p2-caratteristiche-componenti

2. Features of components

> - **React for Dummies**, I.T.T. "Blaise Pascal" @ Cesena, IT
> - Written by: Nicholas Magi - nicholas.magi24[@]gmail.com

Table of contents

2.1 Component Lifecycle

In older React versions you would have designed a component as a **javascript class**. You would have extended a `React.Component` class in your custom component in order to have all the common properties a React component has. This approach has its pros and cons and particular features, like an **explicit lifecycle management**.

A **class component** lifecycle is made of 3 phases - which determine the **birth**, **update** and **death** of the component. These are:

1. **Mounting** := this occurs when the component is being created and placed into the DOM.
2. **Updating** := occurs when a prop or state of the component changes.
3. **Unmounting** := occurs when the component is being removed from the DOM.

In **class components** these phases could be handled with lifecycle methods provided by the superclass `React.Component`, which are:

1. `componentDidMount()`
2. `componentDidUpdate(prevProps, prevState)`
3. `componentWillUnmount()`

Since we'll focus more on **functional components**, you need to know that there aren't direct ways to handle these phases, but there's a hook (*which we'll discuss later*) called `useEffect()` that can mimic the three lifecycle phases.

1. **Mounting**:

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

2. **Updating**:

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

3. **Cleanup**: To perform cleanup actions return a function from your effect,

```
useEffect(() => { return () => {...} })
```

2.2 Conditional Rendering

A component can be rendered according to a specific boolean condition. There are 3 possible ways to do it:

1. With **If statements**
2. With **Ternary operator**
3. With **&& operator**

Check all the examples in `React-Examples` project.

2.3 Events Handling

HTML tags can register some events listener, like a button click (`onClick()`) or a form submit (`onSubmit()`). There are 3 possible ways to handle an event:

1. Declaring an event handler - must be **inside** the component implementation - and passing it to the corresponding event.
2. Declare an event handler directly inside the component event.
3. Use a lambda function inside the component event.

Check all the examples in `React-Examples` project.

2.4 State: A Component's Memory

You have designed your static components with their beautiful UI and you want to add a bit of interactivity to them. Consider a *SimpleGallery* component, which shows a carousel of pictures - each followed by a description. You want to animate the buttons that let the user navigate all the pictures in the carousel, which are stored in an array of objects.

The most intutive approach would be to use an **index variabile** which indicates the position in the array of the picture shown, and design an **increment** and **decrement** functions associated with the corresponding buttons that, as the names suggest, increment or decrement the index variable.

However, this approach **would not work**. And there are two reasons that describe this "strange" behaviour:

1. **Changes to local variables does not trigger a component's re-rendering**;
2. **Local variable does not persist between renders**.

> ⓘ **Our need**
>
> We must say to React to **remember the `idx` value after re-renders**, and this has to do with *state*.

Those reasons introduce the `useState()` hook, the most important hook that manages the state of a component.

2.4.1 `useState()` Hook

Before talking about `useState()` you need to know the meaning of "Hook", because it's a key feature in React - and we'll use this term a lot from now on.

ⓘ Definition

A "Hook" is just a **function** implemented inside the React library that lets the developer use different features of a component.

ⓘ Nice to remember

These functions are called "Hook" because they let you "hook into" (*connect to*) a feature of component, like its state.

ⓘ Info

Note that every Hook name starts with *"use"*.

There are many Hooks inside React, each specifically designed to let the developer deal with a particular feature of a component (`useEffect(), useState(), useMemo(), useRef()` ...). You can find the library standard ones in the official guide: https://react.dev/reference/react/hooks.

When you need to add the `useState()` function inside your React components, first you need to **import it**. At the beginning of the code page, add

```
import { useState } from "react"
```

`useState()` will always return two values:

1. The state you want to use.
2. The **setter function** needed to set the value of the state.

In the case of the first example, we need the **index variable** to be a state of the component *SimpleGallery*, so we would write as follows:

```
const [ idx, setIdx ] = useState(0)
```

Notice that we have passed `0` as parameter of `useState()`. This represents the initial value of `idx`.

ⓘ Info

💡 Traditional naming conventions suggest to always write:

```
const [ something, setSomething ] = useState( ... )
```

2.4.2 State characteristics

1. State is **local** and **isolated**: no one outside the component can see or refer to its state. You can have multiple instances of the same component in your application, and they will all have their different states.
2. You can have **multiple states** inside a component - each one of them will have its variable and setter function.
3. Changing the state of a component (*with* `useState()`) **triggers its re-render**.
4. Changing the state of a component is allowed only at the top level of a component or another Hook - **don't change it inside conditions, loops or nested functions**.
5. Always choose to **replace** an array or an object in a state rather than **mutate** it.

2.5 Styling a Component

There are 3 possible ways to style a component:

1. **Inline styling**: as you would do in a traditional HTML tag, you'd write your styles in the `style` attribute. Useful for few and immediate styles, can get out of hand and become unreadable very quickly. Since we write JSX and not pure HTML, remember to pass an **object** containing the same CSS rules you would have written in pure CSS - but remove hyphens and use camelCase.

```
return <div
    style={
        {
            backgroundColor: "red",
            borderRadius: "50px",
            ... : "..."
        }
    }>
        ...
    </div>
```

2. **CSS Modules**: you can create custom and local CSS classes for your component defined in a file named `<component_name>.module.css`. Remember to define classes only, whose names does not contain **hyphens** — replace them with **underscores**. The module would be imported in the top lines of your code.

3. **Styled-components/CSS-in-JS**: Libraries allowing CSS styles in JavaScript, giving dynamic capabilities – we won't consider this option.

Check all the examples in `React-Examples` project.

2.6 Synchronising with Effects

Effects are some kind of event which happen during the lifecycle of a component. We introduced them in **2.1 Component Lifecycle**, where you've been left with '... *which we'll discuss later*'. Now it's time to talk about them.

`useEffect()` is the second Hook we introduce, and one of the most important to keep in mind if you're willing to build a simple application — along with `useState()`.

To use it inside a component, first you need to **import it**.

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

It's just a function that takes a callback function as the first parameter and an optional array of dependencies as the second one. It optionally returns a function which fires once the component gets removed from the DOM.

```
useEffect(callback, dependencies?) -> function?
```

Remember that what's written inside a `useEffect()` Hook runs every commit to the UI tree — in other words, **whenever the component gets placed in the DOM**.

> ⓘ **Q&A**
>
> **Q**: When do I need to use Effects in my application?
> **A**: You use an Effect when you need to synchronise your application with an external service — like an API, for instance. You can fetch data inside

Recall what the section **2.1 Component Lifecycle** says about the lifecycle of a **functional component**:

1. **Mounting**:

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

2. **Updating**:

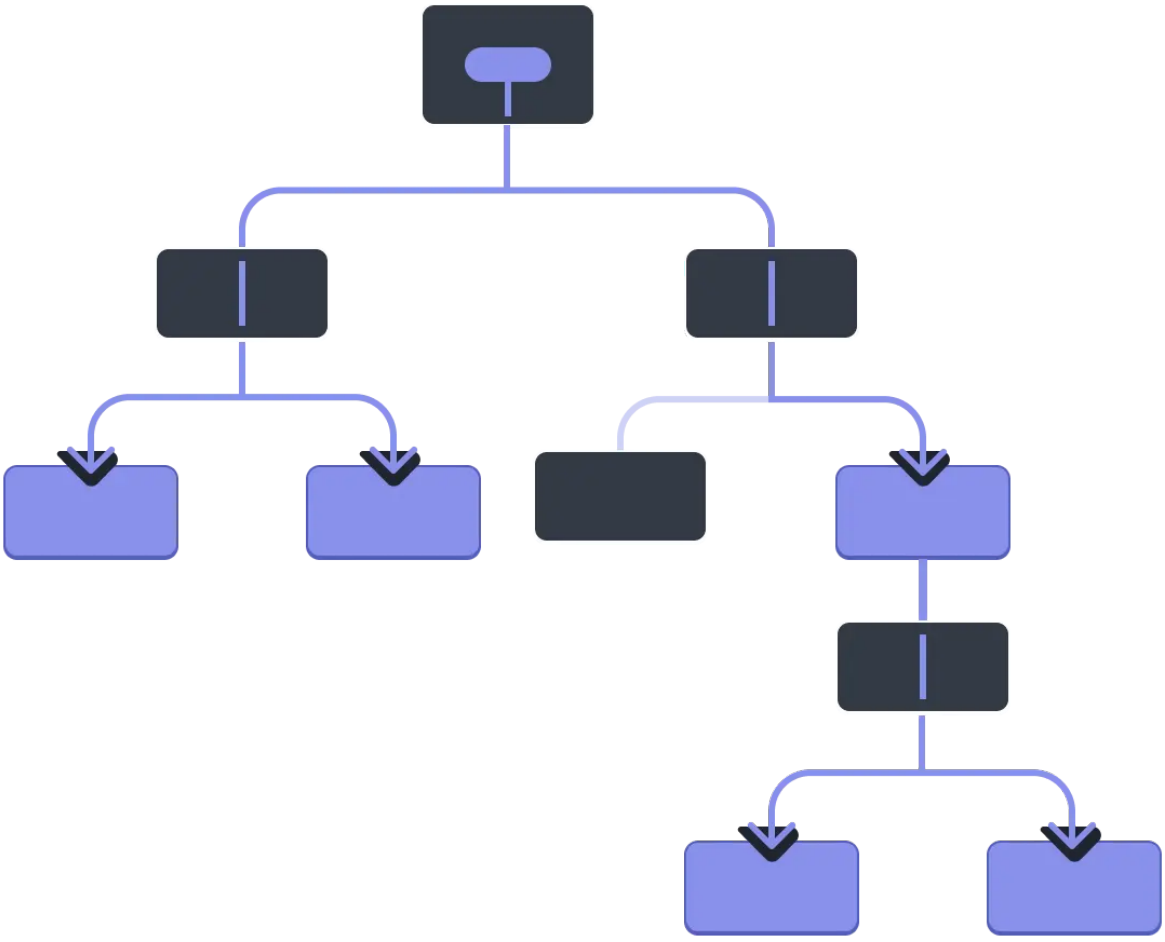
```
useEffect(() => {
        ...
        }, [ yourDependency ])
```

3. **Cleanup**: To perform cleanup actions return a function from your effect,

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

2.7 Drilling Props

It's pretty common to pass props from one parent component to its children, but this can become tedious and verbose if there are **multiple layers** in the middle of the passage.



To simplify this task React provides *Context*, a feature that allows a component to share information to its children without passing it via props.

We will not deep dive too much into this topic, but it is important to remember how Context works because **routing** relies on it!

The hook responsible for this feature is `useContext()`, which needs to be imported to be used inside your code (like the others):

```
import { useContext } from "react"
```

It works as following:

1. You need to define a context (possibly in an external `.js` file), like so:

```
import { createContext } from "react"

const INITIAL_LEVEL = 0
export const LevelContext = createContext(INITIAL_LEVEL)
```

2. You need to wrap the **component** you want to set as the **provider** of the context with a special component called '`<LevelContext.Provider>`'

```
import { useContext } from "react"
import { LevelContext } from "./LevelContext"

export default function Section({ children }) {
        const level = useContext(LevelContext)
        return <article className="border border-1 rounded-1 p-3">
                <LevelContext.Provider value={level + 1}>
                        { children }
                </LevelContext.Provider>
        </article>
}
```

In this example, the `Section` component is the provider of the context called `LevelContext` — **with the value of the context set to `level`** . This means that every children under it can **consume** the value of `level`.

3. Finally, you must declare where the context gets consumed. In this case, the headings need to know the level of nesting they're in, so they will be the **consumer** of the `LevelContext` **context**.

```
import { useContext } from "react"
import { LevelContext } from "./LevelContext"

export default function Heading({ children }) {
        const level = useContext(LevelContext)
        ...
}
```