

# 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)

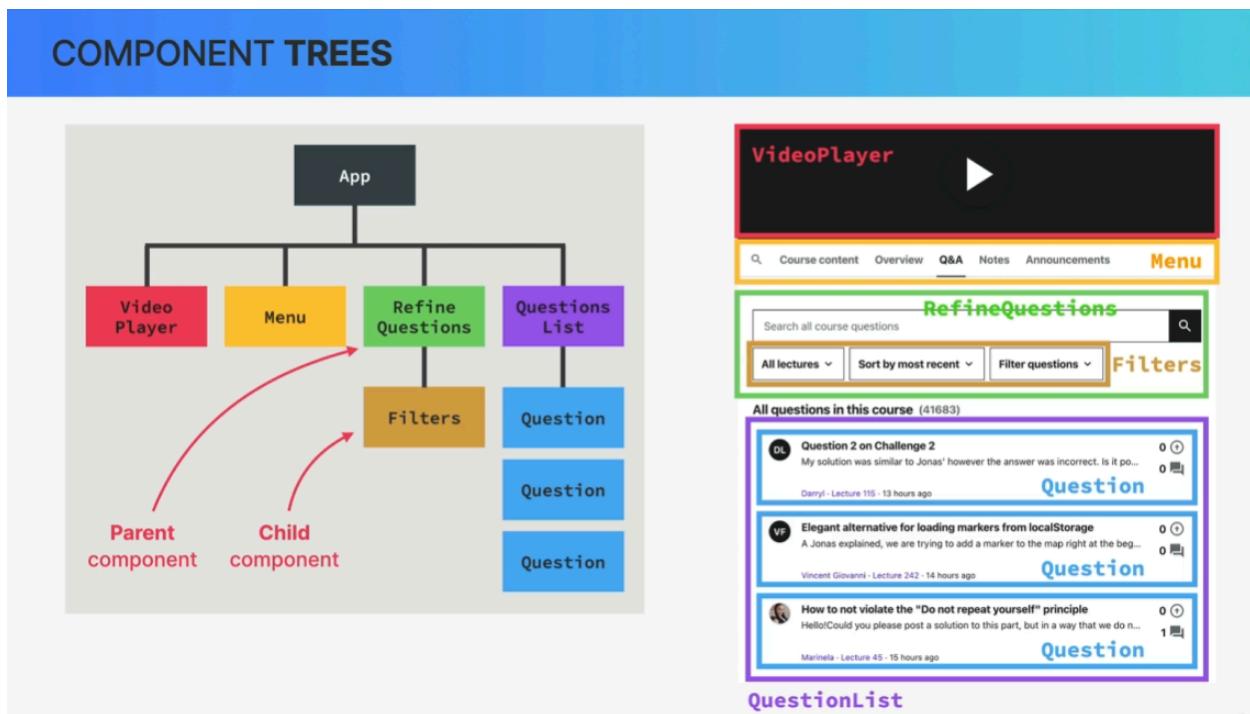
**Question 2 on Challenge 2**  
My solution was similar to Jonas' however the answer was incorrect. Is it po...  
Darryl - Lecture 115 - 13 hours ago  
0 0

**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

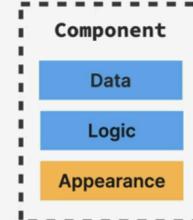
```
const title = document.querySelector("title")
const upvoteBtn = document.querySelector("btn")
title.textContent = "[0] ${question.title}"
let upvotes = 0;
upvoteBtn.addEventListener "click", function(){
  upvotes++;
  title.textContent =
    [${upvotes}] ${question.title};
  title.classList.add("upvoted");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

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>
```

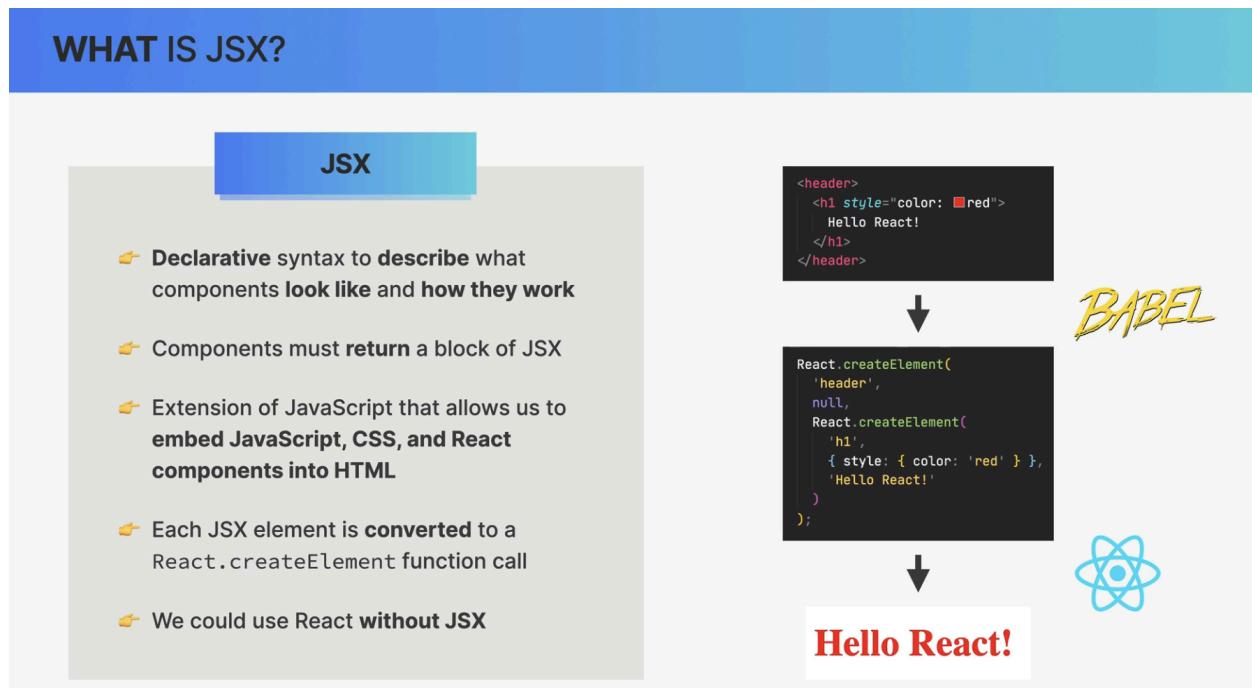


BABEL

```
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);

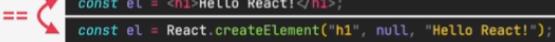
  return (
    <div>
      <h4>question.title</h4>
      <p>question.text</p>
      <UpvoteBtn
        onClick={upvote}
        upvotes={upvotes}
      />
    </div>
  );
}
```

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

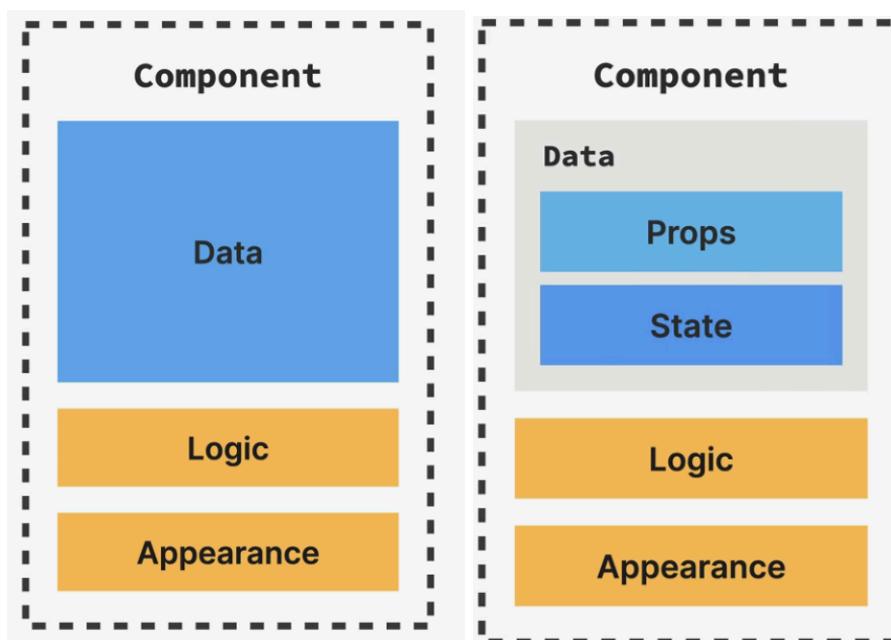
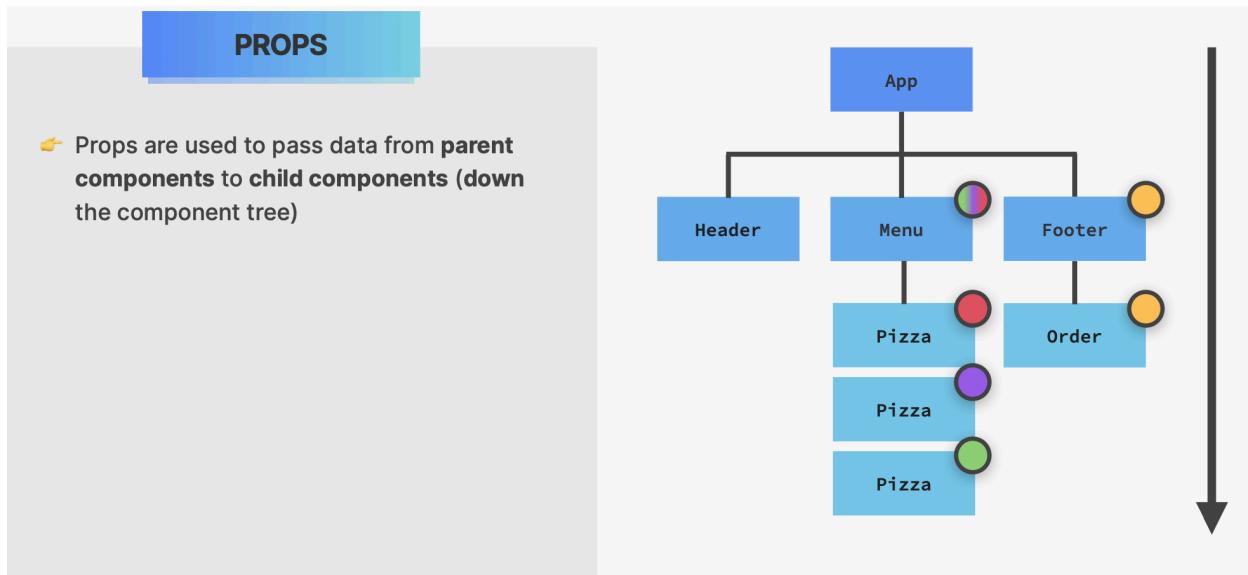
👉 CSS inline styles are written like this: {{<style>}}  
(to reference a variable, and then an object)

👉 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.

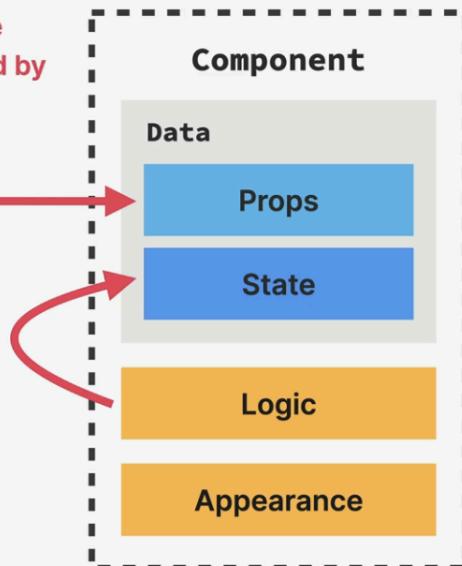


# PROPS ARE READ-ONLY!

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

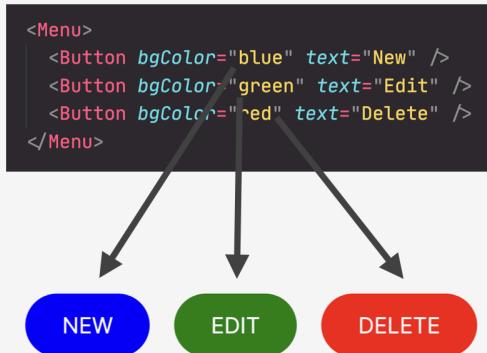
Parent Component

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!

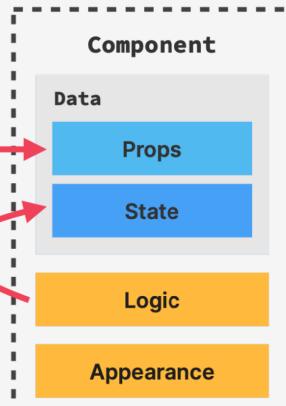
Props is data coming from the outside, and can only be updated by the parent component

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>
}
```



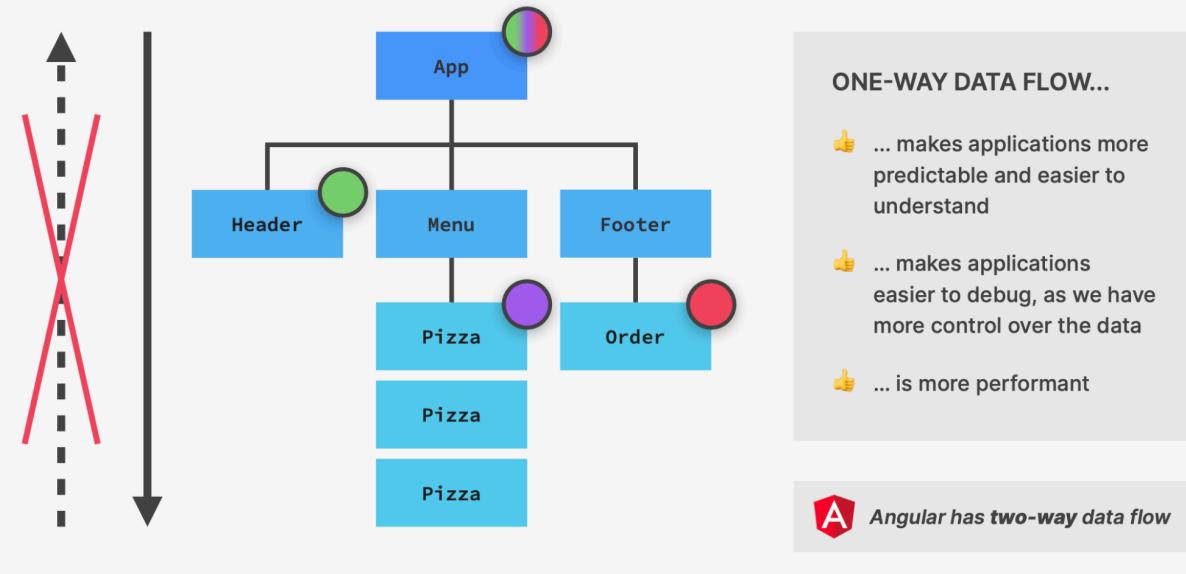
👉 Props are read-only, they are **immutable**! This is one of React's strict rules.

👉 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



## 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>
  );
}
```

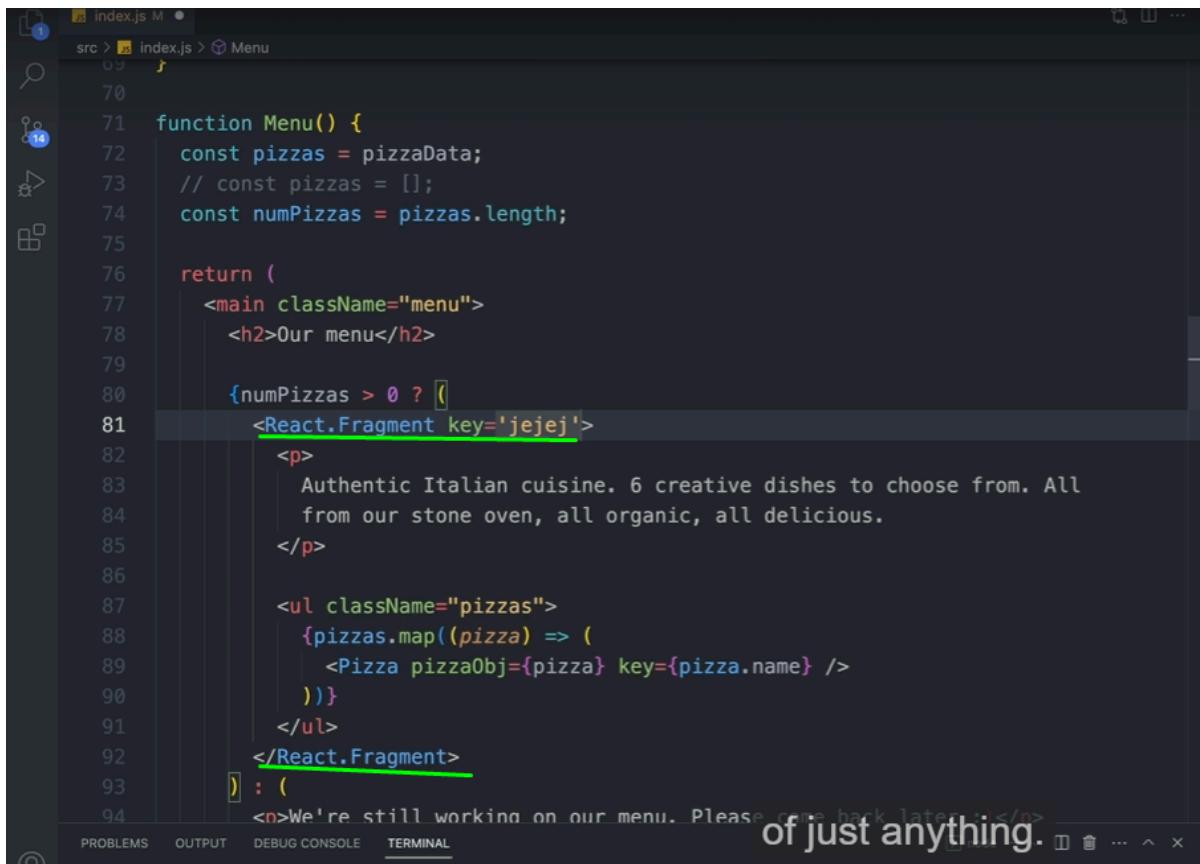
## React Fragments

```
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.
```

## Separation of concerns

### SEPARATION OF CONCERNST?

ONE TECHNOLOGY PER FILE

“Traditional” separation of concerns

**JS**

```
let upvotes = 0;
title.textContent = question.title;
const upvote = () => upvotes++;
upvoteBtn.addEventListener("click", upvote);

if (question.num > 0) {
  answer.classList.toggle("hidden");
  firstAnswer.classList.toggle("hidden");
}
```

**HTML 5**

```
<div>
  <h4 class="title"></h4>
  <button class="btn">Upvote</button>

  <div class="answer"></div>
  <div class="first hidden"></div>
</div>
```

Rise of interactive SPAs

This diagram illustrates the concept of "Traditional" separation of concerns. It shows two separate files: a JavaScript file on the left containing logic and a corresponding HTML file on the right containing UI. A red curved arrow points from the JS code to the HTML code, labeled "Traditional" separation of concerns.

### SEPARATION OF CONCERNST?

ONE TECHNOLOGY PER FILE

“Traditional” separation of concerns

**JS**

```
let upvotes = 0;
title.textContent = question.title;
const upvote = () => upvotes++;
upvoteBtn.addEventListener("click", upvote);

if (question.num > 0) {
  answer.classList.toggle("hidden");
  firstAnswer.classList.toggle("hidden");
}
```

**HTML 5**

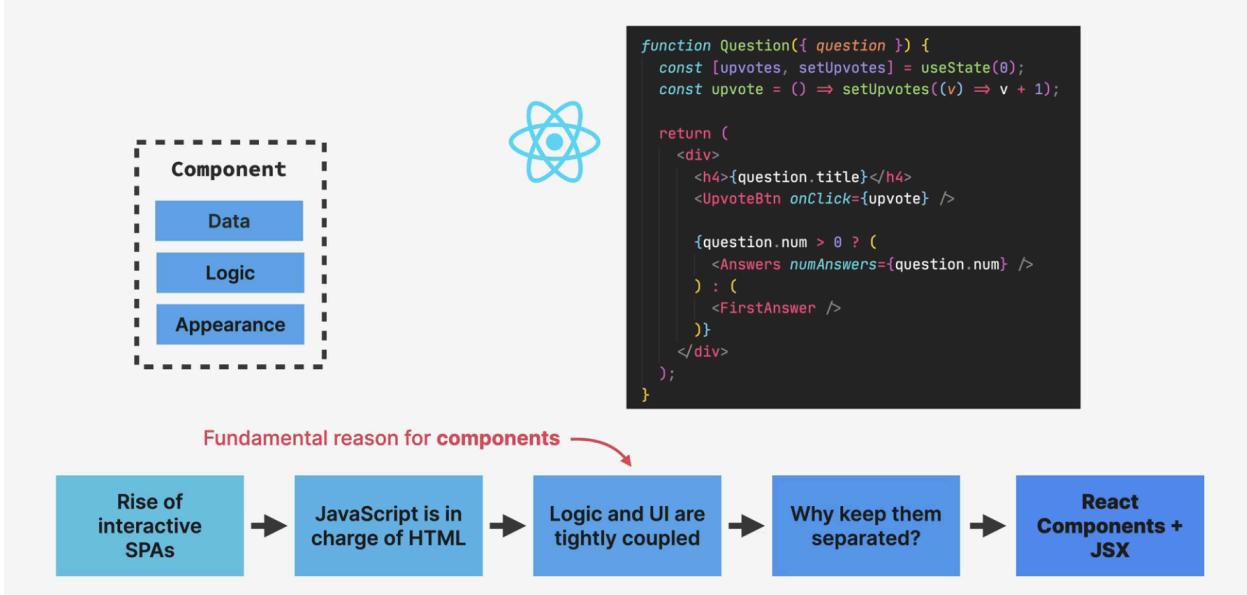
```
<div>
  <h4 class="title"></h4>
  <button class="btn">Upvote</button>

  <div class="answer"></div>
  <div class="first hidden"></div>
</div>
```

Rise of interactive SPAs → JavaScript is in charge of HTML → Logic and UI are tightly coupled → Why keep them separated? → React Components + JSX

This diagram illustrates the tight coupling between JavaScript and HTML in modern Single Page Applications. It shows the same JS and HTML files as the previous diagram, but with yellow arrows indicating bidirectional communication between them. The JS code has arrows pointing to the title and button elements in the HTML, and the HTML code has arrows pointing back to the JS variables and functions. This visualizes how the SPA pattern has shifted away from traditional separation towards a more integrated, coupled architecture.

## SEPARATION OF CONCERNS?

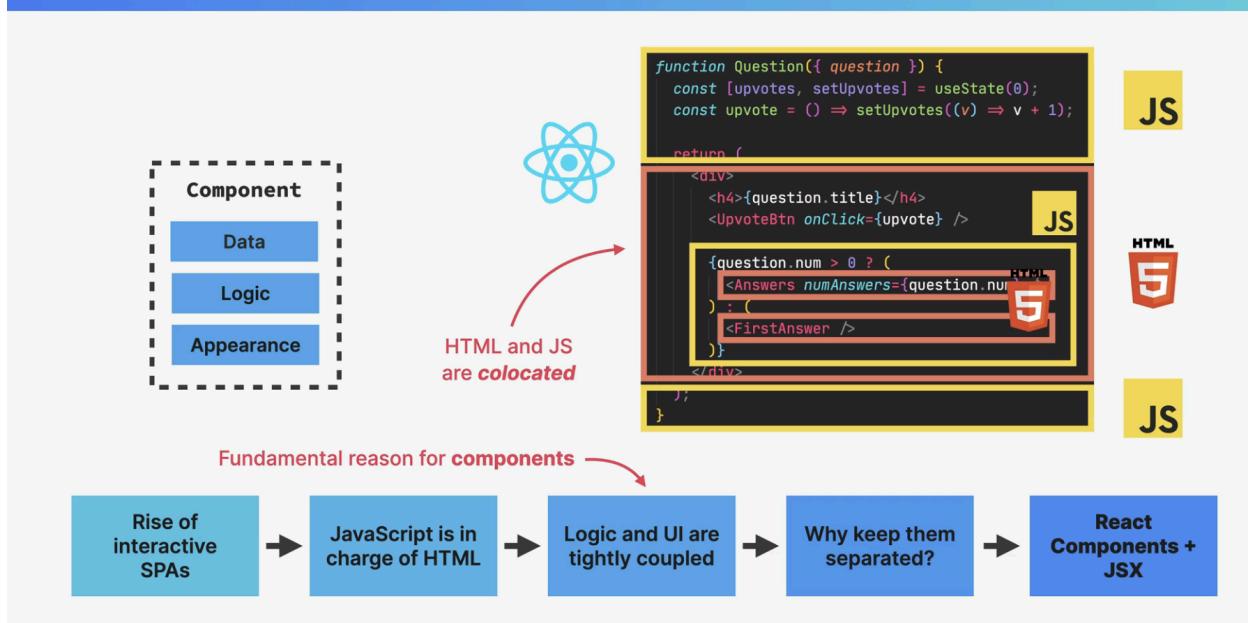


So content and logic are tightly coupled together and so it makes sense that they are co-located here.

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.

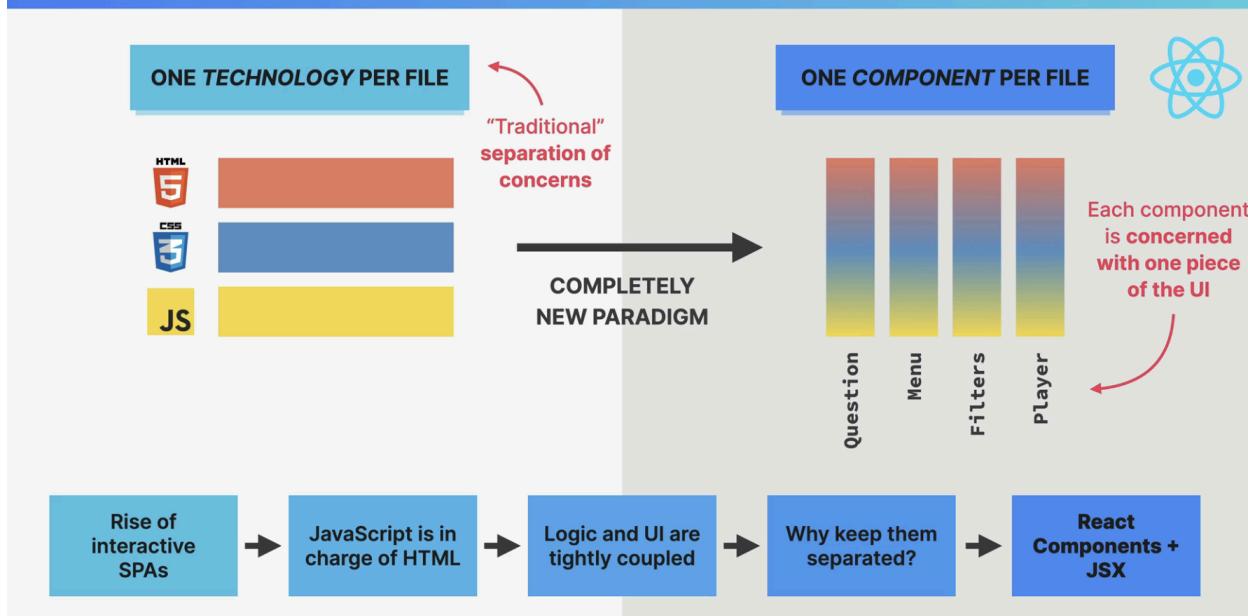
Because React does actually have a separation of concerns. It's just not one concern per file, as we had traditionally but one concern per component.

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**

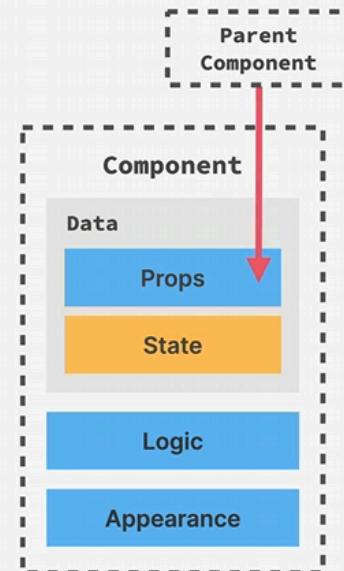
*(So we will keep learning about state throughout the entire course...)*

## 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 several red arrows pointing to specific elements:

- A red arrow points to the "Notifications" section, which shows 9+ new notifications.
- A red arrow points to the "Messages" section, which also shows 9+ new messages.
- A red arrow points to the search bar at the top right containing the text "javascr".
- A red arrow points to the "Notes" tab in the navigation bar, which is currently selected.
- A red arrow points to the "Shopping Cart" section, which displays two courses in the cart: "Node.js, Express, MongoDB & More: The Complete Bootcamp 2022" and "The Complete JavaScript Course 2022: From Zero to Expert!".

## 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

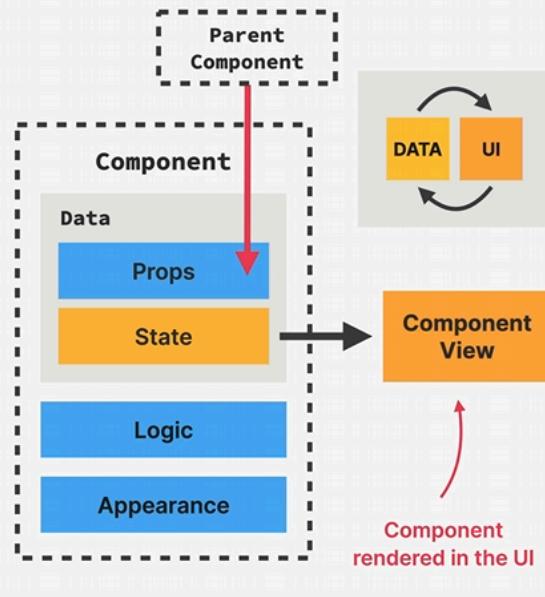
The screenshot shows a user interface with several red arrows pointing to specific elements:

- A red arrow points to the "Notifications" section, which shows 9+ new notifications.
- A red arrow points to the "Messages" section, which also shows 9+ new messages.
- A red arrow points to the search bar at the top right containing the text "javascr".
- A red arrow points to the "Notes" tab in the navigation bar, which is currently selected.
- A red arrow points to the "Shopping Cart" section, which displays two courses in the cart: "Node.js, Express, MongoDB & More: The Complete Bootcamp 2022" and "The Complete JavaScript Course 2022: From Zero to Expert!".

## 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?

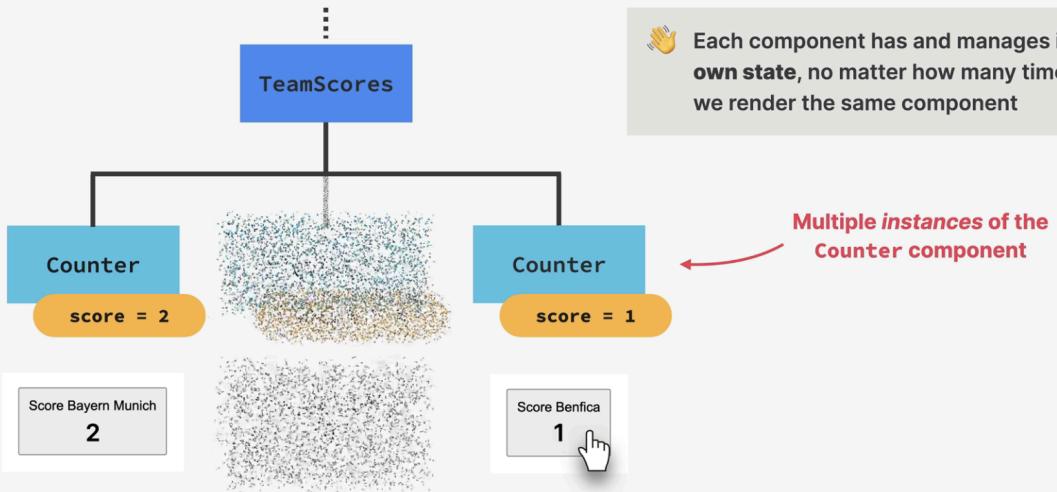
### STATE

- 👉 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

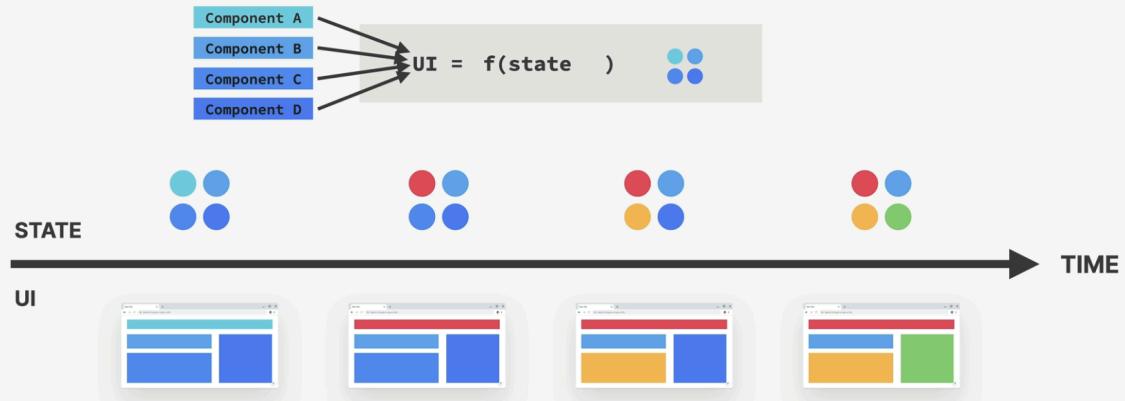
STATE ALLOWS DEVELOPERS TO:

- 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



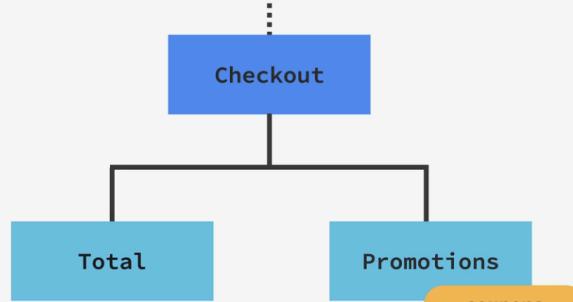
## IN PRACTICAL TERMS...

### 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** with a note about a discount of **€174.98** and **82% off**. On the left, a 'Promotions' component shows a message indicating a coupon has been applied: **'LEARNNEWSKILLS is applied'**, with a button labeled **'Apply'**. A double-headed arrow between the two components indicates they are sharing state. The 'Promotions' component also includes a 'coupons' section and a 'setCoupons' button.

## PROBLEM: SHARING STATE WITH SIBLING COMPONENT

ONE-WAY DATA FLOW

```
graph TD; Checkout --> Total; Checkout --> Promotions; Total --- coupons; Promotions --- setCoupons;
```

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

ONE-WAY DATA FLOW

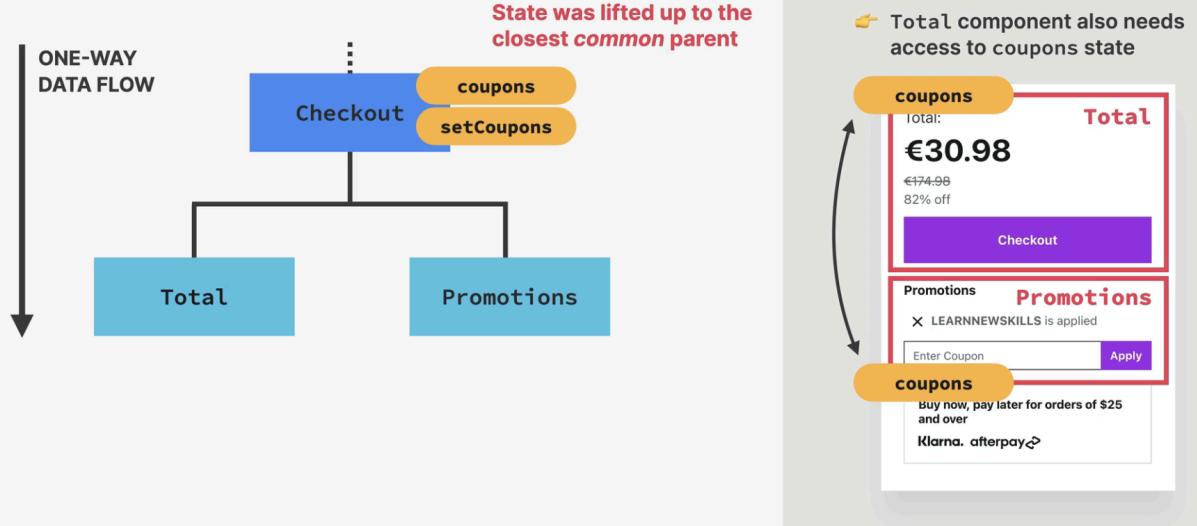
```
graph TD; Checkout --> Total; Checkout --> Promotions; Total --- coupons; Promotions --- setCoupons;
```

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

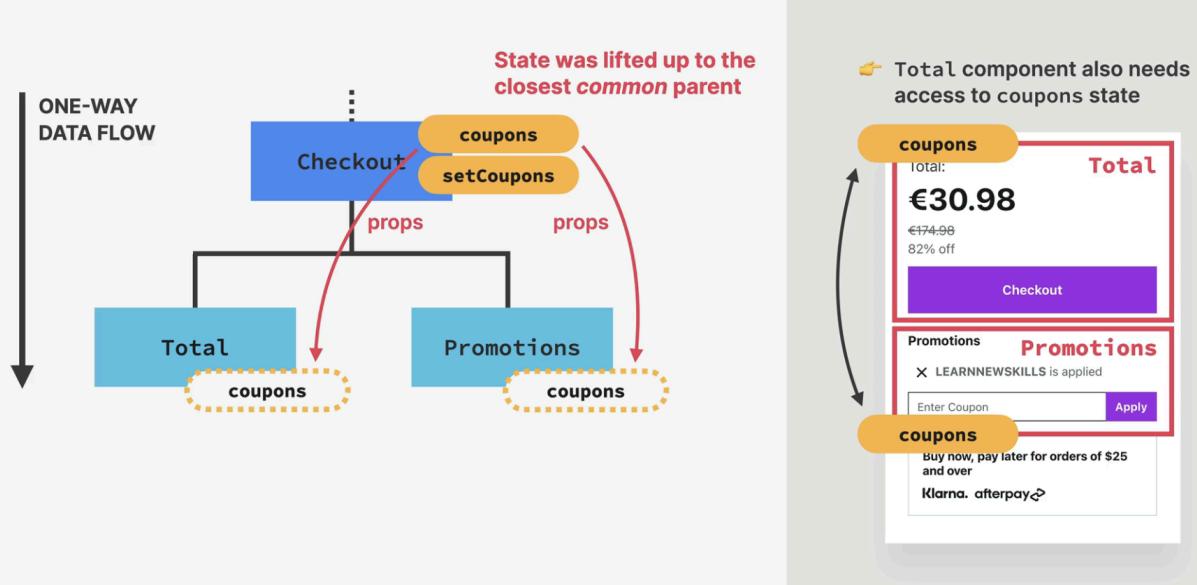
How do we share state with other components?

Total component also needs access to coupons state

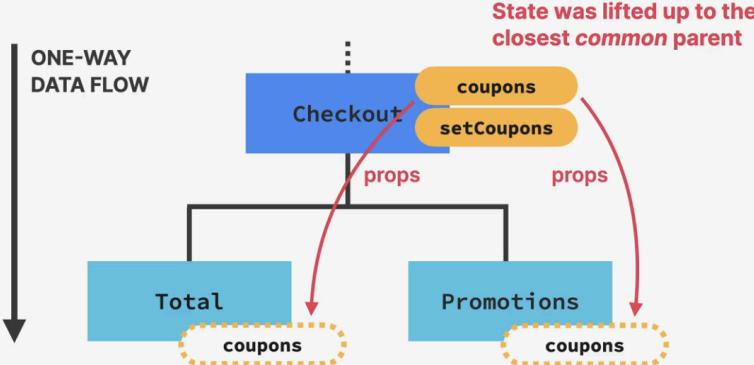
## SOLUTION: LIFTING STATE UP



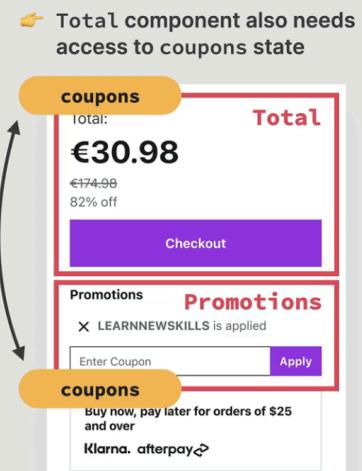
## SOLUTION: LIFTING STATE UP



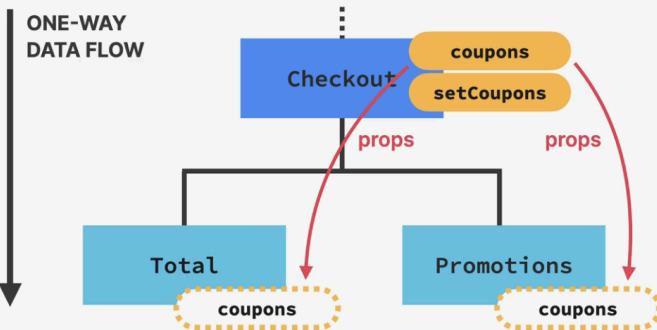
## SOLUTION: LIFTING STATE UP



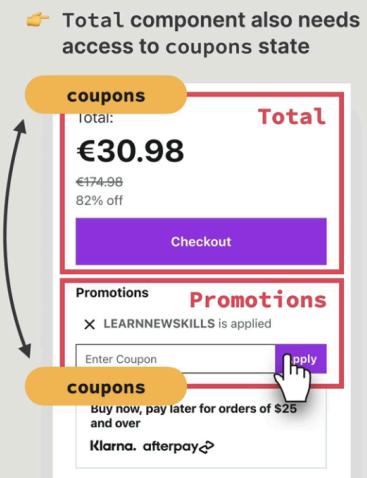
👉 By lifting state up, we have successfully shared one piece of state with multiple components in different positions in the component tree



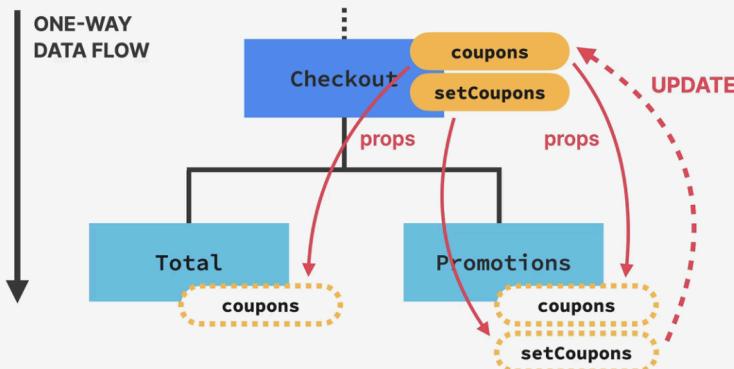
## CHILD-TO-PARENT COMMUNICATION



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



## CHILD-TO-PARENT COMMUNICATION

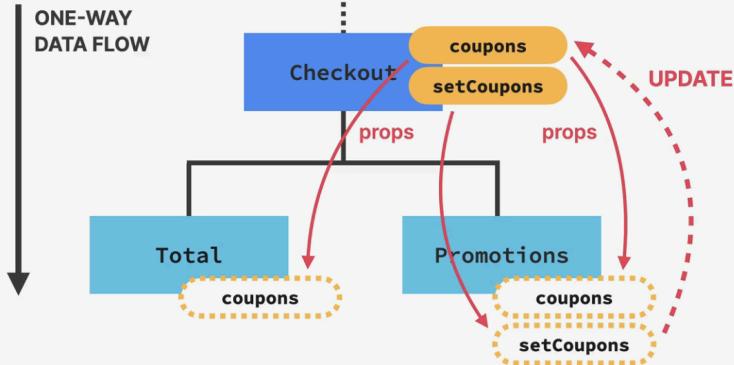


🤔 If data flows from parent to children, how can Promotions (child) update state in Checkout (parent)?

👉 Total component also needs access to coupons state

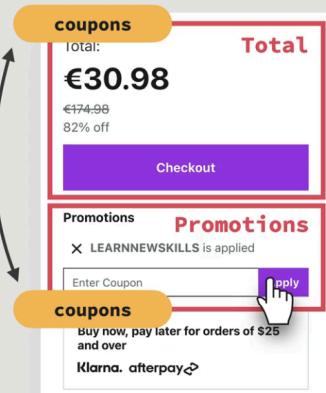


## CHILD-TO-PARENT COMMUNICATION



🤔 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; // ←
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="step" style={{ transform: `translateX(-${(step - 1) * 100}%)` }}>

```

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 () ↴

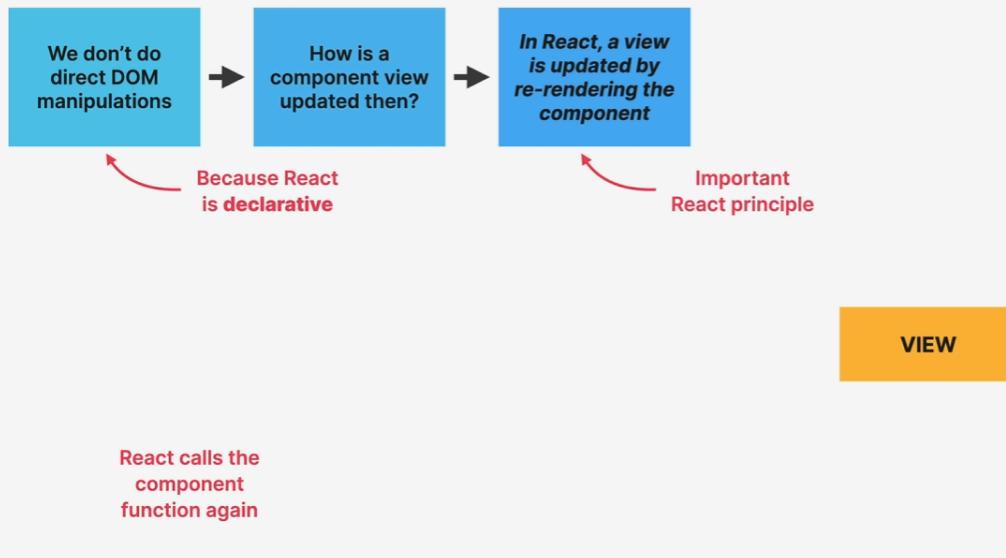
length: 2 ↴

[[Prototype]]: Array(0) ↴

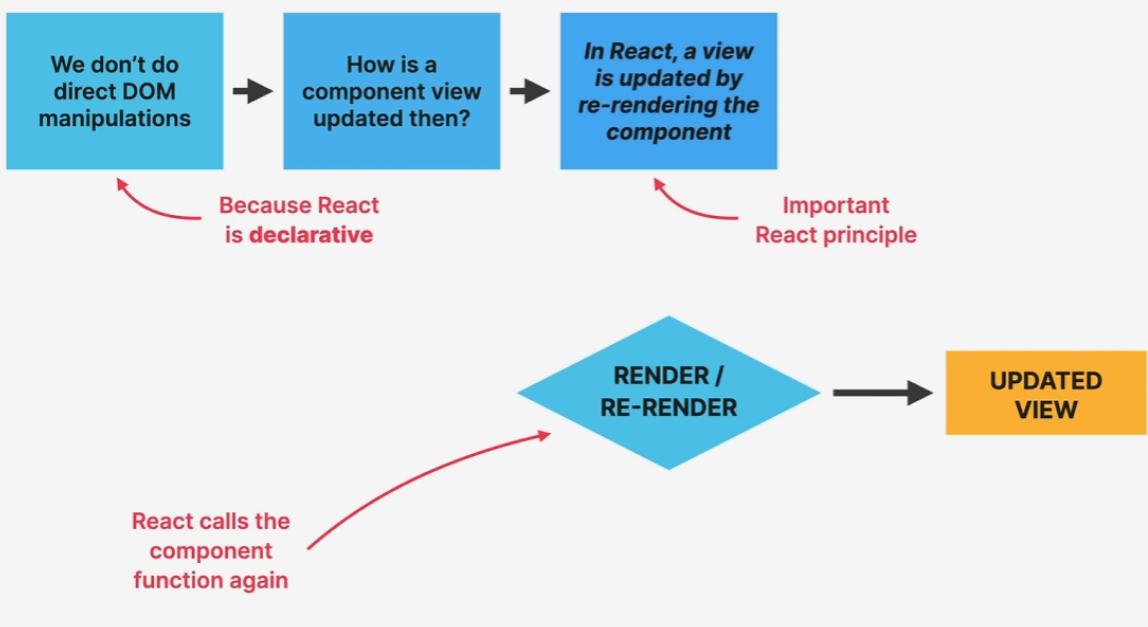
App.js:11

>

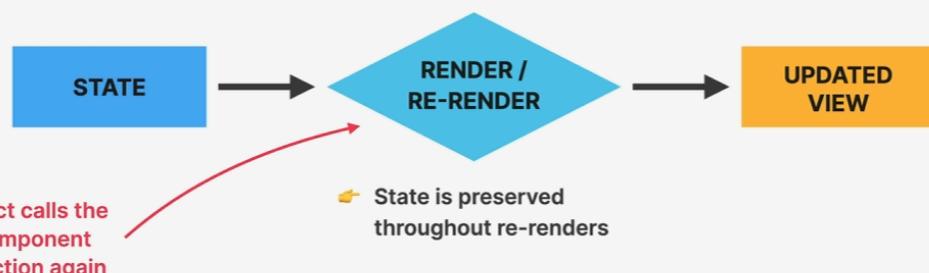
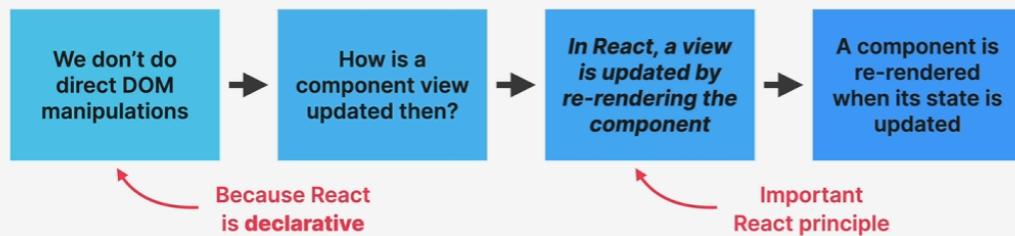
## THE MECHANICS OF STATE IN REACT



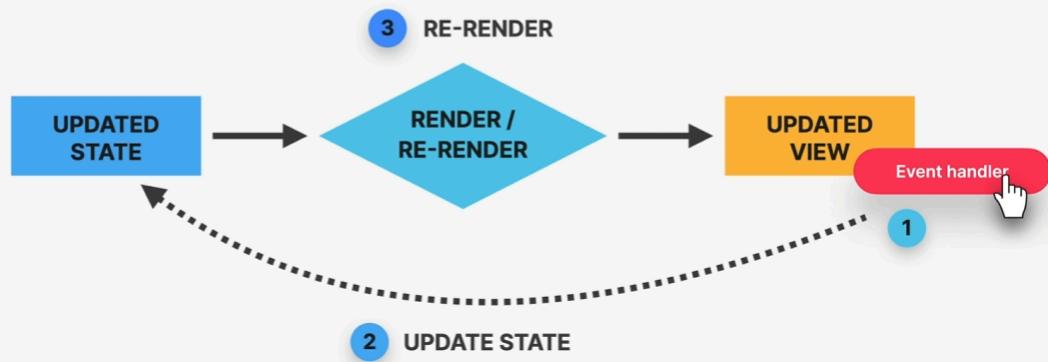
## THE MECHANICS OF STATE IN REACT



## 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



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

RE-RENDER

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

Quality beats quantity.

Get Advice

You have read 13 pieces of advice

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

UPDATE STATE

## 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) {
```

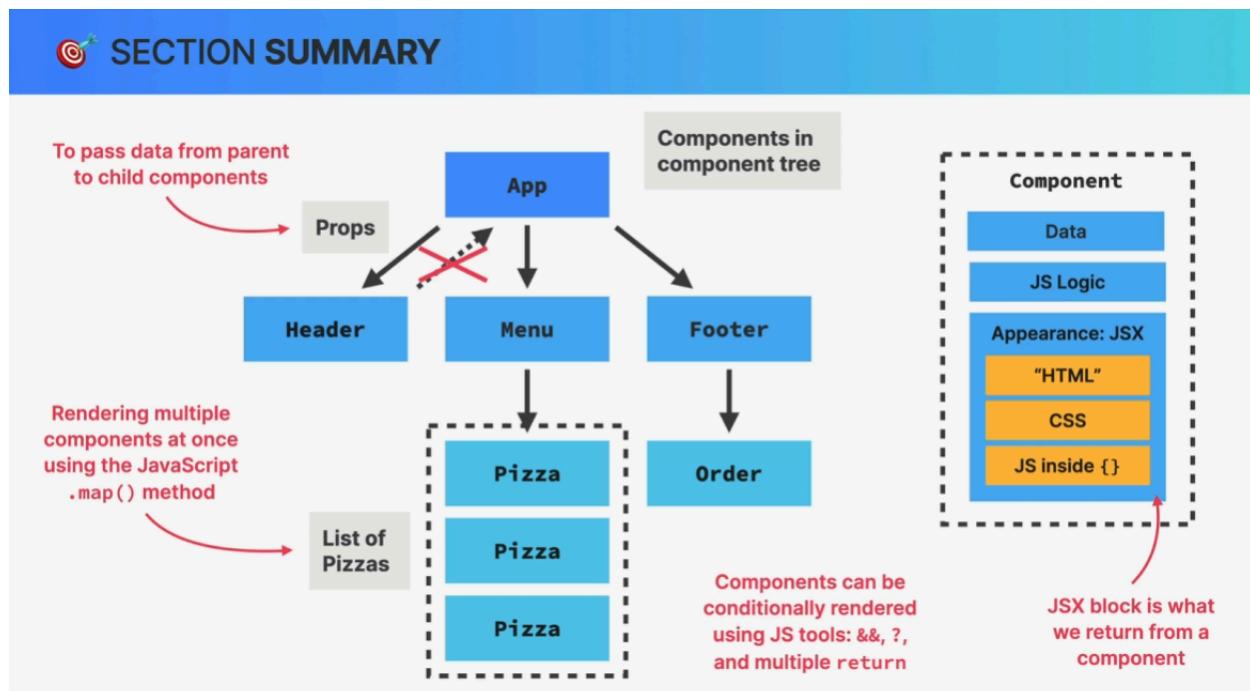
```

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" />
    );
}

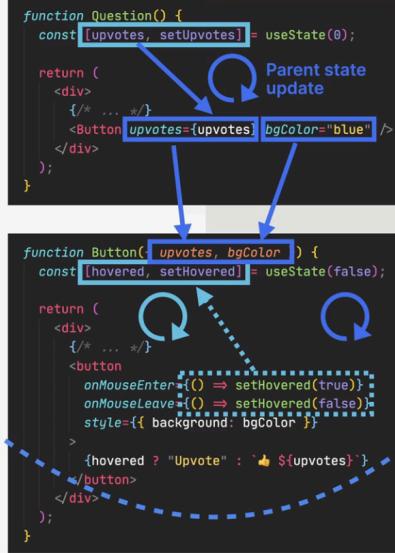
function Button({ upvotes, bgColor }) {
  const [hovered, setHovered] = useState(false);

  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

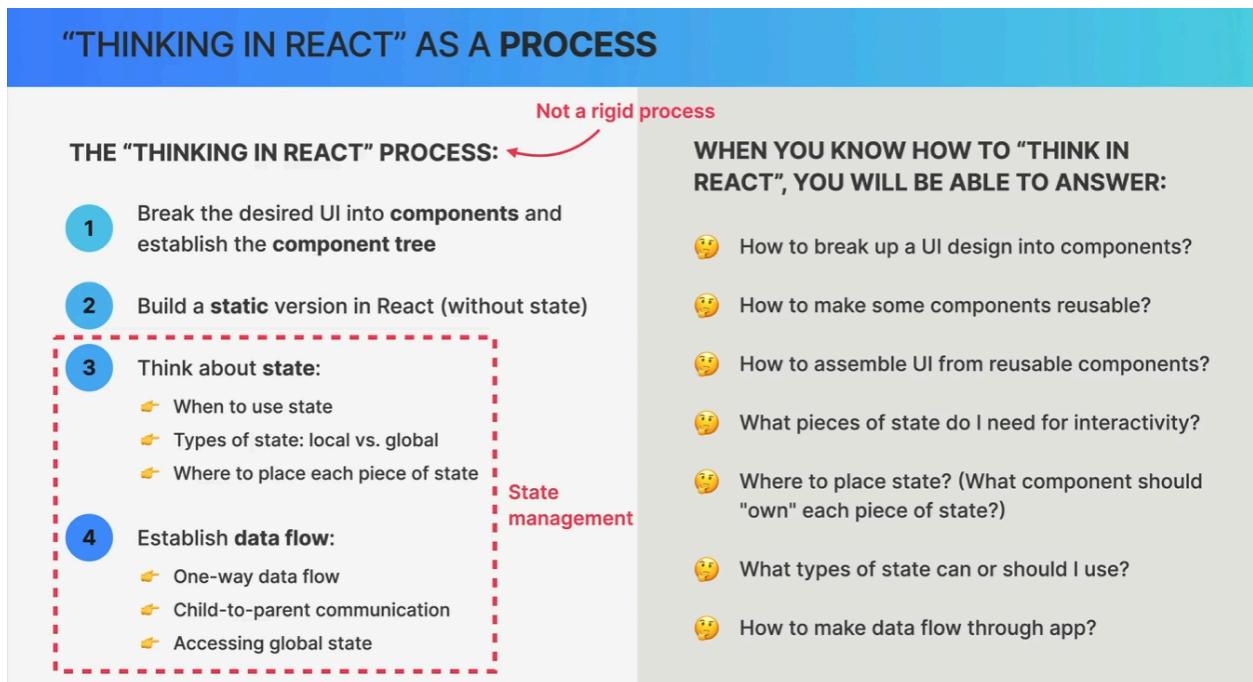
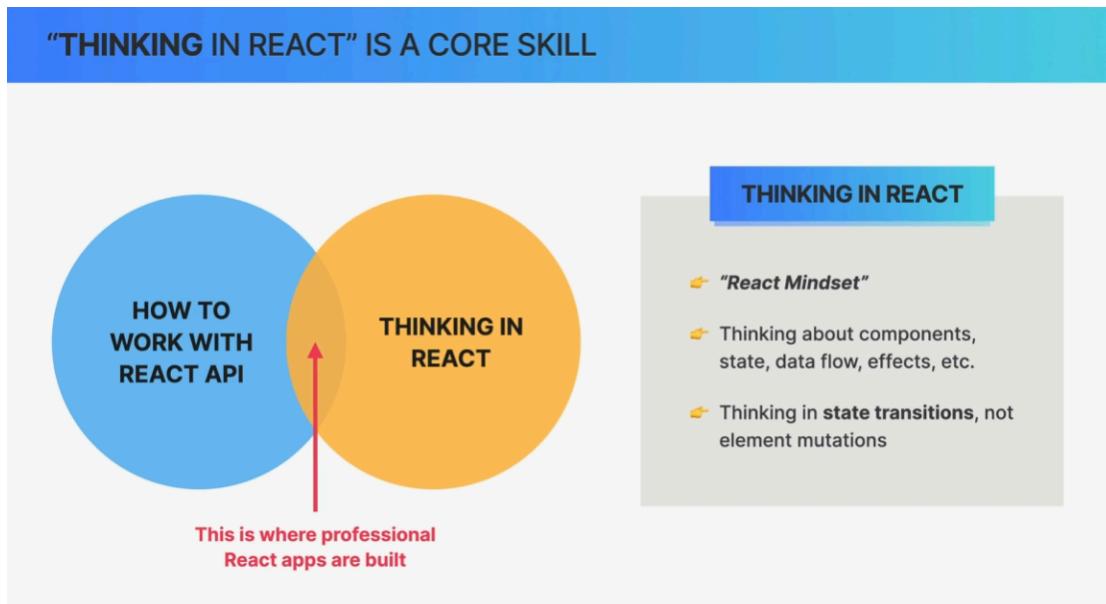


### PROPS

- 👉 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)

udemy Categories  Categories

Shopping Cart Local state

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

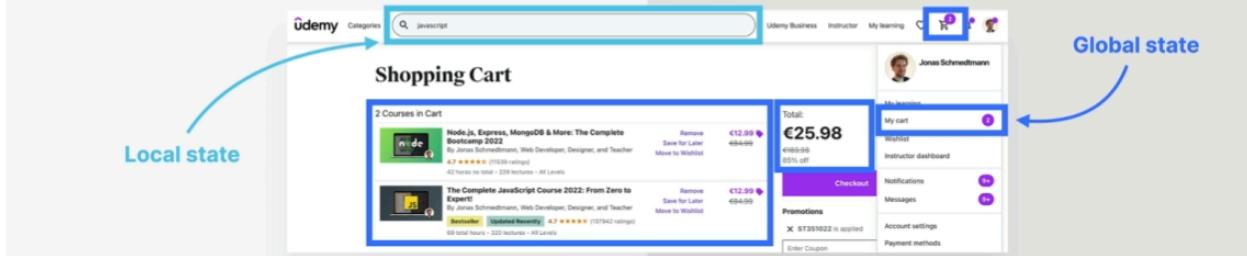
### LOCAL STATE

- 👉 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



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

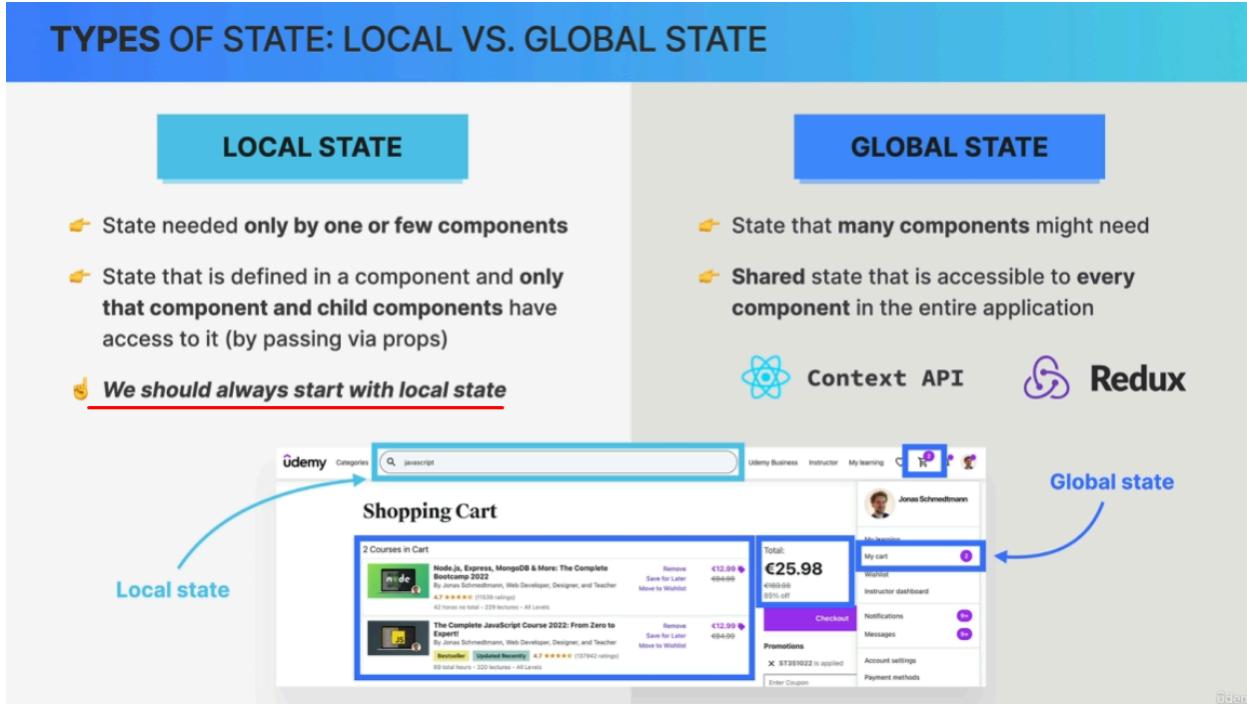
### LOCAL STATE

- 👉 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**

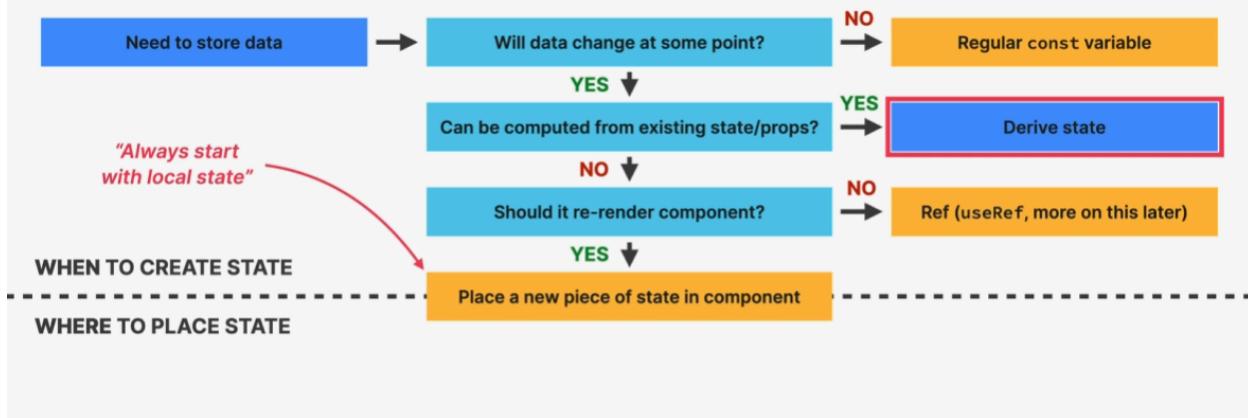
Local state

### GLOBAL STATE

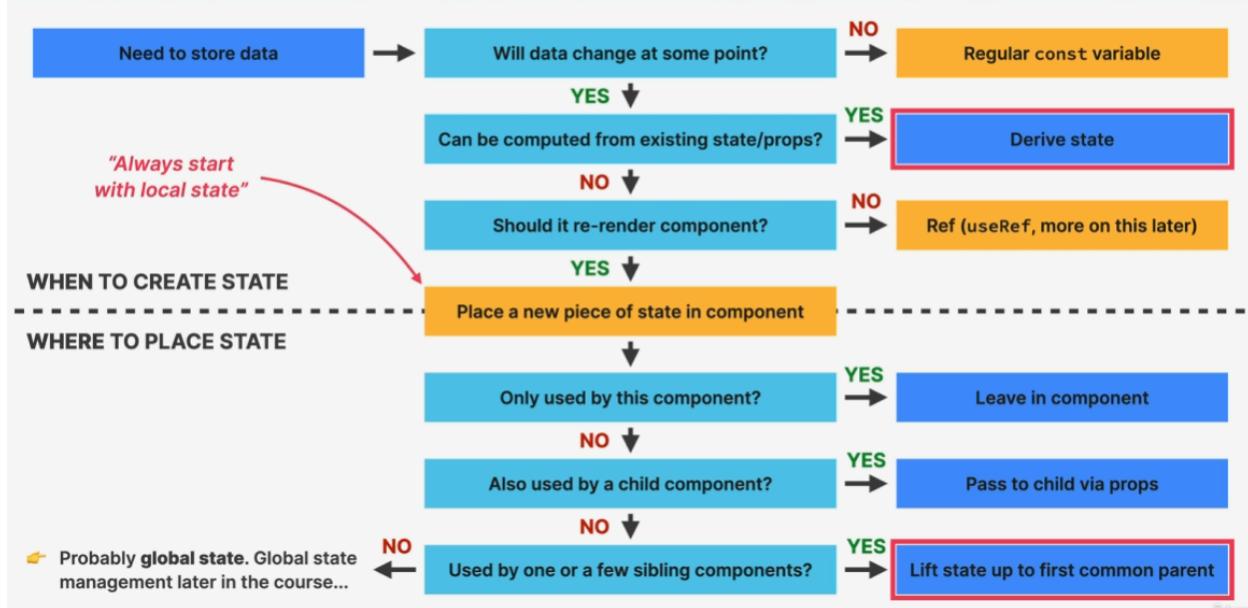
- 👉 State that **many components** might need
- 👉 Shared state that is accessible to **every component** in the entire application



## 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. The text reads: 'An empty "hole" that can be filled by any JSX the component receives as children'.

### 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. The text reads: 'An empty "hole" that can be filled by any JSX the component receives as children'. To the right, there is a purple button labeled 'Previous' with a left-pointing arrow icon. Below it is some JSX code: 

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

 The span element containing the arrow icon is highlighted with a red box.

### THE CHILDREN PROP

**Button**

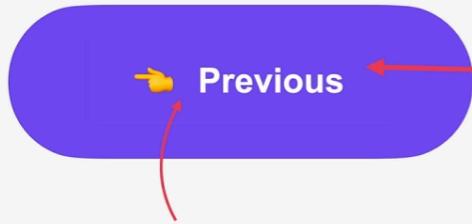
A diagram illustrating the concept of the 'children' prop. It shows a purple rounded rectangle labeled 'Button'. Inside it is a white rectangular box labeled 'props.children'. A red arrow points from the text below to this box. The text reads: 'An empty "hole" that can be filled by any JSX the component receives as children'. To the right, there is a purple button labeled 'Previous' with a left-pointing arrow icon. Below it is some JSX code: 

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

 The span element containing the arrow icon is highlighted with a red box. A red arrow points from the text 'Children of Button, accessible through props.children' to the highlighted span element in the code.

## THE CHILDREN PROP

### Button



Children of Button,  
accessible through  
props.children

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

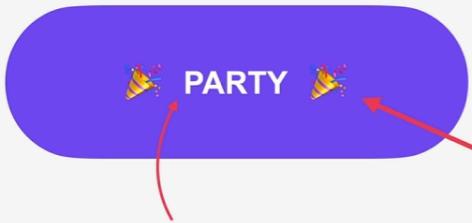
An empty "hole" that can be filled  
by any JSX the component  
receives as children



```
<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)