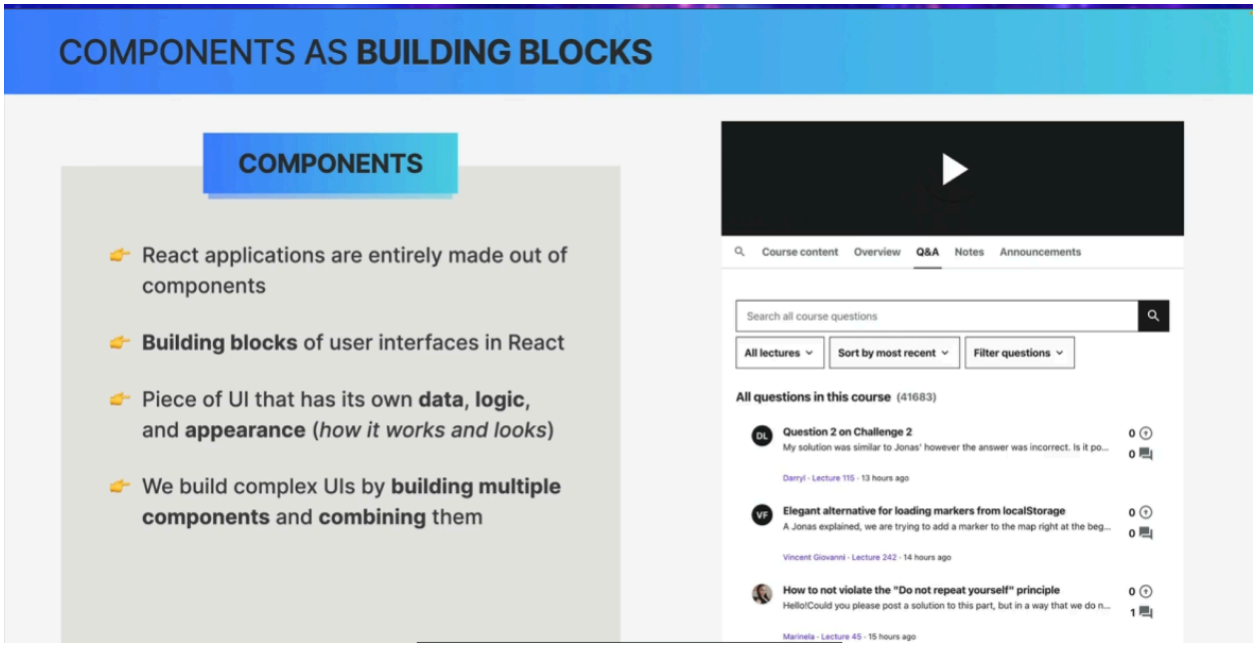
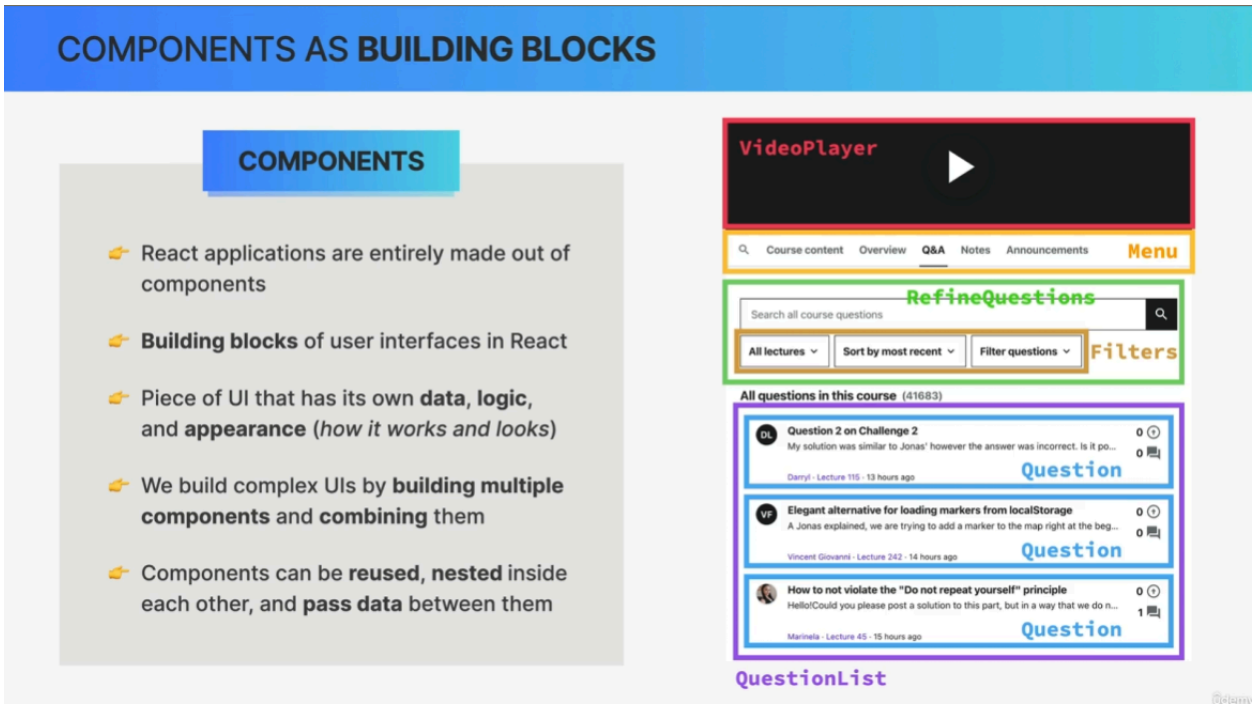
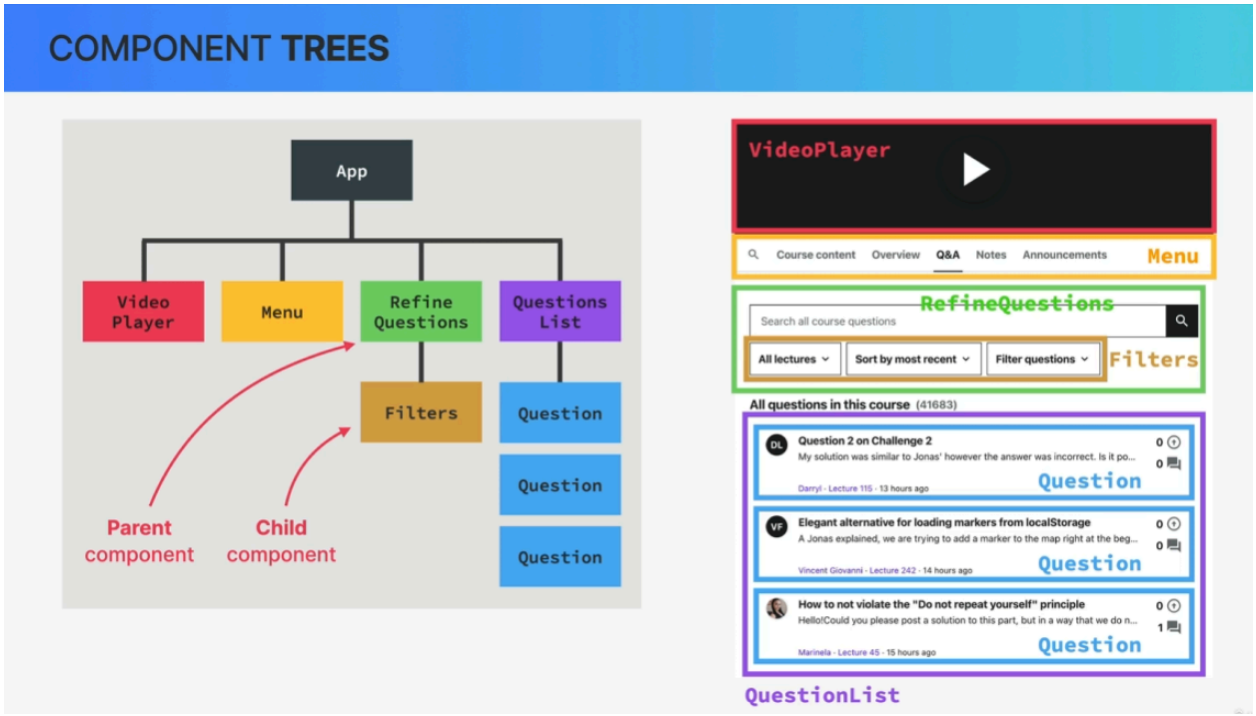
Section 1

Component







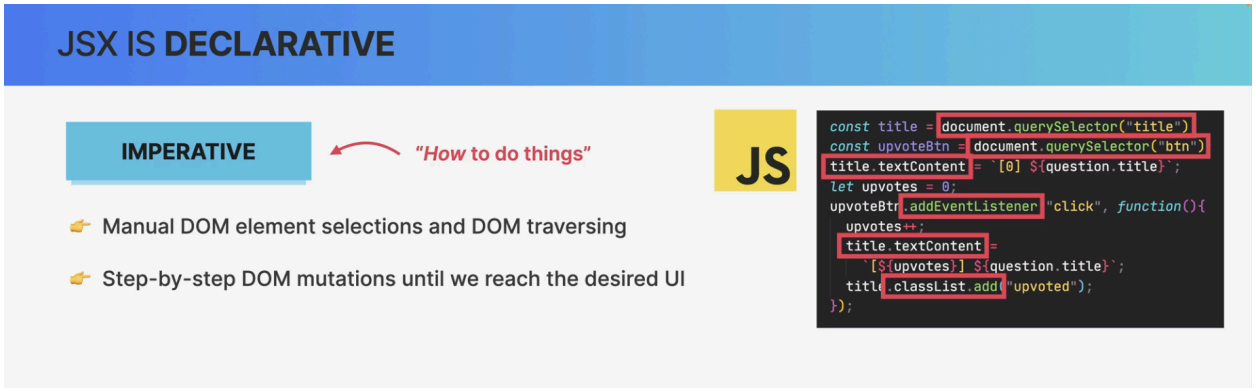
in React is just a function. Now, these functions in React, so these components can return something called JSX, which is a syntax that looks like HTML and describes what we can see on the screen.

JSX

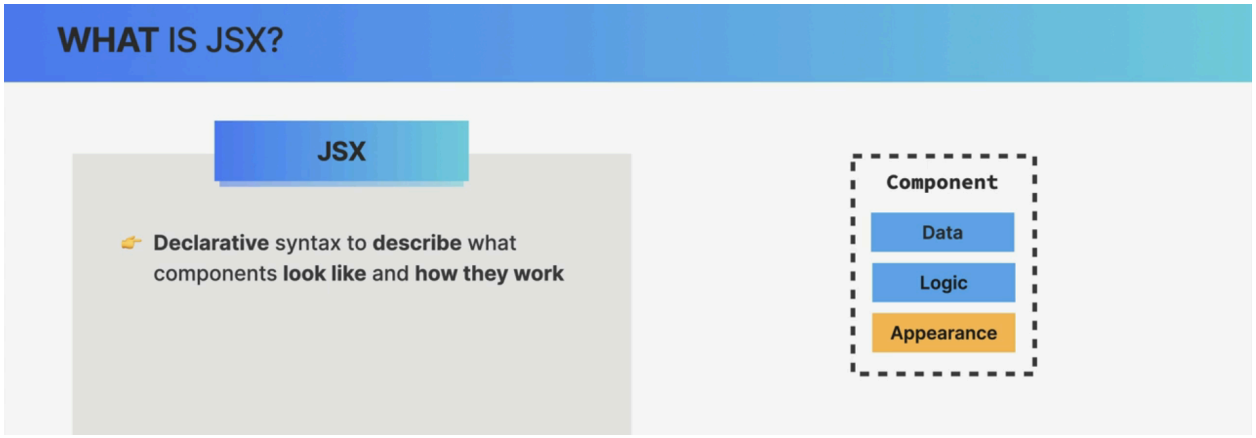
It is basically just like HTML, but we can add some JavaScript to it.

- Declarative syntax to describe what components look like and how they work

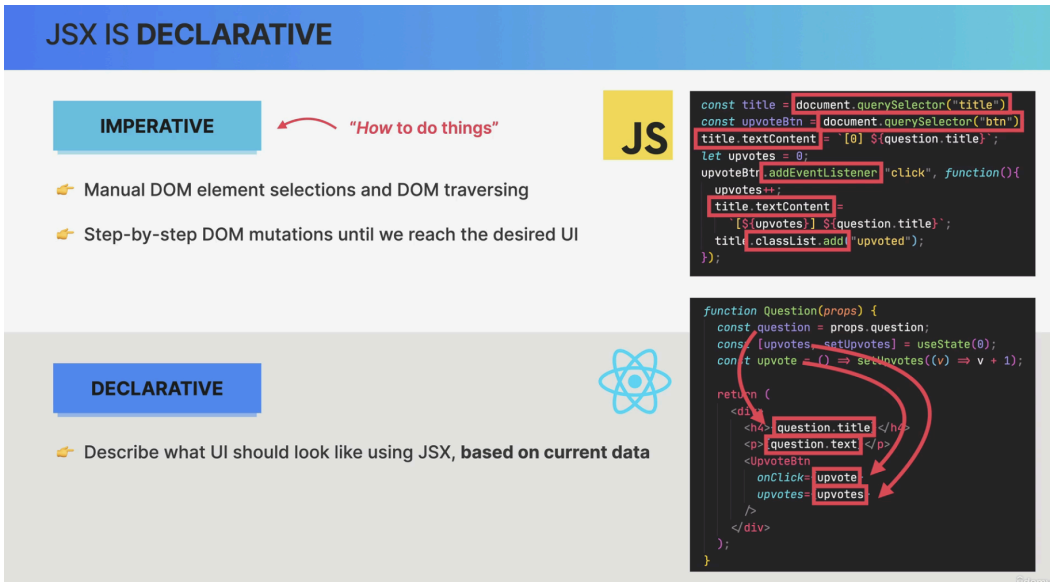
So when we try to build UIs using vanilla JavaScript, we will by default use an imperative approach. This means that we manually select elements, traverse the DOM, and attach event handlers to elements. Then each time something happens in the app like a click on the button, we give the browser step-by-step instructions on how to mutate those thumb elements until we reach the desired updated UI. So in the imperative approach, we basically tell the browser exactly how to do things.

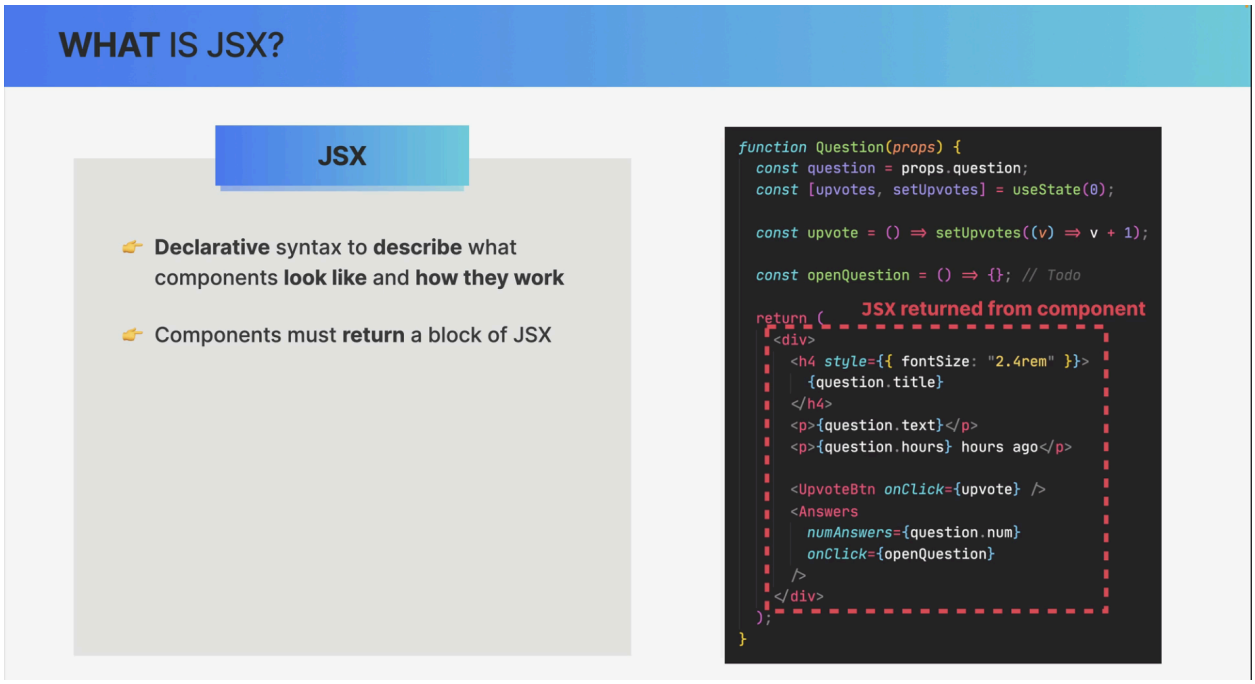


So, a declarative approach is to simply describe what the UI should look like at all times, always based on the current data that's in the component.

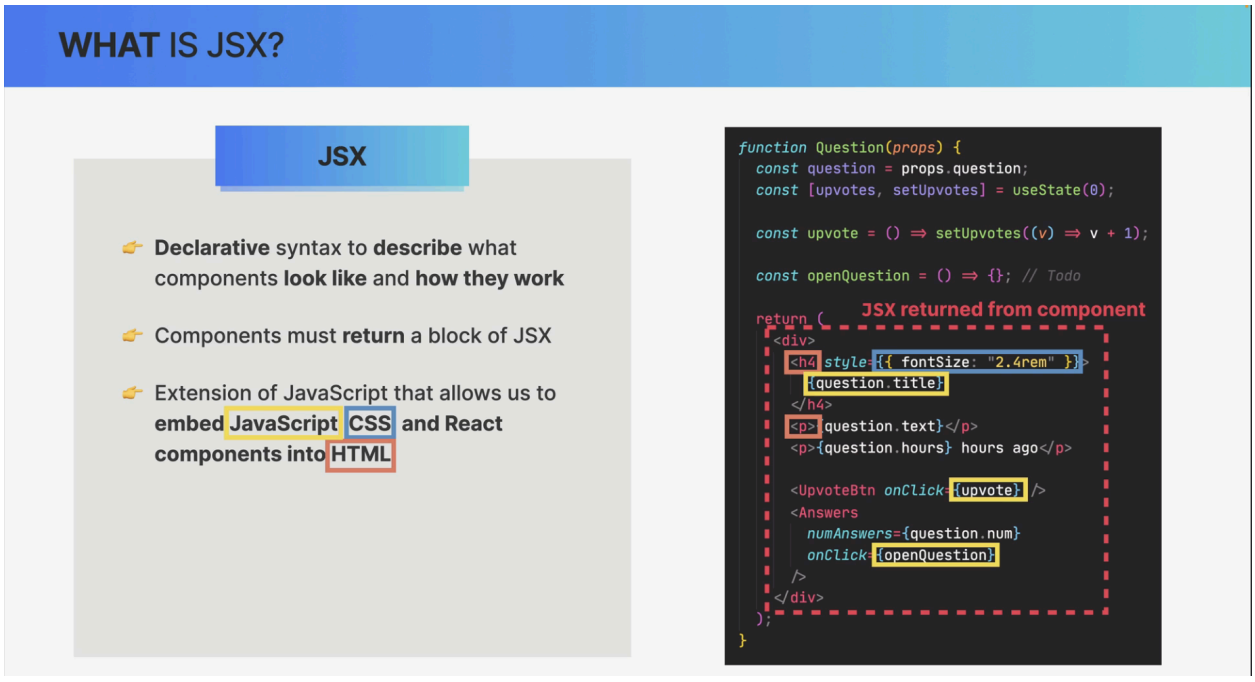


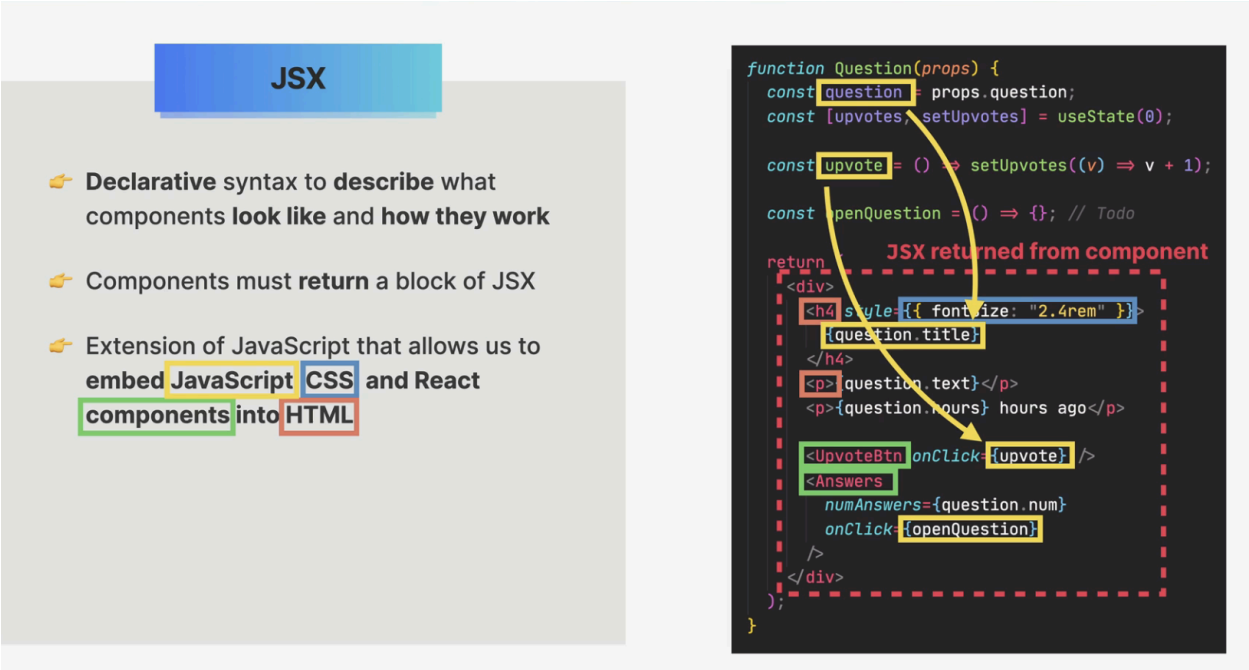
- Components must return a block of JSX





- Extension of JavaScript that allows us to embed JavaScript, CSS, and React components into HTML

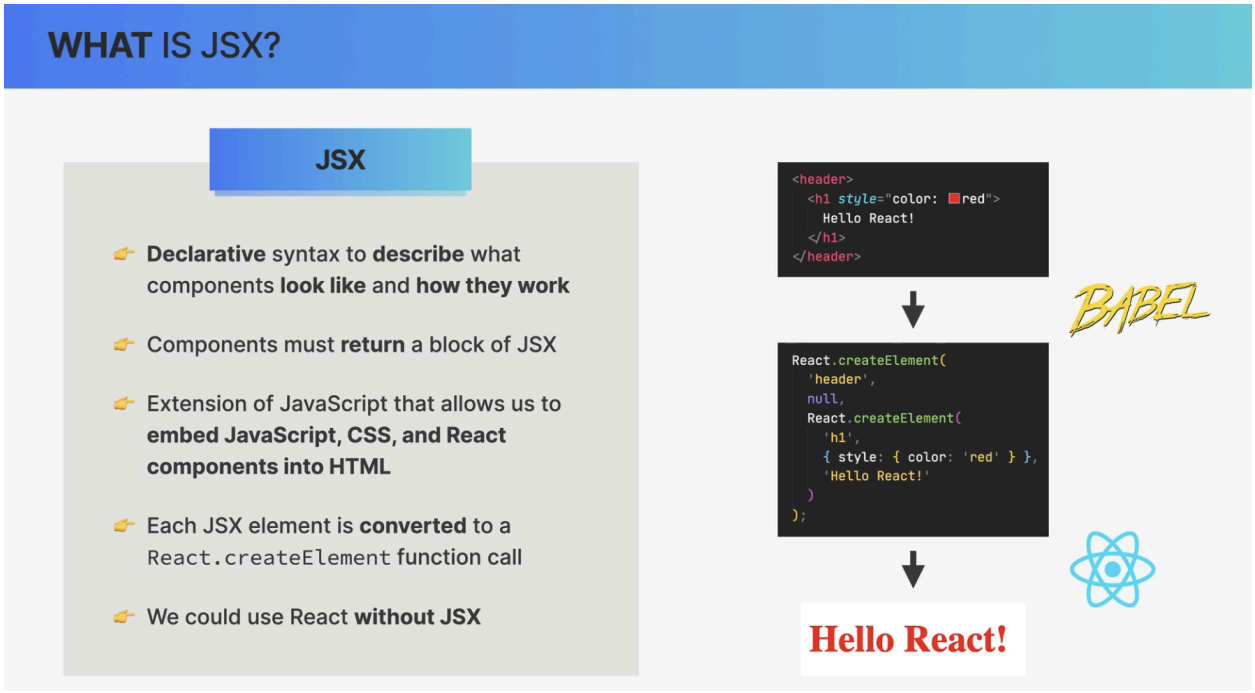




- Each JSX element is converted to a React.createElement function call

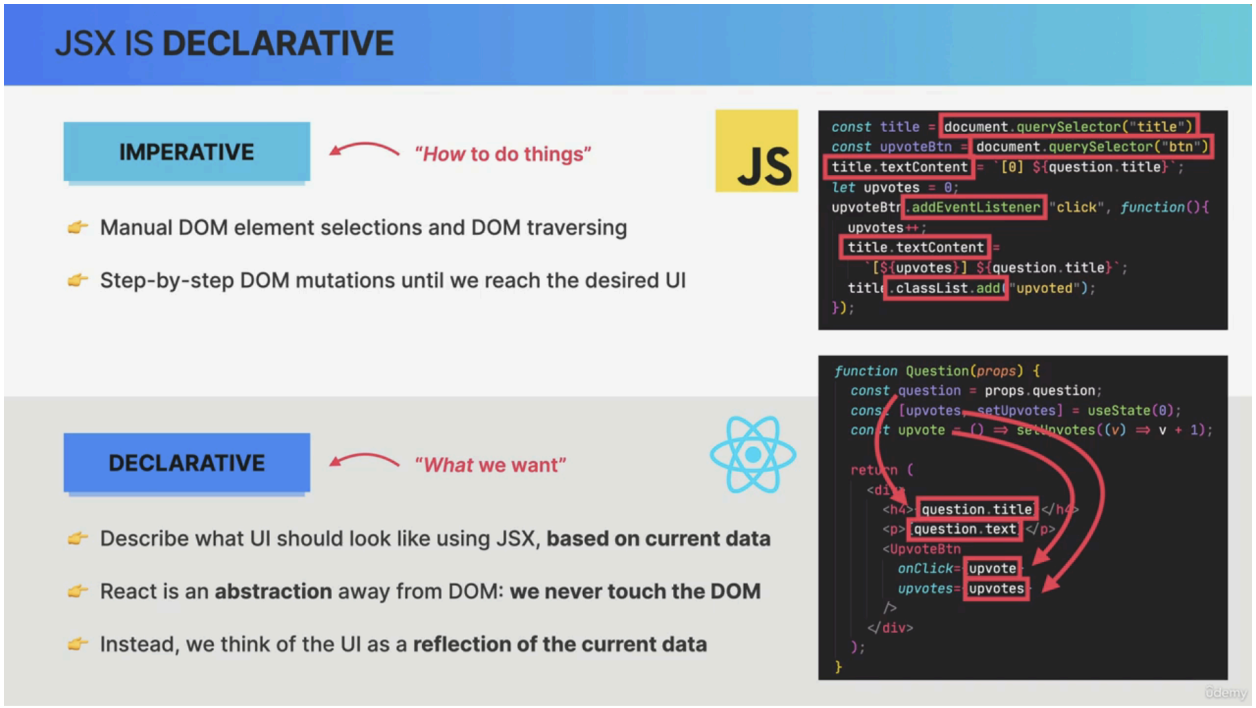
But anyway, this conversion is necessary because browsers of course, do not understand JSX. They only understand HTML. So behind the scenes, all the JSX that we write is converted into many nested React.createElement function calls. And these function calls are what in the end, create the HTML elements that we see on the screen.

Now, what this means is that we could actually use React without JSX at all.



And so again, basically, we use JSX to describe the UI based on props and state. So the data that's currently in the component and all that happens without any DOM manipulation at all.

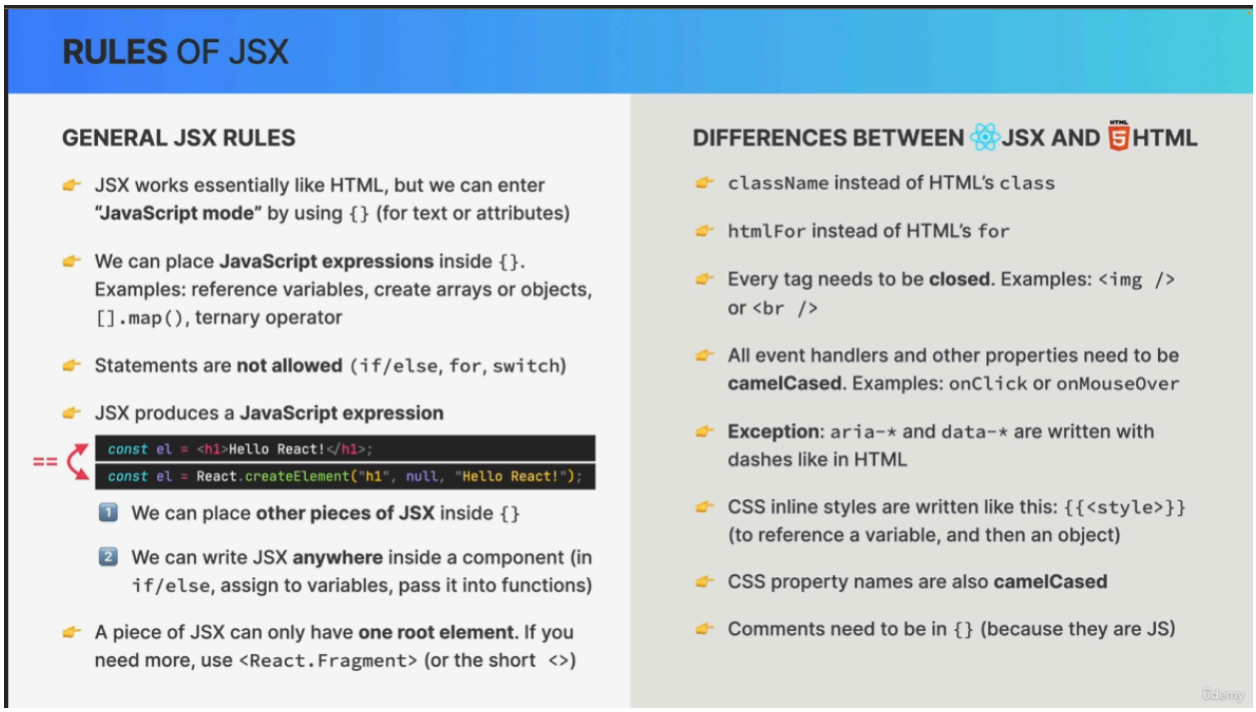
So, there are no Query selectors, no ad event listeners, no class list, and no text content properties anywhere to be seen here because, in fact, React is basically a huge abstraction away from the DOM, so we, developers never have to touch the DOM directly.


JSX IS DECLARATIVE
IMPERATIVE
"How to do things"
Manual DOM element selections and DOM traversing
Step-by-step DOM mutations until we reach the desired UI
JS
const title = document.querySelector("title")
const upvoteBtn = document.querySelector("btn")
title.textContent = `[0] ${question.title}`;
let upvotes = 0;
upvoteBtn.addEventListener("click", function(){
upvotes++;
title.textContent =
`[${upvotes}] ${question.title}`;
title.classList.add("upvoted");
});
DECLARATIVE
"What we want"
Describe what UI should look like using JSX, based on current data
React is an abstraction away from DOM: we never touch the DOM
Instead, we think of the UI as a reflection of the current data
function Question(props) {
const question = props.question;
const [upvotes, setUpvotes] = useState(0);
const upvote = () => setUpvotes((v) => v + 1);
return (

<h4>{question.title}</h4>
{question.text}
<UpvoteBtn
onClick={upvote}
upvotes={upvotes}
/>

);
}

Rules of JSX


RULES OF JSX
GENERAL JSX RULES
JSX works essentially like HTML, but we can enter "JavaScript mode" by using {} (for text or attributes)
We can place JavaScript expressions inside {}. Examples: reference variables, create arrays or objects, [].map(), ternary operator
Statements are not allowed (if/else, for, switch)
JSX produces a JavaScript expression
==
const el = <h1>Hello React!</h1>;
const el = React.createElement("h1", null, "Hello React!");
1 We can place other pieces of JSX inside {}
2 We can write JSX anywhere inside a component (in if/else, assign to variables, pass it into functions)
A piece of JSX can only have one root element. If you need more, use <React.Fragment> (or the short <>)
DIFFERENCES BETWEEN JSX AND HTML
className instead of HTML's class
htmlFor instead of HTML's for
Every tag needs to be closed. Examples: <img /> or
All event handlers and other properties need to be camelCased. Examples: onClick or onMouseOver
Exception: aria-* and data-* are written with dashes like in HTML
CSS inline styles are written like this: {{<style>}} (to reference a variable, and then an object)
CSS property names are also camelCased
Comments need to be in {} (because they are JS)

GENERAL JSX RULES

JSX works essentially like HTML, but we can enter "JavaScript mode" by using {} (for text or attributes)

👉 We can place JavaScript expressions inside {}.
Examples: reference variables, create arrays or objects, [].map(), ternary operator

👉 Statements are not allowed (if/else, for, switch)

👉 JSX produces a JavaScript expression
1️⃣ We can place other pieces of JSX inside {}
2️⃣ We can write JSX anywhere inside a component (in if/else, assign to variables, pass it into functions)
==

👉 A piece of JSX can only have one root element. If you need more, use <React.Fragment> (or the short <>)

DIFFERENCES BETWEEN JSX AND HTML

👉 className instead of HTML's class

👉 htmlFor instead of HTML's for

👉 Every tag needs to be closed. Examples: <img /> or

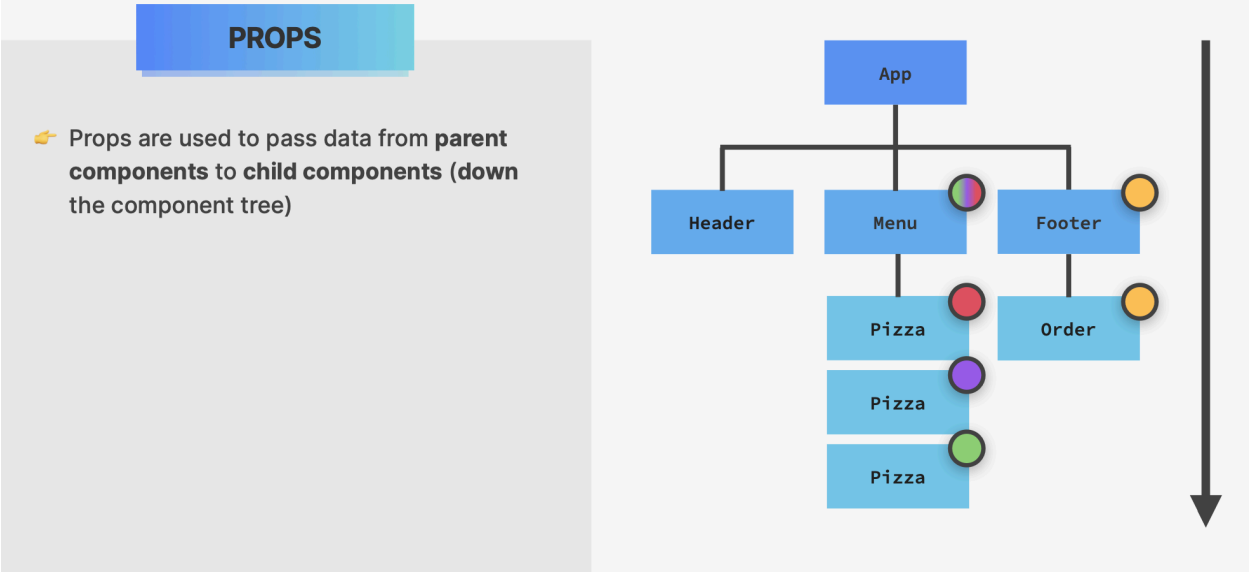
👉 All event handlers and other properties need to be camelCased. Examples: onClick or onMouseOver
👉 Exception: aria-* and data-* are written with dashes like in HTML

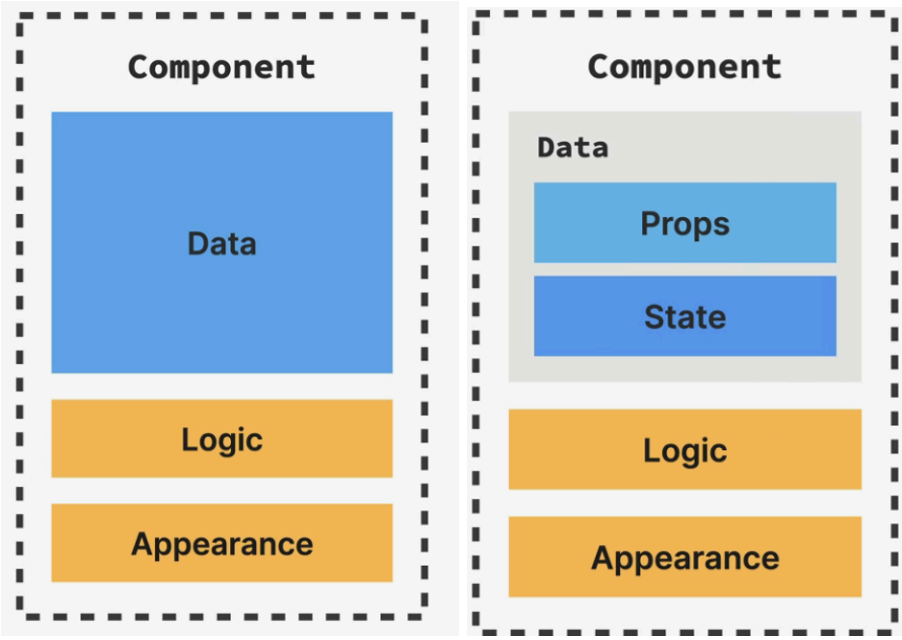
👉 CSS inline styles are written like this: {{<style>}} (to reference a variable, and then an object)

👉 CSS property names are also camelCased
👉 Comments need to be in {} (because they are JS)

Props

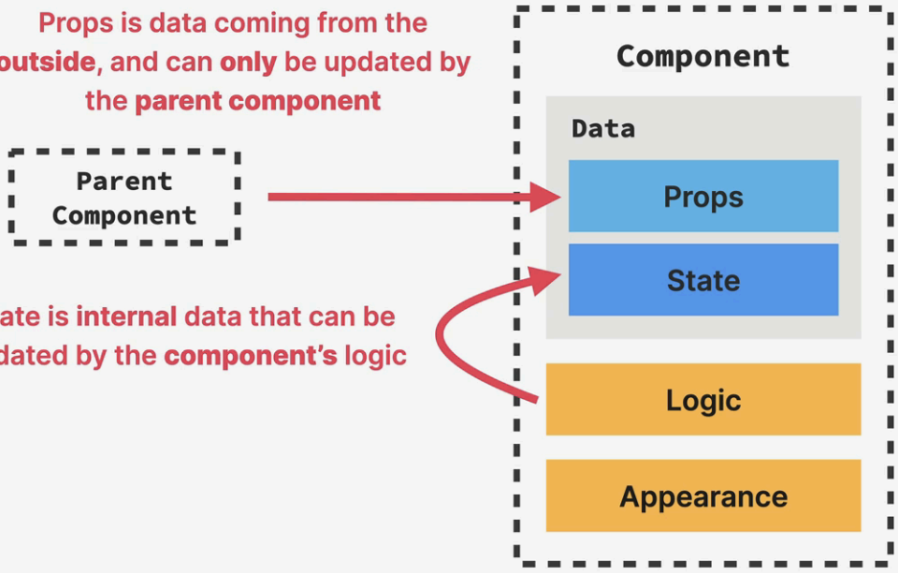
And the prop is basically just like parameters to a function.





PROPS ARE **READ-ONLY!**

Props is data coming from the **outside**, and can **only** be updated by the **parent component**



State is internal data that can be updated by the **component's** logic

PROPS

- Props are used to pass data from **parent components** to **child components** (**down** the component tree)
- Essential tool to **configure** and **customize** components (like function parameters)
- With props, parent components **control** how child components look and work

```
<Menu>
  <Button bgColor="blue" text="New" />
  <Button bgColor="green" text="Edit" />
  <Button bgColor="red" text="Delete" />
</Menu>
```

PROPS

- Props are used to pass data from **parent components** to **child components** (**down** the component tree)
- Essential tool to **configure** and **customize** components (like function parameters)
- With props, parent components **control** how child components look and work
- **Anything** can be passed as props: single values, arrays, objects, functions, even other components

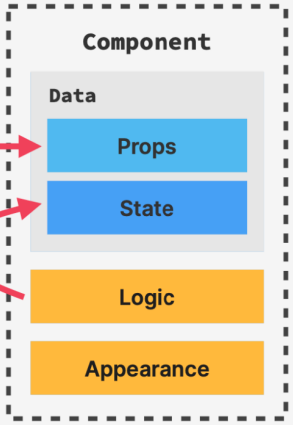
```jsx
function CourseRating() {
  const [rating, setRating] = useState(0);

  return (
    <Rating
      text="Course rating"
      currentRating={rating}
      numOptions={3}
      options={["Terrible", "Okay", "Amazing"]}
      allRatings={{ num: 2390, avg: 4.8 }}
      setRating={setRating}
      component={Star}
    />
  );
}

function Star() {
  // To do
}
```

PROPS ARE **READ-ONLY!**

Props is data coming from the **outside**, and can **only** be updated by the **parent component**

Parent Component

State is internal data that can be updated by the **component's** logic



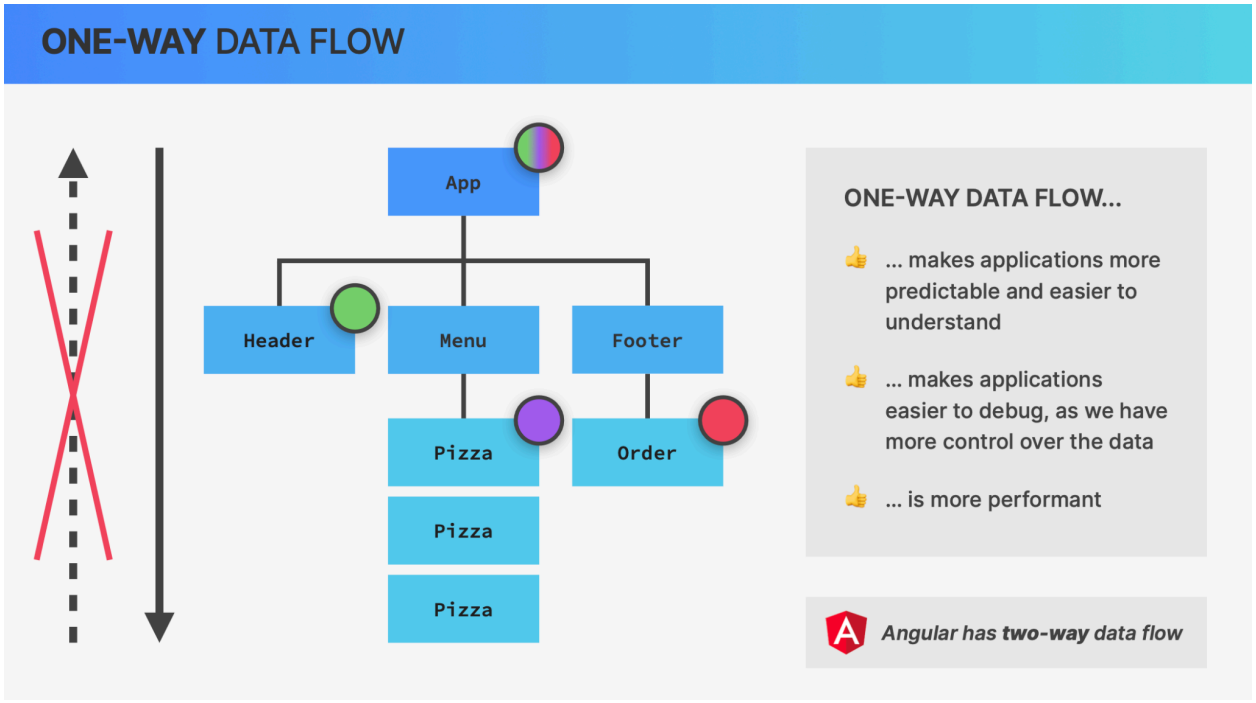

```jsx
let x = 7;

function Component(){
  x = 23
  return <h1>Number {x}</h1>
}
```

Don't do this!

- Props are read-only, they are **immutable**! This is one of React's strict rules.
- If you need to mutate props, you actually **need state**

WHY?

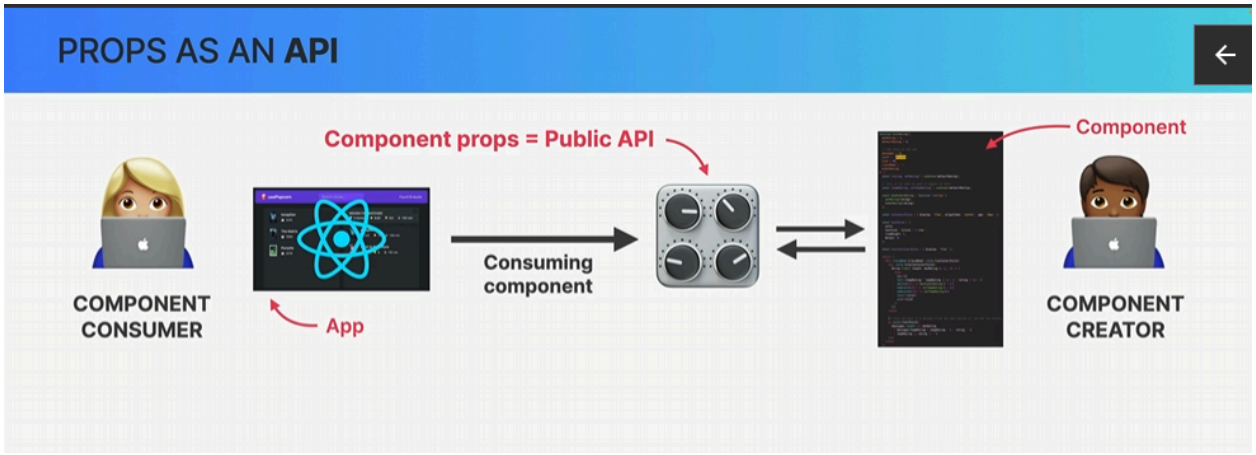
- Mutating props would affect parent, creating **side effects** (not pure)
- Components have to be **pure functions** in terms of props and state
- This allows React to optimize apps, avoid bugs, make apps predictable

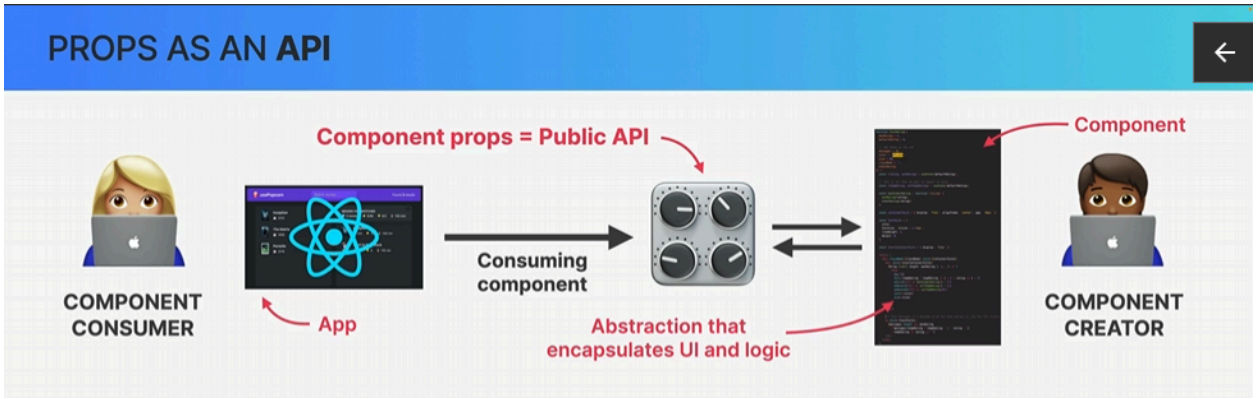


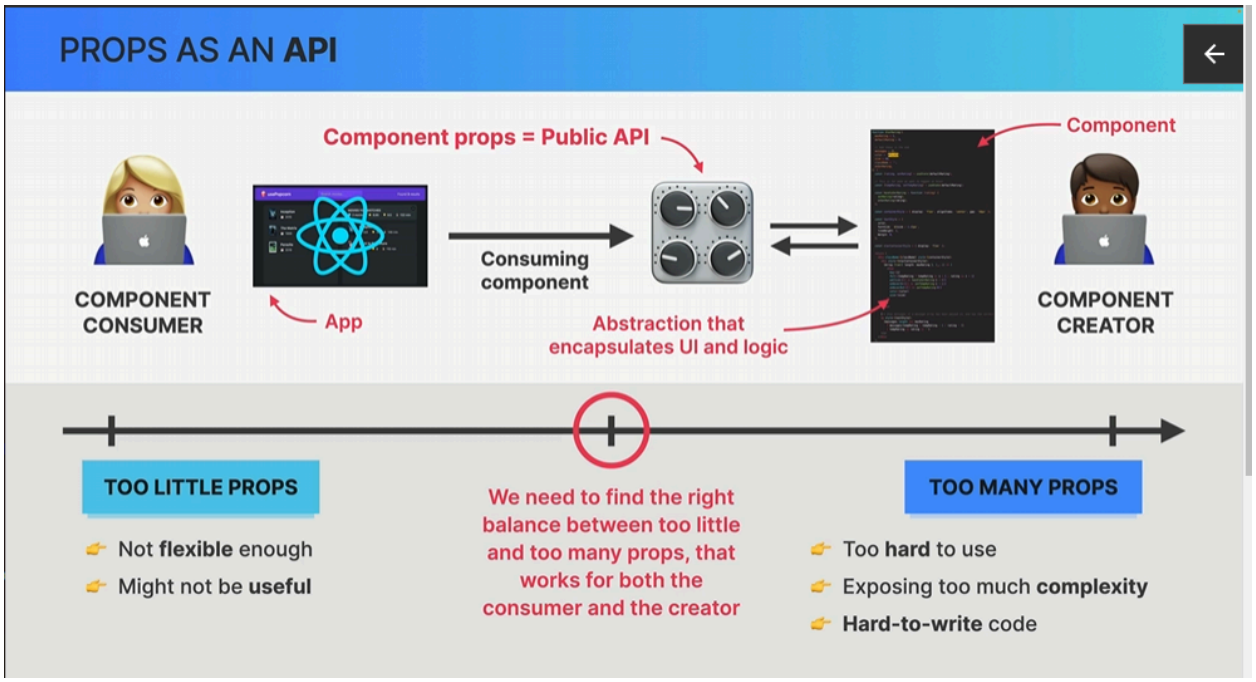
Props drilling

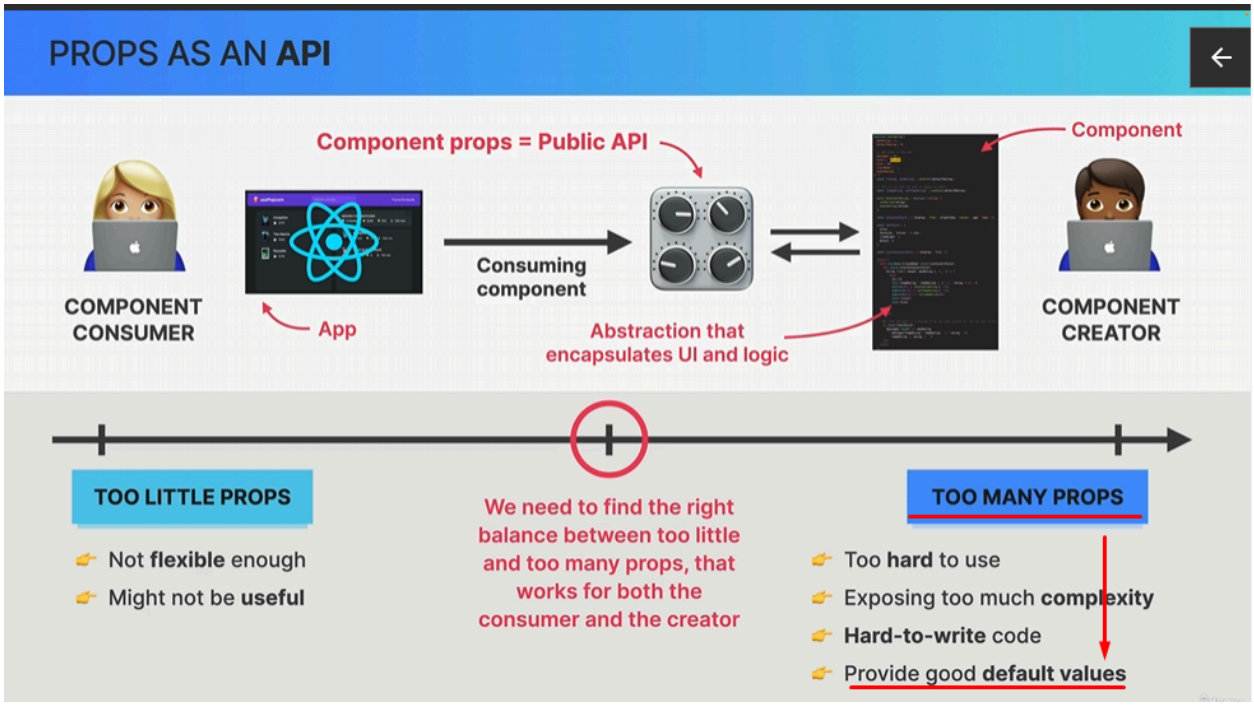
So basically prop drilling means that we need to pass some prop through several nested child components in order to get that data into some deeply nested component.

Props as an API




PROPS AS AN API
Component props = Public API
Component
COMPONENT CONSUMER
Consuming component
App
Abstraction that encapsulates UI and logic
COMPONENT CREATOR


PROPS AS AN API
Component props = Public API
Component
COMPONENT CONSUMER
Consuming component
App
Abstraction that encapsulates UI and logic
COMPONENT CREATOR
TOO LITTLE PROPS
Not flexible enough
Might not be useful
We need to find the right balance between too little and too many props, that works for both the consumer and the creator
TOO MANY PROPS
Too hard to use
Exposing too much complexity
Hard-to-write code



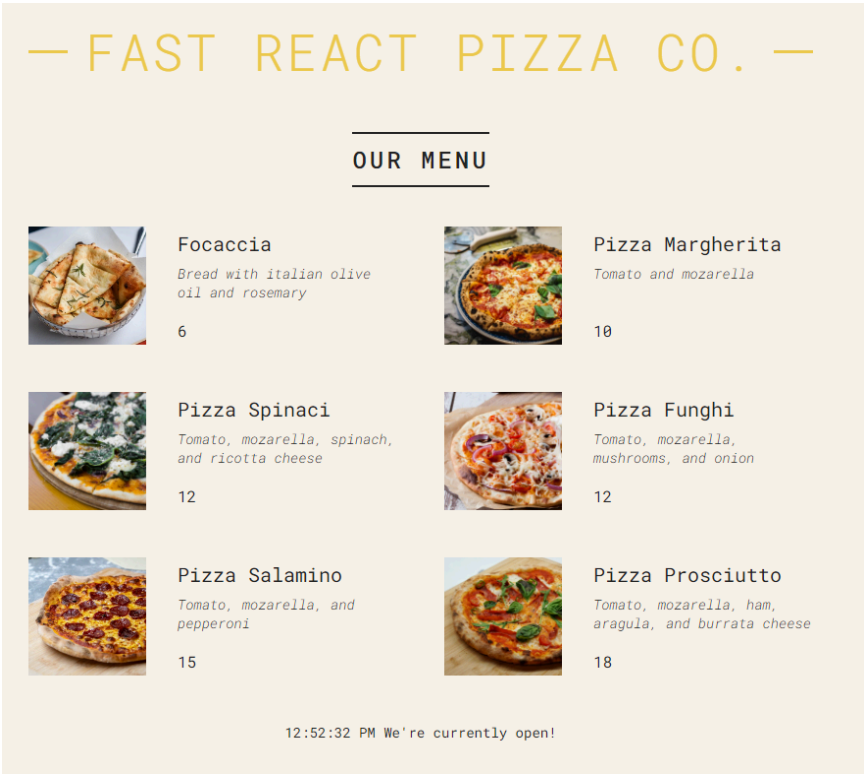
Rendering List

```
const pizzaData = [
  {
    name: "Focaccia",
    ingredients: "Bread with italian olive oil and rosemary",
    price: 6,
    photoName: "pizzas/focaccia.jpg",
    soldOut: false,
  },
  {
    name: "Pizza Margherita",
    ingredients: "Tomato and mozarella",
    price: 10,
    photoName: "pizzas/margherita.jpg",
    soldOut: false,
  },
];
```

```
<ul className="pizzas">
```

```
  {pizzaData.map((pizza) => (
    <Pizza key={pizza.name} pizzaObj={pizza} />
  ))}
</ul>;
```

```
function Pizza(props) {
  return (
    <li className="pizza">
      <img src={props.pizzaObj.photoName} alt={props.pizzaObj.name} />
      <div>
        <h3>{props.pizzaObj.name}</h3>
        <p>{props.pizzaObj.ingredients}</p>
        <span>{props.pizzaObj.price}</span>
      </div>
    </li>
  );
}
```



Conditional Rendering &&

```
function Menu() {
  const pizzas = pizzaData;
  const numPizzas = pizzas.length;
  return (
    <main className="menu">
      <h2>Our Menu</h2>

      {numPizzas > 0 && (
        <ul className="pizzas">
          {pizzaData.map((pizza) => (
            <Pizza key={pizza.name} pizzaObj={pizza} />
          ))}
        </ul>
      )}
    </main>
  );
}
```

Be careful here `numPizzas > 0 -> the value from the expression must be true or false. If the result of the expression is 0, then 0 will be displayed on the UI.`

`For example if the check is`

```
{numPizzas && (
        <ul className="pizzas">
          {pizzaData.map((pizza) => (
            <Pizza key={pizza.name} pizzaObj={pizza} />
          ))}
        </ul>
      )}
```

`Then if the numPizzas is 0, then the second condition will not executed at all, but the first expression which is numPizzas will give 0 as a result and then the 0 will be displayed on the UI. True, or false values are not displayed on the UI.`

Conditional Rendering ?

```
function Menu() {
  const pizzas = pizzaData;
  // const pizzas = [];
  const numPizzas = pizzas.length;
  return (
    <main className="menu">
      <h2>Our Menu</h2>

      {numPizzas > 0 ? (
        <ul className="pizzas">
          {pizzaData.map((pizza) => (
            <Pizza key={pizza.name} pizzaObj={pizza} />
          ))}
        </ul>
      ) : (
        <p>We are still working on our menu. Please come back later</p>
      )}
    </main>
  );
}
```

```
function Footer() {
  const hour = new Date().getHours();
  const openHour = 12;
  const closeHour = 22;
  const isOpen = hour >= openHour && hour <= closeHour;

  return (
    <footer className="footer">
      {isOpen ? (
        <div className="order">
          <p>We'are open until {closeHour}:00. Come visit us or order
online</p>
          <button className="btn">Order</button>
        </div>
      ) : (
        <p>
          We are happy to wellcome you between {openHour}:00 and
```

```
{closeHour}:00
        </p>
      )}
    </footer>
  );
}
```

Conditional Rendering With Multiple Returns

```
//useful when we want to render full components , no some piece of JSX
```

```
if (!isOpen)
  return (
    <p>
      We are happy to wellcome you between {openHour}:00 and {closeHour}:00
    </p>
  );
```

```
function Pizza(props) {
  if (props.pizzaObj.soldOut) return null;

  return (
    <li className="pizza">
      <img src={props.pizzaObj.photoName} alt={props.pizzaObj.name} />
      <div>
        <h3>{props.pizzaObj.name}</h3>
        <p>{props.pizzaObj.ingredients}</p>
        <span>{props.pizzaObj.price}</span>
      </div>
    </li>
  );
}
```

Setting Classes and Text Conditionally

```
function Pizza({ pizzaObj }) {
  return (
    // <li className={"pizza " + (pizzaObj.soldOut ? "sold-out" : "")}>
    // <li className={`pizza ${pizzaObj.soldOut && "sold-out"}`}>
    <li className={`pizza ${pizzaObj.soldOut ? "sold-out" : ""}`}>
      <img src={pizzaObj.photoName} alt={pizzaObj.name} />
      <div>
        <h3>{pizzaObj.name}</h3>
        <p>{pizzaObj.ingredients}</p>
        <span>{pizzaObj.soldOut ? "SOLD OUT" : pizzaObj.price}</span>
      </div>
    </li>
  );
}
```

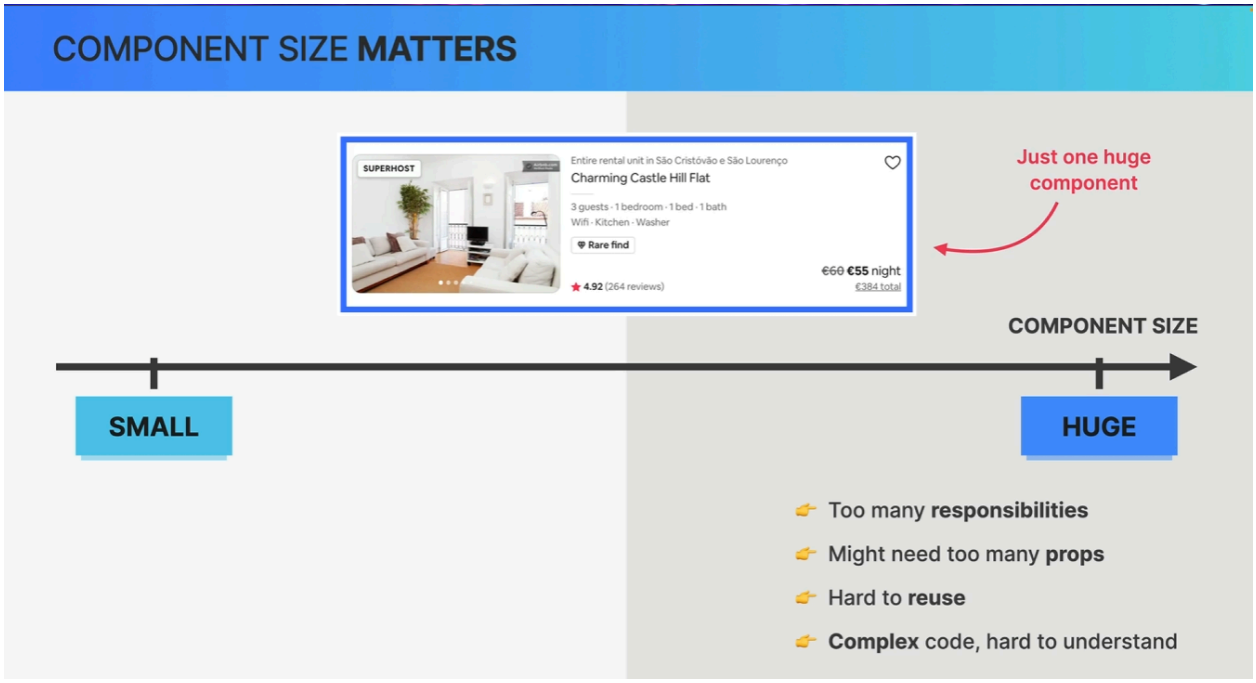
React Fragments

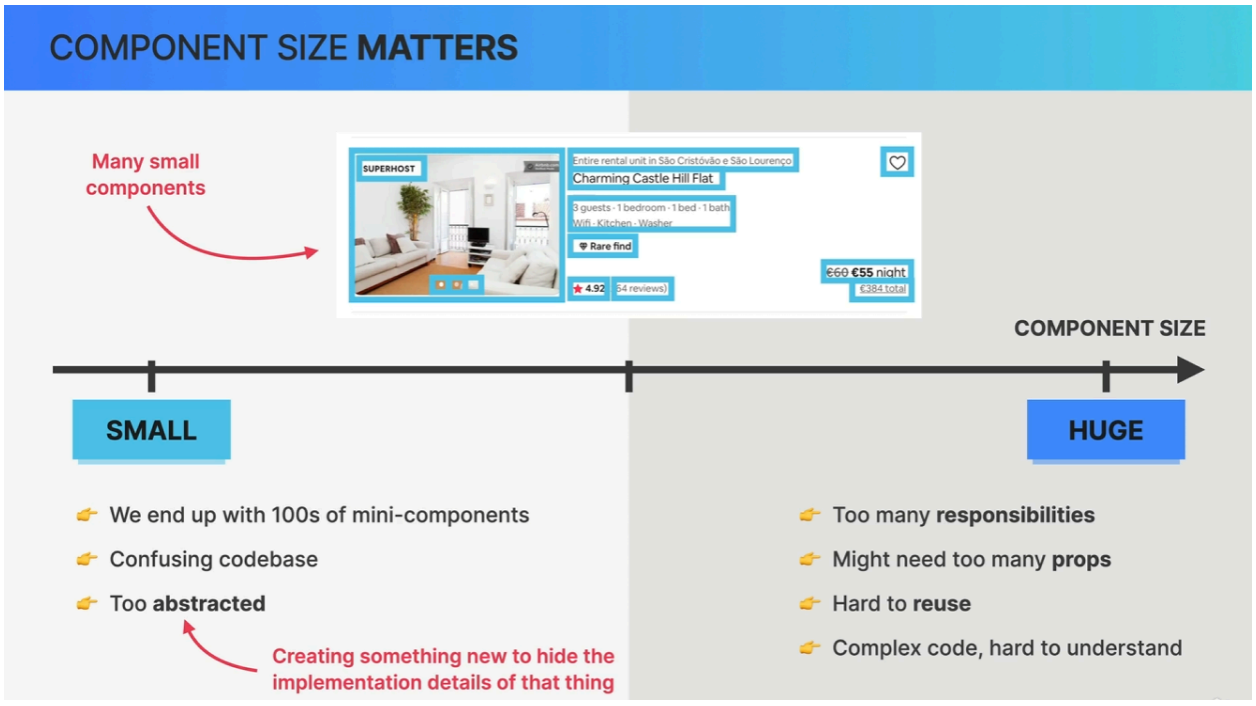
```
function Menu() {
  const pizzas = pizzaData;
  // const pizzas = [];
  const numPizzas = pizzas.length;
  return (
    <main className="menu">
      <h2>Our Menu</h2>

      {numPizzas > 0 ? (
        <>
          <p>
            Authentic Italian cuisine. 6 creative dishes to choose from.
All
            from our stone oven, all organic, all drlicious
          </p>
          <ul className="pizzas">
            {pizzaData.map((pizza) => (
              <Pizza key={pizza.name} pizzaObj={pizza} />
```

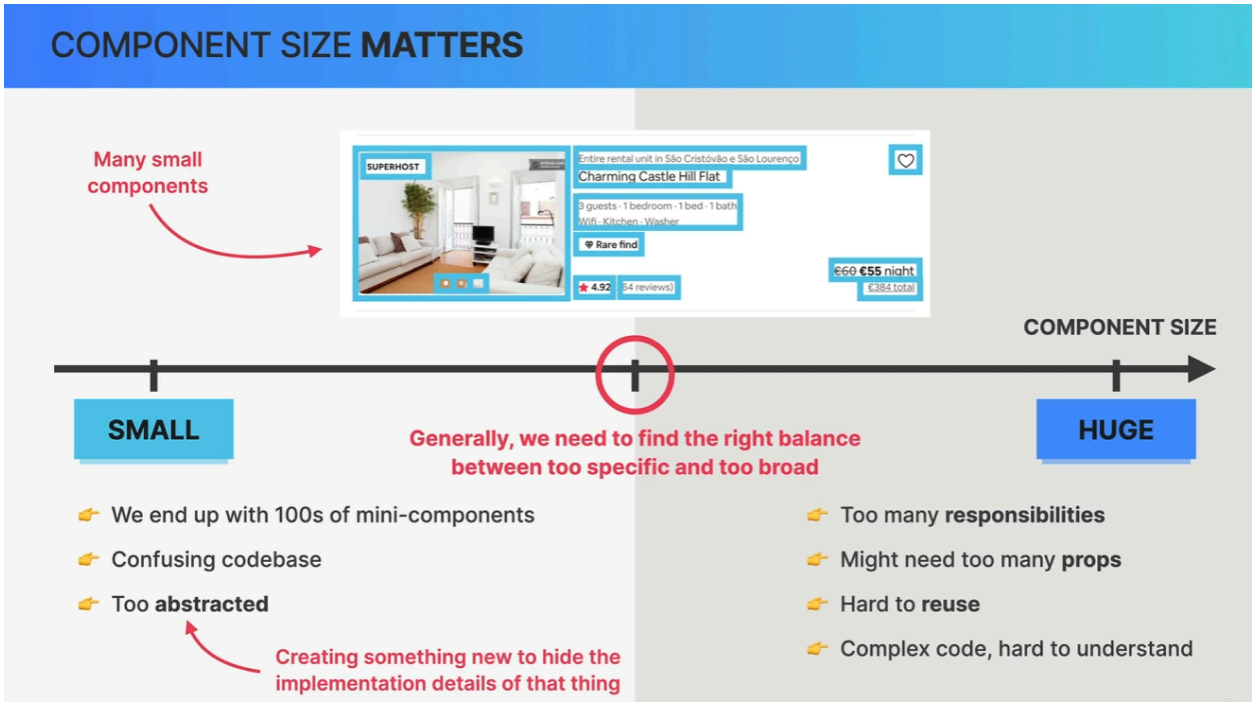
```
            ))}
          </ul>
        </>
      ) : (
        <p>We are still working on our menu. Please come back later</p>
      )}
    </main>
  );
}
```

If we need to put key attribute on the fragment the we must use this syntax

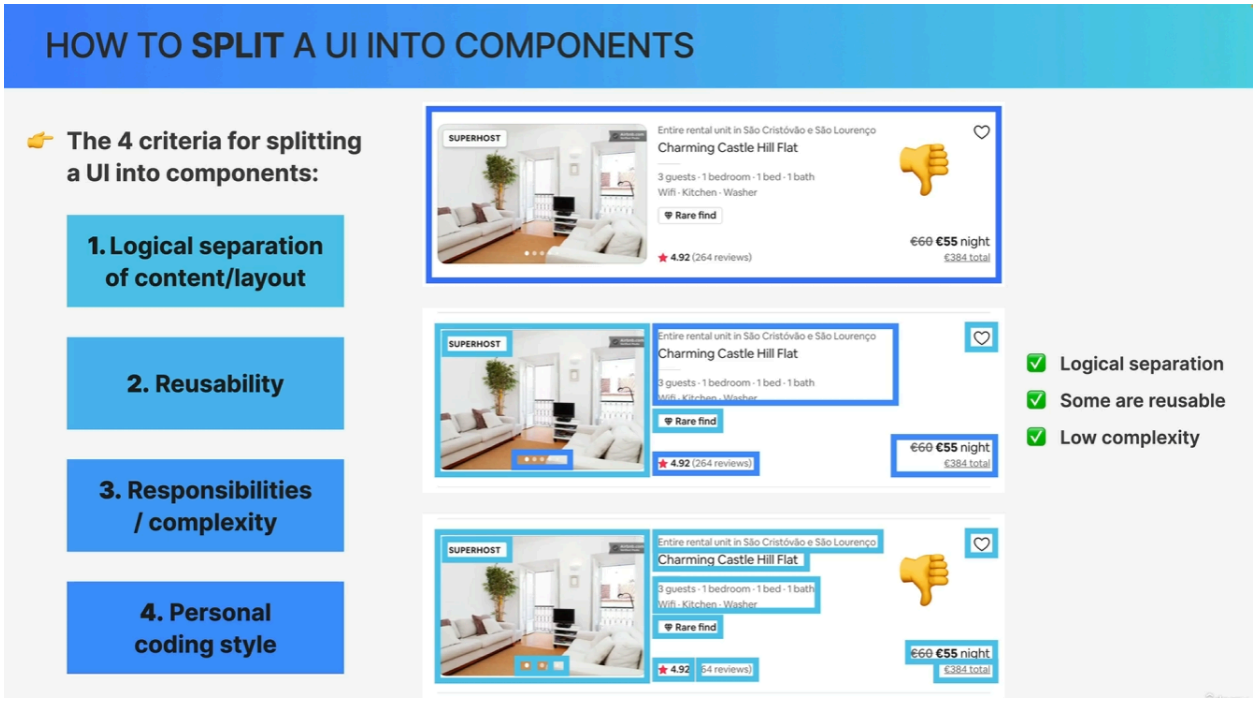
Component Size Matters



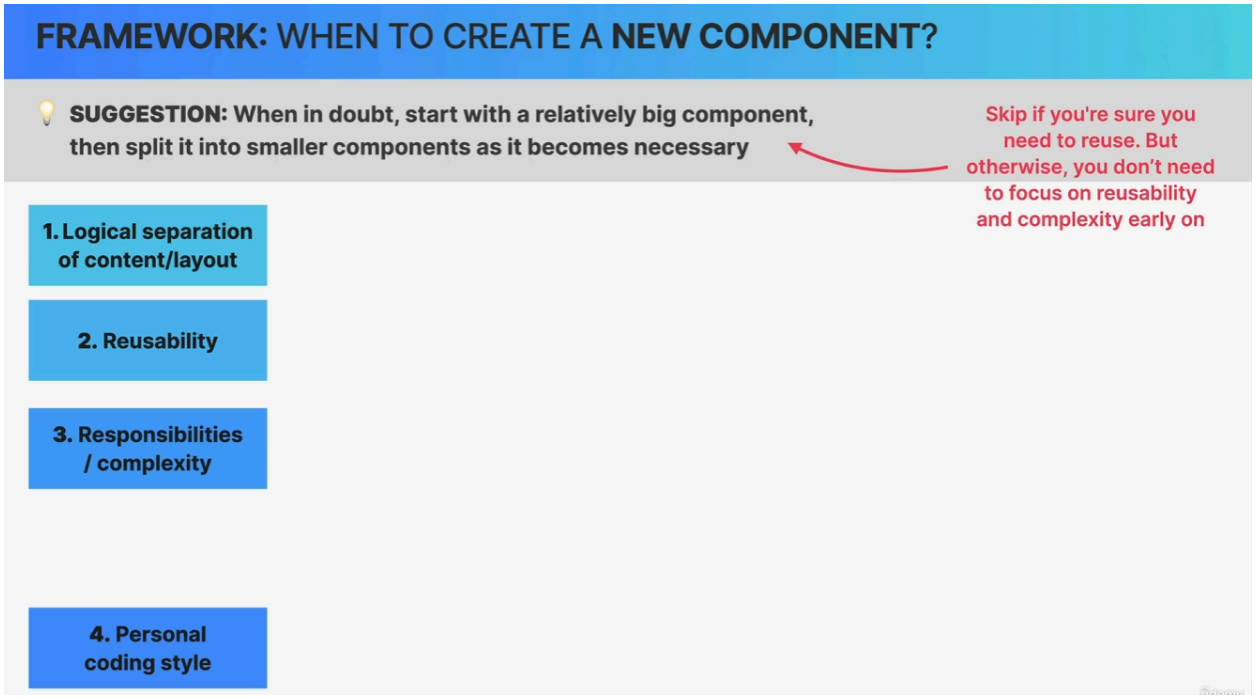




How to split UI into components



When to create a new component



FRAMEWORK: WHEN TO CREATE A NEW COMPONENT?

💡 **SUGGESTION: When in doubt, start with a relatively big component, then split it into smaller components as it becomes necessary**

Skip if you're sure you need to reuse. But otherwise, you don't need to focus on reusability and complexity early on

1. Logical separation of content/layout

- 👉 Does the component contain pieces of content or layout that **don't belong together**?

2. Reusability

- 👉 Is it possible to reuse part of the component?
- 👉 Do you **want** or **need** to reuse it?

3. Responsibilities / complexity

- 👉 Is the component doing too **many different things**?
- 👉 Does the component rely on too **many props**?
- 👉 Does the component have too **many pieces of state** and/or effects?
- 👉 Is the code, including JSX, too **complex/confusing**?

4. Personal coding style

- 👉 Do you prefer **smaller** functions/components?

You might need a new component

👋 *These are all **guidelines**... It will become intuitive over time!*

SOME MORE GENERAL GUIDELINES

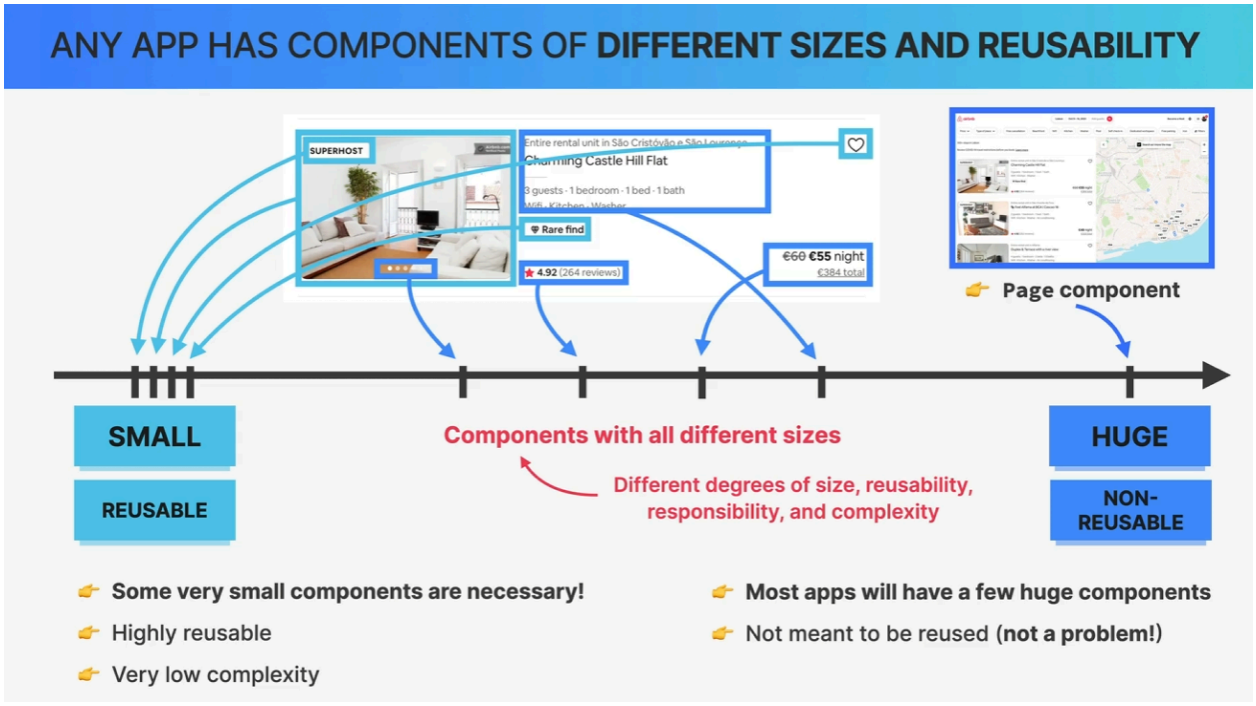
💰 Be aware that creating a new component **creates a new abstraction**. Abstractions have a **cost**, because **more abstractions require more mental energy** to switch back and forth between components. So try not to create new components too early

🏷️ Name a component according to **what it does** or **what it displays**. Don't be afraid of using long component names

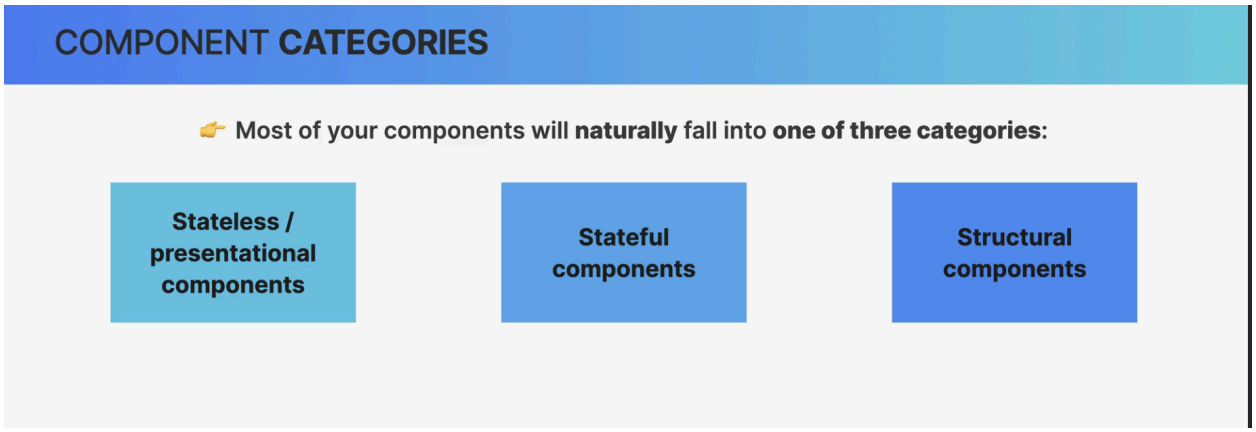
🪆 Never declare a new component **inside another component**!

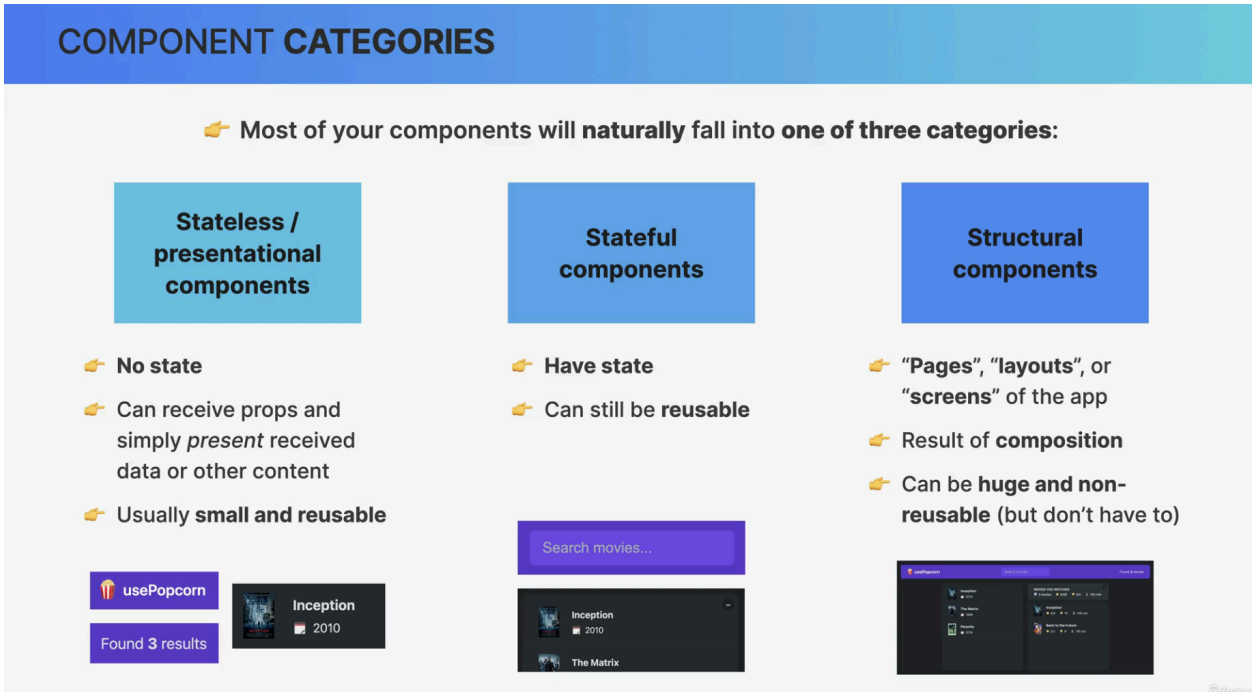
🗄️ **Co-locate related components inside the same file**. Don't separate components into different files too early

↔️ It's completely normal that an app has components of **many different sizes**, including very small and huge ones *(See next slide... 👉)*



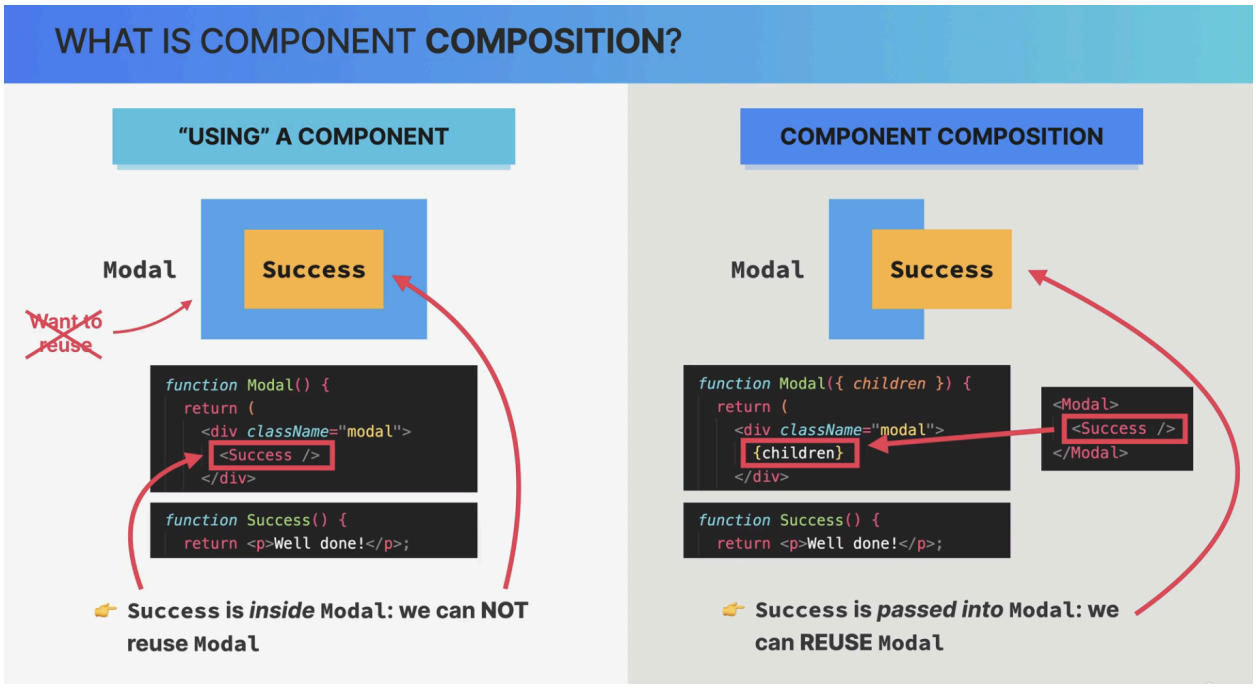
Component Categories





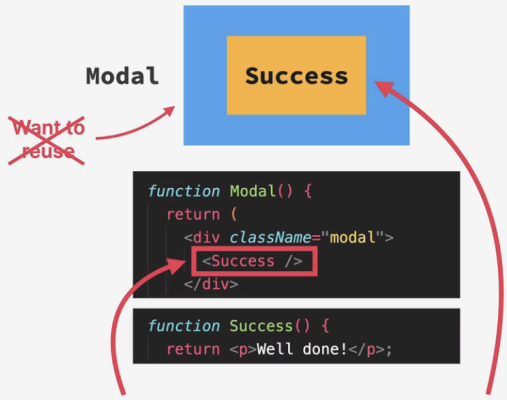
Component Composition

Combining different components using the children prop (or explicitly defined props)



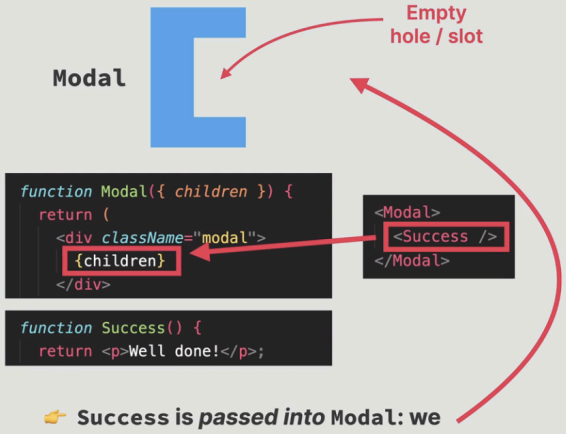
WHAT IS COMPONENT **COMPOSITION**?

"USING" A COMPONENT



👉 Success is *inside* `Modal`: we can NOT reuse `Modal`

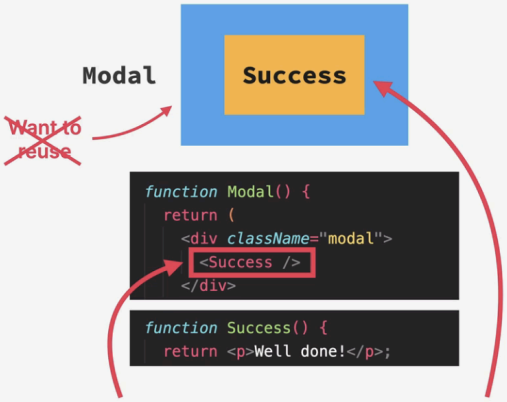
COMPONENT COMPOSITION



👉 Success is *passed into* `Modal`: we can REUSE `Modal`

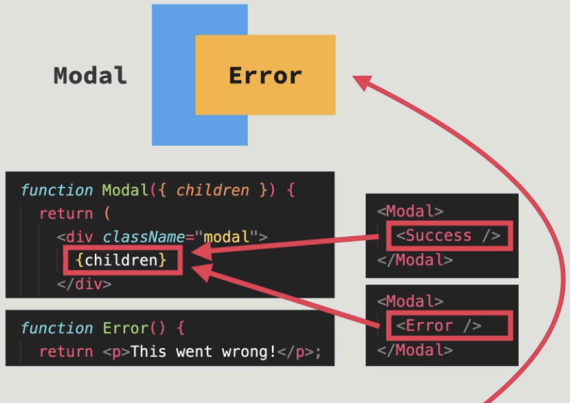
WHAT IS COMPONENT **COMPOSITION**?

"USING" A COMPONENT



👉 Success is *inside* `Modal`: we can NOT reuse `Modal`

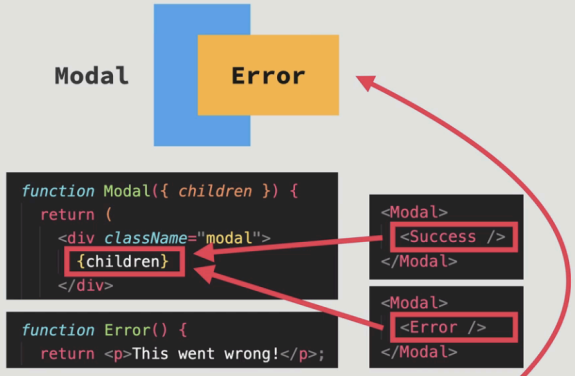
COMPONENT COMPOSITION



👉 Success is *passed into* `Modal`: we can REUSE `Modal`

WHAT IS COMPONENT **COMPOSITION?**

COMPONENT COMPOSITION

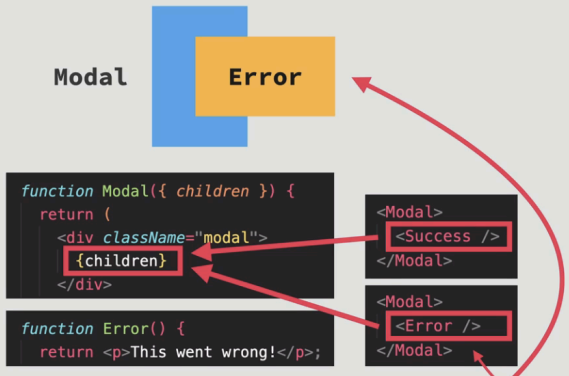


👉 **Success** is *passed into* `Modal`: we can **REUSE** `Modal`

👉 **Component composition**: combining different components using the `children` prop (or explicitly defined props)

WHAT IS COMPONENT **COMPOSITION?**

COMPONENT COMPOSITION



👉 **Success** is *passed into* `Modal`: we can **REUSE** `Modal`

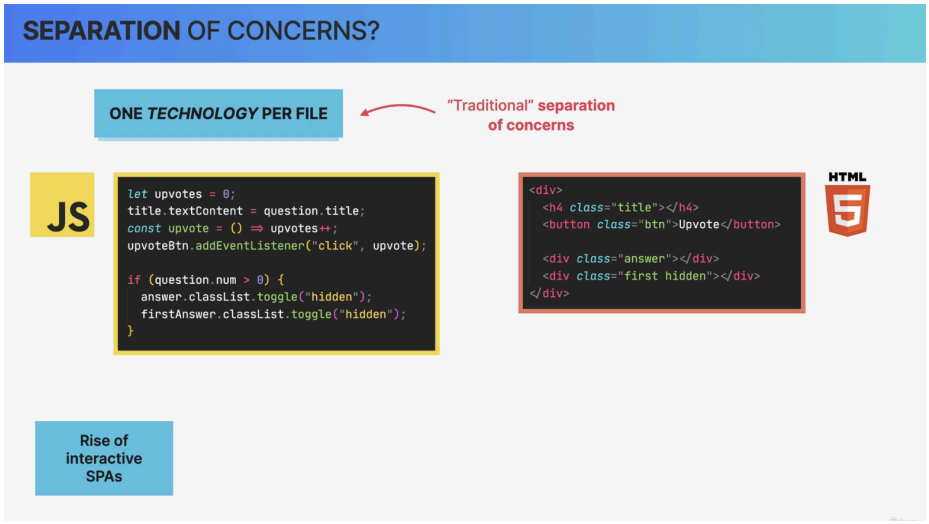
👉 **Component composition**: combining different components using the `children` prop (or explicitly defined props)

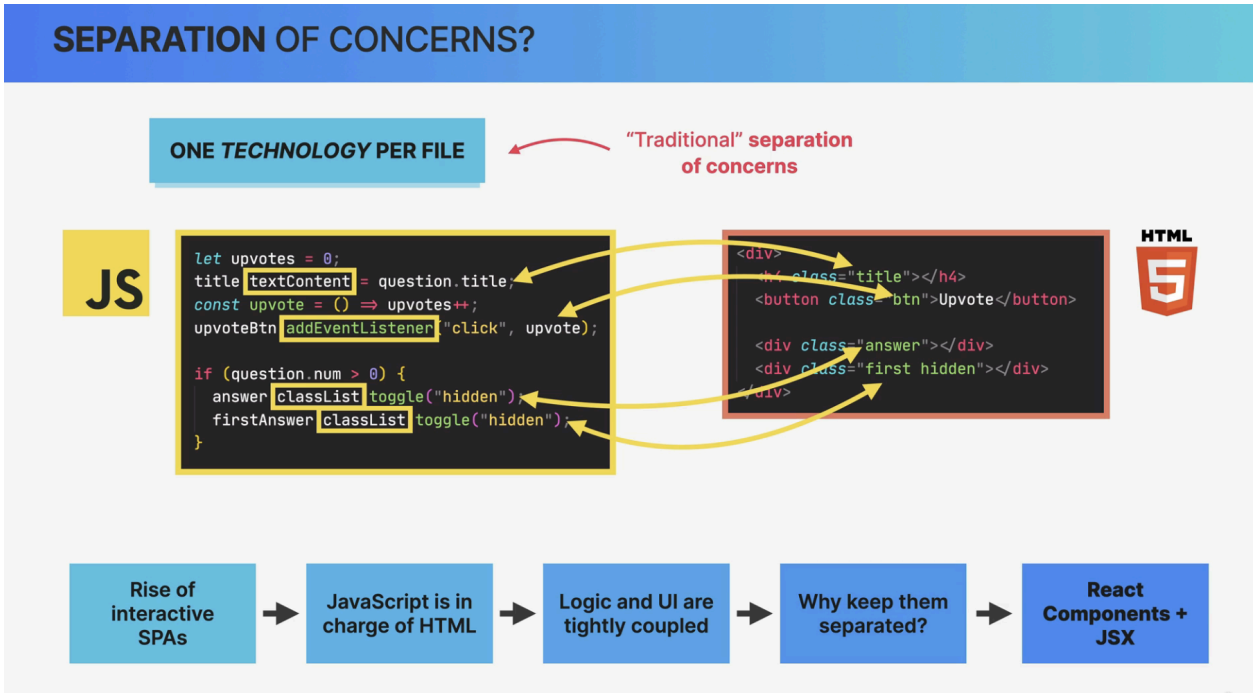
WE COMPONENT COMPOSITION, WE CAN:

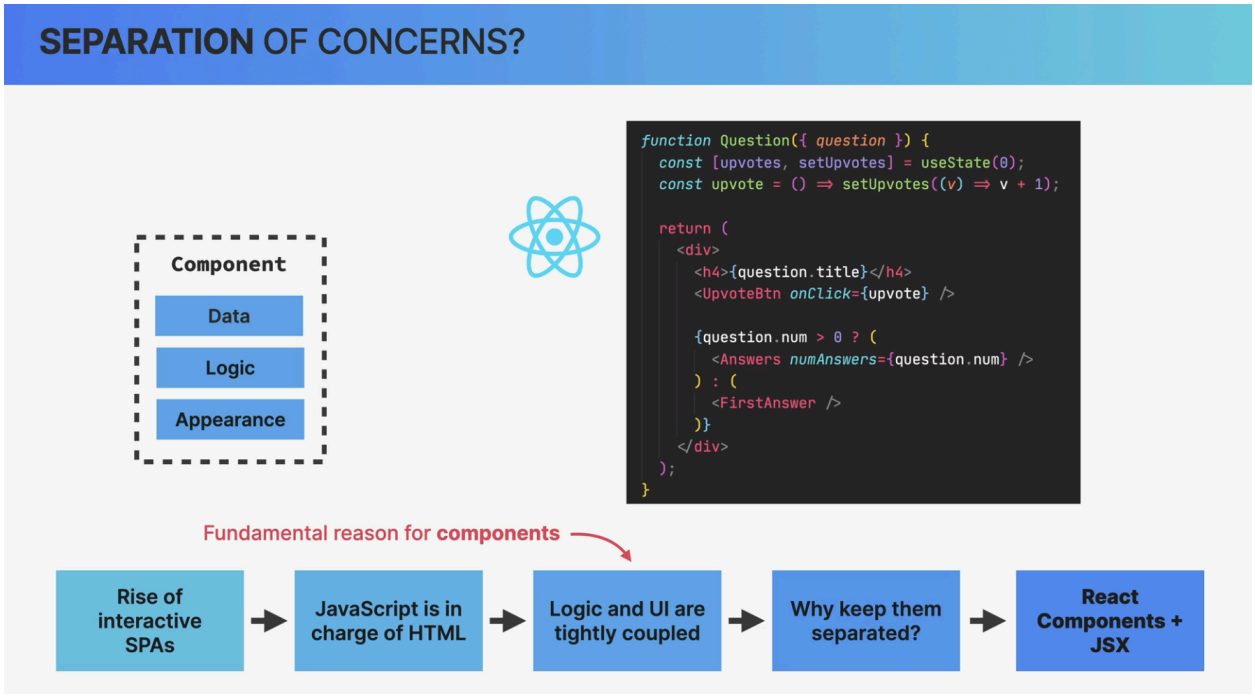
1. Create highly reusable and flexible components
2. Fix prop drilling (great for layouts)

Possible because components don't need to know their children in advance

Separation of concerns

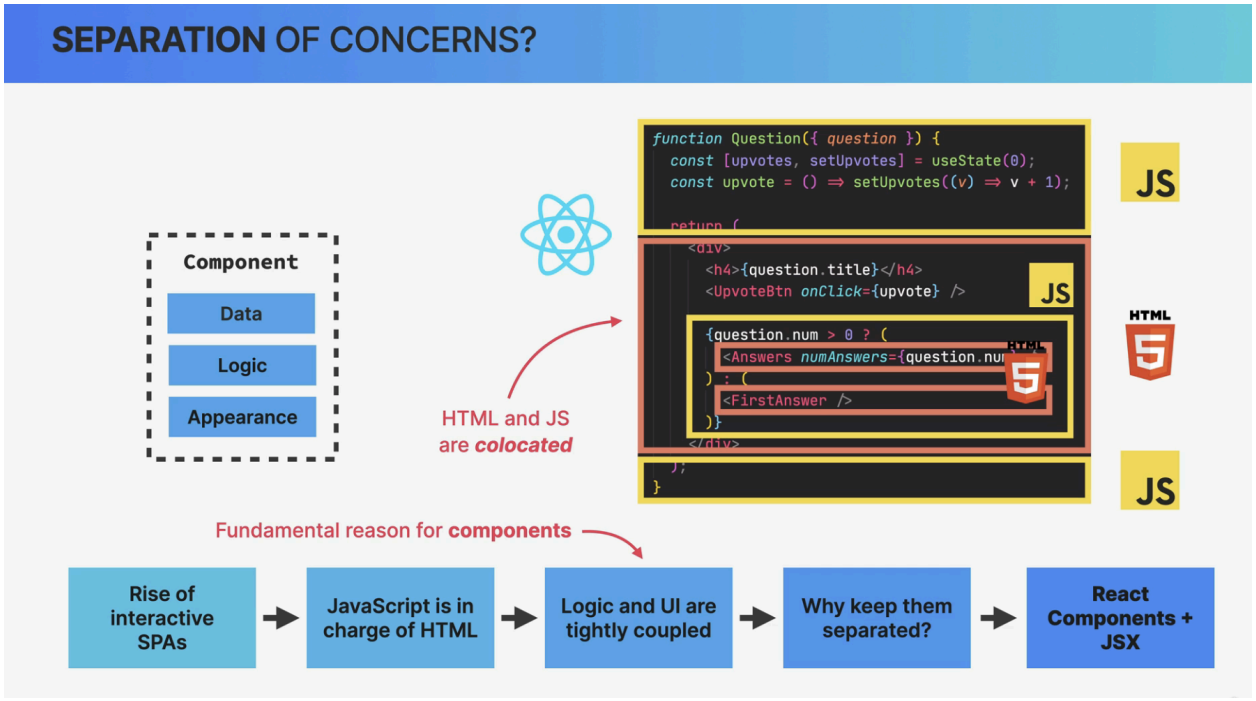






So content and logic are tightly coupled together and so it makes sense that they are co-located here.

Co-located simply means that things that change together should be located as close as possible together.
And in the case of React apps, that means that instead of one technology profile, we have one component profile. So one component that contains data logic and appearance,
all mixed together.



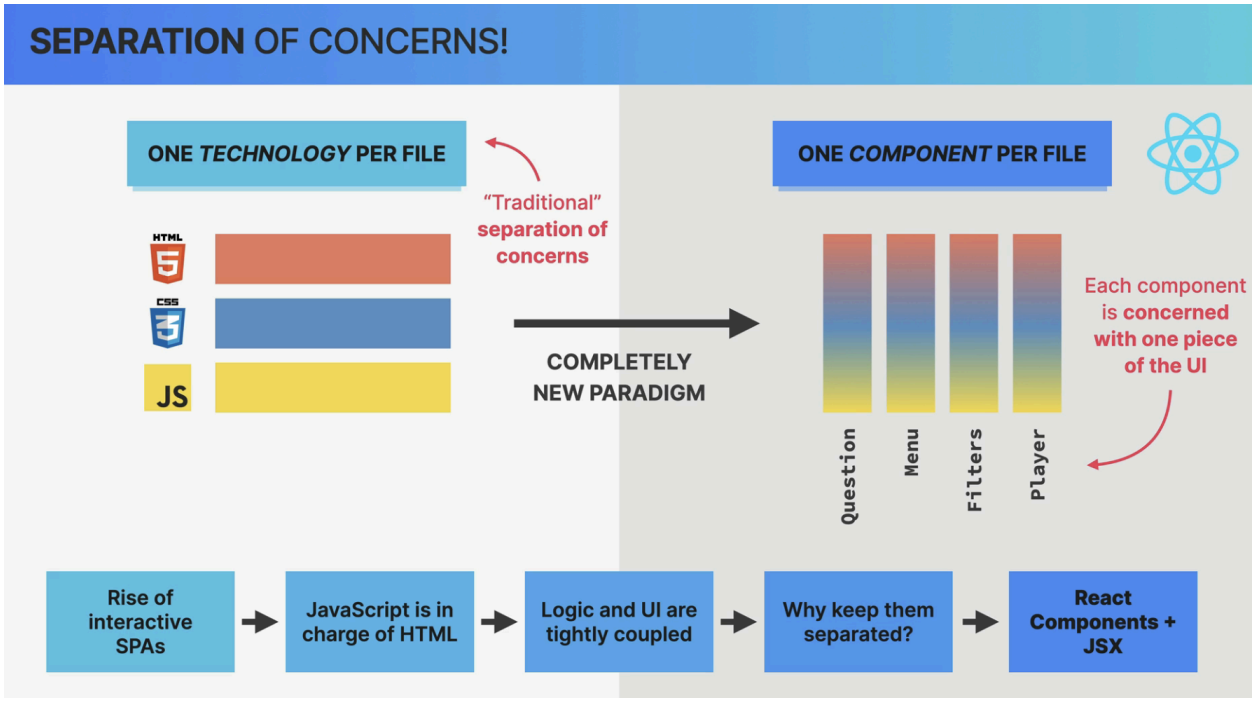
Well, I think that the people who say that React has no separation of concerns, got it all wrong.

Because React does actually have a separation of concerns. It's just not one concern per file, as we had traditionally but one concern per component.

So each component is in fact, only concerned with one piece of the UI.

Then within each of these components, of course we still have the three concerns of HTML, CSS, and JavaScript all mixed up, as we have been discussing.

So compared to the traditional separation of concerns, this is a completely new paradigm that many people were really not used to in the beginning. But now, many years later, we all got used to this and it works just great.



State

And the state is the most fundamental concept of React. So whenever we need something to change in the user interface, we change the state. So we update something that we call state.

📚 WHAT WE NEED TO LEARN

WHAT REACT DEVELOPERS NEED TO LEARN ABOUT STATE:

1. **What is state** and **why** do we need it? — This section
2. How to use state in **practice**?
   - 👉 `useState`
   - 👉 `useReducer`
   - 👉 Context API
3. **Thinking** about state
   - 👉 When to use state
   - 👉 Where to place state
   - 👉 Types of state

Rest of the course...

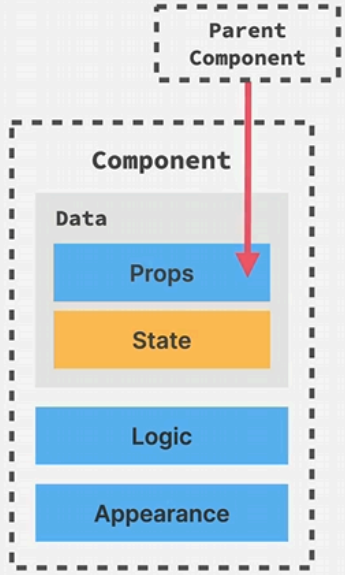
👋 **State** is the most important concept in React

(So we will keep learning about state throughout the entire course...)

WHAT IS **STATE**?

STATE

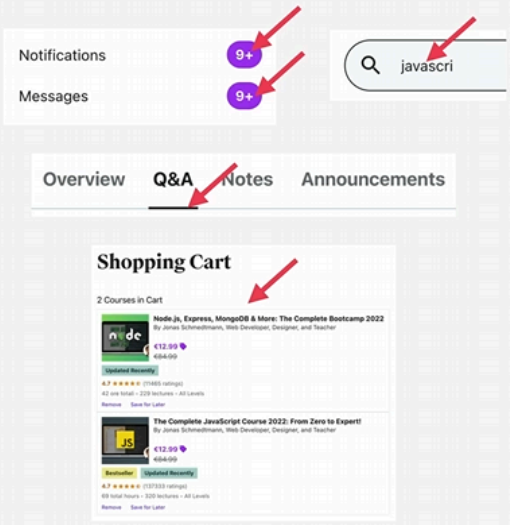
- 👉 Data that a component **can hold over time**, necessary for information that it needs to **remember** throughout the app's lifecycle
- 👉 **"Component's memory"** 🧠



WHAT IS **STATE**?

STATE

- Data that a component **can hold over time**, necessary for information that it needs to **remember** throughout the app's lifecycle
- **"Component's memory"**



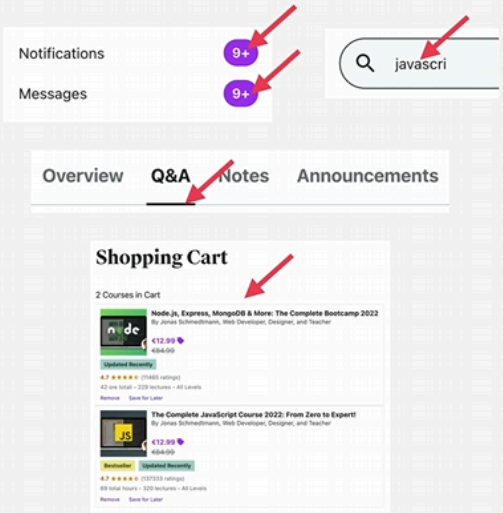
WHAT IS **STATE**?

STATE

- Data that a component **can hold over time**, necessary for information that it needs to **remember** throughout the app's lifecycle
- **"Component's memory"**

- **"State variable" / "piece of state"**: A single variable in a component (component state)

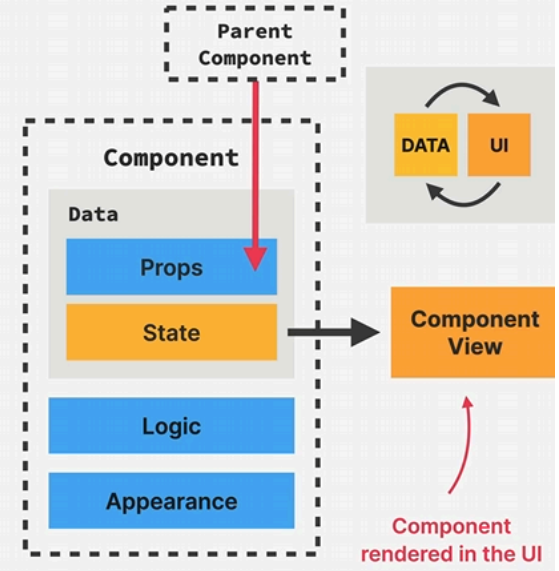
We use these terms interchangeably



WHAT IS **STATE**?

STATE

- Data that a component **can hold over time**, necessary for information that it needs to **remember** throughout the app's lifecycle
- **"Component's memory"** 🧠
- ***Component* state:** Single local component variable ("Piece of state", "state variable")
- Updating **component state** triggers React to **re-render the component**



WHAT IS **STATE**?

STATE

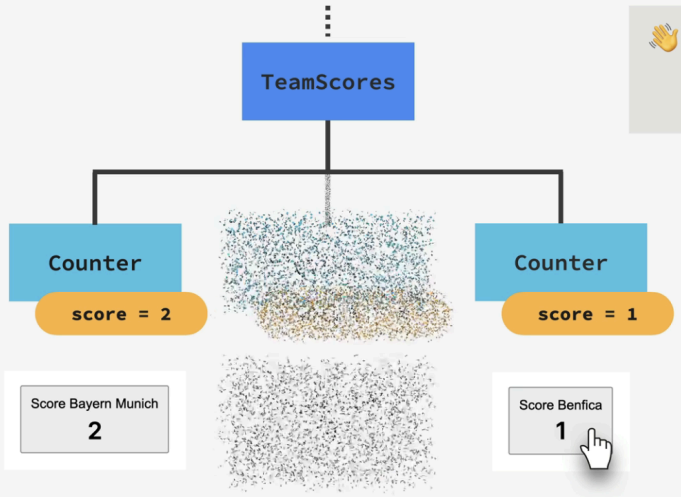
- Data that a component **can hold over time**, necessary for information that it needs to **remember** throughout the app's lifecycle
- **"Component's memory"** 🧠
- ***Component* state:** Single local component variable ("Piece of state", "state variable")
- Updating **component state** triggers React to **re-render the component**

STATE ALLOWS DEVELOPERS TO:

1. Update the component's view (by re-rendering it)
2. Persist local variables between renders

👋 *State is a **tool**. Mastering state will unlock the power of React development*

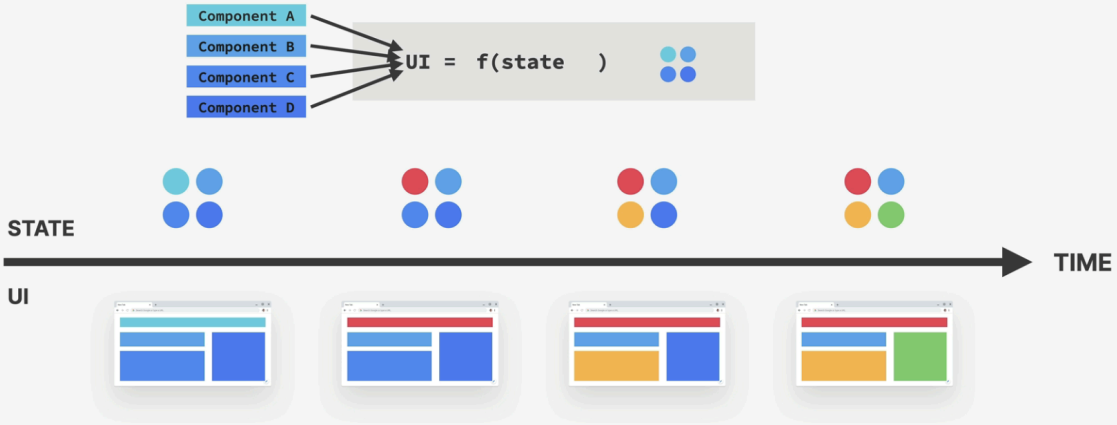
ONE COMPONENT, ONE **STATE**



👋 Each component has and manages **its own state**, no matter how many times we render the same component

Multiple *instances* of the `Counter` component

UI AS A **FUNCTION** OF STATE



DECLARATIVE, *REVISITED*

- 👉 With state, we view UI as a **reflection of data changing over time**
- 👉 We ***describe* that reflection** of data using state, event handlers, and JSX

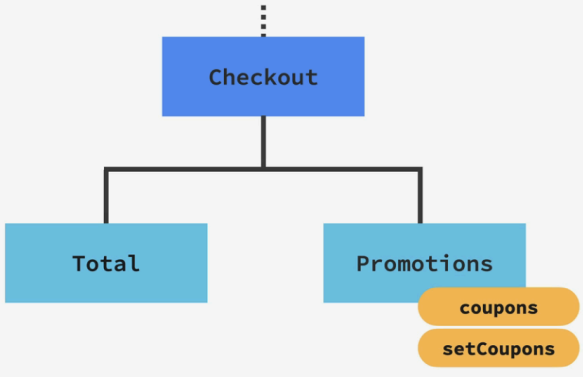
🎯 IN **PRACTICAL** TERMS...

PRACTICAL GUIDELINES ABOUT STATE

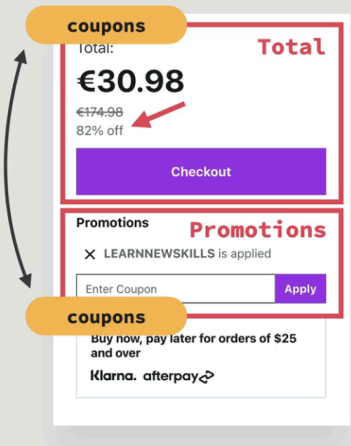
- 👉 Use a state variable for any data that the component should keep track of ("remember") over time. **This is data that will change at some point**. In Vanilla JS, that's a `let` variable, or an `[]` or `{}`
- 👉 Whenever you want something in the component to be **dynamic**, create a piece of state related to that "thing", and update the state when the "thing" should change (aka "be dynamic")
  - 👉 ***Example:*** *A modal window can be open or closed. So we create a state variable `isOpen` that tracks whether the modal is open or not. On `isOpen = true` we display the window, on `isOpen = false` we hide it.*
- 👉 If you want to change the way a component looks, or the data it displays, **update its state.** This usually happens in an **event handler** function.
- 👉 When building a component, imagine its view as a **reflection of state changing over time**
- 👉 For data that should not trigger component re-renders, **don't use state**. Use a regular variable instead. This is a common **beginner mistake**.

Lifting The State Up

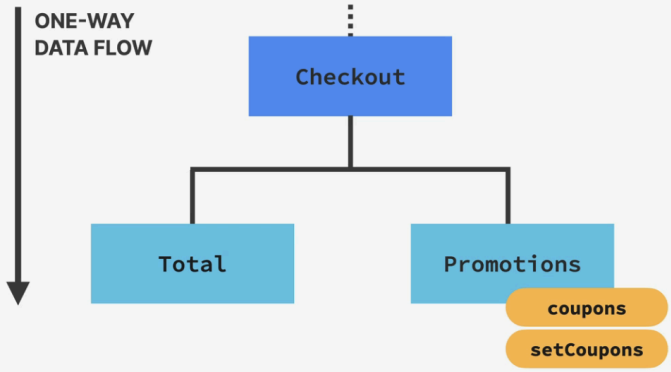
PROBLEM: SHARING STATE WITH SIBLING COMPONENT



👉 `Total` component also needs access to coupons state

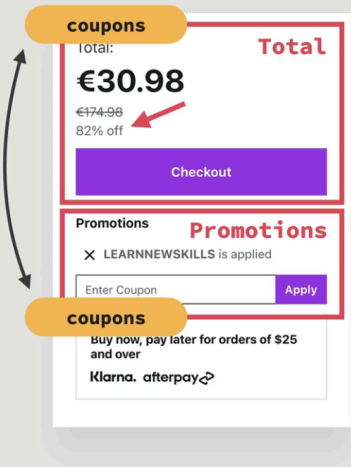


PROBLEM: SHARING STATE WITH SIBLING COMPONENT

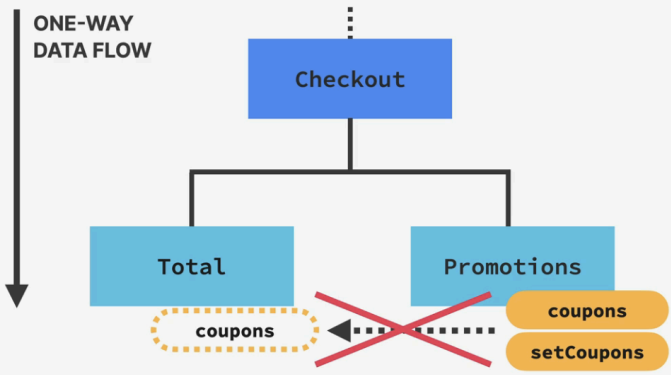


Data can only flow **down to children** (via props), not **sideways to siblings**

👉 `Total` component also needs access to coupons state



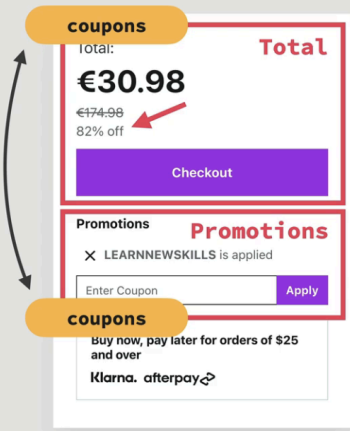
PROBLEM: SHARING STATE WITH SIBLING COMPONENT



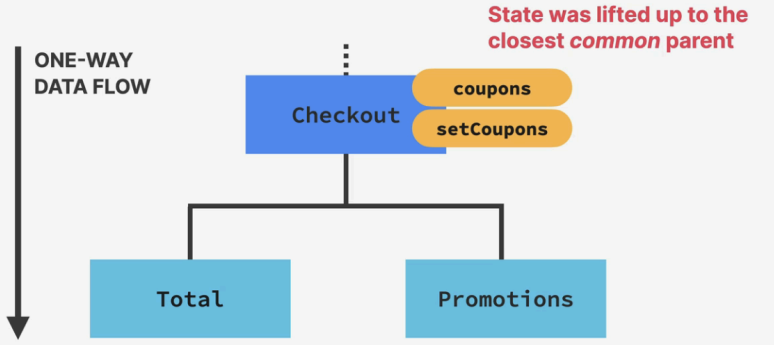
Data can only flow **down to children** (via props), not **sideways to siblings**

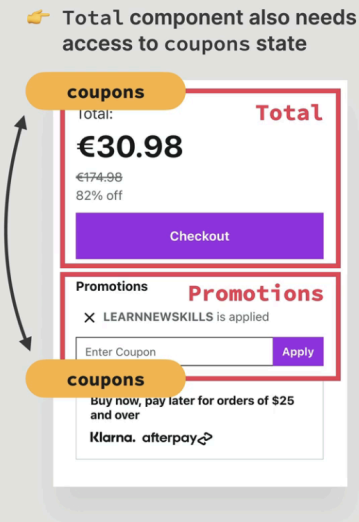
🤔 How do we share state with **other components**?

👉 `Total` component also needs access to coupons state

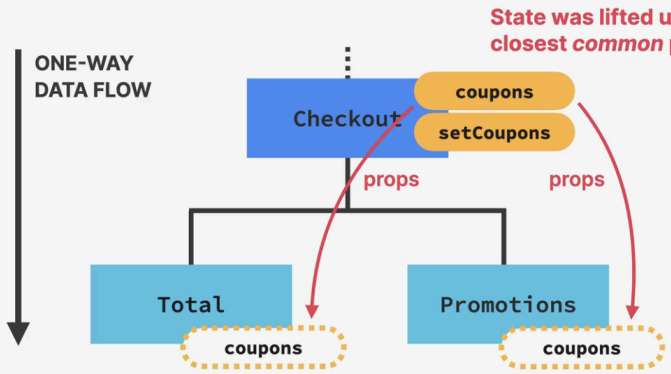


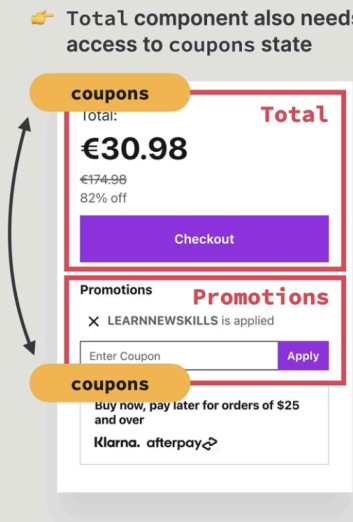
SOLUTION: LIFTING STATE UP



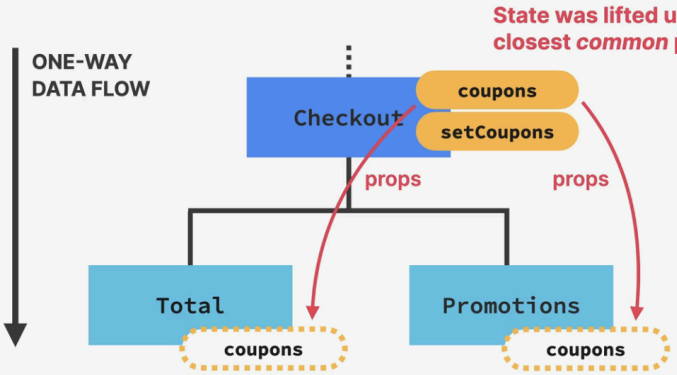


SOLUTION: LIFTING STATE UP



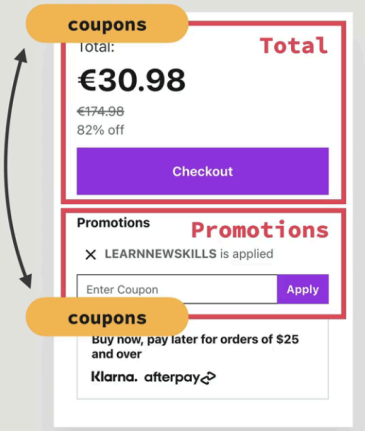


SOLUTION: LIFTING STATE UP

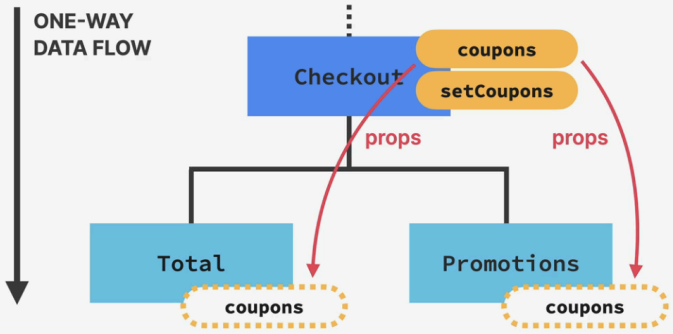


✌️ By lifting state up, we have successfully shared one piece of state with multiple components in different positions in the component tree

👉 `Total` component also needs access to coupons state

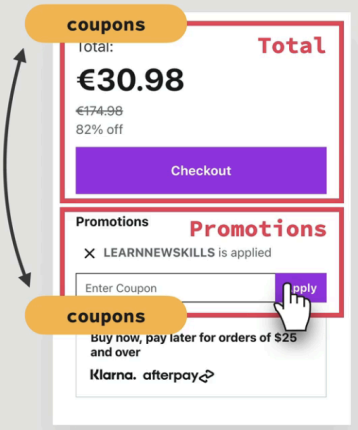


CHILD-TO-PARENT COMMUNICATION

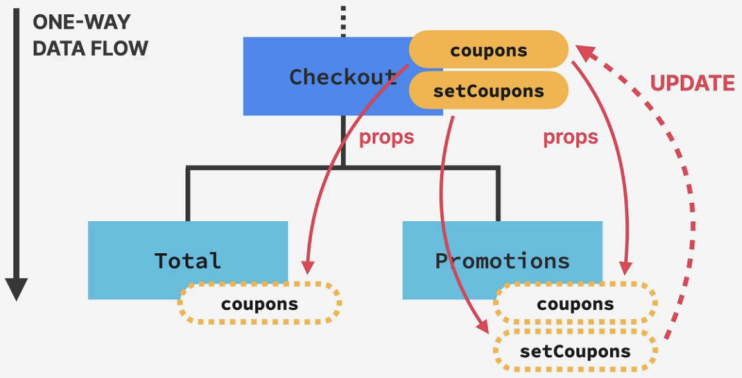


🤔 If data flows from parent to children, how can `Promotions` (child) update state in `Checkout` (parent)?

👉 `Total` component also needs access to coupons state

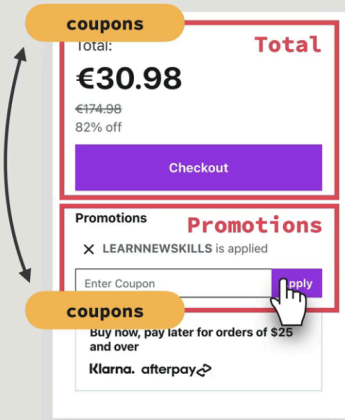


CHILD-TO-PARENT COMMUNICATION



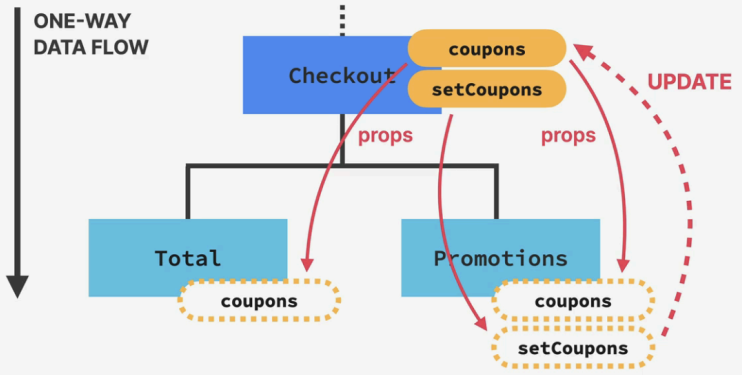
🤔 If data flows from parent to children, how can `Promotions` (child) update state in `Checkout` (parent)?

👉 `Total` component also needs access to coupons state



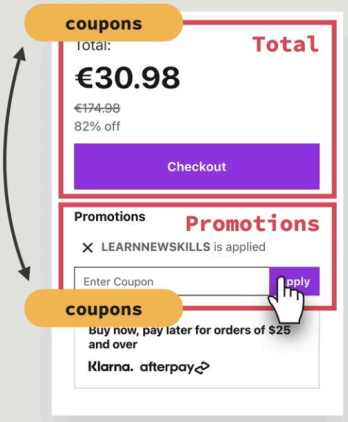
CHILD-TO-PARENT COMMUNICATION

👉 **Child-to-parent communication (inverse data flow):** child updating parent state (data "flowing" **up**)



🤔 If data flows from parent to children, how can `Promotions` (child) update state in `Checkout` (parent)?

👉 `Total` component also needs access to coupons state



Derived state

DERIVING STATE

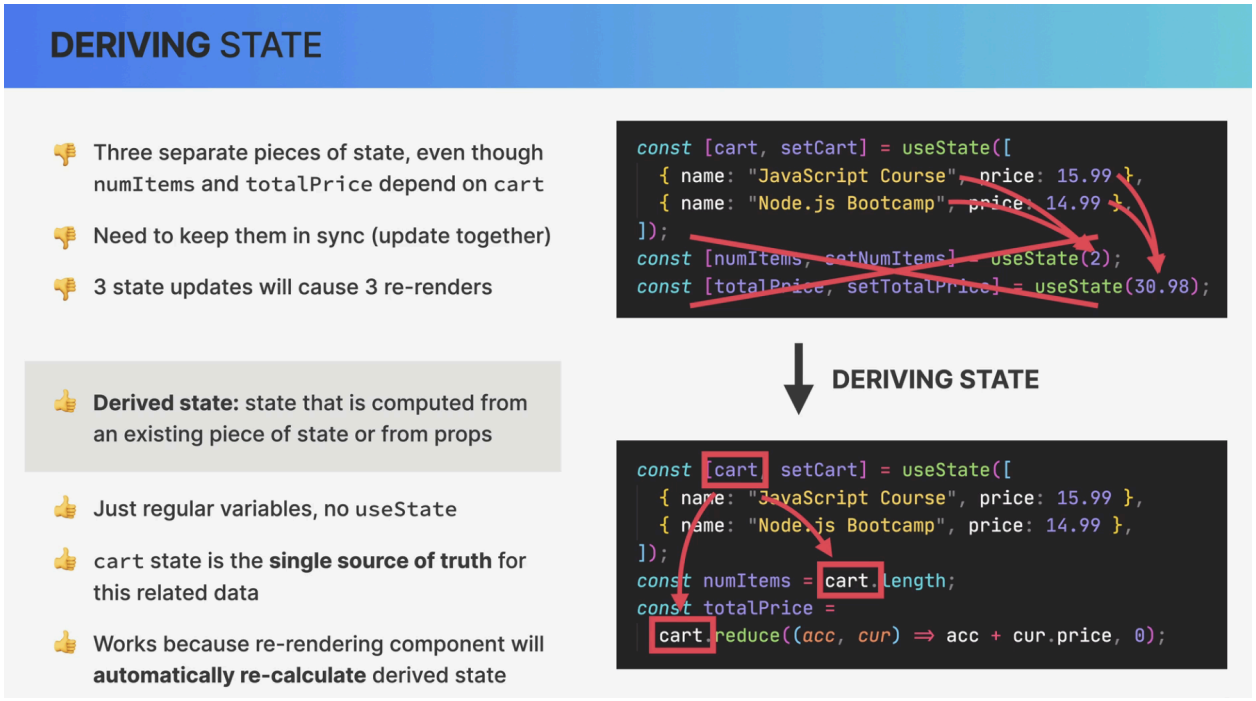
👍 **Derived state:** state that is computed from an existing piece of state or from props

DERIVING STATE

👎 Three separate pieces of state, even though `numItems` and `totalPrice` depend on `cart`

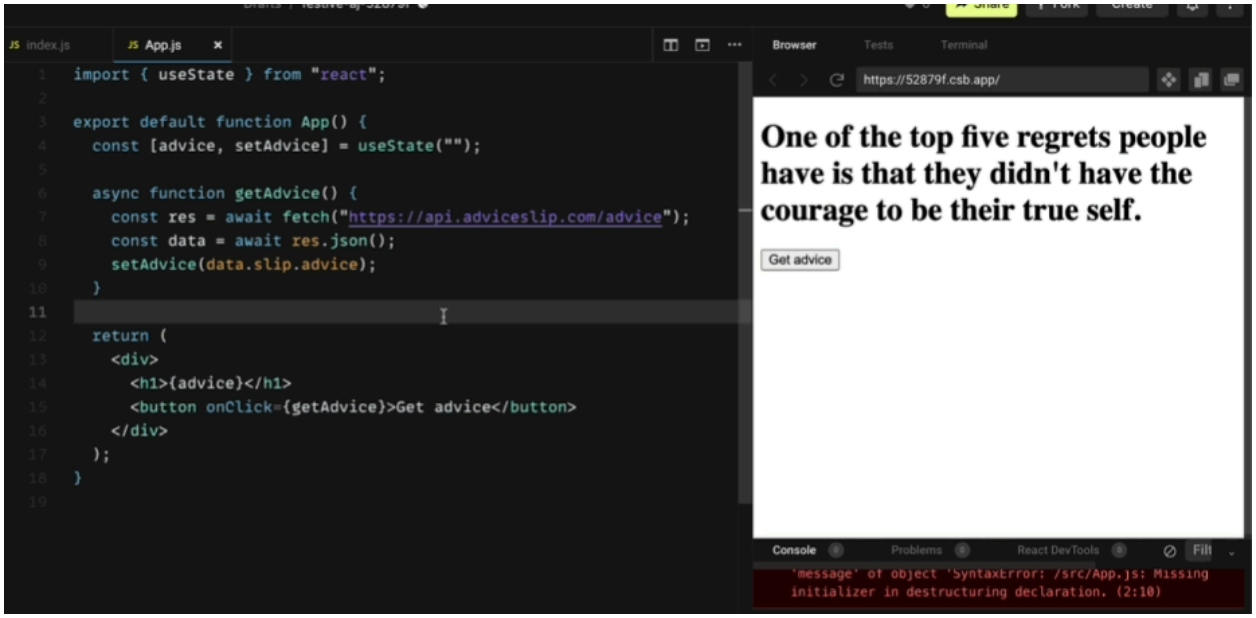
```
const [cart, setCart] = useState([
  { name: "JavaScript Course", price: 15.99 },
  { name: "Node.js Bootcamp", price: 14.99 },
]);
const [numItems, setNumItems] = useState(2);
const [totalPrice, setTotalPrice] = useState(30.98);
```

👍 **Derived state:** state that is computed from an existing piece of state or from props



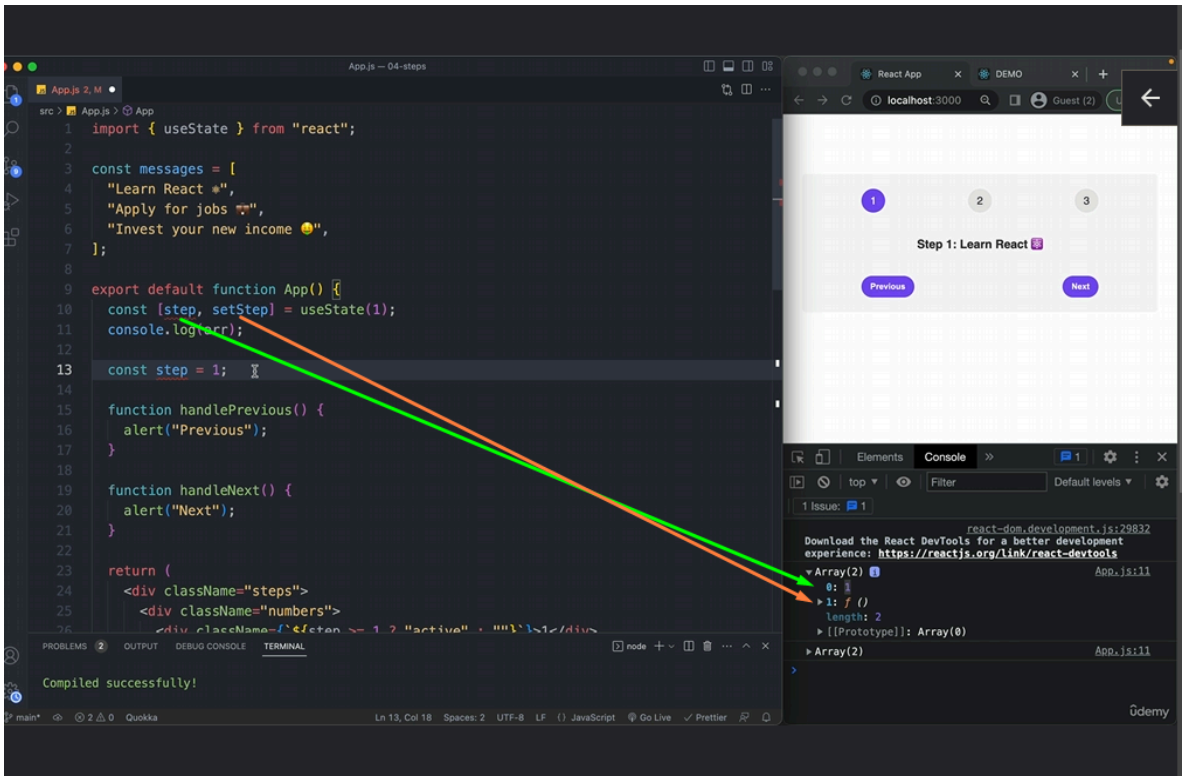
useState

So this useState here is a React function that returns an array.
And so here, we are destructuring that array.
So in the first position of the array, we have the value of the state that we call advice here.
The second value is a setter function.
So a function that we can use to update the piece of state.

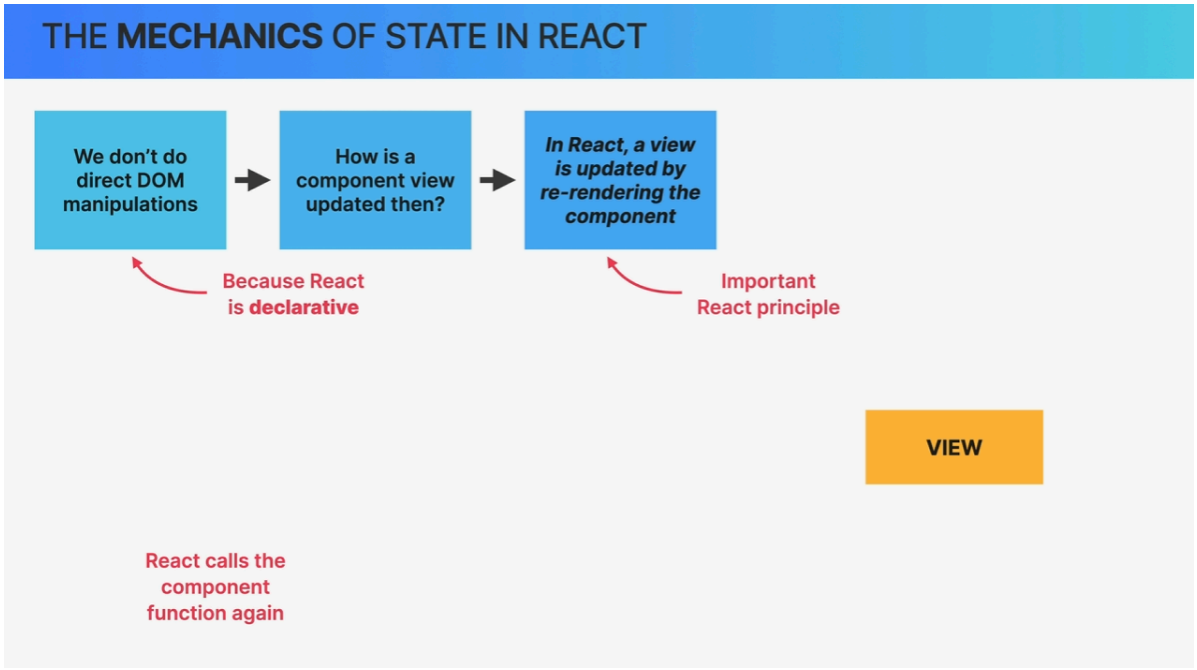

index.js
App.js
import { useState } from "react";
export default function App() {
const [advice, setAdvice] = useState("");
async function getAdvice() {
const res = await fetch("https://api.adviceslip.com/advice");
const data = await res.json();
setAdvice(data.slip.advice);
}
return (

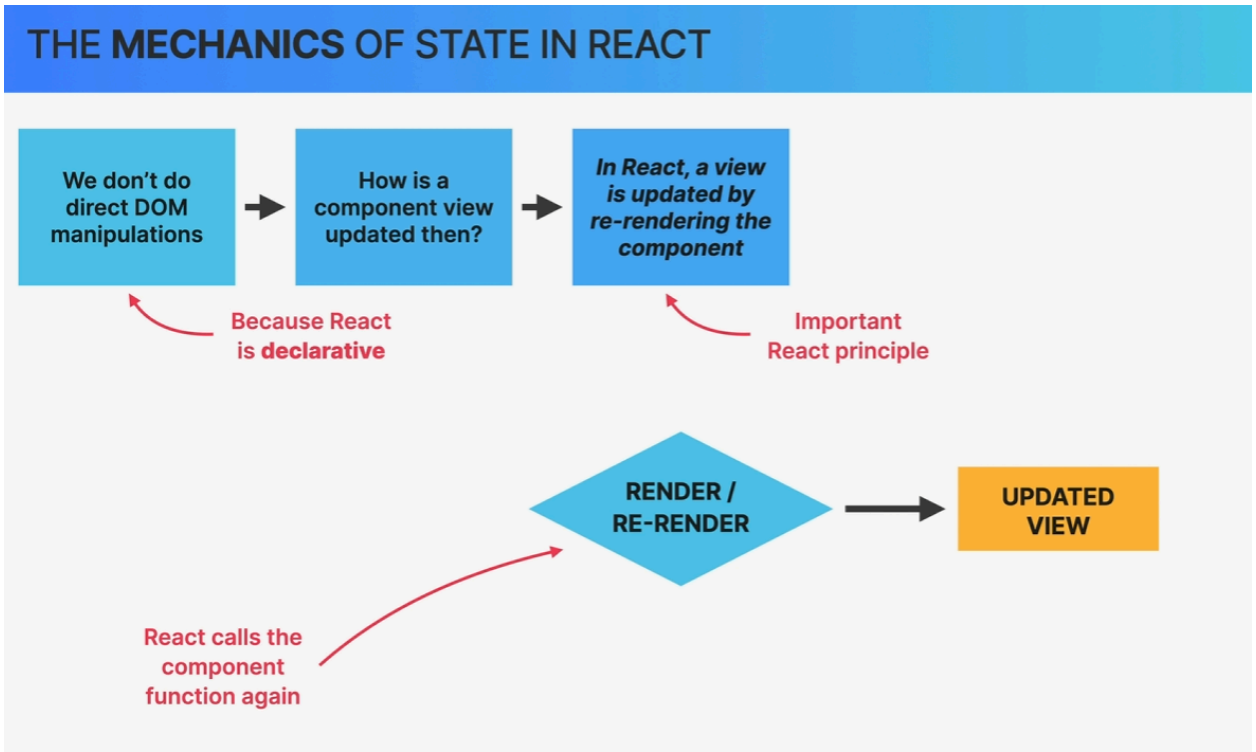
<h1>{advice}</h1>
<button onClick={getAdvice}>Get advice</button>

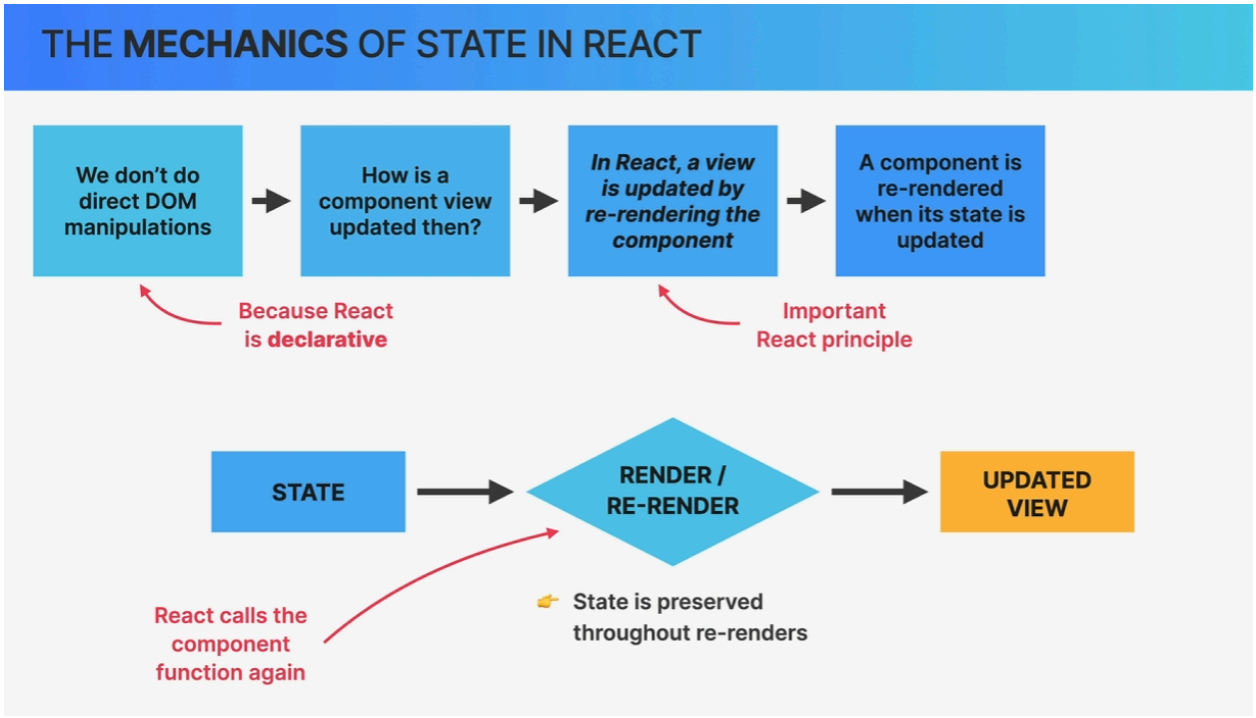
);
}
Browser
Tests
Terminal
https://52879f.csb.app/
One of the top five regrets people have is that they didn't have the courage to be their true self.
Get advice
Console
Problems
React DevTools
'message' of object 'SyntaxError: /src/App.js: Missing initializer in destructuring declaration. (2:10)

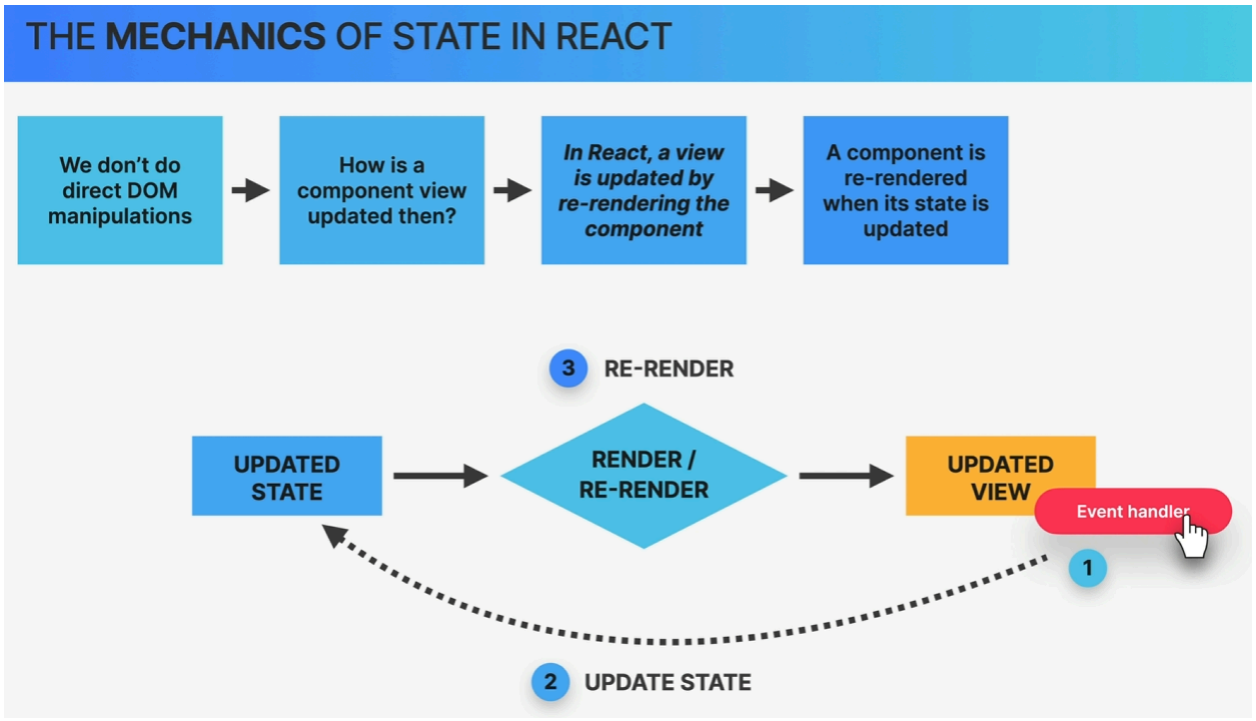

App.js — 04-steps
App.js 2, M
import { useState } from "react";
const messages = [
"Learn React",
"Apply for jobs",
"Invest your new income",
];
export default function App() {
const [step, setStep] = useState(1);
console.log(arr);
const step = 1;
function handlePrevious() {
alert("Previous");
}
function handleNext() {
alert("Next");
}
return (

PROBLEMS 2
OUTPUT
DEBUG CONSOLE
TERMINAL
Compiled successfully!
React App
DEMO
localhost:3000
Guest (2)
1
2
3
Step 1: Learn React
Previous
Next
Elements
Console
Filter
Default levels
1 Issue: 1
react-dom.development.js:29832
Download the React DevTools for a better development experience: https://reactjs.org/link/react-devtools
Array(2)
App.js:11
0: 1
1: ƒ ()
length: 2
[[Prototype]]: Array(0)
Array(2)
App.js:11
Udemy


THE MECHANICS OF STATE IN REACT
We don't do direct DOM manipulations
How is a component view updated then?
In React, a view is updated by re-rendering the component
Because React is declarative
Important React principle
VIEW
React calls the component function again

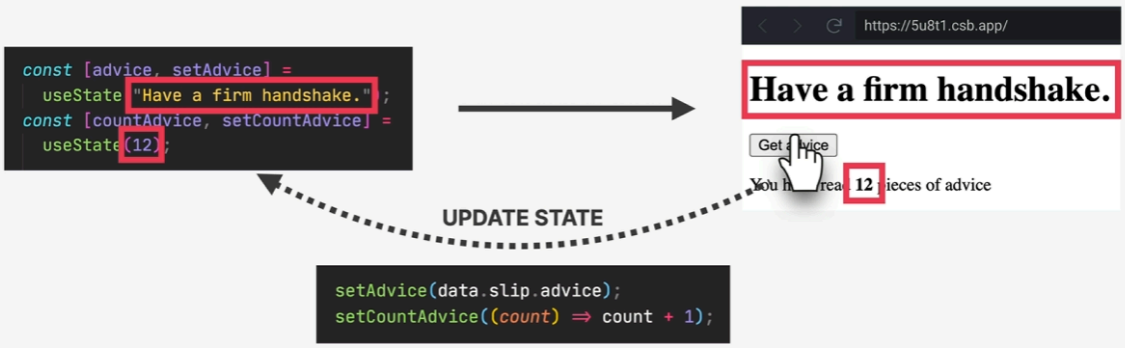

THE MECHANICS OF STATE IN REACT
We don't do direct DOM manipulations
How is a component view updated then?
In React, a view is updated by re-rendering the component
Because React is declarative
Important React principle
RENDER / RE-RENDER
UPDATED VIEW
React calls the component function again


THE MECHANICS OF STATE IN REACT
We don't do direct DOM manipulations
How is a component view updated then?
In React, a view is updated by re-rendering the component
A component is re-rendered when its state is updated
Because React is declarative
Important React principle
STATE
RENDER / RE-RENDER
UPDATED VIEW
React calls the component function again
State is preserved throughout re-renders


THE MECHANICS OF STATE IN REACT
We don't do direct DOM manipulations
How is a component view updated then?
In React, a view is updated by re-rendering the component
A component is re-rendered when its state is updated
3 RE-RENDER
UPDATED STATE
RENDER / RE-RENDER
UPDATED VIEW
Event handler
1
2 UPDATE STATE

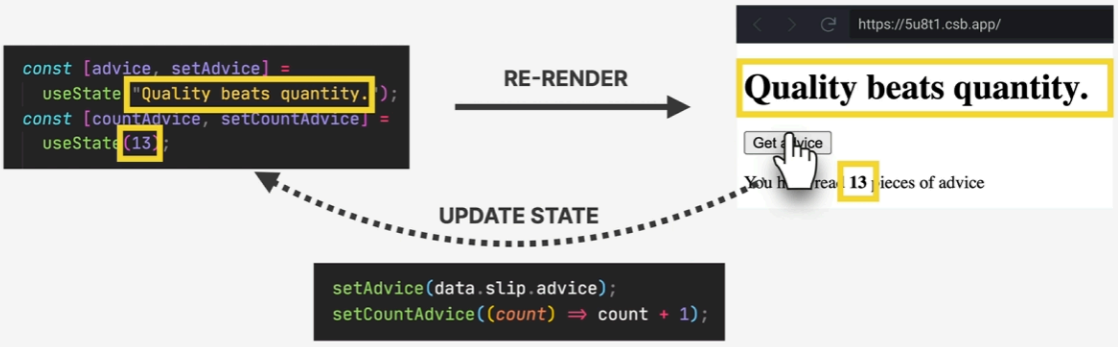
THE **MECHANICS** OF STATE IN REACT

We don't do direct DOM manipulations → How is a component view updated then? → *In React, a view is updated by re-rendering the component* → A component is re-rendered when its state is updated



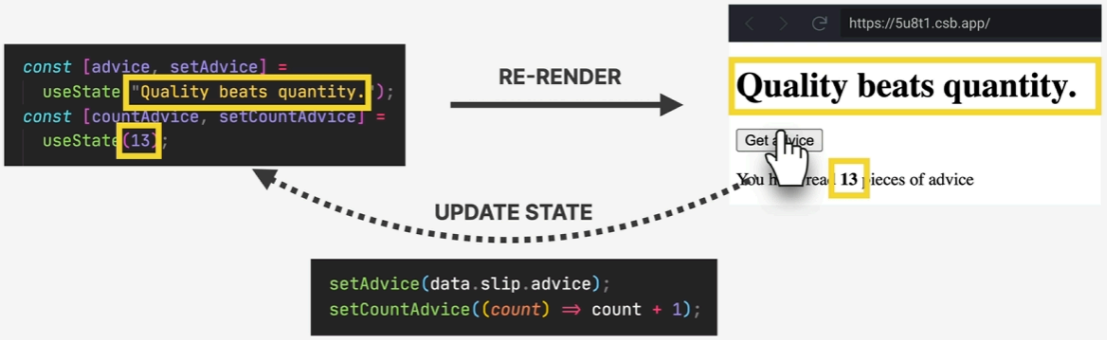
THE **MECHANICS** OF STATE IN REACT

We don't do direct DOM manipulations → How is a component view updated then? → *In React, a view is updated by re-rendering the component* → A component is re-rendered when its state is updated



THE **MECHANICS** OF STATE IN REACT

We don't do direct DOM manipulations → How is a component view updated then? → *In React, a view is updated by re-rendering the component* → A component is re-rendered when its state is updated → **So to update a view, we update state**



THE **MECHANICS** OF STATE IN REACT

We don't do direct DOM manipulations → How is a component view updated then? → *In React, a view is updated by re-rendering the component* → A component is re-rendered when its state is updated → **So to update a view, we update state**

👉 *React is called "React" because...*

REACT *REACTS* TO STATE CHANGES BY RE-RENDERING THE UI

useEffect

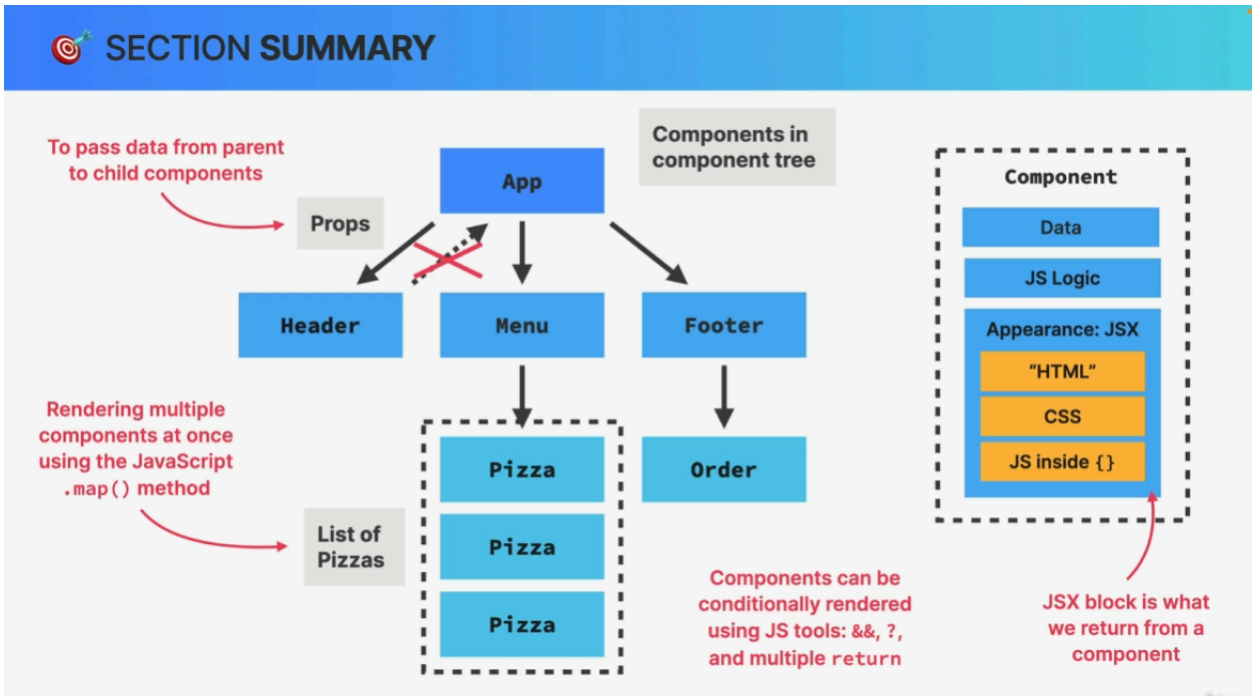
So useEffect takes two arguments. First, a function that we want to get executed at the beginning. So when this component first gets loaded. And then a second argument, which is called the dependency array.

Build our first react app

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

function App() {
  const [advice, setAdvice] = useState("");
  const [count, setCount] = useState(0);


  async function getAdvice() {
    const res = await fetch("https://api.adviceslip.com/advice");
    const data = await res.json();
    setAdvice(data.slip.advice);
    setCount((c) => c + 1);
  }

  useEffect(function () {
    getAdvice();
  }, []);

  return (
    <div>
      <h1>{advice}</h1>
      <button onClick={getAdvice}>Get advice</button>
      <Message count={count} />
    </div>
  );
}

function Message(props) {
```

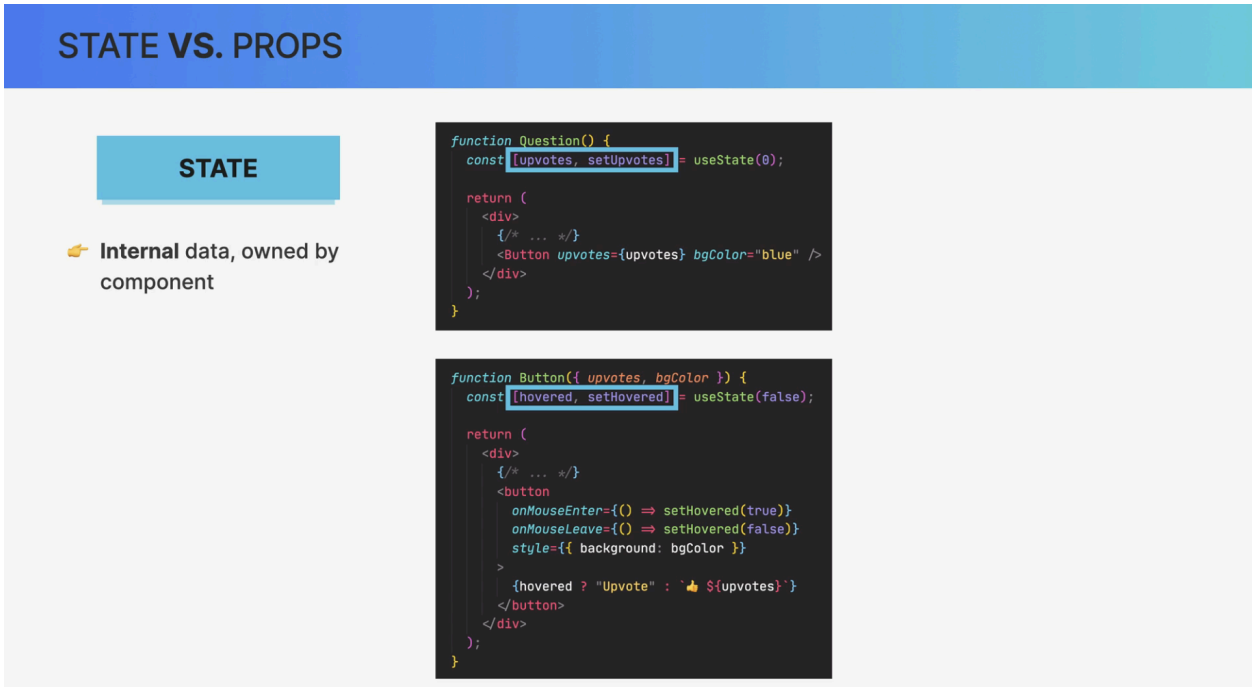
```
  return (
    <p>
      You have read <strong>{props.count}</strong> pieces of advice
    </p>
  );
}

export default App;
```

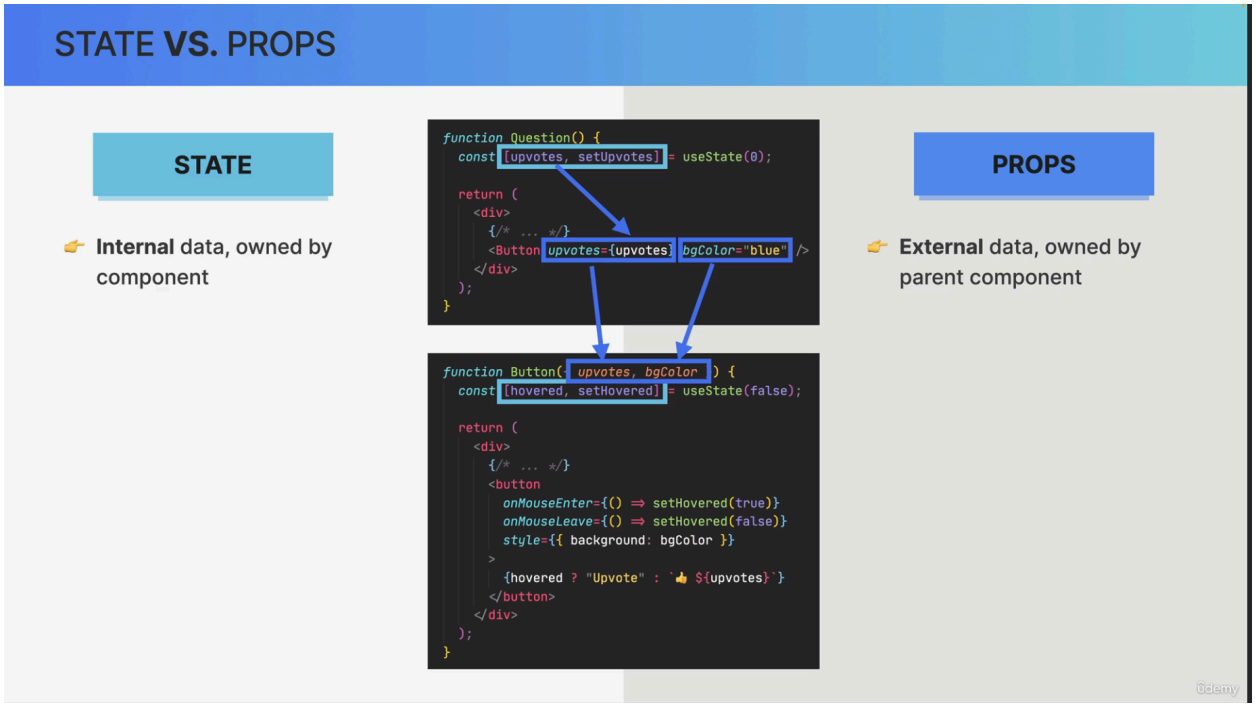
Summary



Section 6

Summary

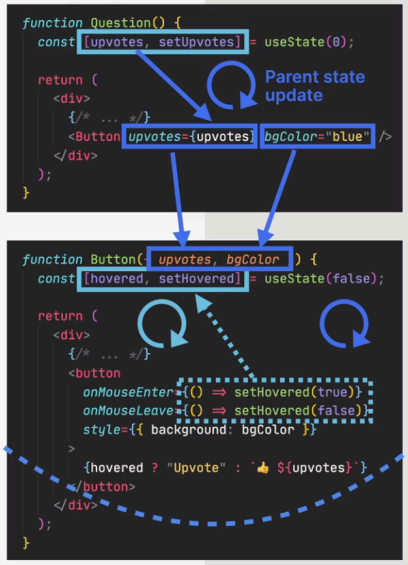




STATE **VS.** PROPS

STATE

- **Internal** data, owned by component
- Component “memory”
- Can be updated by the component itself
- Updating state causes component to re-render
- Used to make components interactive

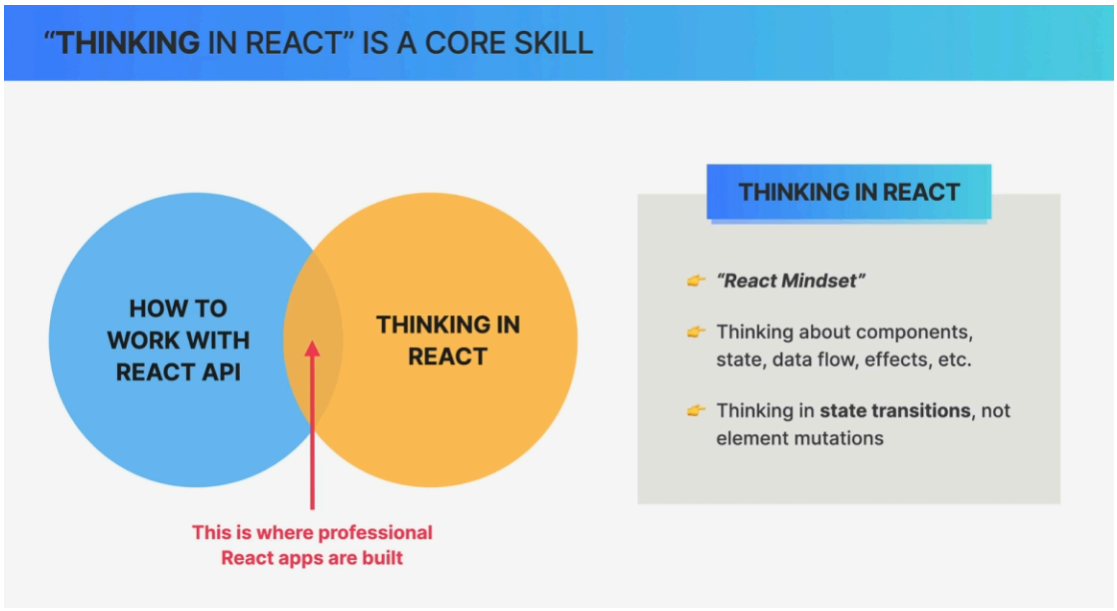


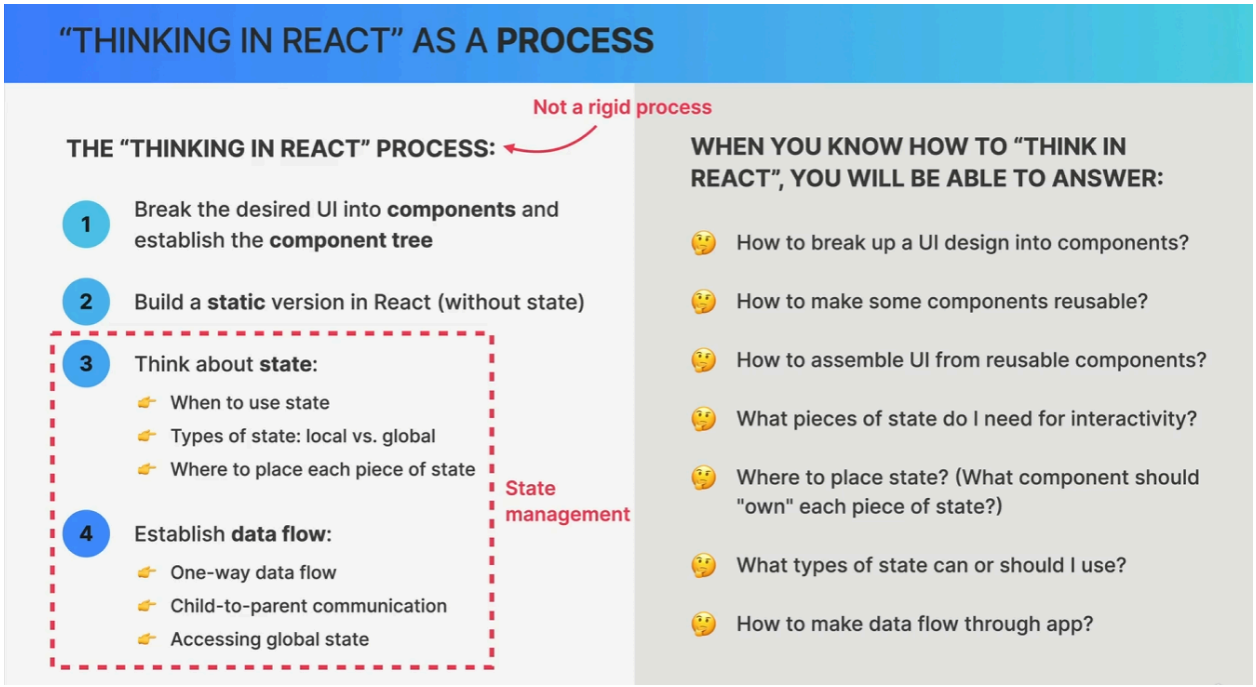
PROPS

- **External** data, owned by parent component
- Similar to function parameters
- Read-only
- **Receiving new props causes component to re-render.** Usually when the parent’s state has been updated
- Used by parent to configure child component (“settings”)

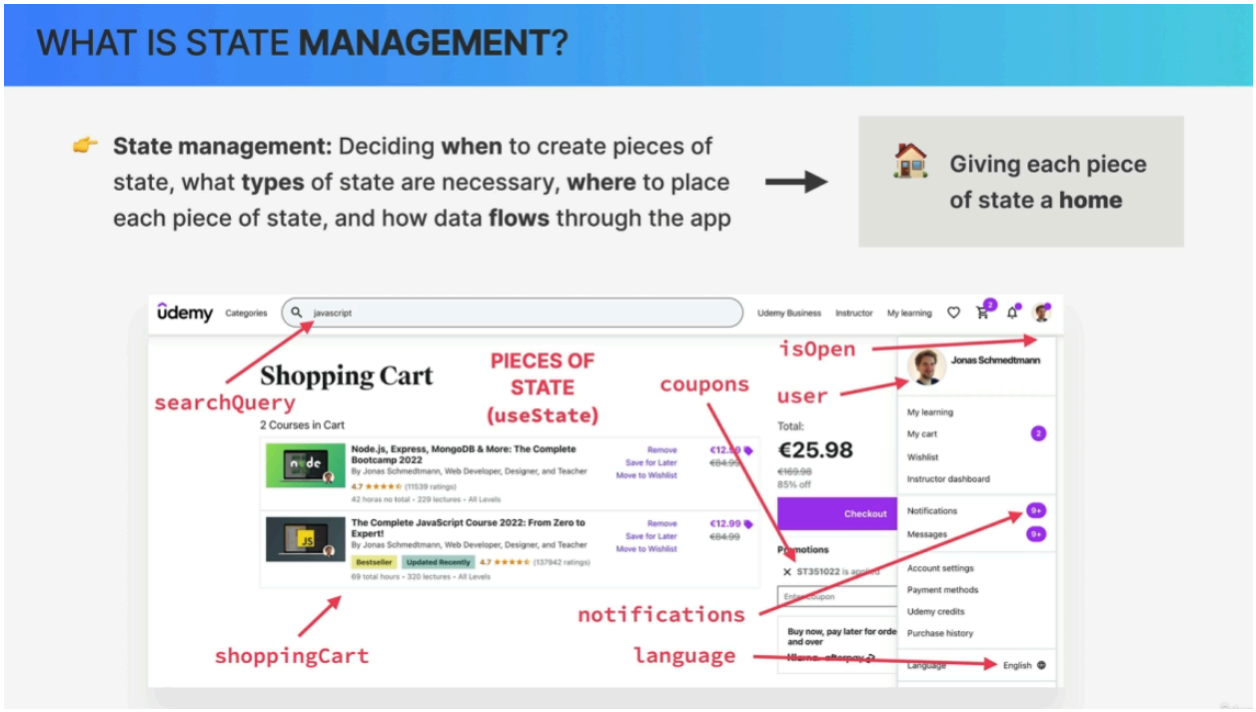
Section 7

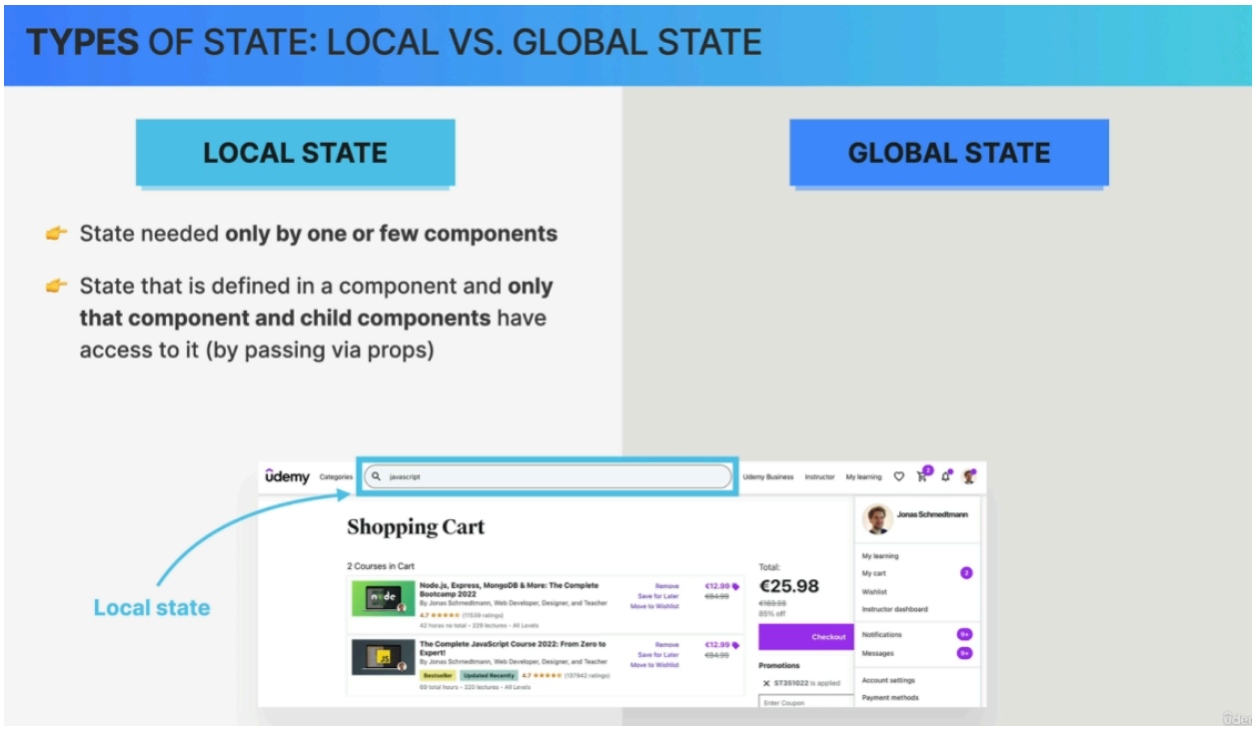
Thinking in React

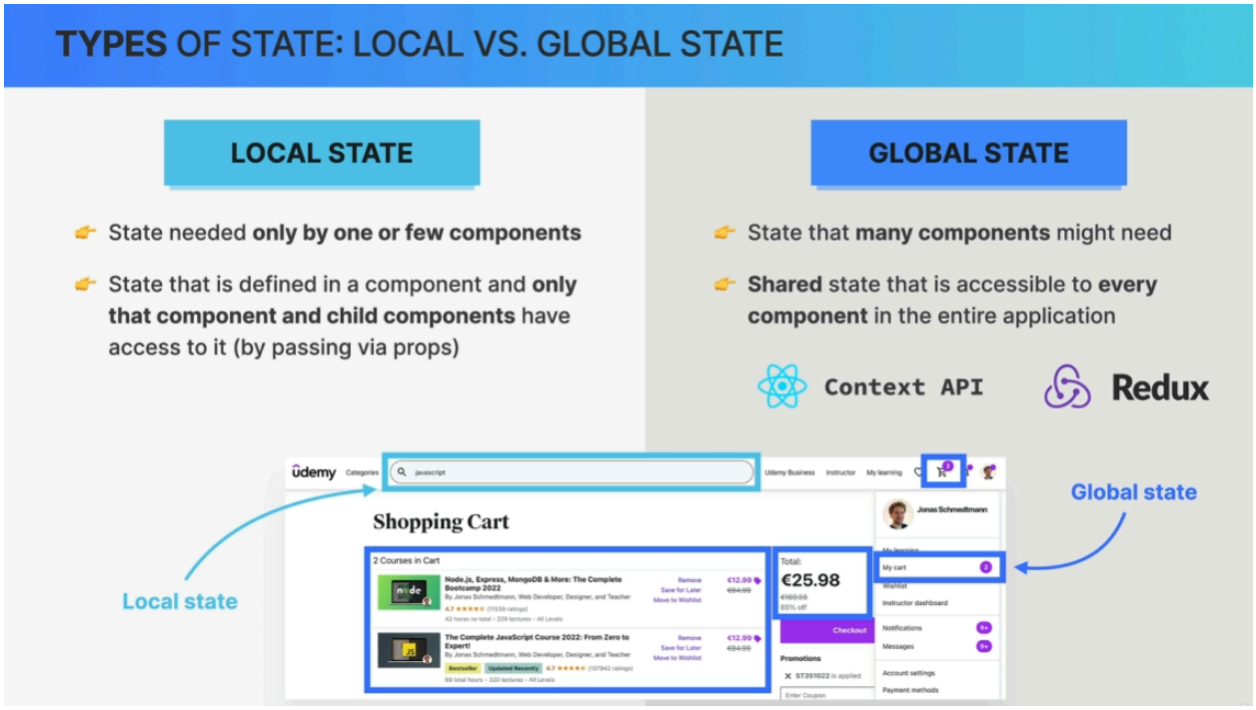




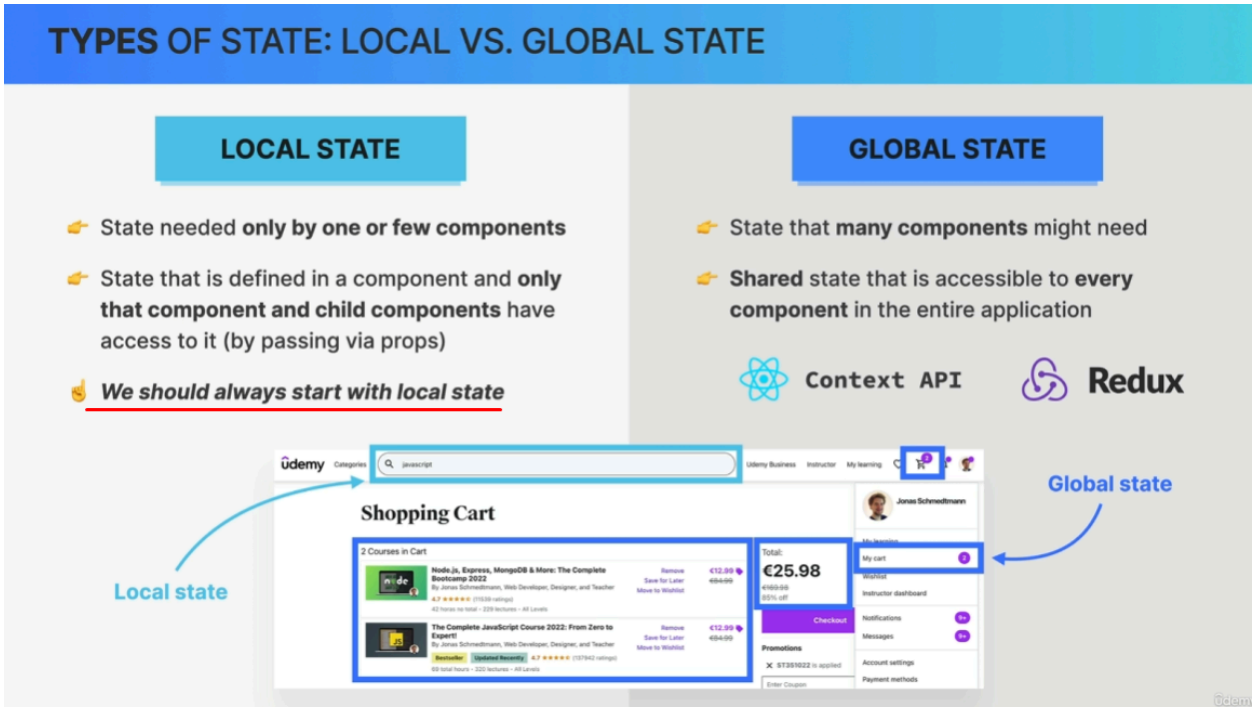
State Management

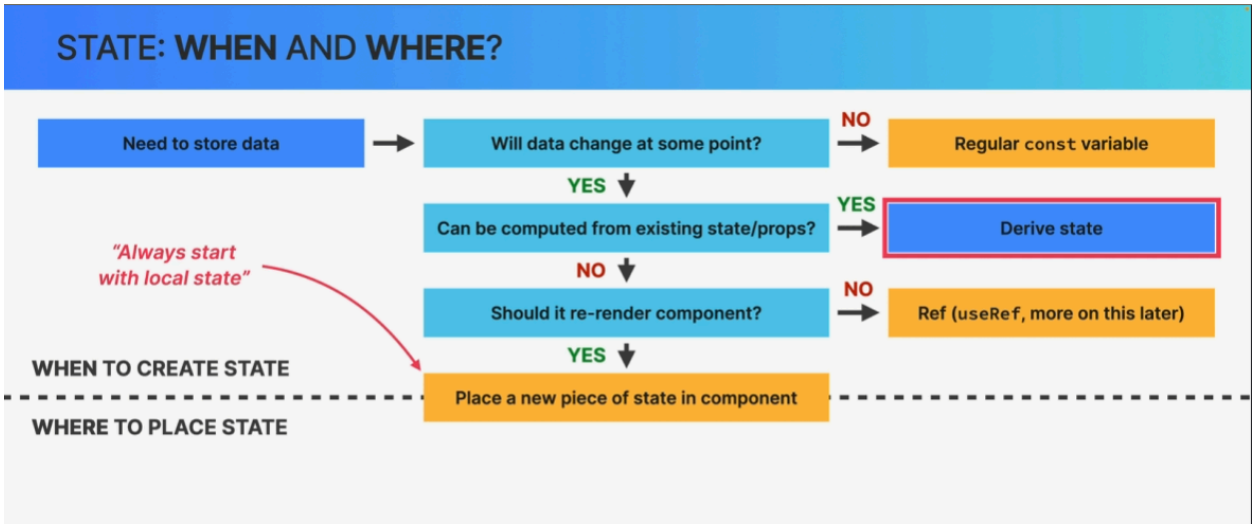


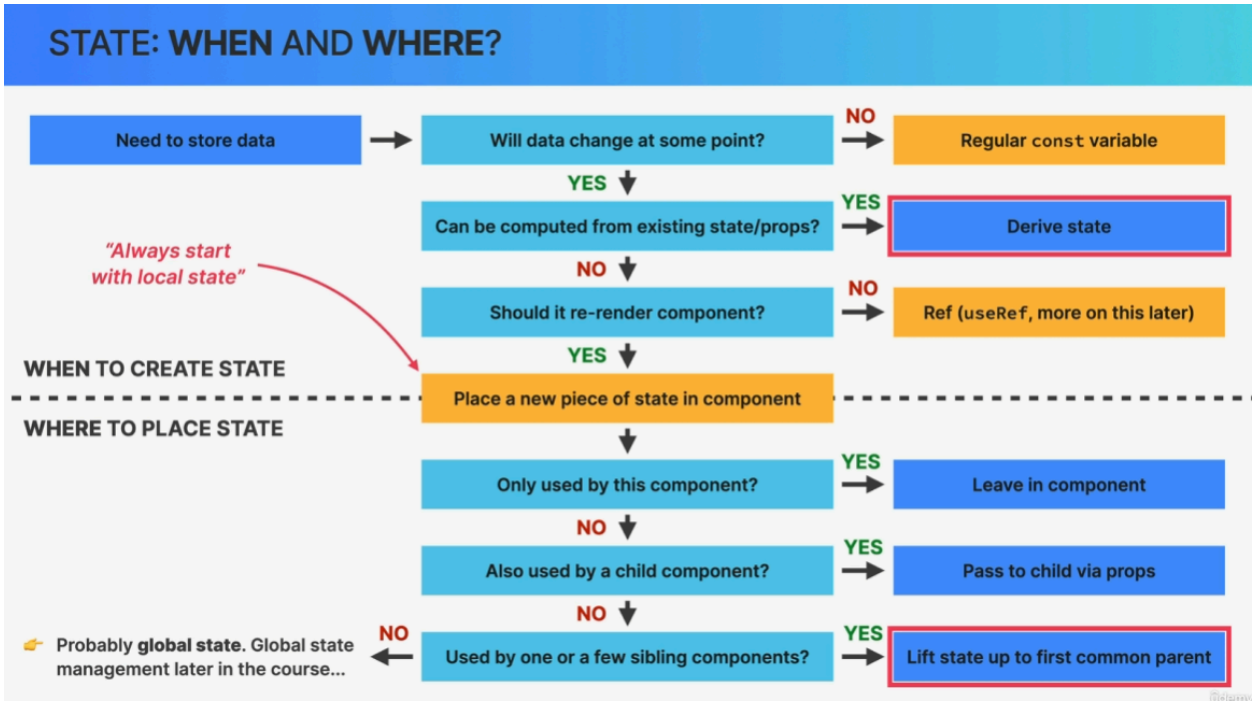




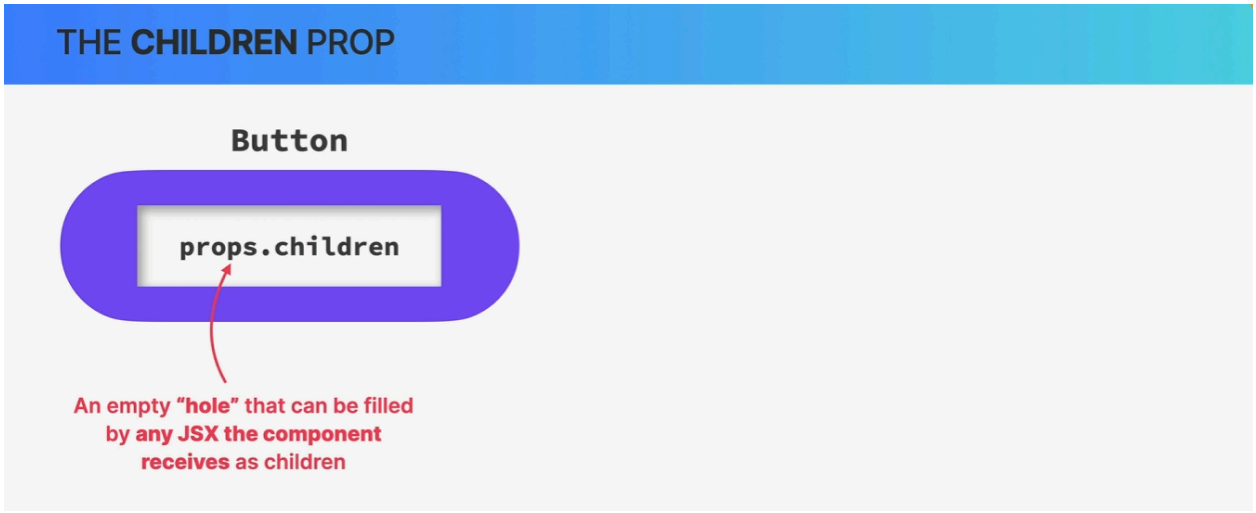
In fact, one important guideline in state management is to always start with local state and only move to a global state if you truly need it.

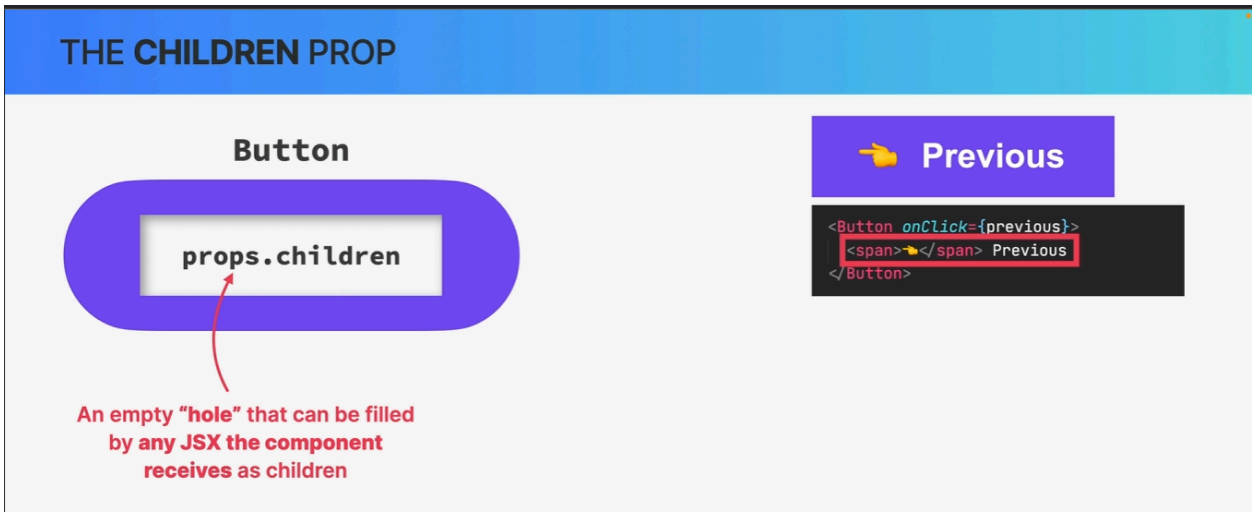


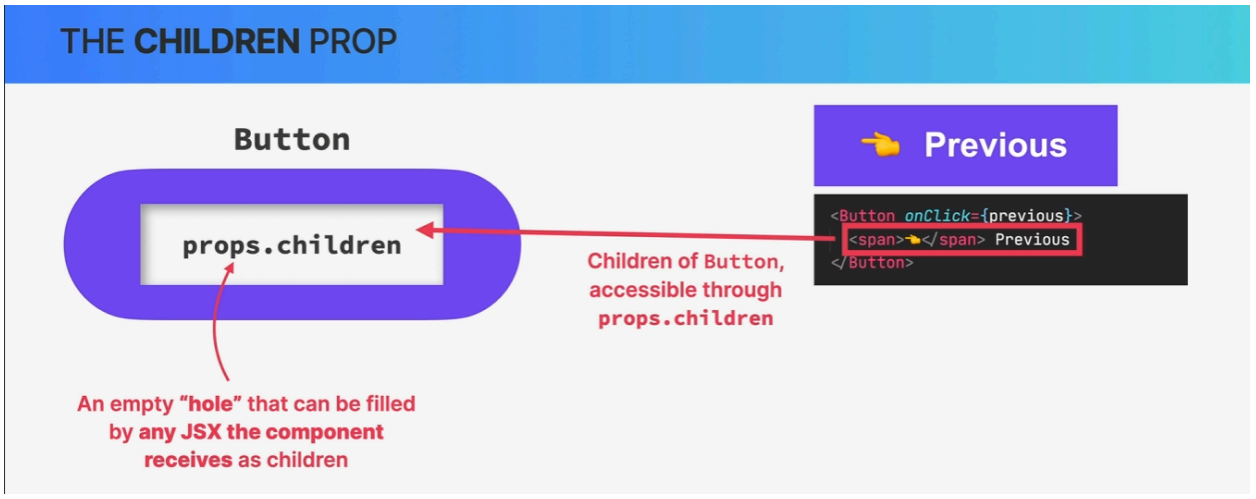

STATE: WHEN AND WHERE?
Need to store data
Will data change at some point?
NO
Regular const variable
YES
Can be computed from existing state/props?
YES
Derive state
"Always start with local state"
NO
Should it re-render component?
NO
Ref (useRef, more on this later)
YES
WHEN TO CREATE STATE
Place a new piece of state in component
WHERE TO PLACE STATE


STATE: WHEN AND WHERE?
Need to store data
Will data change at some point?
NO
Regular const variable
YES
Can be computed from existing state/props?
YES
Derive state
"Always start with local state"
NO
Should it re-render component?
NO
Ref (useRef, more on this later)
YES
WHEN TO CREATE STATE
Place a new piece of state in component
WHERE TO PLACE STATE
Only used by this component?
YES
Leave in component
NO
Also used by a child component?
YES
Pass to child via props
NO
Probably global state. Global state management later in the course...
NO
Used by one or a few sibling components?
YES
Lift state up to first common parent

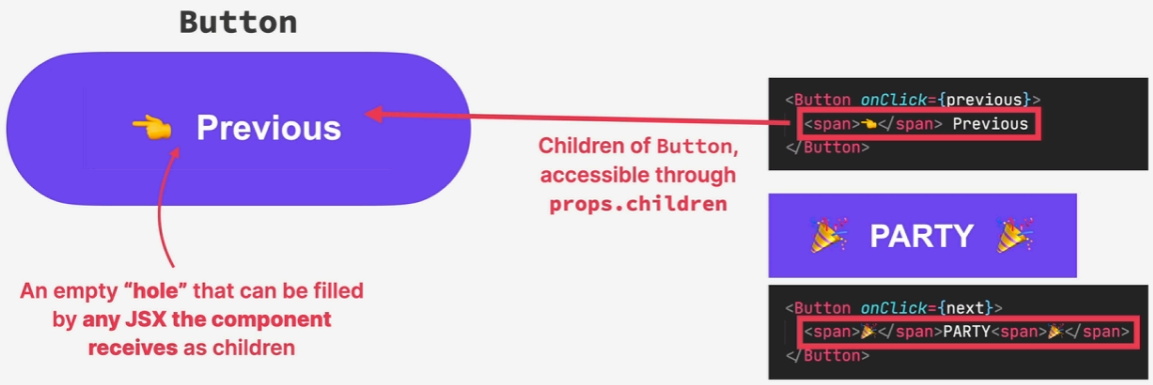
The Children prop





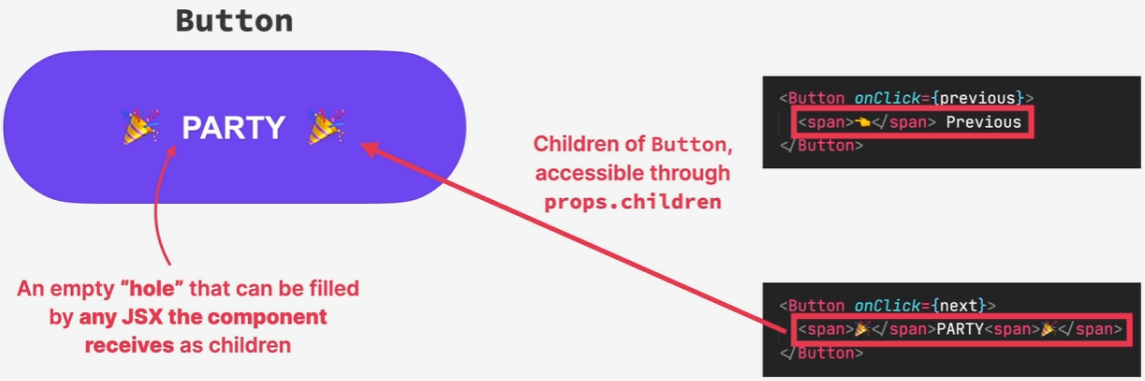


THE **CHILDREN** PROP



- 👉 The children prop allow us **to pass JSX into an element** (besides regular props)
- 👉 Essential tool to make **reusable** and **configurable** components (especially component **content**)

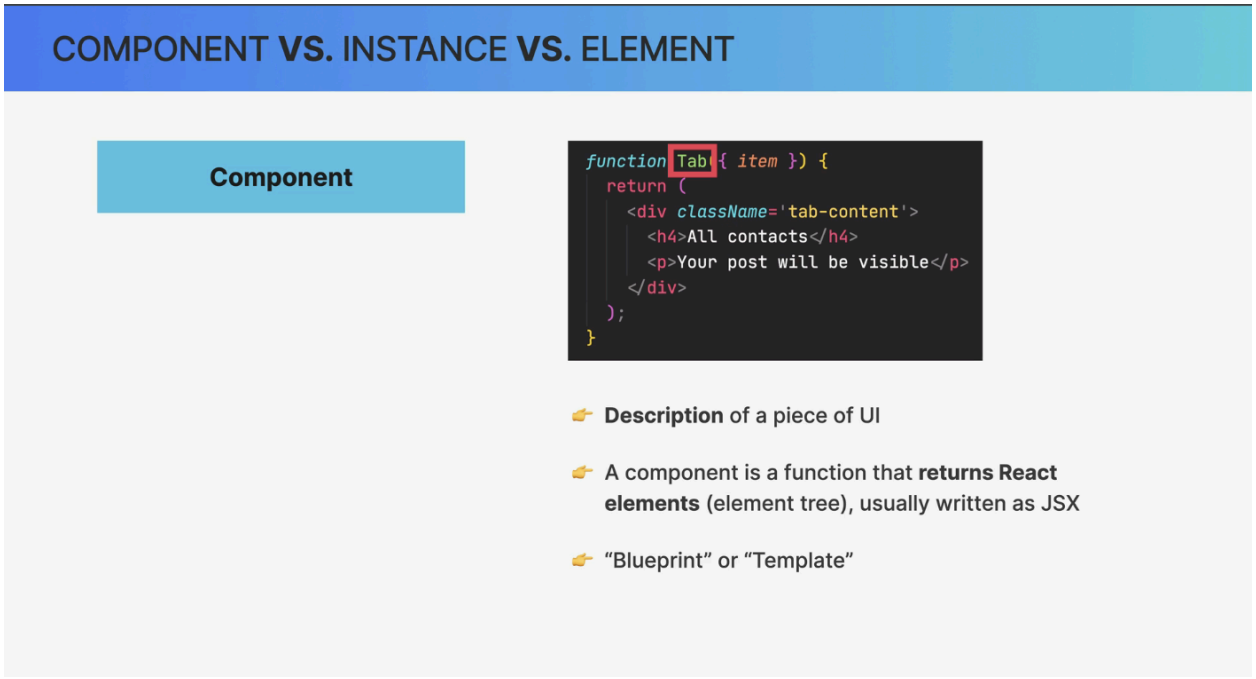
THE **CHILDREN** PROP

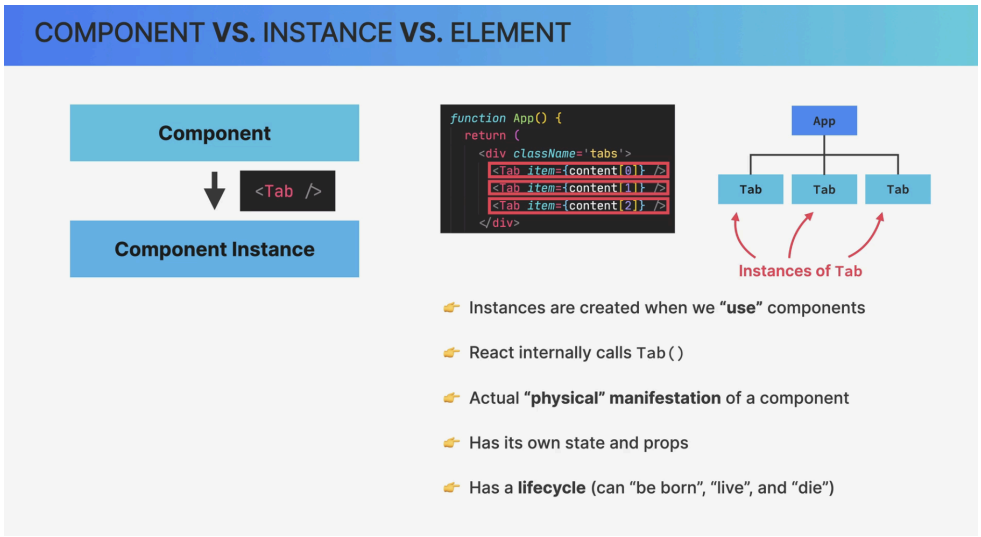


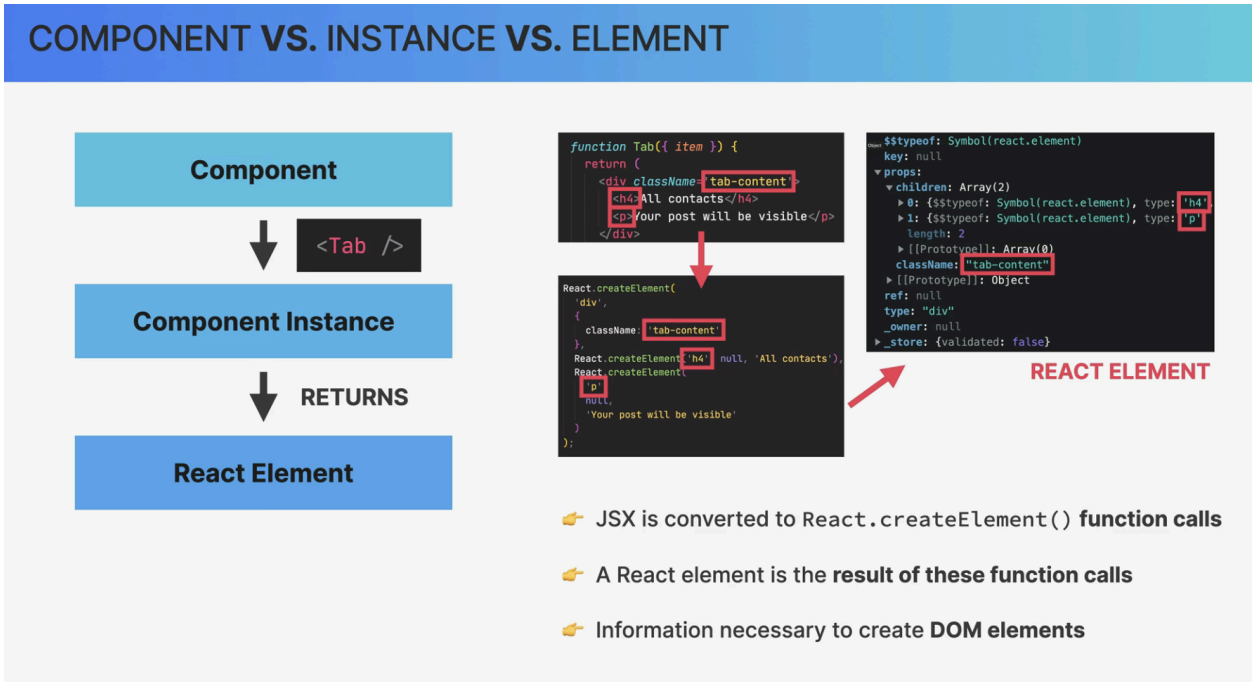
- 👉 The children prop allow us **to pass JSX into an element** (besides regular props)
- 👉 Essential tool to make **reusable** and **configurable** components (especially component **content**)
- 👉 Really useful for **generic** components that **don't know their content** before being used (e.g. modal)

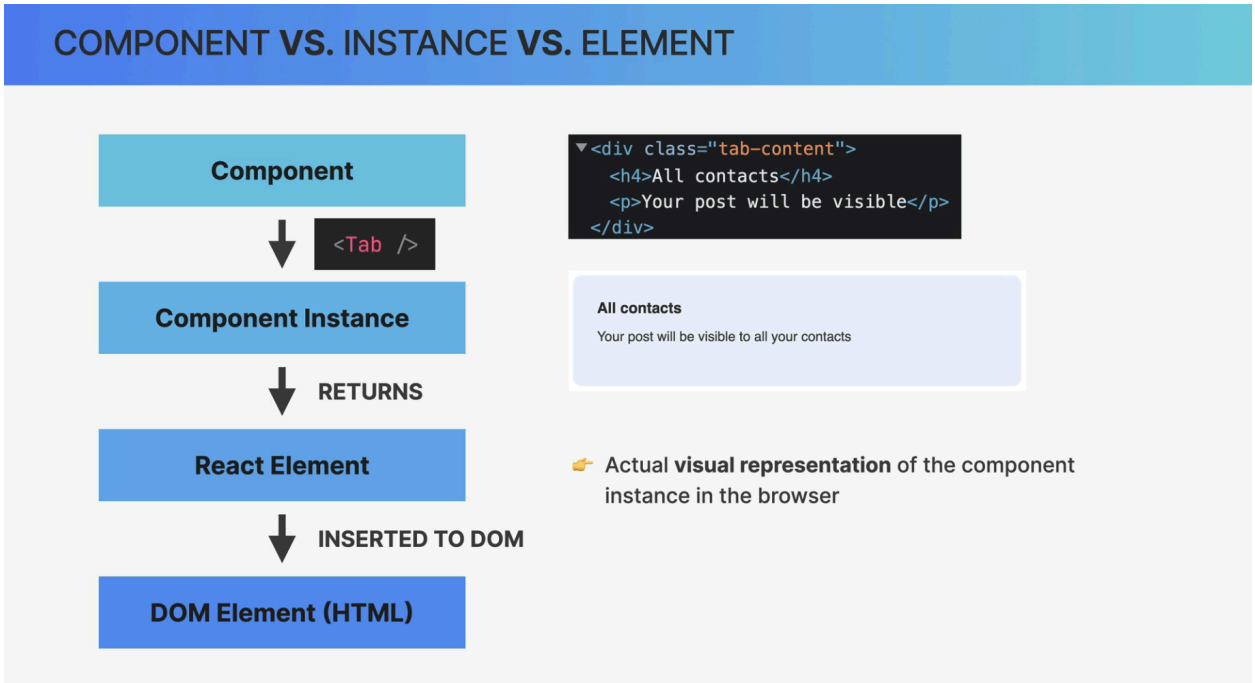
Section 11: How React Works Behind the Scenes

Components, Instances, and Elements



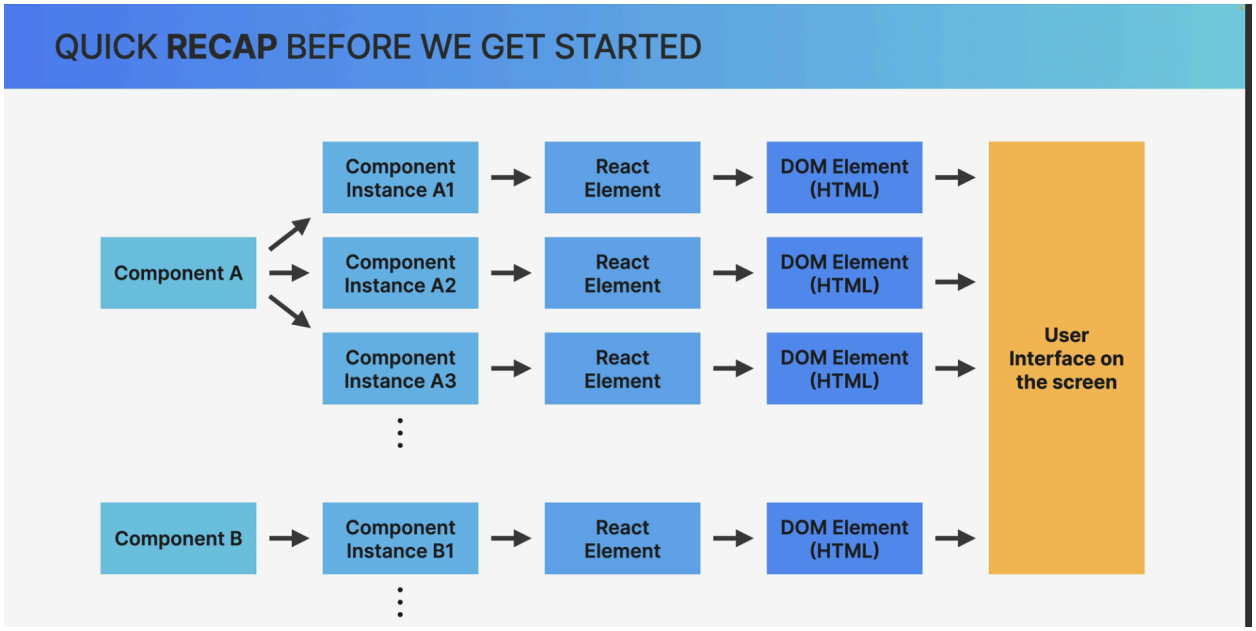


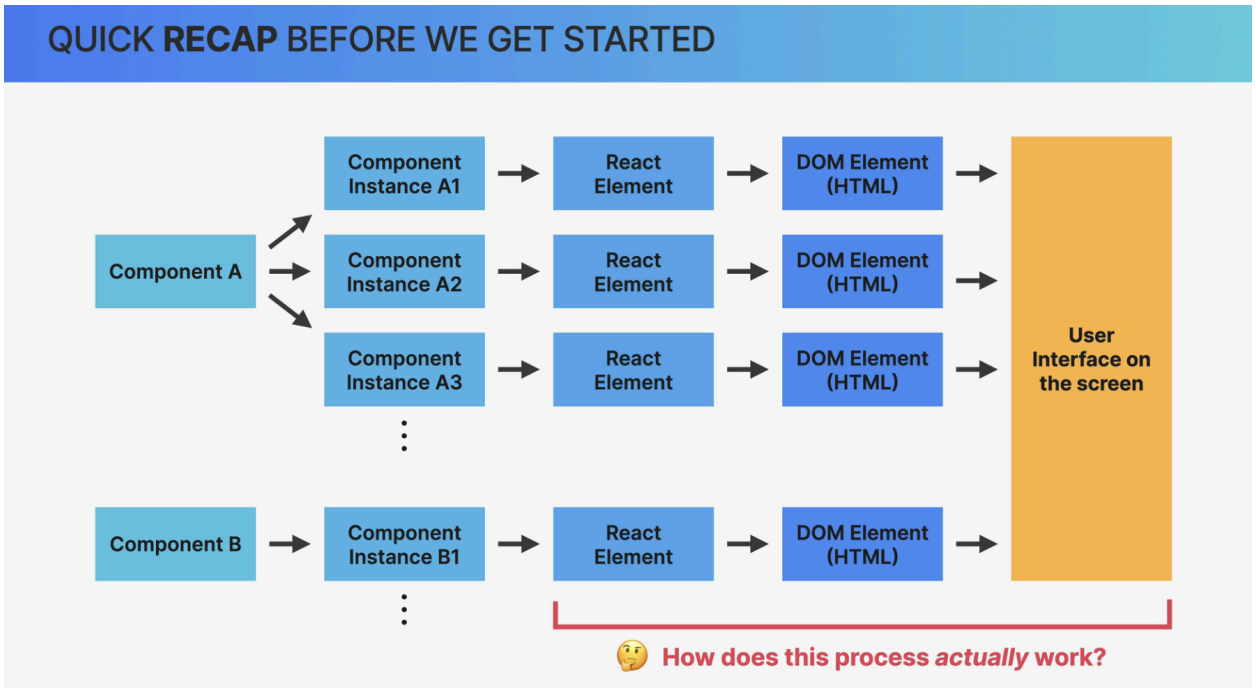




React elements just live inside the React app and have nothing to do with the DOM.
They are simply converted to DOM elements when painted on the screen in this final step.

Quick Recap


QUICK RECAP BEFORE WE GET STARTED
Component A
Component B
Component Instance A1
Component Instance A2
Component Instance A3
Component Instance B1
React Element
React Element
React Element
React Element
DOM Element (HTML)
DOM Element (HTML)
DOM Element (HTML)
DOM Element (HTML)
User Interface on the screen


QUICK RECAP BEFORE WE GET STARTED
Component A
Component B
Component Instance A1
Component Instance A2
Component Instance A3
Component Instance B1
React Element
React Element
React Element
React Element
DOM Element (HTML)
DOM Element (HTML)
DOM Element (HTML)
DOM Element (HTML)
User Interface on the screen
How does this process actually work?

Overview - How components are displayed on the screen

So this process we're about to study is started by React each time a new render is triggered, most of the time by updating the state somewhere in the application.

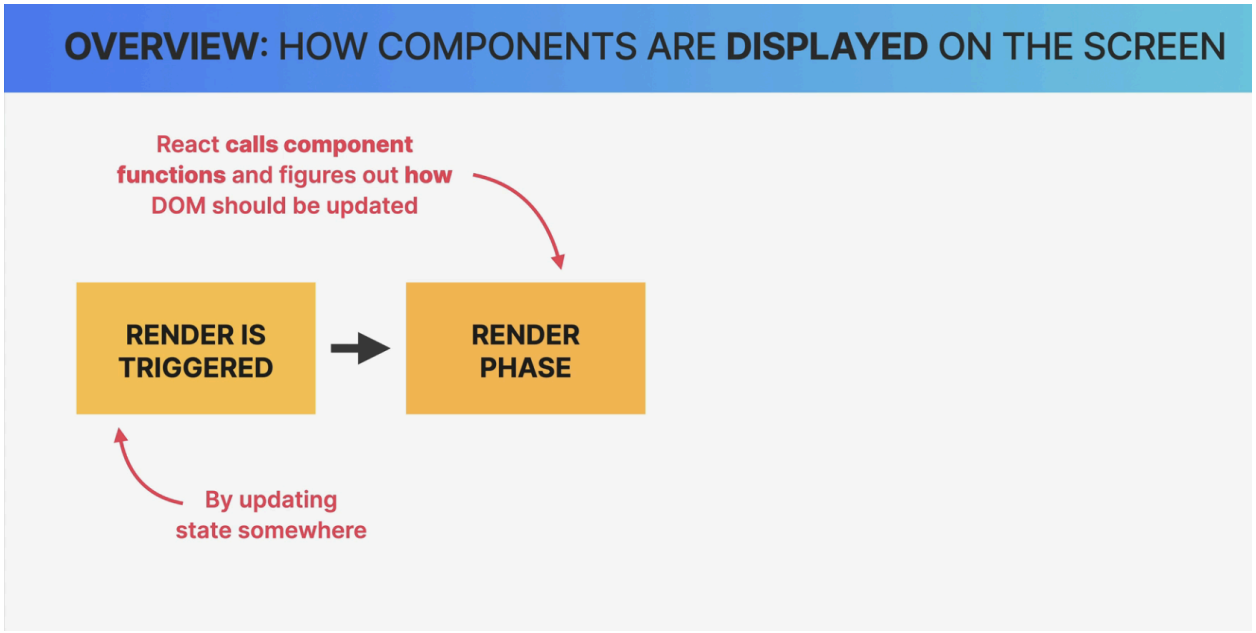
So state changes trigger renders and so it makes sense that the next phase is the render phase.

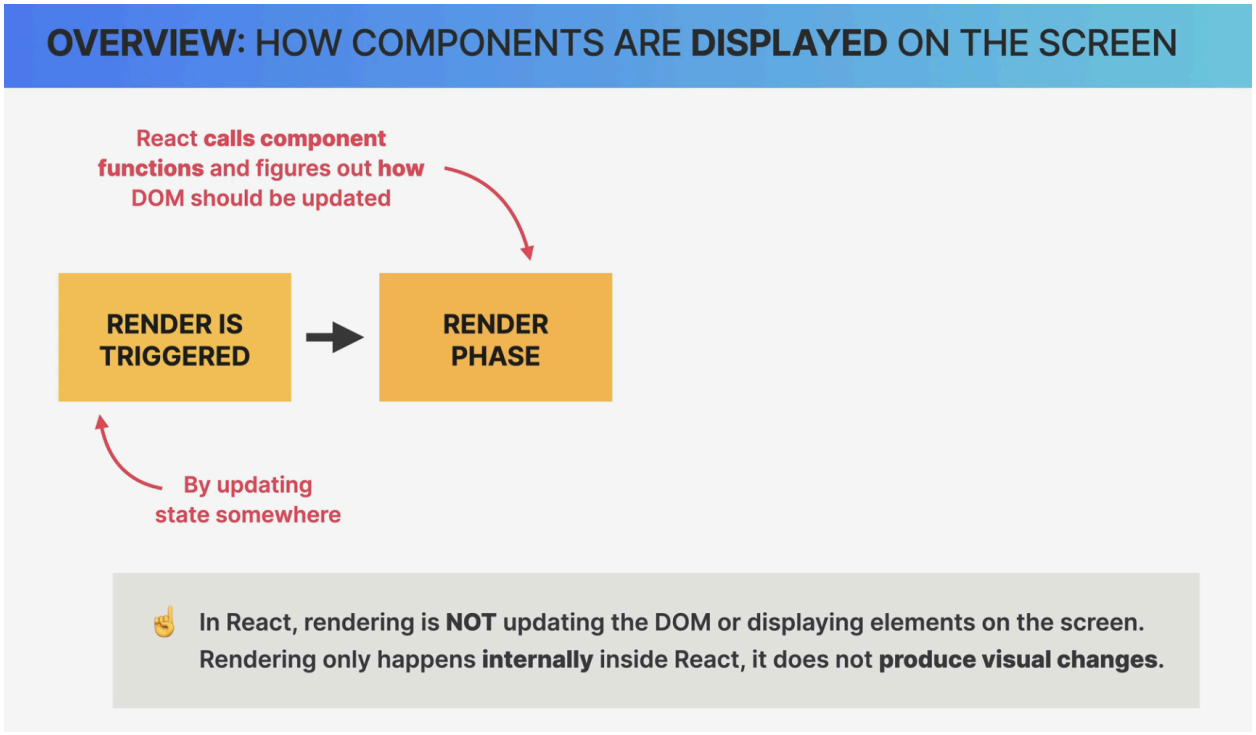
In this phase, React calls our component functions and figures out how it should update the DOM.

However, it does not update the DOM in this phase. And so Reacts definition of render is very different from what we usually think of as a render, which can be quite confusing.

So again, in React, rendering is not about updating the DOM or displaying elements on the screen.

Rendering only happens internally inside of React so it does not produce any visual changes.



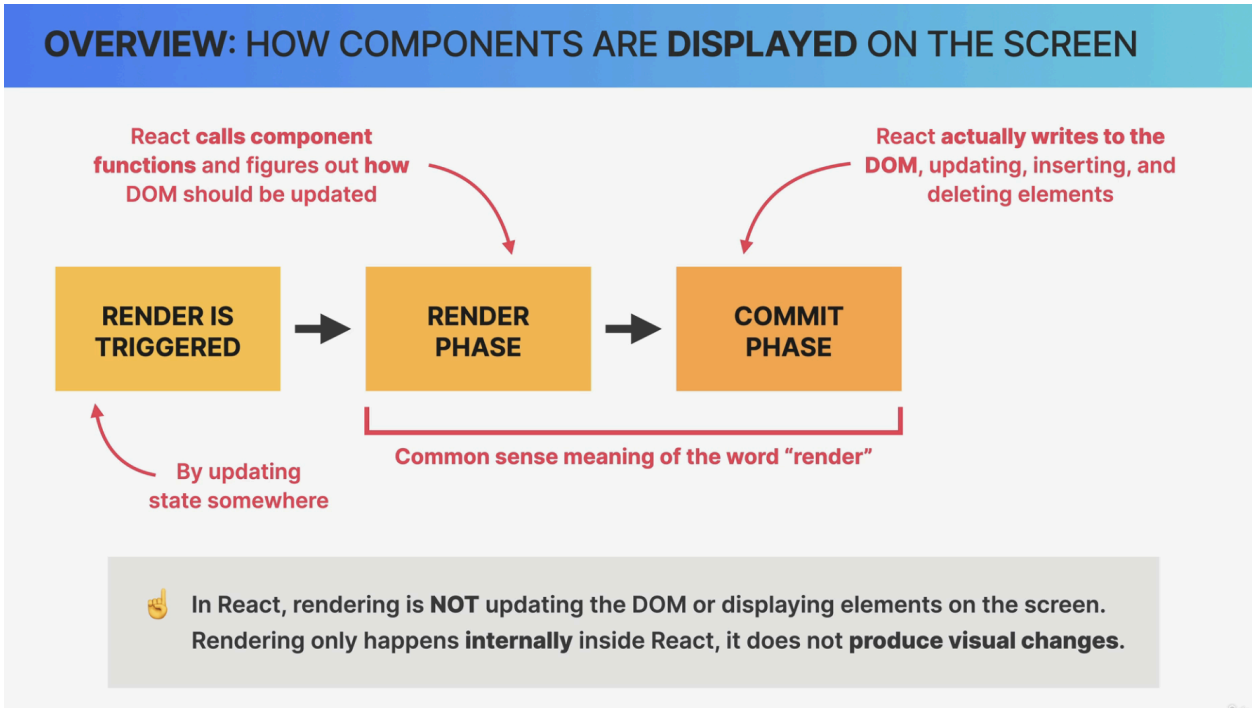


I have always used the term rendering with the meaning of displaying elements on the screen because that was just easy to understand and it just made sense, right?

However, as we just learned, the rendering that I used to mean is this render plus the next phase. And speaking of that next phase, once React knows how to update a DOM, it does so in the commit phase.

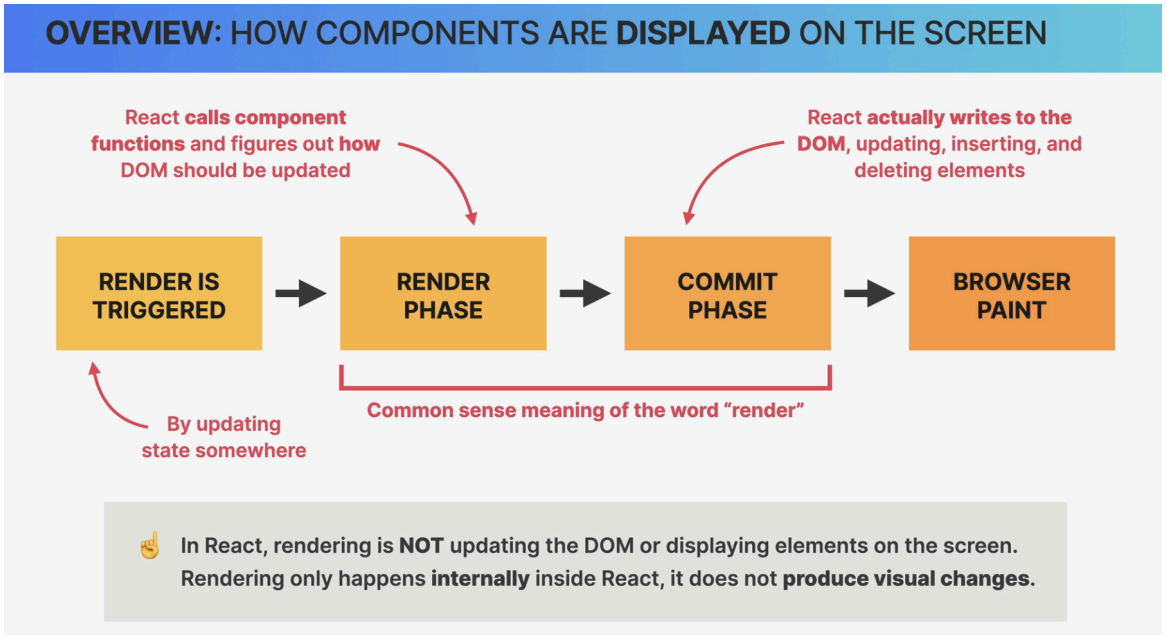
So, in this phase, new elements might be placed in the DOM and already existing elements might get updated or deleted to correctly reflect the current state of the application.

So it's this commit phase that is responsible for what we traditionally call rendering



Then finally, the browser will notice that the DOM has been updated and so it repaints the screen.

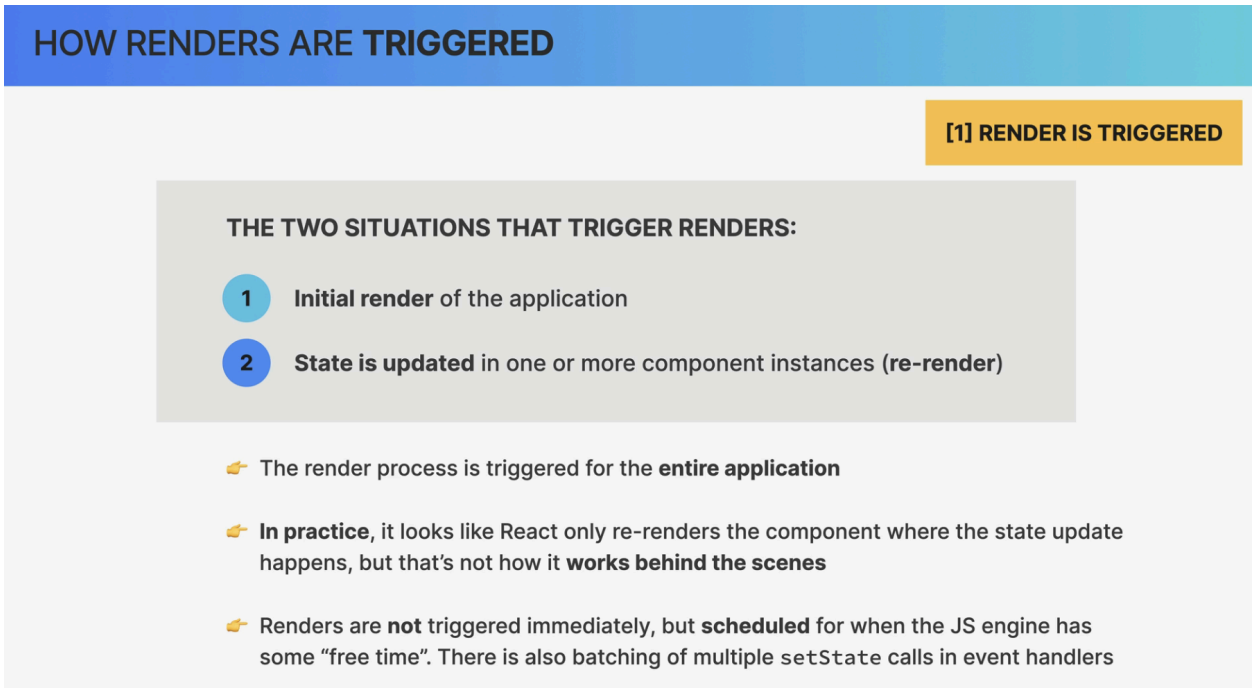
Now this has nothing to do with React anymore but it's still worth mentioning that it's this final step that actually produces the visual change that users see on their screens.



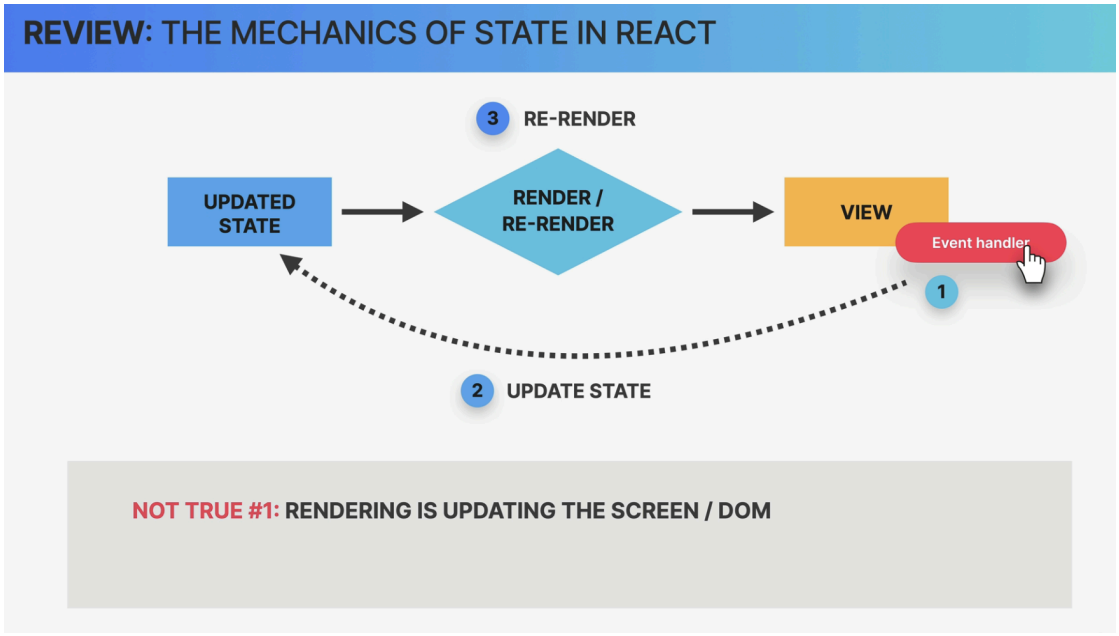
Finally, I just want to mention that a render is not triggered immediately after a state update happens.

Instead, it's scheduled for when the JavaScript engine has some free time on its hands. But this difference is usually just a few milliseconds that we won't notice.

There are also some situations like multiple sets of state calls in the same function where renders will be batched



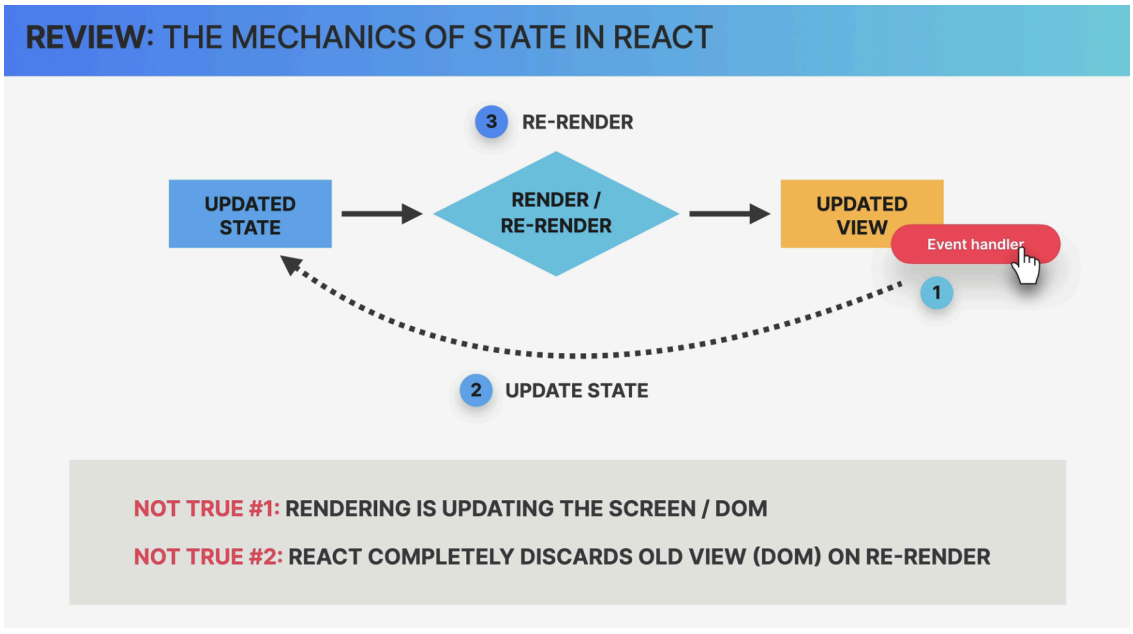
How rendering works



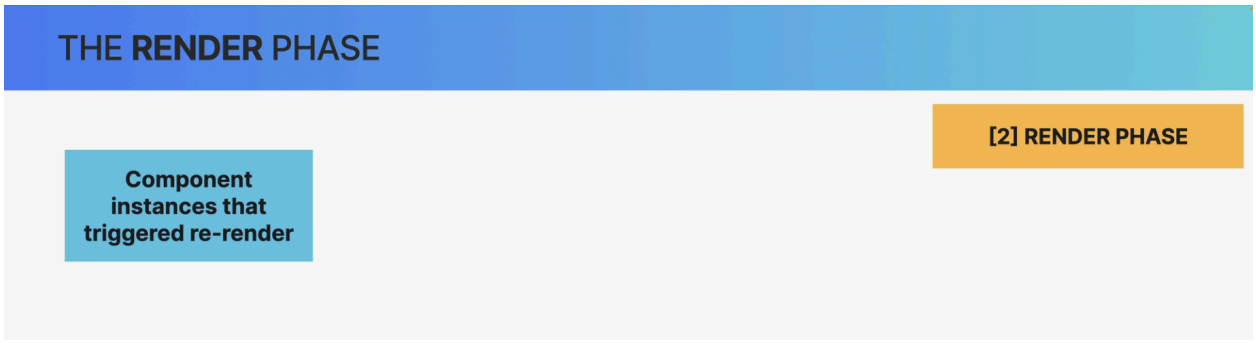
Remember this diagram?

I told you we can conceptually imagine this as a new view rendered on the screen, so into the DOM.

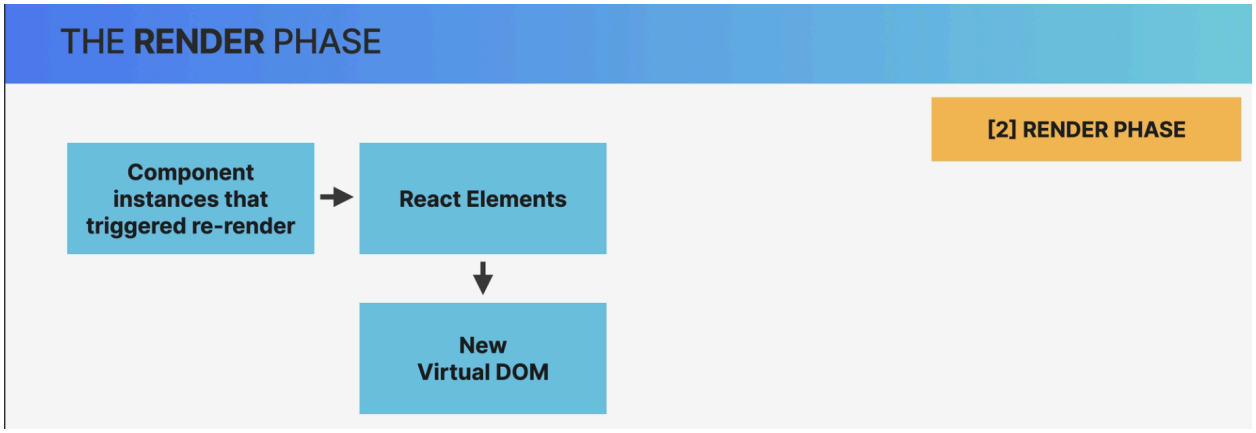
However, now we know that this was **technically not true** because **rendering is not about the screen or the DOM or the view, it's just about *calling component functions***.



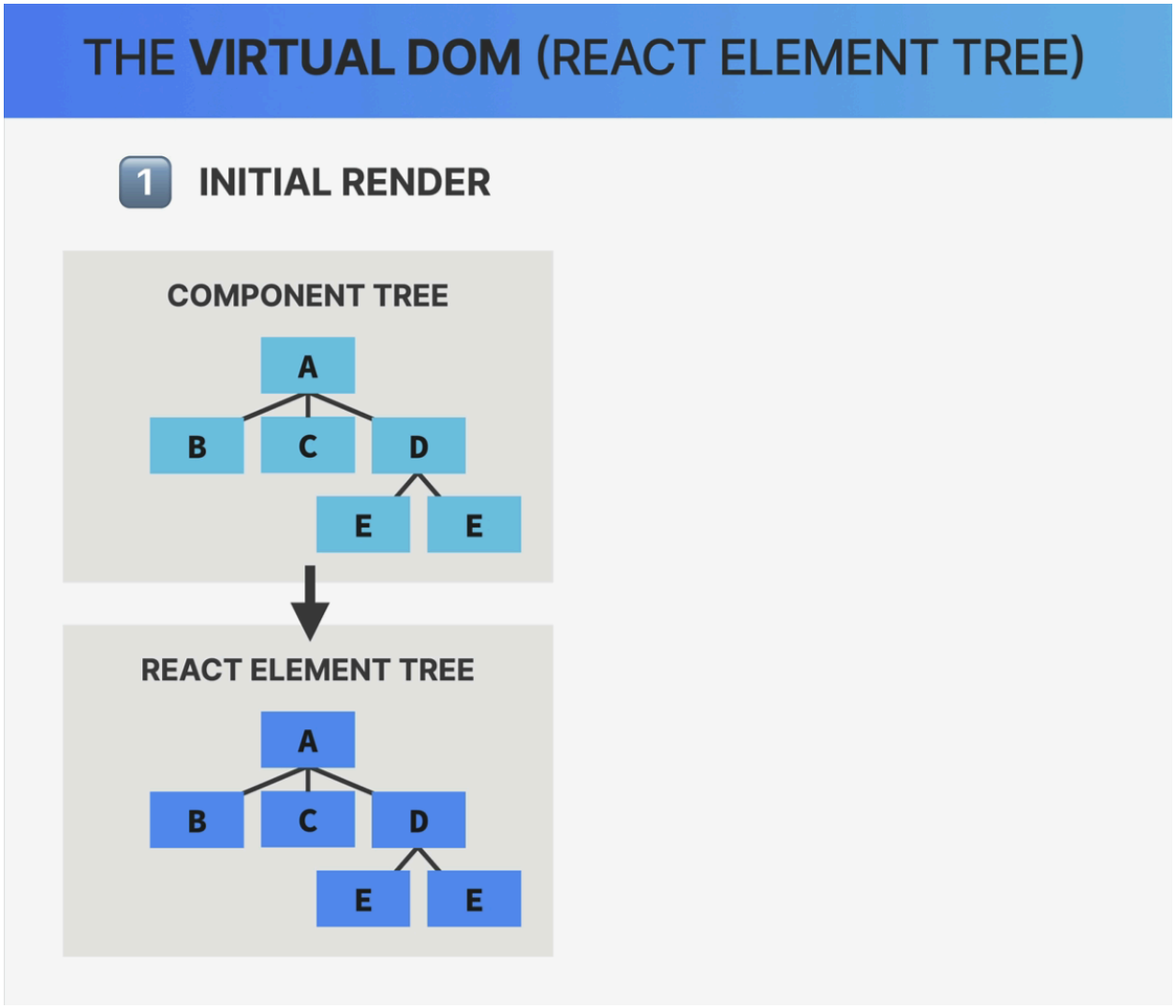
So, at the **beginning of the render phase**, **React will go through the entire component tree**, **take all the component instances that triggered a re-render**, **and render them, which simply means calling the corresponding component functions**



This will create updated React elements that altogether make up the so-called virtual DOM.



So on the **initial render**, **React will take the entire component tree and transform it into one big React element** which will be a React element tree like this one. And this is what we call the **virtual DOM**



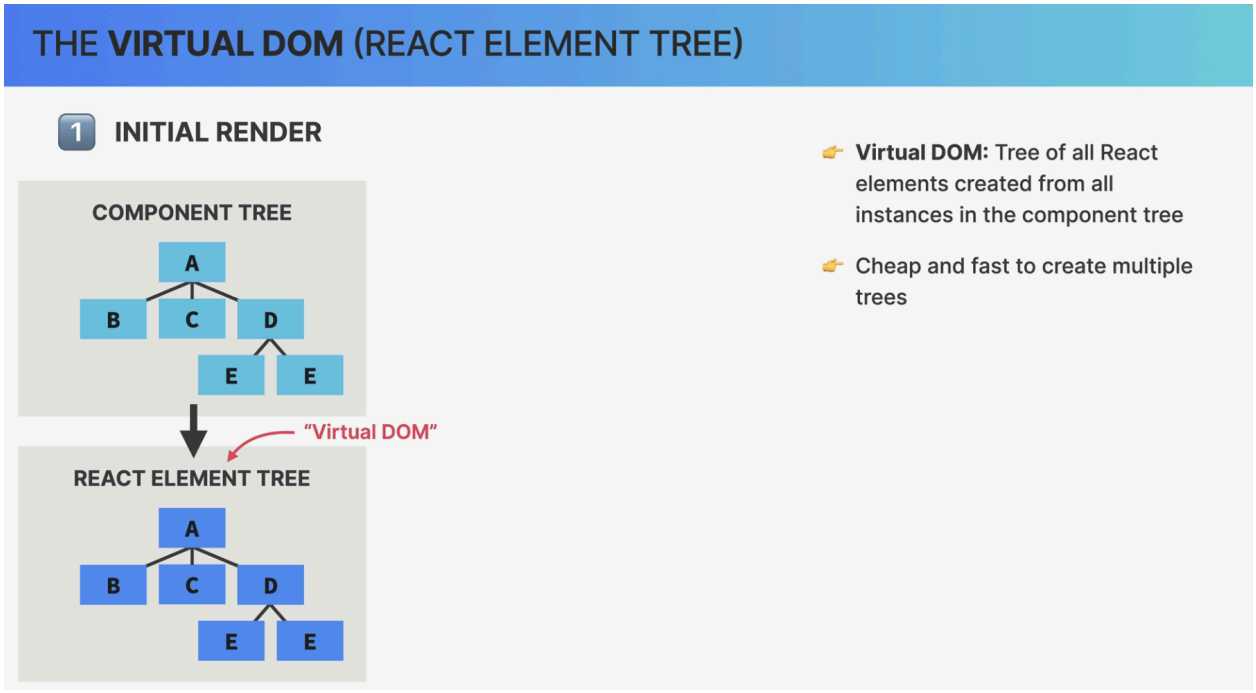
So, the **virtual DOM is just a *tree of all React elements* created from all instances in the component tree.**

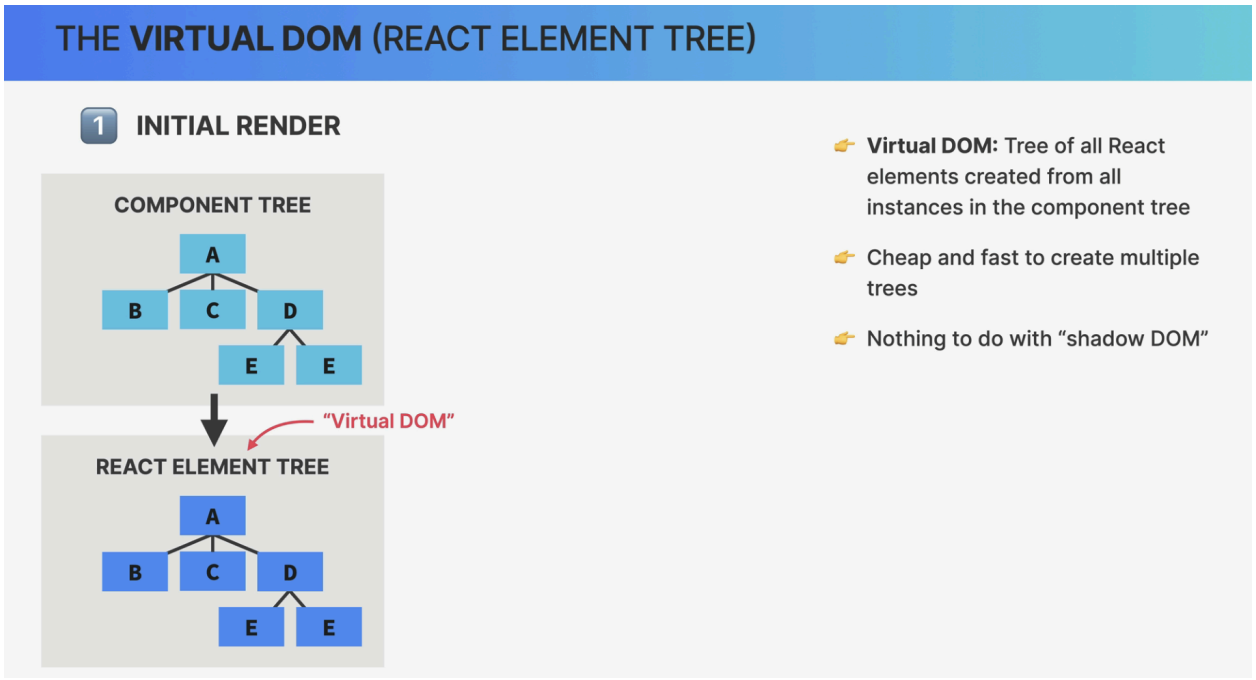
And **it's relatively cheap and fast to create a tree like this**, even if we need many iterations of it because, in the end, **it's just a JavaScript object.**

Now, virtual DOM is probably the most hyped and most used term when people describe what React is and how it works.

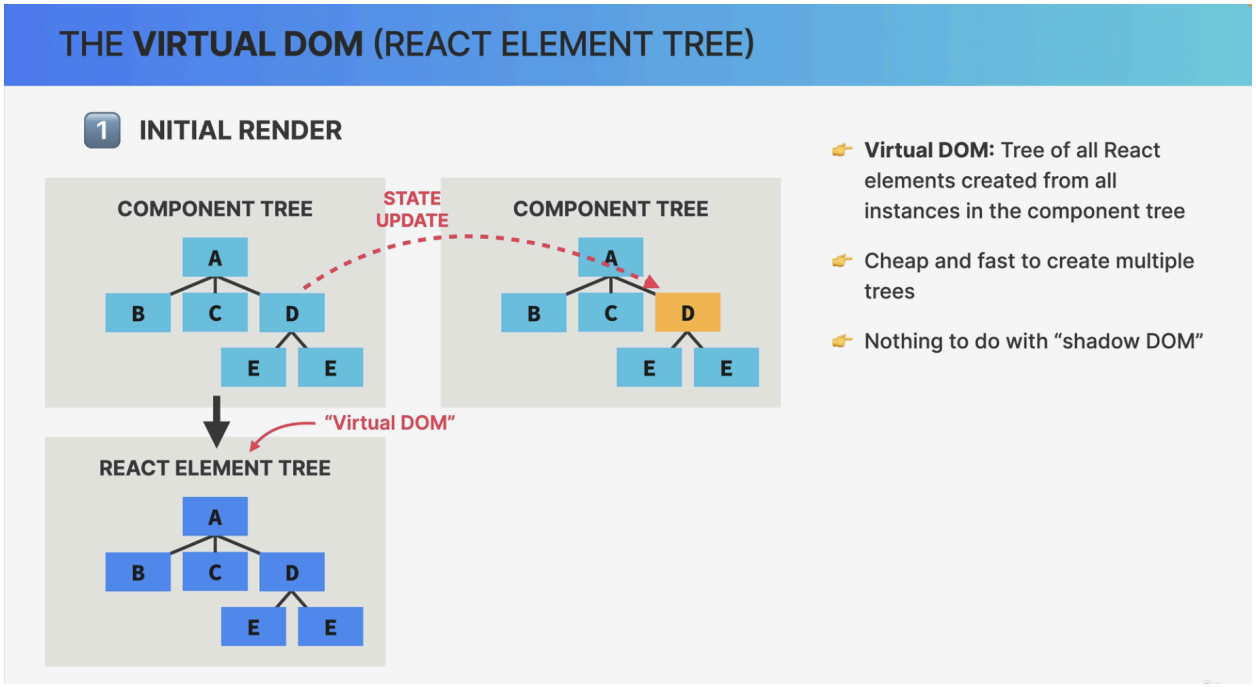
But if we think about it, if the virtual DOM is just a simple object, it's not such a big deal, right?

And that's why the React team has downplayed the meaning of this name. And the official documentation no longer mentions the term virtual DOM anywhere.



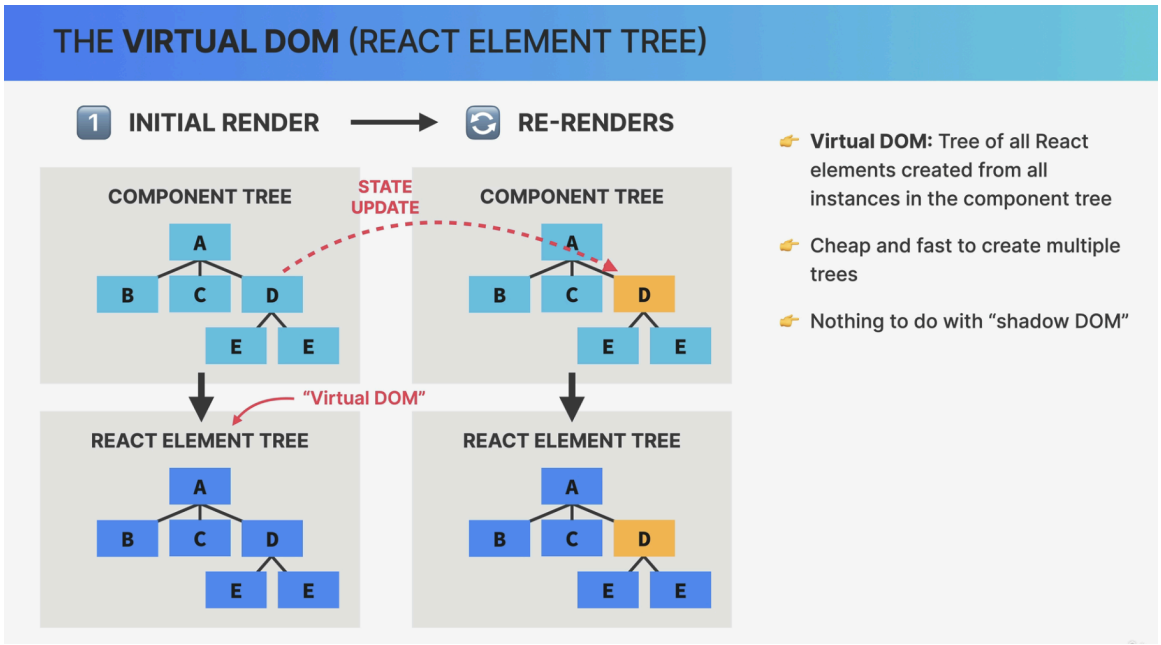


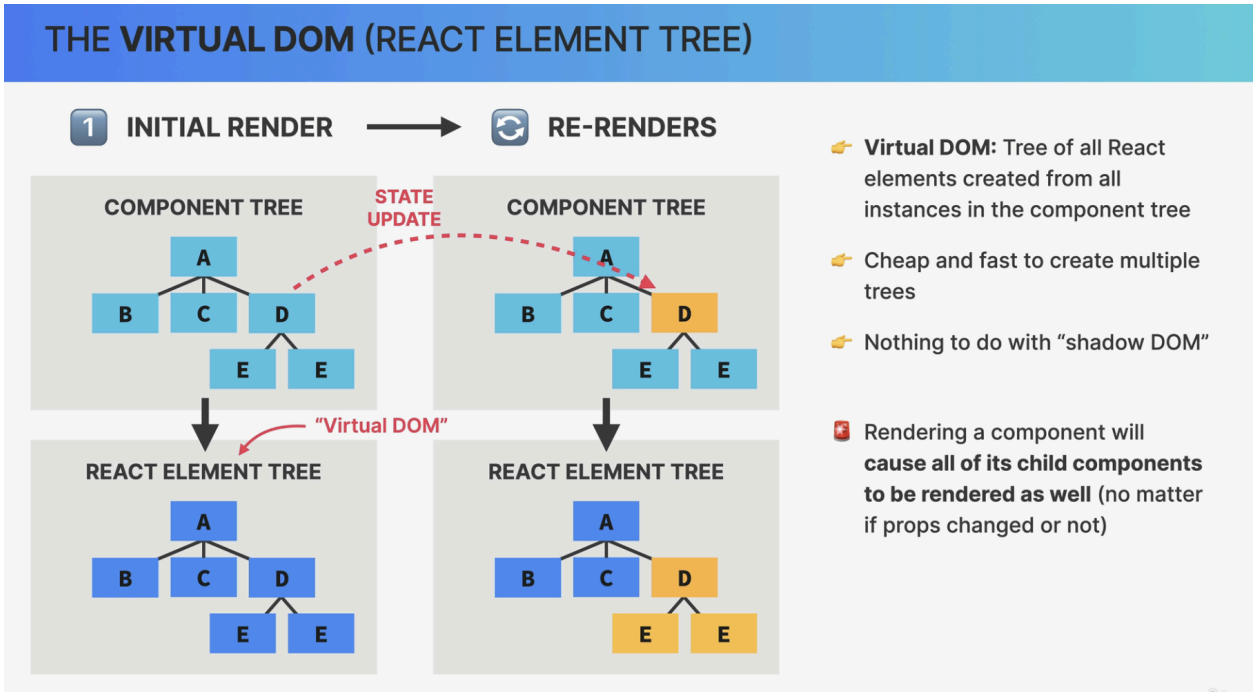
Also, some people confuse the term with the shadow DOM, even though it has nothing to do with the virtual DOM in React.
So the shadow DOM is just a browser technology that is used in stuff like web components.



But anyway, let's now suppose that there is gonna be a state update in component D, which will of course trigger a re-render.

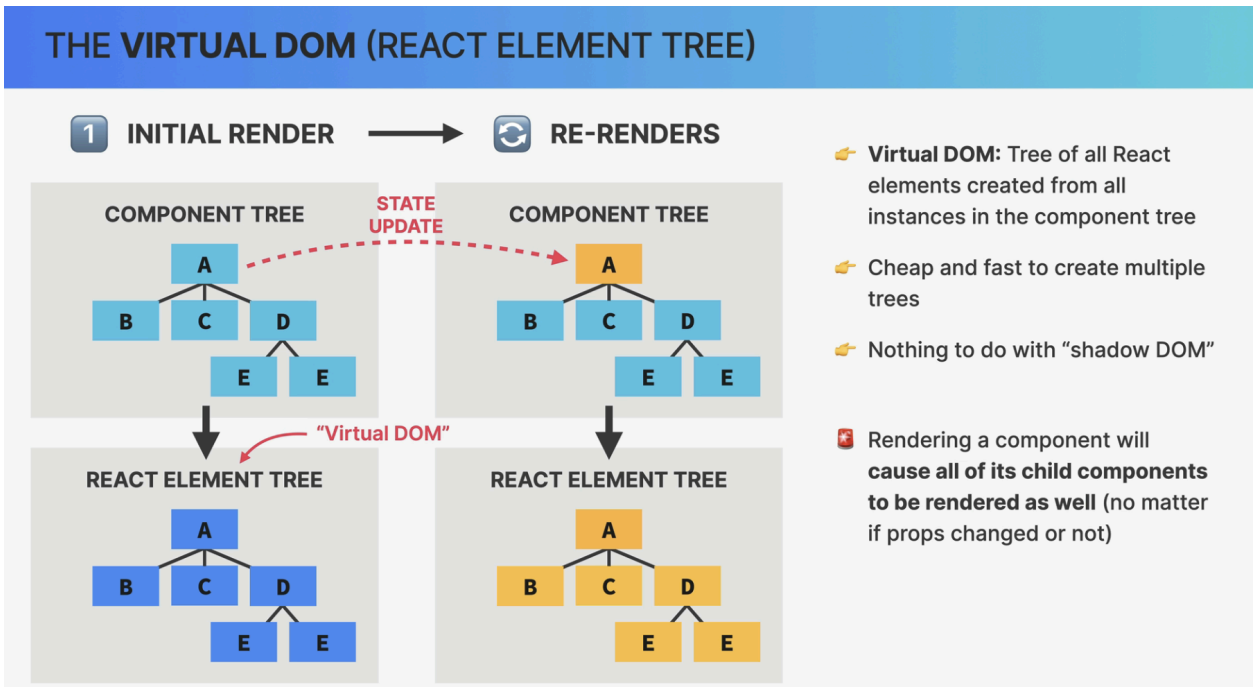
That means that React will call the function of component D again and place the new React element in a new React element tree.

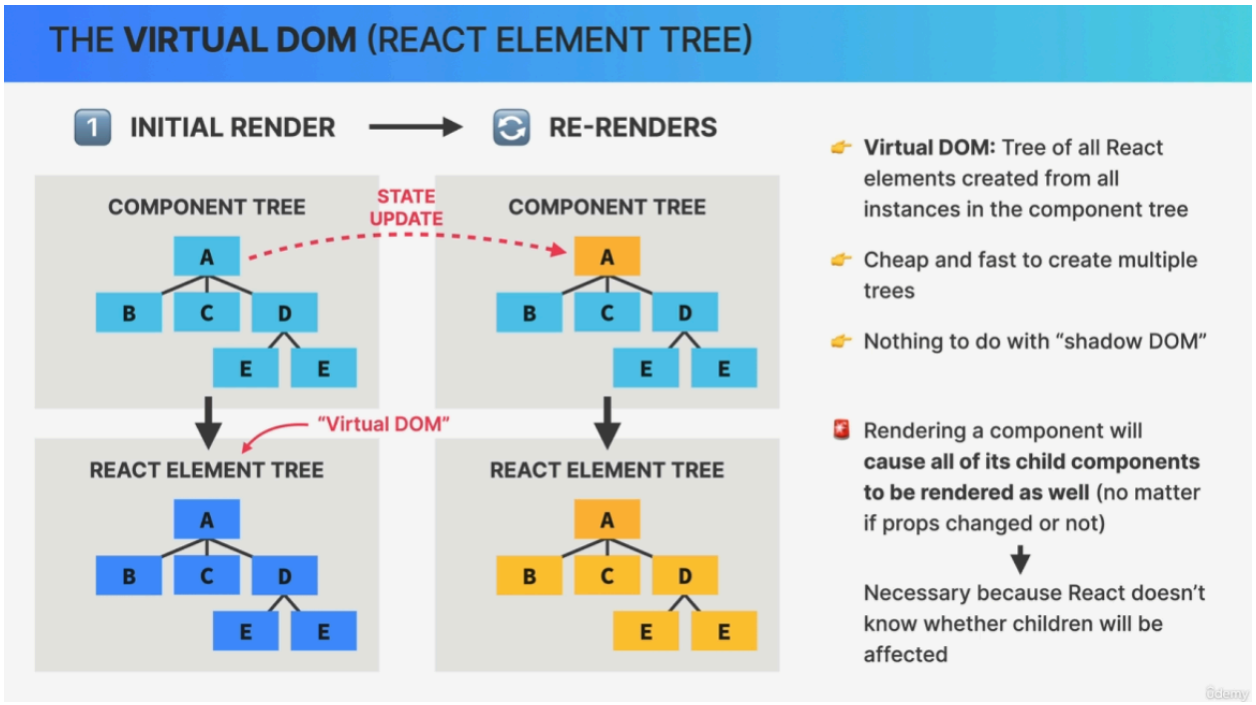




But now comes the **very important** part, which is this.

Whenever React renders a component, that render will cause all of its child components to be rendered as well. And that happens no matter if the props that we passed down have changed or not.



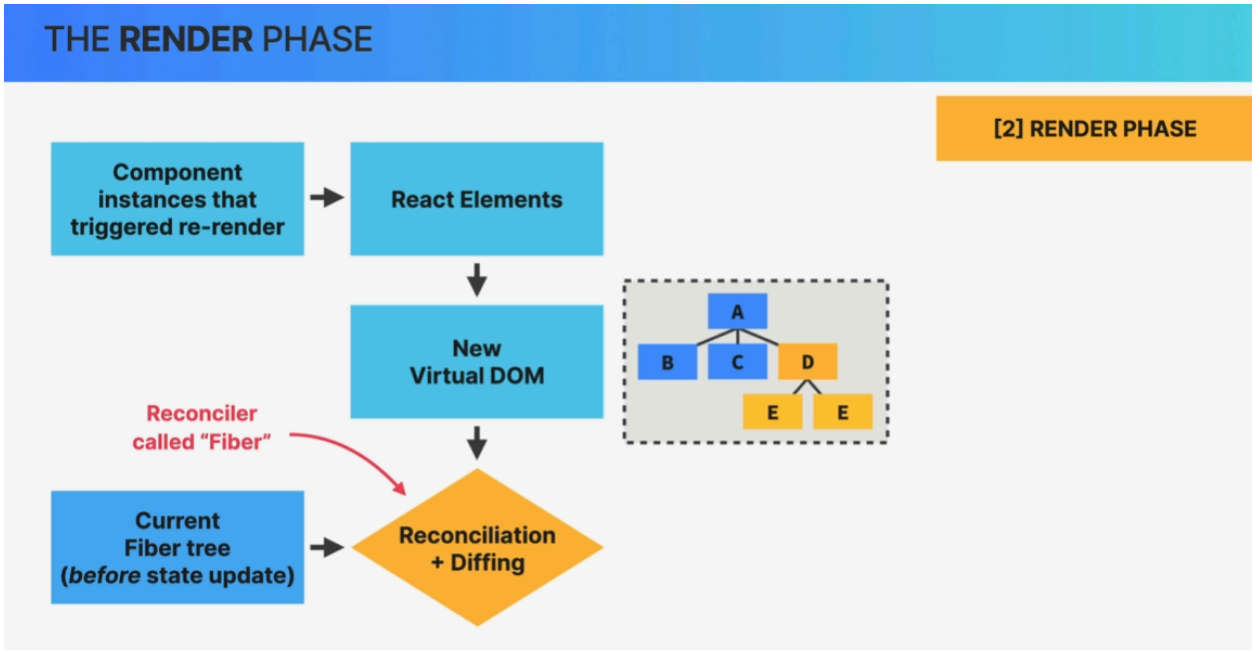


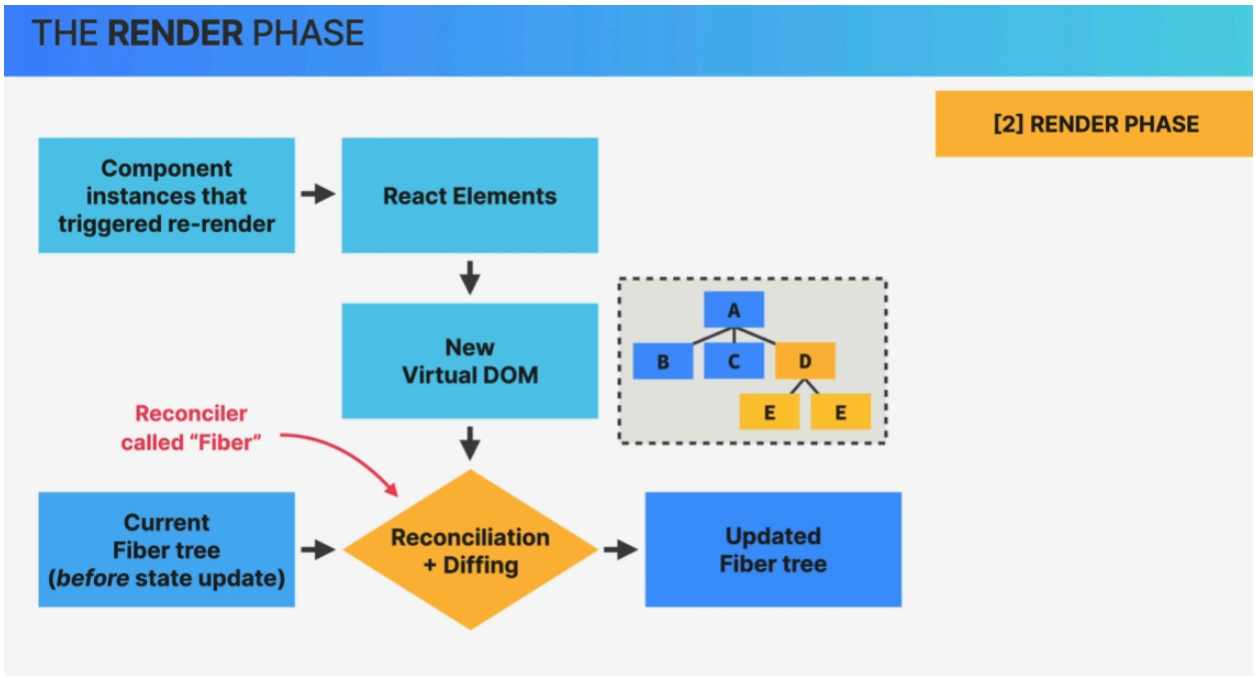
So again, if the updated components return one or more other components, those nested components will be re-rendered as well, all the way down the component tree.

This means that if you update the highest component in a component tree, in this example component A, then the entire application will be re-rendered.

Now, this may sound crazy, but React uses this strategy because it doesn't know beforehand whether an update in a component will affect the child components or not.

Also, keep in mind once again that this does not mean that the entire DOM is updated.
It's just a virtual DOM that will be recreated which is really not a big problem in small or medium-sized applications.



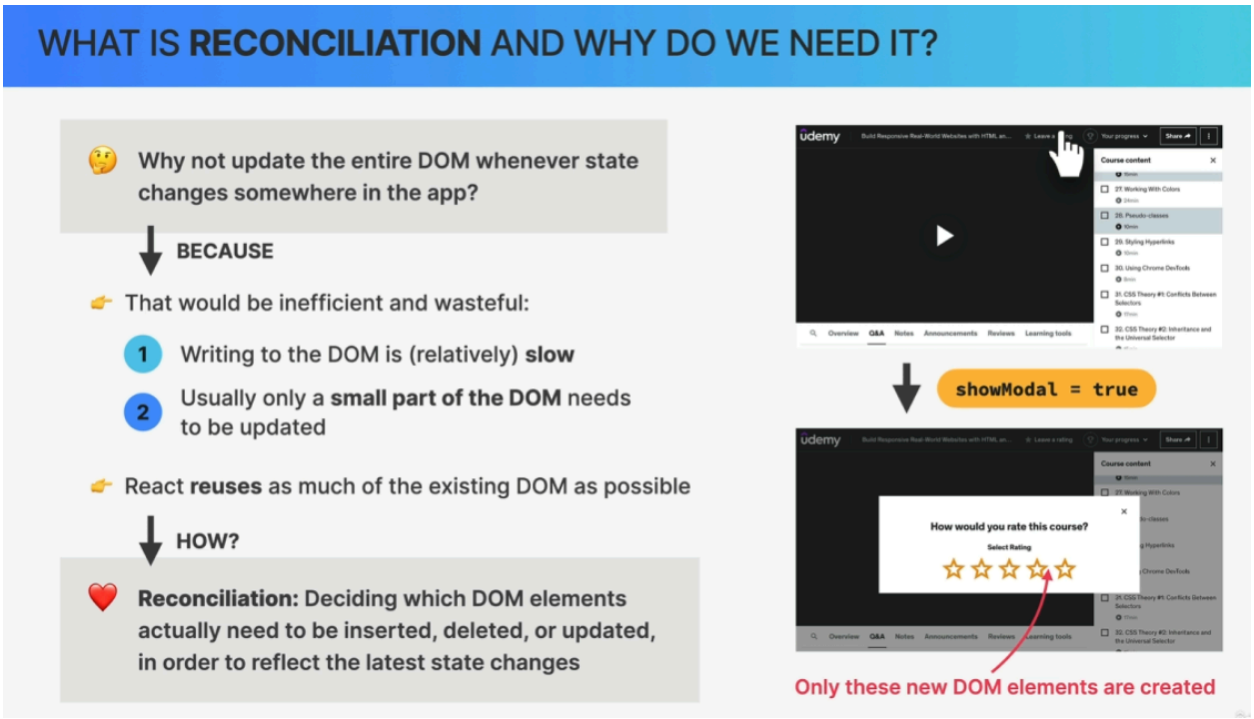


So what happens next is that this new virtual DOM that was created after the state update will get reconciled with the current so-called Fiber tree as it exists before the state update.

Now this reconciliation is done in React's reconciler which is called Fiber. Now that's why we have a Fiber tree.

Then the results of this reconciliation process is gonna be an updated Fiber tree, so a tree that will eventually be used to write to the DOM.

What is reconciliation and why do we need it?

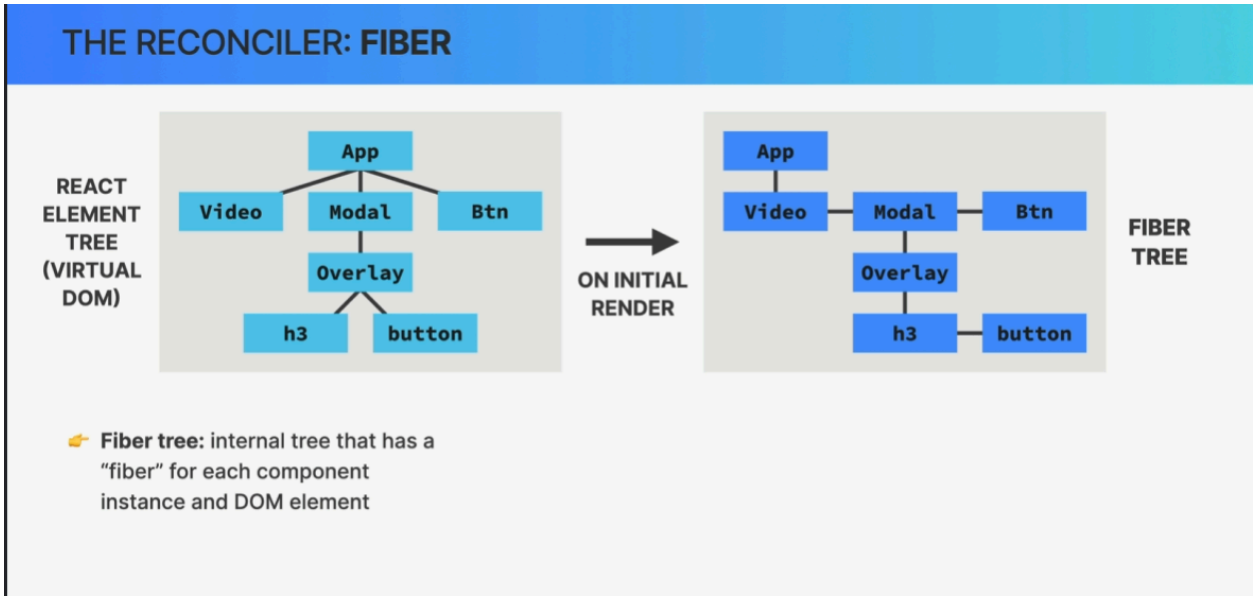


The Reconciler (called Fiber)

So, during the **initial render of the application Fiber takes the entire React element tree, so the virtual DOM, based on it builds yet another tree which is the Fiber tree.**

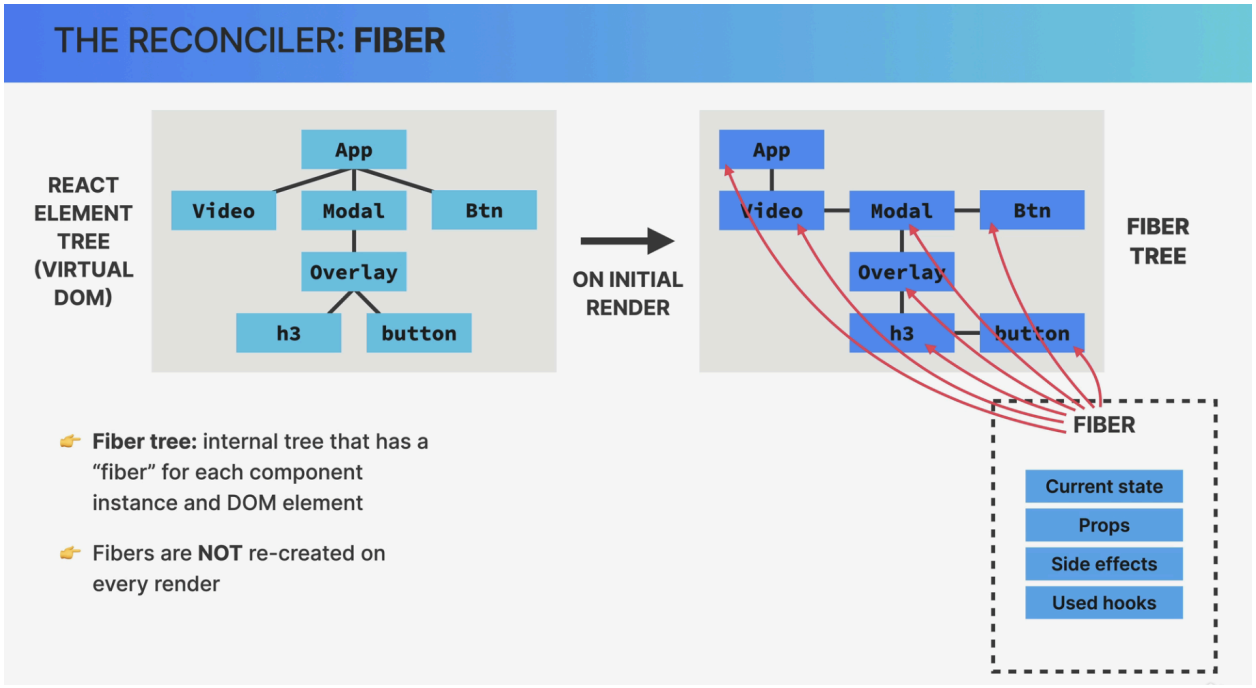
The Fiber tree is a special internal tree where for each component instance and DOM element in the app, there is one so-called Fiber.

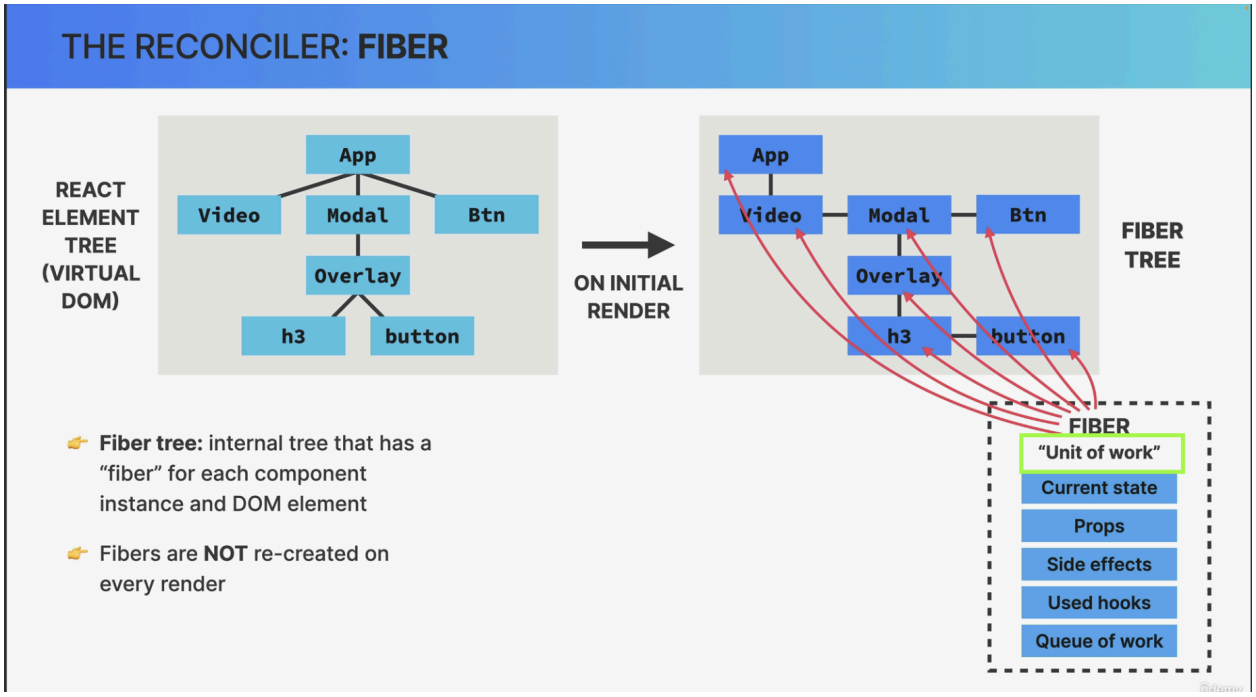
Now, what's **special about this tree is that, unlike React elements in the virtual DOM, Fibers are not recreated on every render. So the Fiber tree is never destroyed. Instead, it's a mutable data structure and once it has been created during the initial render, it's simply mutated over and over again in future reconciliation steps.**



This makes Fibers the perfect place for keeping track of things like the current component state, props, side effects, list of used hooks, and more.

So the actual state and props of any component instance that we see on the screen are internally stored inside the corresponding Fiber in the Fiber tree.





Now, each Fiber also contains a queue of work to do like updating state, updating refs, running registered side effects, performing DOM updates, and so on.

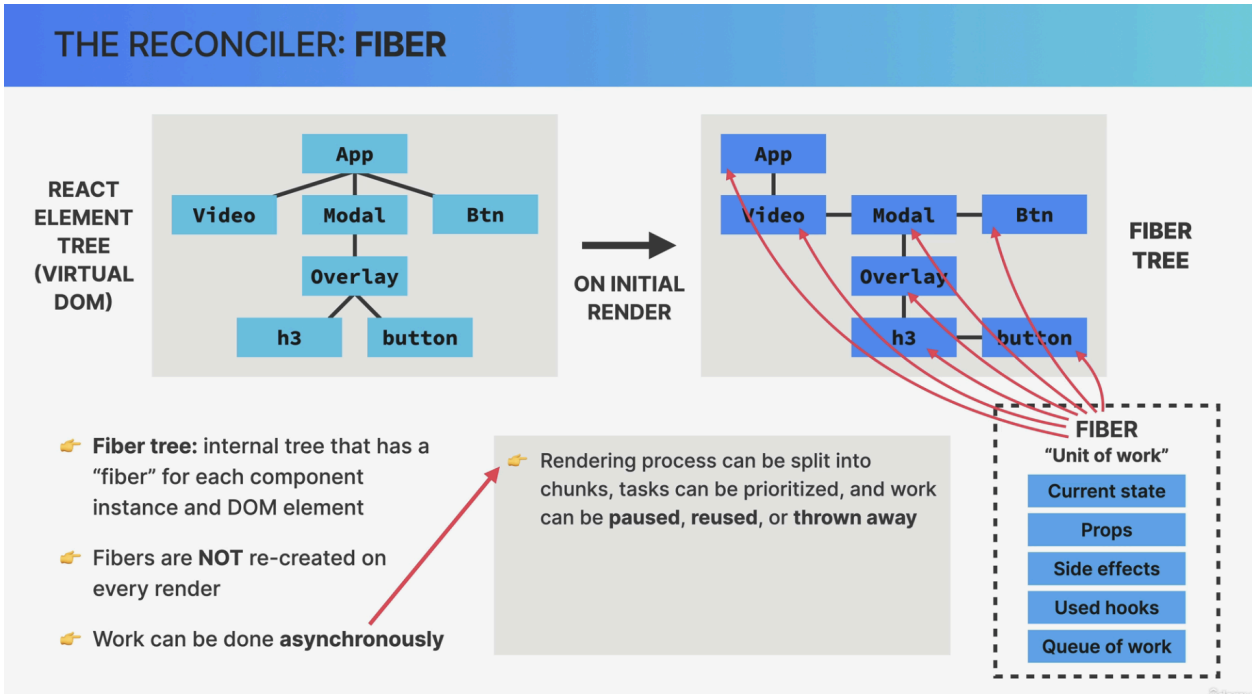
This is why a Fiber is also defined as a unit of work.

Now, if we take a quick look at the Fiber tree we will see that the Fibers are arranged in a different way than the elements in the React element tree.

So instead of a normal parent-child relationship, each first child has a link to its parent and all the other children then have a link to their previous sibling. This kind of structure is called a linked list and it makes it easier for React to process the work that is associated with each Fiber.

We also see that both trees include not only React elements or components but also regular DOM elements, such as the h3 and button elements in this example.

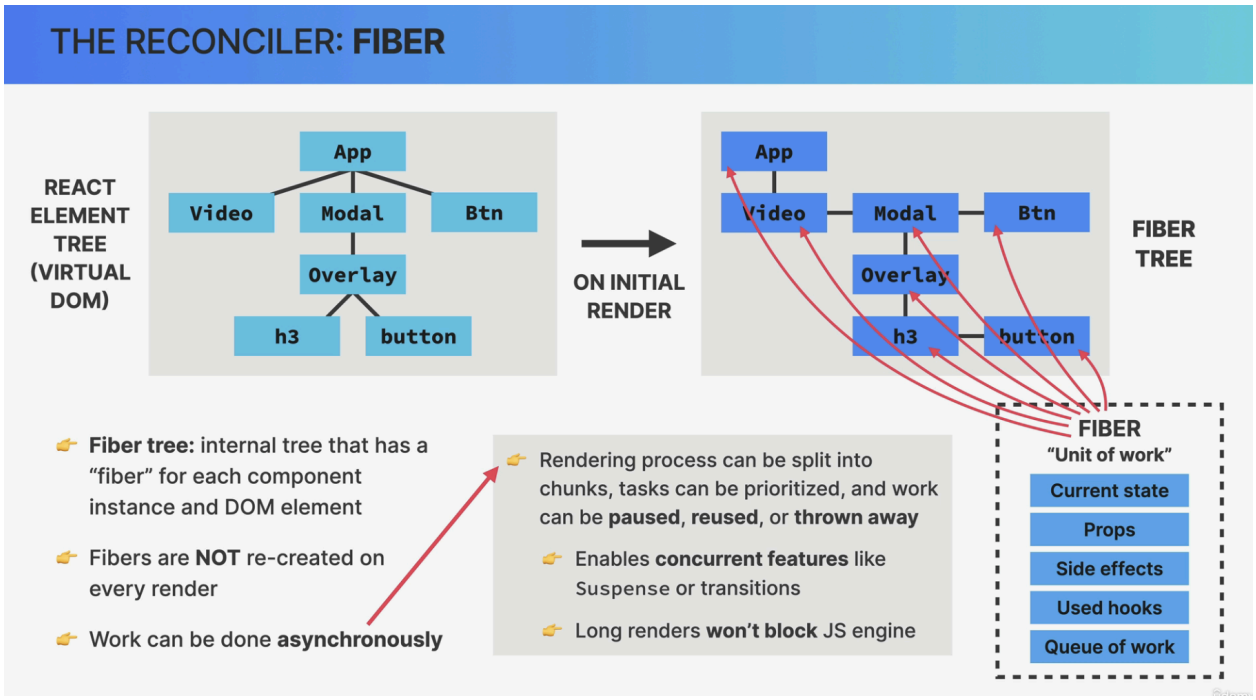
So both trees are a complete representation of the entire DOM structure, not just of React components.



Now going back to the idea that Fibers are units of work, one extremely important characteristic of the Fiber reconciler is that work can be performed asynchronously.

This means that the rendering process which is what the reconciler does, can be split into chunks, some tasks can be prioritized over others and work can be paused, reused, or thrown away if not valid anymore.

Just keep in mind that all this happens automatically behind the scenes.

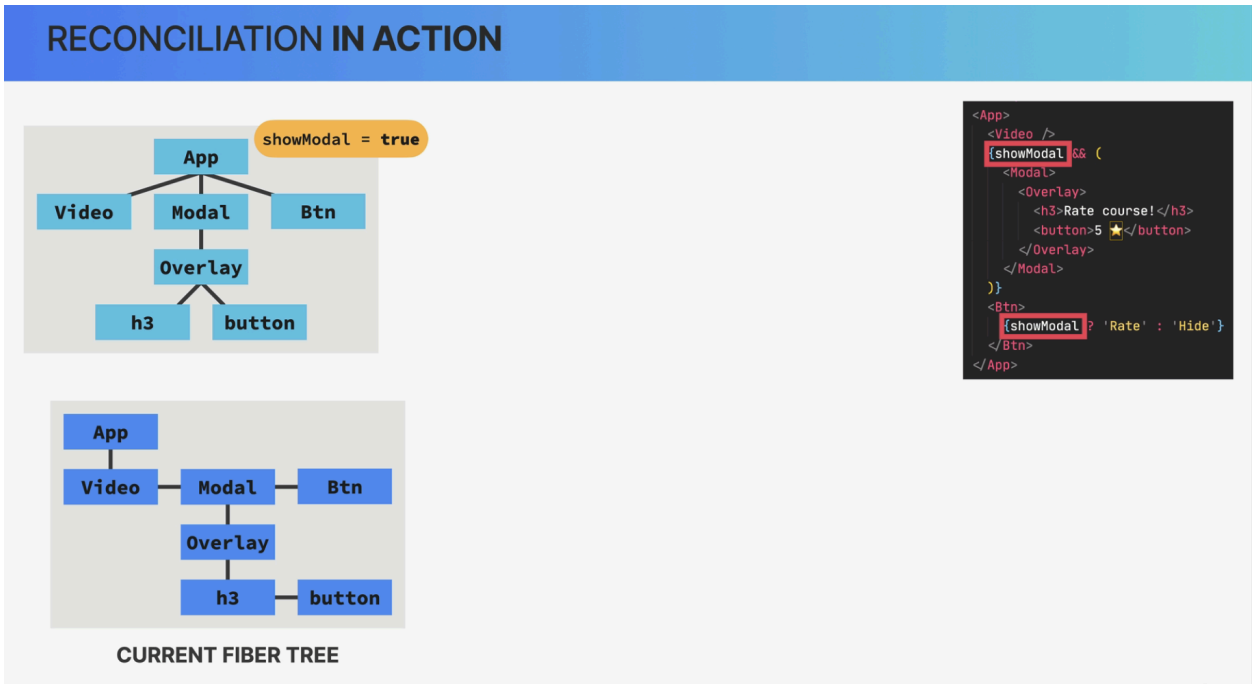


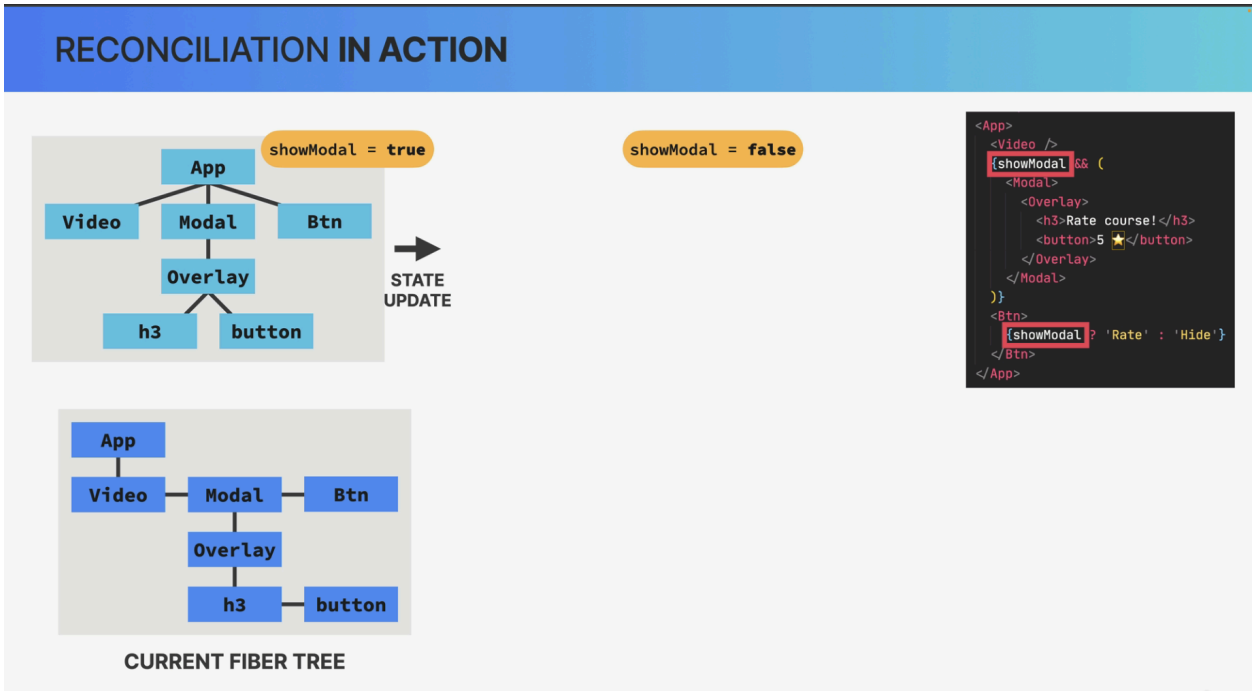
It's completely invisible to us developers. There are, however, also some practical uses of this asynchronous rendering because it enables modern so-called concurrent features like Suspense or transitions starting in React 18.

It also allows the rendering process to pause and resume later so that it won't block the browser's JavaScript engine with two long renders, which can be problematic for performance in large applications.

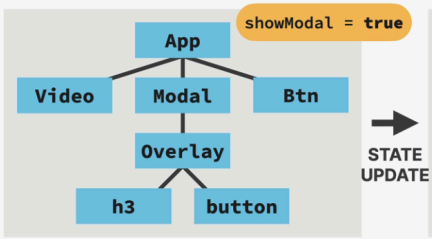
Again, this is only possible because the render phase does not produce any visible output to the DOM yet.

Reconciliation in action

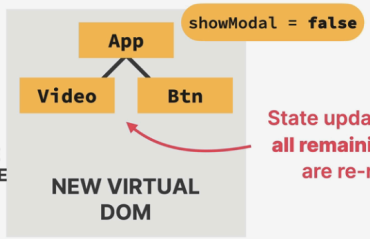




RECONCILIATION **IN ACTION**

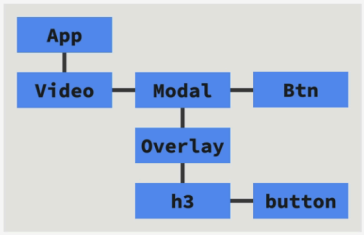


STATE UPDATE



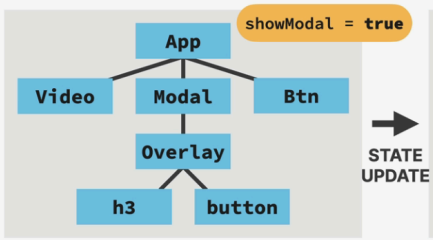
State update in App, so **all remaining children** are re-rendered

```
<App>
  <Video />
  {showModal && (
    <Modal>
      <Overlay>
        <h3>Rate course!</h3>
        <button>5 ⭐</button>
      </Overlay>
    </Modal>
  )}
  <Btn>
    {showModal ? 'Rate' : 'Hide'}
  </Btn>
</App>
```

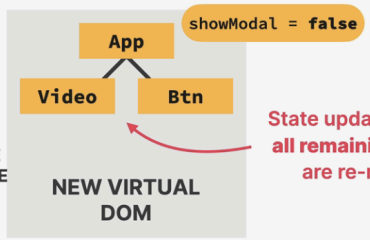


CURRENT FIBER TREE

RECONCILIATION **IN ACTION**

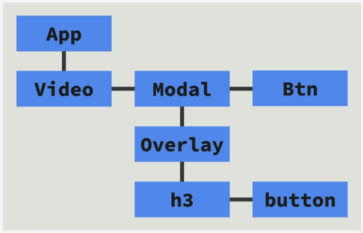


STATE UPDATE



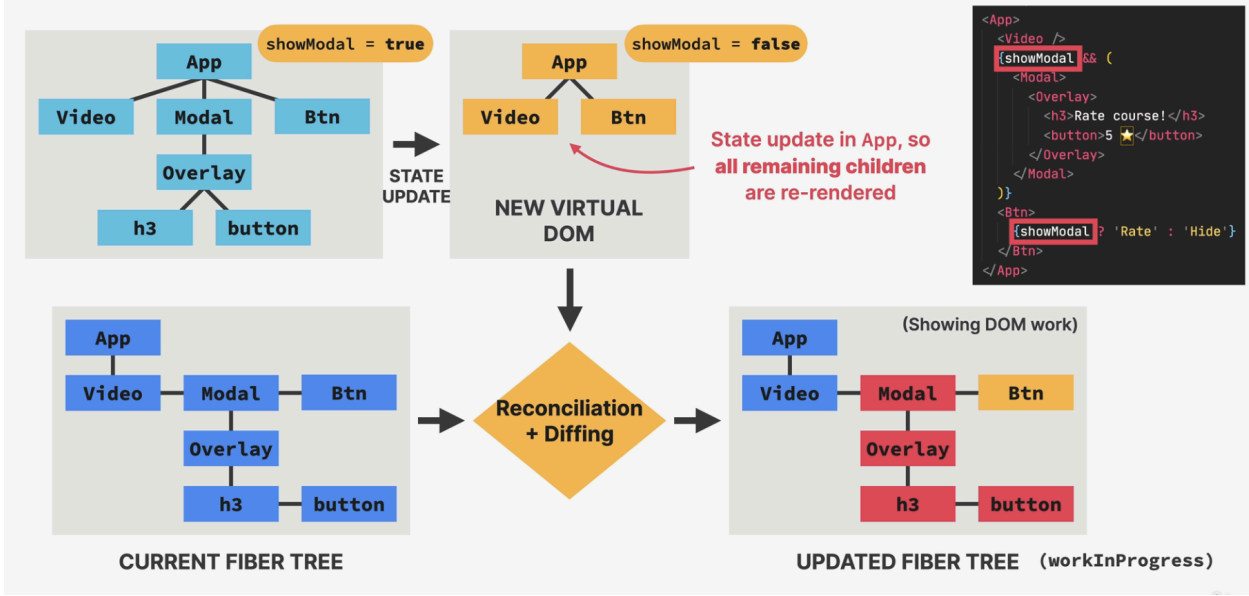
State update in App, so **all remaining children** are re-rendered

```
<App>
  <Video />
  {showModal && (
    <Modal>
      <Overlay>
        <h3>Rate course!</h3>
        <button>5 ⭐</button>
      </Overlay>
    </Modal>
  )}
  <Btn>
    {showModal ? 'Rate' : 'Hide'}
  </Btn>
</App>
```

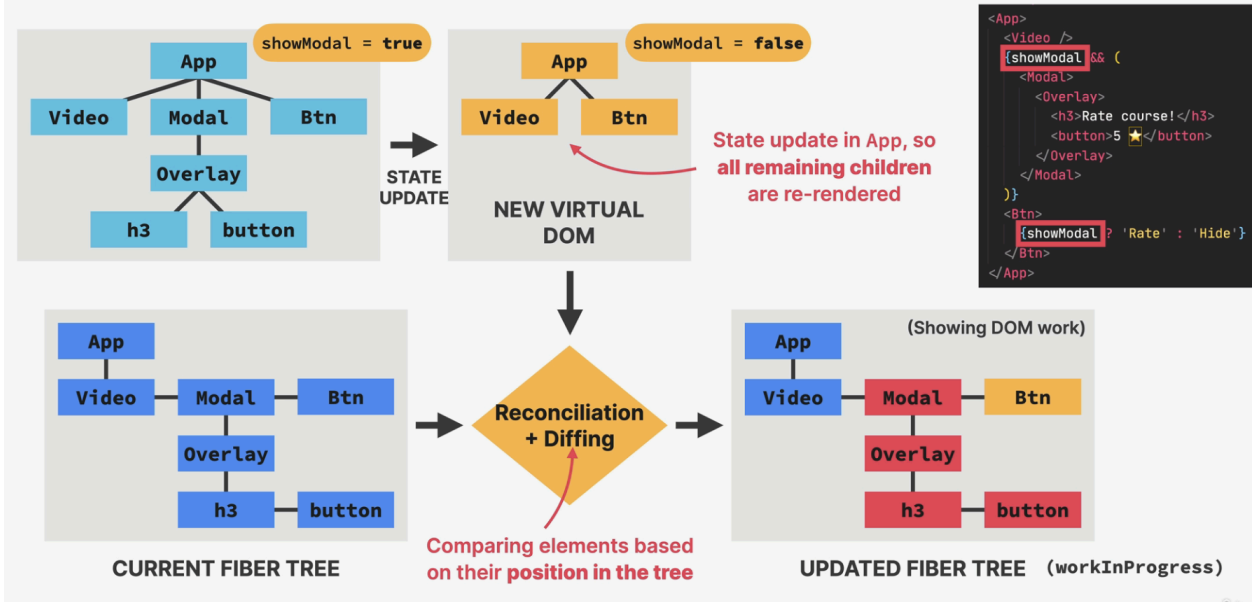


CURRENT FIBER TREE

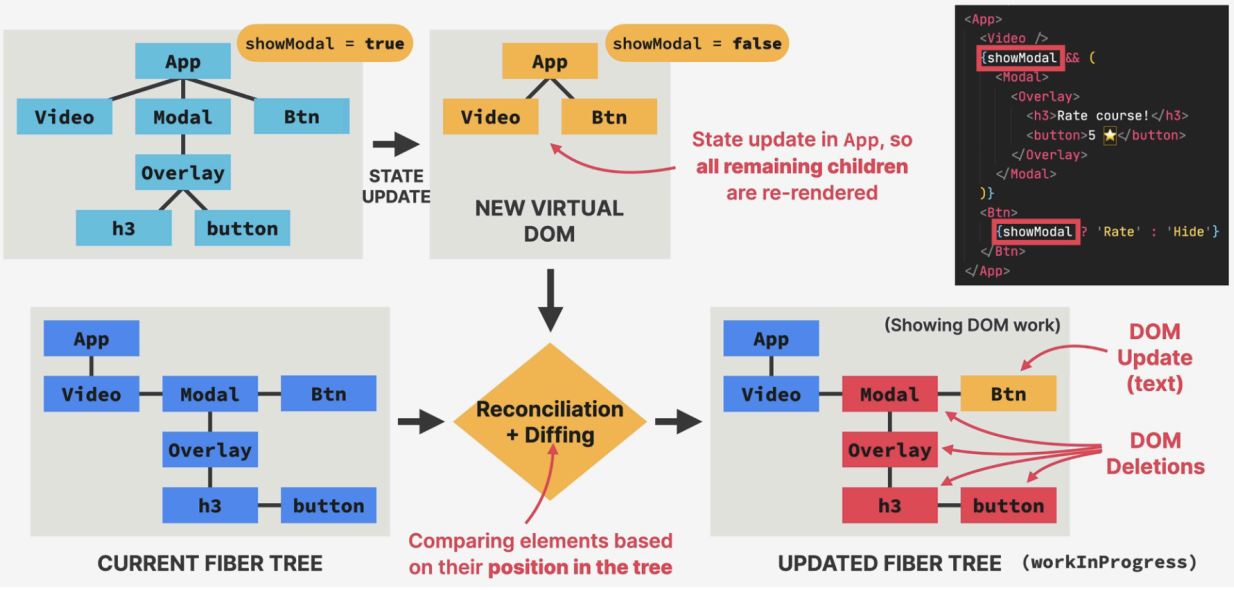
RECONCILIATION IN ACTION



RECONCILIATION IN ACTION



RECONCILIATION IN ACTION



THE RENDER PHASE