

# Section 1

## Component

### COMPONENTS AS BUILDING BLOCKS

#### COMPONENTS

- 👉 React applications are entirely made out of components
- 👉 **Building blocks** of user interfaces in React
- 👉 Piece of UI that has its own **data, logic, and appearance** (*how it works and looks*)
- 👉 We build complex UIs by **building multiple components** and **combining them**

#### VideoPlayer

Course content Overview Q&A Notes Announcements

Search all course questions

All lectures Sort by most recent Filter questions

All questions in this course (41683)

**Question 2 on Challenge 2**  
My solution was similar to Jonas' however the answer was incorrect. Is it po...  
Darryl - Lecture 115 - 13 hours ago  
0 0

**Elegant alternative for loading markers from localStorage**  
A Jonas explained, we are trying to add a marker to the map right at the beg...  
Vincent Giovanni - Lecture 242 - 14 hours ago  
0 0

**How to not violate the "Do not repeat yourself" principle**  
Hello! Could you please post a solution to this part, but in a way that we do n...  
Marinela - Lecture 45 - 15 hours ago  
0 1

### COMPONENTS AS BUILDING BLOCKS

#### COMPONENTS

- 👉 React applications are entirely made out of components
- 👉 **Building blocks** of user interfaces in React
- 👉 Piece of UI that has its own **data, logic, and appearance** (*how it works and looks*)
- 👉 We build complex UIs by **building multiple components** and **combining them**
- 👉 Components can be **reused, nested** inside each other, and **pass data** between them

#### VideoPlayer

Course content Overview Q&A Notes Announcements

Search all course questions

All lectures Sort by most recent Filter questions

RefineQuestions

All questions in this course (41683)

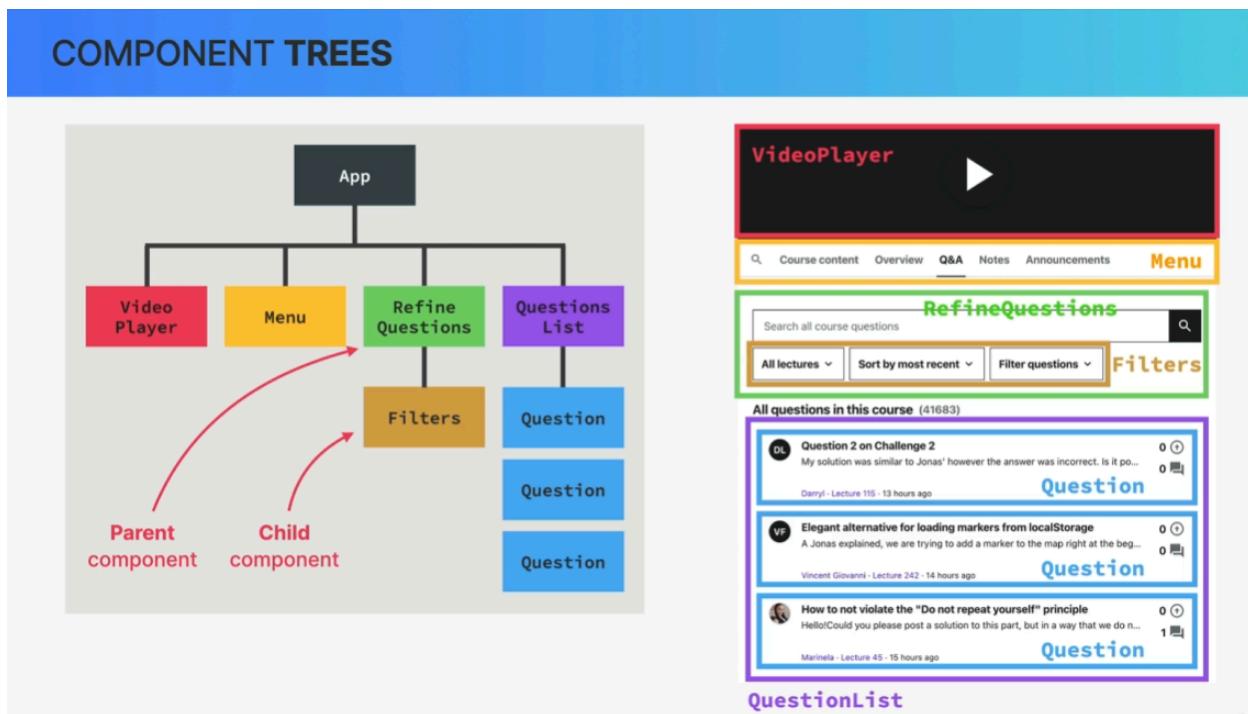
**Question 2 on Challenge 2**  
My solution was similar to Jonas' however the answer was incorrect. Is it po...  
Darryl - Lecture 115 - 13 hours ago  
0 0

**Elegant alternative for loading markers from localStorage**  
A Jonas explained, we are trying to add a marker to the map right at the beg...  
Vincent Giovanni - Lecture 242 - 14 hours ago  
0 0

**How to not violate the "Do not repeat yourself" principle**  
Hello! Could you please post a solution to this part, but in a way that we do n...  
Marinela - Lecture 45 - 15 hours ago  
0 1

#### QuestionList

## COMPONENT TREES



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.

## 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

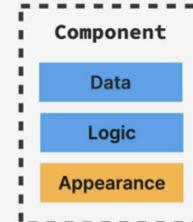
```
const title = document.querySelector("title")
const upvoteBtn = document.querySelector("btn")
title.textContent = "[0] ${question.title}"
let upvotes = 0;
upvoteBtn.addEventListener "click", function(){
  upvotes++;
  title.textContent =
    [${upvotes}] ${question.title};
  title.classList.add("upvoted");
}
```

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.

## WHAT IS JSX?

### JSX

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



- Components must return a block of JSX

## 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



- 👉 Describe what UI should look like using JSX, **based on current data**

```
function Question(props) {
  const question = props.question;
  const [upvotes, setUpVotes] = useState(0);
  const upvote = () => setUpVotes((v) => v + 1);

  return (
    <div>
      <h2>{question.title}</h2>
      <p>{question.text}</p>
      <button onClick={upvote}>upvote</button>
      <div>{upvotes}</div>
    </div>
  );
}
```

## WHAT IS JSX?

### JSX

- 👉 Declarative syntax to describe what components look like and how they work
- 👉 Components must return a block of JSX

```
function Question(props) {  
  const question = props.question;  
  const [upvotes, setUpvotes] = useState(0);  
  
  const upvote = () => setUpvotes((v) => v + 1);  
  
  const openQuestion = () => {}; // Todo  
  
  return (  
    <div>  
      <h4 style={{ fontSize: "2.4rem" }}>  
        {question.title}  
      </h4>  
      <p>{question.text}</p>  
      <p>{question.hours} hours ago</p>  
  
      <UpvoteBtn onClick={upvote} />  
      <Answers  
        numAnswers={question.num}  
        onClick={openQuestion}  
      />  
    </div>  
  );  
}
```

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

## WHAT IS JSX?

### JSX

- 👉 Declarative syntax to describe what components look like and how they work
- 👉 Components must return a block of JSX
- 👉 Extension of JavaScript that allows us to embed **JavaScript** **CSS** and React components into **HTML**

```
function Question(props) {  
  const question = props.question;  
  const [upvotes, setUpvotes] = useState(0);  
  
  const upvote = () => setUpvotes((v) => v + 1);  
  
  const openQuestion = () => {}; // Todo  
  
  return (  
    <div>  
      <h4 style={{ fontSize: "2.4rem" }}>  
        {question.title}  
      </h4>  
      <p>{question.text}</p>  
      <p>{question.hours} hours ago</p>  
  
      <UpvoteBtn onClick={upvote} />  
      <Answers  
        numAnswers={question.num}  
        onClick={openQuestion}  
      />  
    </div>  
  );  
}
```

## JSX

- 👉 Declarative syntax to describe what components look like and how they work
- 👉 Components must return a block of JSX
- 👉 Extension of JavaScript that allows us to embed **JavaScript**, **CSS**, and **React components** into **HTML**

```
function Question(props) {
  const question = props.question;
  const [upvotes, setUpvotes] = useState(0);

  const upvote = () => setUpvotes((v) => v + 1);

  const openQuestion = () => {};
  // Todo

  return (
    <div>
      <h4 style={{ font-size: "2.4rem" }}>
        {question.title}
      </h4>
      <p>{question.text}</p>
      <p>{question.hours} hours ago</p>

      <UpvoteBtn onClick={upvote} />
      <Answers
        numAnswers={question.num}
        onClick={openQuestion}
      />
    </div>
  );
}
```

JSX returned from component

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

```
<header>
  <h1 style="color: red">
    Hello React!
  </h1>
</header>
```

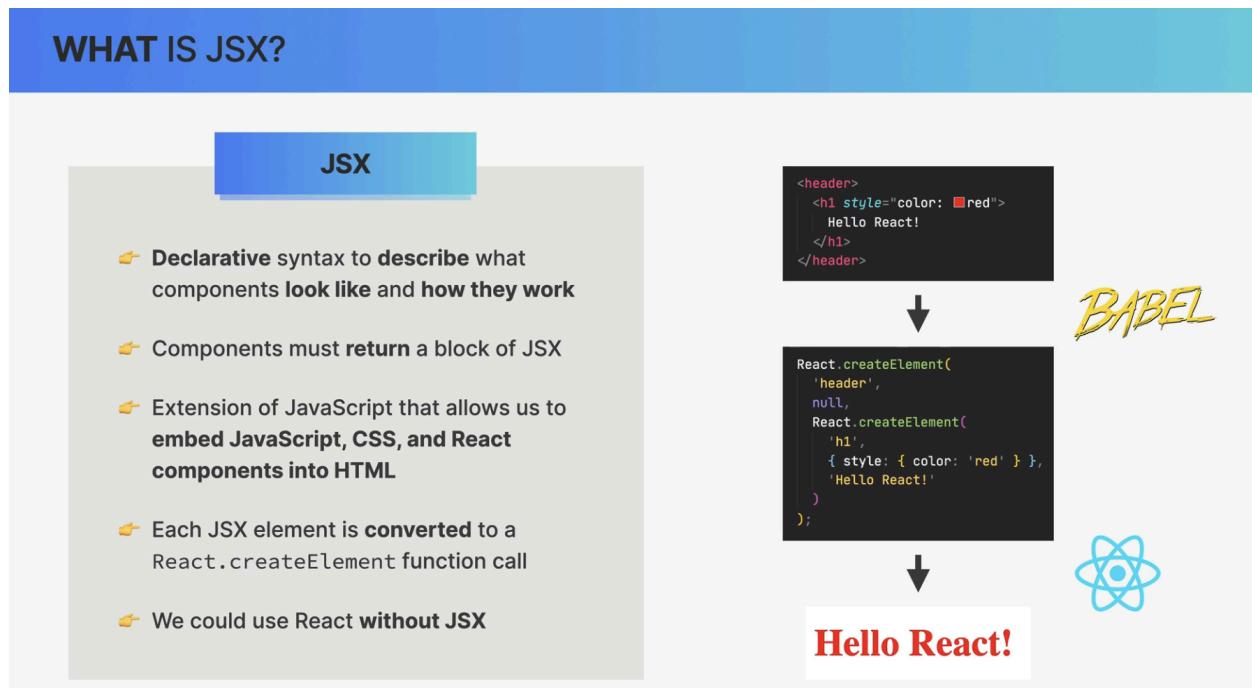


BABEL

```
React.createElement(
  'header',
  null,
  React.createElement(
    'h1',
    { style: { color: 'red' } },
    'Hello React!'
  )
);
```

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



```
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 (
    <div>
      <h4>question.title</h4>
      <p>question.text</p>
      <UpvoteBtn
        onClick={upvote}
        upvotes={upvotes}
      />
    </div>
  );
}
```

Odemy

## 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**  
  - 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 <br />
- 👉 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)

Odemy

## 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

We can place other pieces of JSX inside {}

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 <br />

👉 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

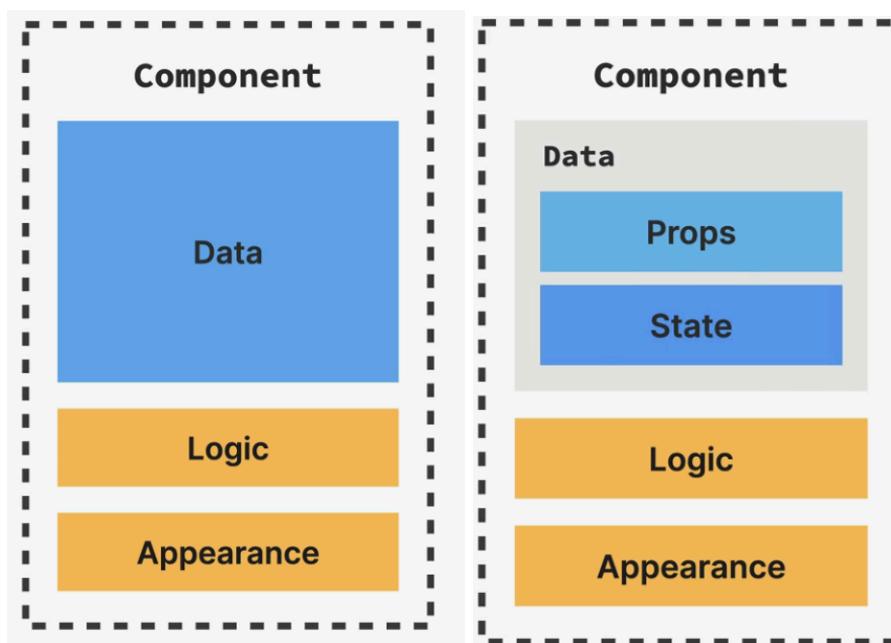
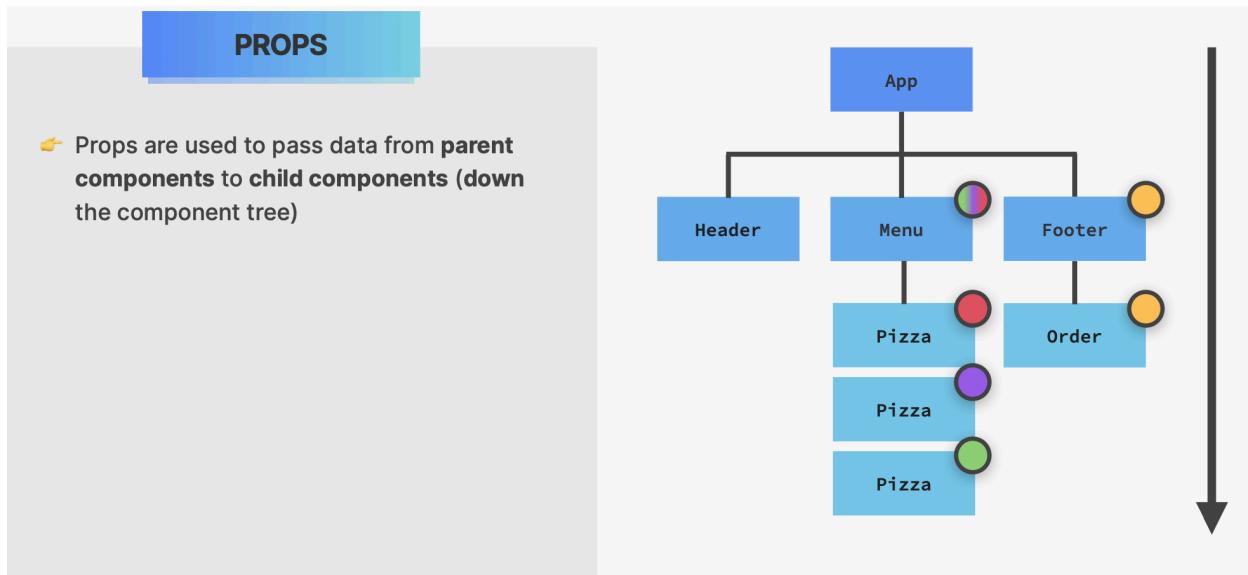
👉 CSS inline styles are written like this: {{<style>}}  
(to reference a variable, and then an object)

👉 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.

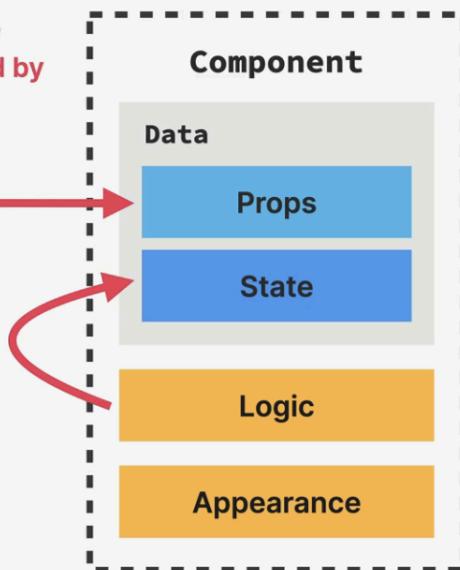


# PROPS ARE READ-ONLY!

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

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



## 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

```
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

```
let x = 7;

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

Component

Data

Props

State

Logic

Appearance

Don't do this!

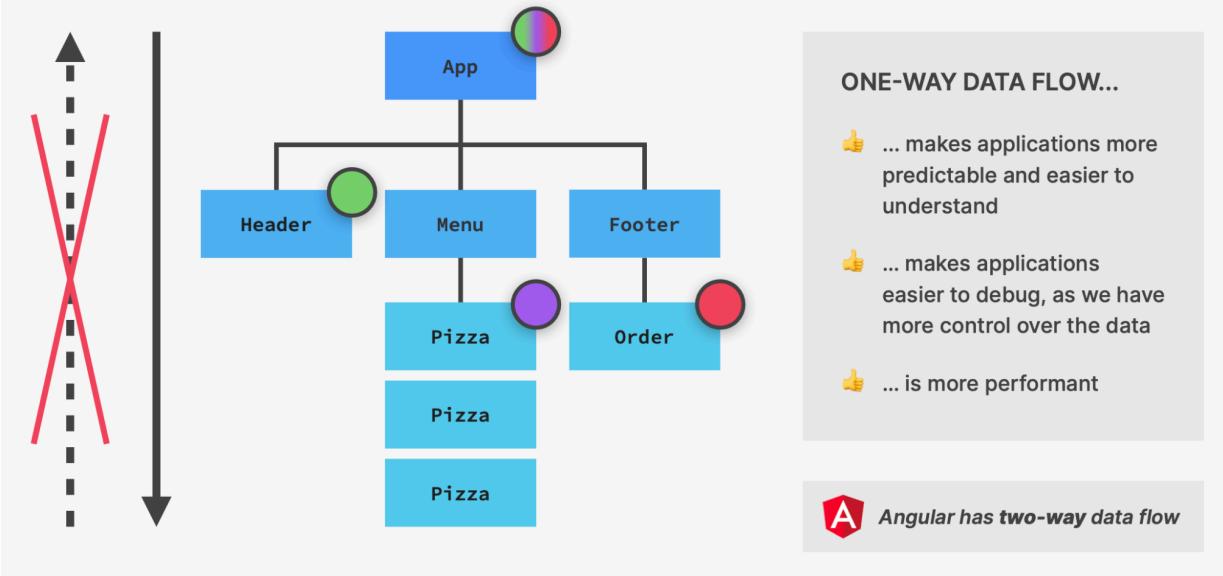
👉 Props are read-only, they are **immutable**! This is one of React's strict rules.

👉 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

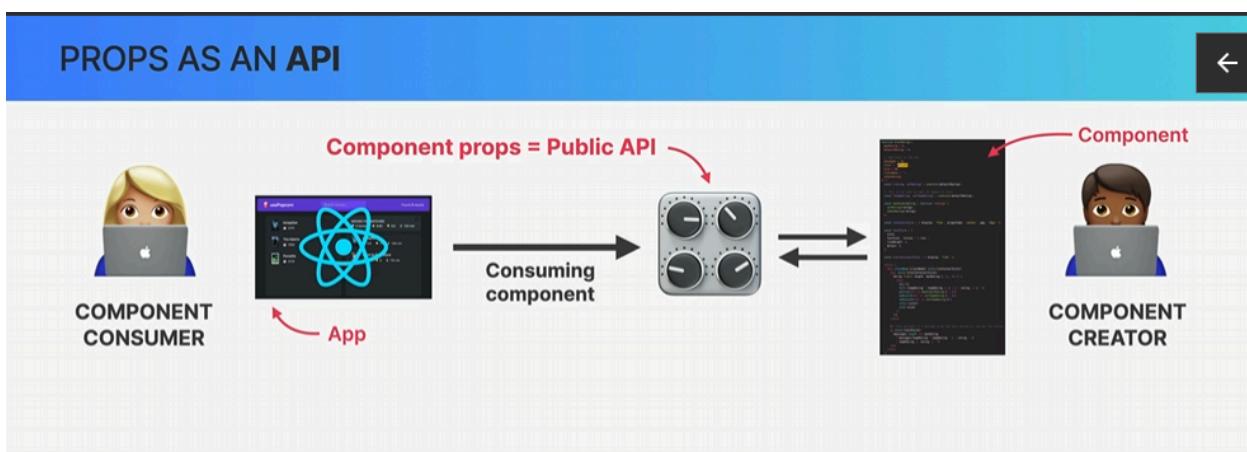
## ONE-WAY DATA FLOW



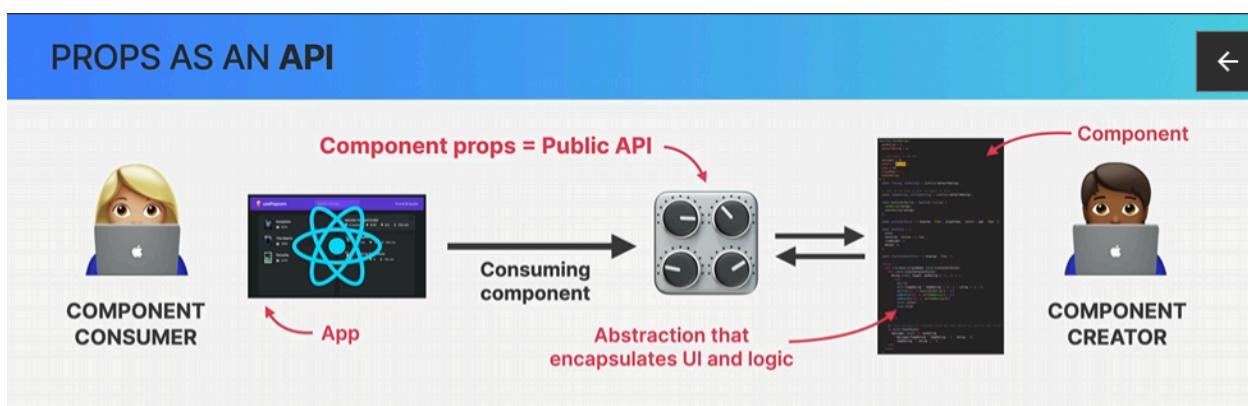
### Props drilling

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.

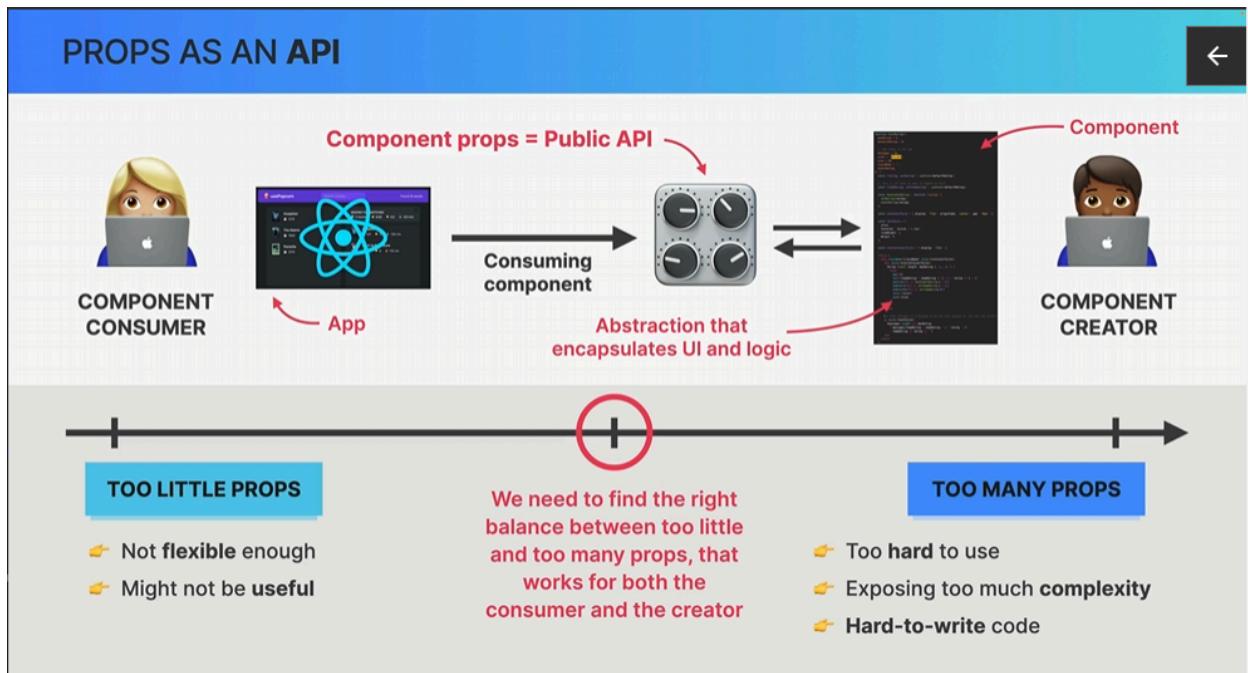
### Props as an API



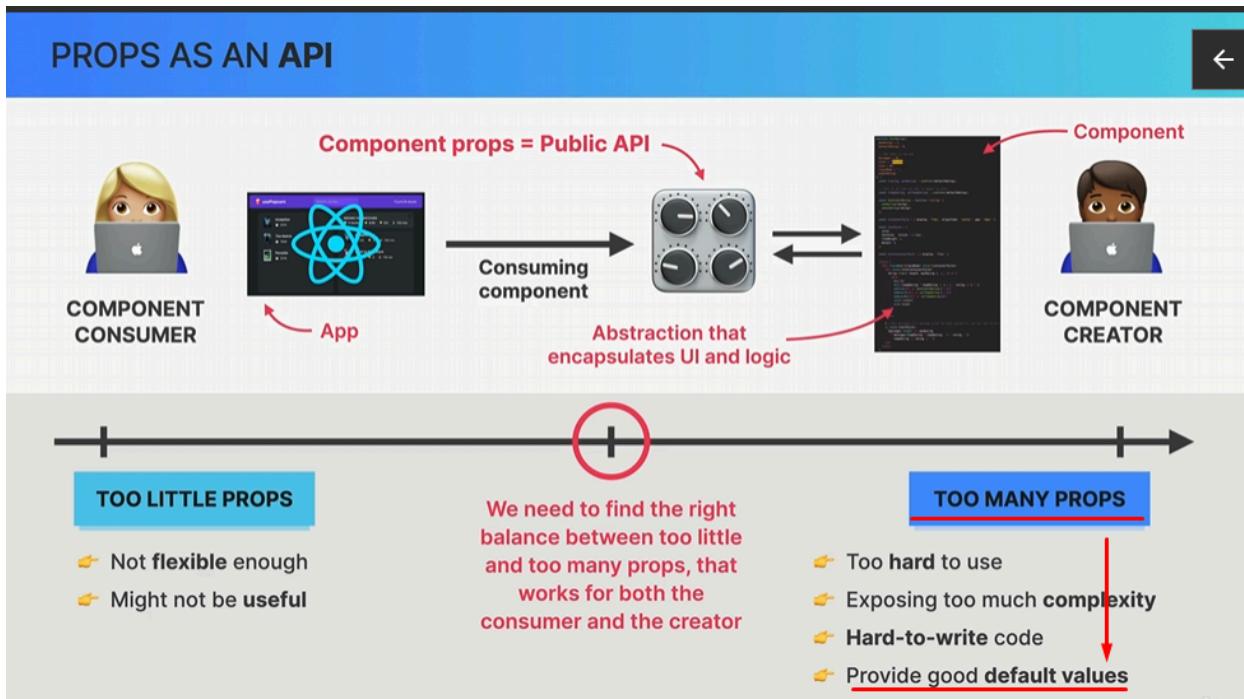
## PROPS AS AN API



## PROPS AS AN API



## PROPS AS AN API



## 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 mozzarella",  
    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>
  );
}
```

— FAST REACT PIZZA CO. —

---

OUR MENU

 Focaccia <i>Bread with italian olive oil and rosemary</i> 6	 Pizza Margherita <i>Tomato and mozzarella</i> 10
 Pizza Spinaci <i>Tomato, mozzarella, spinach, and ricotta cheese</i> 12	 Pizza Funghi <i>Tomato, mozzarella, mushrooms, and onion</i> 12
 Pizza Salamino <i>Tomato, mozzarella, and pepperoni</i> 15	 Pizza Prosciutto <i>Tomato, mozzarella, ham, aragula, and burrata cheese</i> 18

12:52:32 PM We're currently open!

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

```
{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>
  );
}
```

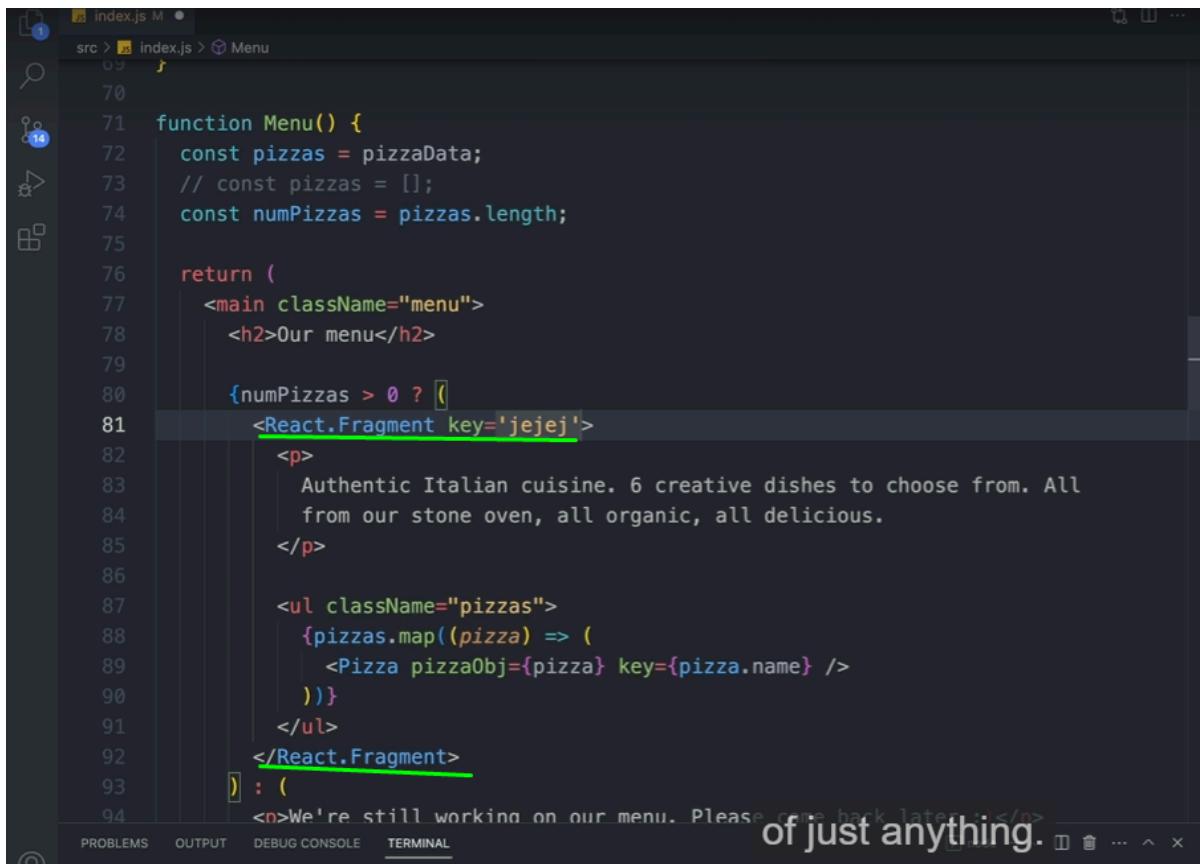
## React Fragments

```
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.
        </p>
        <ul className="pizzas">
          {pizzaData.map((pizza) => (
            <Pizza key={pizza.name} pizzaObj={pizza} />
          ))
        </ul>
      ) : (
        <p>We're temporarily closed! Come back soon!</p>
      )}
    </main>
  );
}
```

```
        ))}
      </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

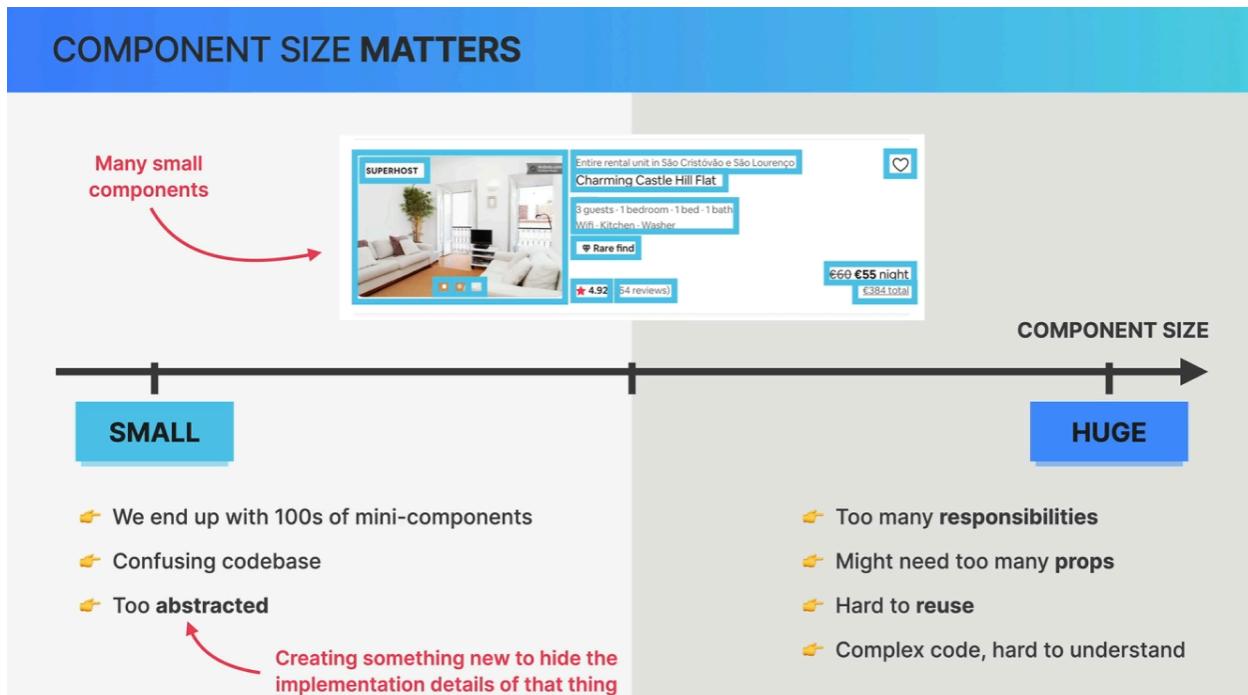
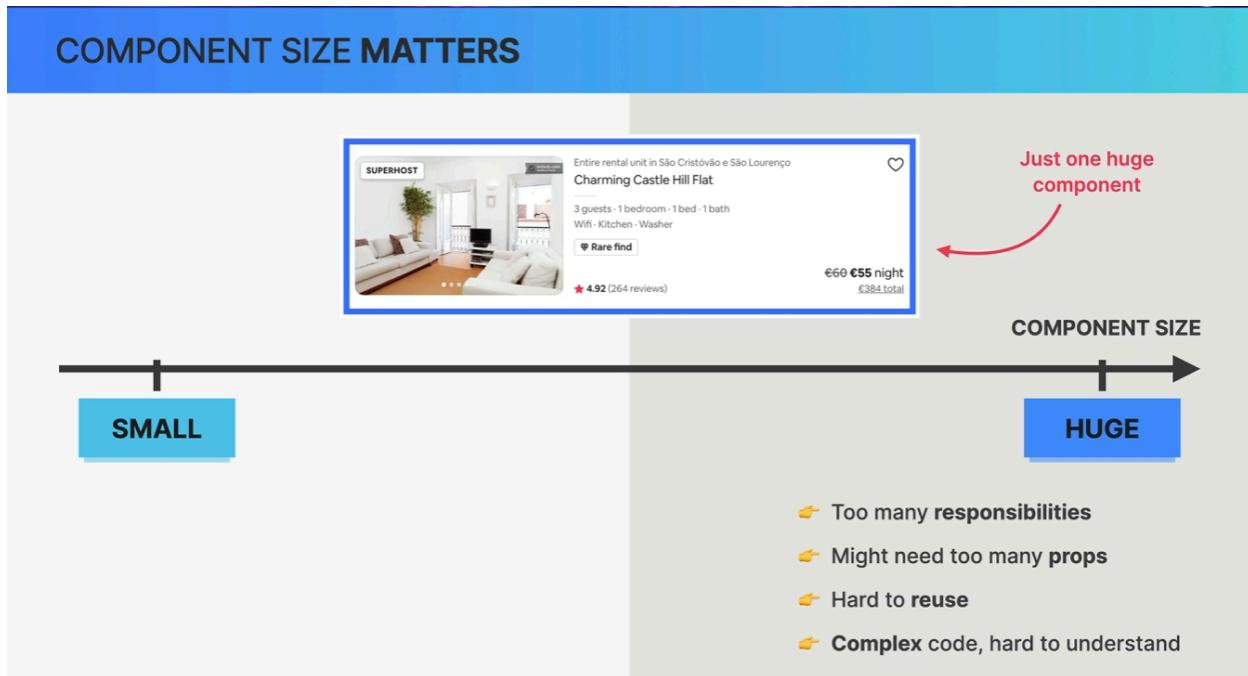


The screenshot shows a code editor window with the file 'index.js' open. The code defines a 'Menu' function that returns a main component containing an h2 and a fragment. The fragment has a key attribute set to 'jejej'. The code editor interface includes a sidebar with icons for search, refresh, and navigation, and a bottom navigation bar with tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, and TERMINAL.

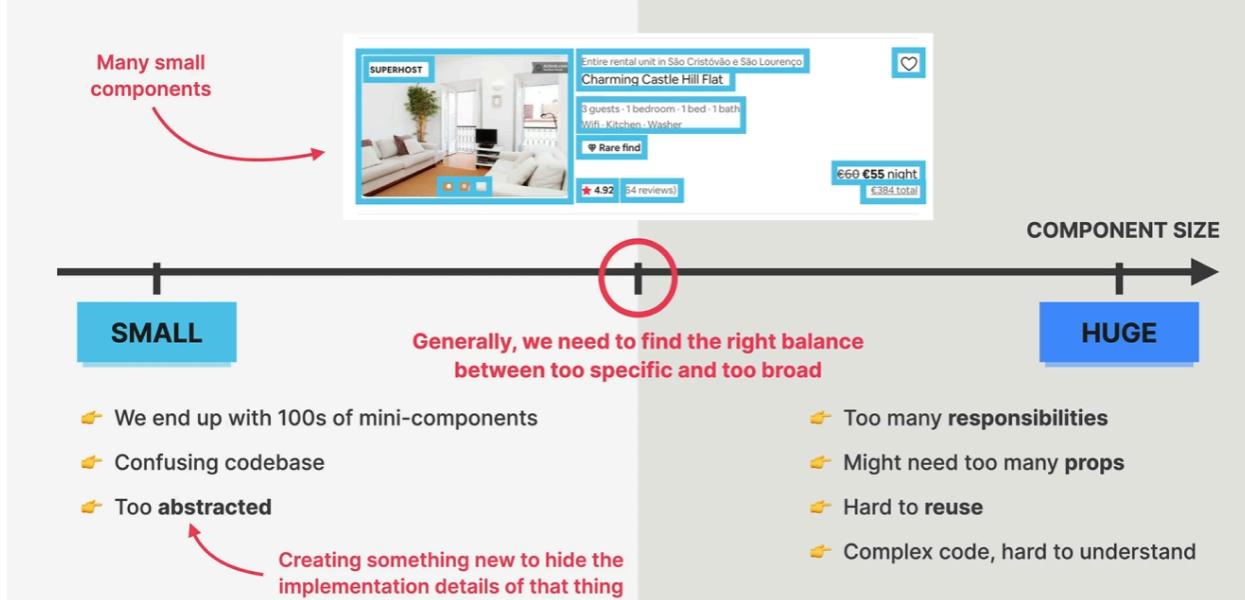
```
src > index.js > Menu
  70
  71  function Menu() {
  72    const pizzas = pizzaData;
  73    // const pizzas = [];
  74    const numPizzas = pizzas.length;
  75
  76    return (
  77      <main className="menu">
  78        <h2>Our menu</h2>
  79
  80        {numPizzas > 0 ? [
  81          <React.Fragment key='jejej'>
  82            <p>
  83              Authentic Italian cuisine. 6 creative dishes to choose from. All
  84              from our stone oven, all organic, all delicious.
  85            </p>
  86
  87            <ul className="pizzas">
  88              {pizzas.map((pizza) => (
  89                <Pizza pizzaObj={pizza} key={pizza.name} />
  90              )));
  91            </ul>
  92            </React.Fragment>
  93        ] : (
  94          <p>We're still working on our menu. Please come back later.</p>
        )
      )
    )
  }
}

of just anything.
```

## Component Size Matters

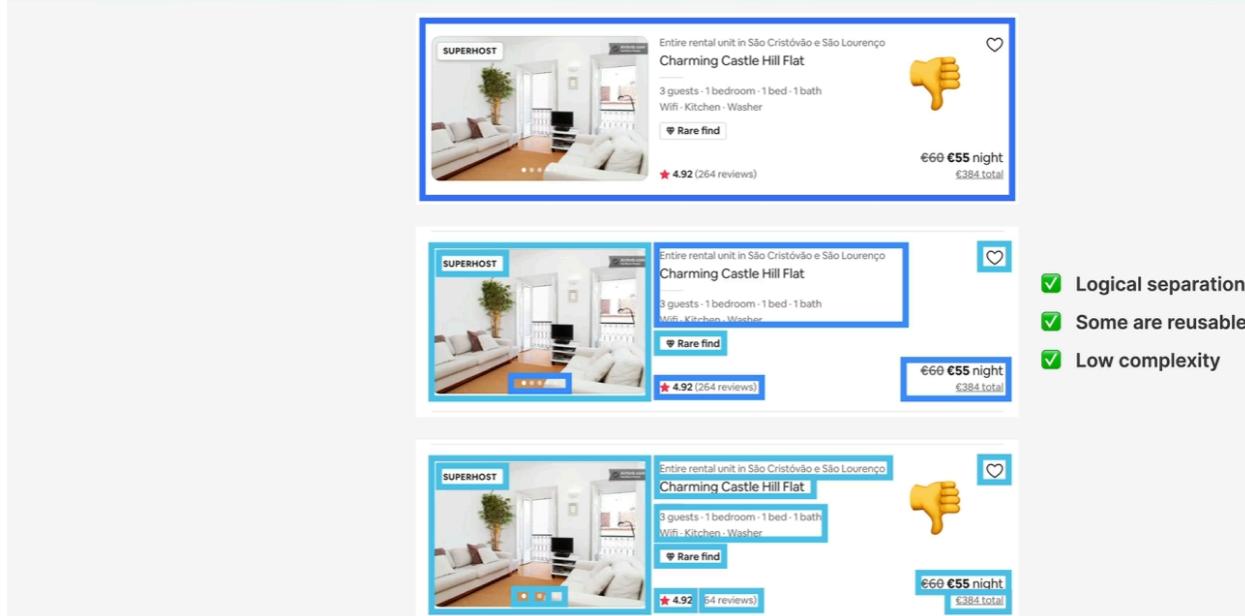


## COMPONENT SIZE MATTERS



How to split UI into components

## HOW TO SPLIT A UI INTO COMPONENTS



## HOW TO SPLIT A UI INTO COMPONENTS

👉 The 4 criteria for splitting a UI into components:

1. Logical separation of content/layout



2. Reusability



- ✓ Logical separation
- ✓ Some are reusable
- ✓ Low complexity

3. Responsibilities / complexity



4. Personal coding style

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

2. Reusability

3. Responsibilities / complexity

4. Personal coding style

## 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!

©demy

## 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**!

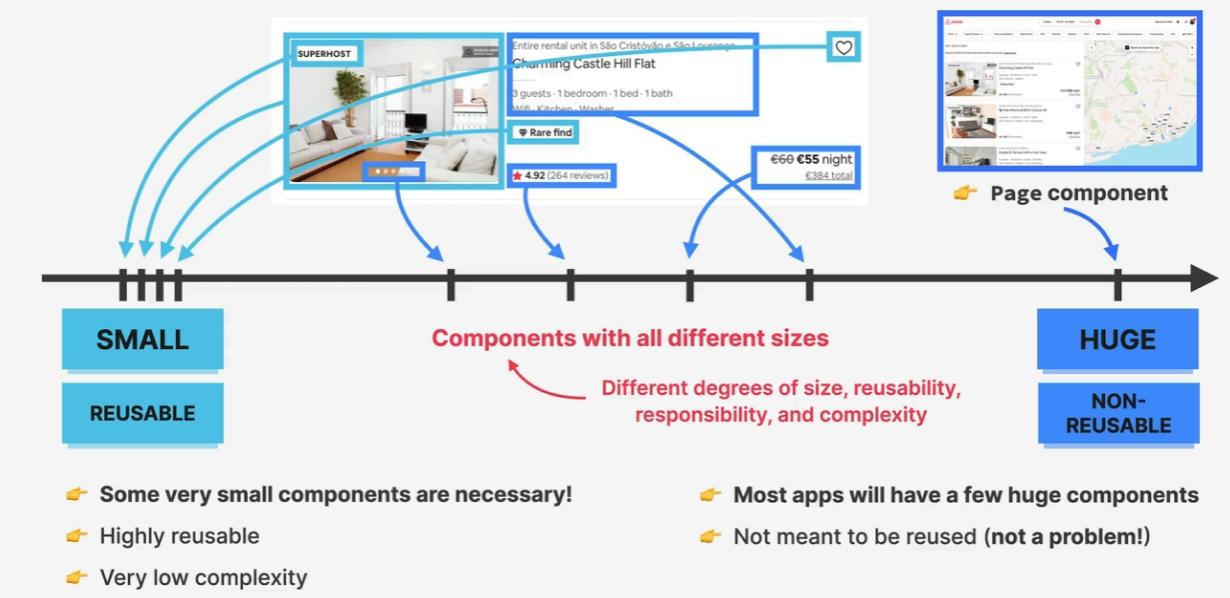


**Co-locate related components inside the same file.** Don't separate components into different files too early



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

## ANY APP HAS COMPONENTS OF DIFFERENT SIZES AND REUSABILITY



## Component Categories

### COMPONENT CATEGORIES

👉 Most of your components will naturally fall into **one of three categories**:

Stateless /  
presentational  
components

Stateful  
components

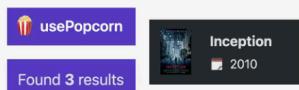
Structural  
components

## COMPONENT CATEGORIES

👉 Most of your components will naturally fall into **one of three categories**:

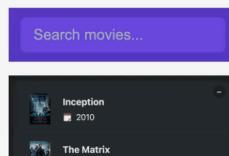
### Stateless / presentational components

- 👉 **No state**
- 👉 Can receive props and simply *present* received data or other content
- 👉 Usually **small and reusable**



### Stateful components

- 👉 **Have state**
- 👉 Can still be **reusable**



### Structural components

- 👉 "Pages", "layouts", or "screens" of the app
- 👉 Result of **composition**
- 👉 Can be **huge and non-reusable** (but don't have to)

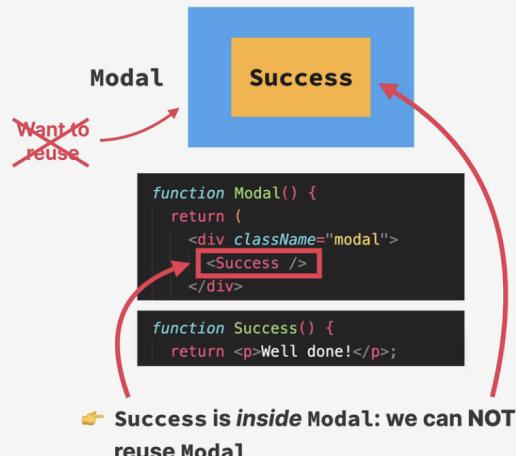


## Component Composition

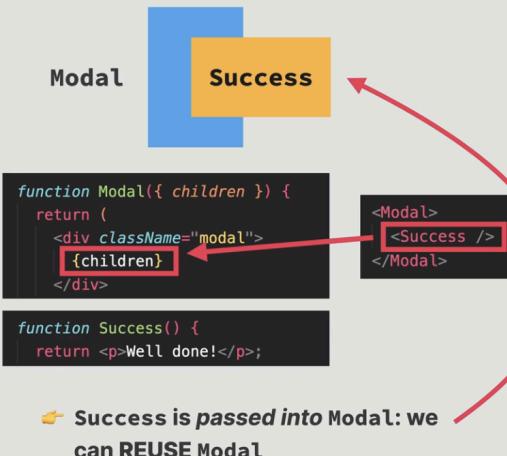
Combining different components using the `children` prop (or explicitly defined props)

## WHAT IS COMPONENT COMPOSITION?

### "USING" A COMPONENT

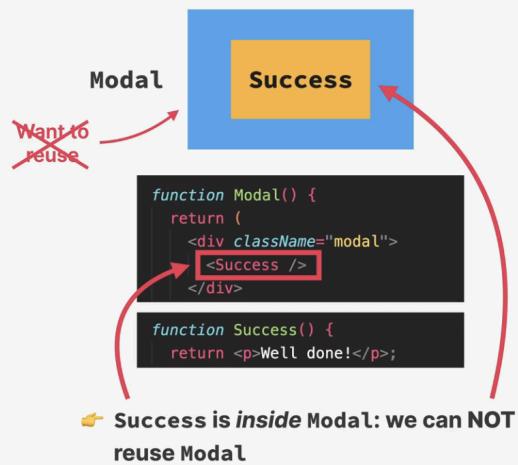


### COMPONENT COMPOSITION

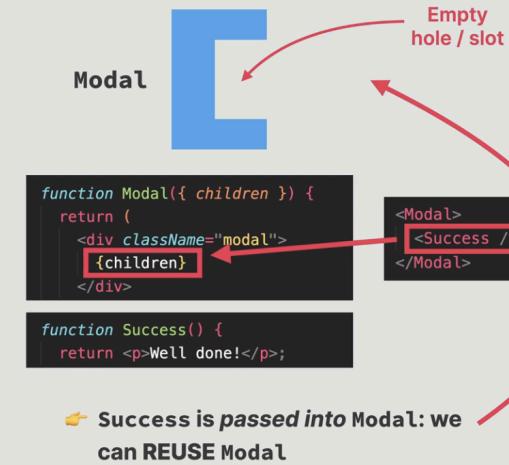


## WHAT IS COMPONENT COMPOSITION?

### "USING" A COMPONENT



### COMPONENT COMPOSITION

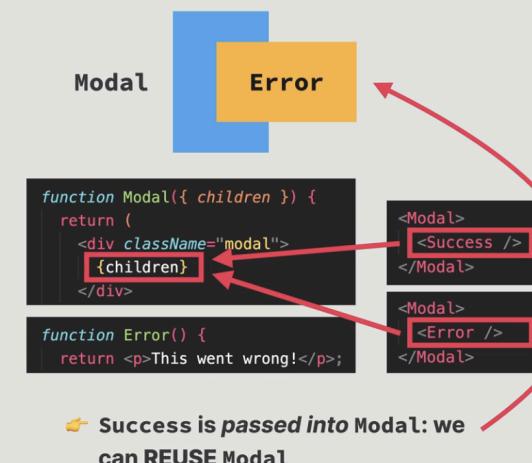


## WHAT IS COMPONENT COMPOSITION?

### "USING" A COMPONENT



### COMPONENT COMPOSITION



## WHAT IS COMPONENT COMPOSITION?

### COMPONENT COMPOSITION

Modal

Error

```
function Modal({ children }) {  
  return (  
    <div className="modal">  
      {children}  
    </div>  
  );  
  
  function Error() {  
    return <p>This went wrong!</p>;  
  }  
}
```

```
<Modal>  
  <Success />  
</Modal>  
  
<Modal>  
  <Error />  
</Modal>
```

👉 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

Modal

Error

```
function Modal({ children }) {  
  return (  
    <div className="modal">  
      {children}  
    </div>  
  );  
  
  function Error() {  
    return <p>This went wrong!</p>;  
  }  
}
```

```
<Modal>  
  <Success />  
</Modal>  
  
<Modal>  
  <Error />  
</Modal>
```

👉 Success is passed into Modal: we can REUSE Modal

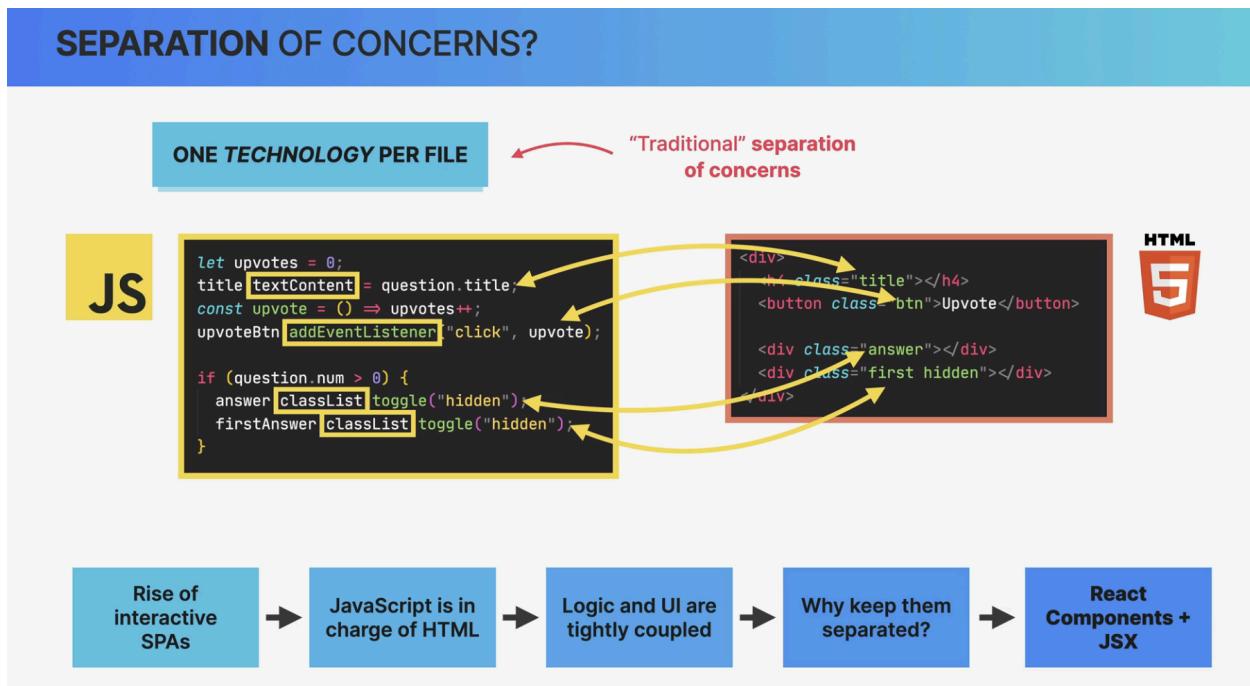
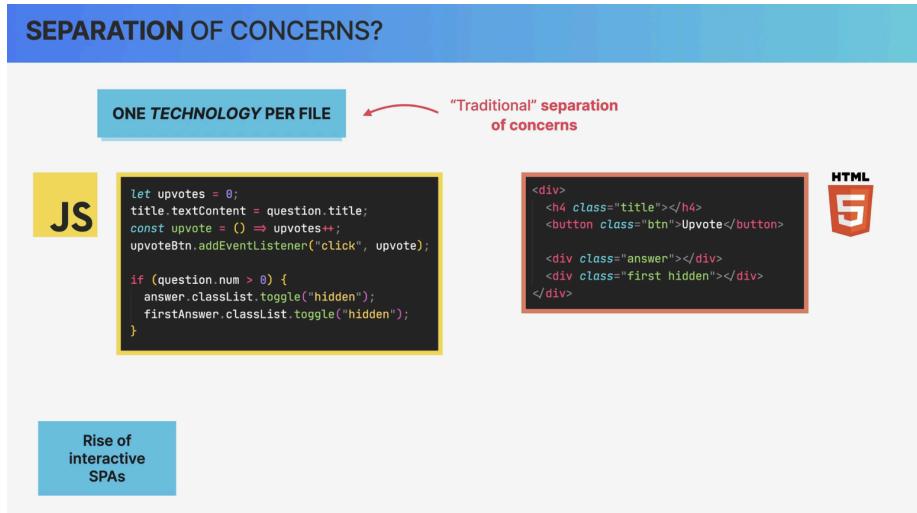
👉 Component composition: combining different components using the `children` prop (or explicitly defined props)

### WE CAN USE COMPONENT COMPOSITION, WE CAN:

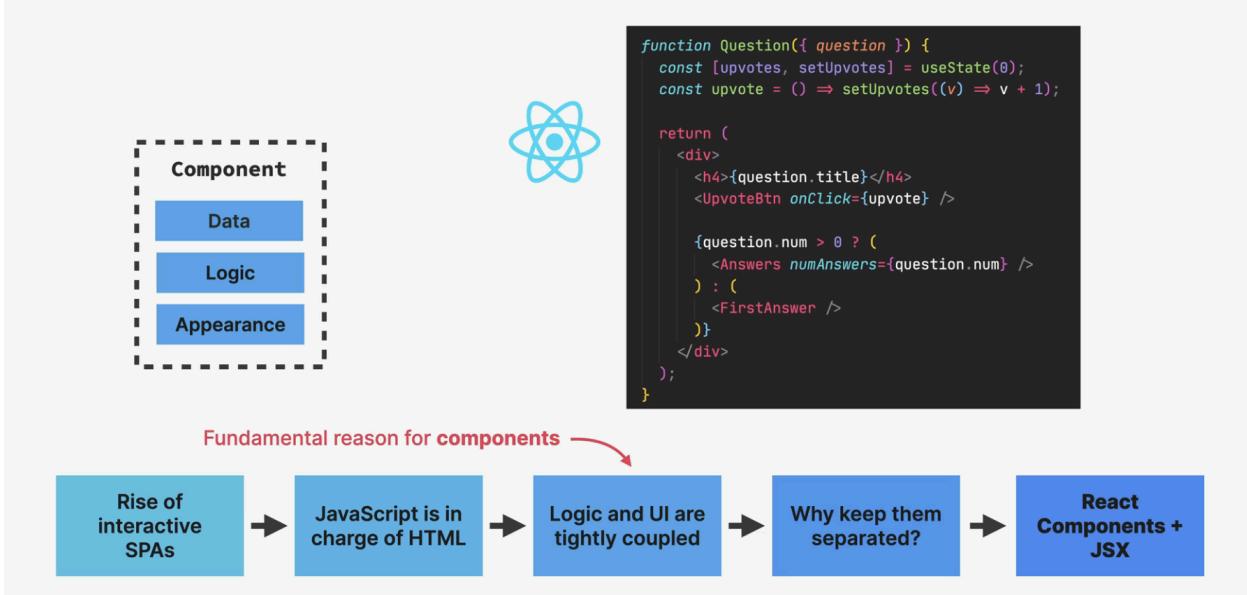
- 1 Create highly reusable and flexible components
- 2 Fix prop drilling (great for layouts)

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

## Separation of concerns



## SEPARATION OF CONCERNS?

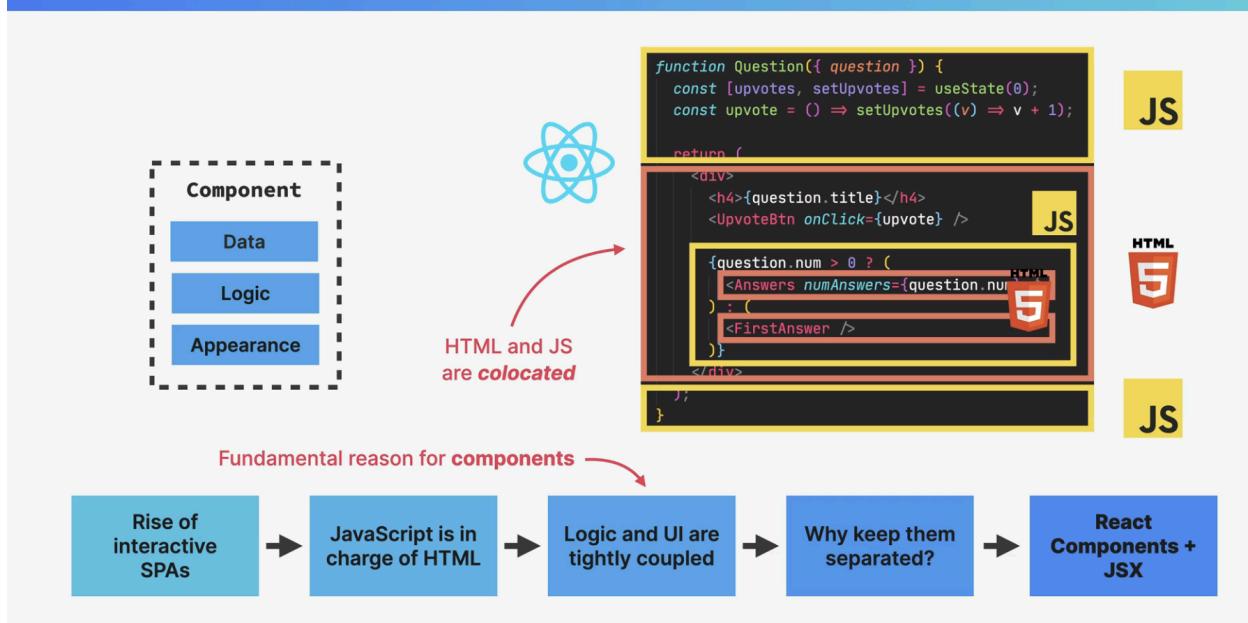


So content and logic are tightly coupled together and so it makes sense that they are co-located here.

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.

## SEPARATION OF CONCERNS?



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

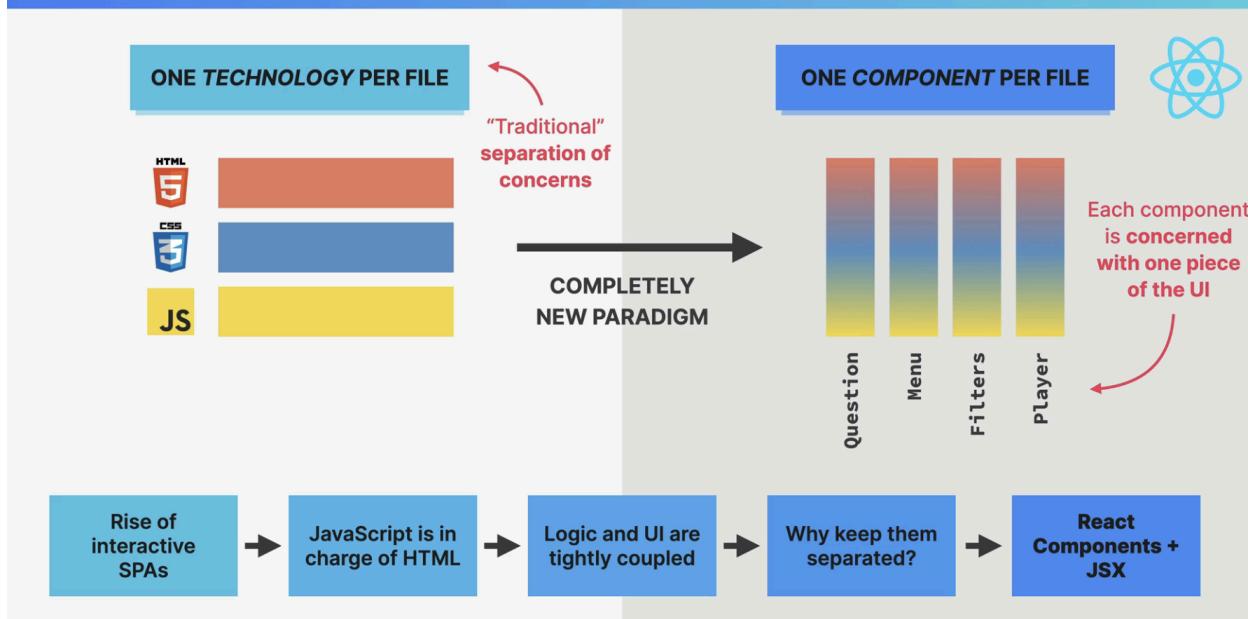
Because React does actually have a separation of concerns. It's just not one concern per file, as we had traditionally but one concern per component.

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.

## SEPARATION OF CONCERNS!



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

Rest of the course...

#### 3 Thinking about state

- 👉 When to use state
- 👉 Where to place state
- 👉 Types of state



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

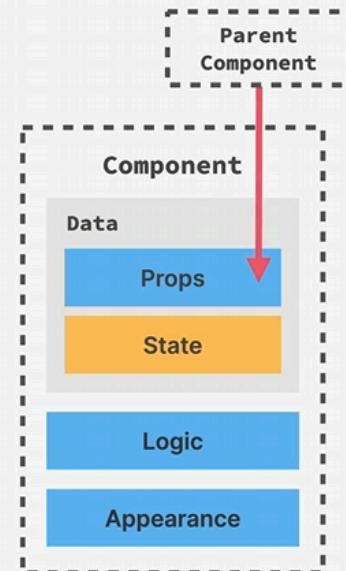
*(So we will keep learning about state throughout the entire course...)*

## 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"



The screenshot shows a user interface with a navigation bar at the top. On the left, there are 'Notifications' and 'Messages' sections, each with a purple circular badge containing '9+'. To the right is a search bar with the text 'javascr' and a magnifying glass icon. Below the navigation bar are tabs for 'Overview', 'Q&A', 'Notes' (which is underlined in red), and 'Announcements'. Under the 'Notes' tab, there is a section titled 'Shopping Cart' with the subtext '2 Courses in Cart'. It displays two course cards: 'Node.js, Express, MongoDB & More: The Complete Bootcamp 2022' and 'The Complete JavaScript Course 2022: From Zero to Expert!'. Each card includes a price of '\$12.99', a '4.7 stars' rating, and a '42 min total' duration.

## 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)

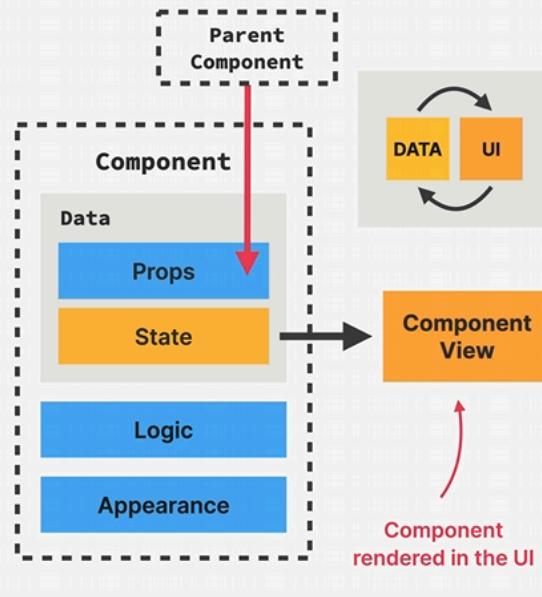
We use these terms interchangeably

The screenshot shows a user interface with a navigation bar at the top. On the left, there are 'Notifications' and 'Messages' sections, each with a purple circular badge containing '9+'. To the right is a search bar with the text 'javascr' and a magnifying glass icon. Below the navigation bar are tabs for 'Overview', 'Q&A', 'Notes' (which is underlined in red), and 'Announcements'. Under the 'Notes' tab, there is a section titled 'Shopping Cart' with the subtext '2 Courses in Cart'. It displays two course cards: 'Node.js, Express, MongoDB & More: The Complete Bootcamp 2022' and 'The Complete JavaScript Course 2022: From Zero to Expert!'. Each card includes a price of '\$12.99', a '4.7 stars' rating, and a '42 min total' duration.

## 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?

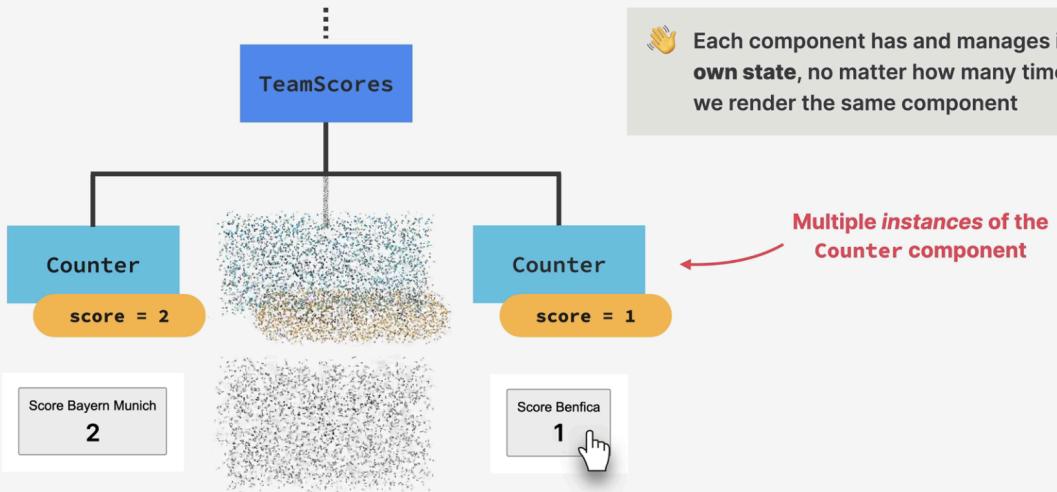
### STATE

- 👉 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

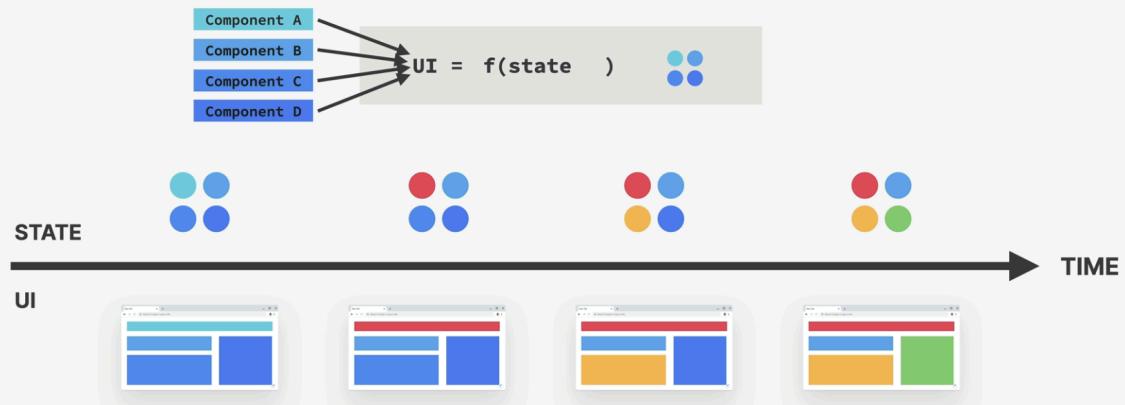
STATE ALLOWS DEVELOPERS TO:

- 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



## 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



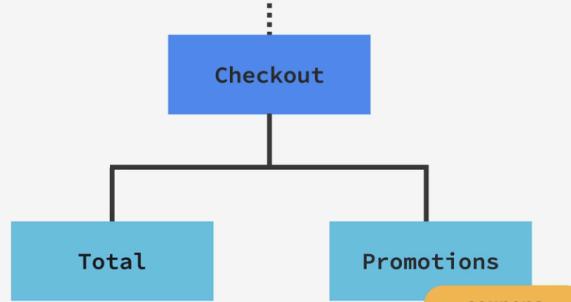
## IN PRACTICAL TERMS...

### 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

The screenshot shows a user interface with two main sections. On the right, a **Total** component displays a total amount of **€30.98**, noting a discount of **€174.98** and **82% off**. On the left, a **Promotions** component shows a message that the coupon **LEARNNEWSKILLS** has been applied, with an **Apply** button. A double-headed arrow between the two components indicates they are sharing state. The **Promotions** component also includes a section for entering a new coupon code.

## PROBLEM: SHARING STATE WITH SIBLING COMPONENT

ONE-WAY DATA FLOW

```

graph TD
    Checkout[Checkout] --> Total[Total]
    Checkout --> Promotions[Promotions]
    Total --- coupons1[coupons]
    Promotions --- coupons2[coupons]
    Promotions --- setCoupons1[setCoupons]
  
```

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

👉 Total component also needs access to coupons state

**Total**

total: **€30.98**  
€174.98  
82% off

Checkout

Promotions **Promotions**  
X LEARNNEWSKILLS is applied

Enter Coupon  Apply

**coupons**

Buy now, pay later for orders of \$25 and over  
Klarna, afterpay ↗

## PROBLEM: SHARING STATE WITH SIBLING COMPONENT

ONE-WAY DATA FLOW

```

graph TD
    Checkout[Checkout] --> Total[Total]
    Checkout --> Promotions[Promotions]
    Total --- coupons1[coupons]
    Promotions --- coupons2[coupons]
    Promotions --- setCoupons1[setCoupons]
  
```

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

**Total**

total: **€30.98**  
€174.98  
82% off

Checkout

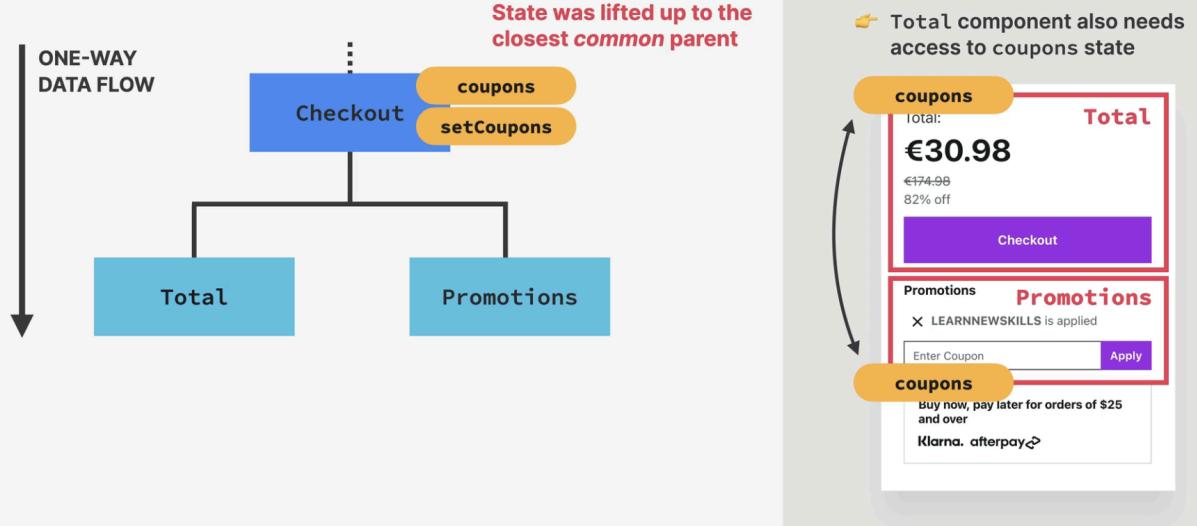
Promotions **Promotions**  
X LEARNNEWSKILLS is applied

Enter Coupon  Apply

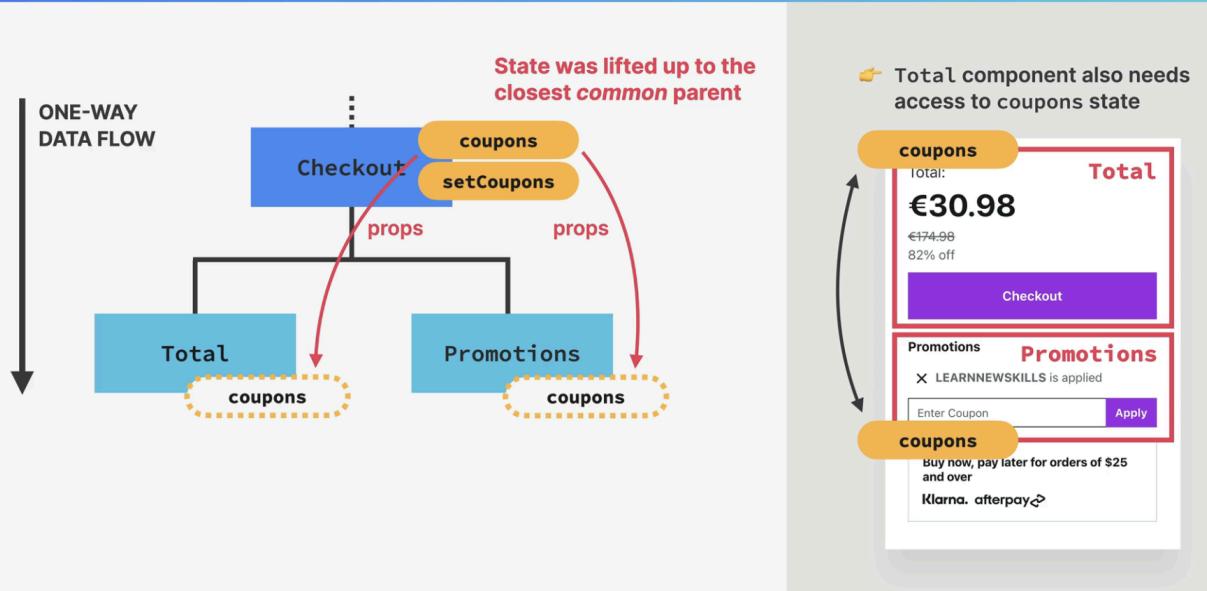
**coupons**

Buy now, pay later for orders of \$25 and over  
Klarna, afterpay ↗

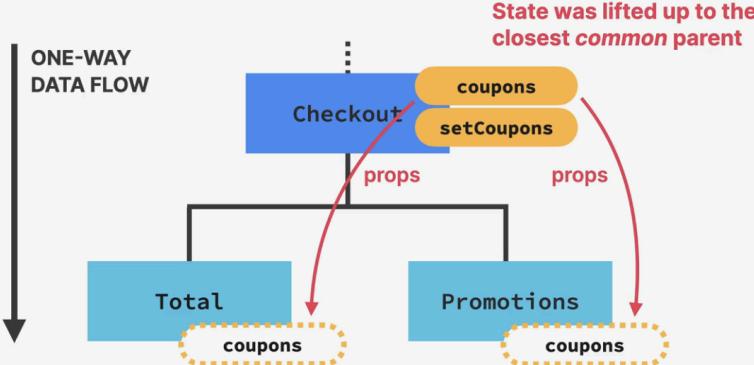
## SOLUTION: LIFTING STATE UP



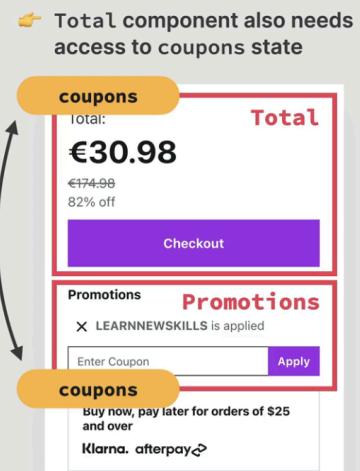
## SOLUTION: LIFTING STATE UP



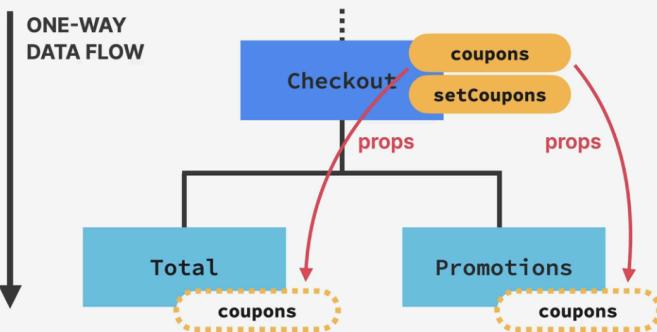
## SOLUTION: LIFTING STATE UP



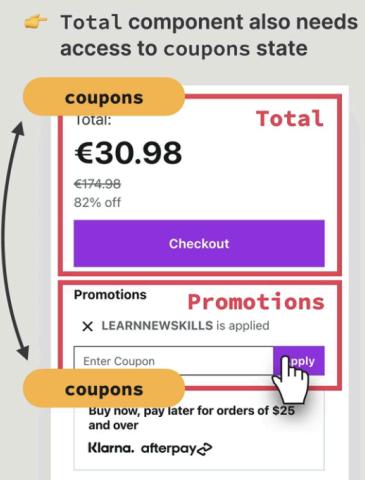
👉 By lifting state up, we have successfully shared one piece of state with multiple components in different positions in the component tree



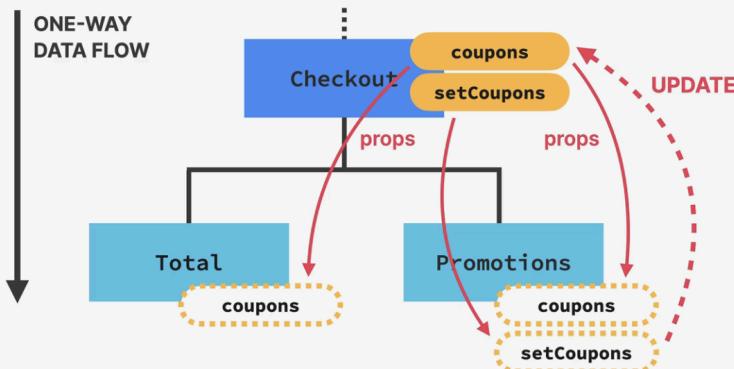
## CHILD-TO-PARENT COMMUNICATION



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

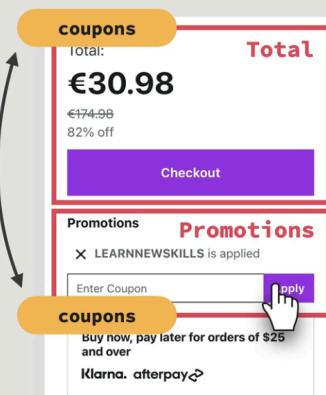


## CHILD-TO-PARENT COMMUNICATION

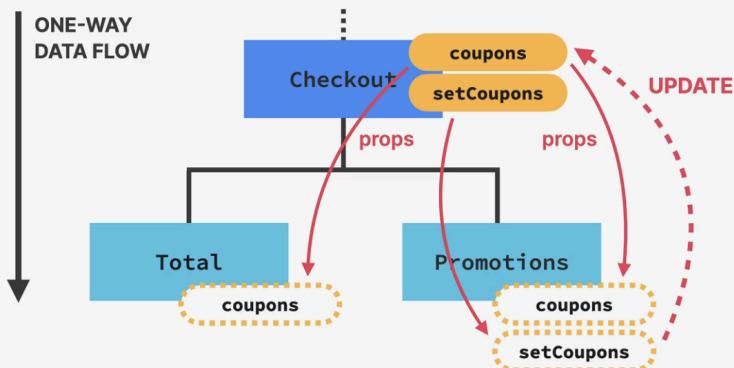


🤔 If data flows from parent to children, how can Promotions (child) update state in Checkout (parent)?

👉 Total component also needs access to coupons state

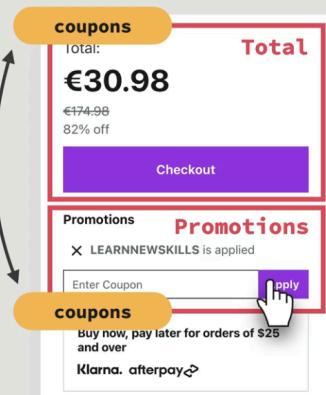


## CHILD-TO-PARENT COMMUNICATION



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

👉 Total component also needs access to coupons state



## 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

## DERIVING STATE

- 👎 Three separate pieces of state, even though numItems and totalPrice depend on cart
- 👎 Need to keep them in sync (update together)
- 👎 3 state updates will cause 3 re-renders

- 👍 Derived state: state that is computed from an existing piece of state or from props
- 👍 Just regular variables, no useState
- 👍 cart state is the **single source of truth** for this related data
- 👍 Works because re-rendering component will automatically re-calculate derived state

```
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);
```

## DERIVING STATE

```
const [cart, setCart] = useState([
  { name: "JavaScript Course", price: 15.99 },
  { name: "Node.js Bootcamp", price: 14.99 }
]);
const numItems = cart.length;
const totalPrice =
  cart.reduce((acc, cur) => acc + cur.price, 0);
```

## 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

```

1 import { useState } from "react";
2
3 export default function App() {
4   const [advice, setAdvice] = useState("");
5
6   async function getAdvice() {
7     const res = await fetch("https://api.adviceslip.com/advice");
8     const data = await res.json();
9     setAdvice(data.slip.advice);
10 }
11
12   return (
13     <div>
14       <h1>{advice}</h1>
15       <button onClick={getAdvice}>Get advice</button>
16     </div>
17   );
18 }
19

```

App.js

Browser

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

'message' of object 'SyntaxError': /src/App.js: Missing initializer in destructuring declaration. (2:10)

App.js — 04-steps

```

1 import { useState } from "react";
2
3 const messages = [
4   "Learn React *",
5   "Apply for jobs 🚧",
6   "Invest your new income 😎",
7 ];
8
9 export default function App() {
10   const [step, setStep] = useState(1);
11   console.log(err);
12
13   const step = 1; // This line has a red underline
14
15   function handlePrevious() {
16     alert("Previous");
17   }
18
19   function handleNext() {
20     alert("Next");
21   }
22
23   return (
24     <div className="steps">
25       <div className="numbers">
26         <div className="center" style={{ transform: `translateX(-50%)` }}>

```

React App

localhost:3000

Step 1: Learn React

Previous Next

Console

react-dom.development.js:29832 Download the React DevTools for a better development experience: <https://reactjs.org/link/react-devtools>

App.js:11

Array(2) 0: f 1: f length: 2 [[Prototype]]: Array(0)

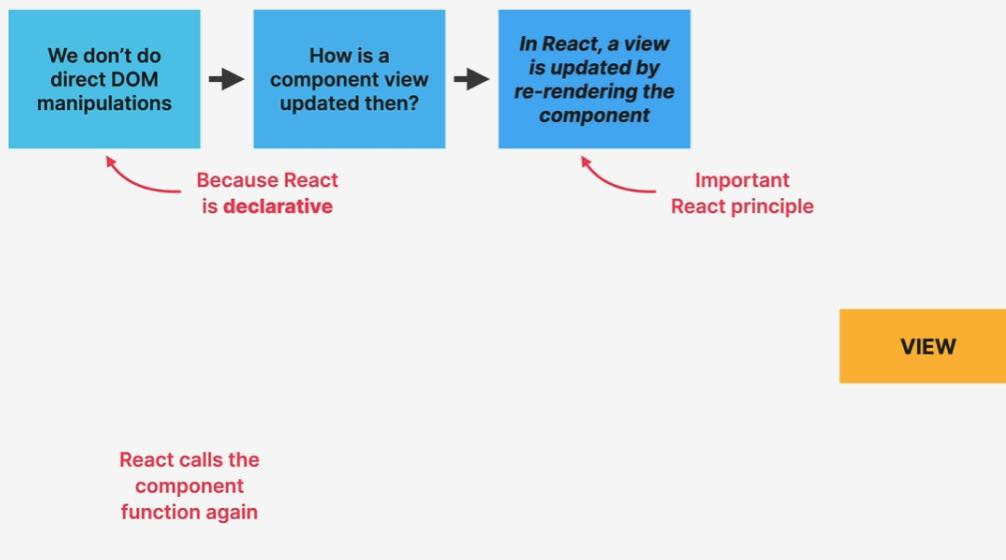
App.js:11

1 Issue: 1

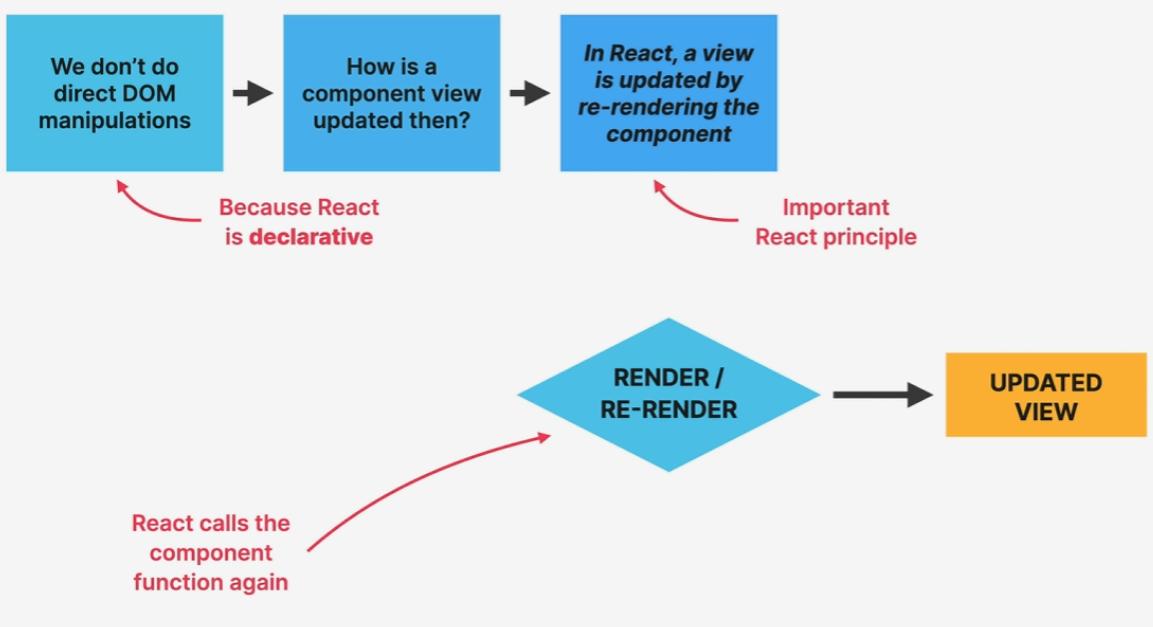
Compiled successfully!

Ln 13, Col 18 Spaces: 2 UTF-8 LF () JavaScript Go Live Prettier

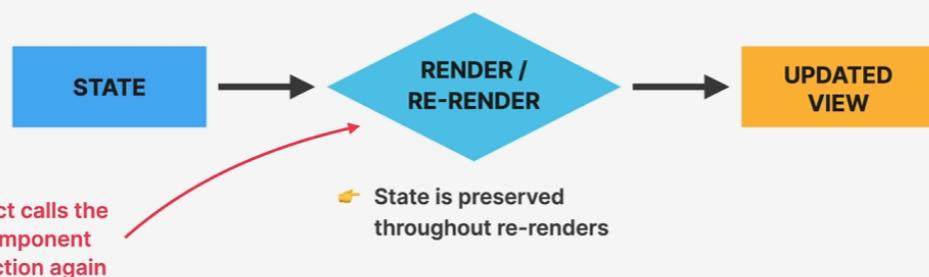
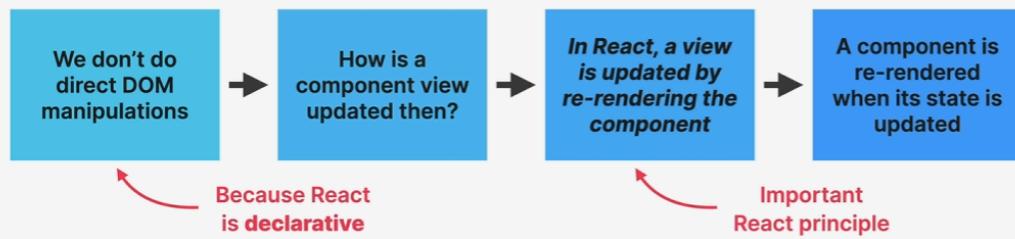
## THE MECHANICS OF STATE IN REACT



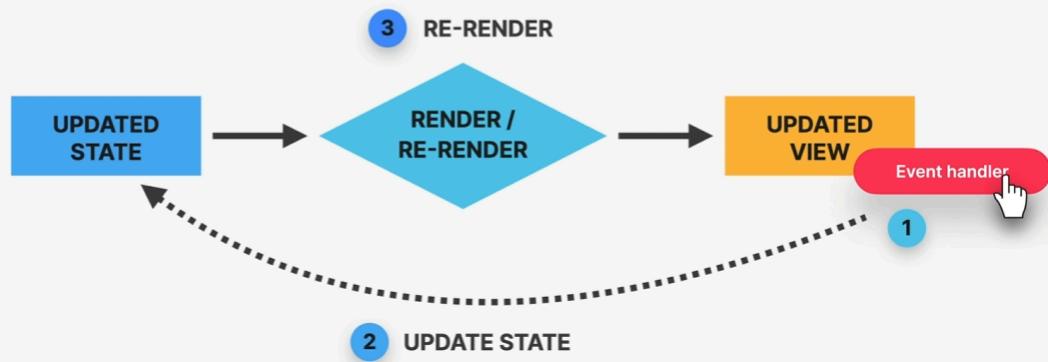
## THE MECHANICS OF STATE IN REACT



## THE MECHANICS OF STATE IN REACT



## THE MECHANICS OF STATE IN REACT



## 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

```
const [advice, setAdvice] =  
  useState("Have a firm handshake.");  
const [countAdvice, setCountAdvice] =  
  useState(12);
```

<https://5u8t1.csb.app/>

Have a firm handshake.

You have read 12 pieces of advice

UPDATE STATE

```
setAdvice(data.slip.advice);  
setCountAdvice((count) => count + 1);
```

## 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

```
const [advice, setAdvice] =  
  useState("Quality beats quantity.");  
const [countAdvice, setCountAdvice] =  
  useState(13);
```

<https://5u8t1.csb.app/>

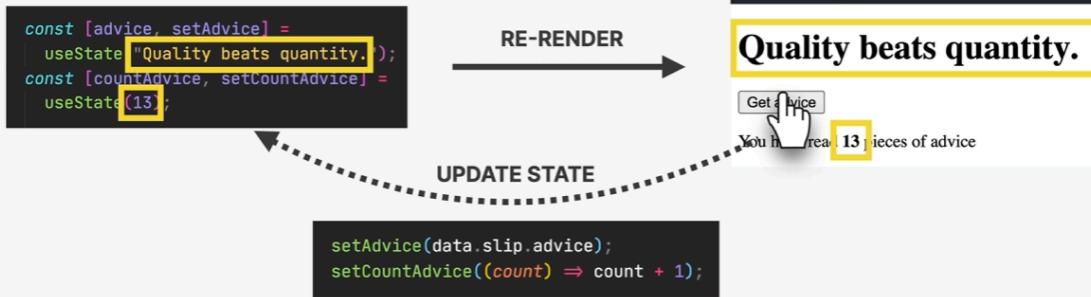
Quality beats quantity.

You have read 13 pieces of advice

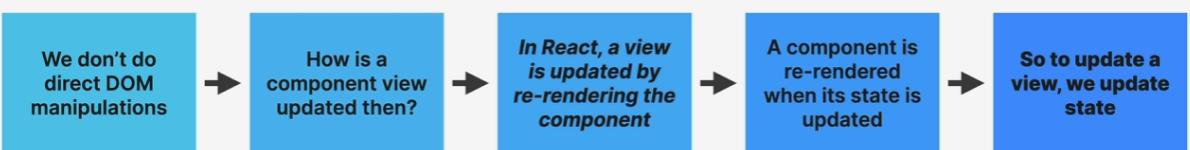
UPDATE STATE

```
setAdvice(data.slip.advice);  
setCountAdvice((count) => count + 1);
```

## THE MECHANICS OF STATE IN REACT



## THE MECHANICS OF STATE IN REACT



👉 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) {
```

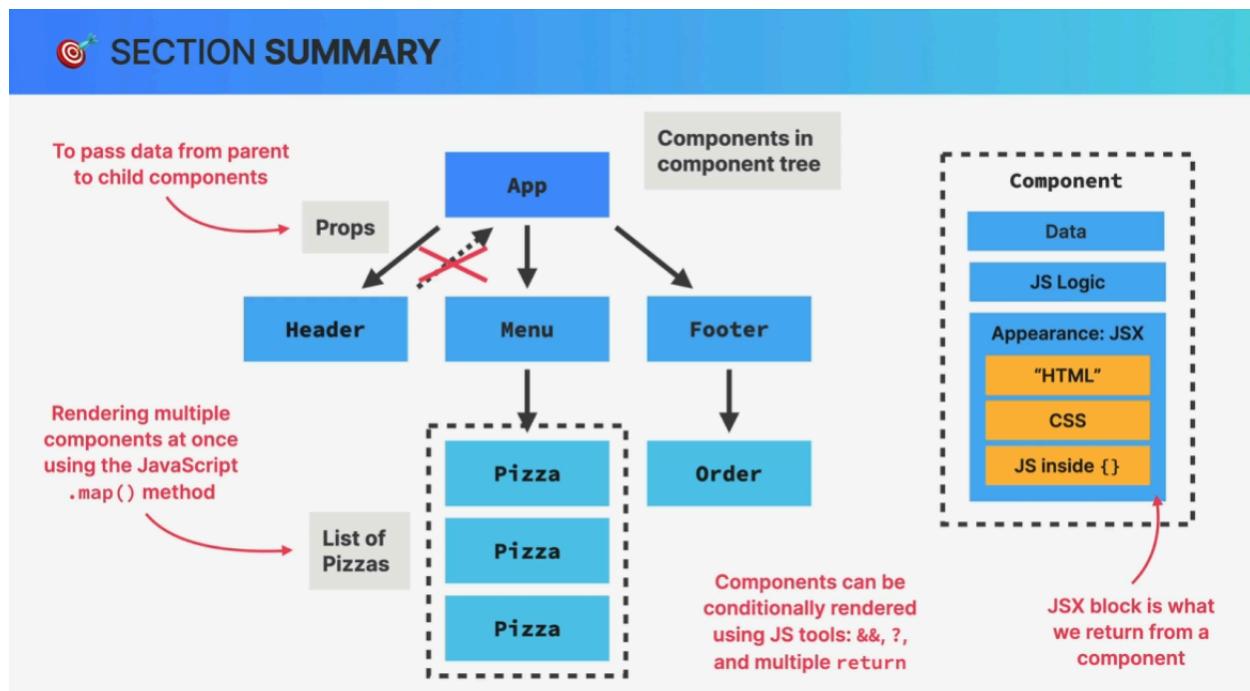
```

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

```
function Question() {
  const [upvotes, setUpvotes] = useState(0);

  return (
    <div>
      {/* ... */}
      <Button upvotes={upvotes} bgColor="blue" />
    );
}
```

```
function Button({ upvotes, bgColor }) {
  const [hovered, setHovered] = useState(false);

  return (
    <div>
      {/* ... */}
      <button
        onMouseEnter={() => setHovered(true)}
        onMouseLeave={() => setHovered(false)}
        style={{ background: bgColor }}
      >
        {hovered ? "Upvote" : `👍 ${upvotes}`}
      </button>
    </div>
  );
}
```

### STATE VS. PROPS

**STATE**

👉 Internal data, owned by component

```
function Question() {
  const [upvotes, setUpvotes] = useState(0);

  return (
    <div>
      {/* ... */}
      <Button upvotes={upvotes} bgColor="blue" />
    );
}
```

**PROPS**

👉 External data, owned by parent component

```
function Question() {
  const [upvotes, setUpvotes] = useState(0);

  return (
    <div>
      {/* ... */}
      <Button upvotes={upvotes} bgColor="blue" />
    );
}

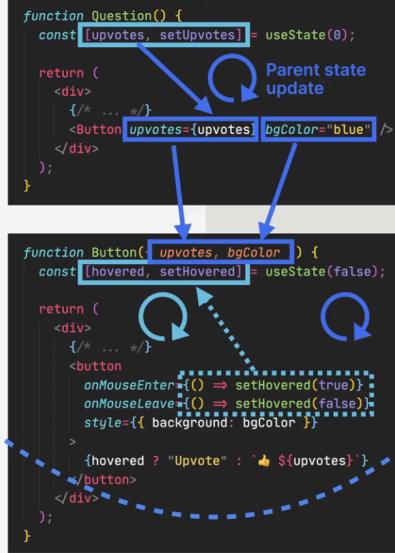
function Button({ upvotes, bgColor }) {
  const [hovered, setHovered] = useState(false);

  return (
    <div>
      {/* ... */}
      <button
        onMouseEnter={() => setHovered(true)}
        onMouseLeave={() => setHovered(false)}
        style={{ background: bgColor }}
      >
        {hovered ? "Upvote" : `👍 ${upvotes}`}
      </button>
    </div>
  );
}
```

## 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

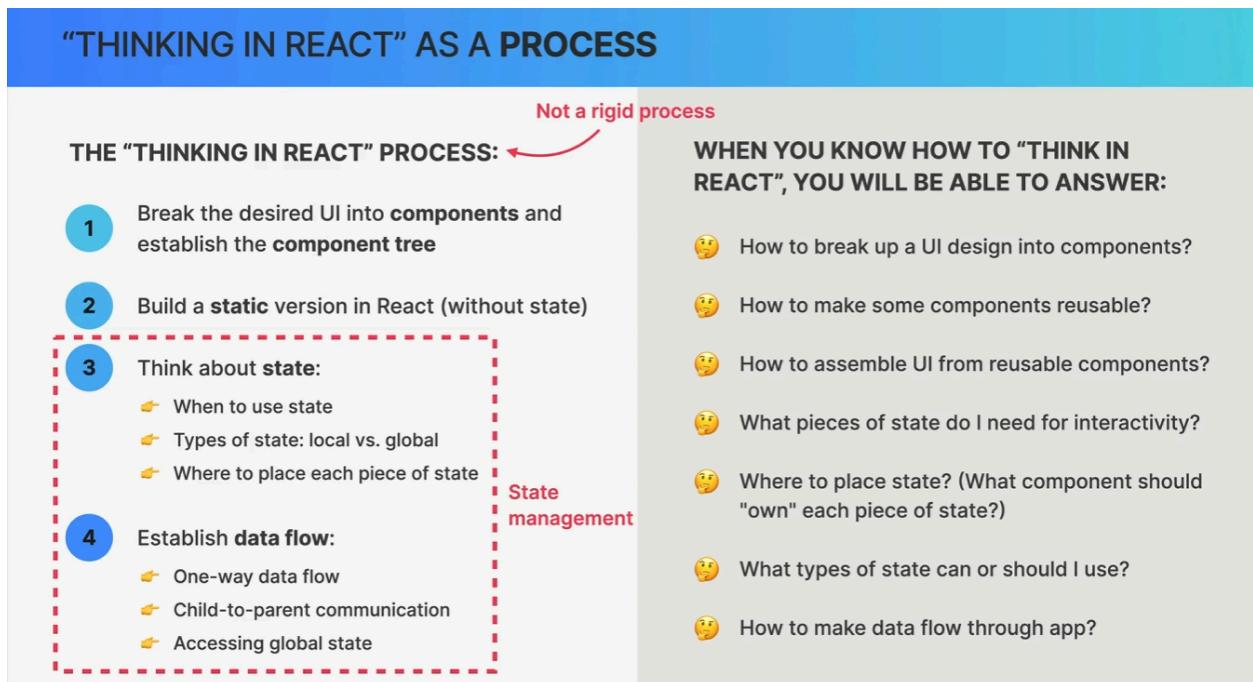
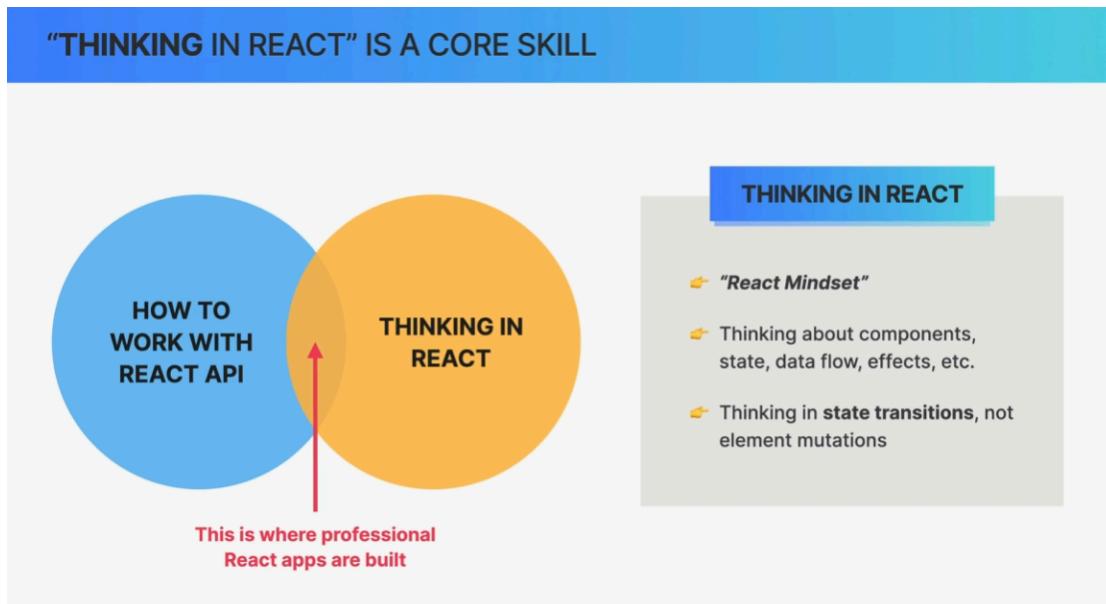


### PROPS

- 👉 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

### WHAT IS STATE MANAGEMENT?

👉 State management: Deciding **when** to create pieces of state, what **types** of state are necessary, **where** to place each piece of state, and how data **flows** through the app

→ Giving each piece of state a **home**

udemy Categories  Categories

Shopping Cart searchQuery

2 Courses in Cart

PIECES OF STATE (useState)

coupons

notifications

language

user

isOpen

Total: €25.98

Checkout

Promotions

Buy now, pay later for order

Notifications

Messages

Account settings

Payment methods

Udemy credits

Purchase history

Language English

### TYPES OF STATE: LOCAL VS. GLOBAL STATE

**LOCAL STATE**

**GLOBAL STATE**

👉 State needed **only** by one or few components

👉 State that is defined in a component and **only** that component and child components have access to it (by passing via props)

Local state

udemy Categories  Categories

Shopping Cart

2 Courses in Cart

Node.js, Express, MongoDB & More: The Complete Bootcamp 2022

The Complete JavaScript Course 2022: From Zero to Expert!

Checkout

Promotions

Buy now, pay later for order

Notifications

Messages

Account settings

Payment methods

Udemy credits

Purchase history

Language English

## TYPES OF STATE: LOCAL VS. GLOBAL STATE

### LOCAL STATE

- 👉 State needed **only by one or few components**
- 👉 State that is defined in a component and **only that component and child components** have access to it (by passing via props)

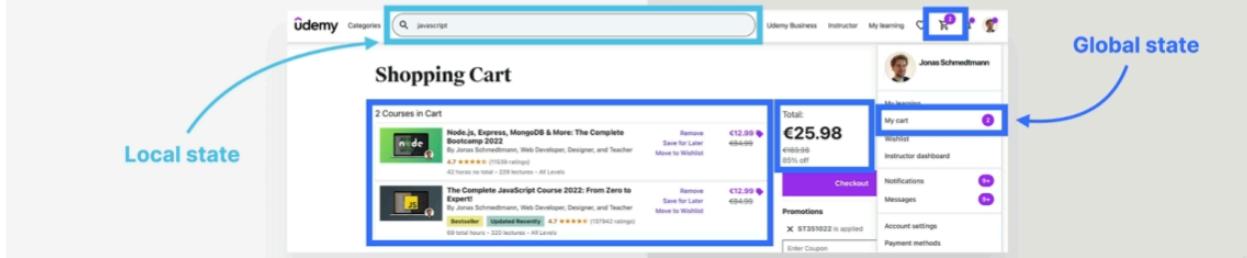
Local state

### GLOBAL STATE

- 👉 State that **many components** might need
- 👉 Shared state that is accessible to **every component** in the entire application



Global state



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.

## TYPES OF STATE: LOCAL VS. GLOBAL STATE

### LOCAL STATE

- 👉 State needed **only by one or few components**
  - 👉 State that is defined in a component and **only that component and child components** have access to it (by passing via props)
- 👉 **We should always start with local state**

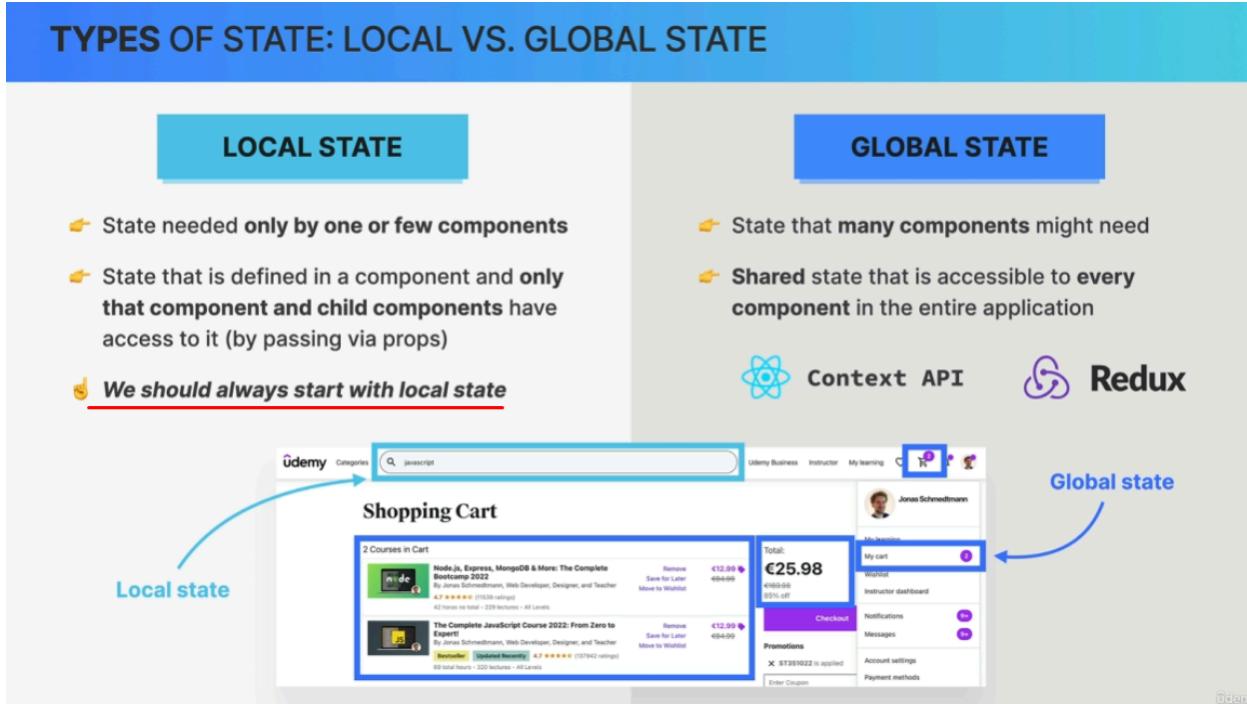
Local state

### GLOBAL STATE

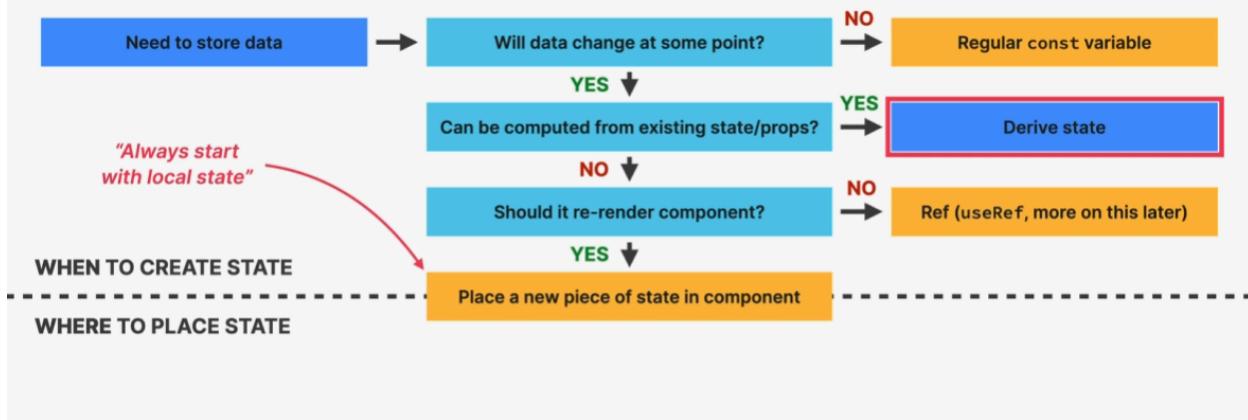
- 👉 State that **many components** might need
- 👉 Shared state that is accessible to **every component** in the entire application



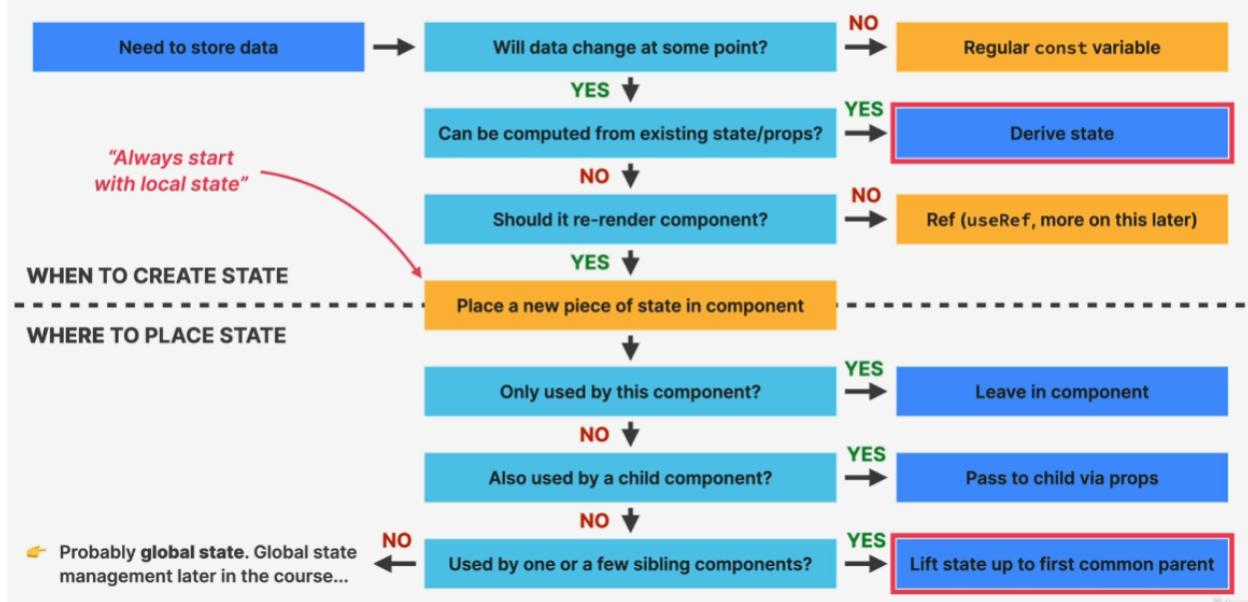
Global state



## STATE: WHEN AND WHERE?



## STATE: WHEN AND WHERE?



## The Children prop

### THE CHILDREN PROP

**Button**

A diagram illustrating the concept of the 'children' prop. It shows a purple rounded rectangle labeled 'Button'. Inside it is a white rectangular box labeled 'props.children'. A red arrow points from the text below to this box. To the right of the button, there is explanatory text.

An empty "hole" that can be filled by any JSX the component receives as children

### THE CHILDREN PROP

**Button**

A diagram similar to the first one, showing a 'Button' component with an internal 'props.children' box. To the right, there is a 'Previous' button with its corresponding JSX code. The code shows a span element with a yellow arrow icon and the word 'Previous'.

Previous

```
<Button onClick={previous}>
  <span>⬅</span> Previous
</Button>
```

An empty "hole" that can be filled by any JSX the component receives as children

### THE CHILDREN PROP

**Button**

A diagram similar to the previous ones, showing a 'Button' component with an internal 'props.children' box. A red arrow points from the explanatory text below to the 'props.children' box. To the right, there is a 'Previous' button with its corresponding JSX code. The code shows a span element with a yellow arrow icon and the word 'Previous'.

Previous

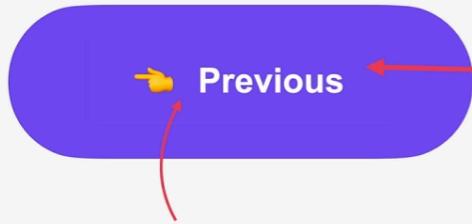
```
<Button onClick={previous}>
  <span>⬅</span> Previous
</Button>
```

An empty "hole" that can be filled by any JSX the component receives as children

Children of Button, accessible through props.children

## THE CHILDREN PROP

### Button



Children of Button,  
accessible through  
props.children

```
<Button onClick={previous}>  
  <span>◀</span> Previous  
</Button>
```

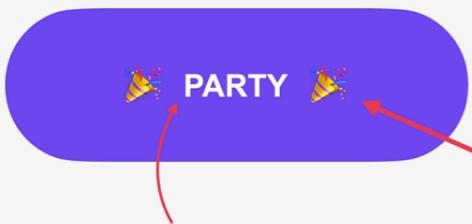
An empty "hole" that can be filled  
by any JSX the component  
receives as children



```
<Button onClick={next}>  
  <span>▶</span>PARTY<span>▶</span>  
</Button>
```

## THE CHILDREN PROP

### Button



Children of Button,  
accessible through  
props.children

```
<Button onClick={previous}>  
  <span>◀</span> Previous  
</Button>
```

An empty "hole" that can be filled  
by any JSX the component  
receives as children

```
<Button onClick={next}>  
  <span>▶</span>PARTY<span>▶</span>  
</Button>
```

- 👉 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

## COMPONENT VS. INSTANCE VS. ELEMENT

Component

```
function Tab({ item }) {
  return (
    <div className='tab-content'>
      <h4>All contacts</h4>
      <p>Your post will be visible</p>
    </div>
  );
}
```

👉 Description of a piece of UI

👉 A component is a function that **returns React elements** (element tree), usually written as JSX

👉 “Blueprint” or “Template”

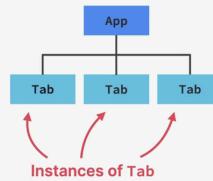
## COMPONENT VS. INSTANCE VS. ELEMENT

Component



Component Instance

```
function App() {
  return (
    <div className='tabs'>
      <Tab item={content[0]} />
      <Tab item={content[1]} />
      <Tab item={content[2]} />
    </div>
  );
}
```



👉 Instances are created when we “use” components

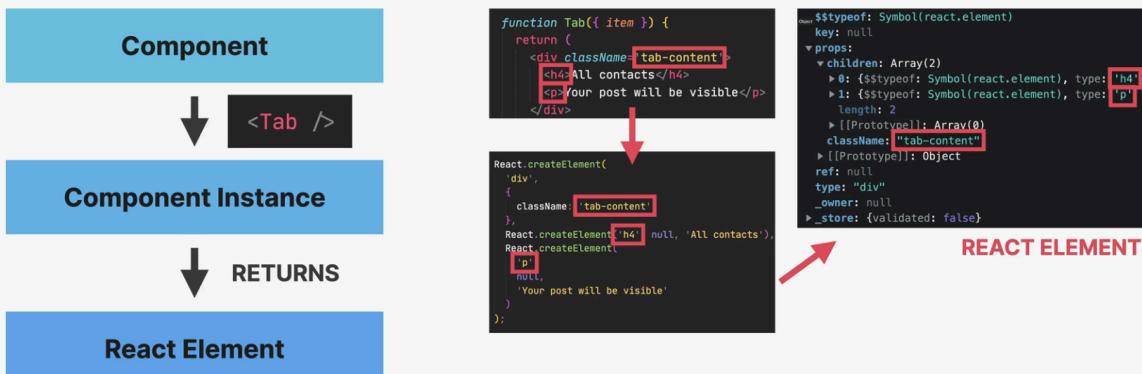
👉 React internally calls Tab()

👉 Actual “physical” manifestation of a component

👉 Has its own state and props

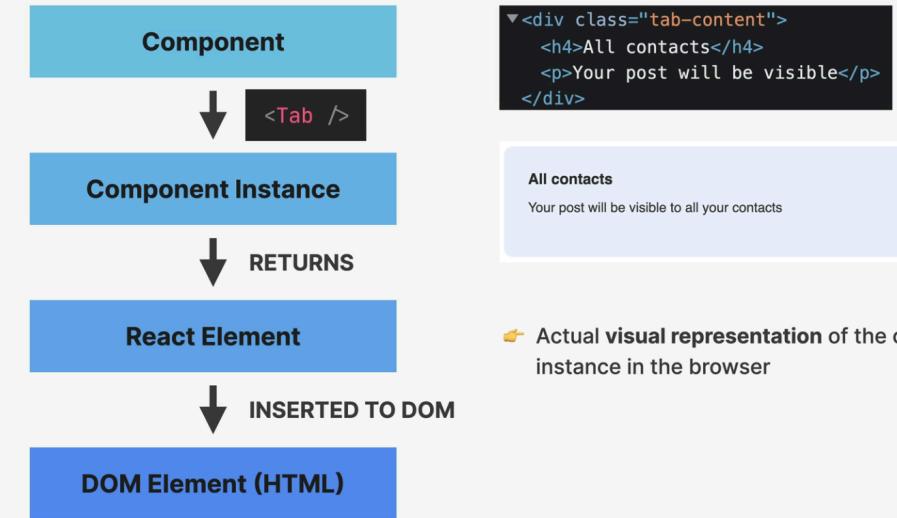
👉 Has a lifecycle (can “be born”, “live”, and “die”)

## COMPONENT VS. INSTANCE VS. ELEMENT



- 👉 JSX is converted to `React.createElement()` function calls
- 👉 A React element is the result of these function calls
- 👉 Information necessary to create DOM elements

## COMPONENT VS. INSTANCE VS. ELEMENT

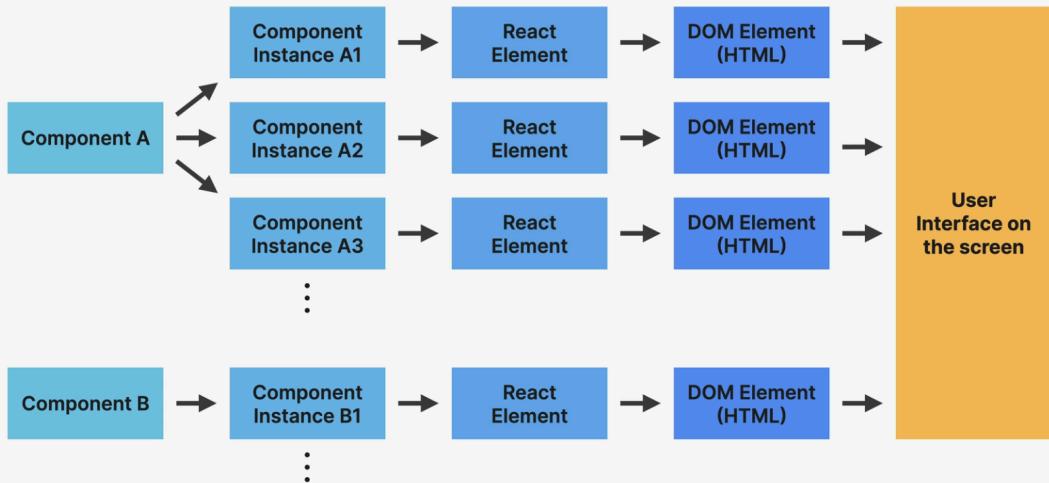


- 👉 Actual visual representation of the component instance in the browser

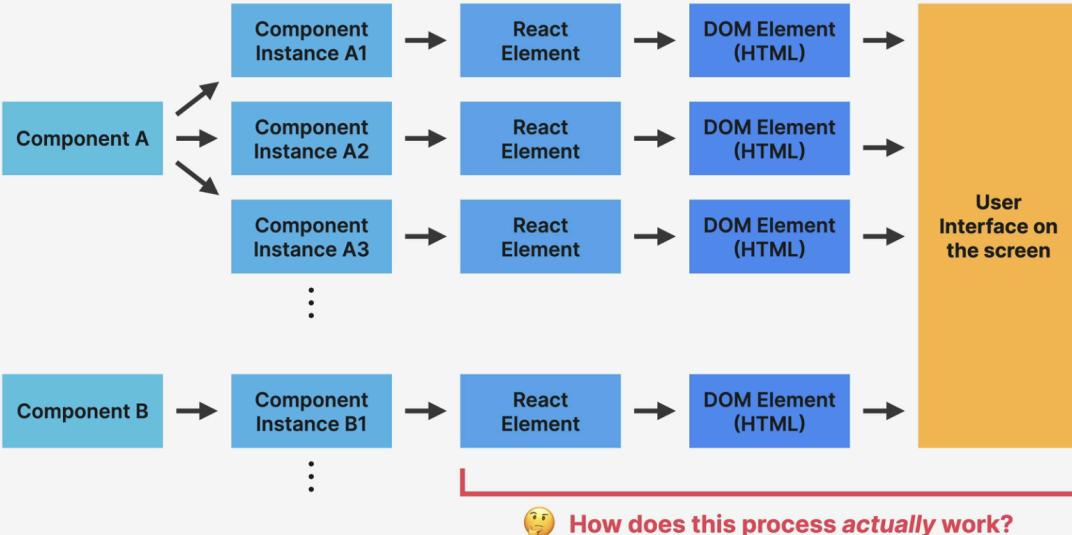
React elements just live inside the React app and have nothing to do with the DOM. They are simply converted to DOM elements when painted on the screen in this final step.

## Quick Recap

### QUICK RECAP BEFORE WE GET STARTED



### QUICK RECAP BEFORE WE GET STARTED



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.

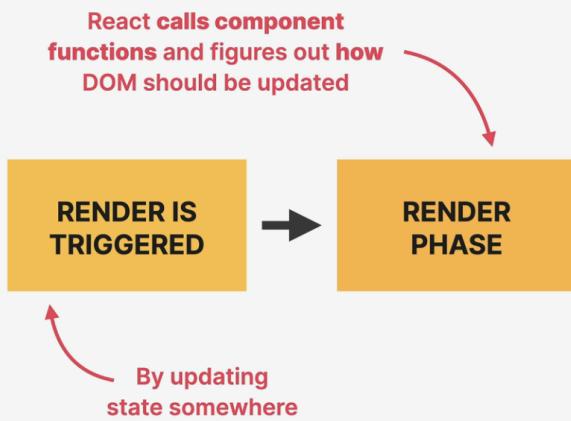
In this phase, React calls our component functions and figures out how it should update the DOM.

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

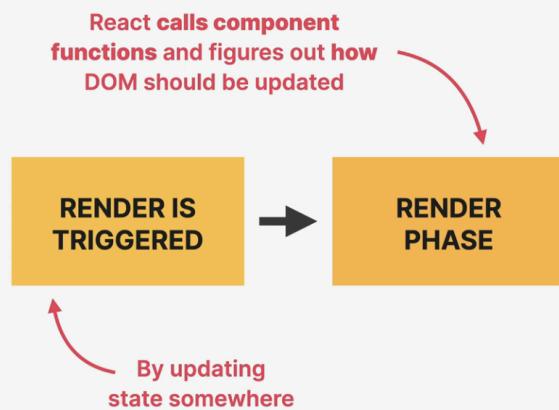
So again, in React, rendering is not about updating the DOM or displaying elements on the screen.

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

## OVERVIEW: HOW COMPONENTS ARE DISPLAYED ON THE SCREEN



## OVERVIEW: HOW COMPONENTS ARE DISPLAYED ON THE SCREEN



In React, rendering is **NOT** updating the DOM or displaying elements on the screen. Rendering only happens **internally** inside React, it does not produce visual changes.

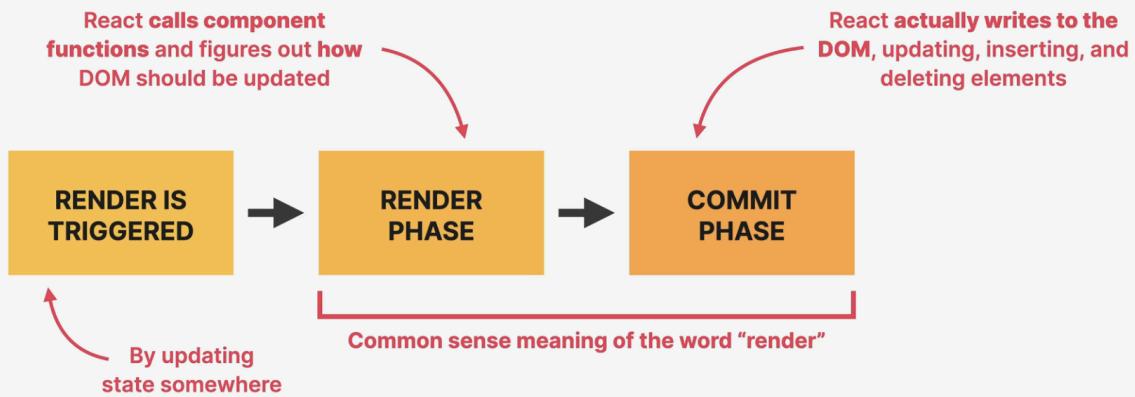
I have always used the term rendering with the meaning of displaying elements on the screen because that was just easy to understand and it just made sense, right?

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

## OVERVIEW: HOW COMPONENTS ARE DISPLAYED ON THE SCREEN

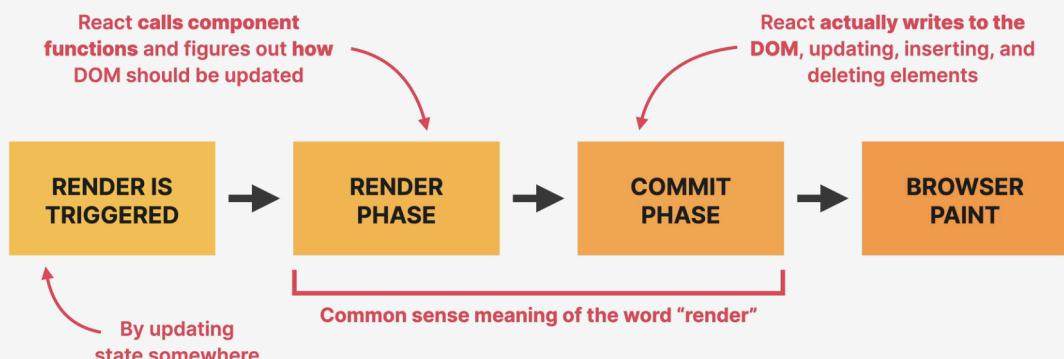


In React, rendering is NOT updating the DOM or displaying elements on the screen. Rendering only happens internally inside React, it does not produce visual changes.

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.

## OVERVIEW: HOW COMPONENTS ARE DISPLAYED ON THE SCREEN



In React, rendering is NOT updating the DOM or displaying elements on the screen. Rendering only happens internally inside React, it does not produce visual changes.

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 RENDERS ARE TRIGGERED

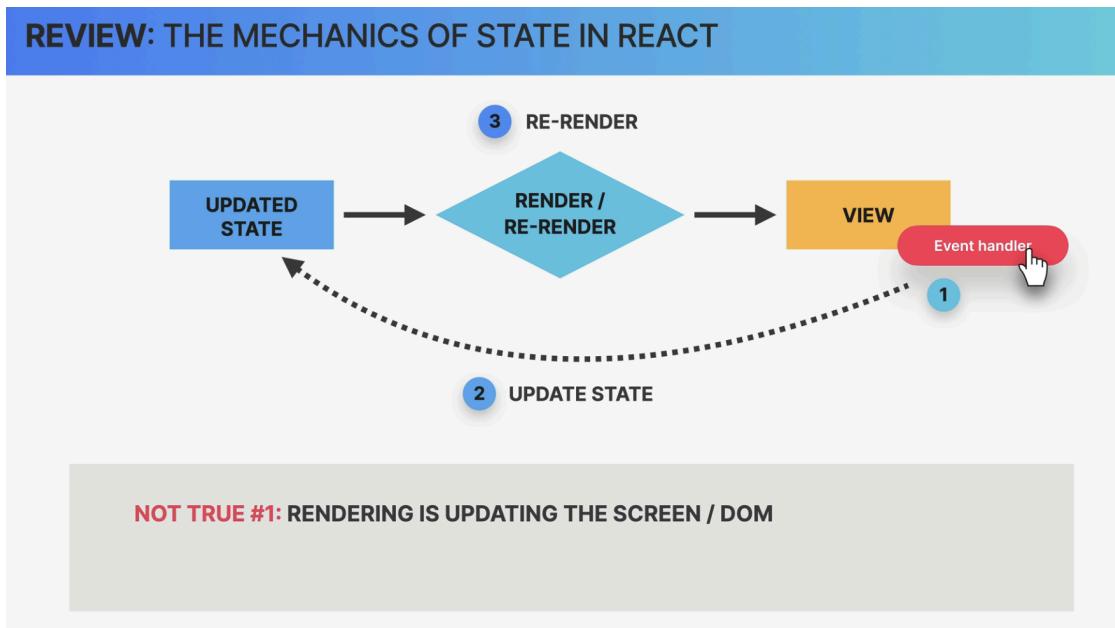
### [1] RENDER IS TRIGGERED

#### THE TWO SITUATIONS THAT TRIGGER RENDERS:

- 1 Initial render of the application
- 2 State is updated in one or more component instances (re-render)

- 👉 The render process is triggered for the **entire application**
- 👉 In practice, it looks like React only re-renders the component where the state update happens, but that's not how it **works behind the scenes**
- 👉 Renders are **not** triggered immediately, but **scheduled** for when the JS engine has some "free time". There is also batching of multiple `setState` calls in event handlers

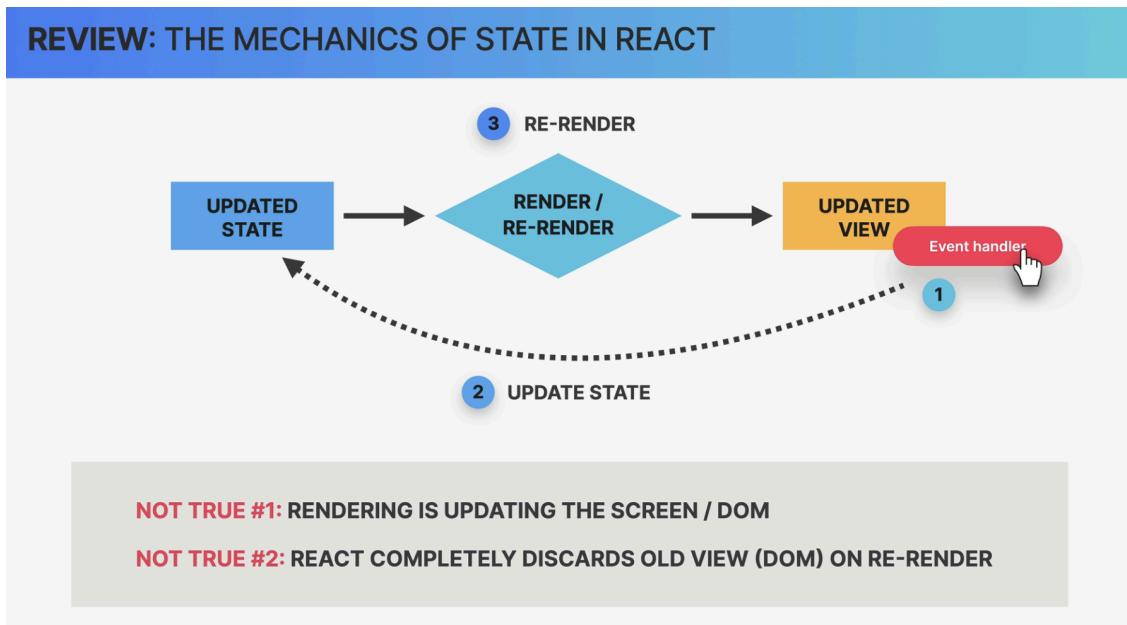
How rendering works



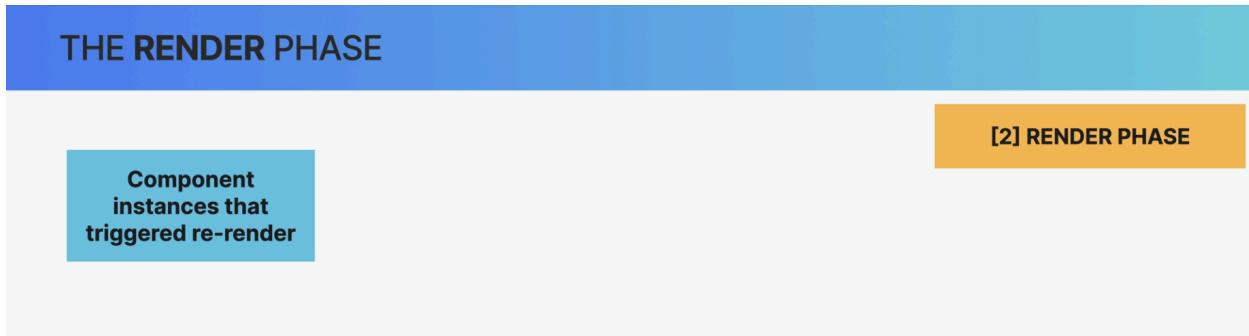
Remember this diagram?

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

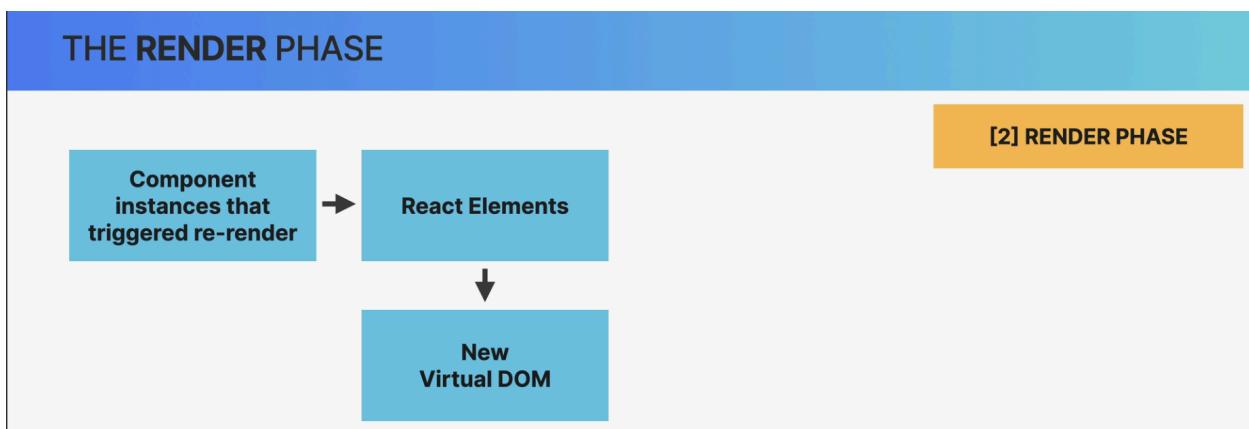
However, now we know that this was **technically not true** because **rendering is not about the screen or the DOM or the view, it's just about calling component functions.**



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



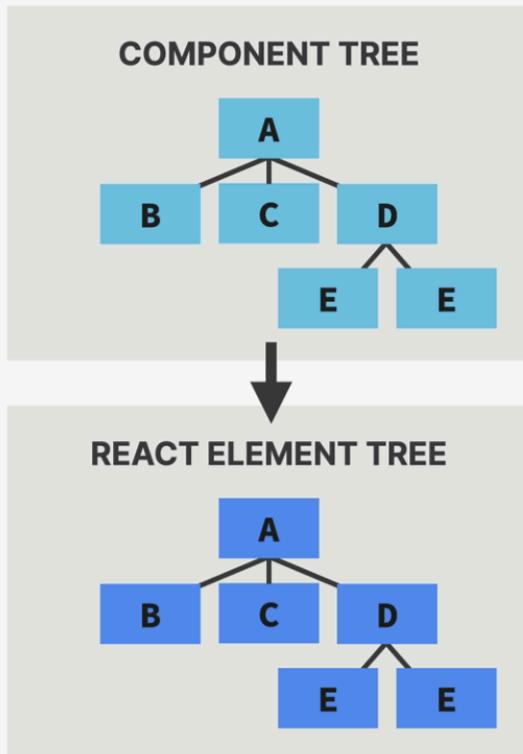
This will create updated React elements that altogether make up the so-called virtual DOM.



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**

# THE VIRTUAL DOM (REACT ELEMENT TREE)

## 1 INITIAL RENDER



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.

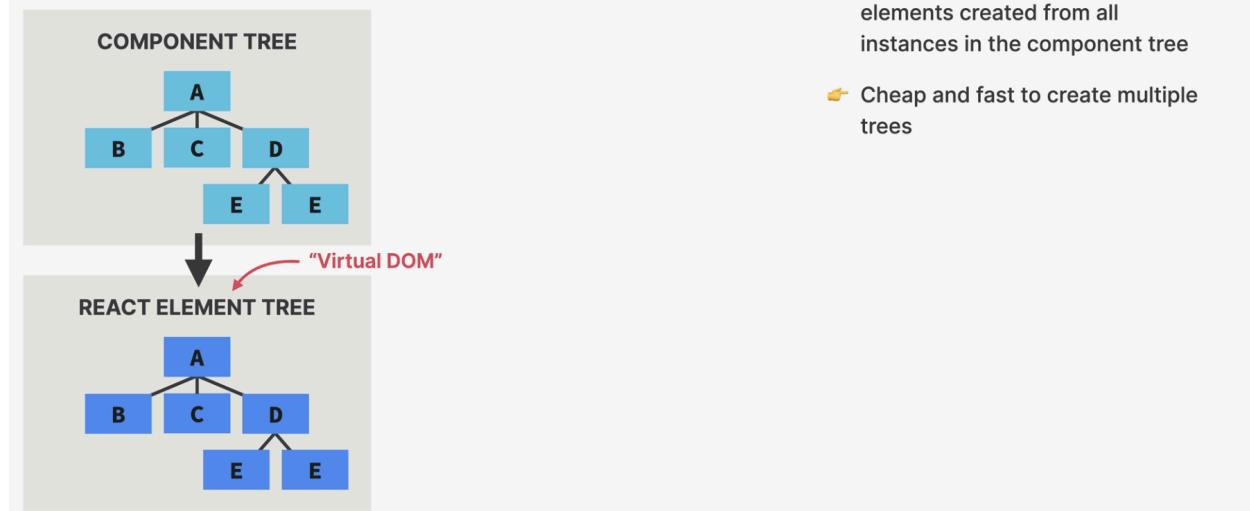
Now, virtual DOM is probably the most hyped and most used term when people describe what React is and how it works.

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.

## THE VIRTUAL DOM (REACT ELEMENT TREE)

### 1 INITIAL RENDER



👉 **Virtual DOM:** Tree of all React elements created from all instances in the component tree

👉 Cheap and fast to create multiple trees

## THE VIRTUAL DOM (REACT ELEMENT TREE)

### 1 INITIAL RENDER



👉 **Virtual DOM:** Tree of all React elements created from all instances in the component tree

👉 Cheap and fast to create multiple trees

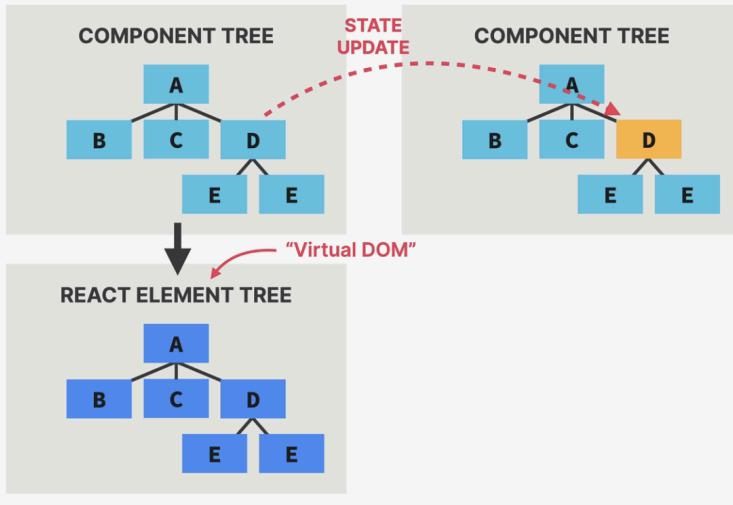
👉 Nothing to do with "shadow DOM"

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.

## THE VIRTUAL DOM (REACT ELEMENT TREE)

### 1 INITIAL RENDER



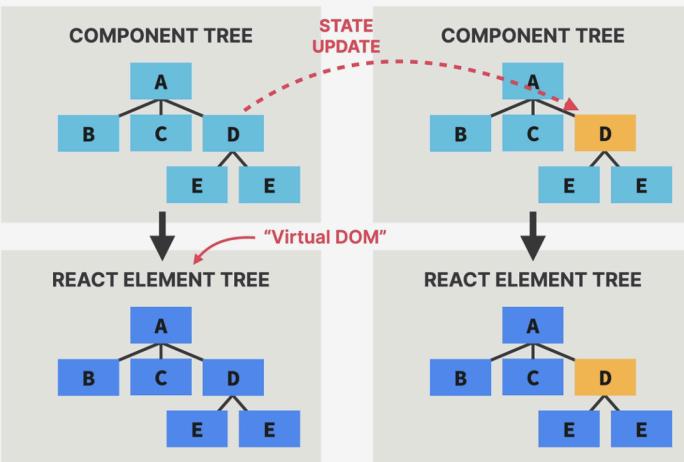
- ↳ Virtual DOM: Tree of all React elements created from all instances in the component tree
- ↳ Cheap and fast to create multiple trees
- ↳ Nothing to do with "shadow DOM"

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.

## THE VIRTUAL DOM (REACT ELEMENT TREE)

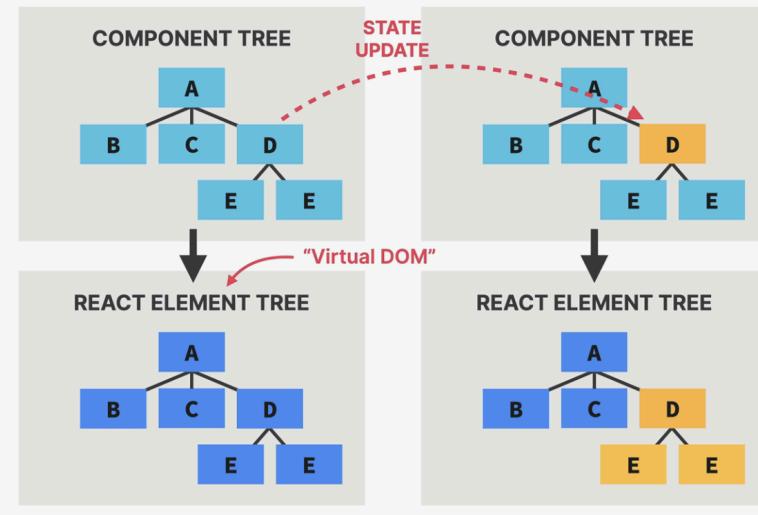
### 1 INITIAL RENDER → RE-RENDERS



- ↳ Virtual DOM: Tree of all React elements created from all instances in the component tree
- ↳ Cheap and fast to create multiple trees
- ↳ Nothing to do with "shadow DOM"

## THE VIRTUAL DOM (REACT ELEMENT TREE)

1 INITIAL RENDER → RE-RENDERS



- ↳ **Virtual DOM:** Tree of all React elements created from all instances in the component tree
- ↳ Cheap and fast to create multiple trees
- ↳ Nothing to do with "shadow DOM"

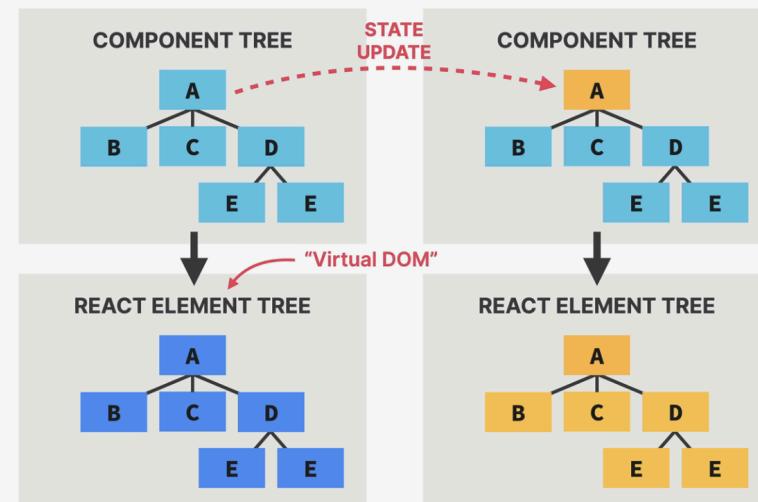
- ⚡ Rendering a component will cause all of its child components to be rendered as well (no matter if props changed or not)

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.**

## THE VIRTUAL DOM (REACT ELEMENT TREE)

1 INITIAL RENDER → RE-RENDERS

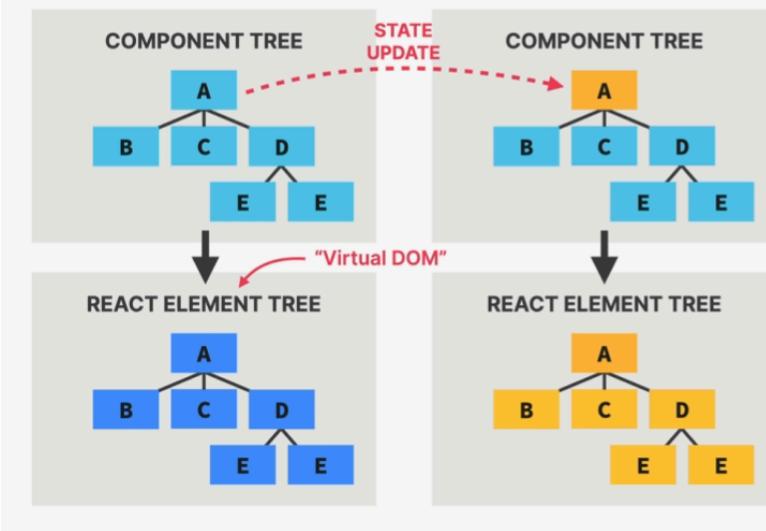


- ↳ **Virtual DOM:** Tree of all React elements created from all instances in the component tree
- ↳ Cheap and fast to create multiple trees
- ↳ Nothing to do with "shadow DOM"

- ⚡ Rendering a component will cause all of its child components to be rendered as well (no matter if props changed or not)

## THE VIRTUAL DOM (REACT ELEMENT TREE)

### 1 INITIAL RENDER → RE-RENDERS



👉 Virtual DOM: Tree of all React elements created from all instances in the component tree

👉 Cheap and fast to create multiple trees

👉 Nothing to do with "shadow DOM"

❗ Rendering a component will cause all of its child components to be rendered as well (no matter if props changed or not)

Necessary because React doesn't know whether children will be affected

Udemy

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.

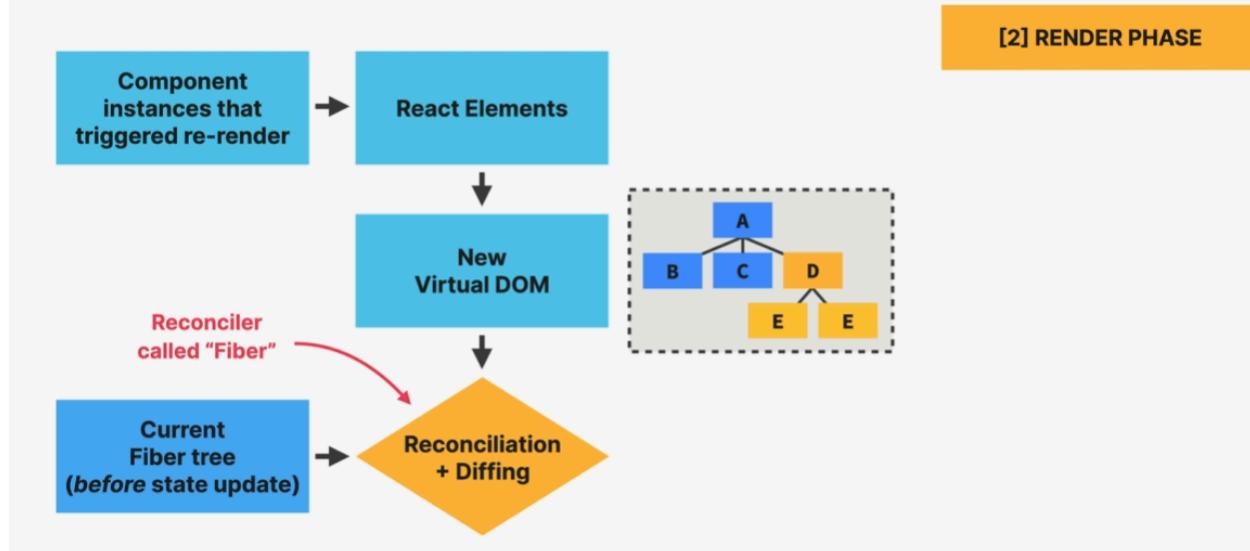
This means that if you update the highest component in a component tree, in this example component A, then the entire application will be re-rendered.

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.

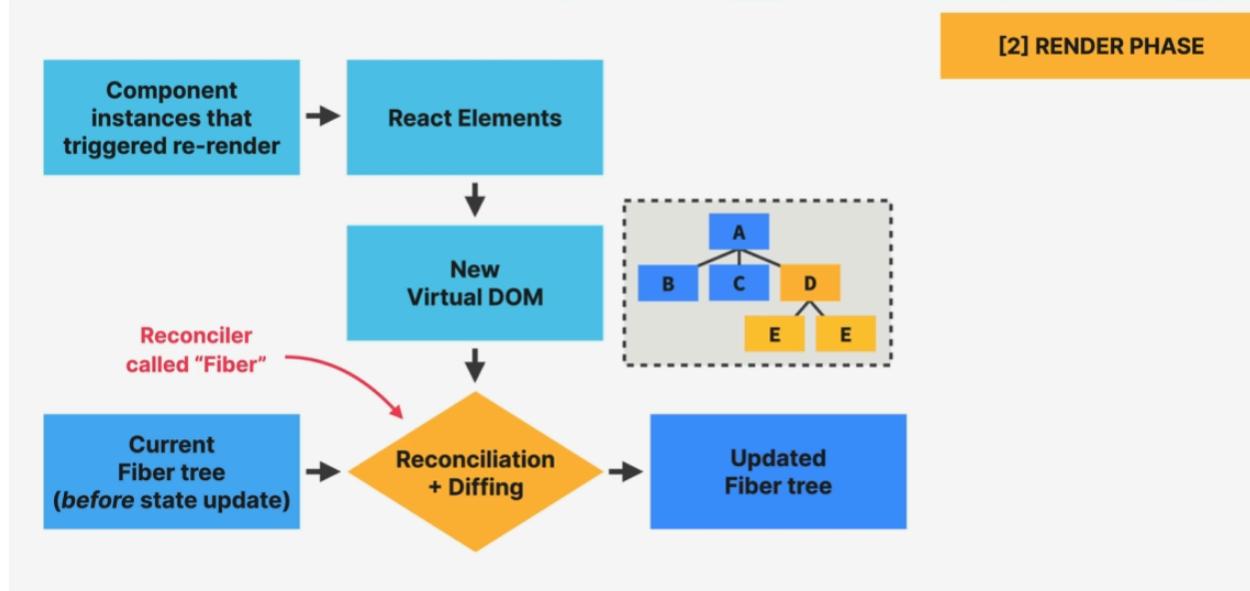
## THE RENDER PHASE

[2] RENDER PHASE



## THE RENDER PHASE

[2] RENDER PHASE



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?

## WHAT IS RECONCILIATION AND WHY DO WE NEED IT?

The diagram illustrates the concept of reconciliation. On the left, a grey box contains a thinking face emoji followed by the text: "Why not update the entire DOM whenever state changes somewhere in the app?". Below this is a downward arrow labeled "BECAUSE". Underneath "BECAUSE" is a list of reasons: "That would be inefficient and wasteful:", followed by two numbered points: "Writing to the DOM is (relatively) slow" and "Usually only a small part of the DOM needs to be updated". Another downward arrow labeled "HOW?" points to a red heart emoji followed by the text: "Reconciliation: Deciding which DOM elements actually need to be inserted, deleted, or updated, in order to reflect the latest state changes".

On the right, there are two screenshots of a Udemy course page. The top screenshot shows a course content list with a highlighted item. A hand cursor is shown clicking on the item. A yellow button labeled "showModal = true" is overlaid on the screen. The bottom screenshot shows a modal dialog titled "How would you rate this course?" with a "Select Rating" button and five yellow stars. A red arrow points from the text "Only these new DOM elements are created" to the modal. The text "Only these new DOM elements are created" is also overlaid at the bottom of the bottom screenshot.

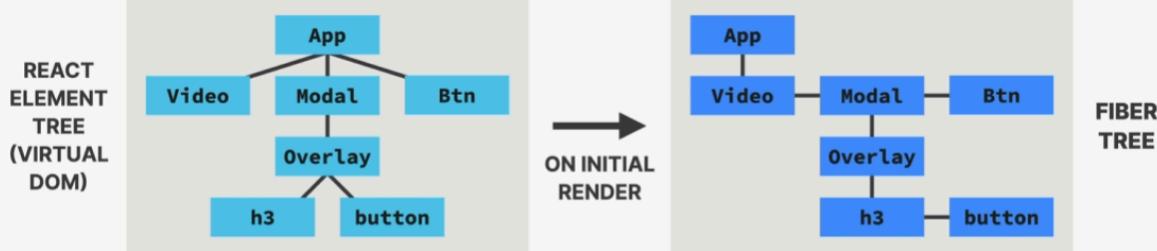
The Reconciler (called Fiber)

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.**

## THE RECONCILER: FIBER

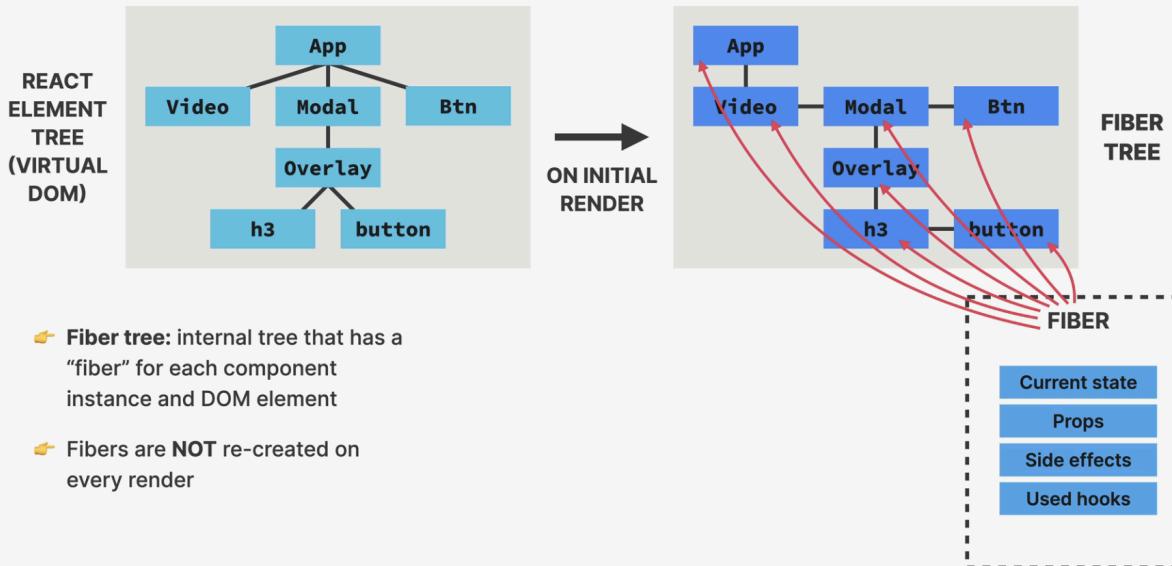


- 👉 Fiber tree: internal tree that has a "fiber" for each component instance and DOM element

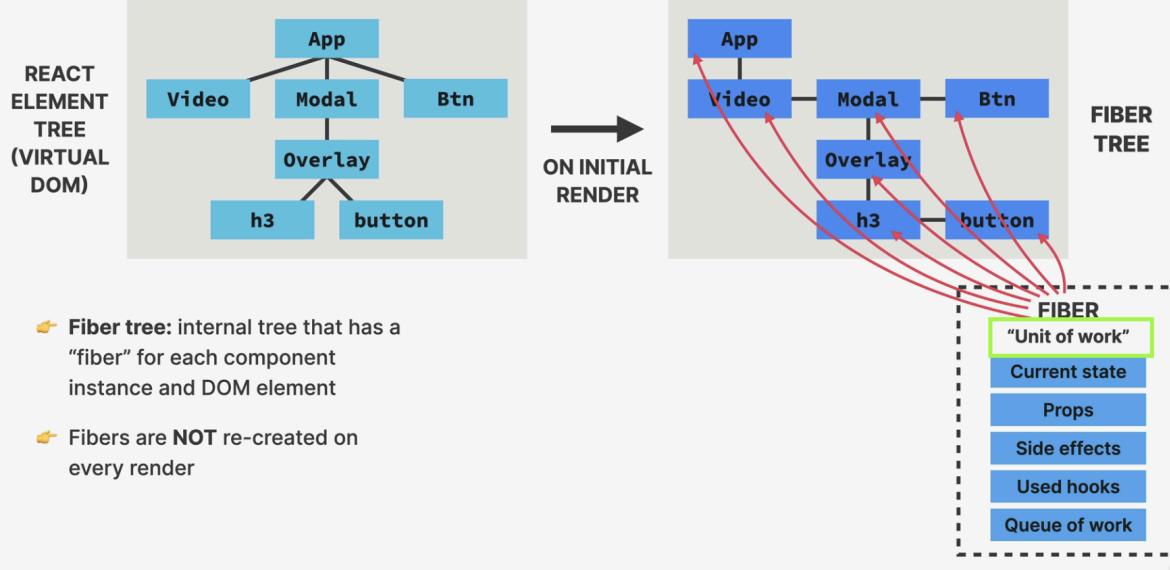
This makes Fibers the perfect place for keeping track of things like the current component state, props, side effects, list of used hooks, and more.

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.

## THE RECONCILER: FIBER



## THE RECONCILER: FIBER



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.

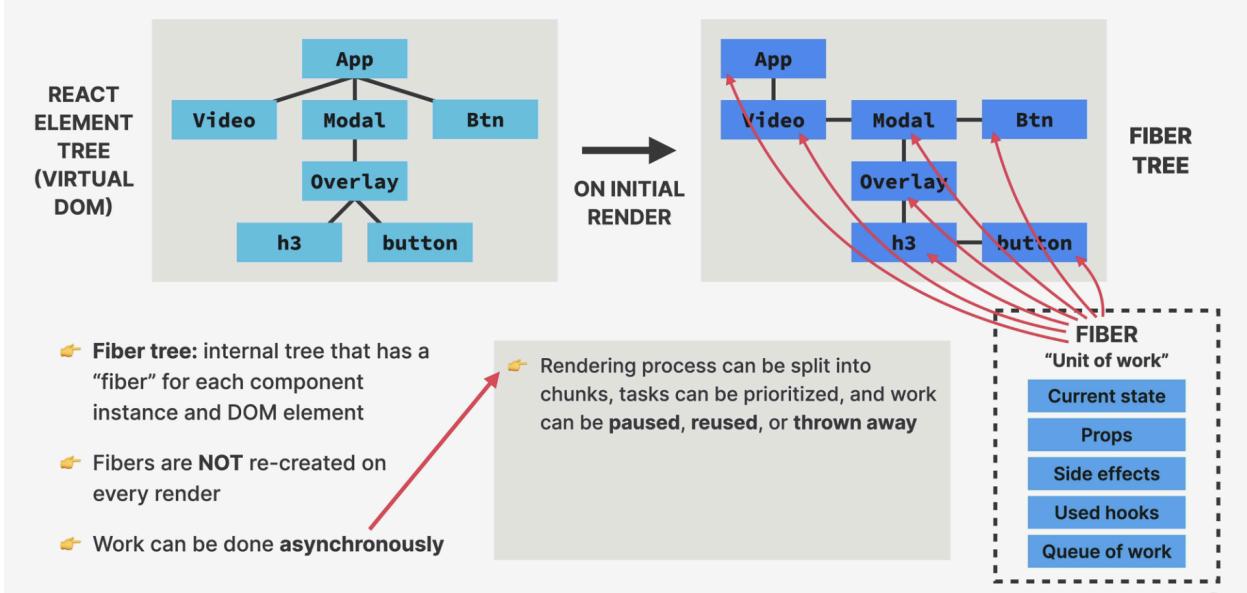
Now, if we take a quick look at the Fiber tree we will see that the Fibers are arranged in a different way than the elements in the React element tree.

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.

## THE RECONCILER: FIBER

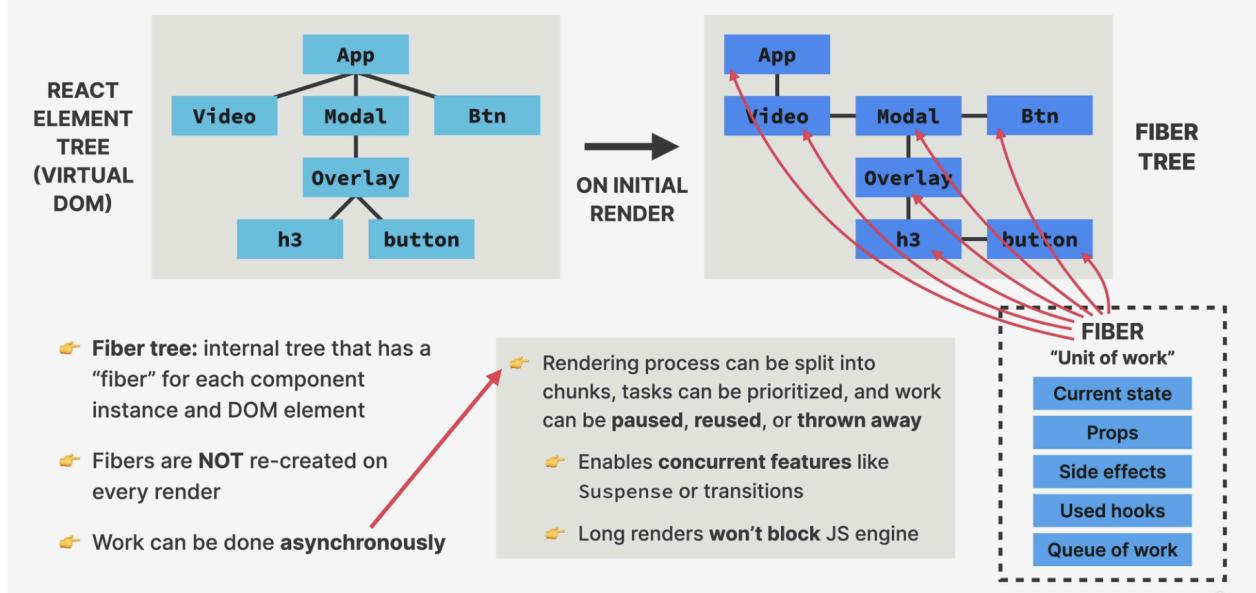


Now going back to the idea that Fibers are units of work, one extremely important characteristic of the Fiber reconciler is that work can be performed **asynchronously**.

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.

## THE RECONCILER: FIBER



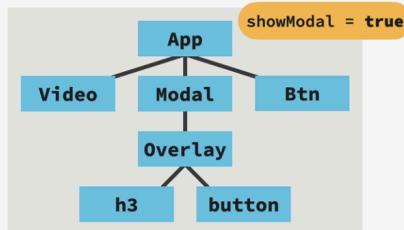
It's completely invisible to us developers. There are, however, also some practical uses of this asynchronous rendering because it enables modern so-called concurrent features like Suspense or transitions starting in React 18.

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.

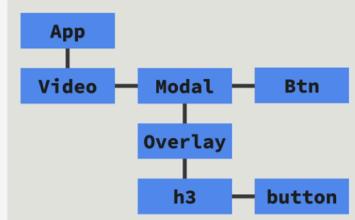
## Reconciliation in action

### RECONCILIATION IN ACTION



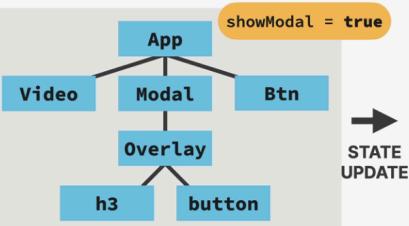
```

<App>
  <Video />
  [showModal] && (
    <Modal>
      <Overlay>
        <h3>Rate course!</h3>
        <button>5 ★</button>
      </Overlay>
    </Modal>
  )
  <Btn>
    [showModal] ? 'Rate' : 'Hide'
  </Btn>
</App>
  
```



CURRENT FIBER TREE

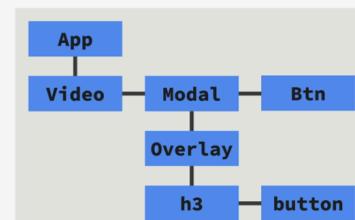
### RECONCILIATION IN ACTION



STATE UPDATE

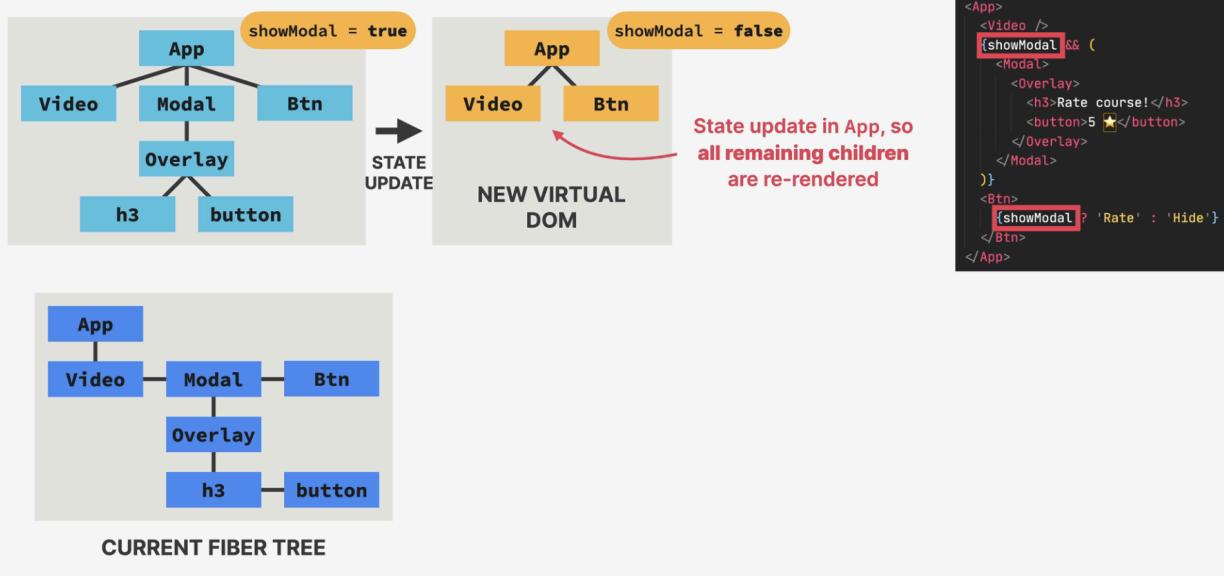
```

<App>
  <Video />
  [showModal] && (
    <Modal>
      <Overlay>
        <h3>Rate course!</h3>
        <button>5 ★</button>
      </Overlay>
    </Modal>
  )
  <Btn>
    [showModal] ? 'Rate' : 'Hide'
  </Btn>
</App>
  
```

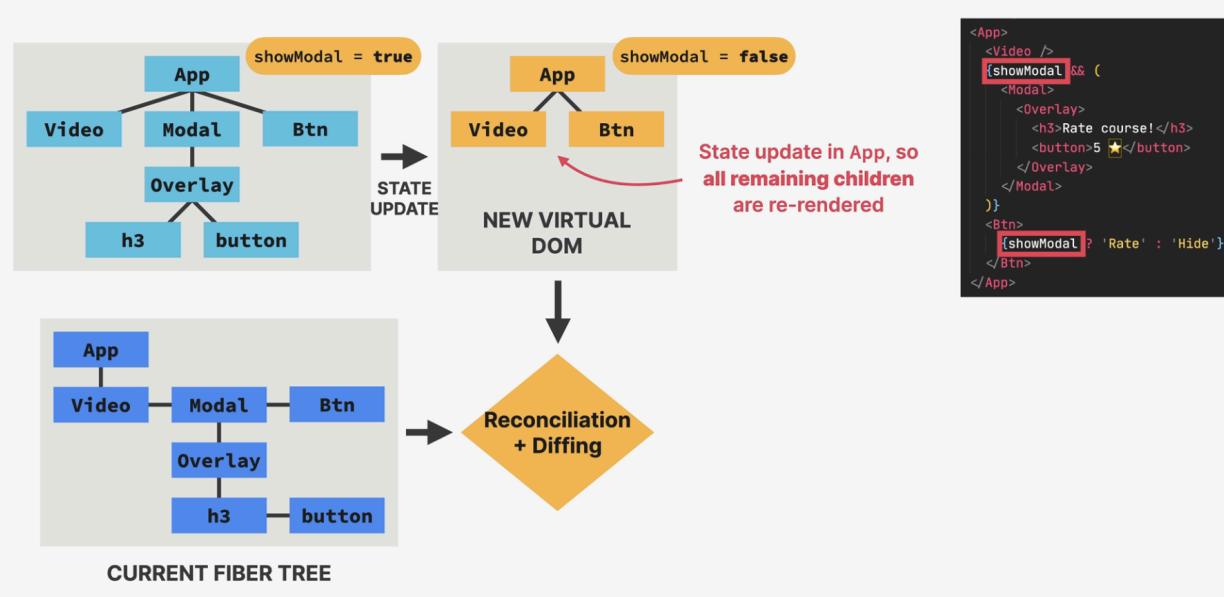


CURRENT FIBER TREE

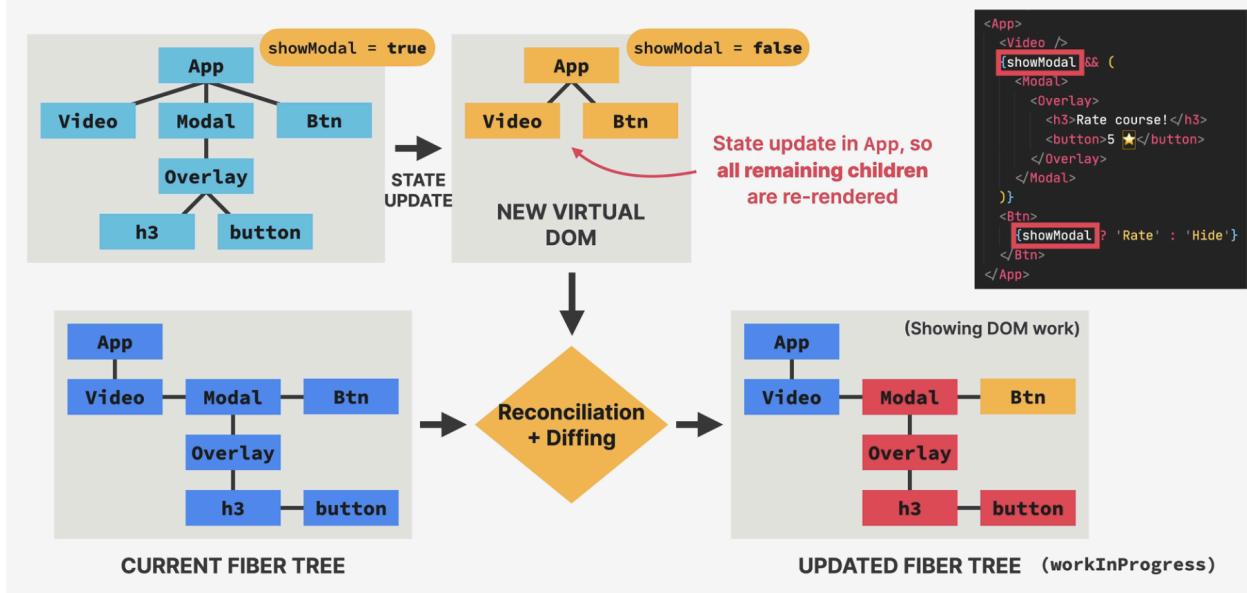
## RECONCILIATION IN ACTION



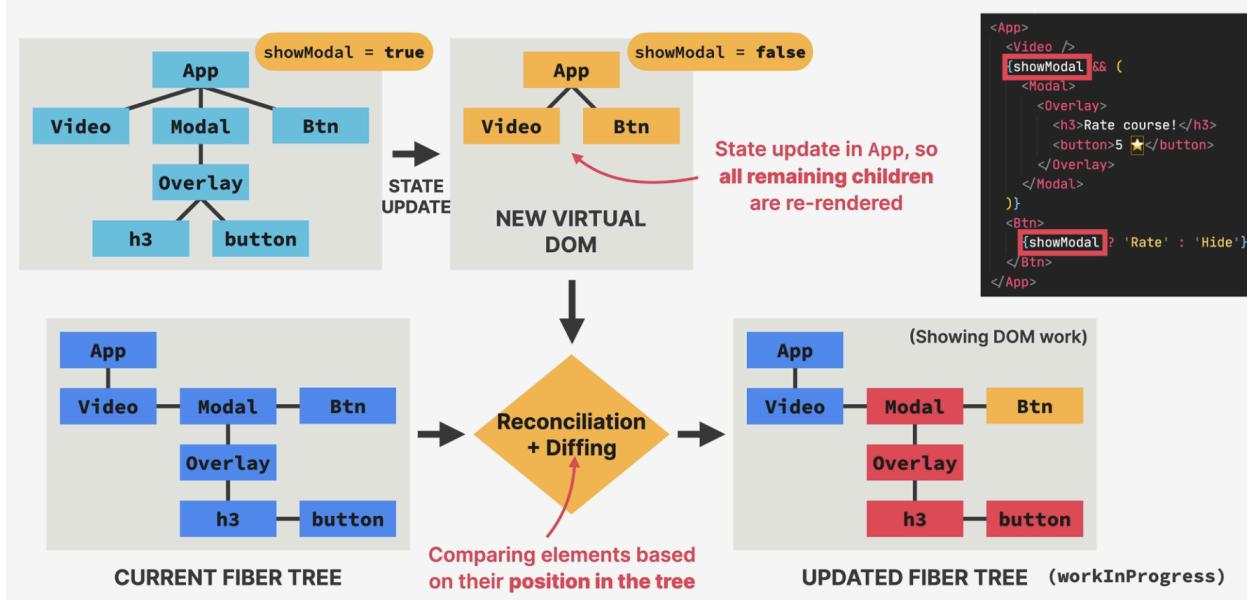
## RECONCILIATION IN ACTION



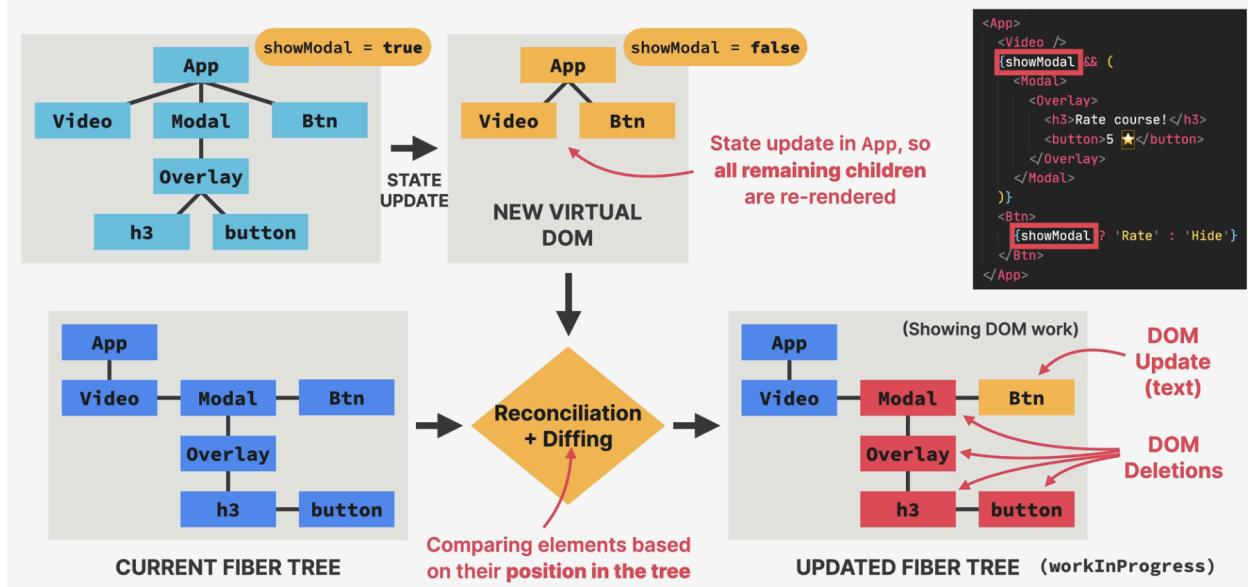
## RECONCILIATION IN ACTION



## RECONCILIATION IN ACTION



## RECONCILIATION IN ACTION



## THE RENDER PHASE

