Puntos importantes:

Notar que el componente projects\practice-project-module-8\src\components\UI\Card.js esta recibiendo el className usando props y que estamos combinando este className con el que se definió internamente en el componente:

**<**div className**={**`${style.card} ${props.className}`**}>{**props**.**children**}</**div**>**

Además ver el uso de {props.children}, que es el contenido que esta declarado dentro de Card en el compoente AddUser:

**<**Card className**={**style**.**input**}>**

**<form** onSubmit**={**addUserHandler**}>**

**<**label htmlFor**=**"username"**>**User name**</**label**>**

**<**input id**=**"username" type**=**"text" **/>**

**<**label htmlFor**=**"age"**>**Age **(**years**)</**label**>**

**<**input id**=**"age" type**=**"number" **/>**

**<button** type**=**"submit"**>**Add user**</button>**

**</form>**

**</**Card**>**

El mismo principio de {props.children} es usado en el componente Button:

**<button**

className**={**style**.button}**

type**={**props**.**type **||** "button"**}**

onClick**={**props**.**onClick**}**

**>**

**{**props**.**children**}**

**</button>**

PORTAL:

React v16.0 brought a concept of Portal that provides a way to transport a piece of UI into some other locations on to the DOM Tree ( even outside the parent hierarchy ). You put your component at one place. After rendering, It appears to be somewhere else on the Actual DOM.

**What is the use?**

Creating Modals / TootTips. (Especially when the modal or the tooltip would break out of the parent div)

**What’s Special?**

The teleport ( or the transfer ) actually happens in the Actual DOM tree not the React Tree. React Tree will still hold the Modal where you rendered it.

\*Important : This means even though a portal can be anywhere in the Actual DOM tree, it behaves like a normal React child in every other way. Features like context and this work exactly the same regardless of whether the child is a portal, as the Modal and its children still exist in the React Virtual DOM tree regardless of it’s position in the Actual DOM tree. This includes event bubbling. An event fired from inside a portal will propagate to ancestors in the containing React tree, even if those elements are not ancestors in the ActualDOM tree.

**useRef:**

You can acces directly any DOM component by using useRef, it is ok to do it when reading values, fot other cases let React to manage access to DOM instead.

Graphical user interface, text

Description automatically generated

**Effects**

Graphical user interface, diagram

Description automatically generated

**What to add & Not to add as Dependencies**

You must add all "things" you use in your effect function **if those "things" could change because your component (or some parent component) re-rendered.** That's why variables or state defined in component functions, props or functions defined in component functions have to be added as dependencies!

Here's a made-up dummy example to further clarify the above-mentioned scenarios:

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

**let** myTimer**;**

**const** MyComponent **=** **(**props**)** **=>** **{**

**const** **[**timerIsActive**,** setTimerIsActive**]** **=** useState**(false);**

**const** **{** timerDuration **}** **=** props**;** // using destructuring to pull out specific props values

useEffect**(()** **=>** **{**

**if** **(!**timerIsActive**)** **{**

setTimerIsActive**(true);**

myTimer **=** **setTimeout(()** **=>** **{**

setTimerIsActive**(false);**

**},** timerDuration**);**

**}**

**},** **[**timerIsActive**,** timerDuration**]);**

**};**

In this example:

* timerIsActive is **added as a dependency** because it's component state that may change when the component changes (e.g. because the state was updated)
* timerDuration is **added as a dependency** because it's a prop value of that component - so it may change if a parent component changes that value (causing this MyComponent component to re-render as well)
* setTimerIsActive is **NOT added as a dependency** because it's that **exception**: State updating functions could be added but don't have to be added since React guarantees that the functions themselves never change
* myTimer is **NOT added as a dependency** because it's **not a component-internal variable** (i.e. not some state or a prop value) - it's defined outside of the component and changing it (no matter where) **wouldn't cause the component to be re-evaluated**
* setTimeout is **NOT added as a dependency** because it's **a built-in API** (built-into the browser) - it's independent from React and your components, it doesn't change

Cleanup function:

<https://dmitripavlutin.com/react-cleanup-async-effects/>

**useReducer:**

Timeline

Description automatically generated

Diagram

Description automatically generated

<https://hswolff.com/blog/why-i-love-usereducer/>

**When do you use React.useReducer instead of React.useState?**

This is quite simple actually.

Use React.useReducer when

* The state value is an object or an array.
* When the logic to update state is super complex
* You need for a more predictable, and maintainable state architecture

A great example that is a good use for React.useReducer are forms.

Form data is typically bundled up in a key-value object. It then gets JSON.stringify() to get sent to an API endpoint.

On top of the form data, you have form status such as pending, success, error.

And those statuses may come with messages or response data.

**Use React.useState when**

* The state value is a [primitive value](https://linguinecode.com/guides/javascript/beginners/types-intro)
* [Simple UI transitions](http://linguinecode.com/post/how-to-add-react-animation)
* Logic is not complicated and can stay within the component

<https://linguinecode.com/post/react-usereducer-vs-usestate>

**Context Api**

Se utiliza esta api para evitar que los componentes reciban propiedades o métodos que no van a usar pero que solo van ser enviados a otros componentes.

Text

Description automatically generated with medium confidence

In order to use Context, you need to follow the next steps:

Define a object that stores it

Text

Description automatically generated

Provide it

Text

Description automatically generated

Consume it

Text

Description automatically generated

You can use useContext hook as well

Diagram

Description automatically generated

Diagram

Description automatically generated

**How React works?**

Graphical user interface, diagram

Description automatically generated

Graphical user interface, diagram, text

Description automatically generated

Diagram

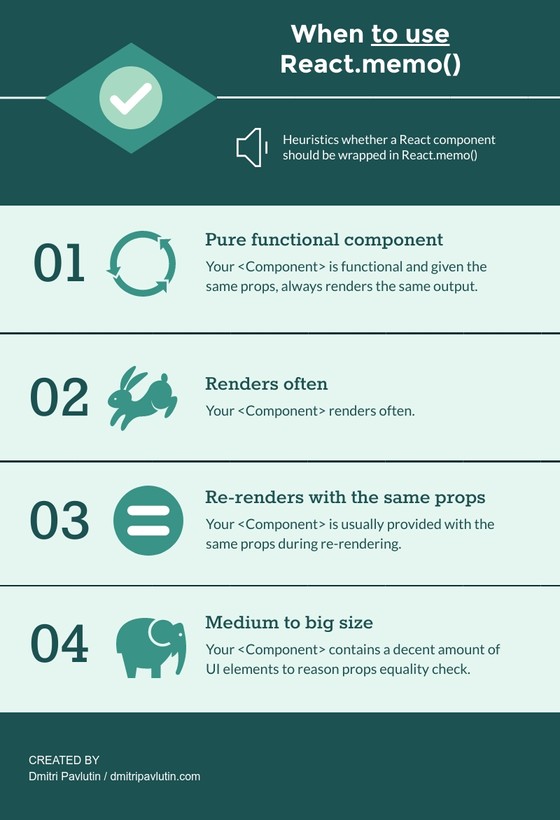
Description automatically generated

**React.memo**

When deciding to update DOM, React first renders your component, then compares the result with the previous render. If the render results are different, React updates the DOM.

Current vs previous render results comparison is fast. But you can *speed up* the process under some circumstances.

When a component is wrapped in React.memo(), React renders the component and memoizes the result. Before the next render, if the new props are the same, React reuses the memoized result *skipping the next rendering*.



**When to avoid React.memo()**

If the component *isn’t heavy* and usually *renders with different props*, most likely you don’t need React.memo().

**The purpose of useCallback()**

Different function objects sharing the same code are often created inside React components:

function MyComponent() {

// handleClick is re-created on each render

const handleClick = () => {

console.log('Clicked!');

};

// ...

}

handleClick is a different function object on every rendering of MyComponent.

Because inline functions are cheap, the re-creation of functions on each rendering is not a problem. *A few inline functions per component are acceptable.*

But in some cases you need to maintain a single function instance between renderings:

1. A functional component wrapped inside React.memo() accepts a function object prop
2. When the function object is a dependency to other hooks, e.g. useEffect(..., [callback])
3. When the function has some internal state, e.g. when the [function is debounced or throttled](https://dmitripavlutin.com/react-throttle-debounce/#2-debouncing-a-callback-the-first-attempt).

That’s when useCallback(callbackFun, deps) is helpful: given the same dependency values deps, the hook returns the same function instance between renderings (aka memoization):

import { useCallback } from 'react';

function MyComponent() {

// handleClick is the same function object

const handleClick = useCallback(() => {

console.log('Clicked!');

}, []);

// ...

}

handleClick variable has always the same callback function object between renderings of MyComponent.

Graphical user interface, diagram

Description automatically generated

Graphical user interface

Description automatically generated

<https://dmitripavlutin.com/how-react-updates-state/>

***useMemo()* hook**

useMemo() is a built-in React hook that accepts 2 arguments — a function compute that computes a result and the depedencies array:

const memoizedResult = useMemo(compute, dependencies);

During initial rendering, useMemo(compute, dependencies) invokes compute, memoizes the calculation result, and returns it to the component.

If during next renderings the dependencies don’t change, then useMemo() *doesn’t invoke* compute but returns the memoized value.

But if dependencies change during re-rendering, then useMemo() *invokes* compute, memoizes the new value, and returns it.