React Tips and Tricks.

Bogdan Adrian

1. Always ask for help
2. JavaScript to know for React
3. How and where to learn React from.
4. How to think of components.
5. Take care of your naming.
6. Use UUID (or similar), on the data source and not at the component level!
7. Use shorthand for boolean props.
8. Write a fragment when a div is not needed.
9. Before going deep into React advanced concepts, make sure you know CSS and HTML well enough.
10. Don’t pass unrecognised props to JSX elements.
11._Don’t enclose string props in curly braces.
12. Write self closing components when there are no children to pass.
13. Create a folder for each component and export an index.tsx from it.
14. Restrict styling with themes.
15. Understand the differences between React.memo and useMemo.
16. Make wise use of the key prop.
17. Learn to always keep your data objects immutated.
18. Don’t abuse the useEffect hook.
19. Avoid setting a new state whenever possible.
20. Learn to recognize the reasons a component re-renders.
21. Decompose your tree strategically.
22. Avoid overusing 3rd party libraries for trivial things.
23. Imports/Exports pattern.
24. Import order matters.
25. React Components and specific hooks for each.

26. Always check for booleans in JSX.
27. Do not use inline CSS.
28. Don’t use inline functions if possible.
29. Avoid stale closures.
30. Use children props to stop re-rendering the children components.
31. Fetching in useEffect does not fit the React mental model.
32. Understand identities when working with lists.
33. Lift the custom hook up.
34. Semantic HTML in React.
35. Don't build blind branches in JSX.
36. Put functions outside the component.
37. Do not put the theme (light/ dark) in any global state (Context, Redux etc).
38. Don't use Context as global state manager.
39. Make use of the useReducer hook more often.
40. Don’t put JSX in custom hooks.
41. Don’t pass entire objects as props if possible.
42. Don’t import SVGs as JSX or directly in React.
43. Always, no matter what, do only one thing in useEffect.
44. Why useEffect needs functions too in its dependency array.
45. Don’t plug in data in weird ways.
46. Don’t try to control renders on React.
47. Don’t use ternary in JSX if possible.
48. What for and when to use the useCallback hook.
49. Don’t use setInterval in useEffect.
50. Whenever you have JSX repeating itself, extract the logic to a config object and loop through it.
51. Destructure Props on Component call.
52. Avoid Nested Ternary Operators.
53. Use composition instead of Context.
54. Do not send the all application bundle to the client.
55. Error handling is as important as all the other React code.

56. Remove all the Listeners When Unmounting Components.
57. Toggle CSS instead of forcing a component to mount and unmount.
58. How and why to Use Dependency Injection in React.
59. Use virtualization for large lists.
60. Learn to work with complex data from parent to child.
61. Understand why you lose all the state when you refresh the browser.
62. Use the useRef hook to access properties on DOM elements.
63. Don’t use “create-react-app” for your production app.
64. Refactor old jQuery apps to React.
65. Use the linter to avoid syntax bugs.
66. Pay careful attention to the function's identity.
67. Why and when to use useMemo.
68. Clean up the useEffect after async operations - use an helper variable if needed.
69. What is a HOC and why you might need it (sometimes).
70. What is Render Props and why you might need it.
71._What is the flow data Flux and why does it matter in React.
72. Learn the difference between reactive and not reactive state to write better React code.
73. Use enums to conditionally return form functions or even components.
74. What global state manager to use.
75. Learn what shallow compared values are and why they matter a lot in React.

1. Always ask for help.

This is not a real React tip but a general one for everyone getting into a new job. However, I feel like I need to tell you this!

If you get a job and you are being assigned a task that you are unsure how to tackle, ask for help!

Don’t be afraid someone will say you don’t know things and maybe you got the job without deserving it.

It’s always better to ask for help up front rather than trying things on your own and getting them wrong.

Usually, the teammates are tolerant of the newcomers and they want to help. Everyone was once a newcomer.

Make sure you can give some context when asking for help. Study the problem, and try to come up with some solution on your own first. Tell all of that to those you ask for help.

Sometimes bad teammates exist of course, but that should not be your problem. Usually, they are being known and observed in the teams.

Nobody will judge you because you are asking for help, especially in the beginning.

You’ll be judged if you write bad code, that is for sure!

Also, before starting to write code, try to understand very well the task you were assigned.

Look at it from a 20k feet view and compare it to the business needs.

Only when you understand the big picture start to write code.

jhessinymattos@gmail.com 02 Oct 2022

2. JavaScript to know for React.

I don’t think there is such a thing like: "***JavaScript to know for React developers***".

There is: **you know JavaScript or you don't**!

Of course, "knowing JavaScript" is very abstract.
The fact is, the more you know, the better React dev you are.

Nobody really knows JavaScript.
Maybe Kyle Simpson and few like him.
Therefore, you don't need to be Kyle Simpson to be a good React dev.

If you don't know yet what closures are, how "this" works or what is the prototypal inheritance, you still can be a good React dev.

Those are knowledge you'll gain in time.

But everything about practical JavaScript is a must.

jhessinymattos@gmail.com 02 Oct 2022

For example, it is not enough you know **Array.map()**; you need to know all the Array methods.

Conditionals of any kind. In React complex conditionals are day by day bread.

Destructuring, spread, rest, ternary, pass by value and pass by reference, functions and all their gotchs etc.

So, don't search for “JavaScript for React developers”.

You’ll find articles like this one:
https://kentcdodds.com/blog/javascript-to-know-for-react

Nothing wrong with the article, it is well written like all the articles form Kent.

Only notice that. If you need to learn JavaScript because you are using React already, you did something wrong with your learning path.

It will only be gatekeeping.

jhessinymattos@gmail.com 02 Oct 2022

3. How and where to learn React from.

If you are a beginner and you feel a bit confused about all this rumour you hear about React, I have good news for you: I felt the same for quite some time! It took me years to become confident in my React skills!

And because I have now some years since I was learning and using React, I can tell you the right way to do it!

Why is React easy to learn but hard to master?

Rect has a couple of APIs which one can learn in a week or so.

Once you know about components, JSX, hooks, state and props you can already write React code.

Still, it takes years to become a master in it, admitting you ever become.

But why?

Because there is so much going on under the hood and those few apis may be used in so many different ways that you can learn all of this only by experience and not by learning.

Want to learn React?

First thing first, make sure you read the previous tip and your JavaScript knowledge is at least average.

Don't start with React without a decent grasp of JavaScript; many concepts and patterns will be un-understandable to you!

Next start by learning React basics:

- What is JSX,how it works (needs transpilation), and the "whys" of JSX;
- How components work, what they are and how can they be composed together by plugging one into another like they were pieces of Lego;
- What is state, how many types of state do we have in React;
- What are props and the pitfalls of them like prop drilling and identity of props.
- Conditional rendering;
- JavaScript in JSX;
- Handling events
- Creating forms and understanding their local state pattern.

After, you can keep learning advanced concepts:

- Hooks. The why and how of hooks.
- Custom hooks, how to create them, why you need them;
- What is global state and how can it be managed
- Context, pros and cons;
- Higher Order Components and why you may need them;
- Refs, why, when and how of refs;
- useEffect and useState overuse pitfalls;
- Debugging too many re-renders;
- What a SPA means and how it works;
- SSR vs CSR;
- Learn why React is a library and about the most important packages like React Query or React Router etc;

Many other topics that I do not remember now but you'll encounter along your preparation for becoming a React dev journey. Problems included.

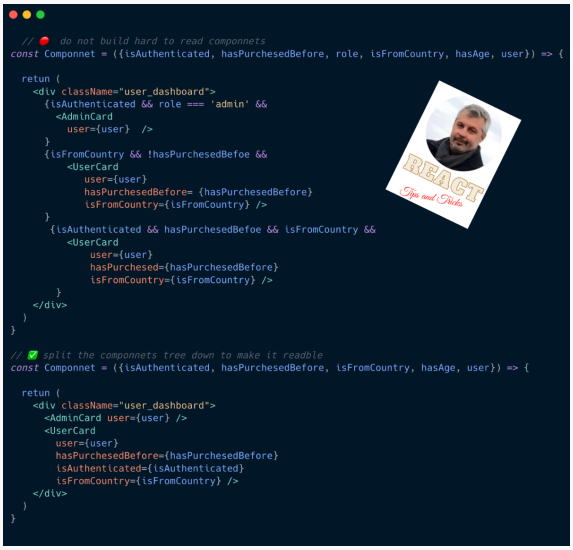
Start by taking a course, there are some of them free.

You can go on Scrimba and take their React course free for example.
You can find some on Youtube, I can say React crash course by Traversy Media for example.

Or on Udemy, there are many. I can recommend to you the React course by Maximilian Scwarzmuller or Andrei Neagoie.
For advanced React you can check Frontend Master for example where you can find very good courses.

But keep in mind, no matter how many courses you'll take and how many times; your React skills will grow only in time and by encountering problems in React apps you develop! There are no shortcuts to this!

4. How to think of components?



Components are the core building blocks of React.

But they can get quite verbose if we do not pay enough attention when building them.

By verbose I mean not necessarily too big but with too many conditions to display.

Those conditions can run on props or on new states. Or derived states, etc.

And if there are too many of them, we'll end up having too many "ifs" to display JSX conditionally in our component.

A better approach would be to split the component in a container component then the enclosed ifs logic in different children components. (See why you may need a HOC tip).

Also, keep in mind to never declare a component inside another component.

I saw such a thing even though you may ask: but who can do that?

5. Take care of your naming.

I know, you already heard about that! But are you doing it right?

More than 50% of the comments I used to receive on my PRs were about naming!

Learn to name things in your code and the comments on your PRs will drop significantly, trust me on this!

"Correct naming eliminates the need for comments".

What if I have a function named: `**handleClick**` somewhere in a component or in a hook, and another `**handleInsertNewProduct**`?
Which one tells you more about what it does?

Keep being consistent with your naming from the upper folders down to the smallest variables.

For example, if you must name a boolean, it should tell what the boolean is up to: `**isLoading**`, `**hasName**`, `**canUseOptions**` etc.

If it is a function, a verb should be part of the name: `**handleLogin**`, `**openScreen**`, `**validateUser**`, etc.

Alternatively, functions and methods can also be used to primarily produce values - then, especially when producing booleans, you could also go with adjectives. For example **isValid()**, **isEmail()**, etc.

Avoid names like **email()**, **product()** etc. These names sound like properties. Prefer **getProduct()** etc., instead.

In React (all JavaScript applications), we have a folder structure to follow. Start by correctly naming your upper folder, then your files, then your components in those files, the hooks, and the utils functions you may have down to the variables.

If you build a feature, let's say you build a "Get User Position" feature in React Native, start by adding an upper folder for it.

If that feature expands on the backend, name it the same way the folder is used in the backend!

For example, add a folder **UserPosition** in `**./src/components**`. Then add an `**index.tsx**` component and export a `**UserPosition.tsx**` component from it.

You may want to add a hook (we will see in another Tip), for dealing with the logic for getting the user position.

Name it `**useUserPosition**`.

Clean code and best practices start all from naming!

If you get that wrong, everything will be wrong!

6. Use UUID (or similar), on the data source and not at the component level!

If you need to list a set of data with **Array.map()** but the data misses an unique identifier such as an id, transform that data at the source and add an id to it.

Do not do it in the component rendering the list.

Whenever the component renders the id will be different because the re-render will call the UUID function again and that confuses React.

One other way to do this is to let React take care of it, especially for lists.

React can assign keys to your children; no need to think of creative ways to come up with them.

Instead of this:

{someData.map(item => {item.title})}

You can do this:

{React.Children.toArray(someData.map(item => {item.title})}

7. Use shorthand for boolean props.

Often there are scenarios where you pass boolean props to a component.

A lot of developers are doing it like this:

<Component isAllowed={true} showToast={true} />

But you don't need to do it necessarily like this because the prop name itself is either truthy (if the prop is passed) or falsy (if the prop is missing).

A cleaner approach would be:

<Component isAllowed showToast />

However, this has no performance implications so if you like to see what that prop really is and want to be specific, please feel free to do so.

8. Write a fragment when a div is not needed.

A fragment is an empty tag to enclose all the JSX: **<> ... </>**

A React component can only render one single HTML tag at its root.

I am sure you already saw an error which tells you that *Adjacent JSX elements must be wrapped in an enclosing tag.*

```
const Component = () => {
  // 🔴 Will throw an error
  return (
    <h1>Welcome!</h1>
    <p>We are a hosting provider</p>
  )
}
```

You need to wrap the rendered output into a fragment, which satisfies React and doesn't render an extra HTML element.

```
const Component = () => {
  return (
   <>
     <h1>Welcome!</h1>
```

jhessinymattos@gmail.com 02 Oct 2022

```
      <p>We are a hosting provider</p>
    </>
  )
}
```

This can be solved with a div tag as well.

But using a div tag will create an unnecessary DOM node.

So whenever you have to use a wrapper tag in React but don't necessarily need an HTML tag, then simply use a fragment.

jhessinymattos@gmail.com 02 Oct 2022

9. Before going deep into React advanced concepts, make sure you know CSS and HTML well enough.

Why?

Whatever job you may find as React developer, will ask you to write components and tree of components and compose them together in something meaningful.

And if you will be able to do it from the point of view of functionality because you are already good at React (??), how those components look on the screen depends all on your CSS skills.

Keep in mind, the final client does not care about React advanced concepts: he doesn't care if you memoize your components or not, if you use too much **useEffect** or **useState**.

He only cares about how the app looks first of all and how it works, second.

jhessinymattos@gmail.com 02 Oct 2022

In whatever job you'll get, there will never be someone else to write CSS for the code that is charged on you.

And if your CSS skills are very low, believe me if I tell you, your employer will get upset.

10. Don't pass unrecognised props to JSX elements.

```jsx
const Component = ({ isTitle, children, ...restProps }) => {
  return (
    <h1
      style={{ fontSize: isTitle ? '24px' : '18px' }}
      {...restProps}
    >
      {children}
    </h1>
  )
}
```

If you have a JSX element, kind of **h1**, **div**, **p** etc, and you try to pass it props that element does not know, you'll see a React error in your console.

Why would you want to pass it props that the JSX element does not know?
Because the props may come from destructuring for example.

In the example above, the **h1** element can take a "**fontSize**" style property and therefore decide on a boolean pass to it that fontSize, but it can't take a, for example: "**isUserLoggedIn**" prop coming from destructuring **restProps** for example.

An **h1** JSX element does not know and has no interest to know if the user is logged in or not, therefore the error.

11. Don't enclose string props in curly braces.

```jsx
// 🔴 Nope
<Component title={'Blog'} />

// ✅ Yes
<Component title="Blog" />
```

This is simple and the linter usually warns you about it!

If you have a prop which is a string, pass the string to it and not the string enclosed in curly braces.

12. Write self closing components when there are no children to pass.

```jsx
// 🔴 With children
<Component title={'Blog'} >{children}</Cildren>

// ✅ without children
<Component title="Blog" />
```

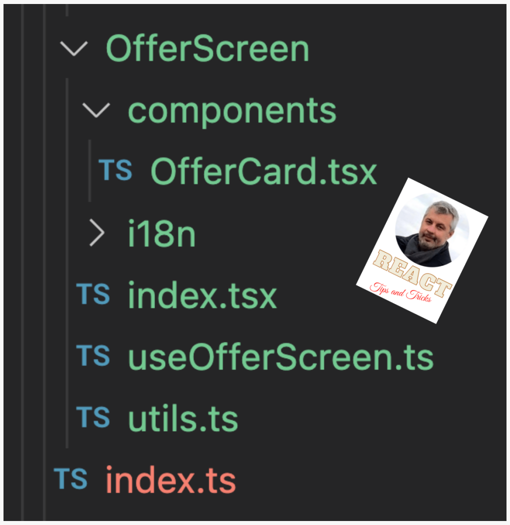
If a component has no children, do not write it with enclosing tags.

It is redundant and unnecessary.

Components can be written with enclosing tags or without them, like in example above.

There is no difference between the 2 ways apart from the need to pass children or not!

13. Create a folder for each component and export an index.tsx from it.



Think of a component like being itself a mini application on the main application.

If the component contains business logic, extract it to a custom hook.

jhessinymattos@gmail.com 02 Oct 2022

If the component needs to be split into subcomponents, create a "components" folder inside the component folder and put each subcomponent in it.
If there is a translation for it, add the folder with languages to it.

If there is a need to create some helper function, put it in the "utils" files.

In this way everything the component needs is enclosed in one folder specific to it.

Placing all files under the same directory is a good way to make the code more comprehensible to others.

It's easier to understand how different files relate to each other when they are grouped under the same directory.

Instead, placing your component's related files by their types in some "type" folders, is terribly cumbersome and not as clear to deal with as we need to jump from the component file to its hook, to its utility function or its subcomponents along all the app structure.

jhessinymattos@gmail.com 02 Oct 2022

14. Restrict styling with themes.

Why?

Think of colours for example.

You need your app to be consistent about the background colour or text colour in every page or component of it.

Meanwhile you can repeat the colour for styling on each element, that will become soon cumbersome to be changed as your app grows.

You'll need to go everywhere you declare the colours and change them.

If you have colours defined for background, text, titles etc in a theme, then you can use that theme everywhere and when a change needs to be done you change the colour in only one place: in the theme provider!

This is what the most of UI libraries do with theming out there, not only for colours but for whatever style.

jhessinymattos@gmail.com 02 Oct 2022

To provide better predictability to your component's behaviour (including its visual appearance) limit the degree of freedom your component consumers have in overriding your component styling properties.

For example declare default media queries for different screen sizes in the theme and use theme everywhere.

Your app will look UI consistent this way.

15. Understand the differences between React.memo and useMemo.

One big misunderstanding I saw beginners often have, is the difference between the **useMemo** hook and **React.memo**

Both APIs are used to memoize code.

But one, the hook, is used to memoize functions returned values. The other, **React.memo**, is used to memoize components.

However, the way both of them work is quite the same. Only when and what you use each of them is different.

With **useMemo** you wrap a function inside a component. Usually a function which performs a hard computation and which does not depend on props or states that change often.

useMemo will memoize the return of the function without performing the calculation again on re-render.

But if that calculation depends on a prop for example, which changes on each re-render, using **useMemo** is a pain rather than a help. So pay careful attention.

React.memo wraps a component and if the props it receives does not change, the component is not called and it does not re-render.

But if a prop changes every render using **React.memo** is damaged rather.

Both of them, **useMemo** and **Reatc.memo** can take a comparison function as argument where you can compare the dependencies for **useMemo**, the props for **React.memo**

Advise: don't use this 2 only if you know well what they do and only if you have optimization problems!

16. Make wise use of the key prop.

```
const Component = ({userAdmin, userRegular, isAdmin}) =>{
  return (
      <>
        {isAdmin ?
           <UserDashboard
              user={userAdmin}
               key={user.id}/>
                 :
            <UserDashboard
               user={userRegular}
               key={user.id}/>
        }
      </>
    )
}
```

You may know the "**key**" prop form looping through an array with the **map()** method and displaying different elements of the array to the UI.

In that case if you do not add a prop "**key**" to each returned element React will throw you an error telling that each element needs a key prop.

Not only components can have a "**key**" prop.

React assigns a "**key**" prop to every single element in the application.

React needs the key prop on every single JSX element basically because the Fiber tree needs to know to which node it relates.

So, the **div** element, the **h1** element, the **p** element, each component and so on will have key props assigned by React. Of course it is hidden from us but it is there.

But, we can make use of that "**key**" prop to programmatically mount or unmount whatever component or JSX element.

Not often you'll need to use the "**key**" prop to programmatically mount/unmount components - that is a React job mainly - but there are some use cases.

For example, if you have a condition and you render the same component, but the component needs to take a different prop based on the condition, the key prop will do the trick like in the snippet above.

17. Learn to always keep your data objects immutated.

No matter whether your data object is passed down as props or you set the state with it or you use it in some helper function or event handler.

Mutatting original object is always poison for JavaScript.

It can introduce a lot of unknown bugs and wreck all React functionality without you being able to understand what happens.

But what keeping an object immutated means?

It means, whenever you need some modification on an object, let's say you need a users array modified because in the **Header** component you are using only the name and the other data contained in the user object, which means returning always a new users array to be used and leaving the original array intacted.

```
const users = [
{name: user.name },
{name user.name}
...
]

const newUsers =
[...users.map(...transform data here) ]
```

Instead, this is bad:

```
users[0].user.name = 'something else'
```

As you can see the **newUser** is a totally different array than the **users** one in the first example. Means the users array stays as it is.

Meanwhile, in the bad example, we are modifying the original **users** array by assigning a new value to the user at **index 0.**

You need to keep every object, array etc, immutated and always create new ones for your needs.

Please notice: keeping reactive data immutated in React is a pattern that will heavily impact your render cycle.

In fact, a few times you may need children tree re-rendering because of the object's identities.

Mostly you'll want children trees re-rendering because values are changing.

You need to find the right balance between immutability and re-rendering.

But never compromise on immutability!

18. Don't abuse the useEffect hook.

useEffect exists to allow you to step into React render cycle and integrate third parties outputs.

It exists only for that purpose basically.

However, we developers tend to use it for much more.

This leads our applications to become un-performant and laggy.

useEffect always runs after the component renders.

So, whatever you do in **useEffect** that is not an end in itself, be aware of that.

Usually, in many of the cases, you'll need to set a new state from running the **useEffect**.

You'll need to ask yourself before using the hook: "*do I really need the useEffect here?*".

Dan Abramov has written some very useful and easy to understand React docs where he explains well why "***You may not need the useEffect***".

Here is a small re-assume:

Do you need to calculate the state from a state or props you already have?

Do that in the component and not in **useEffect**.

One of the big mistake React developers do:

Do that calculation outside the **useEffect**.

You won't need to set a new state with the **newData**! It can be easily derived from data in a case like this!

Do you update state in **useEffect** when a prop changes? Wrong !

Why?

Props are some sort of data that belongs to React already.

A prop comes from a parent not from an external implementation.

Using **useEffect** to update state is wrong because the **props are not a side effect**.

Instead you can derive the needed data from that prop and use it as it is in your JSX.

If you need to compute some value from it and add **setState** to it as well, you may want to lift the **useState** up where the prop comes from and update it there. Then use a callback function to call **setState** in parent.

Next pass the updated prop value to the children you need to consume it.

Advice: treat always the **useEffect** hook like the evil of all the bugs in your React application!

19. Avoid setting a new state whenever possible.

Using useState when re-render is not required is a bad practice.

In functional components, you can use **useState** hook for local state handling.

Although it is pretty straightforward, there can be unexpected issues if it is not used correctly.

Setting an uncessear state will trigger a wasted re-render.

Besides, the component re-renders and if you have a stale closures somewhere that will lead to bugs.

If the component has children, maybe it is close to the tree's top, all the children will re-render.

If the data needed can be derived from the actual state (or props), better you do so.

You can use the component render itself to derive data.

What does it mean: "*using the render itself to derive state?*".

We know JavaScript runs top to bottom, line by line for all the synchronous code.

When a component renders, if we have a state or props with some data that needs to be modified, we can write the modification logic below the state and the render itself will do it for us.

Therefore no need to run any **useEffect** again to set a new state with some derived data!

It is particularly worse if you set an unnecessary state from **useEffect**.

useEffect runs after the re-render and setting the state again will kick off a new re-render cycle, after everything was already settled.

Many times the useState can and should be replaced with useRef.

For example if the user performs an action like typing into input and you need to show an error, if the boolean value for toggle the error visibility is based on some condition let's say, it is useless to use the **useState** to set the error to true.

You can use the **useRef** and set **ref.current** to **true**.

So the next time the user types and the component renders or it uses that value to make an api call and the response determines the component to re-render, the **ref.current** will be true and the message will show up.

You rely on the render itself to read the error boolean value from **ref.current**!

You spare one render and you preserve the state.

Keep in mind that at every render side effects may be introduced.

In refs you can store whatever value you want, besides referencing DOM elements.

Whatever value you store in refs will stay immutated over the re-renders.

Also, the only way to write to a ref is by assigning its **ref.current** value to a new one.

Assigning a new value to **ref.current** does not trigger a re-render.

Which makes refs ideal to store data that can change but no re-render needs to be triggered.

On the other hand, writing to a **ref.current** - which is not triggering a re-render - does not allow the new written value to be displayed in JSX.

Keep in mind, at every re-render, the value displayed in JSX will read from what is at that moment **ref.current**.

So, **if you know that, there will be a re-render anyway**, don't use the **useState**

to store values that need to be shown in JSX.

Just make sure you do not write to ref during the render if you need to read from it the same render phase.

The ref current value may not be set by the time you need to read it and an old value might be read from the ref.

In this case you can use the **useLayoutEffect** to write to the ref.current.

useLayoutEffect will make sure you do not write to the ref during a render but immediately after, so the new **ref.current** value will be available for the next render.

In this case, if you need the value to be reactive to every render, better use the **useState** or **useReducer**.

As a rule of thumb, **if you know the component will re-render for whatever reason, do not set a state**. Use that re-render to derive the state needed!

20. Learn to recognize the reasons a component re-renders.

Many times we need to debug some React component that renders too often.

A part that, knowing exactly why a component re-renders is a basic knowledge a React developer should have anyway.

Here are the reasons a React component re-renders:

- **Parent renders,**
- **New props is received,**
- **State is set,**
- **New Context value arrives.**

A component re-renders many times because the parent renders. That is the main reason usually.

In order to debug that, you need to follow the data flow from the top tree to your bugged component and see which component in the tree renders too much. Once you understand the causes you need to solve them.

Sometimes you can't stop a parent from re-rendering a lot for different reasons. Maybe it shows in the UI some often changing state for example.

In that case what you can do is to plug the children that you need to stop rendering so often as a sibling of that parent.

Another option would be to use the **bailout React mechanism** (works if no new props are received), and pass the child component as a children prop to the parent.

Last option (which I advise against), is to use **React.memo**. Also works only if no new props are received.

As said already, another reason a component re-renders too much, is because it receives new props too often.

This may be the cause if you already excluded the parent render triggering the child component re-render (with bailout or **React.memo**).

In that case, take a look at the props it receives.

Are the props some function which is new on every render of the parent?
Wrap it in **useCallback**.

Are the props objects?

Objects are usually new on every render if they are declared at the component level.

Use Immer for example to keep the object prop immutated.

Or destructure the object and pass its primitive properties as props.

Last reason you may not directly control if a component re-renders too much is because a **new Context value** is plugged in (if you use Context and that is the main reason I advise against using Context).

Setting the state unneeded is another reason your component re-renders too much.

How to debug too many renders.

There is no React developer out there who did not ask himself this question at some point in his career.

It is hard to debug too many renders when the experience is not well consolidated.

You can use **React Dev Tool** to find out which component re-renders too much and how many times.

But it is almost never that component fault.

What you need to do is to go up the tree and ask yourself a question:

How many times do I set the state in every parent of it?

Many times you are setting the state more than necessary somewhere up the tree.

And what might happen in some of the parents, or the component itself, it enters a re-rendering loop waiting for some async operation to complete.

When the answer from async comes, the loop may stop.

Think of some dependencies of **useEffect** relying on that response for example.

But meanwhile, many unwanted re-renders already happened.

This might be one of the causes.

However, remember to check every parent and its **setState** cycle.

The first and most important debugging action of too many re-renders to start with.

21. Decompose your tree strategically.

We know that we need to extract a new component whenever one component becomes too big.

But only extracting a new component because the component is too big does not mean much.

A component can be as big as you want in terms of JSX and there is nothing wrong with that.

Kent C Dodds explain it well: **https://kentcdodds.com/blog/when-to-break-up-a-component-into-multiple-components**

What you need instead is an established pattern that can tell you with certainty when to break a component in multiples.

Here is one rule I follow:

Whenever I need to set a state in the component and there is more JSX than the JSX exclusively interested in the new state change, I break it down into 2 (or more) components and I pull the local state down.

In this way, the JSX that is not interested in the new state does not need to re-render.

22. Avoid overusing 3rd party libraries for trivial things.

Please avoid the usage of the libraries like lodash or ramda (even though it is more about functional programming than a syntactic sugar) for things that can be done with Vanilla JavaScript.

JavaScrip has gone quite far and is now capable of many things that these helper libraries used to provide.

There are, of course, some valid use cases for these depending on the task, it just shouldn't be a default thing to reach for on a daily basis.

23. Imports/Exports pattern.

Imports from general modules should be done from module without specifying path to the module:

// ❌ Don/t import all the path

```
import { useNotifications } from
'../notifications/hooks/useNotifications'
;
```

// ✅ export an index files from hooks folder so every import can be done only form “notifications” without specifying further down path.

```
import { useNotifications } from
'../notifications';

import { useNotifications } from
'../notifications/hooks';
```

To achieve that, create an **index.ts** file in each folder and export the files from it.

Also:

jhessinymattos@gmail.com 02 Oct 2022

Prefer named exports over default.

Avoid exporting the functions, classes or types that are known to never be used and are internal to the feature.

Named exports allow you to do that.

jhessinymattos@gmail.com 02 Oct 2022

24. Import order matters.

The order of import statements isn't random and follows some rules.

This comes down to clean code principles.

Meanwhile, the rules might be decided by your team, here are some guidelines.

The order of package imports is as follows:

1. Native (node) package imports (only applicable to server). Ordered alphabetically (by the package name).

2. Platform/framework-level imports, starting with the most important ones, core technology, like react for Web app, react then react-native for Native App or express for Server App.

3. Helper libraries that are specific to the platforms/frameworks, like react-intl, @react-navigation, react-router, styled-components, etc. Ordered by their relative importance, there are no strict rules, move up the

jhessinymattos@gmail.com 02 Oct 2022

ones that have more "impact" on the app. If there are packages of the same importance, then they should be just ordered alphabetically.

4. Non-platform-specific libraries (like d3) and smaller libraries that extend those libraries (like d3-path-interpolation for d3). Ordered alphabetically but grouped around some "main libs” (like all the d3-related imports would be grouped together instead of being scattered amongst other imports).

5. Helper libraries like date-fns, lodash, etc. Ordered alphabetically.

6. Company's own libraries like **'@company/package-theme'**, **'@company/package-ui-kit-native'** go last. Ordered alphabetically.

7. Relative imports are always ordered alphabetically (this also makes the ones that have more ../s in their path naturally appear at the top).

Note: There should be a blank line between package imports (e.g., **import { ... } from 'some-package'**) and relative imports (e.g., **import { ... } from '../../somewhere/in/our/code'**) and no other blank lines between imports.

Note that the imported items are also ordered alphabetically on their own:

```
// ❌ everything compact
import { useState } from 'react';
import {
Page,
Typography,
ScrollView,
useTheme,
Content,
dimensions,
Paper,
Button,
} from '@company/package-ui-kit-native';

// ✅ a blank line between native and framework based imports.
import { useState } from 'react';

import {
Button,
Content,
dimensions,
Page,
Paper,
Typography,
ScrollView,
useTheme,
} from '@company/package-ui-kit-native';
```

NOTE: The import statements order can (and should) be automated with ESLint!

Native App import order example:

```
import React, { ... } from 'react';
import { ... } from 'react-native';
import { ... } from 'react-intl';
import { ... } from
'@react-navigation/_';
import { ... } from 'styled-components';
import { ... } from 'formik';
import { ... } from 'use-deep-compare';
import { ... } from '@apollo-client';
import { ... } from 'big.js';
import _ as d3 Array from 'd3-array';
import \* as d3Scale from 'd3-scale';
import { ... } from
'd3-path-interpolate';
import { ... } from 'date-fns';

import { ... } from 'lodash';
import { ... } from 'yup';
import { ... } from
'@bitwala/package-theme';
import { ... } from
'@company/package-ui-kit-native';

import { ... } from
'../../../../../config/something';
import { ... } from
'../../../../../generated/graphql';
import { ... } from
'../../../../../notifications';
import { ... } from
'../../../someOtherFeature/components';
import { ... } from '../components';
import { ... } from '../hooks';
import { ... } from './useMyComponent';
import { ... } from './SomeSubComponent';
```

25. React components and specific hooks for each.

Why?

Never put any business logic into UI components!

Pattern: do you have an **useState**, **useEffect** or whatever hook into the UI component?

Bad!

Extract all of that to a custom hook!

Components returning JSX do not need and have no need to know about any business logic, small or big!

This is the approach to create a nice, clean, structured and consistent React Components for the Native and Web app.

There are some principles behind the folder structure for top level components, reusable feature components and feature-agnostic UI components but

the way they are implemented internally is pretty much universal.

Use Functional Components (or FC for short) exclusively in your codebase, alongside with React Hooks instead of the obsolete class components.

So, everything in this guide implies the usage of functional components.

Component-specific hooks.

For simple components, it's completely fine to just place everything into the component file itself, even if it's a simple data-fetching query hook and a bunch of intl's formatMessage calls.

Once the component gets more complicated, with more and more data to fetch, more computed data to calculate, more static strings to format, effects to run, etc, the component might become quite a bit bloated - in this case it's advised to extract such logic into the component-specific hook.

Component-specific hook is named strictly after the component (using the **use<COMPONENT_NAME>** convention) and is collocated within the component folder.

So, if there was some **MyAwesomeComponent.tsx** (or **MyAwesomeComponent/index.tsx**), the default component-specific hook file would be **AwesomeComponent/useAwesomeComponent.ts**. It is imported by the component using relative import path:

import { useMyAwesomeComponent } from './useMyAwesomeComponent';

This hook isn't necessarily for business-logic-heavy data-driven components only.

They can be very useful for complex UI-only components as well.

Please note that these hooks are intentionally not reusable, so while there are extremely rare cases where some other component borrows another component's component-specific hook, it's

really specific to the original component it was created for.

Don't move purely UI-specific things into a default component-specific hook.

These include UI-level aspects like dealing with theme/palette, device screen sizes and such.
It is possible to move those to a hook if that hook is specific to such UI-only aspects.

Component aspect-specific hooks.

Component-specific hooks might get complicated over time (e.g., for complex components) and while this might mean the itself does too much on its own (remember - this hook is just a part of component), there might also be multiple component-specific hooks each tailored to a certain aspect of component's behaviour thus becoming component aspect-specific hooks.

While the default hook, **useMyComponent**, is generic in what it does for the

component (anything except purely UI things), there might be hooks like:

useMyComponentData that would be only concerned about fetching and preparing the data component should display
useMyComponentAnimations that would only be responsible for animations for the component
useMyComponentEffects that would only run the component's effects (if there are many of those and they have quite a bit of logic).

useMyComponent<ASPECT> - any aspect can be extracted into a hook.

Note though, that since aspects can be reusable too, chances often are that a regular reusable hook independent from a component would work better.

There can be any combination of component aspect-specific hooks and having a default/generic component-specific hook is not necessary.

For example, if component is all about complex animations but doesn't really

have anything else, consider only having the **useMyComponentAnimations** hook for it and handle everything else (e.g., if it needs a couple formatted translation strings besides the animations) in the component itself.

Note of warning! Resist the temptation of putting unrelated stuff into aspect-specific hook.

If a hook is strictly about "animations", don't think that "Ah, I already have some hook for the component, so one little nice data fetching query amidst the animation logic won't hurt anyone" - the thing is that it WILL, that query will get lost and someone will pull their hair wondering "where the hell that query is made from, this component only seems to have animations rolling O_o".

Note that a default component-specific hook, **useMyComponent** can have some animation logic along with some other bits, but the aspect-specific (e.g., **useMyComponentAnimations**) hook shouldn't have anything besides animations.

26. Always check for booleans in JSX.

Always pay careful attention to what you are checking on the left side of a logical statement everywhere but especially in JSX.

Is it a string or a number and you are sure it can't be anything else than a string or a number?

Check them with the double bang always!

```
!!someString && ...
```

Or:

```
!!undefined && ...
!!null && ...
!!0 && ...
```

Don't do only

```
undefined && ...
null && ...
0 && ...
```


Is it an array and you need to check if it exists?

```
Array.isArray(myArray) && ...
```

Or is it an array and you need to check if it is not empty?

Don’t check only the length. Check if it is bigger than 0.

```
myArray.length > 0 && ...
!!myArray.length && ...
```

Is it an object?

Don't check if it exists! There is a default empty object somewhere for sure!

It is useless to check if an object exists in JSX especially!

Check for some property on it or just check if it has any property!

And put the double bangs on the property if you know that is a string, number or undefined!


```
!!obj.name && ...
Object.entries(obj).length > 0 && ...
```

Is it a function?

```
typeof someFunction === 'function' && ...
```

You rarely will need to check for **typeof function** in JSX however.

Please notice: the double bang “!!” can be replaced by the Boolean constructor!

Instead of: **!!undefined**, you can use **Boolean(undefined)**.

In my opinion this is only a matter of personal preference besides if we do not care Boolean calls the **toBoolean** method under the hood.

27. Do not use inline CSS.

```jsx
// 🔴 Avoid using inline CSS
const Comp = () => {
 return (
   <>
     <input
       style={{
         border: '1px solid red',
         width: '20px',
         height: '10px'
       }}
       onChange={(e) => setValue(e.target.value)} />
   </>)
};


  // ✅ use className
const Comp = () => {
 return (
   <>
     <input
       className="input"
       onChange={(e) => setValue(e.target.value)} />
   </>)
};
```

If you use inline CSS, whenever the component renders a new CSS object will be created.

Creating new objects in JavaScript has its cost.

Try to avoid using inline CSS.

Sometimes you may need to dynamically set some CSS based on some state value.

In that case you can use inline CSS.

However, if you need to set a lot of CSS dynamically, consider switching to styled components.

Another option would be to create different CSS classes and switch the classes dynamically.

```jsx
<input
className={isAllowed ? 'input_allowed' :
'input_not_allowed'
... />
```

28. Don’t use inline functions if possible.

```jsx
// 🔴 don't use inline functions
<input onChange={e => handleChnage(e, id) />


  // ✅ use function.bind
<input onChange={handleChange.bind(this, id)} />
```

Why is creating inline functions a bad pattern?

An inline function is an anonymous function. It will get created on every render!

Creating new functions, same as creating new objects, comes with its costs.

However, nowadays browsers are very powerful so a tiny inline function will not have any big impact on the performance.

Therefore if you like using inline functions please feel free to do so.

This is more about good practices than performance.

Using **bind** on a handler might also be “unreadable” too many.

I do not consider “bind” unreadable, but yes, that may be true.

Ok, what are the advantages instead?

“bind” creates a new function with “this” modified and bound to the button. That function “can be called later”.

The handler however, is the same with an unbound handler from an identity point of view, no matter if it is a bound function.

Moreover, if you wrap the bound function in **useCallback** you preserve its identity over the re-renders.

That will preserve the “input” from re-rendering because the handler prop is

not new and will spare a function from being created.

If you don't like to see "this" in your code you can use "null". It does not matter, the "this" keyword is not used anywhere here but it is needed by the "bind" syntax.

And because of that the handler can be an arrow function also.

Try to understand this:

An inline function means a new prop every single render!

Does not matter if **setState** has a stable identity.

```
<button onClick={() => setState(value)}>
 Click
</button>
```

The button in this case is only a JSX element. It will re-render internally every time.

But if it was a component:

```
<Button onClick={() => setState(value)}>
 Click
</Button>
```

The component will re-render always because the **onClick={() => setState(value)}** will be a new props every single time, no matter the **setState** function identity.

Again, this is only up to you and your team if you want or not to use it!

29. Avoid stale closures.

Stale closures in React happen very often.

A stale closure means some of the code runs, may be a function, component or a hook, and it sees an old value of a state.

Stale closures may happen in many situations but the most frequent ones are when data that changes are missing from a dependency array in **useEffect**, **useCallback** or **useMemo**.

So, please add whatever dependency Eslint tells you to add in the dependency array!

In React everything works thanks to closures. Every hook exists and can be used thanks to closures!

Keep in mind that when a component re-renders everything in it is new.

Hooks, functions, JSX, etc. Only the state and refs are preserved.

When a handler or a hook runs after a re-render, some values may be older or newer.

The old or new values of what a **useState** contains, for example, will be there over the re-renders because the **useState** hook uses a closure internally and saves the state outside the component.

And so do **useEffect**, **useCallback** or **useMemo**.

When a component re-renders these hooks may run or not run again, it depends on what the dependency array tells them to do!

If a hook does not re-run, it will keep referencing the old values!

It does not matter if the hooks are new every re-render (as identity), they will keep referencing an old value because React uses closures to define these 3 hooks!

React “feeds” these 3 hooks with the external values, automatically, on the first render. Then React updates those values only if it is told to do so!

They are the “outer” function of closure!

```
const outerFunction = (argOuter) => {
  return innerFunction = (argInner) => {
    return argInner + argOuter
  };
};
```

jhessinymattos@gmail.com 02 Oct 2022

```
const inner = outerFunction(1);
inner(1) // 2
inner(1) // 2
inner(1) // 2
```

No matter how many times you call **inner(1)**, the output will always be 2!

No matter how many times you (React in this case), call the hooks; they won’t see the outside world of the current values because they have “closed over” a previous render (**outerFunction(1)**).

They will see the outside world from the render where they were born!

Changing the outside values does not work for them! (and that is what a closure is).

These 3 hooks have (all of them), a dependency array.

The dependency array is there to tell the hook when it needs “to see” the outside world, the one in the component it lives

jhessinymattos@gmail.com 02 Oct 2022

in, whenever the values in its dependency array change.

Therefore if you do not pass the outside values to one of that dependency array, a stale closure will happen. The hook will be holding old and not updated values!

That will keep going on until an unmount and the mount will happen when everything is reset and a “first render” will take place again.

I know closures are one of the hardest patterns to understand in JavaScript. And it impacts a lot the ability to write React code. Good React code.

Therefore here is a simple rule to follow until you can be in control of your closures:

Whatever external value you need to use in one of these hooks, pass it into the dependency array!

Another case when a stale closure happens is when you set a state in a handler function, or in **useEffect**, then check in

jhessinymattos@gmail.com 02 Oct 2022

the same handler if the state is the new value you set.

The component needs to re-render for the new state to be seen as new in the handler (or useEffect), and that will happen after the handler finishes its execution for that call.

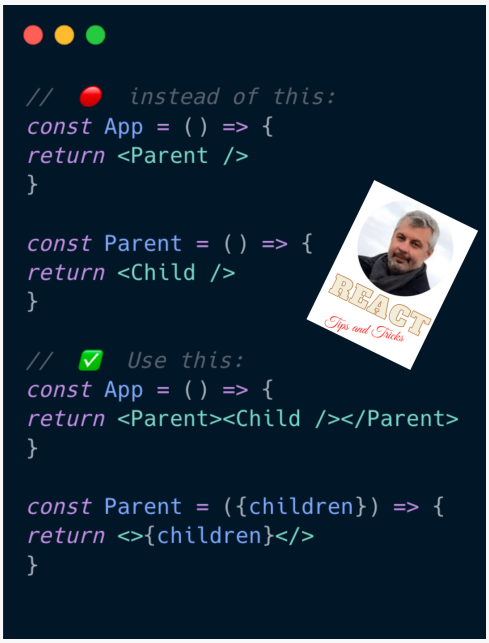
Another case of a stale closure might happen between components.

Components are JavaScript functions. Special functions because a component needs to return JSX in order to be a React component but what a JavaScript function does and how it behaves applies fully to the components.

When a component renders, because a new state is set, let's say, it renders again. That state may be consumed by other components or custom hooks.

If the consumer of the new state does not run (for example because you stopped it from running programmatically), the new state won’t be read.

30. Use children props to stop re-rendering the children components.



Passing children components with children props is an underrated powerful React pattern.

In the snippet above, because Child component is passed as the children prop, the **createElement** is not called in this case!

This is due to the "bailout" mechanism that React applies here.

Do you know why an empty component (like Child in this case), will re-render if passed regularly to the App component?

Because **React.createElement(Child, null)** will create an element like this: **{ type: Child, props: {} }**.

There is a props empty object and that forces the children component to re-render together with its parent every time.

But if Child is passed with the children props that won't happen.

I know this is a pretty harsh pattern to wrap your head around, but once you do that you will have much more control on your React render cycle.

Instead of using **React.memo** to stop a child component to re-render if it does not receive new props, use React bailout mechanism to stop children components from re-rendering by passing children components as children props to parents.

This helps a lot applying the Composition pattern also.

31. Fetching in useEffect does not fit the React mental model.

It may create waterfalls.

What a waterfall is?

For example you fetch some data in a component and render a children component only when that data becomes available.

Then in the children you need to fetch some other data again. The fetching in children will have to wait for the data in Parent to be available. It is called a waterfall.

Provocs unwanted and uncontrolled rerenders.

You'll never know when the data comes back.

All you can do is wait for it and start a new render cycle when the data becomes available.

But you have no control of that render cycle anymore.

The response decides for you.

May trigger unperformant renders.

What if the component renders exactly when the response comes back and a state is set?

Do not fetch in **useEffect** if not absolutely necessary.

Use a third-party abstraction. The best one (in my opinion), is React Query. But also RTK, SWR etc.

Keep in mind **useEffect** "interrupts" the usual React render cycle and breaks it down into a new one.

That is the main reason you may want to be very careful using the hook!

32. Understand identities when working with lists.

Let's say you are a policeman at the airport. Your job is to let people in or not based on their passports.

When an aeroplane lands, what do you do?

Do you let people in because the aeroplane comes from the US so you know everyone is American and has the right to enter the country, or do you check everyone's passport to see if there is an American passport in the first place?

Exactly the same happens with React. If you have a list, you may have only one component for each element of that list, but the elements will be rendered multiple times, with many identities for every element in the array!

In the example above, we have an Accordion with more elements.
What we do is to grab the identity of each element and see if it is set to "active" or not.

In that regard we are setting a state with an object where the key is the index of each element and the value is a boolean.

The boolean may be toggled to switch it between "true" and "false" in a handleToggle function.

Now we will have an object which contains information about the active state of each element. **Issuing the passport!**

In order to grab that state, we use another function **isActive()** which grabs the "active" state of each element and passes it down to the **AccordionItem.tsx** component. **Controlling the passport!**

It has to be a function in order to grab that state. If we pass the object value directly as a prop to AccordionItem, this will not be passed on Accordion component render because there is nothing in charge to grab dynamically that new value from clicked object state and an old value will be passed as prop because of the closure!

This pattern was something I struggled with in the beginning, so I advise you to learn it well!

33. Lift the custom hook up.

This is paraphrasing the "lift the state up!" statement.

If you have 2 sibling components and one of them needs a custom hook to abstract the business logic away but the other component needs to consume the return value of that custom hook statement, it would not work unless you lift the hook up to the nearest parent that both components are plugged in!

Then you must pass the returned value or the handler function down to the sibling components as props.

Just plugging in the hook in the sibling component won't work and the last value returned by the custom hook won't be available to both siblings. The hook contains an isolated state for every instance of it (the call site where it is called)!
(read "what are HOCs and why you may need them tip").

Nothing can determine a component to re-render at least it is forced to do so by receiving new props, a set state inside it or the render of parent.

In this specific case we rely on the render of the parent to have both sibling components displaying the latest value returned by the custom hook!

34. Semantic HTML in React.

At least you do not build a blog or e-commerce with React, you don't need semantic HTML imo!

But if you care about accessibility and as a good practice to follow you should use semantic HTML.

React is not very good at SEO. A React app renders on client so the Google bots have hard times to find and index React pages.

With React you can build dynamic data applications. For example dashboards.

Whatever app that stands behind authentication!

Even e-commerce apps too, but in that case make sure you build an entry point to the e-commerce with a SSR tech (NextJs for example).

The rule is this: is the app you build something for the large consumers that needs to be found in Google?

Do not use React. If still, you want to use React then write semantic Html.

For whatever renders on the server, like NextJs, Gatsby, Remix etc write semantic Html.

If you chose to go with a SSR framework chances are you want to be found in Google also!

For an analytics dashboard (for ex)? In my opinion semantic HTML is redundant.

35. Don't build blind branches in JSX.

```tsx
// 🔴 Avoid: all the JSX written here will get compiled
// even though it is on the false branch
const Component = ({ isLoggedIn }: { isLoggedIn: boolean }) => {

  return (
  <>
    {isLoggedIn && <div /*a lot of JSX */.../>}
    {!isLoggedIn && <div /*a lot of JSX */.../>}
  </>)
}

  // ✅ Good: one of the 2 components does not get mounted.
const Component = ({ isLoggedIn }: { isLoggedIn: boolean }) => {

  return (
  <>
    {isLoggedIn && <LoggedInComponent/>}
    {!isLoggedIn && <NotLoggedInComponent/>}
  </>)
}
```

If you put all the JSX in one component, all the JSX in all the branches will get compiled even though it is in the "no branch".

Instead having different components for different conditions will mount only the component for the true one.

36. Put functions outside the component.

```tsx
// 🔴 Avoid creating functions inside the component
const Com = ({ users }: { users: User[] }) => {
  const returnJsx = (user: User) => {
    return <div ... /* some JSX */ />
  }

  return (
  <>
    {users.map(user => returnJsx(user))}
  </>
  )
}

  // ✅ Good: put the function outside the component
 const returnJsx = (user: User) => {
    return <div ... /* some JSX */ />
  }

 const Com = ({ users }: { users: User[] }) => {
  return (
  <>
    {users.map(user => returnJsx(user))}
  </>
  )
};
```

In the case when you create functions inside the component, the function will be recreated new on every render of the component.

Of course, you can use the **useCallback** on it to preserve its identity but using **useCallback** has its own costs anyway.

A much better approach is to put the function outside the component.

You can do that however when the function does not close over the state or props, for example it takes the “users” from its scope rather than being passed to it.

Moreover, if the function returns JSX like in the example, you better make a new component from it than a function inside the component.

37. Do not put the theme (light/dark) in any global state (Context, Redux etc).

```tsx
body[data-theme='light'] {
--color-text-primary: hsl(0, 0%, 2%);
--color-bg-primary: rgb(255, 255, 255);
}

body[data-theme='dark'] {
--color-text-primary: hsl(0, 0%, 100%);
--color-bg-primary: hsl(0, 0%, 1%);
}

document.body.dataset.theme = 'dark';
//or
document.body.dataset.theme = 'light';
```

Use CSS variables instead.

Puting the theme in Context or whatever state manager you want, means only more re-renders when the theme changes. Besides, you need to subscribe all the components to that changing state.

With CSS variables you avoid those re-renders all together and the UI is more consistent!

I have a blog written on this where I explain how to do it, so please go and read it:
https://www.bogdan.digital/next/how-to-build-a-toggle-dark-light-mode-with-dataset-html-and-css-variables-in-a-nextjs-app

38. Don't use Context as global state manager.

Here is an extract from a popular blog post by Mark Erikson:
https://blog.isquaredsoftware.com/2021/01/context-redux-differences

Is Context and a global state manager - like Redux for example - the same thing?

No. They are different tools that do different things, and you use them for different purposes.

Is Context a "state management" tool?

No. Context is a form of Dependency Injection. It is a *transport* mechanism - it doesn't "manage" anything. Any "state management" is done by you and your own code, typically via useState/useReducer.

Are Context and useReducer a replacement for a global state manager?

No. They have some similarities and overlap, but there are major differences in their capabilities.

When should I use Context?

Any time you have some value that you want to make accessible to a portion of your React component tree, without passing that value down as props through each level of components.

When should I use Context and useReducer?

When you have moderately complex React component state management needs within a specific section of your application.

Also, do not use Context for passing user actions between components.

Neither local, nor global.

Context is not the right tool for doing that.

Use composition or pass props. If it gets big, use Recoil, Zustand, RTK etc.

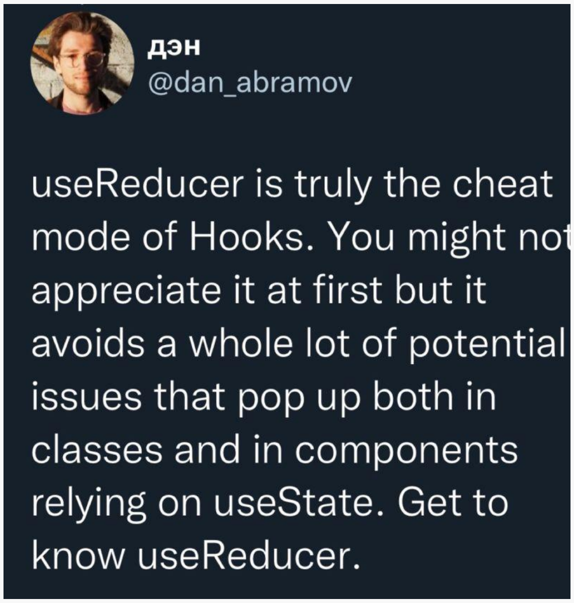
Use Context only for dependency injection.

For example, inject translation to the whole app with Context. Or the theme. (Consider CSS variables for changing themes, a better option).

When the Context value changes in the Provider, pull the Context in **App.tsx**.

App.tsx will render and all the enclosed components will render picking up the updated language setting for the entire app.

39. Make use of the useReducer hook more often.



React docs:

“*useReducer is usually preferable to useState when you have complex state logic that involves multiple sub-values or when the next state depends on the previous one. useReducer also lets you optimise performance for components that trigger deep updates because you can pass dispatch down instead of callbacks.*

We recommend passing dispatch down in context rather than individual callbacks in props.”

Passing “dispatch” down as a prop has the advantage that it has a stable identity (compared to a callback which updates the state).

A reducer is a pure function that doesn’t depend on your component. This means that you can export and test it separately in isolation.

When to use useReducer instead of useState?

If you have 3 or more useState in a component and especially if 2 or more of them need to update together, that is a clear sign you should switch to the useReducer hook!

Too many **useState** in the same component may introduce hard to catch bugs in your app. It all depends when and how those **setState** are called. Instead, **useReducer** dispatch function is predictable and has no such a pitfall.

The dispatch function - which in **useReducer** hook is used to update that state - is never new inside the component. (Neither is **setState**)

In the reducer function how every slice of state is updated is nicely organised and well separated by a switch statement which acts on the types that the dispatch function sends through.

So, it is easy to work and track even a very complex shape of state, contrary to if it was updated in useState.

No more batching issues.

40. Don’t put JSX in custom hooks.

Custom hooks should be used only for business logic.

If you add JSX in a custom hook that is no longer a custom hook but a component.

And it is not used properly because it is pulled in another component at the top level.

This is about good practices and consistency rather than performance or functionality.

41. Don’t pass entire objects as props if possible.

Pass only primitives if that is possible.

An object will change its identity on every render so you’ll have unwanted re-renders if you rely on React bailout mechanism.

Extract the properties you need from the object and pass them one by one as props.

This is also valid for the dependency array of **useEffect**, **useCallback** or **useMemo**!

42. Don’t import SVGs as JSX or directly in React.

If you import a SVG like in the image above, you will include the SVGs into the bundle which will make the app slow to load.

Each SVG has hundreds of elements so when React generates the Fiber tree, you will end up with a giant object with thousands of unnecessary data which will increase the memory consumption.

The Renderer (ReactDOM) will keep track of all these unnecessary nodes in the VDOM and it will decrease the performance especially on the low-end devices.

Preventing this issue early will save you from a huge cost of changes in the future.

I will link you to an article which explains why this is not good and what to do: **https://benadam.me/thoughts/react-svg-sprites**

Here just an extract:

“When using the first approach (in the image above), an image tag referencing an image asset (png, svg, etc.), the first time the page is downloaded there is a flicker before the icons render. This happens because of a request waterfall.

First the Browser downloads the HTML document. It then makes the subsequent requests to fetch all the assets for the page (images, scripts, stylesheets, etc). The benefit of referencing an external asset is caching. The browser (or CDN) can cache the asset and reference it on subsequent requests. The technique is optimised for subsequent requests, but the initial loading experience is less than desirable. The other downside of this technique is that we can't style the svg using CSS when it is referenced in an image tag 😢.”.

Basically, if you create a component for that SVG, the component may be stopped to render using **React.memo** (here a good use of React.memo).

But if you import the SVG in one of your components, the component may need to run and the SVG will be added to Fiber every single time.

43. Always, no matter what, do only one thing in useEffect.

```
// 🔴 Avoid doing more than on thing in useEffect
// Split the example code below in 2 useEffect
useEffect(() => {
  if (user){
    setUserAuthOnDate(new Date())
  }

  if (user && user.role === 'admin'){
    dispatch(setDashbord(true))
  }

}, [user, dispatch])
```

Why using only one **useEffect** for performing more actions on a component is not a good idea?

A part of the separation of concerns principle, which is good to follow always, that **useEffect** will run when the "dispatch" identity changes here but will run on "user" also (above example).

"Dispatch" does not change its identity as it is stable, but think of any other example when a function changes its identity etc.

When it runs on "dispatch" everything inside its body runs in the example above.

It will dispatch the Redux action and will also set the state (again) too.

One of those 2 actions will not need to run for sure.

We intend to run the **useEffect** for performing only one of the 2 actions inside its callback.

Use 2 **useEffect** instead. Each doing only one thing at the time!

44. Why useEffect needs functions too in its dependency array.

```
useEffect(() => {
  dispatch(setAuthenticated(true))
}, [dispatch])
```

Why useEffect asks to add a function used inside its body to the array dependency every single time?

The function is just called inside the body of **useEffect**, many times it is not a value used inside it.

The most well known example is the Redux "dispatch" function.

But any function used in **useEffect** must be added to the dependency array, at least if you don't tell Eslint to not check on exhaustive depths.

The functions need to go into the array dependency simply because it is used in the **useEffect** callback and because of that **useEffect** needs to run whenever the "dispatch" identity changes.

useEffect can't know if "dispatch" in this case, returns the same api, in this case dispatching a Redux action. It sees the function as a new value every time it runs. The fact "dispatch" does not change it's just a case but **useEffect** can't know that.

Therefore it asks "dispatch" to be added to its dependency array!

If we have a handler for example, created by us, useEffect asks our handler to be added into its dependency array because the hander it's called inside **useEffect** callback.

In that case it's absolutely necessary to add it to the dependency array because what it returns may be different from one render to another, as long as **useEffect** knows!

In that case, the handler identity will overwrite the returned value however and determine **useEffect** to re-run.

That means, even though the return value might be the same as in the previous render, its identity will be different!

That is a classic case when using **useCallback** to wrap the handler it's needed!

45. Don't plug in data in weird ways.

React components must be pure.

Given the same props or state they should return always the same JSX.

Therefore injecting data in any other way will make a component impure and will lead to hard to catch bugs!

Just follow the React tree unique flow data pattern.

When you need to bring in state in the middle of the tree, make sure you do that

in a reactish way and not using your hacks!

React docs:

"*React's rendering process must always be pure. Components should only return their JSX, and not change any objects or variables that existed before rendering—that would make them impure!*"

In the bit of code above the **externalCount** is some data injected from outside React in the wrong way.

46. Don't try to control renders on React.

React is declarative.

That means mostly you have to make sure that you don't do hacks to control its render cycle.

Does your state, global or local, change?

Let React know!

But don't tell it that it must re-render here or there, now or then.

At least if you are not 100% sure what you are doing!

Very few programmatic re-renders might be needed, but that is used by very advanced patterns.

For example plugging in third parties outputs.

Re-rendering is not your job. Your job is to feed it with state. Nothing more!

Can you imagine a river?

Be like a river!

Feed it water, don't tell the water what to do, where to flow to!

I know this is more conceptual and because of that giving an example is not simple.

But here is something you can think about:

Using **React.memo** is a try at your side to control the render cycle.

That is why it is not something you should do except only in very particular cases.

47. Don't use ternary in JSX if possible.

In the image above, if you take down the key prop and switch the condition by clicking on the button, you'll see the input changing because you'll see a different placeholder.

In reality, the "input" tag is not changing. It stays the same. Only the placeholder prop changes.

It does not change because switching it like this with the ternary keeps it in the same position in the tree.

React does not unmount it for mounting the other one. It just "renders the same tag", (may be a component with the same name) with a different placeholder prop.

To see that is true, type in the input (without the key prop), and see how what you typed stays as you are switching the condition.

The key prop tells React each "input" tag has a different position in the tree so now React can unmount the one on the false branch and mount the one in the true branch.

Avoid using ternary altogether.

Use **&&** and keep elements (tag or component) separated from each other!

{condition && <input />}
{!condition && <input />}

48. What for and when to use the useCallback hook.

Why is it so hard for so many to understand when, where, and how to use the useCallback hook in React?

I had people asking me this question all the time!

Here's why:

In order to understand why the **useCallback** hook exists you need to have some solid understanding of a couple of core JavaScript principles:

- object types compare to primitives;
- functions are of the type object;
- get be reference vs get by value;
- immutability;
- garbage collector.

Without a good understanding of all of this, **useCallback** won't make much sense to you.

Let me try a brief explanation!

When you are using functions inside a component, those functions do their job whenever they are invoked. Next, they get garbage collected if nobody references them anymore!

Next time you invoke them again they are brown new.

And this brings us to immutability.

A function is born, lives, and dies.

It does not matter, it is called **"handleClick"** by example and you think it is the same function because it has the same name!
For the JavaScript engine, it is a new function every time it gets called!

Whenever a function is born it gets a name (a space in memory that JavaScript references).

When you pass that function to a children component as a prop, or to an **useEffect** array dependency if it has a new name the children will re-render (or **useEffect** will run).

jhessinymattos@gmail.com 02 Oct 2022

useCallback memoaizes that function. This means it does not let it die (by a closure that references it so it hangs around in memory).

Whenever you need to call that function again, you call the same old function you called before.

And because it is the same, the immutability is preserved.

This means the children that the function is passed to as a prop do not re-render.

Or **useEffect** does not run!

jhessinymattos@gmail.com 02 Oct 2022

49. Don’t use setInterval in useEffect.

setInterval and **setTimeout** are kind of special functions. They are JavaScript interfaces to call browser APIs.

When a **setInterval** or a **setTimeout** is invoked in JavaScript, it starts running and will only end its cycle when the time is up or when/if it is cleared out.

So, if you kick off a **setInterval** (or **setTimeout**) on a specific render, means when a snapshot of state is available, those 2 browser apis functions will refer to that specific snapshot of state until their end.

If by chance a new render happens and a new snapshot of state is created, **setInterval** can’t pick it up as it has closed over the previous snapshot of state.

That is a stale clouser basically.

When a new **setInterval** is invoked again on the next render, it will be totally

jhessinymattos@gmail.com 02 Oct 2022

new compared to the previous **setInterval** (which probably didn’t get to the end yet).

So, we will have a **setInterval** running on top of another **setInterval**. And maybe on top of a new one. Each of them having access only to the state snapshot they were invoked with.

How to solve this problem?

Create a custom hook and make sure to pass the callback and the delay to **useEffect** (where **setInterval** or **setTimeout** runs) dependencies arrays.

In this way the **useEffect** callback runs again on delay or **setInterval** changing, and the new **setInterval** (or **setTimeout**) can pick the delay from scratch starting fresh.
Don’t forget to clear the old one when a new **setinterval** is kicked off.
Dan Abramov explained this behaviour very well in this article, besides giving us the custom hook already functional:
https://overreacted.io/making-setinterval-declarative-with-react-hooks/

50. Whenever you have JSX repeating itself, extract the logic to a config object and loop through it.

```jsx
import { BrowserRouter, Routes, Route } from 'react-router-dom';
import {
  Comp1,
  Comp2,
  Comp3,
  Teams,
  Team1,
  Team2,
  Team3,
  ErrorComp
} from './Components';

const ROUTES = {
  Comp1: { element: Comp1, path: '/' },
  Comp2: { element: Comp2, path: 'comp2' },
  Comp3: { element: Comp3, path: 'comp3/:id' },
  Teams: { element: Teams, path: 'teams' },

  Team1: { element: Team1, path: 'teams/:id' },
  Team2: { element: Team2, path: 'teams/team1' },
  Team3: { element: Team3, path: 'teams/team3' },
  Error: { element: ErrorComp, path: '/*' }
};

export const App = () => {
  return (
    <BrowserRouter>
      <Routes>
        {Object.entries(ROUTES).map(route => {
          const [key, value] = route;
          const Component = value.element;
          return <Route key={key} path={value.path} element={<Component />} />;
        })}
      </Routes>
    </BrowserRouter>
  );
};
```

Don’t create JSX for every variant!

It’s not a good practice to repeat the JSX.

Better a config object which will contain the logic of that DOM node.

Why does an object ROUTES and not an array?

Because this is a "config object" and not a “config array”.

It has to do with data normalisation. It is a lot easier and safer to access propriety in an object and not an index in the array. What if an element is unshifted in that array? accessing the element at the given index will error out.

Maybe you need the route down in a Children component.

Object.entries is a native JavaScript API, so no worries about performance in this case!
But if you like an array better, feel free to use an array.

Moreover, this is not about putting the routes in a config object but whatever can be put in a config object and not repeating the JSX for as many elements as there might be.

Specific for routes we have a specific React Router Dom API: **useRoutes**.

51. Destructure Props on component call.

```jsx
// 🔴 Wrong
const Component = (props) => {
  const [state, setState] = useState('')

  return (<div>
      <h1>{props.title}</h1>
      <Input value={props.value} onClick={props.onChnage} />
      ...
  </div>)
}

// ✅ Good
interface ComponentProps {
  value: string;
  title: string;
  onChange: (e: ChnageEvent<HTMLElemnet>) => void;
}

const Component = ({ value = '', title = '', onChange }: ComponentProps) => {
  const [state, setState] = useState('')

  return (<div>
      <h1>{title}</h1>
      <Input value={value} onClick={onChnage} />
      ...
  </div>)
}
```

Also, add defaults to the values to make the component resilient to errors.

Most of the React components are just JavaScript functions.

A specific type of function which can take whatever arguments (called props), and which always returns JSX markup.

There is no need to repeat **props** everywhere in front of the value they contain.

Using **props.value** it's redundant and unnecessary.

If you are using TypeScript you don't need default values.

52. Avoid Nested Ternary Operators.

```
// 🔴 bad
isAllowed ? (
  <UserAllowed />
) : isAdmin ? (
  <AdminDashboard />
) : (
  <HomePgae />
)

// ✅ better extract a component for each option in the nested ternary:

const UserStateComponent = ({ allowed, user }) {
  if (allowed) {
    return <UserAllowed />
  }

  if (user.isAdmin) {
    return <AdminDashboard />
  }

  return <HomePage />
}

const App = () => {
  return (
    <UserStateComponent
      allowed={allowed}
      user={user}
    />
  )
}
```

Nested ternaries are bad because they are hard to read and prone to errors.

Ternary operators become hard to read after the first level.

53. Use composition instead of Context.

```
export default function App() {
  return (
    <ColorPicker>
      <p>Hello, world!</p>
      <ExpensiveTree />
    </ColorPicker>
  );
}

function ColorPicker({ children }) {
  let [color, setColor] = useState("red");
  return (
    <div style={{ color }}>
      <input value={color} onChange={(e) => setColor(e.target.value)} />
      {children}
    </div>
  );
}
```

What is "composition" in React first of all?

Composition is a recommended React pattern where we pass components as props instead of plugging them into each other.

We can pass children components to other components as children prop, or we can pass thema as regular props.

Basically we return a component from a parent which takes a children prop or other component (or components) or which can take props as other subchildren components.

Something like this:

```
const Component = () => {
  const [data, setData] = useState([])

  return (
    <>
      <Children1
        left={<Children2 data={data}/>}
        right={<Children3 />}>
        <SomeOtherComponent>{children}<SomeOtherComponent />
          <div ... />
         // more JSCX to be passed as children to Children1
      </Children1>
        <Children4 setData={() => setData(newData)} >
        <AnotherComponent data={data} left={<AnotherComponent2 />}/>
      </Children4 />
    </>
  )
}
```

Which are the advantages using the composition pattern?

- Avoiding props drilling. If you have a state in the App, you can pass it to the Children3 directly from the App. In practice you can pass "data" down several components without any prop drilling and without using any state manager.

- Using the native bail-out React mechanism and limiting the number of re-renders. In the case of Children2 and Children3 no React.createElement is called in the App so no new props are passed down to them, even when there is no prop at all! So, no re-render!

Think of a huge React application, kind of an enterprise one.

Using React Query for example for the server state management and composition, you do not need any Context, Recoil, Redux etc; even though it may look hard to achieve.

If your app is made from many top level trees, then other second leve trees and so one (which every app is made like basically), composing the trees themselves first then into the superior level tree, you can do everything you may want with the global local state.

The composition pattern is a bit hard to get your hands on in the beginning but

once you start to see the potential you'll never drop it again.
A bil like TDD!

React Docs:

React has a powerful composition model, and we recommend using composition instead of inheritance to reuse code between components.
At Facebook, we use React in thousands of components, and we haven't found any use cases where we would recommend creating component inheritance hierarchies.
Props and composition give you all the flexibility you need to customise a component's look and behaviour in an explicit and safe way. Remember that components may accept arbitrary props, including primitive values, React elements, or functions.

What Is React Composition?

React Composition is a development pattern based on React's original component model where we build components from other components using explicit

defined props or the implicit children prop.
In terms of refactoring, React composition is a pattern that can be used to break a complex component down to smaller components, and then composing those smaller components to structure and complete your application.

Why Use React Composition?

This technique prevents us from building too many similar components containing duplicate code and allows us to build fewer components that can be reused anywhere within our application, making them easier to understand and maintain for your team.

If you want to find out more on React Composition, Kent C Dodds has a very good article:
https://epicreact.dev/one-react-mistake-thats-slowing-you-down

54. Do not send the all application bundle to the client.

Split the code and send only what is needed.
Wat
How much JavaScript you send to the client in terms of MB, is the number one performance factor of an app.
Before any other optimization you may think of.

Do not confound the speed of your app with the bundle size sent to the client.

Once the app downloads in the client browser, the speed may be amazing indeed.

What if the user has to wait dozens of seconds for it to download? Will it wait?

Applications that are not fast to download and ready to be used in an instant, are also the main reason an user leaves the domain and Google downgrades the SEO.

That means money, a lot!

In React we have the Lazy loading technique which we can use to split the bundle size sent to the client.

We can split the application at the route level!

https://reactjs.org/docs/code-splitting.html

Prefetching also may be a good idea.

There are many other techniques and you can find out about them if you want as there are a lot of articles talking about them.

55. Error handling is as important as all the other React code.

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    return { hasError: true };
  }

  componentDidCatch(error, errorInfo) {
    //log the error to an error reporting service
    errorService.log({ error, errorInfo });
  }

  render() {
    if (this.state.hasError) {
      return <h1>Oops, something went wrong.</h1>;
    }
    return this.props.children;
  }
}
```

First thing is to make sure the errors you are receiving from the backend are consistent and follow the same pattern everywhere in your app.

For example a **GET** request can error out with Error type and so you can access an **error.data.message**, and a **POST** request may error out with some custom error or a wrapper error and you may not have access to **error.data.message** but only to an **error.message**.

This can lead you to write bad code because you may be checking a property on the Error object, which may not exist!

TypeScript helps a lot in this case.

What exactly does error handling mean in React?

You need to make sure you do all of this:

- **catch errors;**
- **handle the UI accordingly;**
- **log the errors.**

Error Boundaries specialised component.

It is a custom class component that is used as a wrapper of the entire application.

The lifecycle method **componentDidCatch()** is able to catch errors during the rendering phase of all the trees that the ErrorBoundry component wraps.

When an error arises during that phase, it bubbles up and gets caught by the ErrorBoundary component.

If you're using a logging service (which I also highly recommend), this is a great place to connect to it.

Think of Sentry for example.

The static function **getDerivedStateFromError()** in the ErrorBoundary class component is called during the render phase and is used to update the state of the ErrorBoundary Component.

Based on this state, we can conditionally render an error UI.

The big drawback of this approach is that it doesn't handle errors in asynchronous callbacks, on server-side-rendering, or

in event-handlers because they're outside the boundary.

If you don’t like to use the class component in your application, because you love hooks, there are Error Boundaries hooks also. For example:

https://github.com/JoschuaSchneider/use-error-boundary

56. Remove all the Listeners When Unmounting Components.

```
useEffect(() => {
   window.addEventListener("scroll", handleScroll);

   return () => {
      window.removeEventListener("scroll", handleScroll);
    }
}, []);
```

We are using Event Listeners a lot in React.

onClick for example is probably the most used one, but there are many others: **onScroll**, **onMouseEnter**, **onFocus** etc.

A complete list of them here:
https://reactjs.org/docs/events.html

Sometimes we need to listen for DOM events in **useEffect**, when the component mounts.

For example, listen for **onScroll** to dynamically perform some action when the user scrolls the page.

When the user moves away from the component, we must clear the listener.

Why?

Because it happens that the listener will stay alive even after the component is unmounted.

So, the next time a user comes to the same component (mount it again), a new listener will be created on top of the old one.

It’s enough that the user moves away and comes back a couple of times and we’ll have a memory leak.

For clearing out listeners we may use the cleanup function in **useEffect**.

57. Toggle CSS instead of forcing a component to mount and unmount.

When a component is in the sight of the user, we say that the “component is mounted”.

When the component is hidden from the user we say it is unmounted.

Beware, I am saying “component” and not necessarily “page”. Or “screen”

A component might return JSX which is actually seen in the browser, visually, or it can be a HOC for example or return another component simply.
In that case there is nothing to be seen on the browser, visually.

Think of the **App.tsx**. You do not see it in the browser as per example you can see a **Button.tsx**. But the **App.tsx** is mounted (it is the main container for the entire app).

If you unmount **App.tsx** a blank screen will be left to look at.

Rendering is costly, especially when the change must be committed to the real DOM.

Mounting and unmounting, especially heavy components, consume resources.
If the consumed resources are too big the app will become unperformant.

There is another thing you can do instead of unmounting a component.

Not always this is possible, depending on the use case, but for example in case of a list (rendered with the map method), if the list is small, just a few elements, we can hide the elements which need to not be seen at the given moment by simply setting their visibility to 0.

In this way the hidden elements, which are components basically, will still stay mounted in the DOM.

This is possible only if keeping all the components mounted won't cause any issue.

For example the height of the page might be altered if all the elements are still mounted, etc.

Toggling the opacity to 0 has almost zero cost for the browser (since it doesn't cause a reflow) and should be preferred over visibility & display changes whenever possible.

58. How and why to Use Dependency Injection in React.

```tsx
wxport type Locale = 'en' | 'de';

interface LanguageContextProps {
  language: Locale;
  setLanguage: (language: Locale) => void;
}

export const LanguageContext = createContext<LanguageContextProps>({
  language: 'en',
  setLanguage: () => {},
});

export const getInitialLanguage = (): 'en' | 'de' => {
  if (Localization.locale.toLowerCase().startsWith('de')) {
    return 'de';
  }
  return 'en';
};

export const LanguageProvider: FC = ({ children }) => {
  const [locale, setLocale] = useState<Locale>(getInitialLanguage());


  const setLanguage = (language: Locale) => {
    I18n.locale = language;
    setLocale(language);
  };

  return (
    <LanguageContext.Provider
      value={{
        language: locale,
        setLanguage,
      }}>
      {children}
    </LanguageContext.Provider>
  );
};
```

```tsx
// Then we can creat a hook to be used wherever needed:

import { useContext } from 'react';

import { LanguageContext } from '@providers/LanguageProvider';

const useLanguage = () => useContext(LanguageContext);

export default useLanguage;

// Now wrap the App.tsx in LanguageProvider

  <ThemeProvider>
      <ErrorBoundary FallbackComponent={ErrorFallback}>
        <PortalProvider>
          <QueryClientProvider>
            <AuthProvider>
              <LanguageProvider>
                <OnboardingTipsProvider>
                  <StatusBar style="dark" />
                  <App />
                </OnboardingTipsProvider>
              </LanguageProvider>
            </AuthProvider>
          </QueryClientProvider>
        </PortalProvider>
      </ErrorBoundary>
    </ThemeProvider>
```

What is dependency injection?

Think at your app using more than one language for example.

When you switch the language, you need the entire app to update with the newly set language.

Which means you need the entire app to re-render and to paint the DOM all over again the new language.

You can do that with **dependency injection**.

Context is the ideal tool for doing that.

**Try to not use Context for passing user actions between components.*

Context is not the right tool for passing user state around.

Dependency injection is a fundamental software design practice that helps to manage the dependency lifecycle of a module, object or in our case a React component or tree of components.

In practice, dependency injection is frequently used to inject services such as loggers and translators, themes or whatever state is needed globally, used

in the entire app or locally to some tree.

In theory we can pass whatever state to each component by props.

But like it was the case of the language, passing a new language by props would mean a prop that contains the language state possessed to all the tree's components, top to bottom.

Not very efficient as we are triggering props drilling.

We need a more efficient way to pass a global state such as language to each component containing text that needs to be translated.

Context allows us to do that in a very natural way.

Wew set the Context to the selected language, then whatever component consuming that Context will pull the current Context language value.

Do you know how the Providers pattern works in React?

It uses Context under the hood.

So, if you have Redux, Recoil or React Query etc, all of them have a Provider at the top level which encloses all the App components.

Then, all of these libraries use Context as best as they know to share state down the tree.

The Provider patterns are basically, under the hood, a HOC which injects state values to the children tree.

In this way the state travels down the tree in an alternative way and not via props.

And because that state value is available to every component in the tree, it may trigger a re-render of each in 2 ways:
- by directly consuming the value
- because a parent consumes it and renders it.

59. Use virtualization for large lists.

If you have an array of 1k elements for example fetched from the backend, do not try to get all the elements rendered on the DOM at once.

Use **pagination**, use **filtering** or use **virtualization**.

The Dom can only deal with a limited amount of data on its nodes.

A common repeating performance problem in React JS is a complex and long list of items to be displayed on the browser.

Say the app needs to render many user elements with the attached data.

When the list of users is pretty big, let’s say thousands of users, which are composed of avatars, names, emails, links to blogs, roles or ratings etc.

The slowdown may be noticed very quickly and the app becomes unresponsive, especially on slow devices.

This happens because React needs to observe each change in every element of the list, build the Fiber tree and paint the DOM.

This process consumes lots of resources.

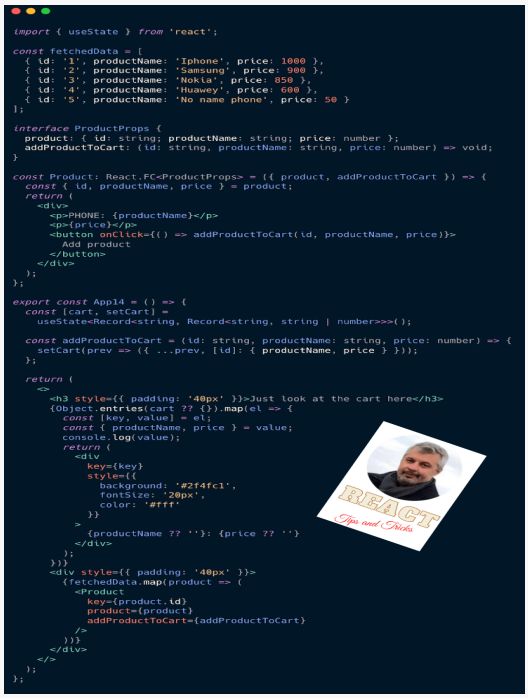
Luckily, it isn't that difficult to fix these kinds of issues.

Virtualizing the list is a well known solution.

A virtualized list is a list that renders only the content visible on the screen and blanks out all other elements until the user decides to scroll down or up in order to see them, thus saving lots of valuable resources.

The most popular library for list virtualization is **react-window**.

60. Learn to work with complex data from parent to child.



One of the most confusing patterns in React but also the most useful is passing complex data from parent to Children in order to compose new data and use it for API calls.

What is it all about?

Let's suppose you are making an API call in a parent or anyway getting some data from a server in a parent component in any way you want.

That data, very often, is an array of objects. As it is an example of products in the code snippet here up.

Your job will often be to take each object from that data array, display it in the UI (you'll use the "map" array for that most probably); grab some properties from it, think at the id, and create a new element - a new object with some user inputs added to the "product" object newly created.

Next, you may need to send that new data array back to the service.

Only the new data you'll send will be a newly created array of specialised "products" and will contain some properties from the data you fetched combined with some new user-created data properties which will be added to that new element.

As an example of this pattern would be an array of products fetched, then the user decides to buy some of them (or only one), and it adds new properties to newly created objects gathered in an array of let's say called "cart".

The newly created "cart" of products will need the id of the product then they will need a purchase **property: true** or **addedToCart: true** or only **productSeen: true** (for marketing reasons), sent back and saved on the server.

Here is what you need how to think about this pattern:

I pass the received object to a Children 'DisplayProduct' component by **products.map** method.

Then, whenever the user clicks on **‘Add to cart’**, let's say, I need to save a new object to a new state with the id of that product and the addToCart property set to true.

A new object may contain more than one product id and action of the user.

Next, when the user clicks on **‘Purchase’** for example, I get that new state, make it an array if needed and send it to the services.

The confusion I was talking about comes usually from not being able to understand the difference between the fetched products data array and the need to create new data objects to be POST to the service, which has nothing to do with the fetched data! Losing track of the transformations that need to happen with the data.

Basically a simple mapping of the cart with some newly created products data which contains some new properties but totally new.

61. Understand why you lose all the state when you refresh the browser.

Why do you lose all the state when you refresh the browser?

React is a SPA.
Single Page Application.

It means the entire app runs in only one instance.

Meanwhile components rendering and the UI are managed by React, global state is managed by you.

Global state lives somewhere outside React.

It may be **Redux store, React QueryCache client, Recoil atoms** or **Apollo GraphQl** engine.

So, when a component mounts or unmounts the local state is created or destroyed.

But the global state keeps living in that outside store.

The outside store is directly tied to a browser instance.

It is born and dies with that browser instance.

When you navigate from one page to another with whatever React Router, basically you are switching UI views with “if else” under the hood.

The browser instance keeps living when a route changes with the if else, between different UI views (pages).

But when you refresh the browser that instance dies.

The global store dies too for that instance. It clears out, resets itself to 0, no data in it!

It is born again for the next one and it has to be fed with data again.

Therefore all those "fetches” you do in **useEffect** with empty array dependencies are fired again and the responses are

feeding again that empty store with new data just arrived.

(Use **React Query** to fetch server data, don't do that in **useEffect**).

The user clicking on UI elements also produces “client global or local state”.

SPA apps, like React, are a totally different paradigm compared to how the internet used to work before they appeared.

There is the possibility to save some of that state to **Local Storage** and retrieve it immediately after the refresh, but that is not “keeping global state” as many may think.

62. Use the useRef hook to access properties on DOM elements.

If you work as frontend developer with React, one day no and the other day yes you'll need this:

```
const scrollToBottom = () => {
containerRef.current?.scrollToEnd()
};
```

This is a function you'll add where you add the useRef and you need the component (or often the page), to scroll to the bottom.

Then you can pass this function to whatever component may need to call it.

That may happen on a click, on focus, in a **useEffect** when that component mounts or something happens into it like for example some data becomes available.

Learn it and try it. You'll need it very soon.

If you want, you can pass the ref itself with **forwardRef**.

jhessinymattos@gmail.com 02 Oct 2022

But that is a bit more cumbersome to do.

There is also a:
containerRef.current?.scrollTo(x-ccord, y-coord)

Where **x-coord** and **y-coord** are screen coordinates to scroll to.

containerRef.current?.scrollTo(0) is going to take the user to the top of the page.

jhessinymattos@gmail.com 02 Oct 2022

63. Don’t use “create-react-app” for your production app.

CRA works fine for your training todos app. But try to avoid it for your production app!

Why?

Create React App is a very well made Webpack configuration to start a React app in one go. It works well and it has everything you need.

But,

It is too much. There are configurations which you’ll never need. CRA will bundle everything.

Vite on the other hand uses Esbuild and is written in Go. It is much faster, lighter and performant than CRA.

Vite improves the development server start time by dividing the modules of an application into two categories: dependencies and source code.

jhessinymattos@gmail.com 02 Oct 2022

Dependencies are primarily plain JavaScript that does not often change during development. Some significant dependencies (ex MaterialUi) are also quite expensive to process.

64. Refactor old jQuery apps to React.

Do you know that, if you have an old site made with JS, maybe JQuery, you can add React gradually to it and refactor step by step the implementation?

All React needs is "a hook into HTML, remember **id="root"** ?

Then just use CDN to add React to your site.

```
  <script
src="https://unpkg.com/react@18/umd/react
.production.min.js" crossorigin></script>

  <script
src="https://unpkg.com/react-dom@18/umd/r
eact-dom.production.min.js"
crossorigin></script>

  <script
src="https://unpkg.com/@babel/standalone/
babel.min.js"></script>

  <script src="like-button.js"
type="text/babel"></script>
```

Add capability to add JSX:

- **npm init -y**
- **npm install babel-cli@6 babel-preset-react-app@3**

And maybe more than one React app? Ex: first React app ** **, ** ** etc.

Next tie the index.tsx to that root ud and start building your React app on top of the old site.

You can keep it hidden with flags etc, until it is ready for release.

65. Use the linter to avoid syntax bugs.

Code linters find problems in your code as you write it, helping you fix them early.
ESLint is a popular, open source linter for JavaScript.

You don't want to write any React code without the Linter.

Install ESLint with the recommended configuration for React:
https://www.npmjs.com/package/eslint-config-react-app

Integrate ESLint in VSCode with the official extension:
https://marketplace.visualstudio.com/items?itemName=dbaeumer.vscode-eslint

Make sure that you've enabled all the eslint-plugin-react-hooks rules for your project. They are essential and catch the most severe bugs early. The recommended eslint-config-react-app preset already includes them.

66. Pay careful attention to the function's identity.

Do you need a function inside **useEffect** but the function changes its identity every render? For example a function passed through Context?

For example, in the snippet of code above you need to set the language with the Context.

But when you pass the **setLanguage** to **useEffect** in **App.tsx**, you'll run in an infinite loop!

Just extract that function outside the effect, wrap it in **useCallback** then pass it to **useEffect**, stabilised as identity!

setLanguage comes from the useContext hook, which I forgot to add in the **App.tsx** ;)

jhessinymattos@gmail.com 02 Oct 2022

67. Why and when to use useMemo.

Optimising with useMemo is only valuable in a few cases:

- The calculation you're putting in useMemo is noticeably slow, and its dependencies rarely change
- The value you're passing is later used as a dependency of some Hook. For example, maybe another useMemo calculation value depends on it. Or maybe you are depending on this value from useEffect.

There is no benefit to wrapping a calculation in **useMemo** in other cases.

There is no significant harm to doing that either, so some teams choose to not think about individual cases, and memoize as much as possible.

The downside of this approach is that code becomes less readable.

Also, not all memoizations are effective: a single value that's "always new" is enough to break memoization for an entire component.

jhessinymattos@gmail.com 02 Oct 2022

68. Clean up the useEffect after async operations - use an helper variable if needed.

Whatever you do in useEffect, and you should only use the **useEffect** hook to plug in third parties outputs, you may want to clean up it when the component unmounts.

Why?

For example, if you are fetching in **useEffect** and expect an answer form an api to come in, you'll probably set some state with that data arrived in response to use it in your app.

But if for whatever reason the user moves away from the component before the response arrives, by clicking a navigate to another page link for example, the async request will not be truncated right there.

It will keep working even though the component is no longer in the view (mounted). And the answer arrives, the

jhessinymattos@gmail.com 02 Oct 2022

state you want to set with the response will try to fire.

But because the component is unmounted, the state cannot be set. Therefore a memory leak will happen.

In **React 18** this case is managed by React itself, but is good practice you should cancel the try to set a state on unmounted components anyway.

A better approach for an api call would be to use **AbortController** api.

Once the component unmounts you call **AbortController** and the api request stops there.

But **AbortController** is only available for fetch or in **Axios**.

You may be doing something different in that **useEffect**.

For example, if you use **Firebase**, it gives you an unsubscribe api that you can call on the cleanup function. But Firebase is a happy case.

What if the async operation you perform in **useEffect** has no method to be cancelled?

For example you perform some DOM updates etc?

In that case you may use a variable helper function like in this example:

69. What is a HOC and why you might need it (sometimes).

To understand why HOCs exist and why Hooks have replaced them quite completely after they were introduced, we need to understand the problems one and the other try to solve.

Both of them are React tools which help with sharing logic between components mainly. And I am talking about custom hooks mainly. Native hooks are React components APIs.

Besides that, hooks also solve a lot of other unrelated problems.
They brought the functional paradigm to React and can be seen as a functional programming implementation.

HOCs on the other side were extremely powerful. If we do not talk about class components, and we don't, HOCs allow function compositions also.

But HOC could become verbose when they had to deal with complex logic.

Meanwhile hooks apply very well the separation of concerns principles, are tied to a component instance and they are much more readable on the human side thean HOCs.

However, one can be easily a replacement for the other.

But when can a HOC still be a better choice?

I think they can successfully replace hooks when they are used locally.

For example in Forms. A form container component can take as argument form elements and enhance them with some functionality.

The relationship between parent and children is highly tied and the combo always works well together.

It works well with hooks too, but the data should be passed outside the strict relationship between parent and children in an external entity which is the hook.

HOCs are React components which take other components as arguments and return those components with some functionality added.

In JavaScript:

The exact same principle applies to React components (which are just JavaScript functions basically).

A component can take another component as an argument and return the component with some functionality enhanced.

Let's say you have a custom hook which reacts to some state change.

If you plugin the custom hook in 2 different components, the component which sets the state will have the hook reacting. The other one won't.

A hook instance belongs only to the component calling it and cannot be shared. You need to uplift the hook to the parent and pass data as props down to the children that need it.

But using a HOC you can easily share the updated value by that hook to the 2 different components!

In the example below the hook count output is passed to Component through the **withComponent** HOC.

It could be passed to whatever component that **withComponent** can take as an argument.

The HOC adds the title and the **someFunctionalityMock** which "enhances" the Component, and could enhance any component passed as argument.

```tsx
const useCount = () => {
  const [count, setCount] = useState(0);

  const useCountUpdater = () => {
    setCount(count + 1);
  };
  return { count, useCountUpdater };
};

const Component: React.FC<{ title: string; num: number }> = ({
  title,
  num
}) => {
  const { count, useCountUpdater } = useCount();
  return (
    <div>
      <h1>{title}</h1>
      <h1>{num}</h1>
      {count} <button onClick={useCountUpdater}>Click</button>
    </div>
  );
};

const withComponent =
  (WrappedComponent: React.FC<{ title: string; num: number }>) =>
  (props: any) => {
    const someFunctionalityMock = Math.random();
    return (
      <>
        <WrappedComponent
          title="A simple HOC"
          num={someFunctionalityMock}
          {...props}
        />
      </>
    );
  };

export default withComponent(Component);
```

Recap: A HOC is a component container which can take other components as arguments and add functionality to them.

They can easily become verbose and error prone if the logic is too big.

After the hooks were introduced HOCs lost their importance.

But as you can see, HOCs can still be very useful to pass functionality to different children components and reactivity too (reactivity that you can't have with hooks).

70. What is Render Props and why you might need it.

A good example of render props can be seen in Formik library for handling forms:

```tsx
<Formik initialValues={{ email: '' }} onSubmit={onSubmitForm} validationSchema={resetSchema}>
  {({ handleChange, handleBlur, handleSubmit, values, errors, touched }) => (
    <>
      <TextInput
        keyboardType="email-address"
        placeholder={i18n('ForgotPasswordScreen.enterYourEmail')}
        onChangeText={handleChange('email')}
        onBlur={handleBlur('email')}
        value={values.email}
        isValid={!!values.email && !errors.email}
        errorMessage={touched.email ? errors.email : ''}
      />
      <Button
        hidden={!values.email}
        iconName="arrowRight"
        variant="round"
        loading={isLoading}
        position="absolute"
        onPress={handleSubmit}
      />
    </>
  )}
</Formik>
```

Also the render props pattern is a bit forgotten after the hooks took over and functional components, it still might be useful to learn and to use.

Render props is an advanced React pattern which allows one component to accept a function as prop.

The function can be called from the props or from the children props.

When and why you might need this pattern instead of hooks?

Working with forms indeed and if you do not use any library like Formik.

Anyway, nowadays this pattern is less and less used and it’s difficult to find it in any code base written in the last, let’s say 2 years.

The render props pattern is more reliable than hooks from an encapsulation point of view of your components!

Render props are very useful if you want to isolate part of the JSX and inject some state without introducing side effects to your components.

71. What is the unidirectional flow data Flux and why does it matter in React.

React is declarative.

And this matters the most.

But what does that mean exactly?

To answer this question we need to answer a question first: What React wants to achieve basically?

React is an UI Library. It manages the UI for us. We do not interact with the DOM directly (apart when using refs), but React does that for us.

So, what does declarative mean? It means you don’t need and you don’t have to write code **which tells the DOM what to do, when to do it and where to do it.**

React is very good at doing all of this for us. What React needs form us instead, is to feed it with data frame after frame.

And the data we feed React with is divided in snapshots (or frames), like it was a picture of all the data and how it looks at every given moment in time.

React then acts, or better said - reacts - on those snapshots of data in a way it knows how to do it the best.

The snapshot of data at any given moment in time is a “**picture of data**” where every single piece of data is in its place having its given form.

The next snapshot is another “picture” where every piece of data may be painted in different places with different shapes.

All these snapshots, or pictures, are composing a movie to say so.

The **movie** it’s what you see on the screen one moment after another moment when a new frame takes the place of the old one.

We usually watch a movie from beginning to the end.

The data flows from top of the app to the bottom of it, frame after frame!

Unidirectional data flow describes a one-way data flow where the data can move in only one pathway top to bottom usually.

React uses unidirectional data flow at a conceptual level.

The data coming down to children from the parent is known as **props**.

Data produced at the components level is called local state. Data entering the app form outside is an external state and if it is used in many places of the app it is called global state.

You can only theoretically transfer data from parent to child and not vice versa.

This means that the child components cannot update or modify the data on their

parents, making sure that a clean data flow architecture is followed.

In practice we may need to use a callback function from children to set a state in the parent for example.

But that does not break the principle itself as the new state has an impact on children.

Top to bottom data flow, frame after frame.

72. Learn the difference between reactive and not reactive state to write better React code.

The definition of state can be something like: "**the entire snapshot of data at a given moment in time that the app has access to**!"

In order to understand what a "snapshot of data" means, you need to be able to distinguish between implementation code and resource code.

Implementation code might be for example a function. Resource code might be the arguments the function takes.

State is resource code. Sometimes it is easy to get confused and not make the distinction between the two.

For example an interface, is it implementation code or resource code?

In order to answer, you need to take a close look at it and understand if the code it contains is a placeholder for values that can be assigned during the

execution, and values that can change during execution.

The interface is a container of some sort of data shape and it is implementation code. The variable names it contains are resource code.

Here are some example of resource code in React:

- Api responses,
- State from useState,
- Values from refs,
- User actions,
- Hard coded variables.

Even though hard coded variables do not change, they are assigned on execution and are not implementation code.

Once you understand this, the next step is to clearly understand the difference between reactive and non reactive code.

React is a declarative library. That means, your duty is to feed it with that snapshot of data (every moment) and React will take care of it by painting every

snapshot of data into the UI, moment after moment.

Therefore, we can say React "reacts" to reactive data!

It does not do anything with non reactive data, the next moment than it did a moment ago.

For example a hard coded variable which contains a web link, will stay stable during all the execution so React cannot and does not need to react to it apart from when it is mounting.

But when a state in **useState** changes React needs to react immediately so the state in **useState** is absolutely reactive data.

The **api response** on the other hand is debatable as it is fed into the application at the beginning of execution and may stay as it is during all the execution span, means may spread along many moments from that snapshot of data.

However, if you are using React Query for example or any sync with the server mechanism, an api response might change quite often therefore I can affirm that an **api response** is reactive data too.

Once you understand what reactive data is and you can recognize it, you may paint into your imagination a series of snapshots in sequence and therefore understand what React will do with each of them.

73. Use enums to conditionally return from functions or even components.

Enums are a wonderful replacement for condition in functions.

For example, if you have a function which needs to switch between let's say: **requested**, **pending**, **approved**, **rejected** etc; instead of using **if elses** you can put those conditions into an enum and switch the return based on them:

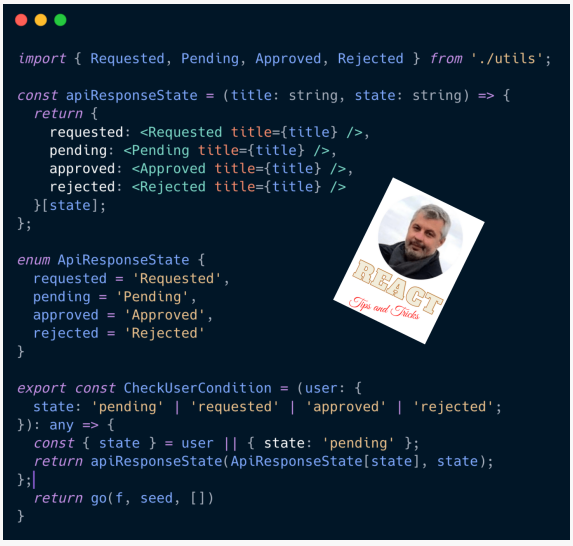
So **checkUserCondition** is going to be Requested. It also can be a component if

you want too. Just return a component from the enums instead of a string.

If we need to return a React component with props instead, then the approach should be different.

We need a "locked up" object to return the component to which we will pass the state.

In the example below, apart passing the state in order to get back the right component, we pass the title as well using the same state on an enum:



74. What global state manager to use?

The answer to this question is so opinionated, therefore I won’t tell you which tool to use to manage the global state.

As always, it depends. Depends on how big your app is, what tool the team is most comfortable with, what you need to achieve etc.

I’ll tell you which tool I prefer only and leave the choice to you.

There are tools which are very good at something, very good in general or very good at everything.

The most well known are:

- **Redux,**
- **Redux Toolkit (RTK),**
- **Recoil,**
- **Zustand,**
- **Valtio,**
- **Jotai,**
- **Mobx**

There are also particular tools which can’t be put in the category of global state manager because they can only do one thing.

For example, in order to manage the server state React Query is the best tool out there in my opinion.

It manages the async state, the api calls basically, in a brilliant way.
But it does not manage the client state.
For that you need to combine it with some other tool.

My preferred one is Recoil in combination with React Query.

But again, this is very opinionated so I invite you to read and inform yourself well before deciding which one to use.

75. Learn what shallow compared values are and why they matter a lot in React.

Shallow comparison takes into account objects or functions identities and not their properties or return boolean values from the comparison.

An object may contain exactly the same properties with the same values assigned to them but have a new identity because it was somehow and somewhere mutated.

Same for functions. Old functions can be garbage collected and new functions can be born in their place with totally new identities.

React docs say this:

“If you provide a custom **arePropsEqual** implementation, you must compare every prop, including functions.

Functions often **close over** the props and state of parent components. If you return true when

oldProps.onClick !== newProps.onClick, your component will keep “seeing” the props and state from a previous render inside its onClick handler, leading to very confusing bugs.

Avoid doing deep equality checks inside **arePropsEqual** unless you are 100% sure that the data structure you’re working with has a known limited depth.

Deep equality checks can become incredibly slow and can freeze your app for many seconds if someone changes the data structure later.”

What does this mean?

Always React docs give us an example about a compare props function can be implemented:

```
function arePropsEqual(oldProps, newProps) {
  return (
    oldProps.dataPoints.length === newProps.dataPoints.length &&
    oldProps.dataPoints.every((oldPoint, index) => {
      const newPoint = newProps.dataPoints[index];
      return oldPoint.x === newPoint.x && oldPoint.y === newPoint.y;
    })
  );
}
```

But what does a component that re-renders close over old props from parent means?

It simply means your comparison function might be kicked off on component render but the parent re-renders for its own business and the props or state value on it might change.

But because the comparison function is already running, it can’t see the new props from the parent.

In this way everything is blown up and unknown bugs may happen.