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>