where/when to load data from remote APIs?
- Delays and “loading...”
- Updating remote data

# 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



# Rehydrating And Dehydrating

- Application State is retrieved via HTTP APIs
  - e.g., from info stored in a DB and accessed through the HTTP API server
- **Rehydrating** the Application State means getting it from the HTTP APIs
  - Must happen when the React application mounts
  - **Best place:** inside a `useEffect(fn, [])` method
- Rehydrating should **also** happen when we want to “refresh” the state
  - The React app cannot know whether *others* changed the info provided by the API server
- **Dehydrating** the Application State means extracting it from the React application
  - May happen several times during the React app execution
  - Should happen whenever something (in the Application State) is modified

# Rehydrating At Mount Time

- Very similar to what we saw before...
- Rehydrating may **require some time**, while the component renders “empty” (with the initial state)
  - use the approach for *handling slow responses*
  - i.e., a **loading/waiting** local state

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

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

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

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

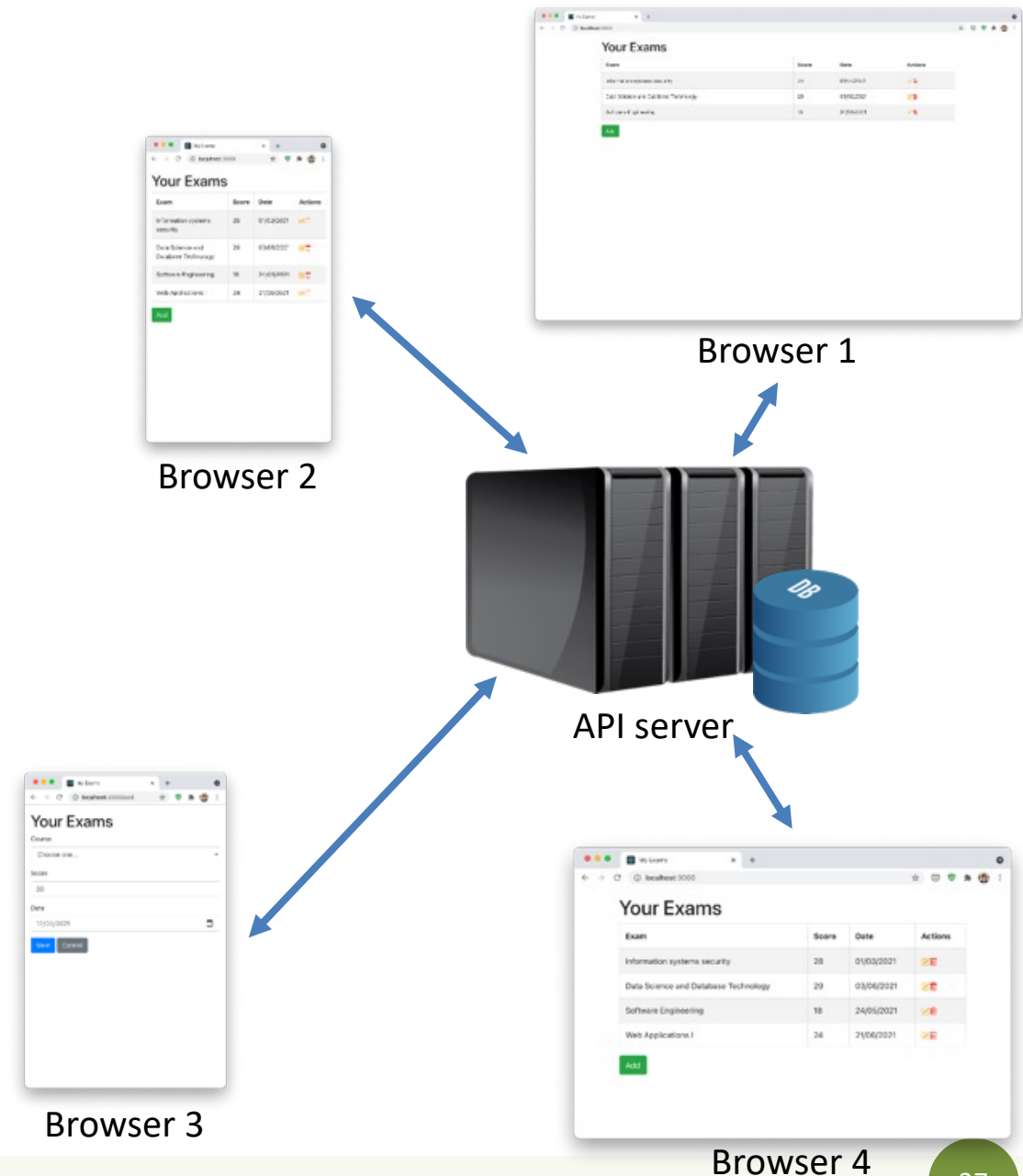
# Rehydrating To Refresh The State

- **Once you know** that something has been changed in the API server, you can use `useEffect()` as before
  - with one or more dependencies, e.g., `[dep1, dep2]`
- **Beware:** two problems might arise
  - the "n-clients problem"
  - infinite loops



# The “N-Clients Problem”

- We are creating a web application that will be opened on *multiple* browsers at once
  - They read and write info from a *unique* API server, however
- What happens in the web app running in, e.g., Browser 1 when Browser 3 updates something in the API server?
  - How can one web app know that *someone else* changed *something* in the server?



# The “N-Clients Problem”

## The Better-Than-Nothing Solution

- The *web app* asks for data as **frequently** as possible
  - when it loads a new page/view
  - after adding/updating/removing something
  - periodically (i.e., polling)
  - ...
- **Not a solution:** just a way to *minimize* the problem

## The Real Solution

- The *server* communicates changes as soon as they **appear**
  - to *all* the current consumers of its information
- **Out of scope** for this course
  - unfortunately!
- For the curious:
  - WebSockets (e.g., <https://socket.io>)
  - PubSub mechanisms

# Infinite Loops with `useEffect`

- One of the main pitfalls that might happen with `useEffect`
  - infinite loops both in rendering and in external (e.g., HTTP) calls
  - especially when `useEffect` is used with `useState`
- Two significant cases:
  1. The dependency array is `missing`, but it should not
  2. One of the items in the dependency array is a JavaScript `Object` `{ }` or `Array` `[ ]`

Examples from: <https://dmitripavlutin.com/react-useeffect-infinite-loop/>

# Example: Missing Dependencies

What is wrong,  
here?

```
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> );
}
```



# 1. Set Up Dependencies Correctly

- Without the dependency (no dependency array), the code in the example will re-render the component **forever**
- It is also a clear error:
  - the update of count **depends** on the change of value

```
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> );
}
```

# Example: Objects as Dependencies

What is wrong,  
here?

```
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>
  );
}
```

## 2a. Avoid Objects As Dependencies

- **Problem:** secret as a dependency!
- Inside `useEffect`, when the input value equals 'secret', `setSecret()` is called
- `setSecret()` increments the secrets counter, but also **creates** a new object
  - secret is now a new object, and the dependency has changed
- So `useEffect` invokes again the callback that updates the state, and a new secret object is created again, etc.
- **How to solve:** do not use objects as dependencies!

```
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>
  );
}
```

## 2b. Avoid Arrays As Dependencies

```
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();
  }, [list]); // don't use: [list]

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

- The same issue might happen with arrays...
- ... so, it is better to avoid arrays as dependencies
  - you can use an empty dependency array []
  - or an additional state to trigger useEffect
  - or any item in the array, the length property (if appropriate), or ...

# Dehydrating During Updates

```
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>  
  ...);
```

The two updates (remote API, local state) run **in parallel**.

**Optimistic** state update: it assumes that remote state will be updated without errors => **Risky!**

# Dehydrating During Updates – Alternative

```
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...

# During Updates: Dehydrate And Rehydrate

```
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



Full Stack React, “Appendix C: React Hooks”

React Handbook, chapter “Hooks”

<https://reactjs.org/docs/hooks-rules.html>

Peeking Under the Hood

# THE RULES OF HOOKS



# Quiz

- What is the “magic” behind `useState`?
- How can the same function return different state variables?
- How can the values be persisted across function calls?

```
function Example(props) {  
  
  [hidden, setHidden] = useState(true) ;  
  [count, setCount] = useState(0) ;  
  [mode, setMode] = useState('view') ;  
  
  . . .  
  
  setHidden(false) ;  
  
  . . .  
  setCount( c => c+1 ) ;  
  
  . . .  
  setMode('edit') ;  
  
  . . .  
  
}
```

# Answer

- React associates to each functional component an array of Hook “slots”
  - Slots are stored with the function, therefore they are persistent
- Each time you call a Hook, a new “slot” is used
  - The first time, it is created
  - The other times, it is reused

```
function Example(props) {  
  
  [hidden, setHidden] = useState(true) ;  
  [count, setCount] = useState(0) ;  
  [mode, setMode] = useState('view') ;  
  
  . . .  
  
  setHidden(false) ;  
  . . .  
  setCount( c => c+1 ) ;  
  . . .  
  setMode('edit') ;  
  . . .  
  
}
```

# Corollary

- React must “know” which functions may host Hooks
- Hooks must always be called in the same order each time a component renders

```
function Example(props) {  
  
  [hidden, setHidden] = useState(true) ;  
  [count, setCount] = useState(0) ;  
  [mode, setMode] = useState('view') ;  
  
  . . .  
  
  setHidden(false) ;  
  . . .  
  setCount( c => c+1 ) ;  
  . . .  
  setMode('edit') ;  
  . . .  
  
}
```

# Hook Usage Rules

- Only Call Hooks at the Top Level
  - Always call Hooks at the top level of your React function
  - Do not call Hooks inside loops, conditions, or nested functions
- Only Call Hooks from React Functions
  - Do not call Hooks from regular JavaScript functions
  - You may call Hooks from React function components
  - You may call Hooks from custom Hooks

<https://reactjs.org/docs/hooks-rules.html>

# License

- These slides are distributed under a Creative Commons license “**Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)**”
- **You are free to:**
  - **Share** — copy and redistribute the material in any medium or format
  - **Adapt** — remix, transform, and build upon the material
  - The licensor cannot revoke these freedoms as long as you follow the license terms.
- **Under the following terms:**
  - **Attribution** — You must give [appropriate credit](#), provide a link to the license, and [indicate if changes were made](#). You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
  - **NonCommercial** — You may not use the material for [commercial purposes](#).
  - **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the [same license](#) as the original.
  - **No additional restrictions** — You may not apply legal terms or [technological measures](#) that legally restrict others from doing anything the license permits.
- <https://creativecommons.org/licenses/by-nc-sa/4.0/>

