



# TABLE OF CONTENTS: THEORY LECTURES (CLICK THE TITLES)

- [1 Watch Before You Start!](#)
- [2 Why Do front-end Frameworks exist?](#)
- [3 What is React?](#)
- [4 Setting Up a New React Project: The Two Options](#)
- [5 Components as Building Blocks](#)
- [6 What is JSX?](#)
- [7 Separation of Concerns](#)
- [8 Props, Immutability, and One-Way Data Flow](#)
- [9 The Rules of JSX](#)
- [10 Section Summary](#)
- [11 What is State in React?](#)
- [12 The Mechanics of State](#)
- [13 More Thoughts About State + State Guidelines](#)
- [14 State vs. Props](#)
- [15 What is "Thinking in React"?](#)
- [16 Fundamentals of State Management](#)
- [17 Derived State](#)
- [18 The children Prop: Making a Reusable Button](#)
- [19 How to Split a UI Into Components](#)
- [20 Component Categories](#)
- [21 Component Composition](#)
- [22 Props as a Component API](#)
- [23 Components, Instances, and Elements](#)
- [24 How Rendering Works: Overview](#)
- [25 How Rendering Works: The Render Phase](#)
- [26 How Rendering Works: The Commit Phase](#)
- [27 How Diffing Works](#)
- [28 The Key Prop](#)
- [29 Rules for Render Logic: Pure components](#)
- [30 State Update Batching](#)
- [31 How Events Work in React](#)
- [32 Libraries vs. Frameworks & The React Ecosystem](#)
- [33 Section Summary: Practical Takeaways](#)
- [34 The Component Lifecycle](#)
- [35 A First Look at Effects](#)
- [36 The useEffect Dependency Array](#)
- [37 The useEffect Cleanup Function](#)
- [38 React Hooks and Their Rules](#)
- [39 useState Summary](#)



# TABLE OF CONTENTS: THEORY LECTURES (CLICK THE TITLES)

- [40 Introducing Another Hook: useRef](#)
- [41 What are Custom Hooks? When to Create One?](#)
- [42 Class Components vs. Function Components](#)
- [43 Managing State With useReducer](#)
- [44 Section Summary: useState vs. useReducer](#)
- [45 Routing and Single-Page Applications \(SPAs\)](#)
- [46 Styling Options For React Applications](#)
- [47 What is the Context API?](#)
- [48 Thinking In React: Advanced State Management](#)
- [49 Performance Optimization and Wasted Renders](#)
- [50 Understanding memo](#)
- [51 Understanding useMemo and useCallback](#)
- [52 Optimizing Bundle Size With Code Splitting](#)
- [53 Don't Optimize Prematurely!](#)
- [54 useEffect Rules and Best Practices](#)
- [55 Introduction to Redux](#)
- [56 Redux Middleware and thunks](#)
- [57 What is Redux Toolkit \(RTK\)?](#)
- [58 Redux vs. Context API](#)
- [59 Application Planning \("Fast React Pizza Co."\)](#)
- [60 What is Tailwind CSS?](#)
- [61 Application Planning \("The Wild Oasis"\)](#)
- [62 What is Supabase?](#)
- [63 Modeling Application State](#)
- [64 What is React Query?](#)
- [65 An Overview of Reusability in React](#)

# PART 01

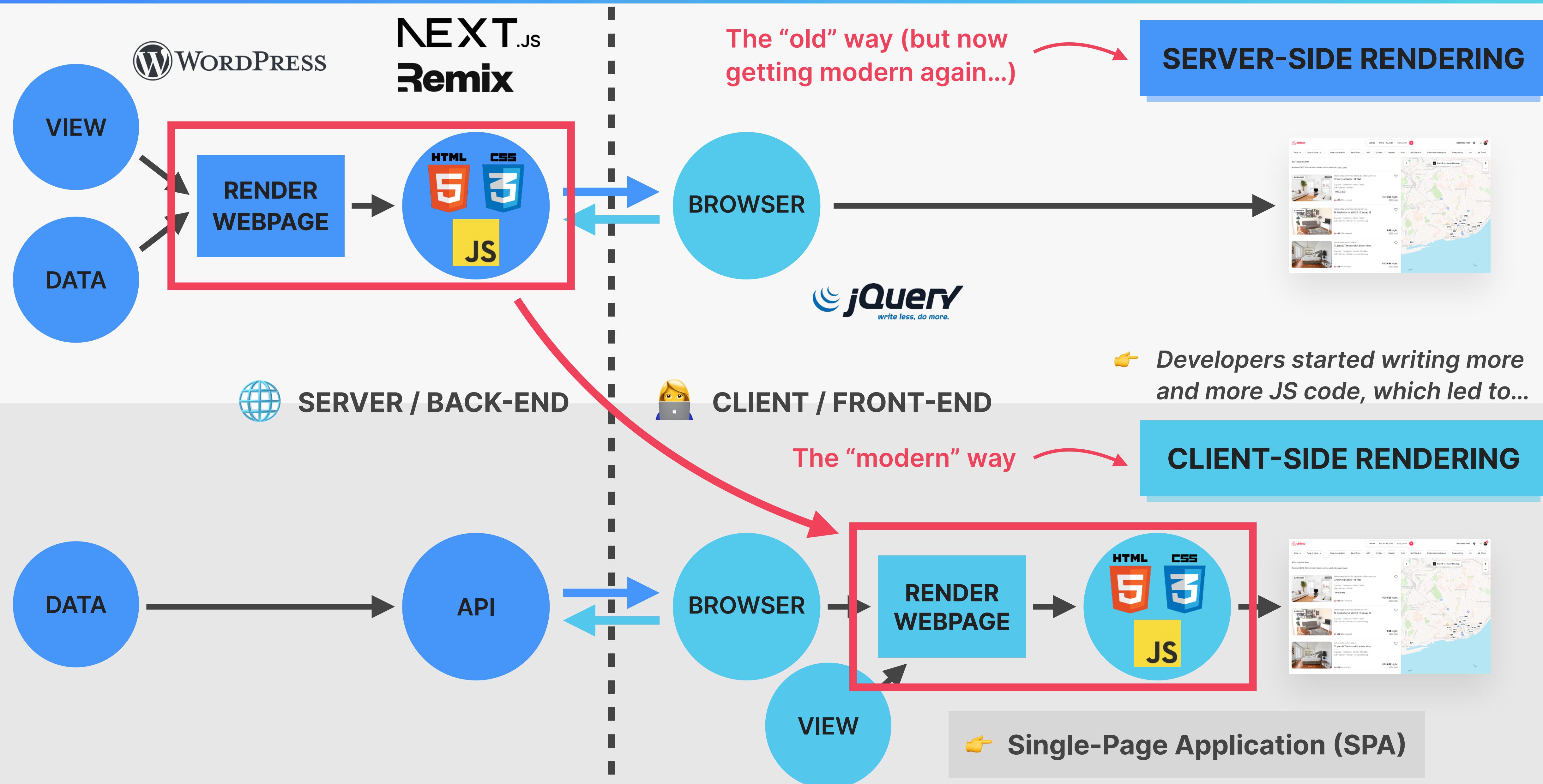
---

# REACT FUNDAMENTALS

# A FIRST LOOK AT REACT



# THE RISE OF SINGLE-PAGE APPLICATIONS



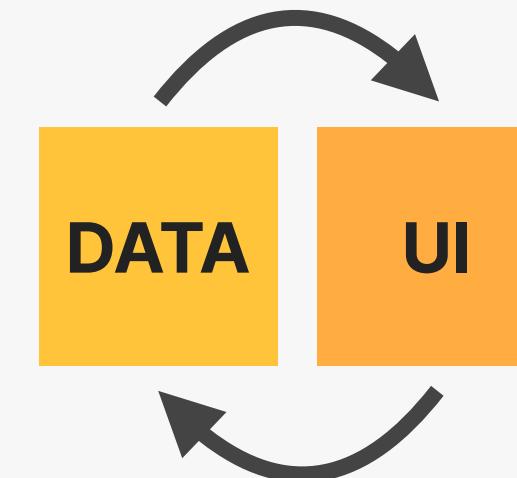
# SINGLE-PAGE APPLICATIONS WITH VANILLA JAVASCRIPT?

👉 *Front-end web applications are all about...*

**Handling data + displaying  
data in a user interface**

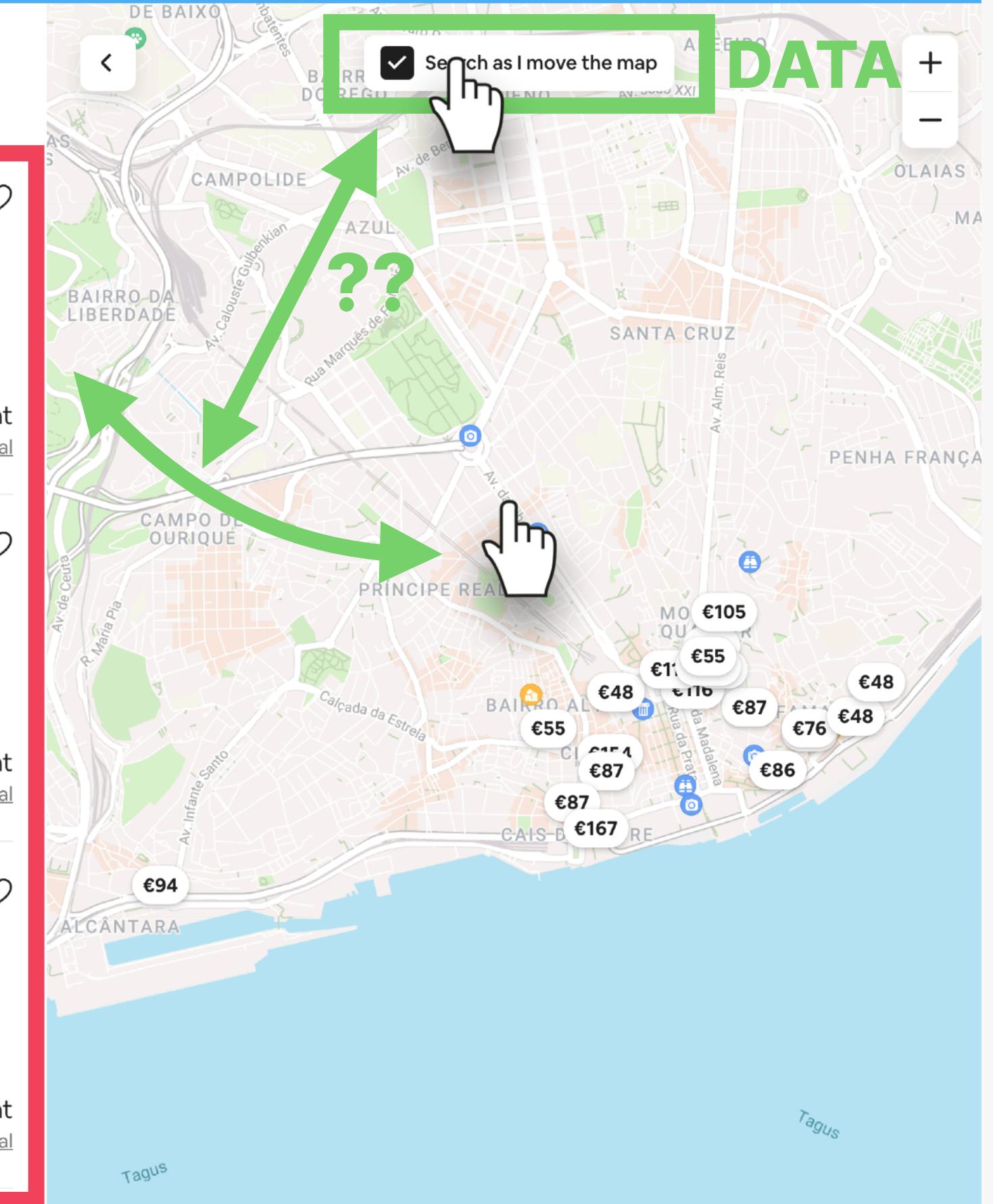
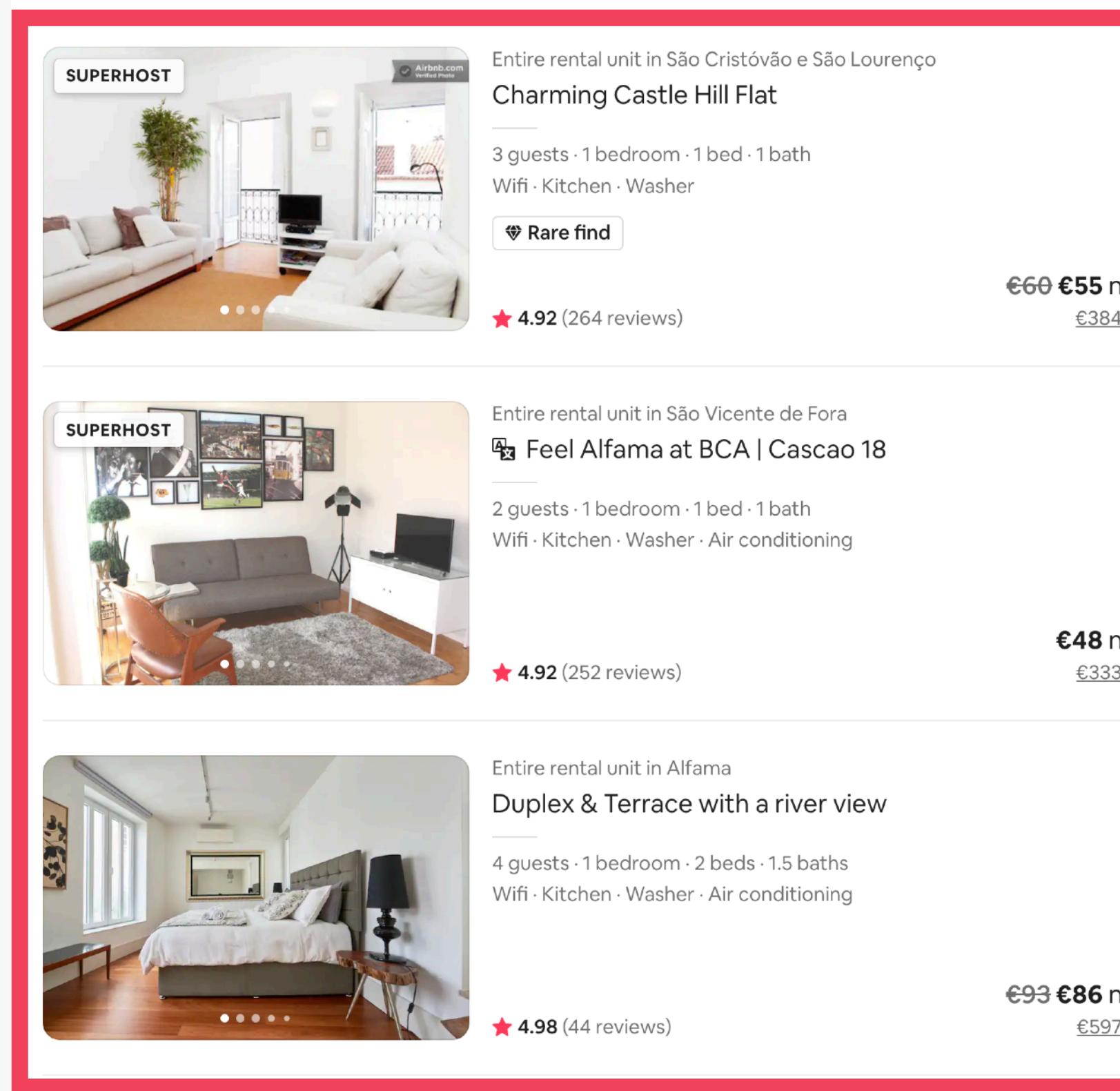
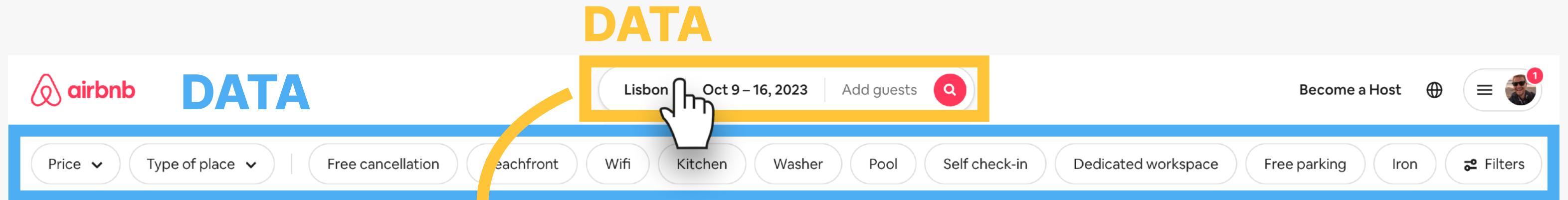


User interface needs to  
**stay in sync** with data



**Very hard problem** to solve!

# KEEPING UI IN SYNC WITH DATA



The image shows the Airbnb search interface and listing details page. A yellow arrow points from the search bar to the word 'DATA'. Another green arrow points from the search bar to the word 'DATA' on the map view.

👉 Keeping UI  
and data in sync  
would be virtually  
impossible with  
just vanilla  
JavaScript

Piece of data  
=  
Piece of state

```
const data = {  
  currency: {  
    currencyCountries: [],  
    currencySelectorExpanded: false,  
    selectedCurrencyCode: 'EUR',  
    loadingCurrencies: false,  
  },  
  footer: {  
    footerExpanded: false,  
  },  
  map: {  
    hoveredListingId: null,  
    hoveredDestinationPlaceId: null,  
    clickedListingId: null,  
    expandMapToItLocations: null,  
  },  
  saveToListModal: {  
    entity: null,  
    entityId: null,  
    entityType: null,  
    fetchListsError: false,  
    isCreatingList: false,  
    isFetchingLists: false,  
    isListsCacheValid: true,  
    entityMap: {  
      'listing-36109352': {},  
      'listing-34888453': {},  
      'listing-13357289': {},  
      'listing-29842619': {},  
      'listing-21693919': {},  
      'listing-44308963': {},  
      'listing-45085968': {},  
    },  
    lastError: null,  
    lastListSavedTo: null,  
    lastListSavedToOperation: null,  
    lists: {  
      id: 870054853,  
      name: 'Bons',  
      listing_ids: [  
        36109352, 34888453, 13357289, 29842619, 21693919, 44308963, 45085968,  
      ],  
      mt_template_ids: [],  
      place_activity_ids: [],  
      place_ids: [],  
      article_ids: [],  
      mt_scheduled_template_ids: [],  
      is_china_wishlist_home_collection: false,  
      settings_disabled: false,  
      airbnb_canonical_place_ids: [],  
      listing_id_str: [  
        '36109352',  
        '34888453',  
        '13357289',  
        '29842619',  
        '21693919',  
        '44308963',  
        '45085968',  
      ],  
      newListName: null,  
      newTask: {  
        actionText: '',  
        listRef: null,  
        message: null,  
      },  
      requiresSignup: false,  
      savingFrom: null,  
      visible: false,  
    },  
    ui: {  
      hideMap: true,  
      openedFilterId: null,  
      openedSearchInputField: null,  
      shouldLoadInterceptSurvey: false,  
      visiblePromos: {},  
    },  
    header: {  
      user: {  
        isLoggedIn: true,  
        profilePicUrl: 'https://a.muscache.com/im/pictures/user/9ed27941-a4f0-4c5e-b347-959a7f',  
        name: 'Jonas',  
        currency: 'EUR',  
        isHost: false,  
        guidebooksCount: 0,  
      },  
      dynamicColorTheme: null,  
      activeNavigationItem: null,  
      navigationItemsWithNotifications: 0,  
      flyoutMenuIsOpen: false,  
    },  
};
```

# SINGLE-PAGE APPLICATIONS WITH VANILLA JAVASCRIPT?

👉 *Front-end web applications are all about...*

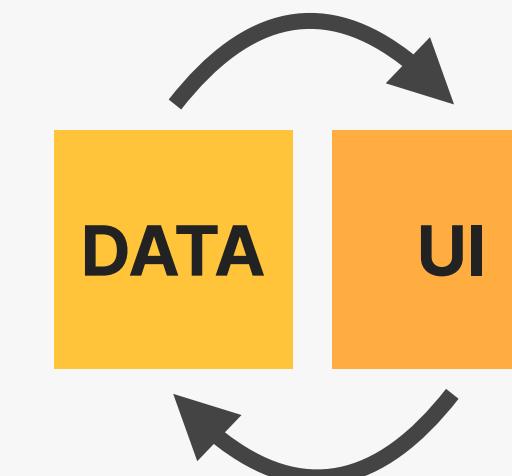
**Handling data + displaying  
data in a user interface**



User interface needs to  
**stay in sync** with data



**Very hard problem** to solve!



## PROBLEMS WITH



1

Requires lots of direct DOM manipulation and  
traversing (*imperative*) ➡ “Spaghetti code” 🍝

```
const guestsEl = document.querySelector('.guests');
const guestsPickerEl = document.querySelector('.picker');

guestsEl.addEventListener('click', function () {
    guestsEl.classList.toggle('inactive');
    guestsEl.classList.toggle('active');

    if (guestsPickerEl.style.display === 'block') {
        guestsPickerEl.style.display = 'none';
        guestsEl.firstElementChild.textContent = 'Add guests';
    } else {
        guestsPickerEl.style.display = 'block';
        guestsEl.firstElementChild.textContent = '';
    }
})
```

2

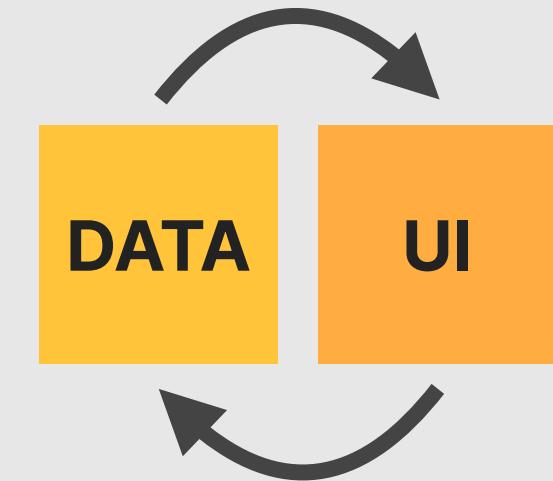
Data (state) is usually **stored in the DOM**, shared  
across entire app ➡ Hard to reason + bugs 🐛

# WHY DO FRONT-END FRAMEWORKS EXIST?

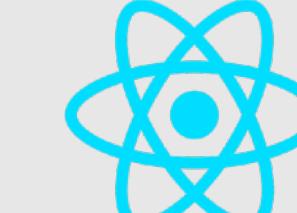
1

JavaScript front-end frameworks exist because...

KEEPING A USER INTERFACE IN SYNC WITH DATA  
IS REALLY HARD AND A LOT OF WORK



Front-end frameworks **solve this problem** and take hard work away from developers 🎉



←

Different approaches, same goal

2

They enforce a “**correct**” way of structuring and writing code (therefore contributing to solving the problem of “spaghetti code” 🍝)

3

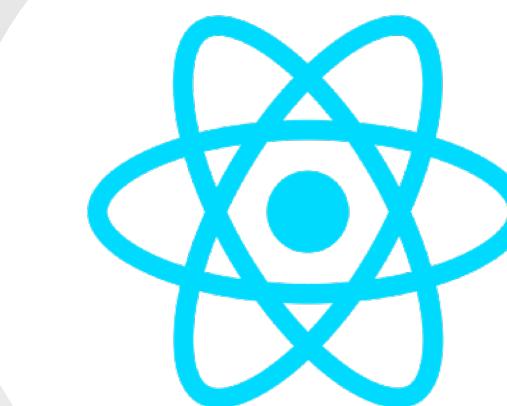
They give developers and teams a **consistent** way of building front-end applications



# WHAT IS REACT?

REACT

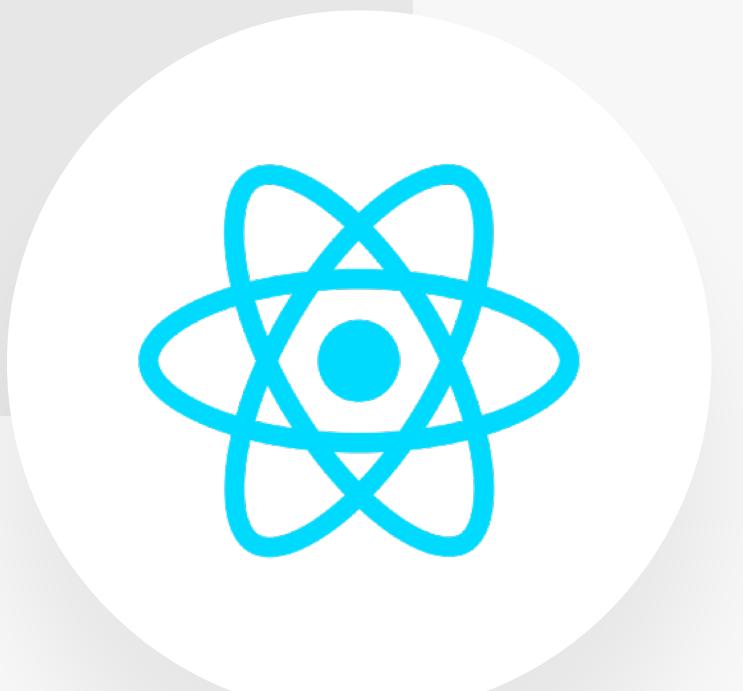
JAVASCRIPT LIBRARY FOR BUILDING  
USER INTERFACES



# WHAT IS REACT?

REACT

EXTREMELY POPULAR DECLARATIVE,  
COMPONENT-BASED STATE-DRIVEN JAVASCRIPT  
LIBRARY FOR BUILDING USER INTERFACES,  
CREATED BY FACEBOOK



# REACT IS BASED ON COMPONENTS

Based on components

Declarative

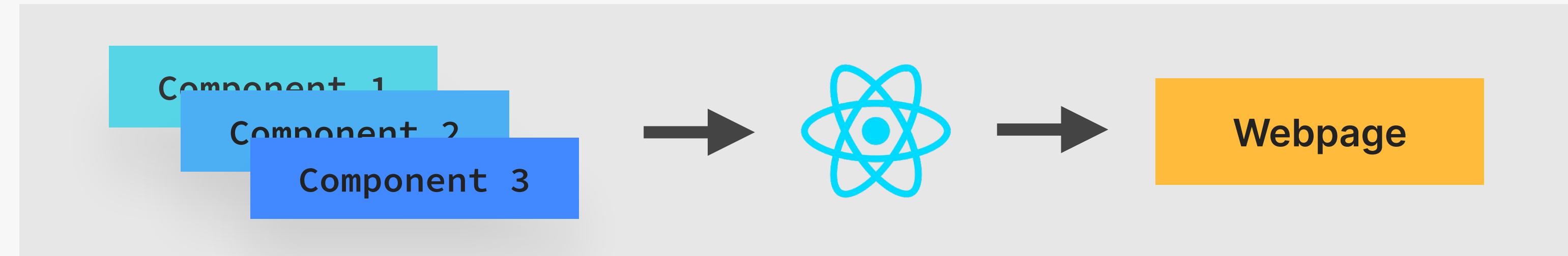
State-driven

JavaScript library

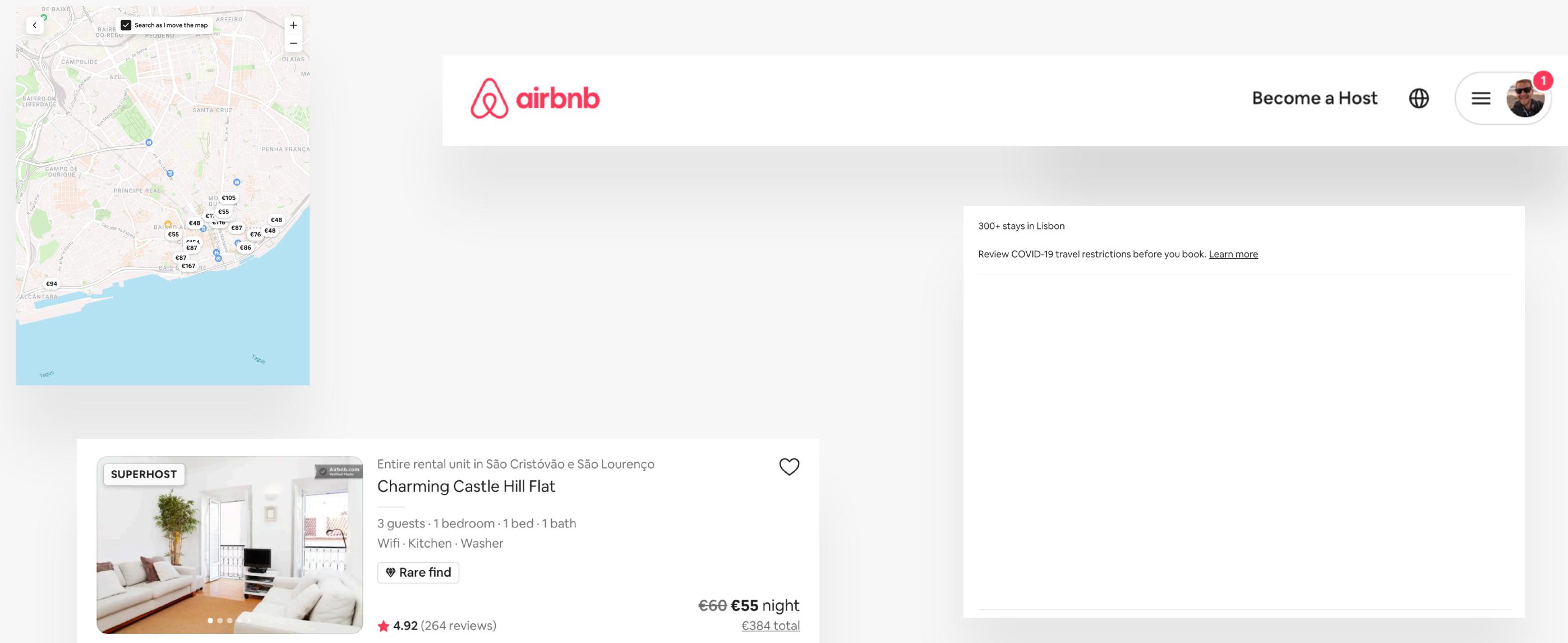
Extremely popular

Created by facebook

👉 Components are the **building blocks** of user interfaces in React



👉 We build complex UIs by **building and combining multiple components**



# REACT IS BASED ON COMPONENTS

Based on components

Declarative

State-driven

JavaScript library

Extremely popular

Created by facebook

The screenshot shows the Airbnb search interface. At the top, there is a red header bar containing the Airbnb logo, a 'NavBar' section, search filters (Price, Type of place, etc.), and a search bar. Below the header, a purple box labeled 'Results' contains three listing cards, each with a photo, title, details, and price. To the right of the results is a green box labeled 'Map' showing a map of Lisbon with price markers. The entire interface is built using React components.

**NavBar**

Lisbon | Oct 9–16, 2023 | Add guests

Search

Become a Host

Price ▾ Type of place ▾ | Free cancellation Beachfront Wifi Kitchen Washer Pool Self check-in Dedicated workspace Free parking Iron Filters

300+ stays in Lisbon

Review COVID-19 travel restrictions before you book. [Learn more](#)

**Results**

**Listing**

**Listing**

**Listing**

Map

Tagus

# REACT IS DECLARATIVE

Based on components

Declarative

State-driven

JavaScript library

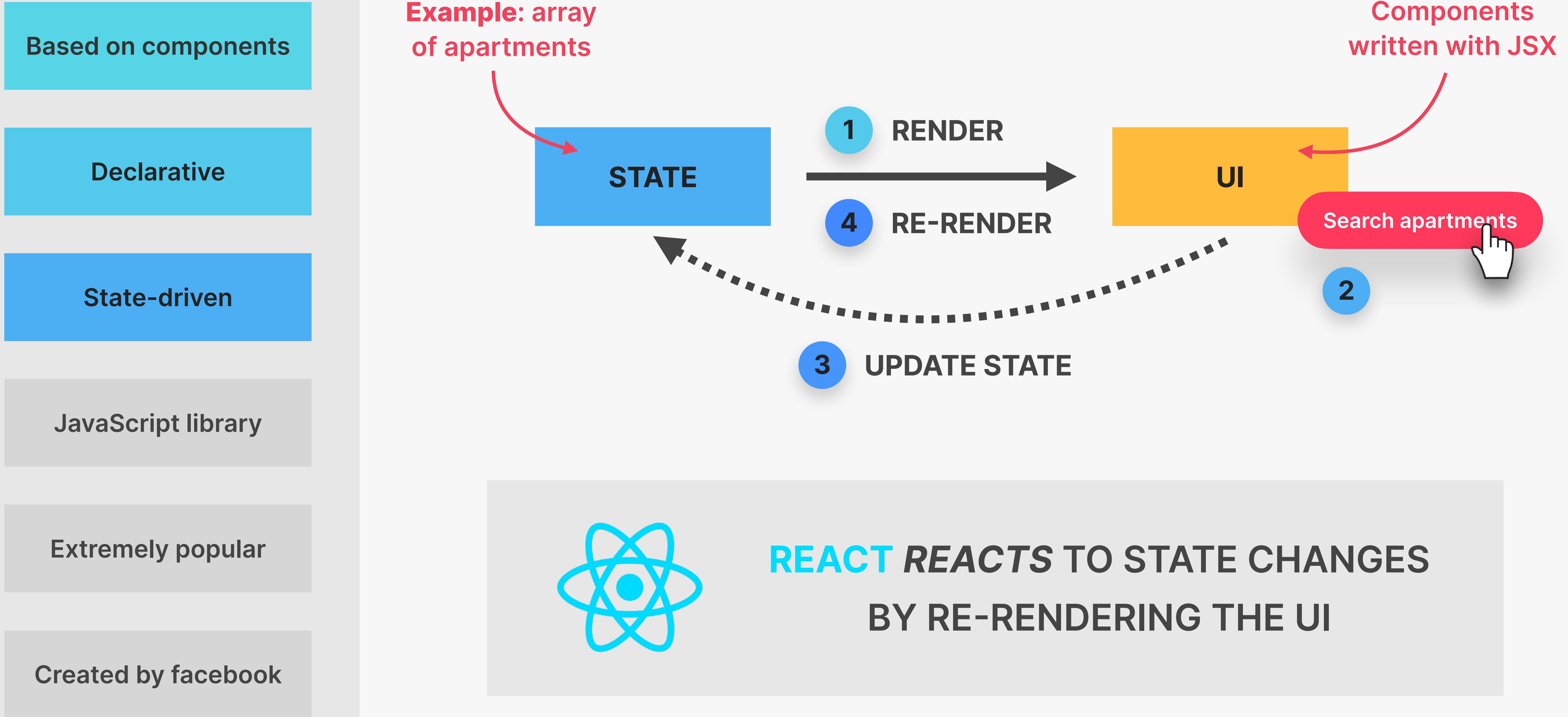
Extremely popular

Created by facebook

- 👉 We describe how components look like and how they work using a **declarative syntax called JSX**
- 👉 **Declarative:** telling React what a component should look like, **based on current data/state**
- 👉 React is **abstraction** away from DOM: we **never touch the DOM**
- 👉 JSX: a syntax that **combines** **HTML** **CSS** **JavaScript** as well as referencing **other components**

```
JSX returned from a component
return (
  <main>
    <NavBar>
      <h1 style={{ fontSize: '3.2rem' }}>AirBnB</h1>
      <Search />
      <a href="#">Become a host</a>
    </NavBar>
    <Results>
      <p style={{ fontSize: '1.6rem', margin: '1.2rem' }}>
        {numListings} stays in Lisbon
      </p>
      <Listing listing={listings[0]} />
      <Listing listing={listings[1]} />
      <Listing listing={listings[2]} />
    </Results>
    <Map listings={listings} onClick={moveMap} />
  </main>
);
```

# REACT IS STATE-DRIVEN



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

# REACT IS A JAVASCRIPT LIBRARY

Based on components

Declarative

State-driven

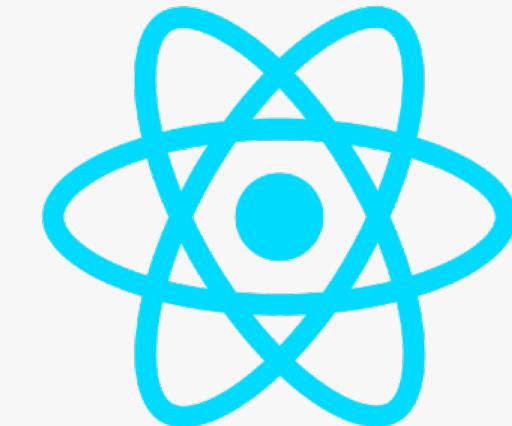
JavaScript library

Extremely popular

Created by facebook



Is React a **library** or a framework?



Because React is only the “view” layer. We need to pick multiple external libraries to build a complete application

**NEXT.js      Remix**

Complete frameworks built on top of React

# REACT IS EXTREMELY POPULAR

Based on components

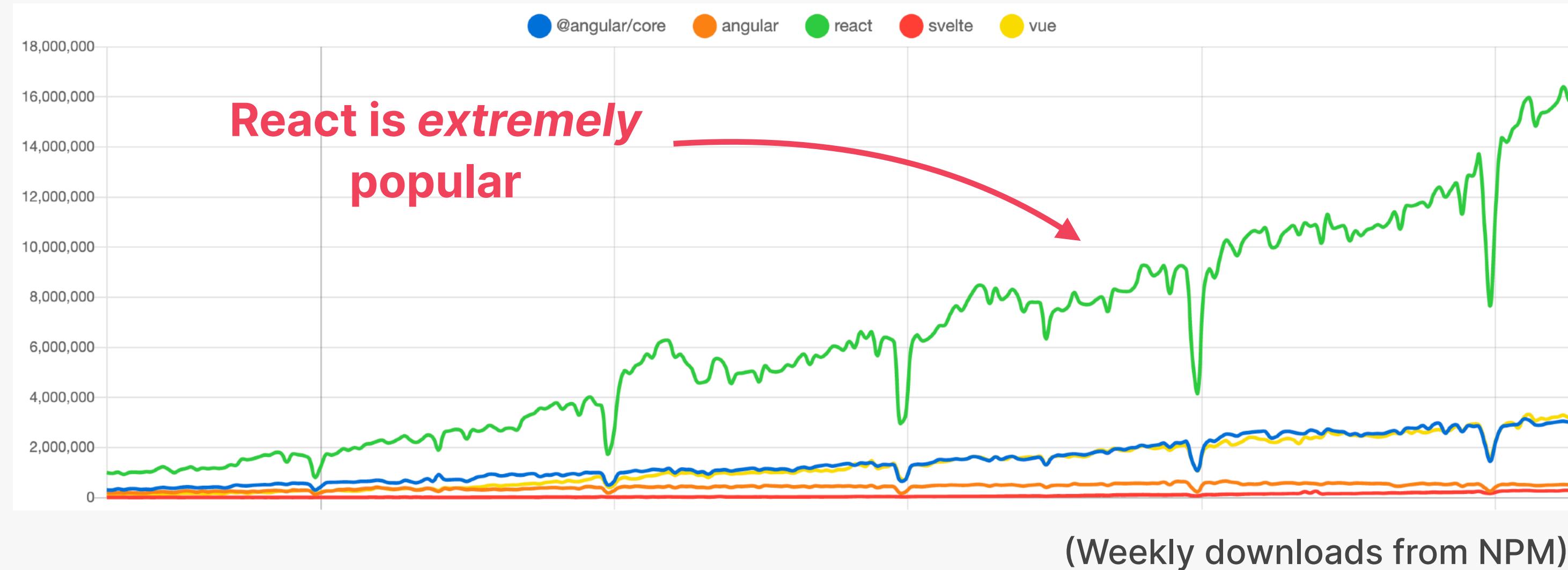
Declarative

State-driven

JavaScript library

Extremely popular

Created by facebook



✓ Many large companies have adopted React



✓ Huge job market with high demand for React developers 💰

✓ Large and vibrant React developer community

✓ Gigantic third-party library ecosystem

# REACT WAS CREATED BY FACEBOOK

Based on components

Declarative

State-driven

JavaScript library

Extremely popular

Created by facebook



- 👉 React was created in **2011** by Jordan Walke, an engineer working at Facebook at the time
- 👉 React was open-sourced in **2013**, and has since then completely transformed front-end web development

facebook

Meta



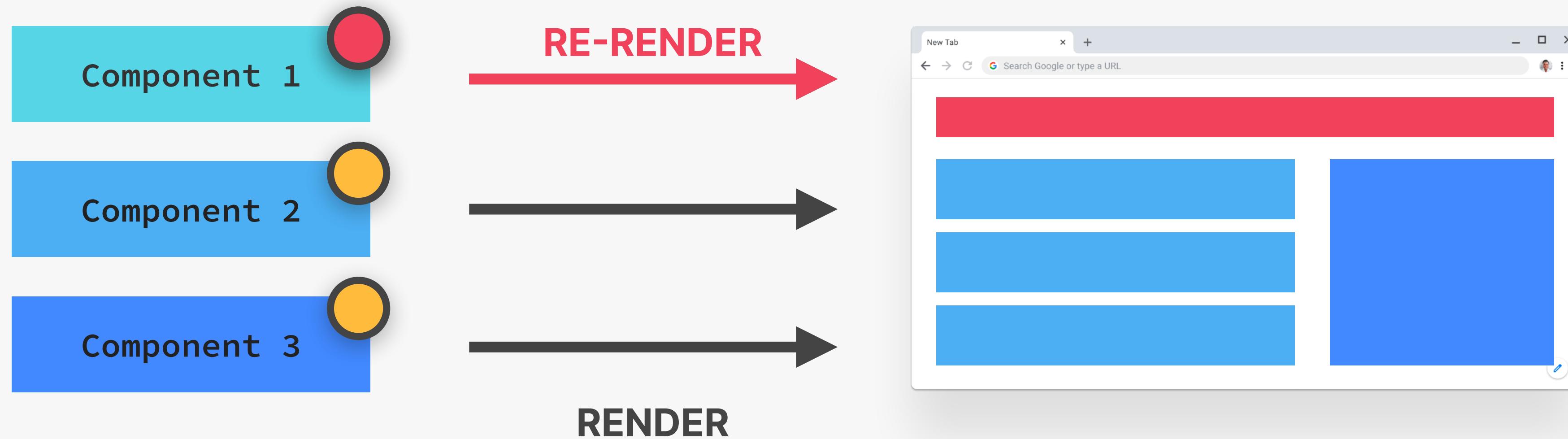
# SUMMARY

1

Rendering components on a webpage (UI) based on their current state

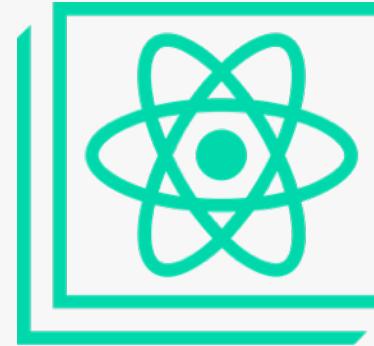
2

Keeping the UI in sync with state, by re-rendering (*reacting*) when state changes

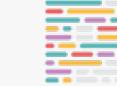




# THE TWO OPTIONS FOR SETTING UP A REACT PROJECT



## CREATE-REACT-APP

- 👉 Complete “starter kit” for React applications
- 👉 Everything is already configured: ESLint, Prettier, Jest, etc.  ESLint  Prettier  Jest
- 👉 Uses slow and outdated technologies (i.e. webpack)



## VITE

- 👉 Modern build tool that contains a template for setting up React applications
- 👉 Need to manually set up ESLint (and others)
- 👉 Extremely fast hot module replacement (HMR) and bundling



Most of the course

- ✓ Use for tutorials or experiments
- ✗ Don't use for a real-world app



By the end of the course

- ✓ Use for modern real-world apps

# WHAT ABOUT REACT FRAMEWORKS?

NEXT.js

Remix

- 👉 The React team now advises to use a “**React Framework**” for new projects
- 👉 Many people think that this is not the best idea: “**vanilla**” React apps are important too!
- 👉 This only makes sense for building actual products, **not for learning React**
- 👉 Of course, you still need to **learn React itself**

✌️ Don’t worry about this recommendation for now. Let’s just learn React!

LEARN REACT > INSTALLATION >

👉 react.dev

## Start a New React Project

If you want to build a new app or a new website fully with React, we recommend picking one of the React-powered frameworks popular in the community. Frameworks provide features that most apps and sites eventually need, including routing, data fetching, and generating HTML.

### Production-grade React frameworks

#### Next.js

**Next.js** is a full-stack React framework. It’s versatile and lets you create React apps of any size—from a mostly static blog to a complex dynamic application. To create a new Next.js project, run in your terminal:

Terminal

```
npx create-next-app
```

Copy

If you’re new to Next.js, check out the [Next.js tutorial](#).

Next.js is maintained by [Vercel](#). You can [deploy a Next.js app](#) to any Node.js or serverless hosting, or to your own server. [Fully static Next.js apps](#) can be deployed to any static hosting.

#### Remix

**Remix** is a full-stack React framework with nested routing. It lets you break your app into nested parts that can load data in parallel and refresh in response to the user actions. To create a new Remix project, run:



# WORKING WITH COMPONENTS, PROPS, AND JSX

# COMPONENTS AS BUILDING BLOCKS

## COMPONENTS

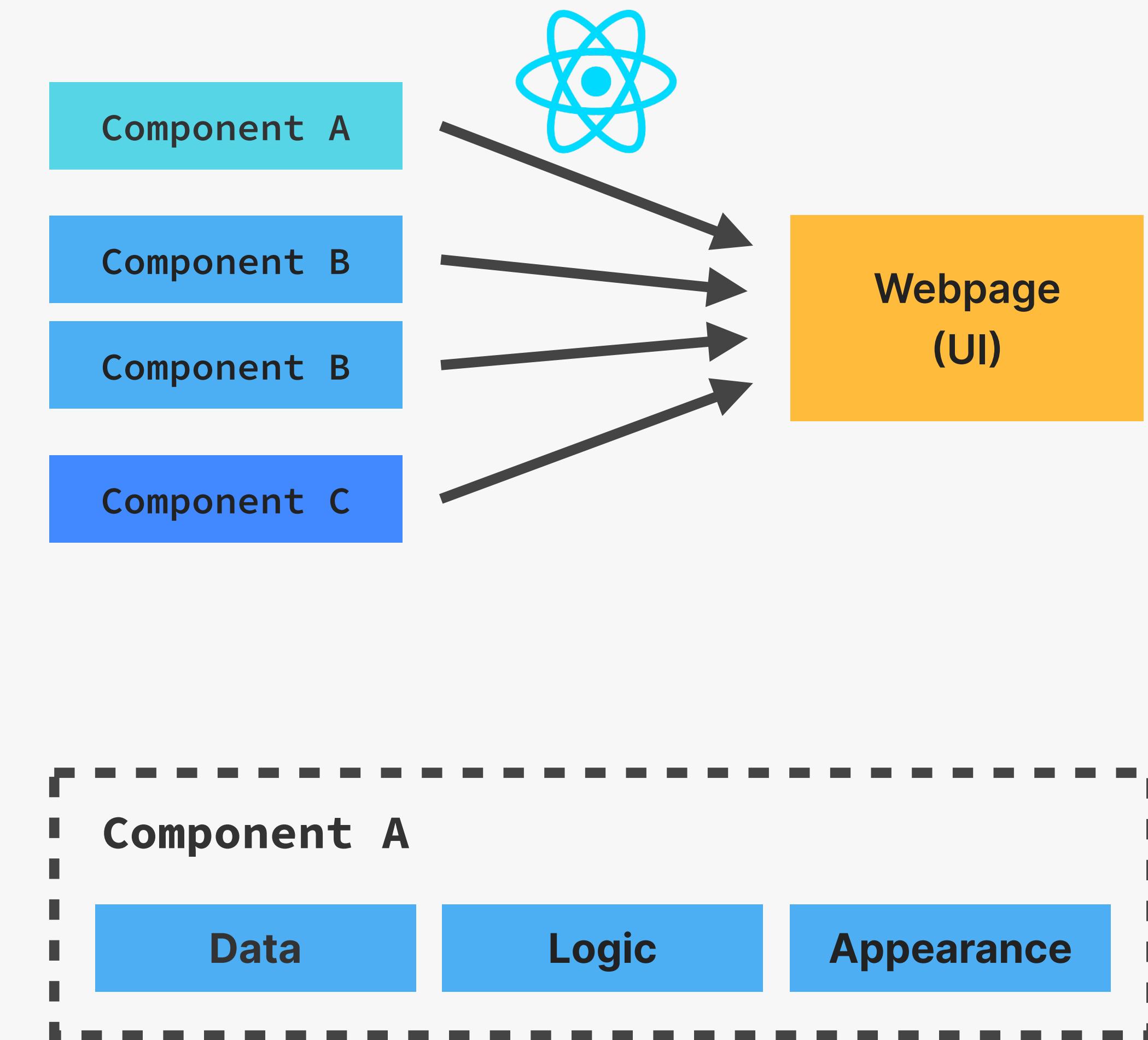
- 👉 React applications are entirely made out of components
- 👉 Building blocks of user interfaces in React

The screenshot shows a portion of the Udemy website. At the top right, there's a search bar with the placeholder "Search for anything". Below it is a purple "Add to cart" button. To the left of the search bar is a "Filter questions" dropdown menu with three options: "Questions I'm following", "Questions I asked", and "Questions without responses", each with an unchecked checkbox. On the far right, there are icons for Udemy Business, Instructor, My learning, a heart, a shopping cart, a bell, and a user profile. The main content area displays a course card for "The Complete JavaScript Course 2022: From Zero to Expert!" by Jonas Schmedtmann. The card features a thumbnail of a laptop screen showing code and a yellow "JS" logo, the course title, the instructor's name, a 4.7-star rating with 136,226 reviews, the price of €19.99, and a "Bestseller" badge. To the right of the course card is a sidebar titled "Topic" with a list of categories: Web Development (694), PHP (665), React (576), and CSS (493). A "Show more" link is also present.

# 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 AS BUILDING BLOCKS

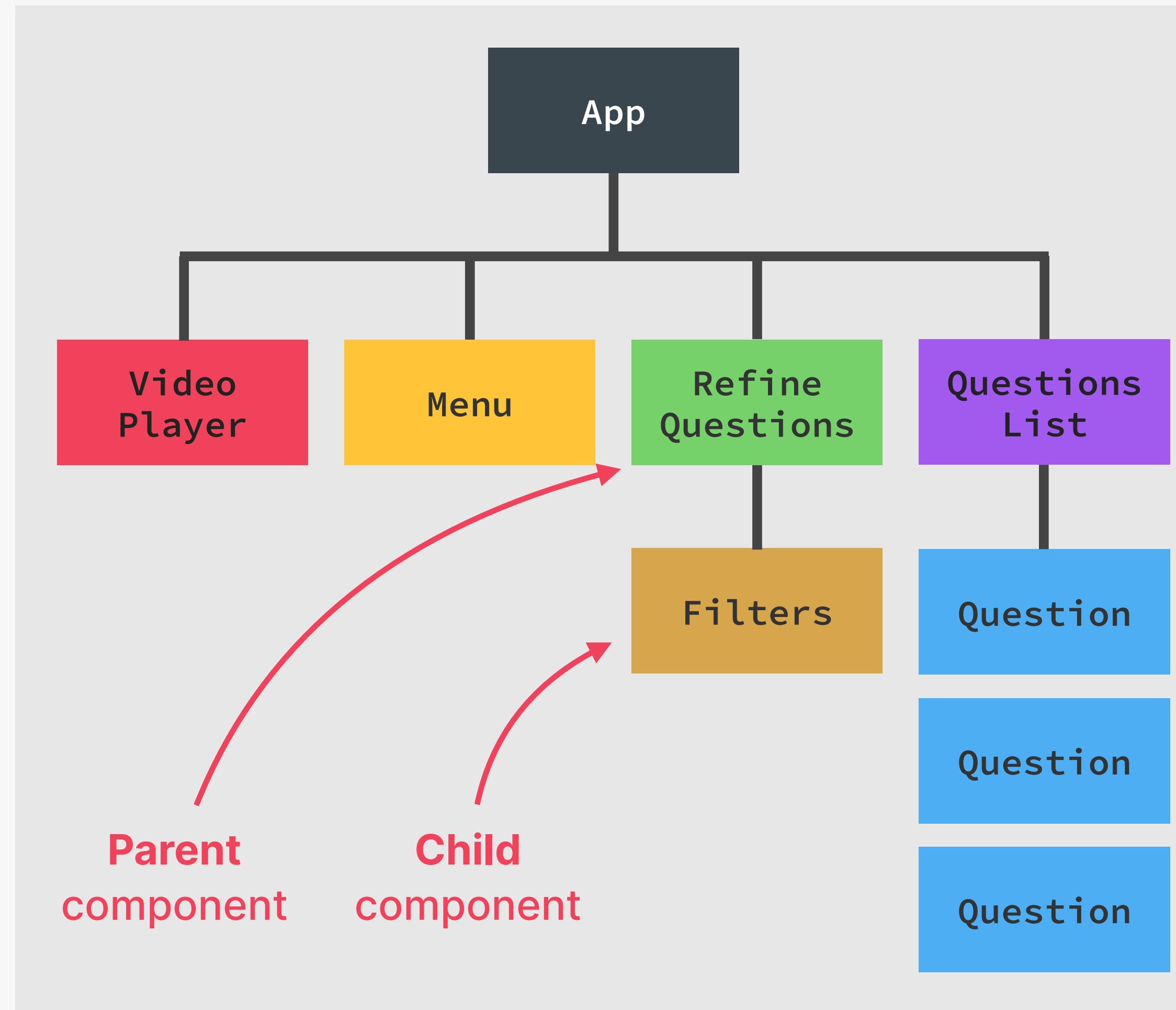
## COMPONENTS

- 👉 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

The screenshot shows a course interface with the following components:

- VideoPlayer**: A large video player at the top right.
- Menu**: A yellow navigation bar with links: Course content, Overview, Q&A (underlined), Notes, Announcements, and a search icon.
- RefineQuestions**: A green search bar labeled "Search all course questions" with a magnifying glass icon.
- Filters**: A section below the search bar with dropdown menus: All lectures, Sort by most recent, and Filter questions.
- QuestionList**: A purple section titled "All questions in this course (41683)". It lists three questions:
  - Question 2 on Challenge 2** by Darryl (Lecture 115, 13 hours ago). It has 0 upvotes and 0 comments.
  - Elegant alternative for loading markers from localStorage** by Vincent Giovanni (Lecture 242, 14 hours ago). It has 0 upvotes and 0 comments.
  - How to not violate the "Do not repeat yourself" principle** by Marinela (Lecture 45, 15 hours ago). It has 0 upvotes and 1 comment.

# COMPONENT TREES



**VideoPlayer**

Course content Overview Q&A Notes Announcements **Menu**

**RefineQuestions**

Search all course questions

All lectures ▾ Sort by most recent ▾ Filter questions ▾ **Filters**

All questions in this course (41683)

**Question**

**Question**

**Question**

**QuestionList**



# 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 returned from component

# 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**
- 👉 Each JSX element is **converted** to a `React.createElement` function call
- 👉 We could use React **without** JSX

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

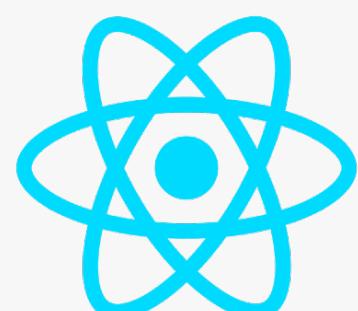


```
React.createElement(
  'header',
  null,
  React.createElement(
    'h1',
    { style: { color: 'red' } },
    'Hello React!'
)
);
```



Hello React!

BABEL



# JSX IS DECLARATIVE

IMPERATIVE

*"How to do things"*

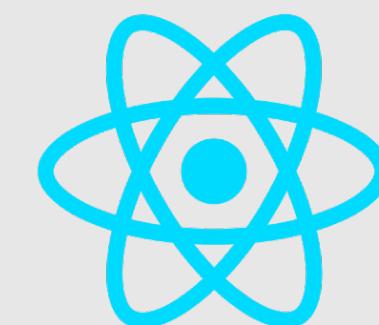


- 👉 Manual DOM element selections and DOM traversing
- 👉 Step-by-step DOM mutations until we reach the desired UI

```
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"*



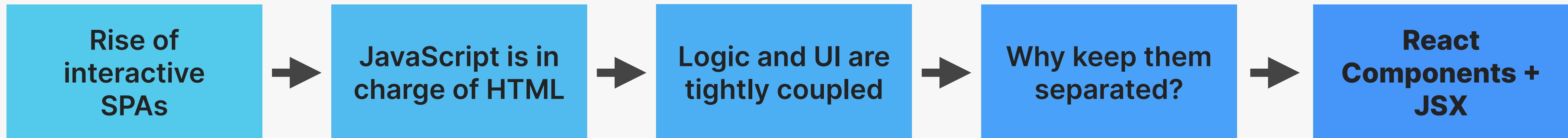
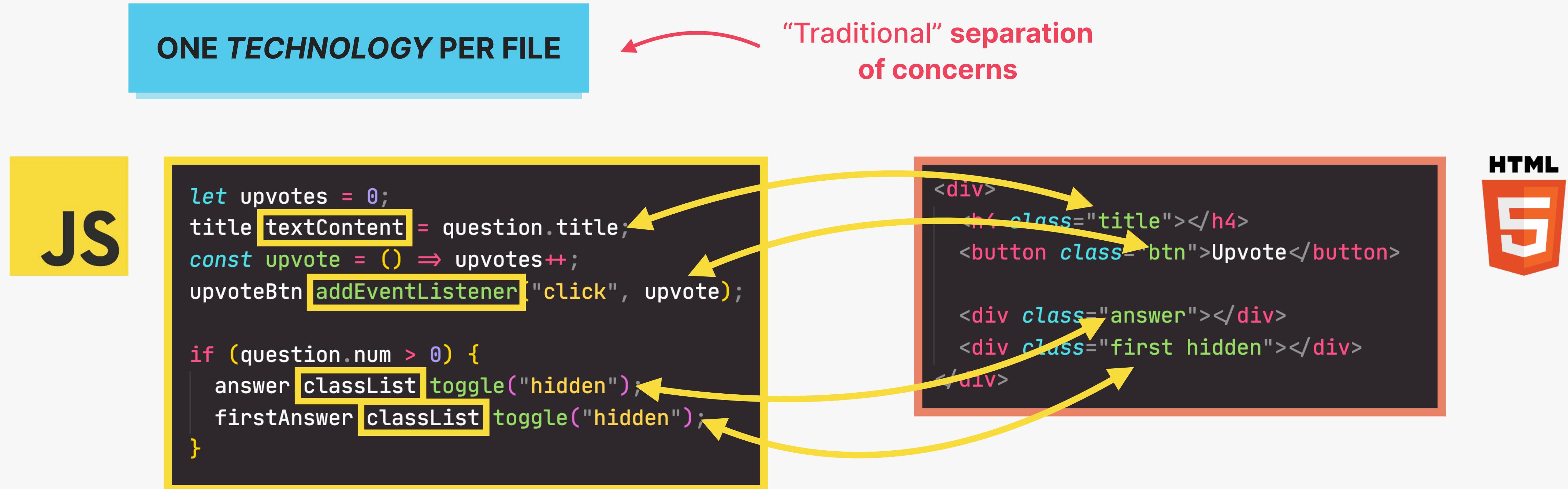
- 👉 Describe what UI should look like using JSX, based on current data
- 👉 React is an abstraction away from DOM: we never touch the DOM
- 👉 Instead, we think of the UI as a reflection of the current data

```
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>
  );
}
```

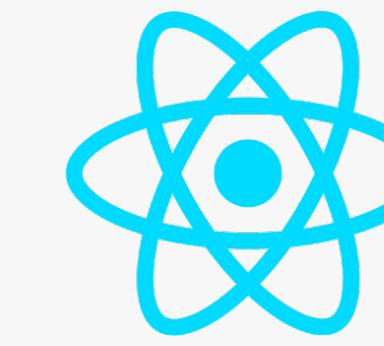
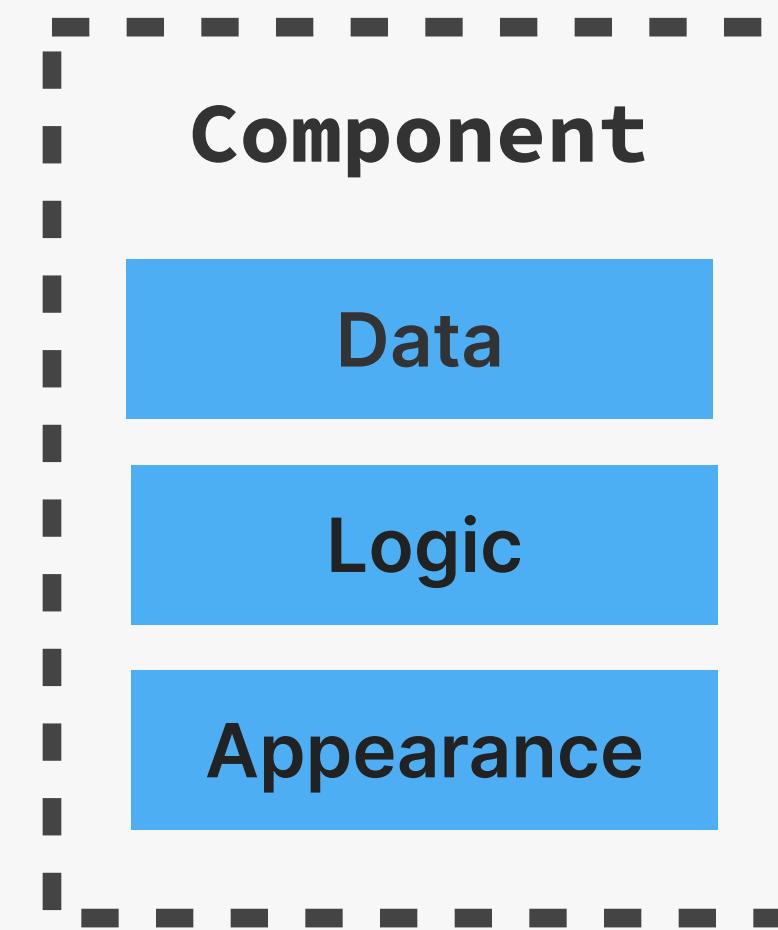


# SEPARATION OF CONCERNS?



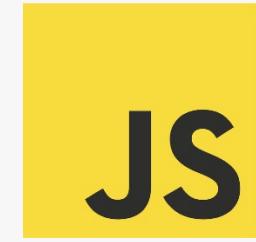
# SEPARATION OF CONCERNS?

ONE COMPONENT PER FILE

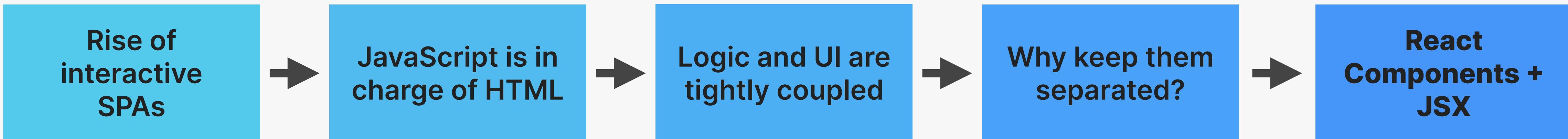


HTML and JS  
are *colocated*

```
function Question({ question }) {  
  const [upvotes, setUpvotes] = useState(0);  
  const upvote = () => setUpvotes((v) => v + 1);  
  
  return (  
    <div>  
      <h4>{question.title}</h4>  
      <UpvoteBtn onClick={upvote} />  
      {question.num > 0 ? (  
        <Answers numAnswers={question.num}></Answers>  
      ) : (  
        <FirstAnswer />  
      )}  
    </div>  
  );  
}
```



Fundamental reason for components



# SEPARATION OF CONCERNS!

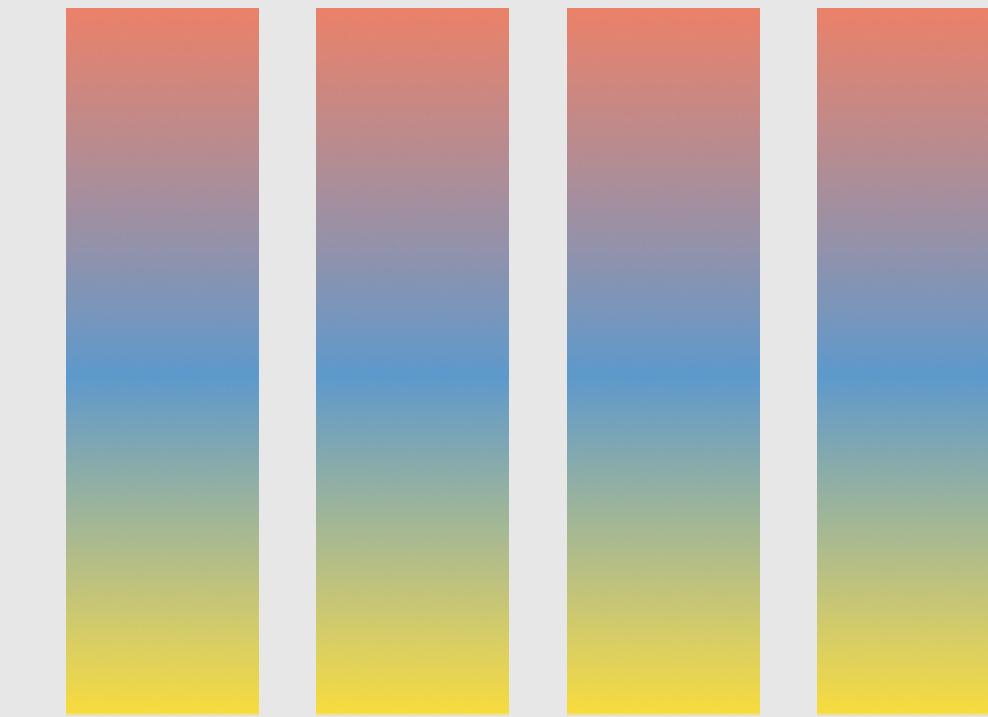
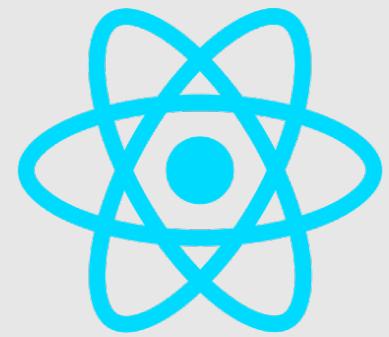
ONE TECHNOLOGY PER FILE



“Traditional”  
separation of  
concerns

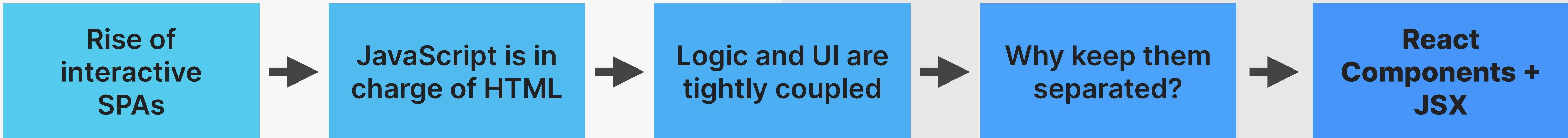
COMpletely  
NEW PARADIGM

ONE COMPONENT PER FILE



Each component  
is concerned  
with one piece  
of the UI

Question  
Menu  
Filters  
Player

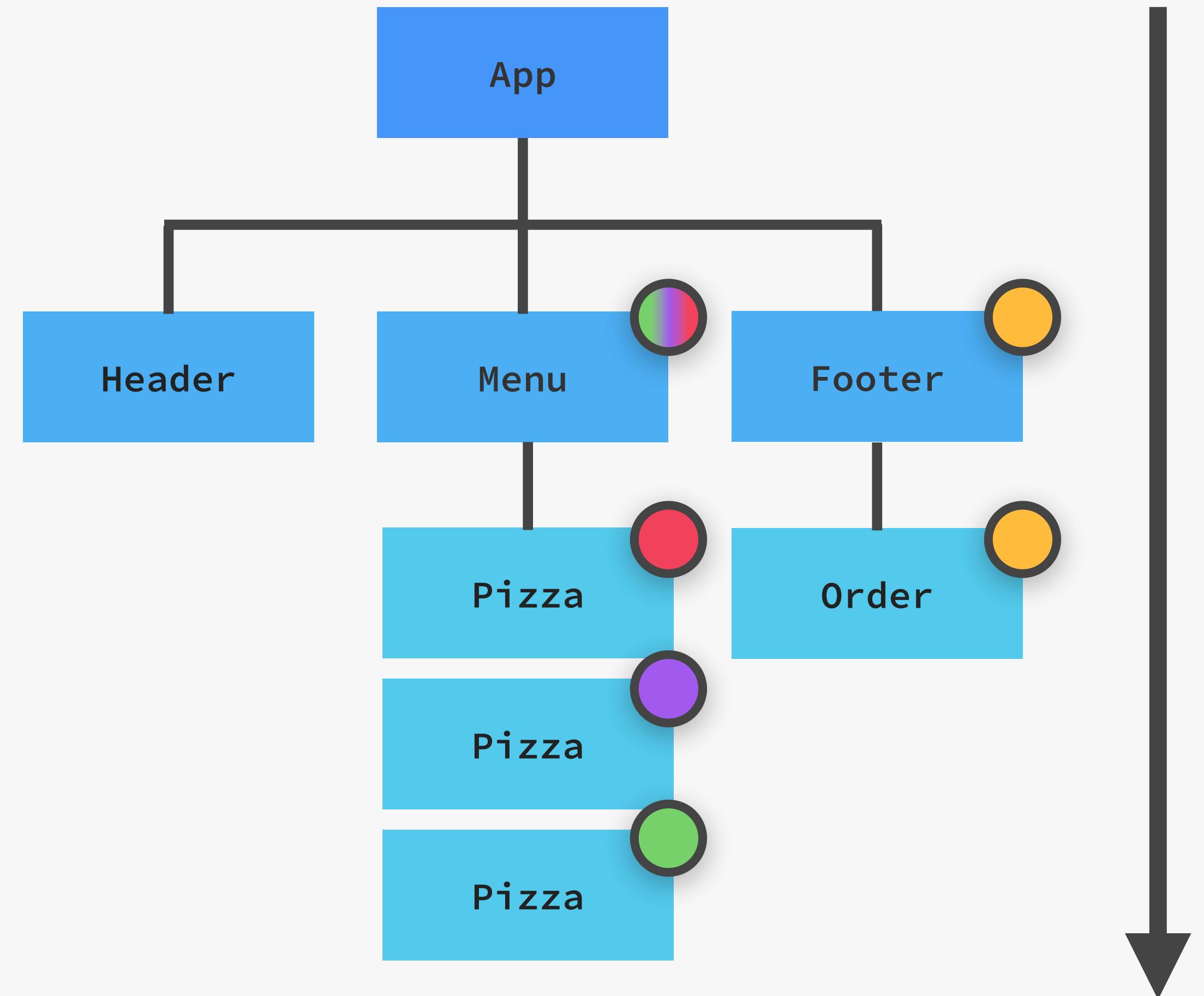




# REVIEWING PROPS

## PROPS

- 👉 Props are used to pass data from **parent components** to **child components** (down the component tree)

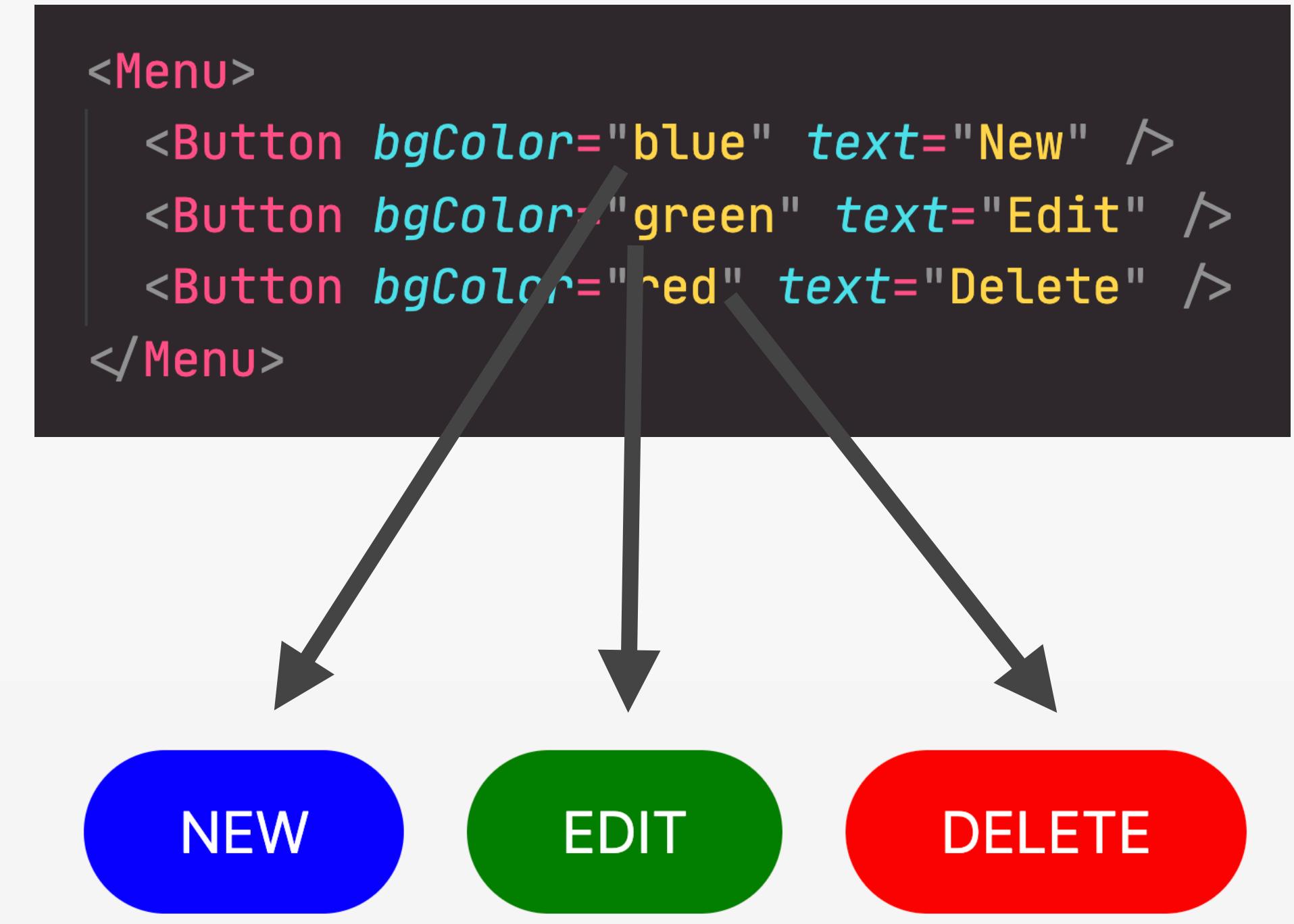


# REVIEWING PROPS

## PROPS

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

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



# REVIEWING PROPS

## 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  
  }  
}
```

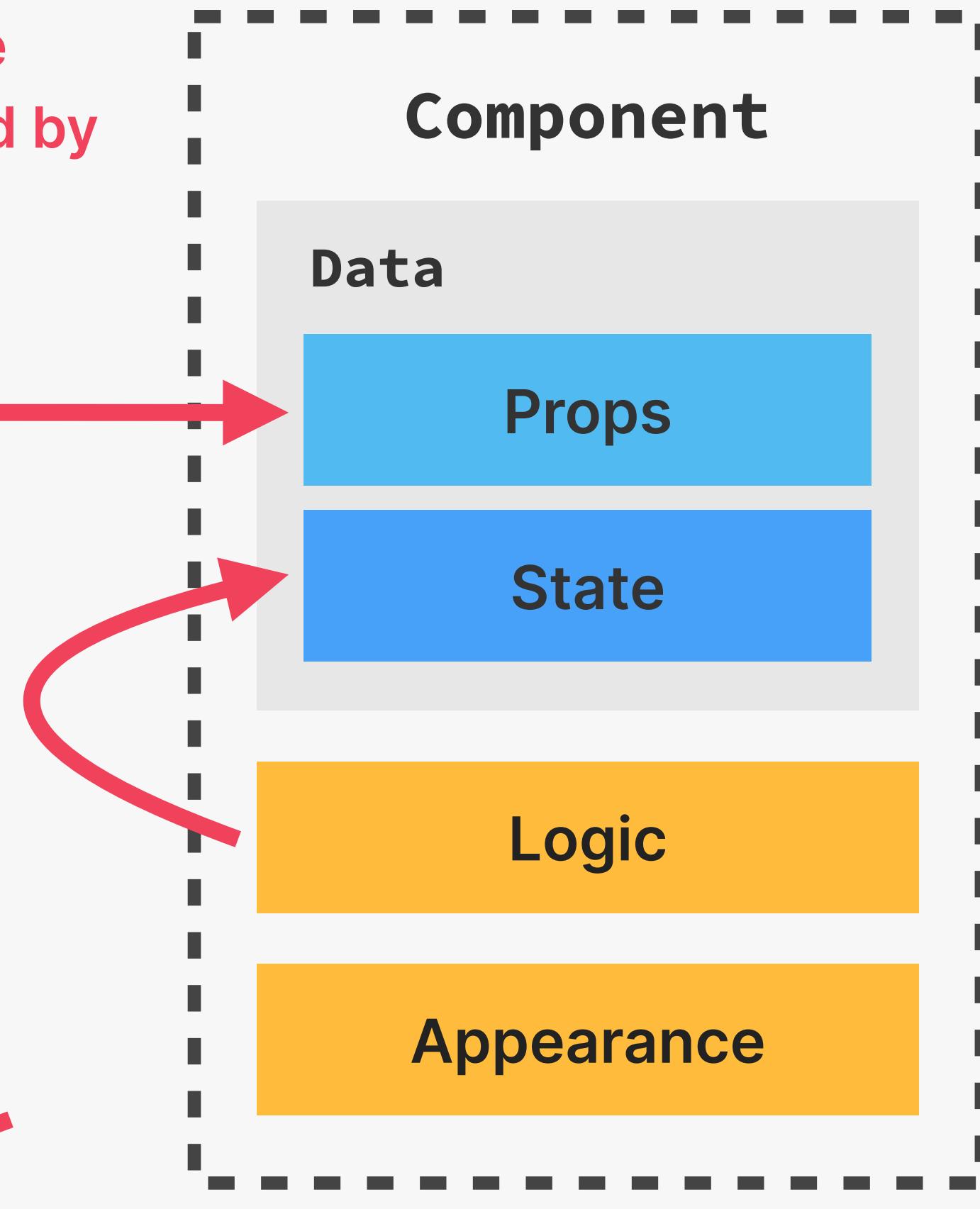
# PROPS ARE READ-ONLY!

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>  
}
```



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**

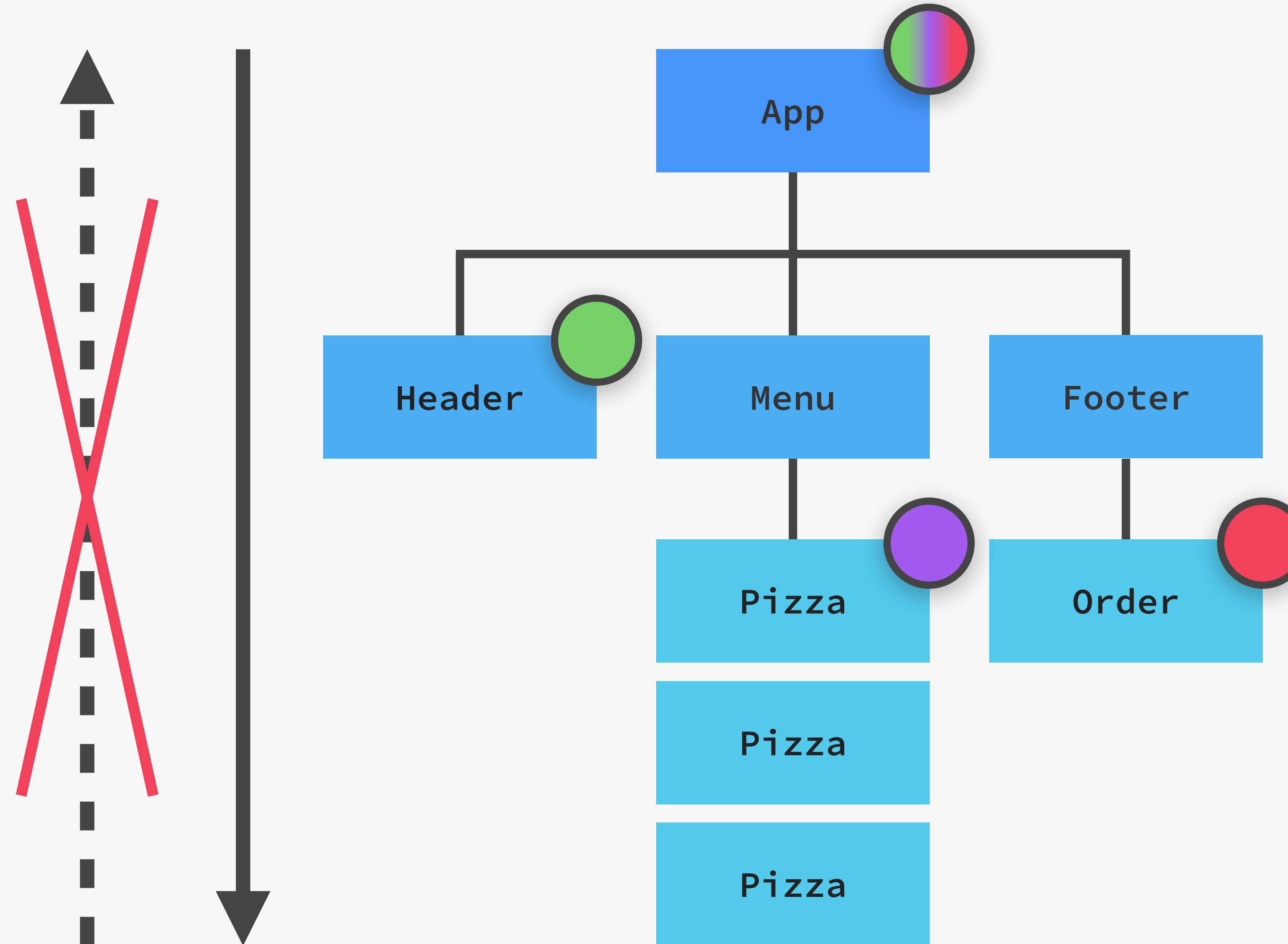
↓ WHY?

👉 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



## ONE-WAY DATA FLOW...

- 👍 ... makes applications more predictable and easier to understand
- 👍 ... makes applications easier to debug, as we have more control over the data
- 👍 ... is more performant



*Angular has two-way data flow*



# RULES OF JSX

## GENERAL JSX RULES

- 👉 JSX works essentially like HTML, but we can enter “**JavaScript mode**” by using {} (for text or attributes)
  - 👉 We can place **JavaScript expressions** inside {}.  
Examples: reference variables, create arrays or objects, [] .map(), ternary operator
  - 👉 Statements are **not allowed** (if/else, for, switch)
  - 👉 JSX produces a **JavaScript expression**
- ==  `const el = <h1>Hello React!</h1>;`  
`const el = React.createElement("h1", null, "Hello React!");`
- 1 We can place **other pieces of JSX** inside {}
  - 2 We can write JSX **anywhere** inside a component (in if/else, assign to variables, pass it into functions)
  - 👉 A piece of JSX can only have **one root element**. If you need more, use <React.Fragment> (or the short <>)

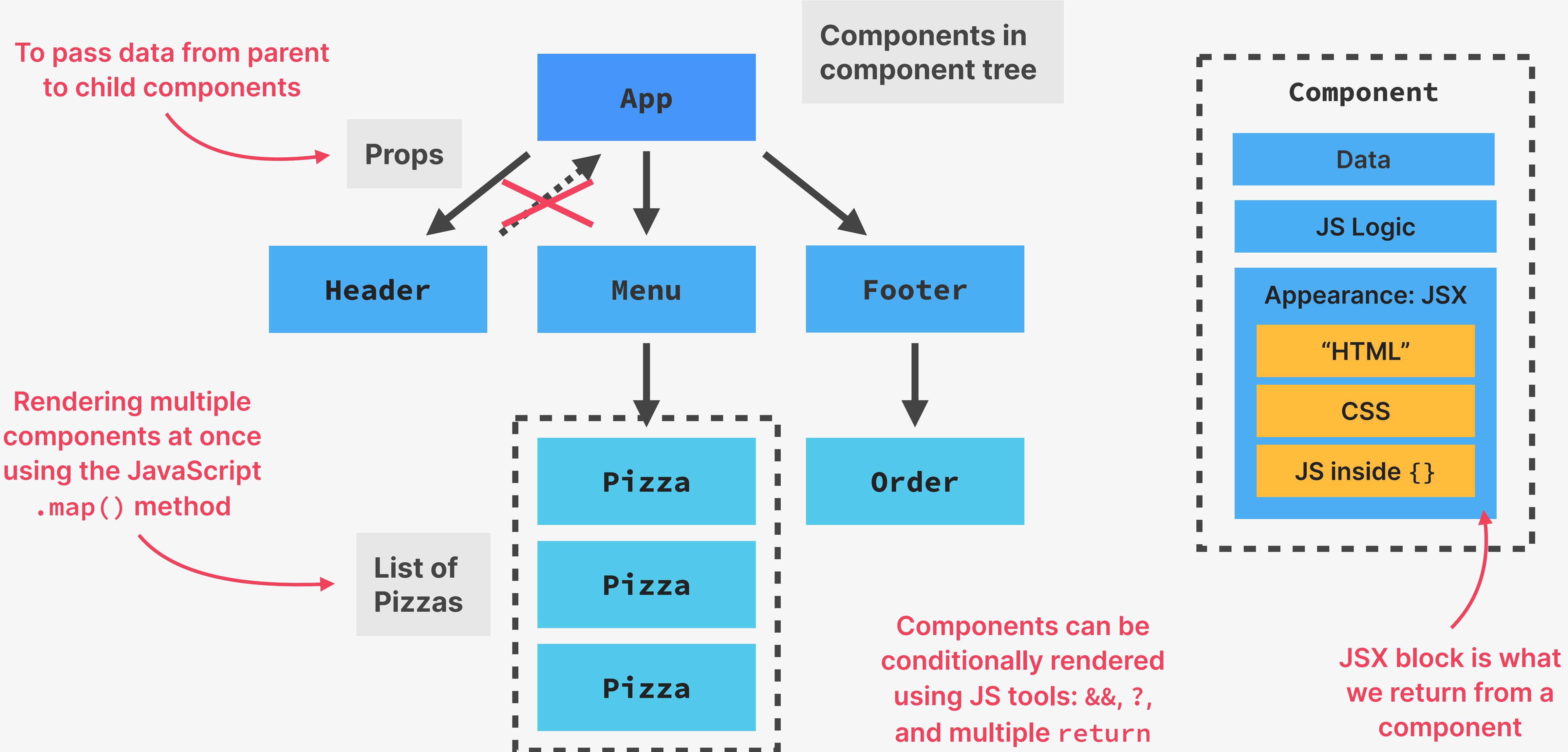
## DIFFERENCES BETWEEN JSX AND HTML

- 👉 `className` instead of HTML's `class`
- 👉 `htmlFor` instead of HTML's `for`
- 👉 Every tag needs to be **closed**. Examples: `<img />` or `<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)





# SECTION SUMMARY

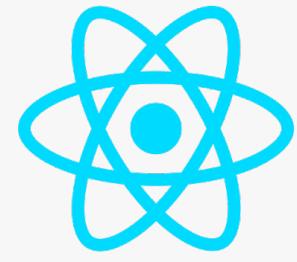




# STATE, EVENTS, AND FORMS: INTERACTIVE COMPONENTS



# WHAT WE NEED TO LEARN



## WHAT REACT DEVELOPERS NEED TO LEARN ABOUT STATE:

1

**What is state and why do we need it?**

This section

2

**How to use state in practice?**

- 👉 useState
- 👉 useReducer
- 👉 Context API

3

**Thinking about state**

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

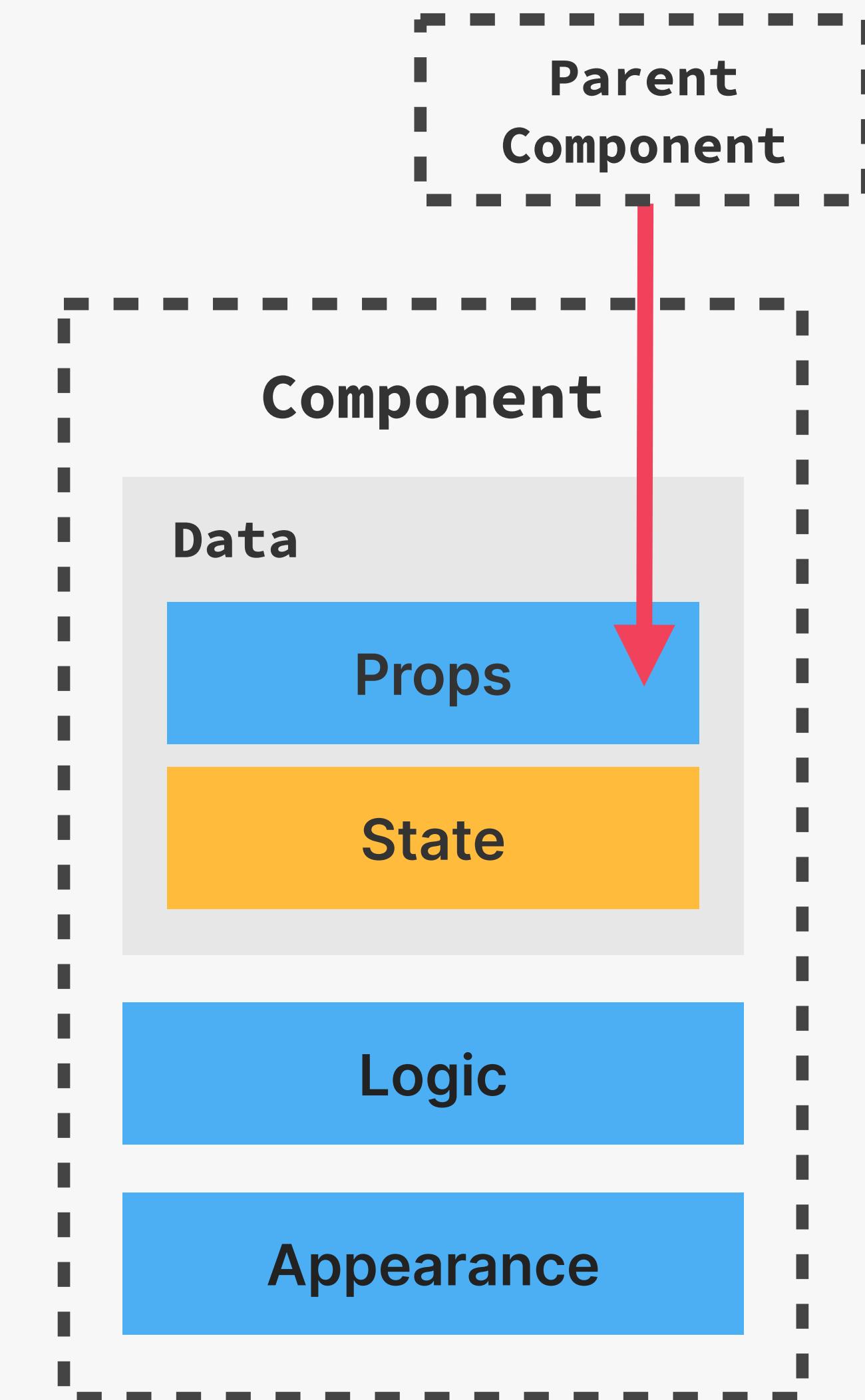
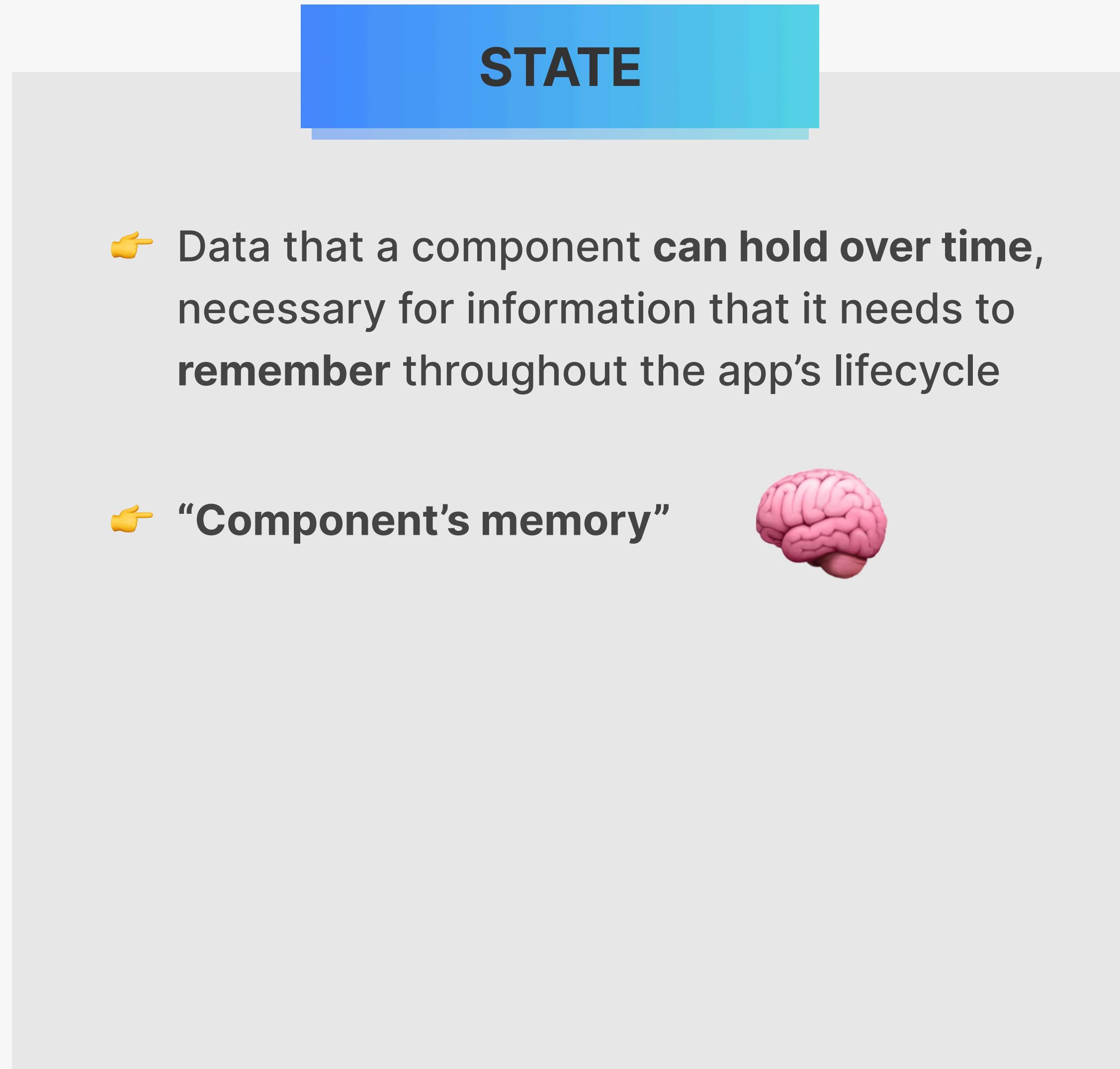
Rest of the course...



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

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

# WHAT IS STATE?



# 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

Notifications

Messages



javascr

Overview

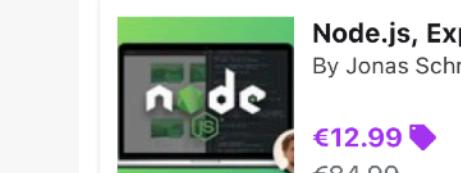
Q&A

Notes

Announcements

## Shopping Cart

2 Courses in Cart



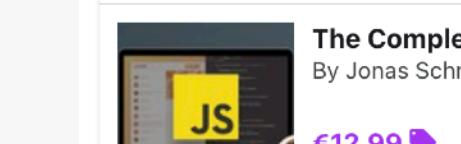
Node.js, Express, MongoDB & More: The Complete Bootcamp 2022  
By Jonas Schmedtmann, Web Developer, Designer, and Teacher

€12.99 ⚡  
€84.99

Updated Recently

4.7 ★★★★★ (11465 ratings)  
42 ore totali • 229 lectures • All Levels

Remove Save for Later



The Complete JavaScript Course 2022: From Zero to Expert!  
By Jonas Schmedtmann, Web Developer, Designer, and Teacher

€12.99 ⚡

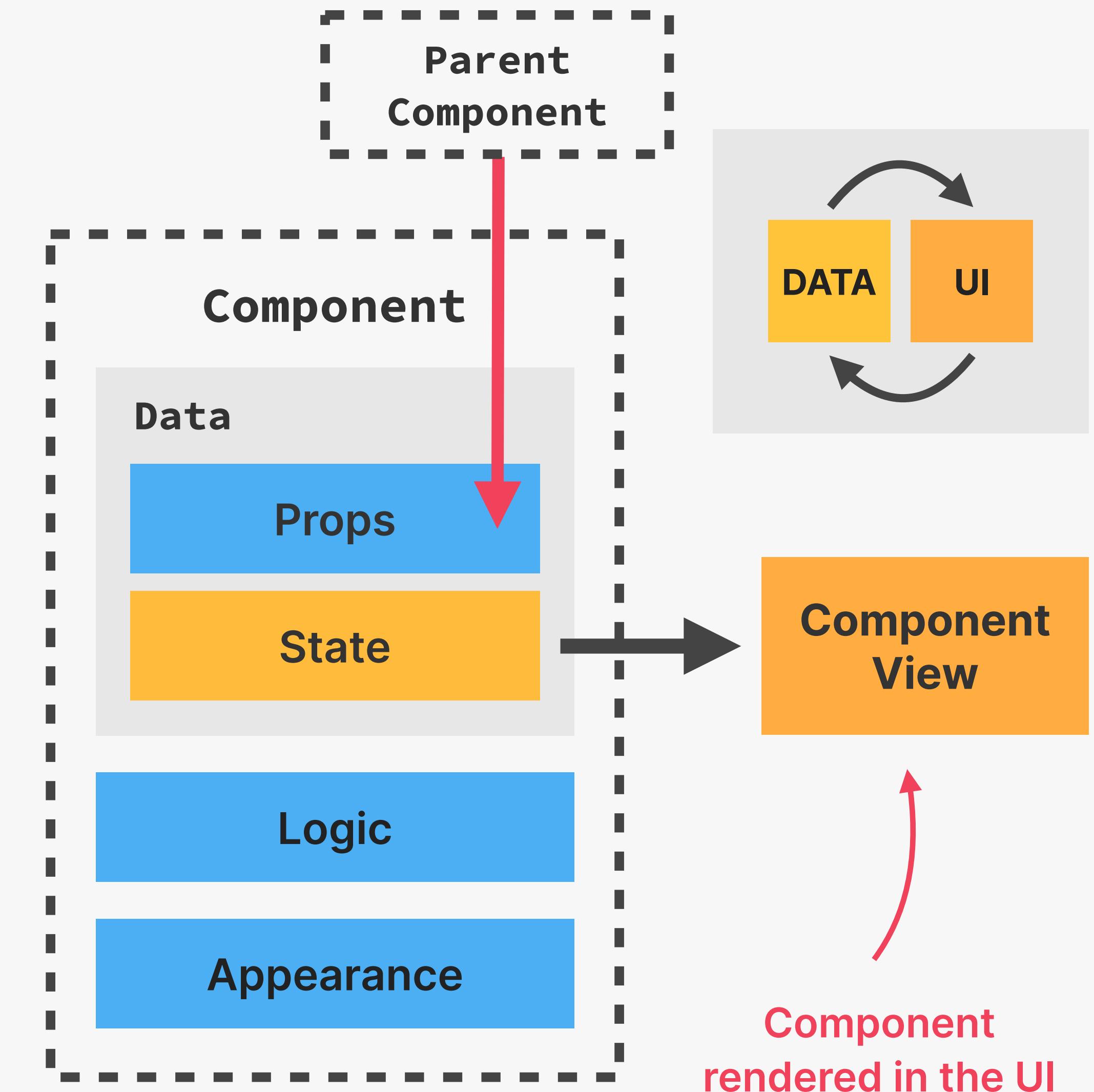
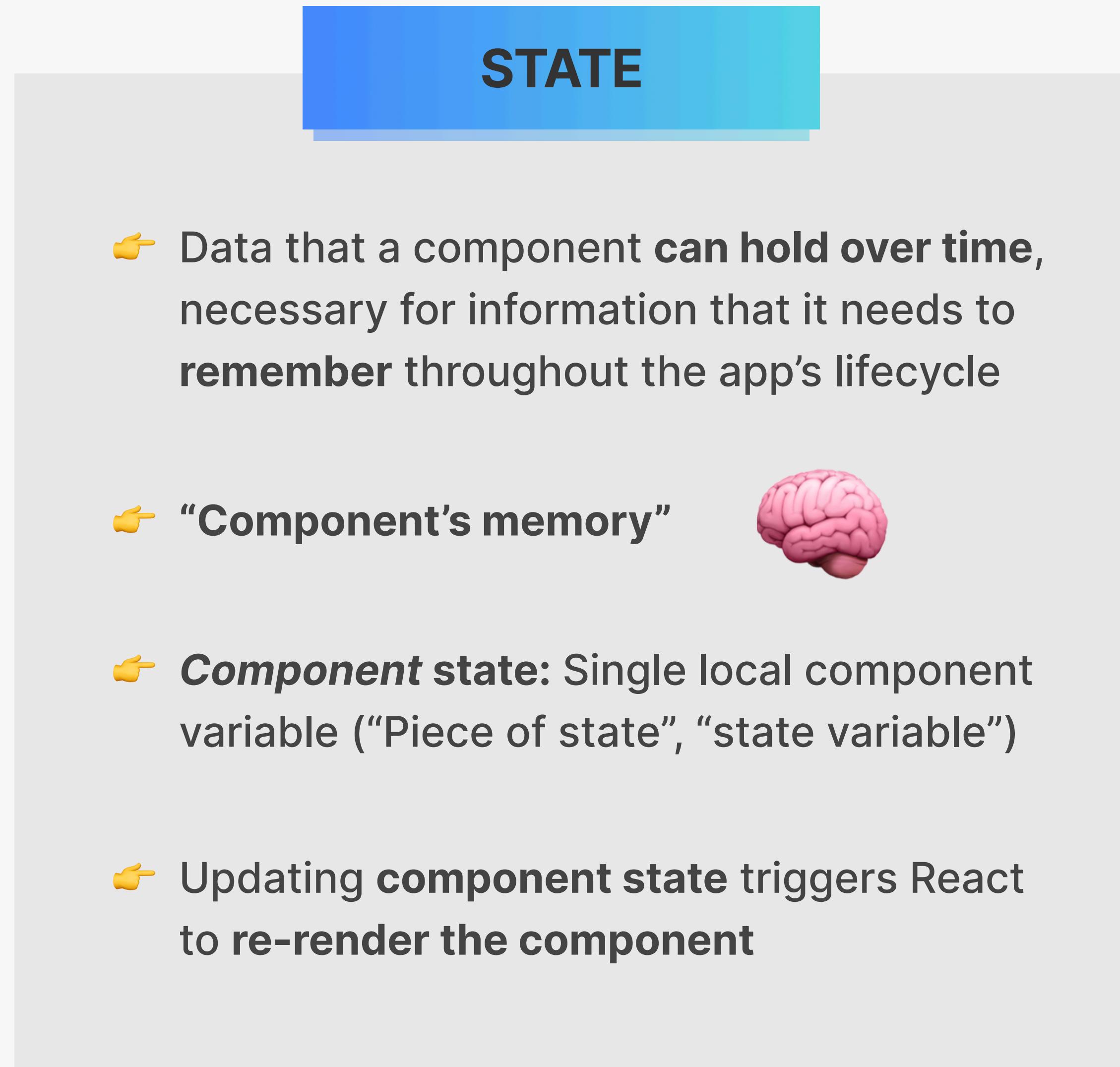
€84.99

Bestseller Updated Recently

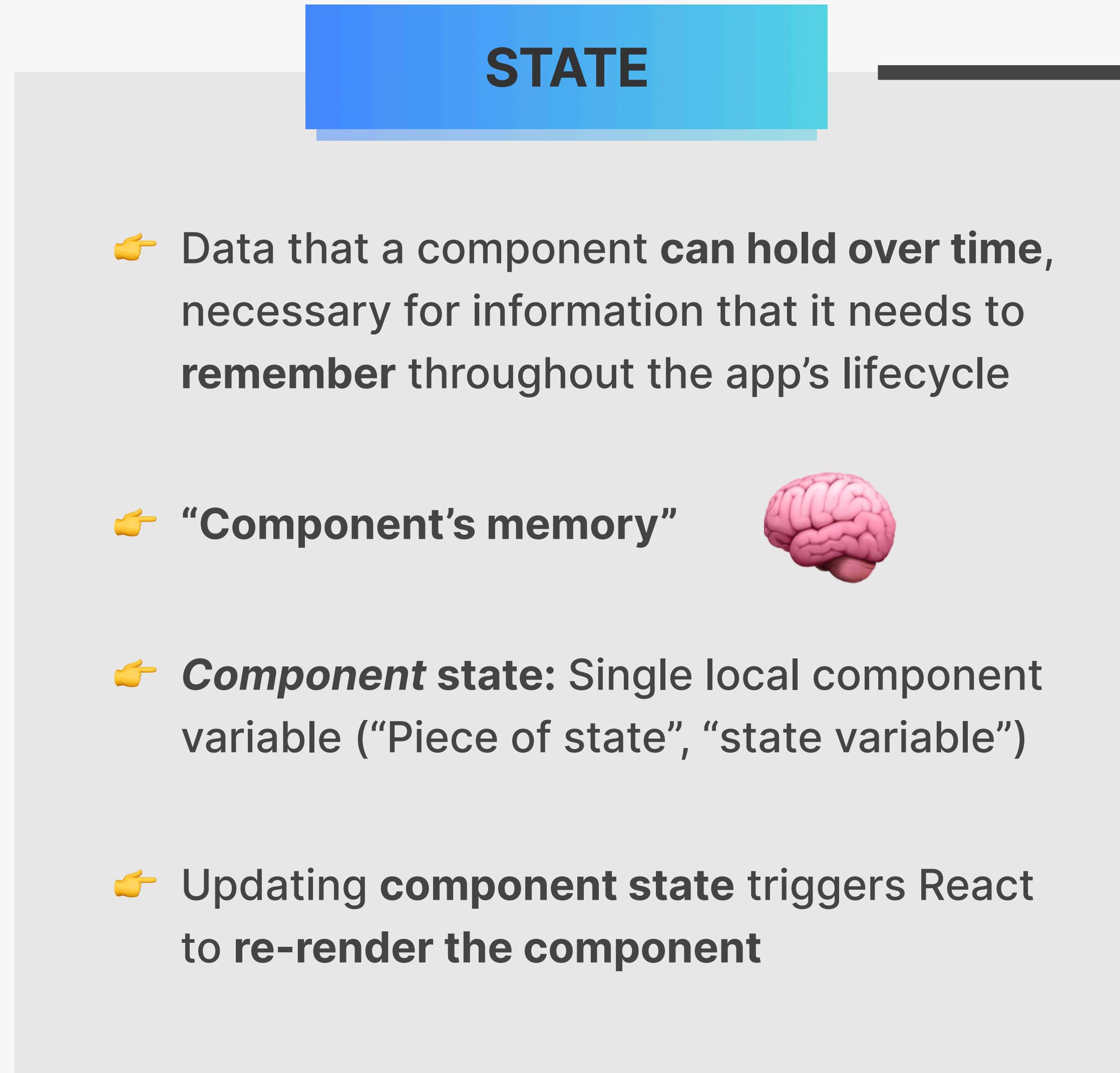
4.7 ★★★★★ (137333 ratings)  
69 total hours • 320 lectures • All Levels

Remove Save for Later

# WHAT IS STATE?



# WHAT IS STATE?



## STATE ALLOWS DEVELOPERS TO:

1

Update the component’s view (by re-rendering it)

2

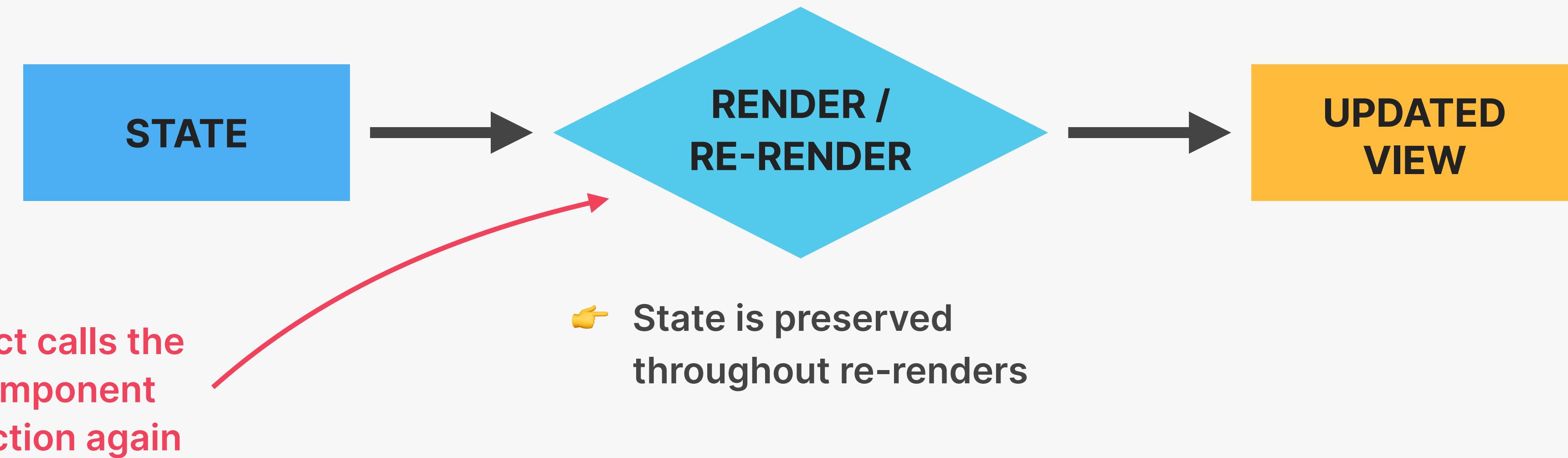
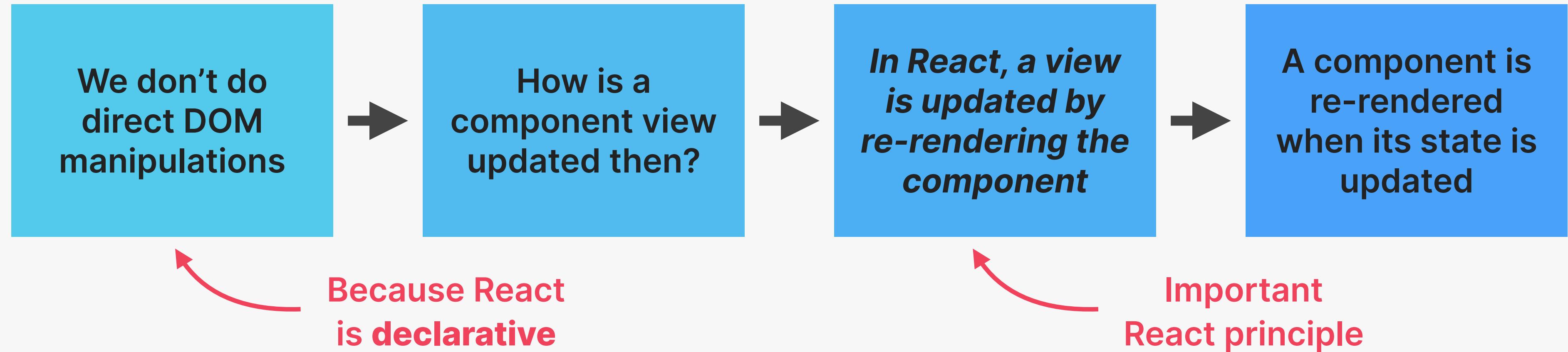
Persist local variables between renders



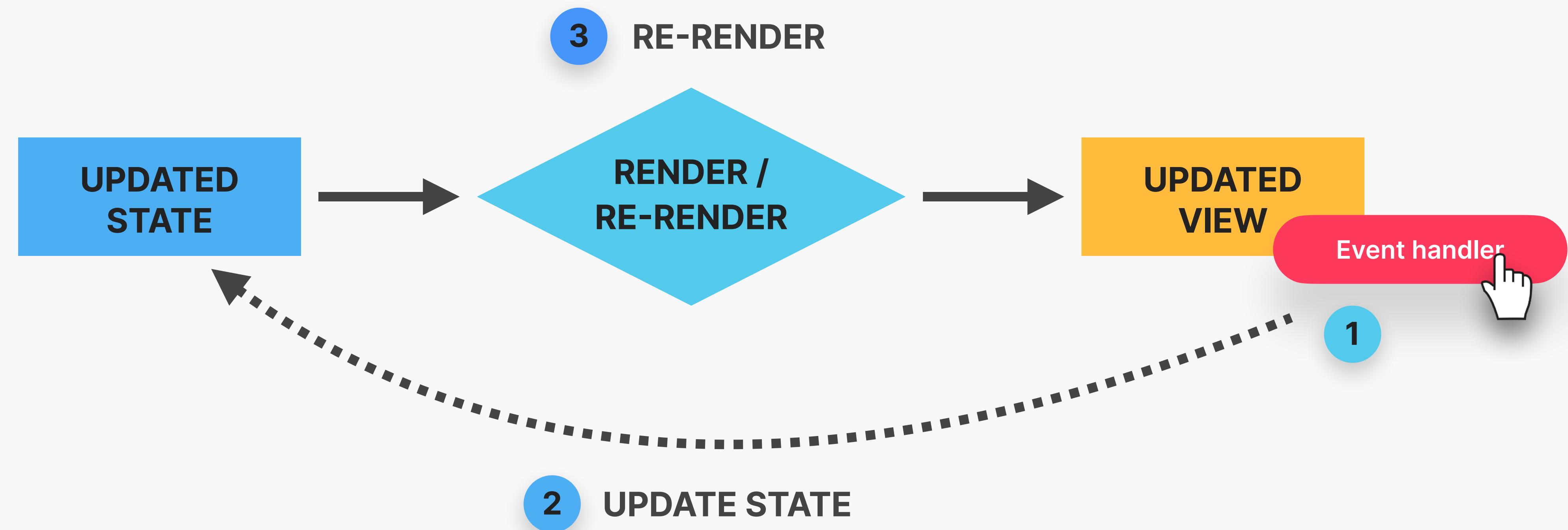
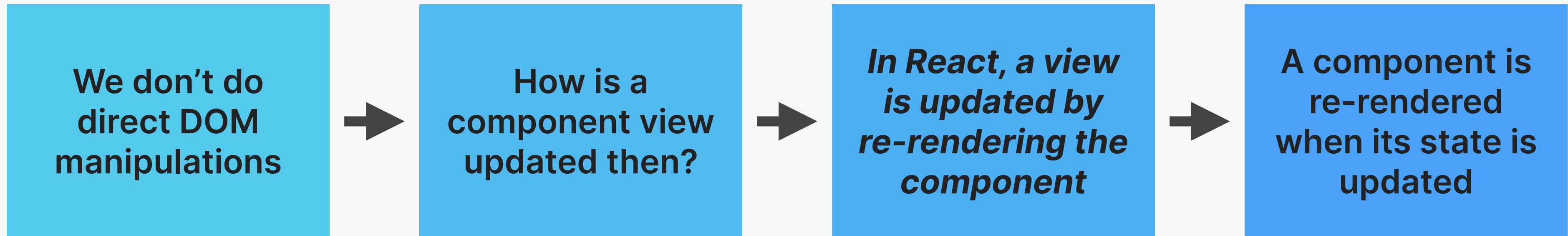
*State is a tool. Mastering state will unlock the power of React development*



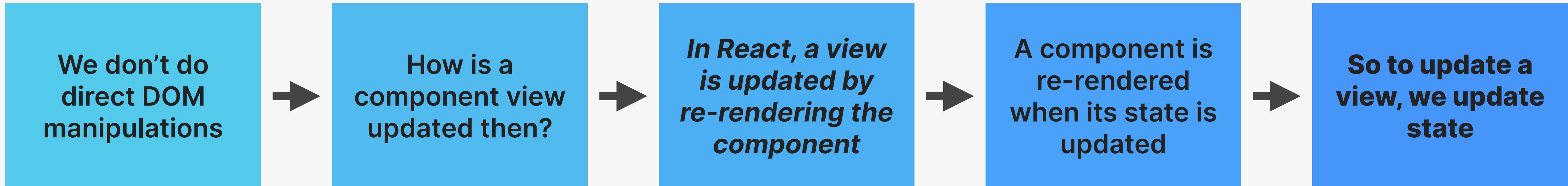
# THE MECHANICS OF STATE IN REACT



# THE MECHANICS OF STATE IN REACT



# THE MECHANICS OF STATE IN REACT

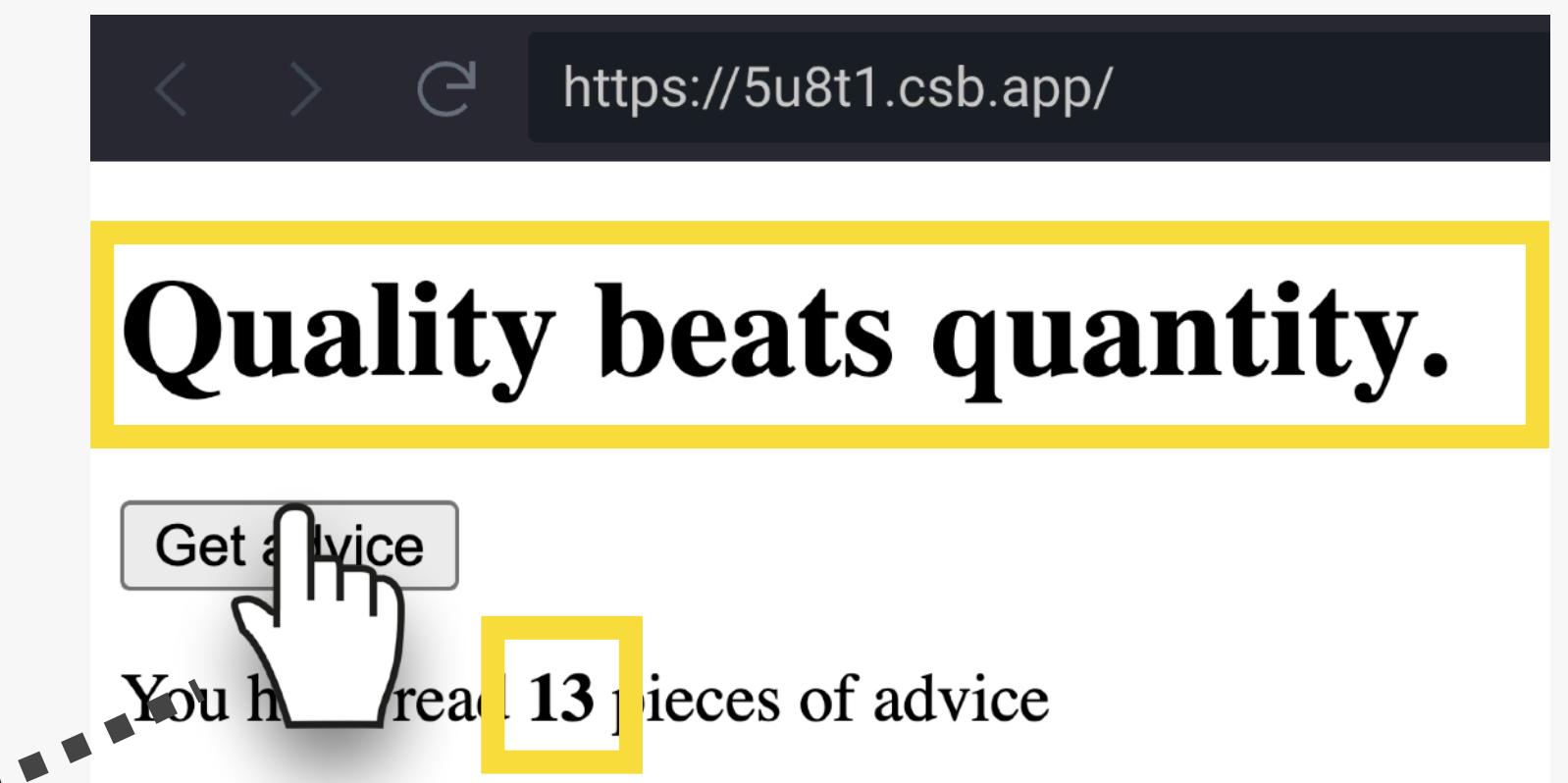


```
const [advice, setAdvice] =  
  useState("Quality beats quantity.");  
const [countAdvice, setCountAdvice] =  
  useState(13);
```

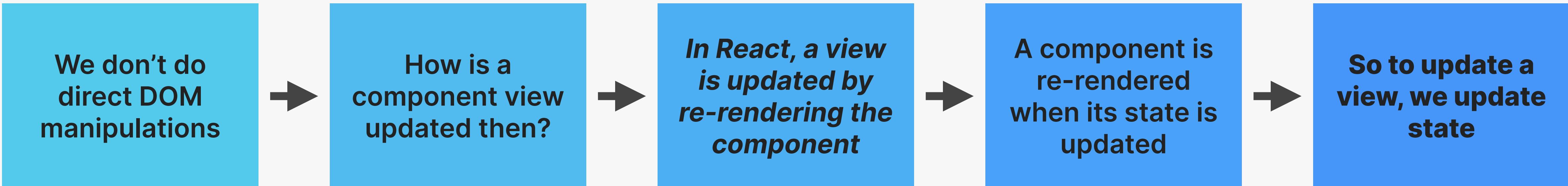
RE-RENDER

UPDATE STATE

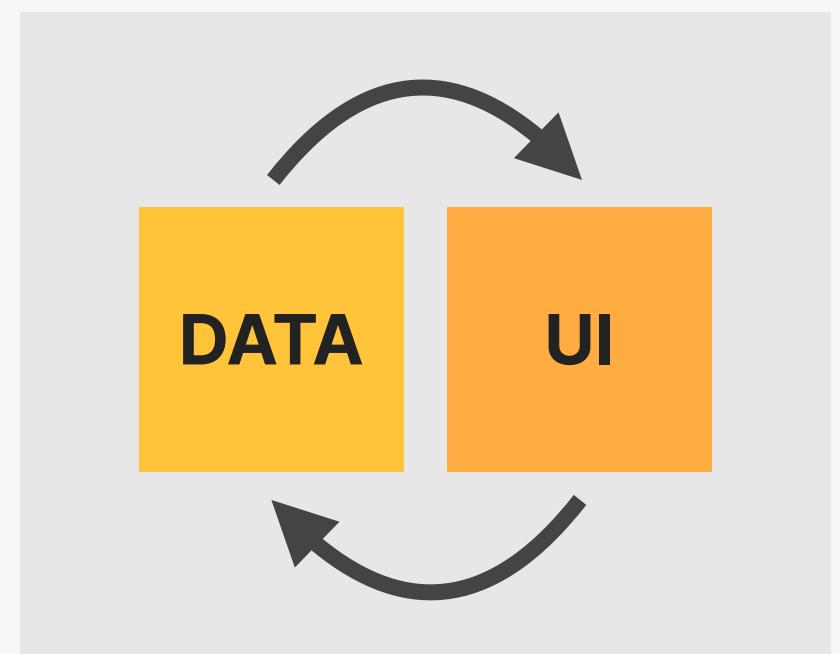
```
setAdvice(data.slip.advice);  
setCountAdvice((count) => count + 1);
```



# THE MECHANICS OF STATE IN REACT

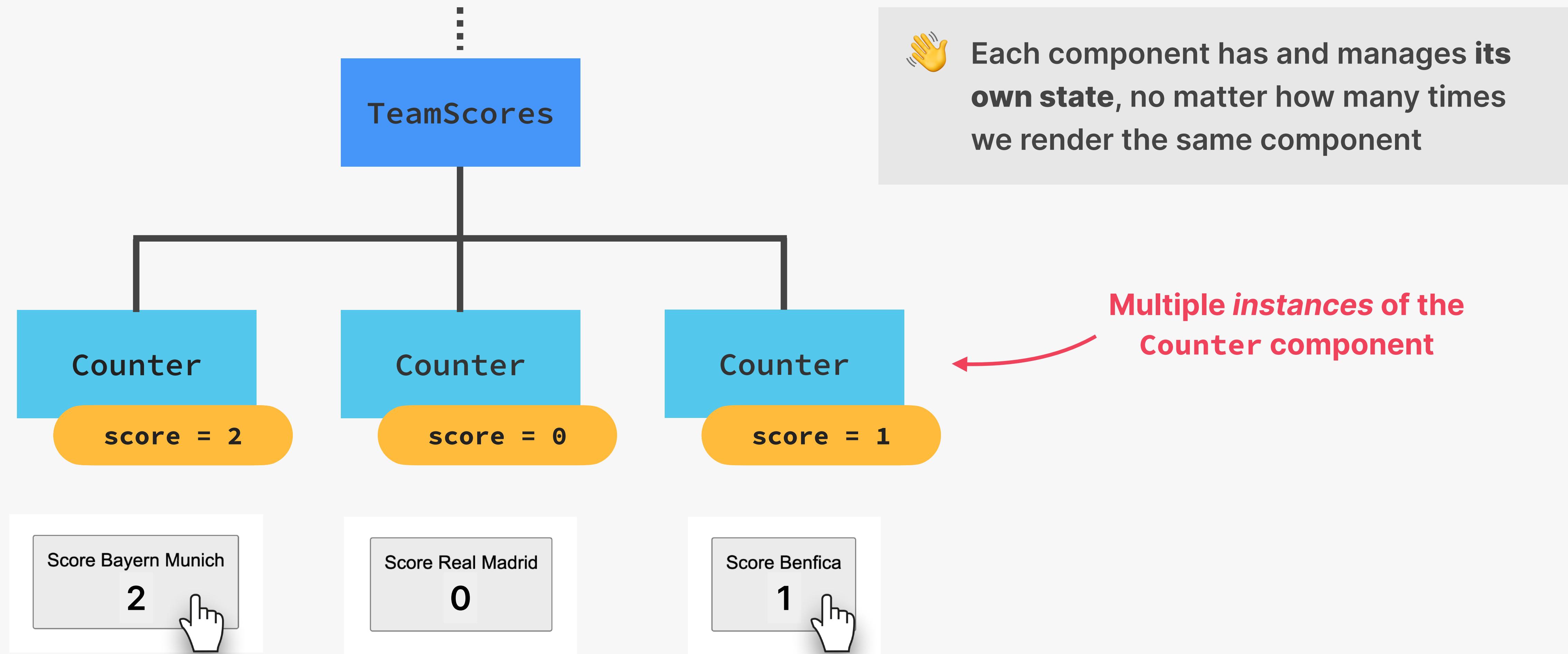


👉 React is called “React” because...

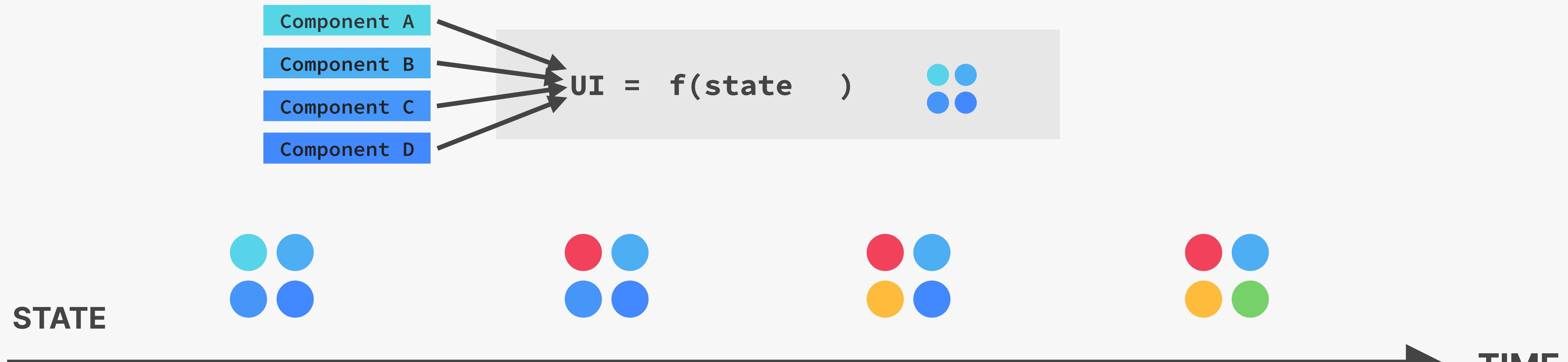




# ONE COMPONENT, ONE STATE



# 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

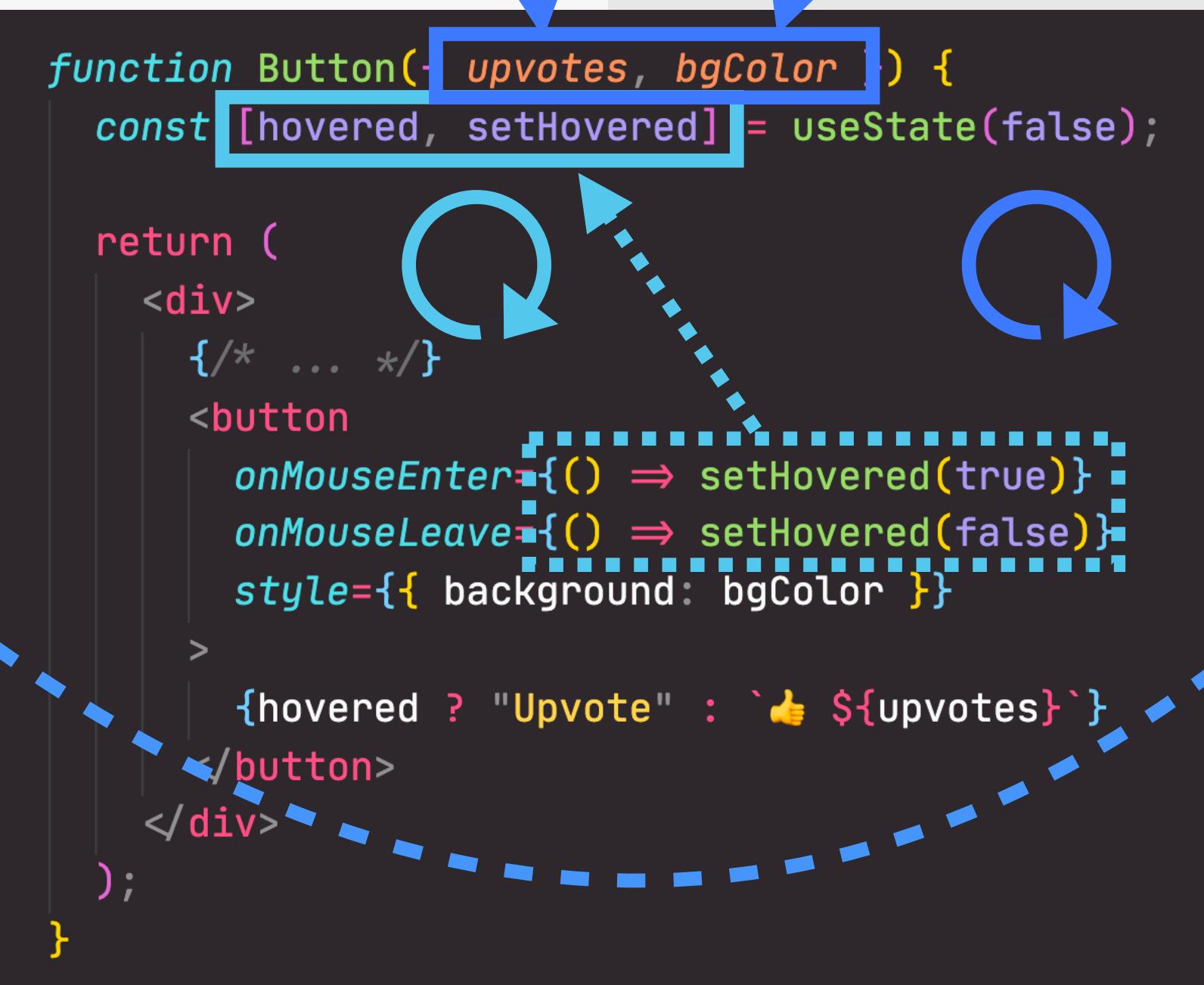
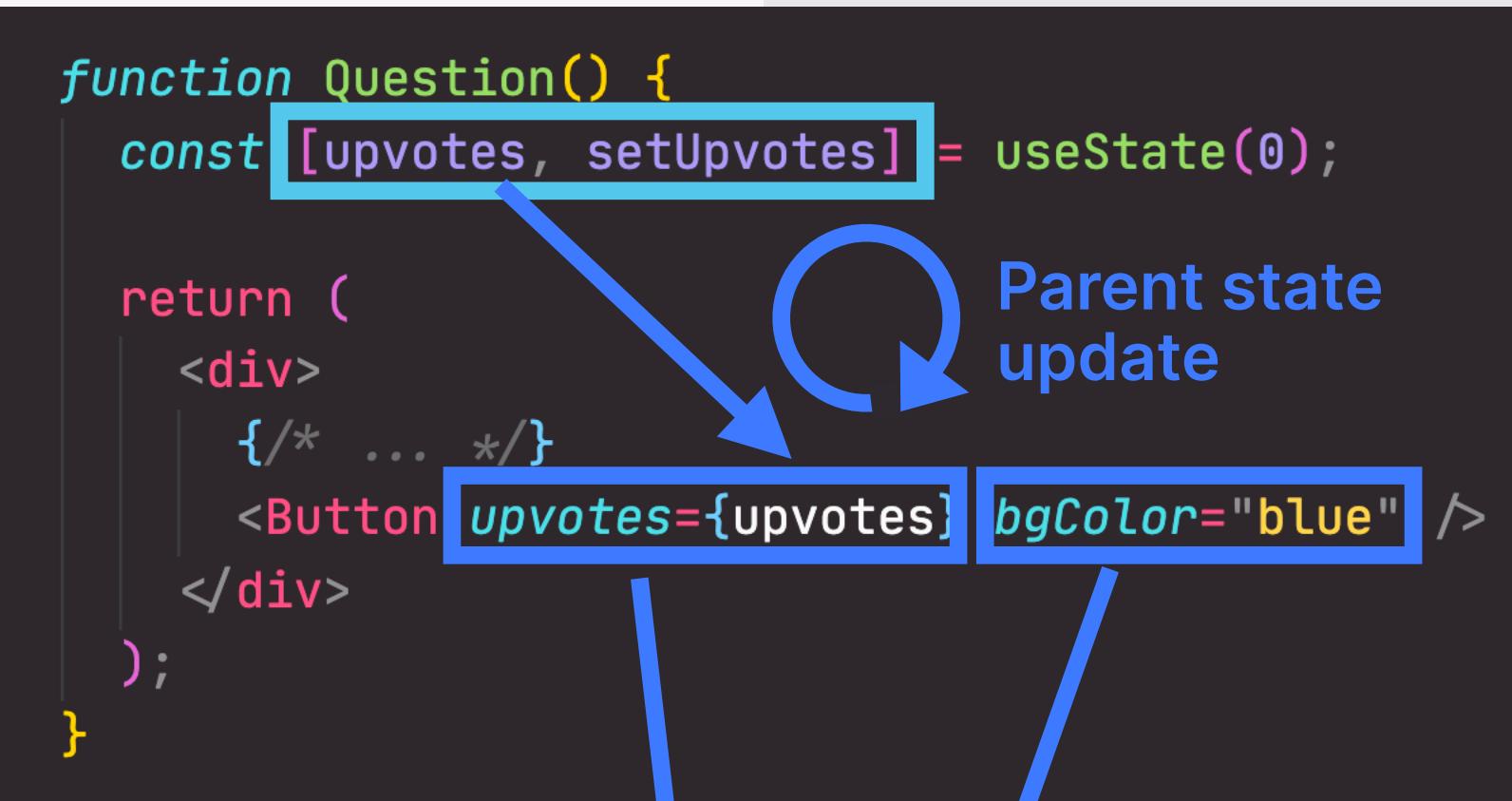
- 👉 Use a state variable for any data that the component should keep track of (“remember”) over time. **This is data that will change at some point.** In Vanilla JS, that’s a `let` variable, or an `[]` or `{}`
- 👉 Whenever you want something in the component to be **dynamic**, create a piece of state 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.**



# 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



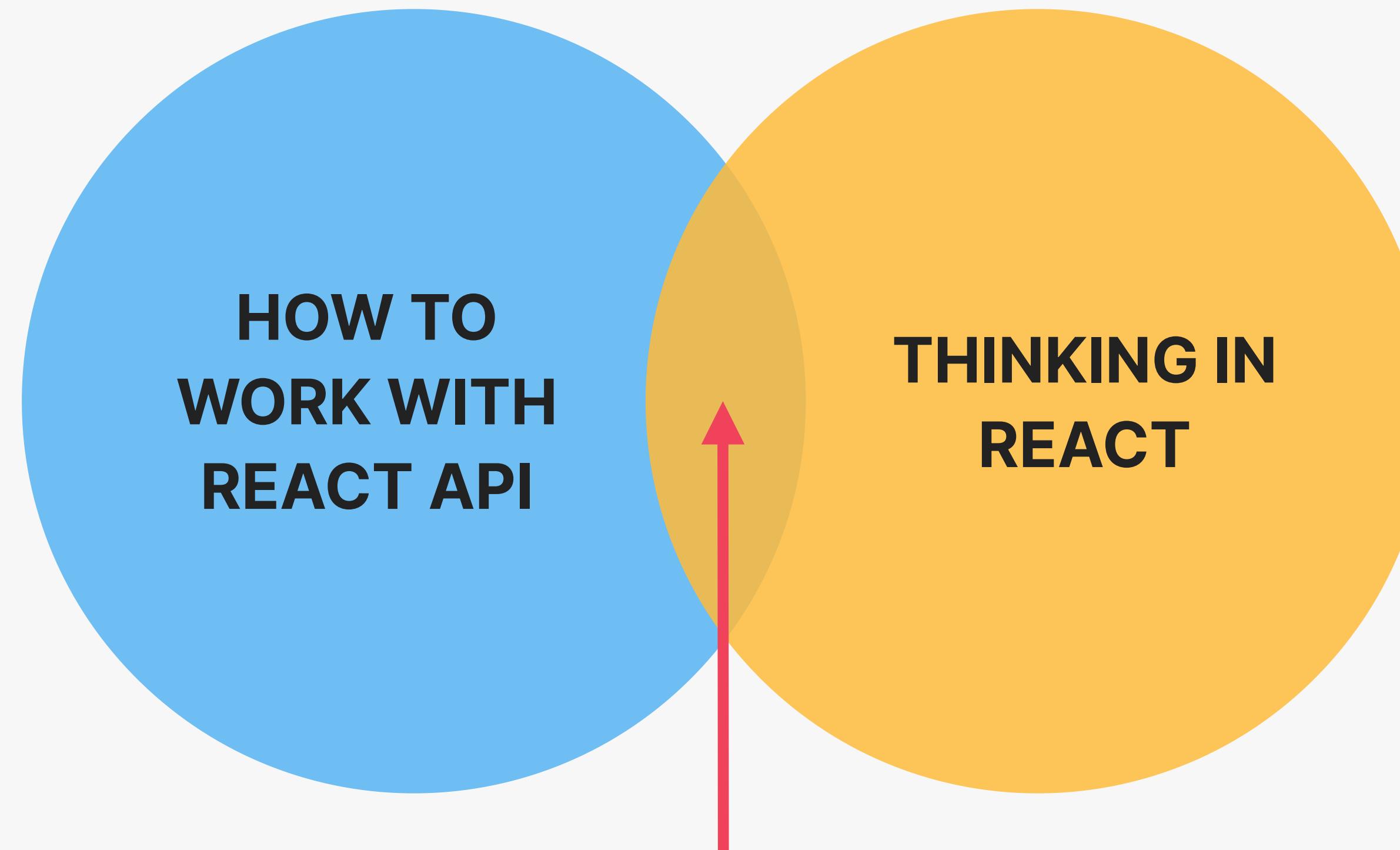
## PROPS

- 👉 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”)



# THINKING IN REACT: STATE MANAGEMENT

# “THINKING IN REACT” IS A CORE SKILL



This is where professional  
React apps are built

## THINKING IN REACT

- 👉 “React Mindset”
- 👉 Thinking about components, state, data flow, effects, etc.
- 👉 Thinking in **state transitions**, not element mutations

# “THINKING IN REACT” AS A PROCESS

Not a rigid process

## THE “THINKING IN REACT” PROCESS:

- 1 Break the desired UI into **components** and establish the **component tree**
- 2 Build a **static** version in React (without state)
- 3 Think about **state**:
  - 👉 When to use state
  - 👉 Types of state: local vs. global
  - 👉 Where to place each piece of state
- 4 Establish **data flow**:
  - 👉 One-way data flow
  - 👉 Child-to-parent communication
  - 👉 Accessing global state

State  
management

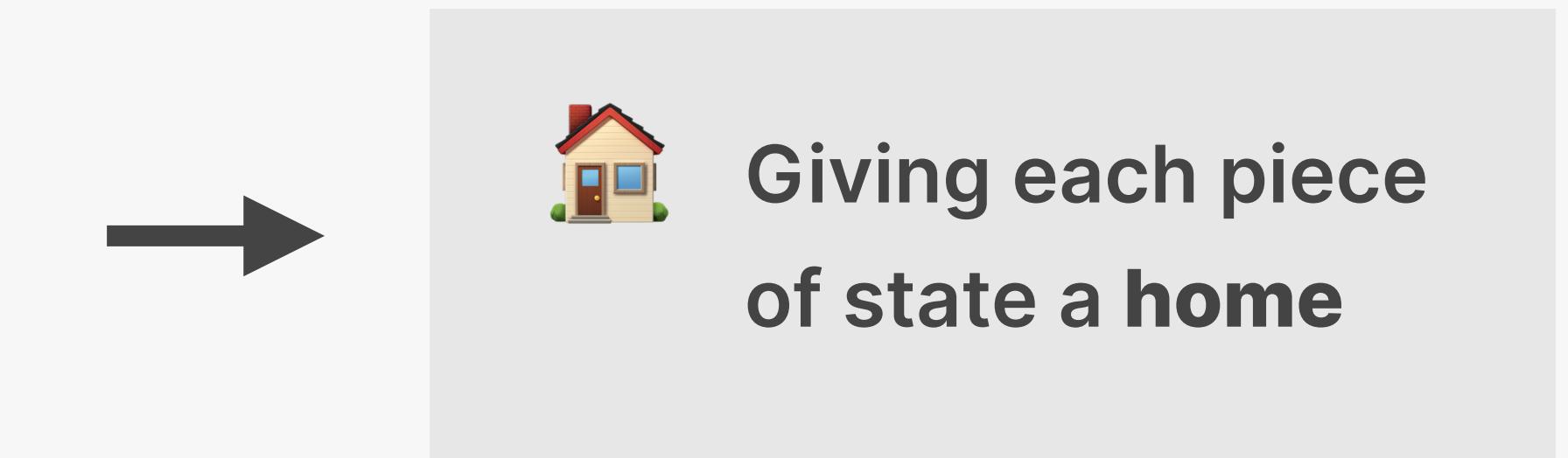
## WHEN YOU KNOW HOW TO “THINK IN REACT”, YOU WILL BE ABLE TO ANSWER:

- 🤔 How to break up a UI design into components?
- 🤔 How to make some components reusable?
- 🤔 How to assemble UI from reusable components?
- 🤔 What pieces of state do I need for interactivity?
- 🤔 Where to place state? (What component should “own” each piece of state?)
- 🤔 What types of state can or should I use?
- 🤔 How to make data flow through app?



# 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



The screenshot shows the Udemy homepage with a search bar containing "javascript". Below it, a "Shopping Cart" section displays two courses: "Node.js, Express, MongoDB & More: The Complete Bootcamp 2022" and "The Complete JavaScript Course 2022: From Zero to Expert!". A red arrow points from the word "searchQuery" to the search bar. Another red arrow points from "shoppingCart" to the shopping cart section. In the center, the text "PIECES OF STATE (useState)" is displayed in red. To the right, a user profile for "Jonas Schmedtmann" is shown with a red arrow pointing from "isopen" to his profile picture. Below the profile, a "Checkout" button is highlighted with a red arrow. Further down, a "Notifications" section is labeled with a red arrow pointing to a list of notifications. At the bottom right, a "Language" section is labeled with a red arrow pointing to the language selection dropdown set to "English".

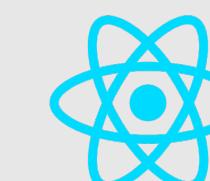
# TYPES OF STATE: LOCAL VS. GLOBAL STATE

## 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)
- 👉 **We should always start with local state**

## GLOBAL STATE

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



Context API



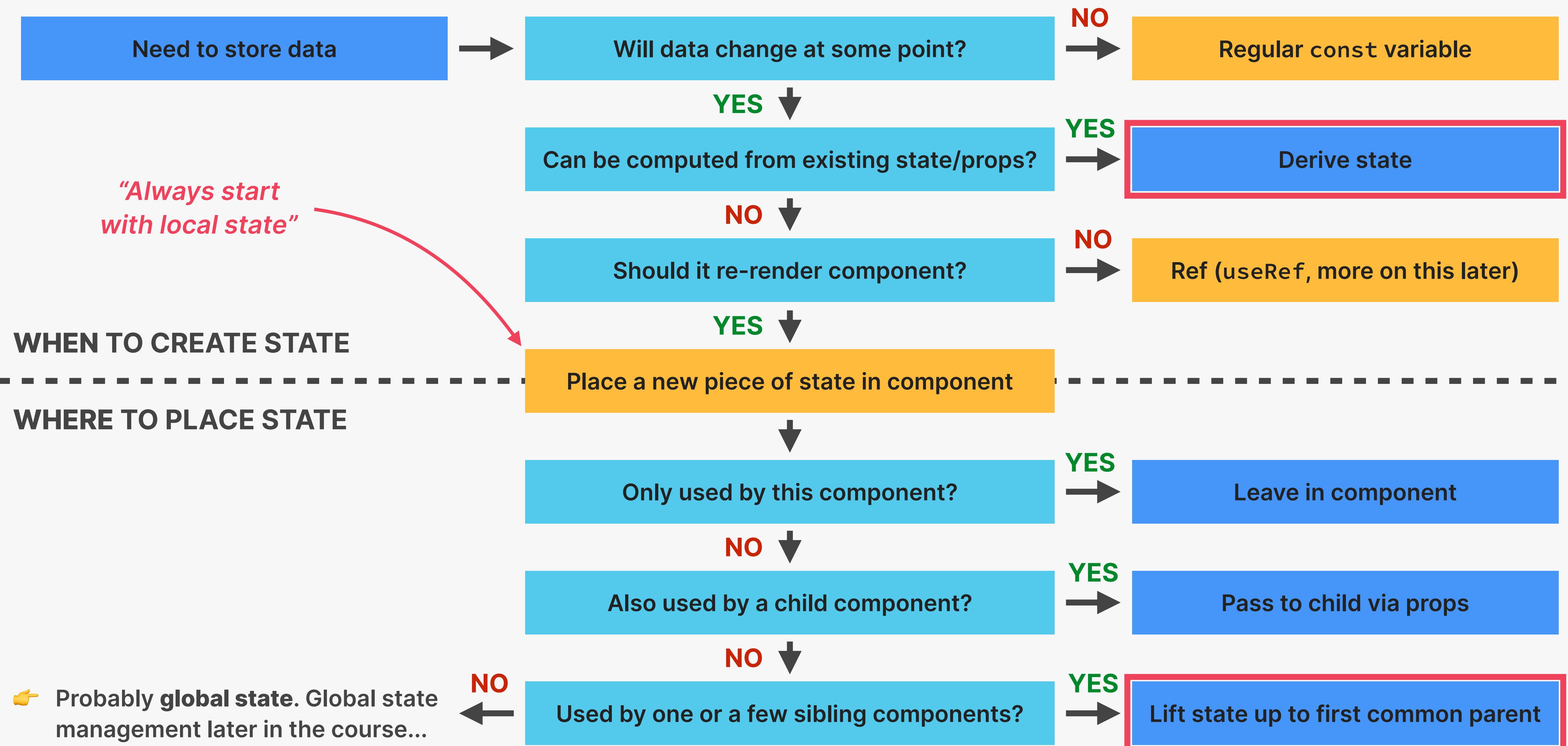
Redux

A screenshot of the Udemy website's shopping cart page. At the top, there is a search bar with the text "javascript". Below the search bar, the word "Shopping Cart" is displayed. The main content area shows two courses in the cart: "Node.js, Express, MongoDB & More: The Complete Bootcamp 2022" and "The Complete JavaScript Course 2022: From Zero to Expert!". Each course listing includes a small thumbnail, the course title, the instructor's name, ratings, and a brief description. To the right of the course lists, there are buttons for "Remove", "Save for Later", and "Move to Wishlist". Next to the "Remove" button, the original price is listed as "€84.99" and the discounted price is "€12.99". A "Checkout" button is located at the bottom right of this section. On the far right, a sidebar displays the user's profile picture, name "Jonas Schmedtmann", and a summary of their learning progress. The sidebar also includes links for "My learning", "Notifications" (with 9+ notifications), "Messages" (with 9+ messages), "Account settings", and "Payment methods". A blue arrow points from the text "Local state" to the search bar. Another blue arrow points from the text "Global state" to the sidebar.

Local state

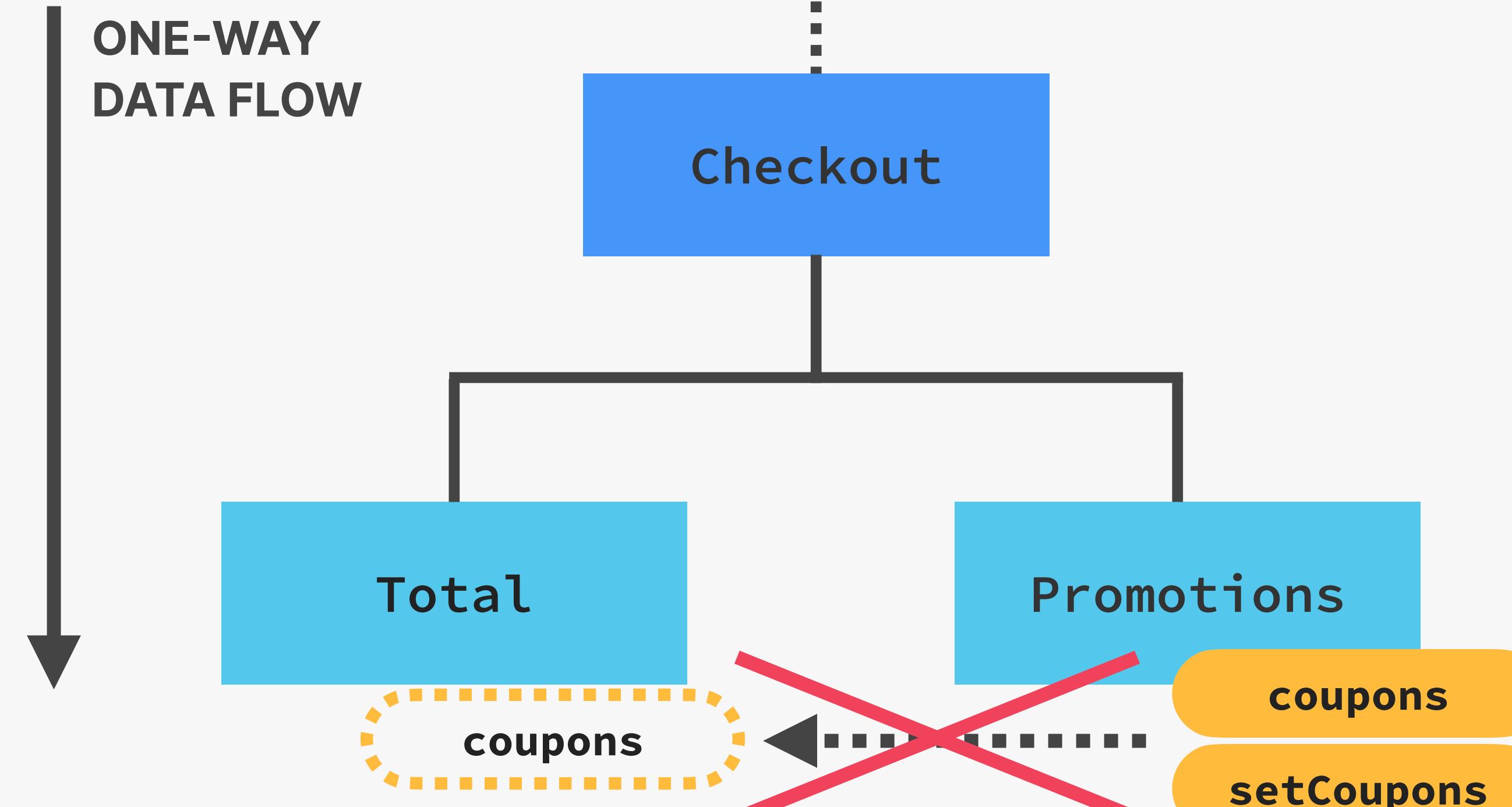
Global state

# STATE: WHEN AND WHERE?





# PROBLEM: SHARING STATE WITH SIBLING COMPONENT

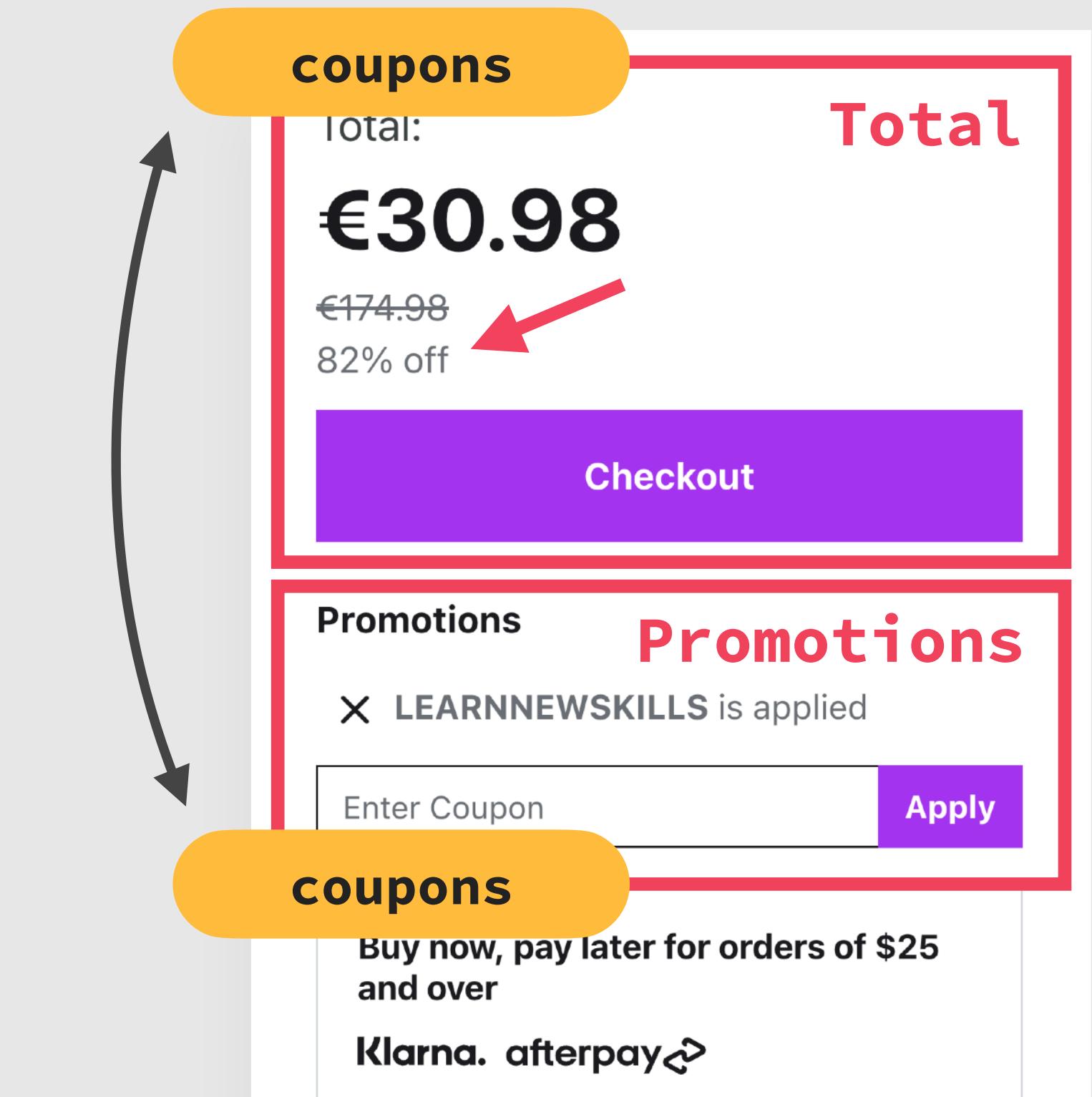


Data can only flow down to children  
(via props), not sideways to siblings

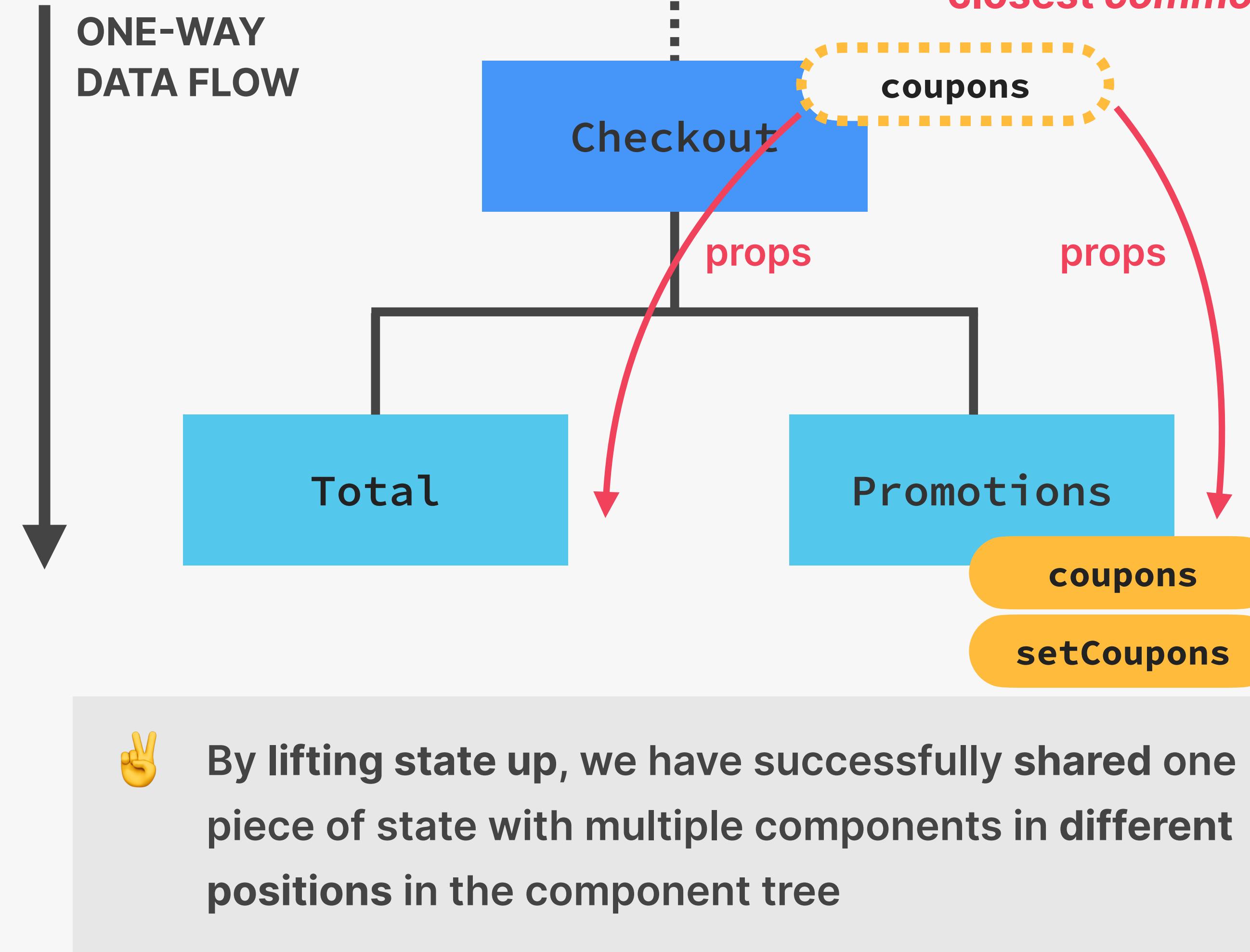


How do we share state with other components?

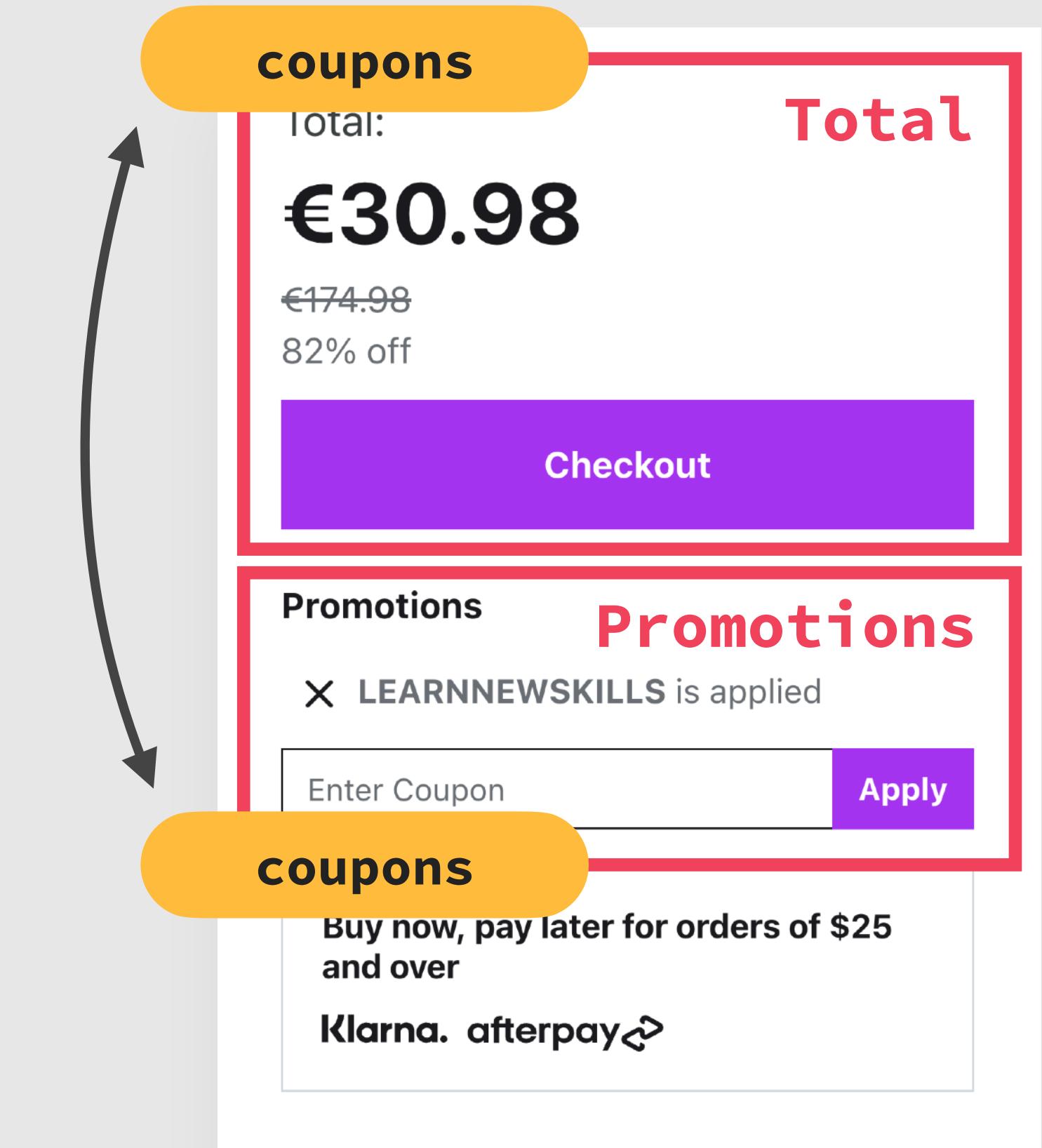
👉 Total component also needs access to coupons state



# SOLUTION: LIFTING STATE UP

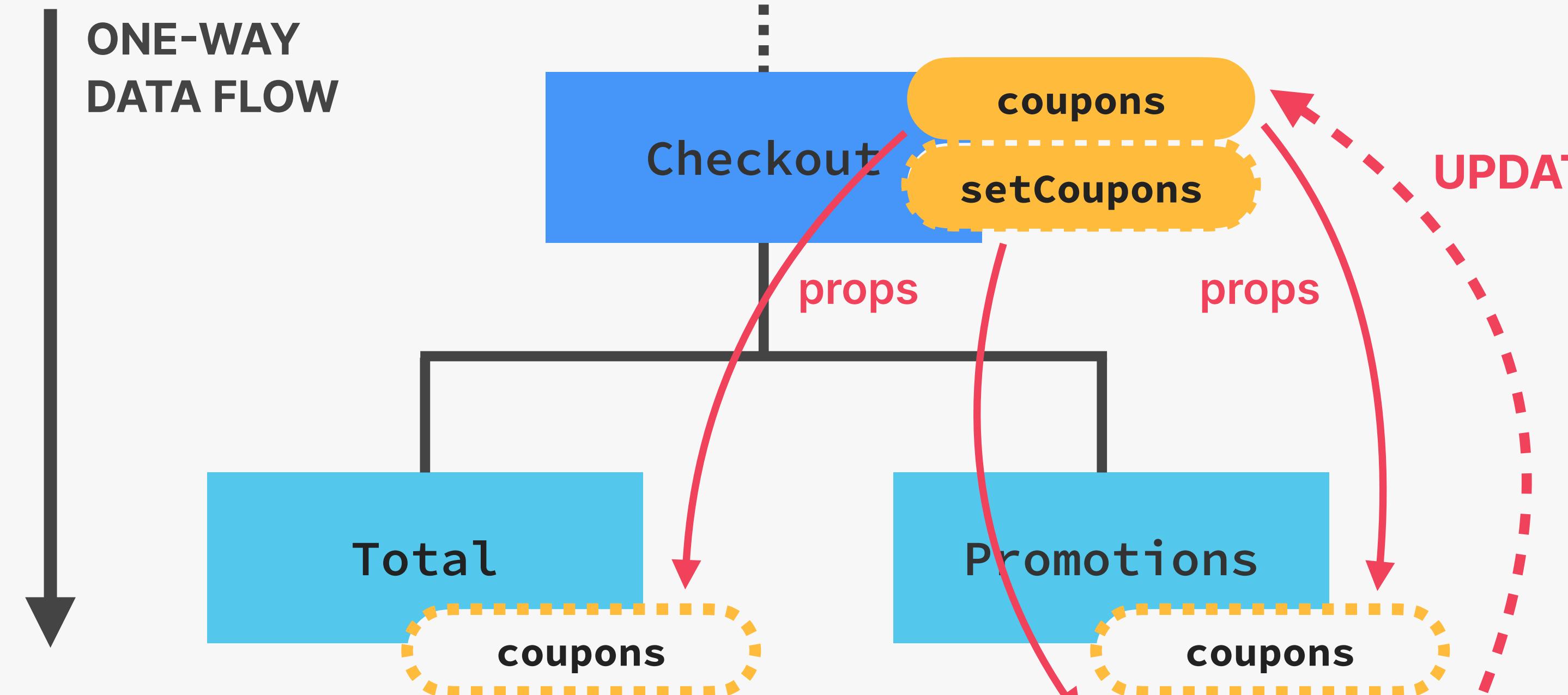


👉 Total component also needs access to coupons state



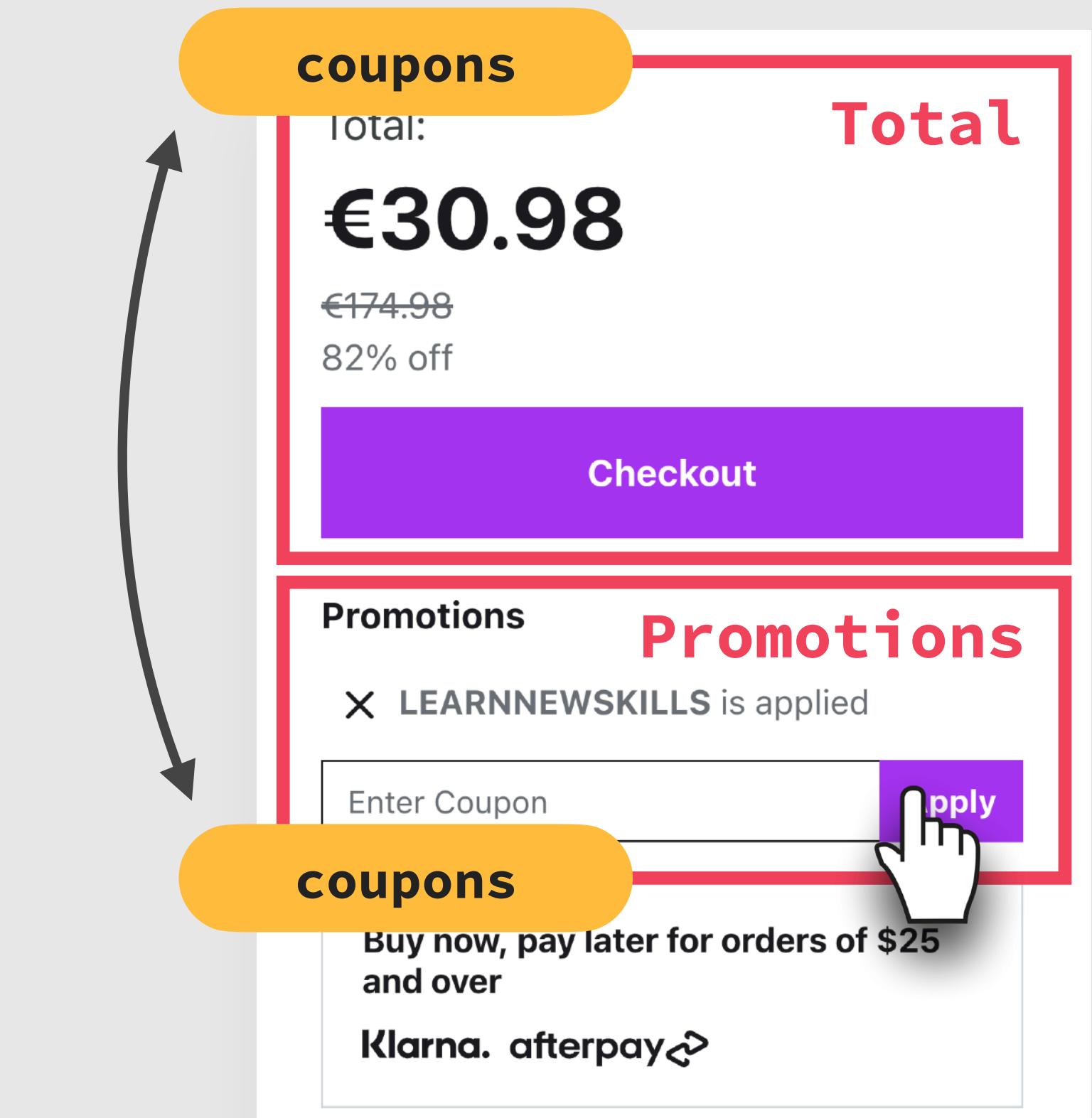
# CHILD-TO-PARENT COMMUNICATION

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



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

- 👉 Total component also needs access to coupons state





# 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

```
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

- 👍 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

## 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);
```



# THE CHILDREN PROP



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

Children of Button,  
accessible through  
`props.children`

The diagram shows two code snippets demonstrating the use of the `children` prop.

**Previous Component:**

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

**Party Component:**

```
<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)



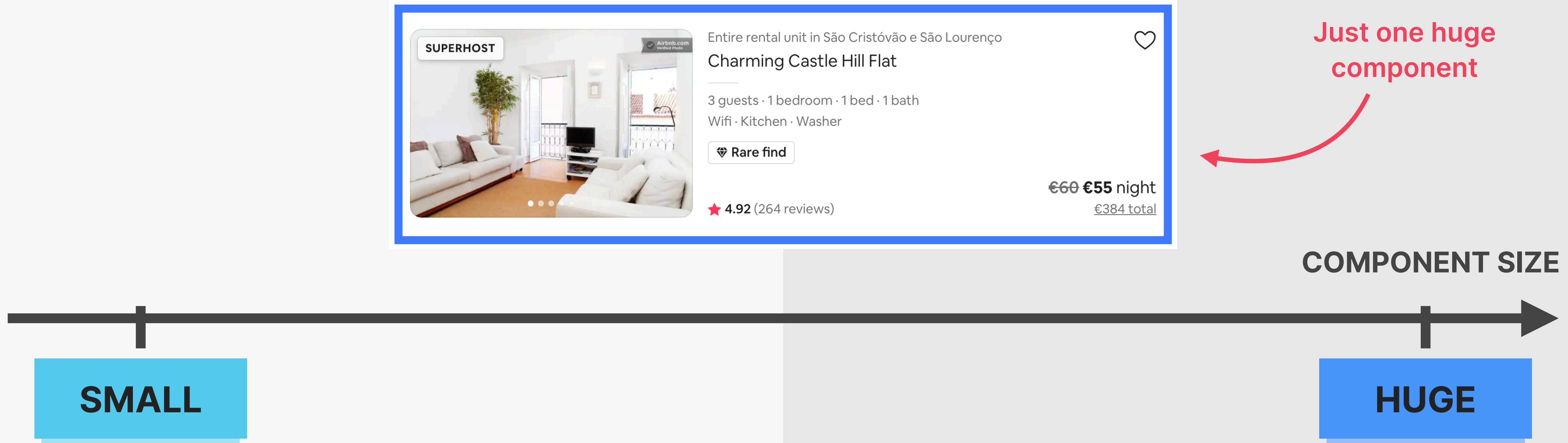
# PART 02

---

# INTERMEDIATE REACT

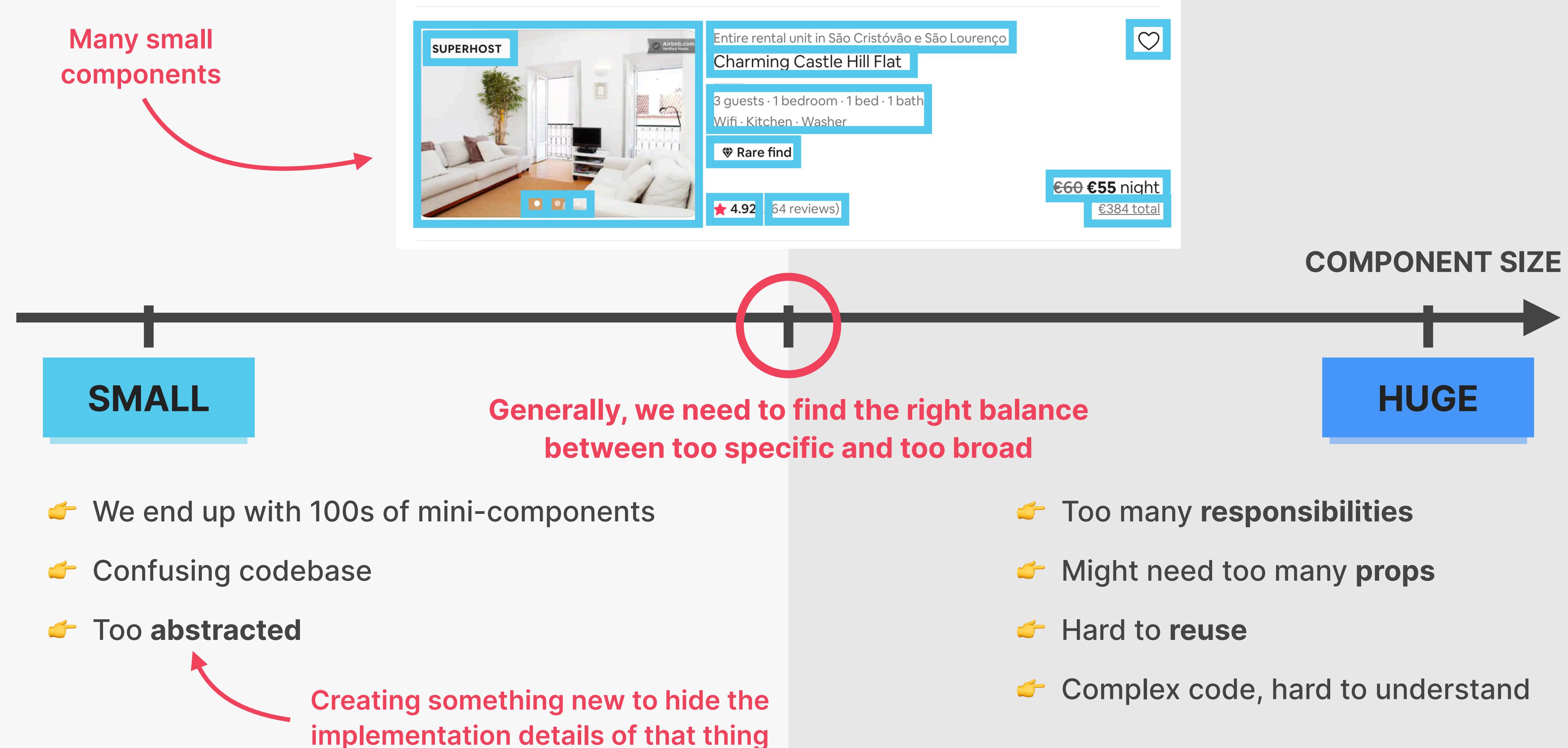
# THINKING IN REACT: COMPONENTS, COMPOSITION, AND REUSABILITY

# COMPONENT SIZE MATTERS



- 👉 Too many **responsibilities**
- 👉 Might need too many **props**
- 👉 Hard to **reuse**
- 👉 Complex code, hard to understand

# COMPONENT SIZE MATTERS



# HOW TO SPLIT A UI INTO COMPONENTS

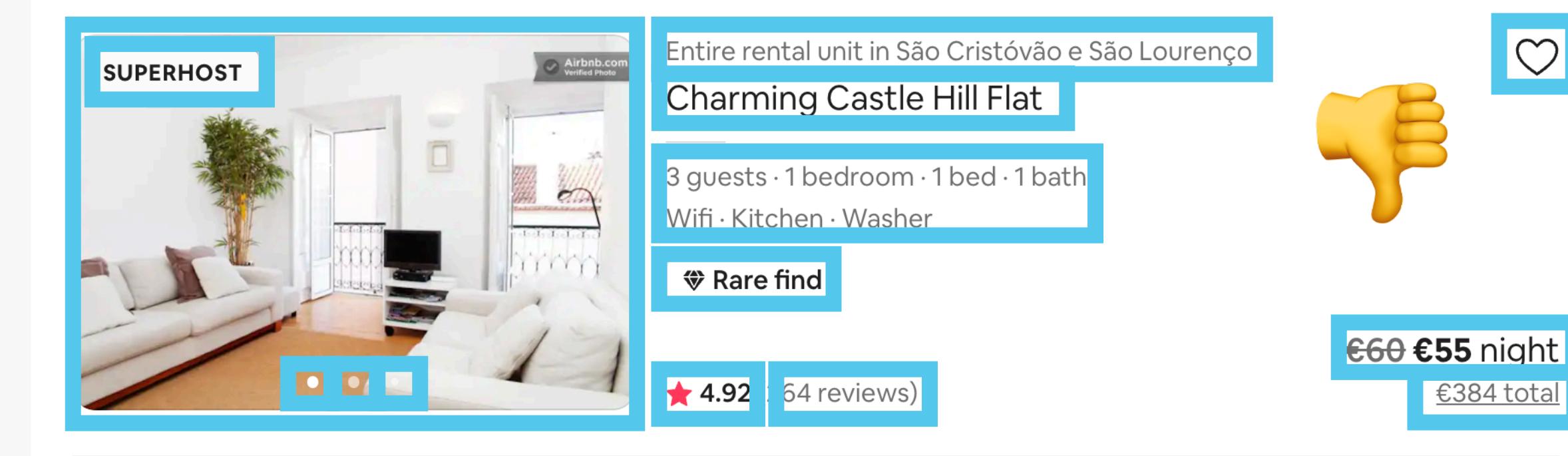
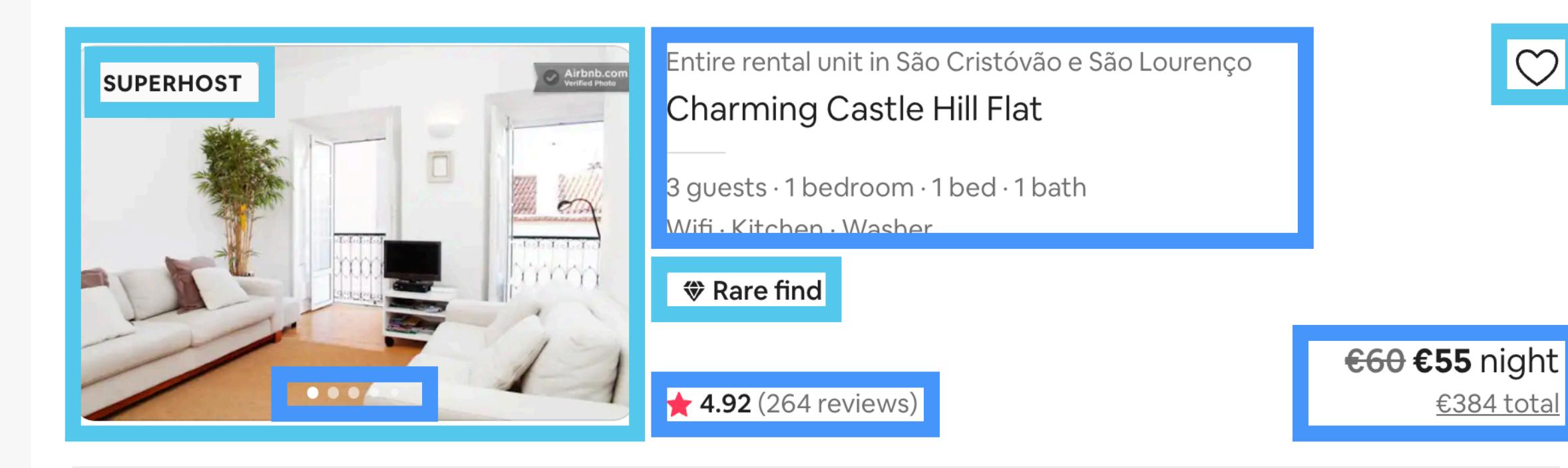
👉 The 4 criteria for splitting a UI into components:

**1. Logical separation of content/layout**

**2. Reusability**

**3. Responsibilities / complexity**

**4. Personal coding style**



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

# FRAMEWORK: WHEN TO CREATE A NEW COMPONENT?



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

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

## 1. Logical separation of content/layout

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

## 2. Reusability

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

## 3. Responsibilities / complexity

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

## 4. Personal coding style

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



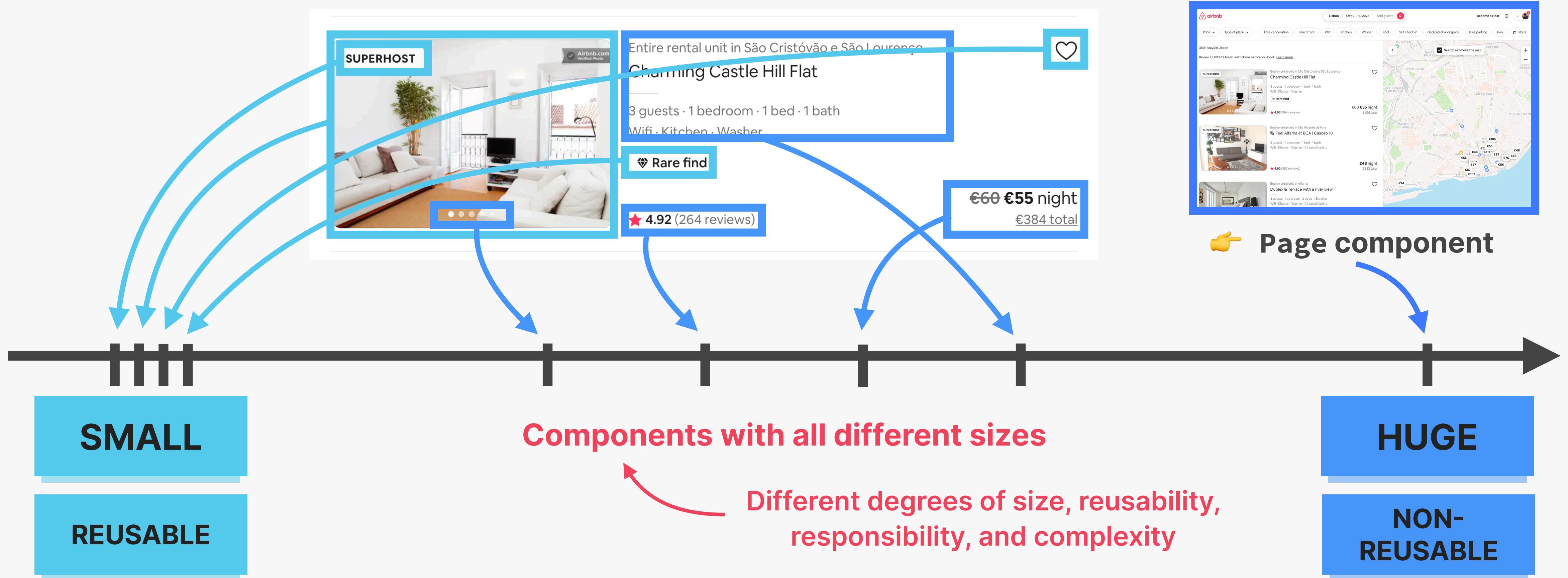
You might need a new component

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

# SOME MORE GENERAL GUIDELINES

- 💰 Be aware that creating a new component **creates a new abstraction**. Abstractions have a **cost**, because **more abstractions require more mental energy** to switch back and forth between components. So try not to create new components too early
- 🏷️ Name a component according to **what it does or what it displays**. Don't be afraid of using long component names
- 🧩 Never declare a new component **inside another component!**
- 📁 **Co-locate related components inside the same file.** Don't separate components into different files too early
- 🔄 It's completely normal that an app has components of **many different sizes**, including very small and huge ones (See next slide... 👉)

# ANY APP HAS COMPONENTS OF DIFFERENT SIZES AND REUSABILITY



- 👉 Some very small components are necessary!
- 👉 Highly reusable
- 👉 Very low complexity

- 👉 Most apps will have a few huge components
- 👉 Not meant to be reused (not a problem!)

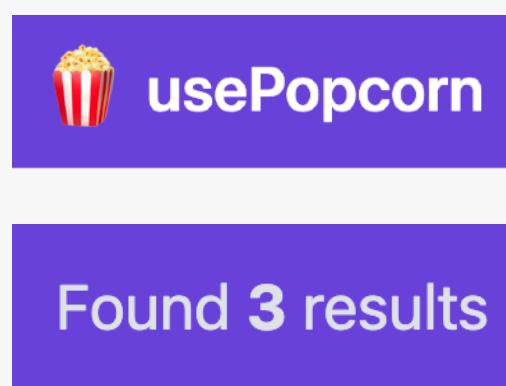


# COMPONENT CATEGORIES

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

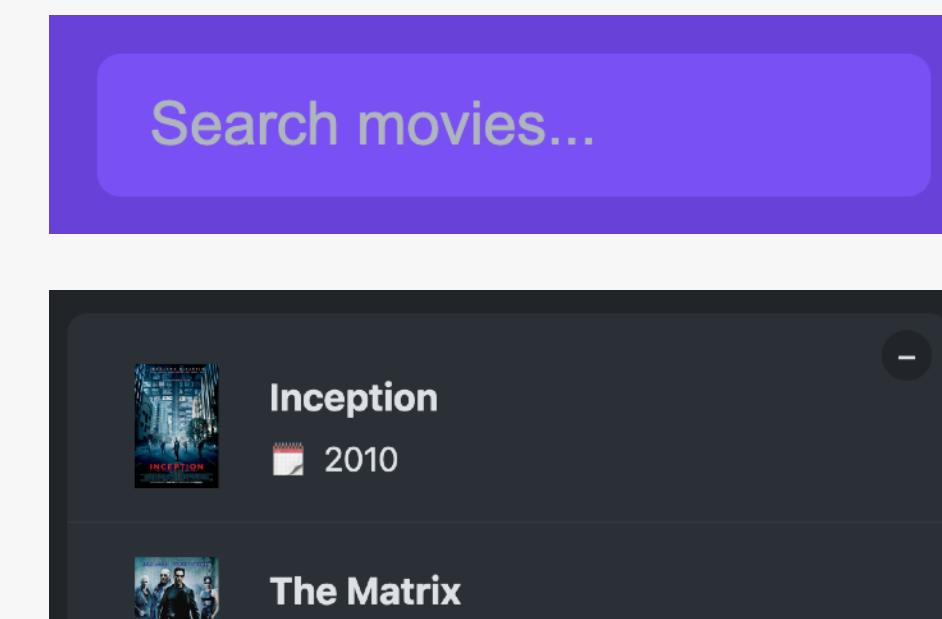
**Stateless /  
presentational  
components**

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



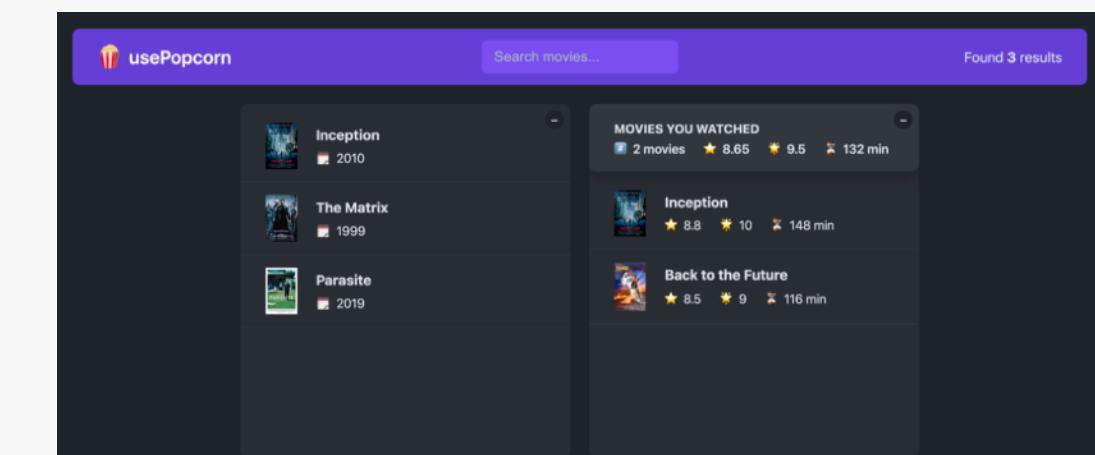
**Stateful  
components**

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



**Structural  
components**

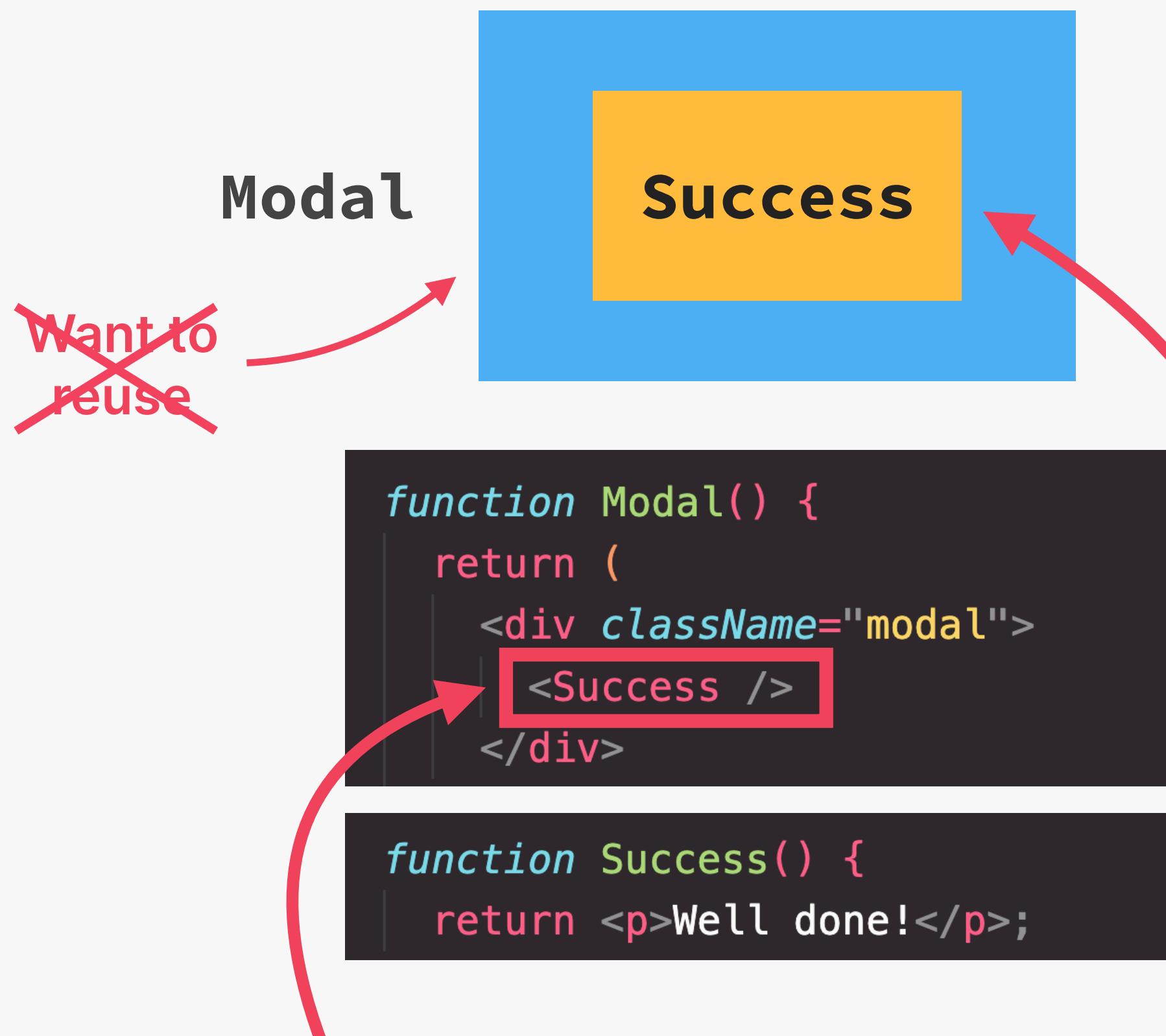
- 👉 “Pages”, “layouts”, or “screens” of the app
- 👉 Result of **composition**
- 👉 Can be **huge** and non-reusable (but don't have to)



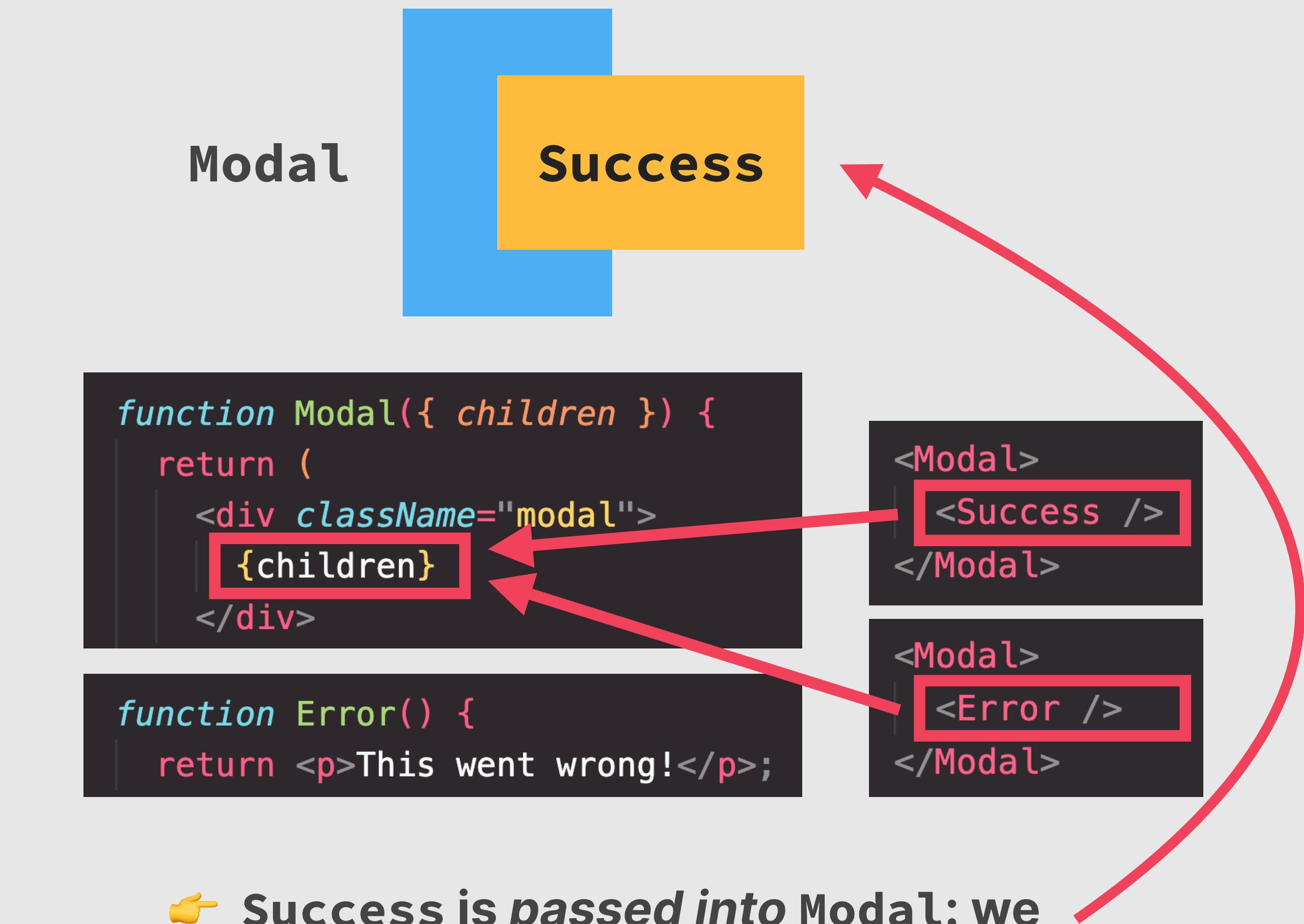


# WHAT IS COMPONENT COMPOSITION?

## “USING” A COMPONENT



## COMPONENT COMPOSITION



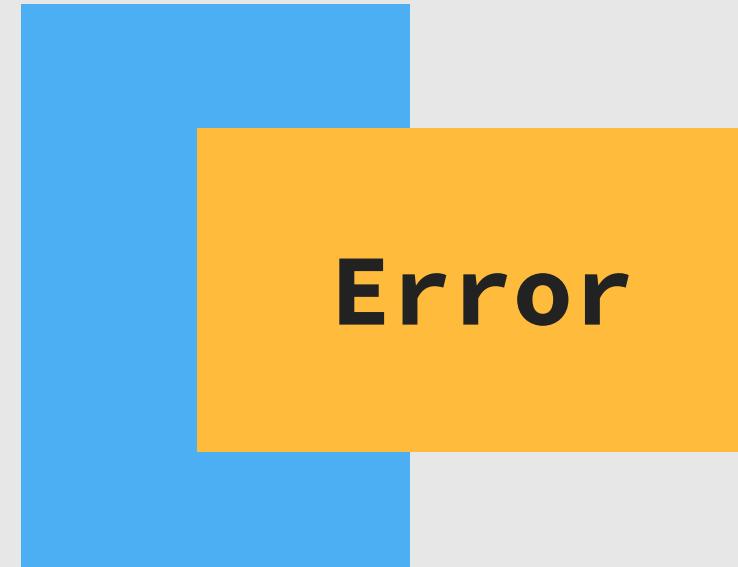
👉 Success is *inside* Modal: we can NOT reuse Modal

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

# WHAT IS COMPONENT COMPOSITION?

## COMPONENT COMPOSITION

Modal



```
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 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



# PROPS AS AN API



COMPONENT CONSUMER



Component props = Public API

Consuming component



Abstraction that encapsulates UI and logic

```
function StarRating({  
  maxRating = 5,  
  defaultRating = 0,  
  ...  
} = {}){  
  const [rating, setRating] = useState(defaultRating);  
  
  // Add these in the end  
  const messages = [];  
  const color = "#ffccbc";  
  const size = 16;  
  const className = `star-rating ${rating}`;  
  const onRatingChange = (rating) => useState(rating);  
  
  const handleSetting = function (rating) {  
    setRating(rating);  
  };  
  
  const containerStyle = { display: 'flex', alignItems: 'center', gap: '16px' };  
  
  const starContainerStyle = { display: 'flex' };  
  
  return (  

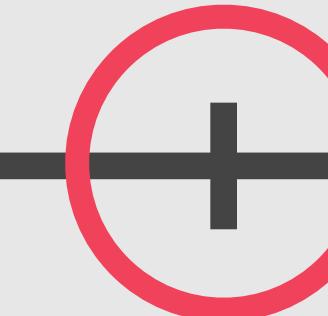

<div style={starContainerStyle}>  
      {Array.from({ length: maxRating }, (_, i) => {  
        <Star rating={i}>  
        if(i < rating) tempRating > i + 1 ? rating >= i + 1 :  
          onClick={() => handleSetting(i + 1)}  
          onMouseEnter={() => setTempRating(i + 1)}  
          onMouseLeave={() => setTempRating(0)}  
          color={color}  
          size={size}  
        />  
      })</div>  
    </div>  
  );  
}  
  
// Show messages if a message array has been passed in, and has the correct  
// length  
if(messages.length === maxRating - 1 || tempRating - 1 >= rating - 1){  
  <p style={textStyle}>  
    {messages.map((message, index) =>   
      <span>{message.message}</span>  
    )}  
  </p>  
}


```

Component



COMPONENT CREATOR



TOO LITTLE PROPS

- 👉 Not flexible enough
- 👉 Might not be useful

We need to find the right balance between too little and too many props, that works for both the consumer and the creator

TOO MANY PROPS

- 👉 Too hard to use
- 👉 Exposing too much complexity
- 👉 Hard-to-write code
- 👉 Provide good default values



# HOW REACT WORKS BEHIND THE SCENES

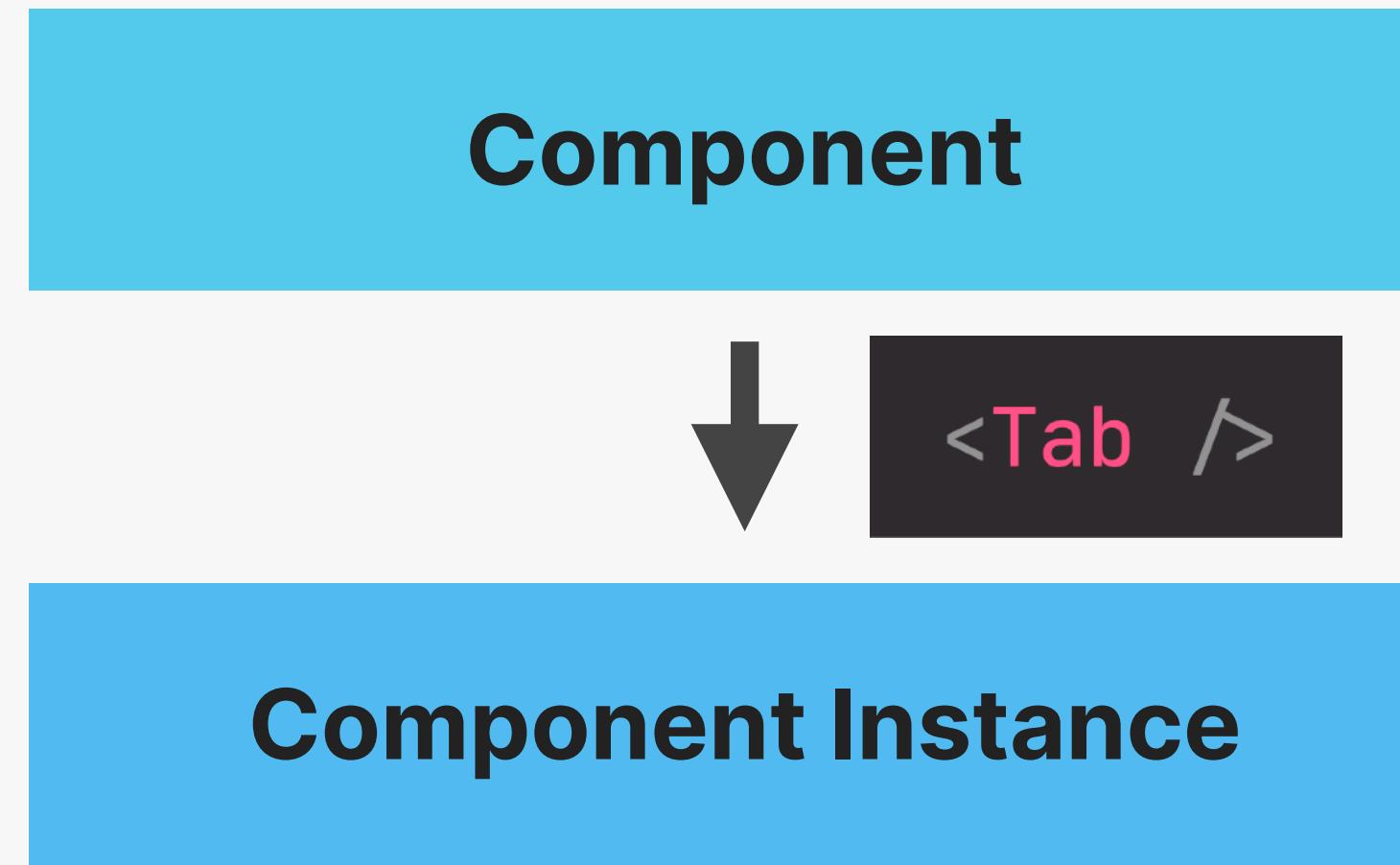
# COMPONENT VS. INSTANCE VS. ELEMENT

## Component

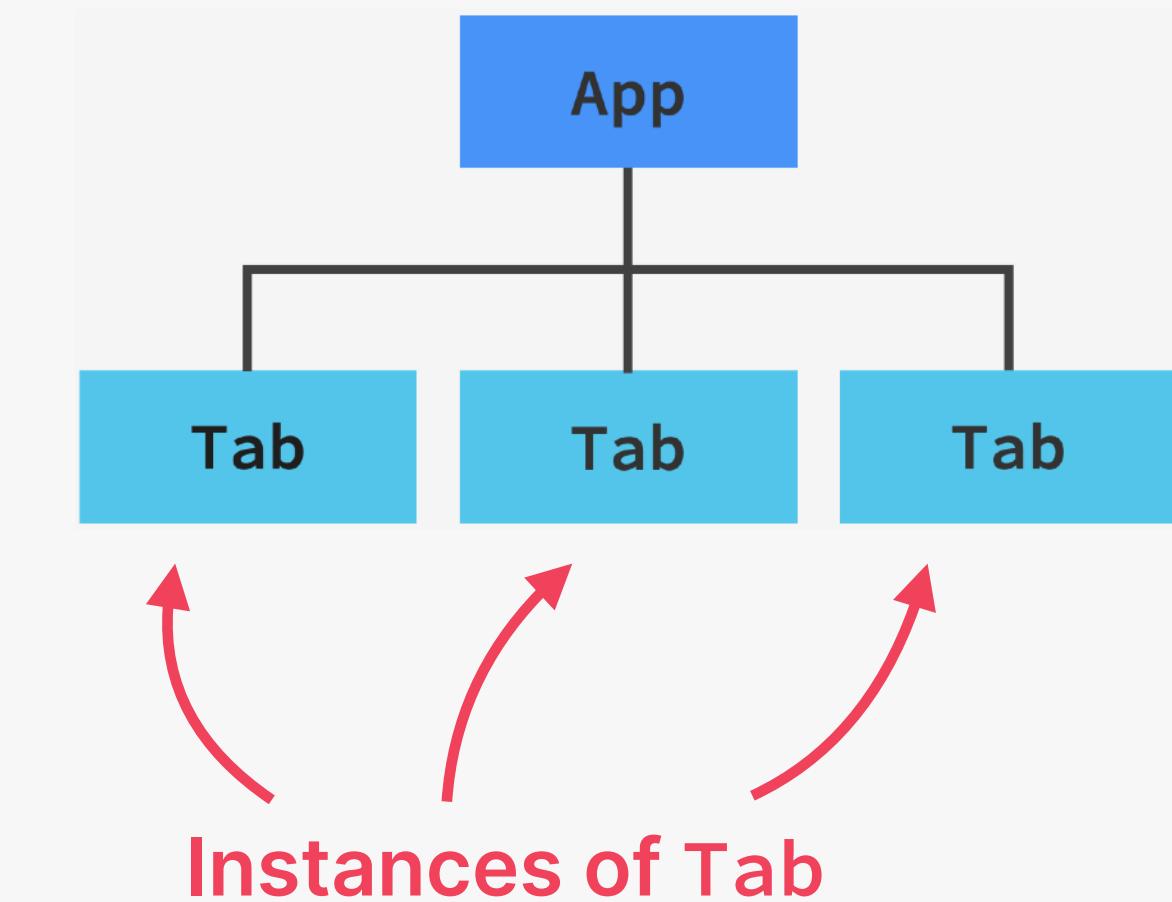
```
function Tab({ item }) {  
  return (  
    <div className='tab-content'>  
      <h4>All contacts</h4>  
      <p>Your post will be visible</p>  
    </div>  
  );  
}
```

- 👉 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

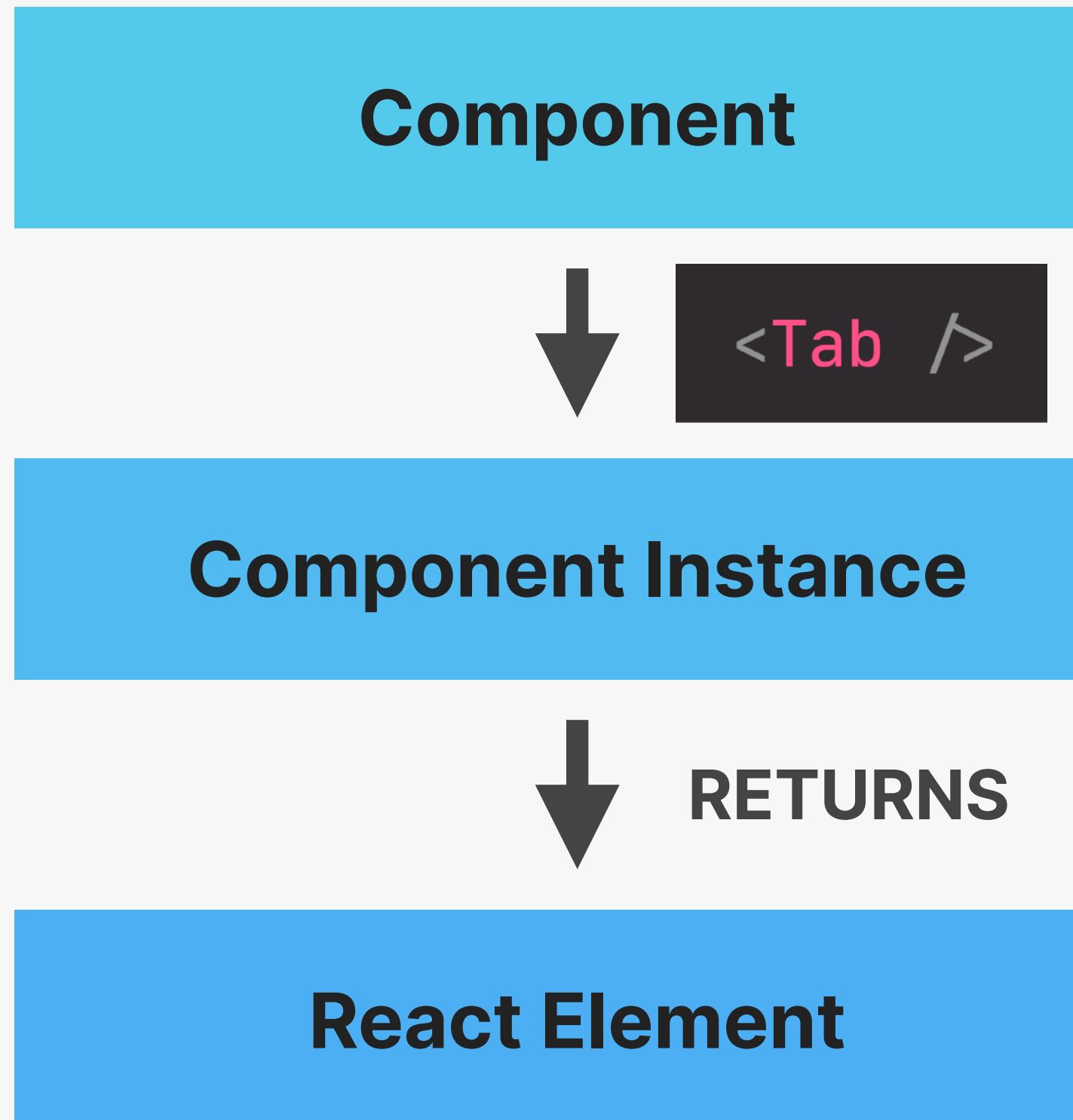


```
function App() {  
  return (  
    <div className='tabs'>  
      <Tab item={content[0]} />  
      <Tab item={content[1]} />  
      <Tab item={content[2]} />  
    </div>  
  );  
}
```



- 👉 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



A screenshot of a browser's developer tools element inspector. It shows a node with the following properties:

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

React.createElement(
 'div',
 {
 className: 'tab-content'
 },
 React.createElement('h4', null, 'All contacts'),
 React.createElement(
 'p',
 null,
 'Your post will be visible'
 )
);

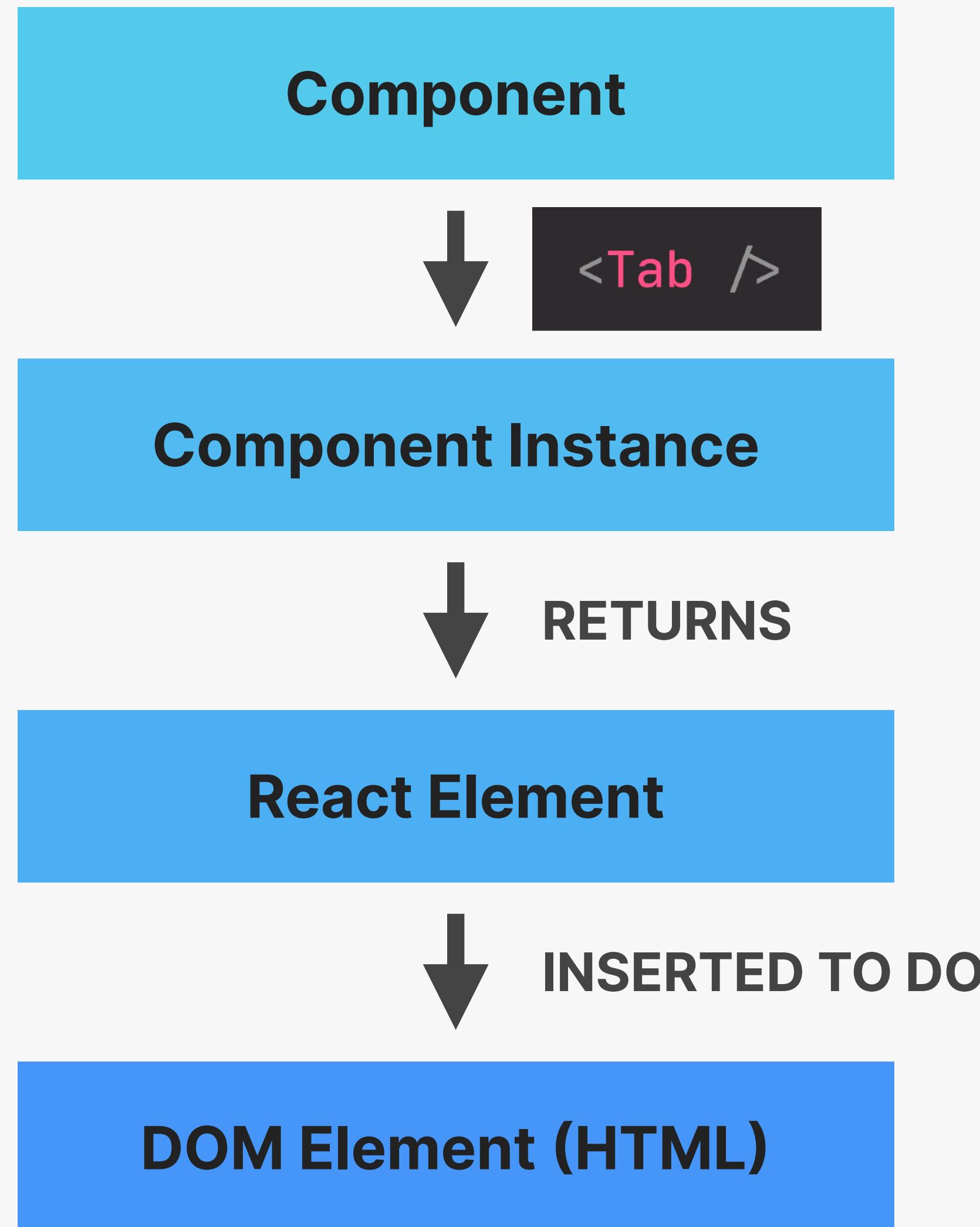
A screenshot of a browser's developer tools element inspector, showing a detailed view of a React Element object. The object has the following structure:

```
Object {$$typeof: Symbol(react.element), key: null, props: { children: Array(2), [Symbol(react.element)]: Object, length: 2, [Symbol(Symbol.iterator)]: Array(0), className: "tab-content", type: "div", _owner: null, _store: {validated: false} }}
```

**REACT 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



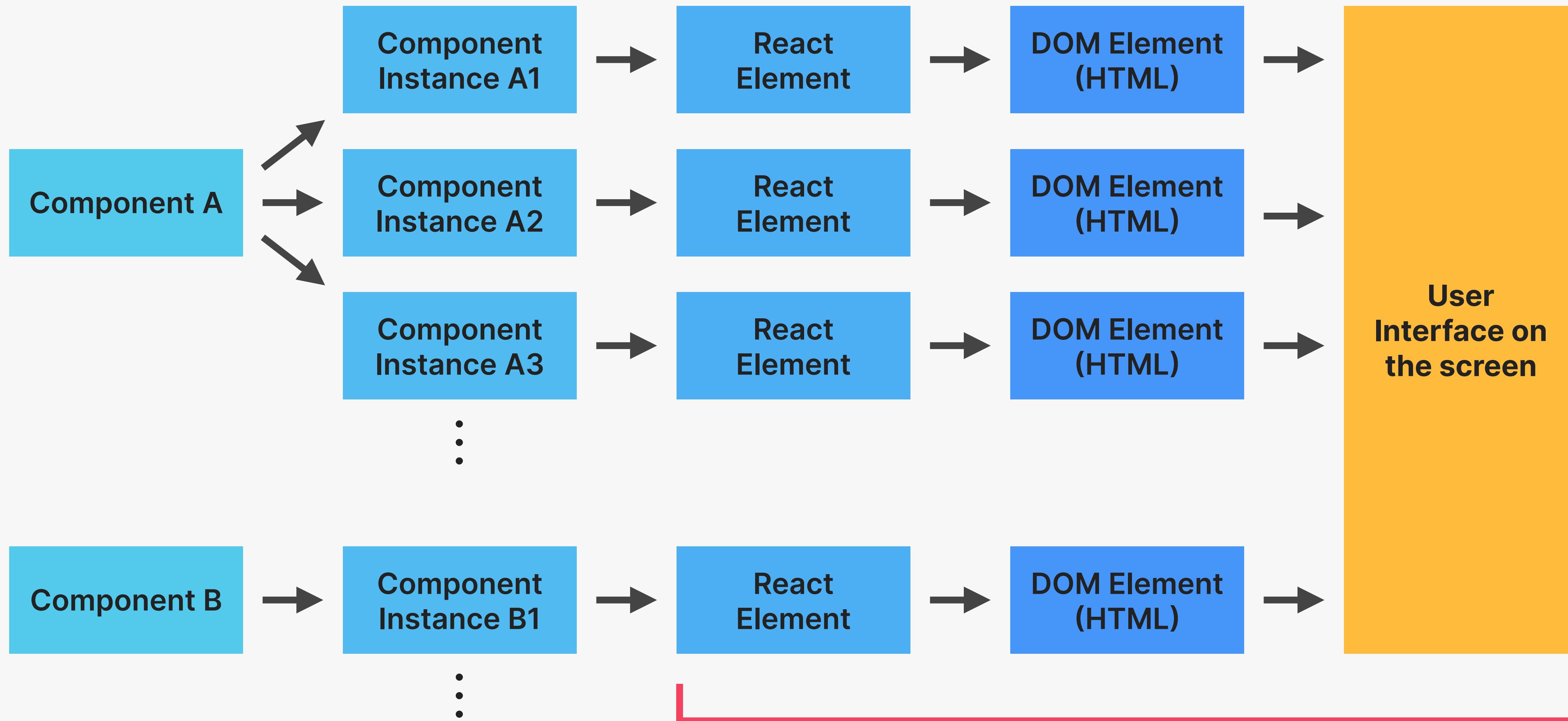
```
▼<div class="tab-content">
  <h4>All contacts</h4>
  <p>Your post will be visible</p>
</div>
```

All contacts  
Your post will be visible to all your contacts

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

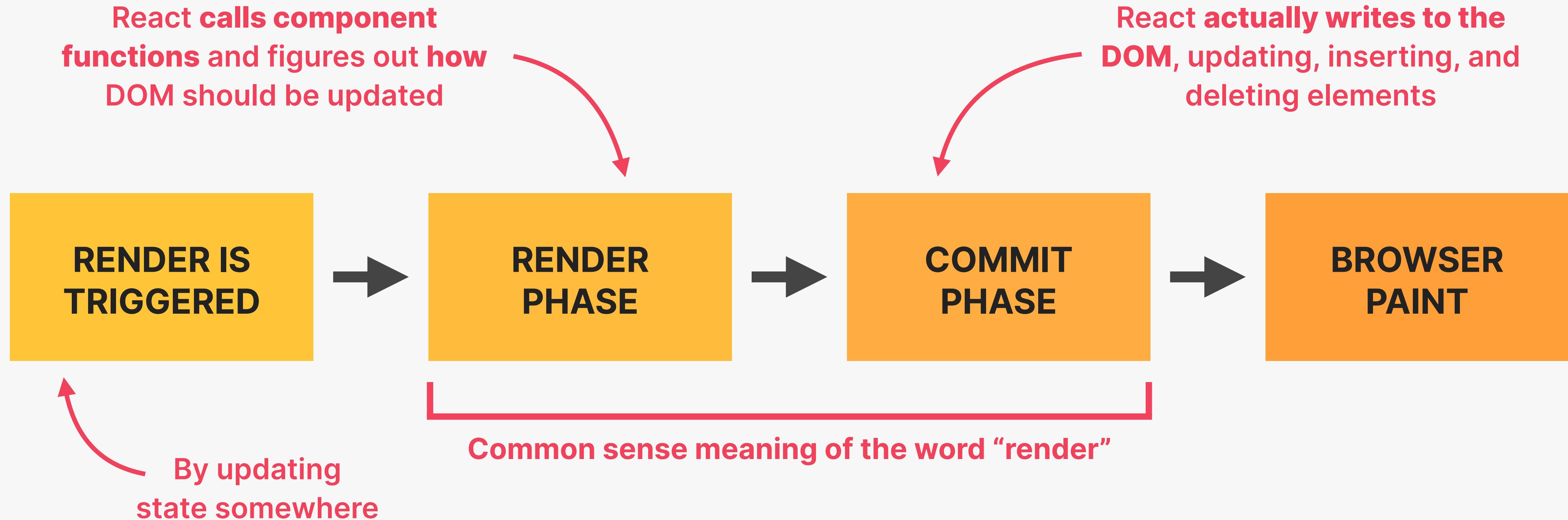


# QUICK RECAP BEFORE WE GET STARTED



How does this process *actually* work?

# 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**.

# HOW RENDERS ARE TRIGGERED

[1] RENDER IS TRIGGERED

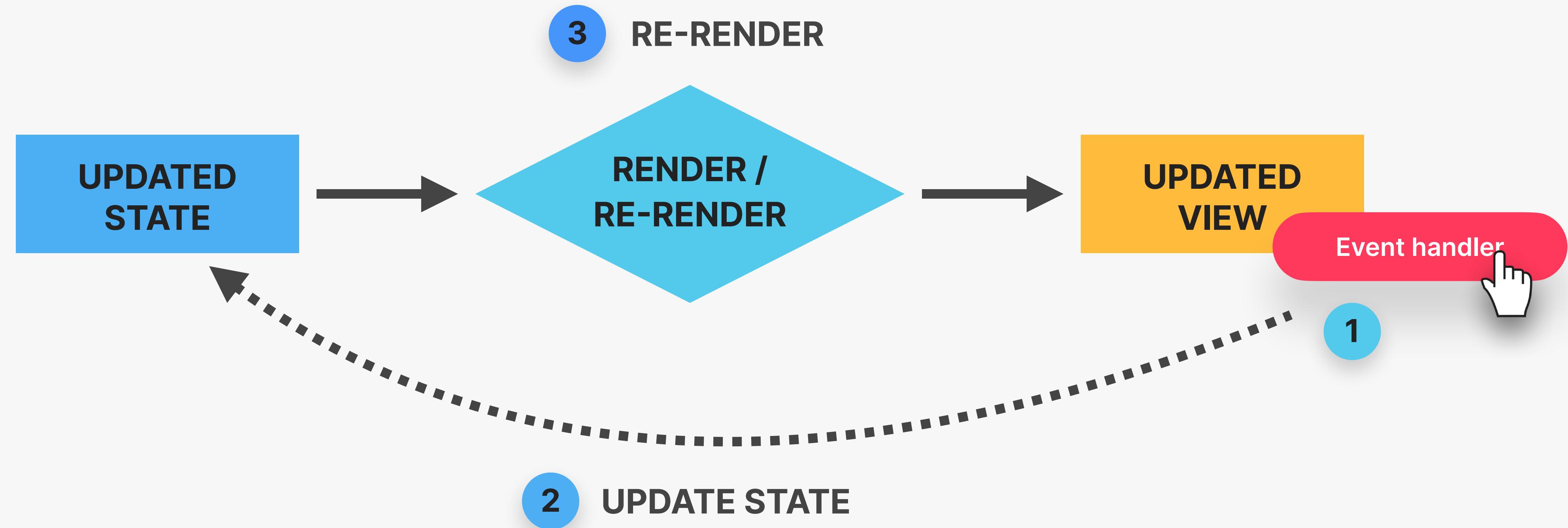
## THE TWO SITUATIONS THAT TRIGGER RENDERS:

- 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



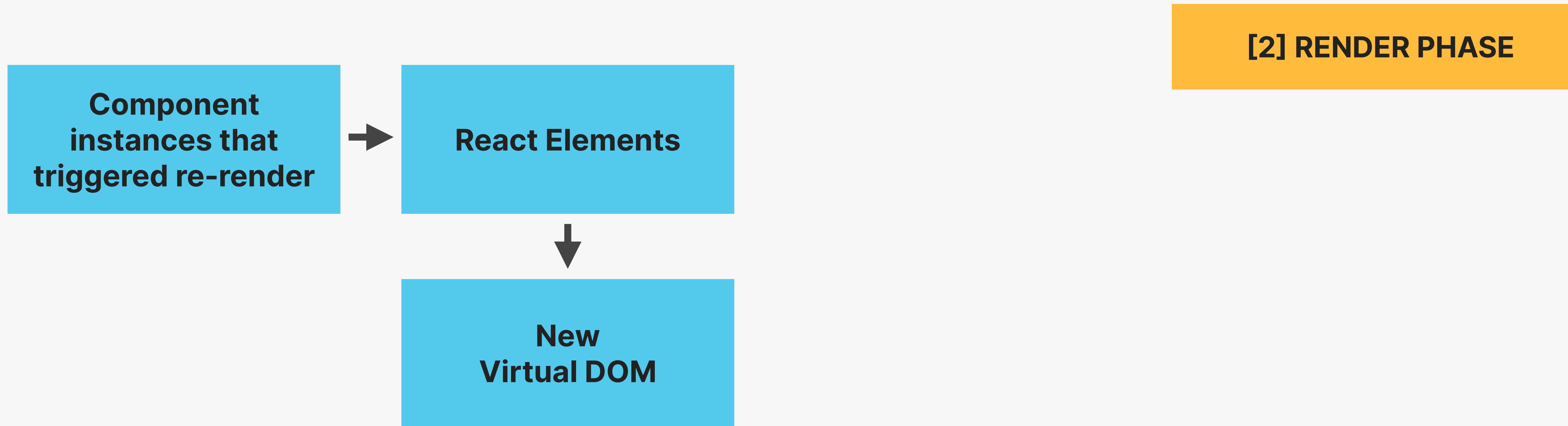
# REVIEW: THE MECHANICS OF STATE IN REACT



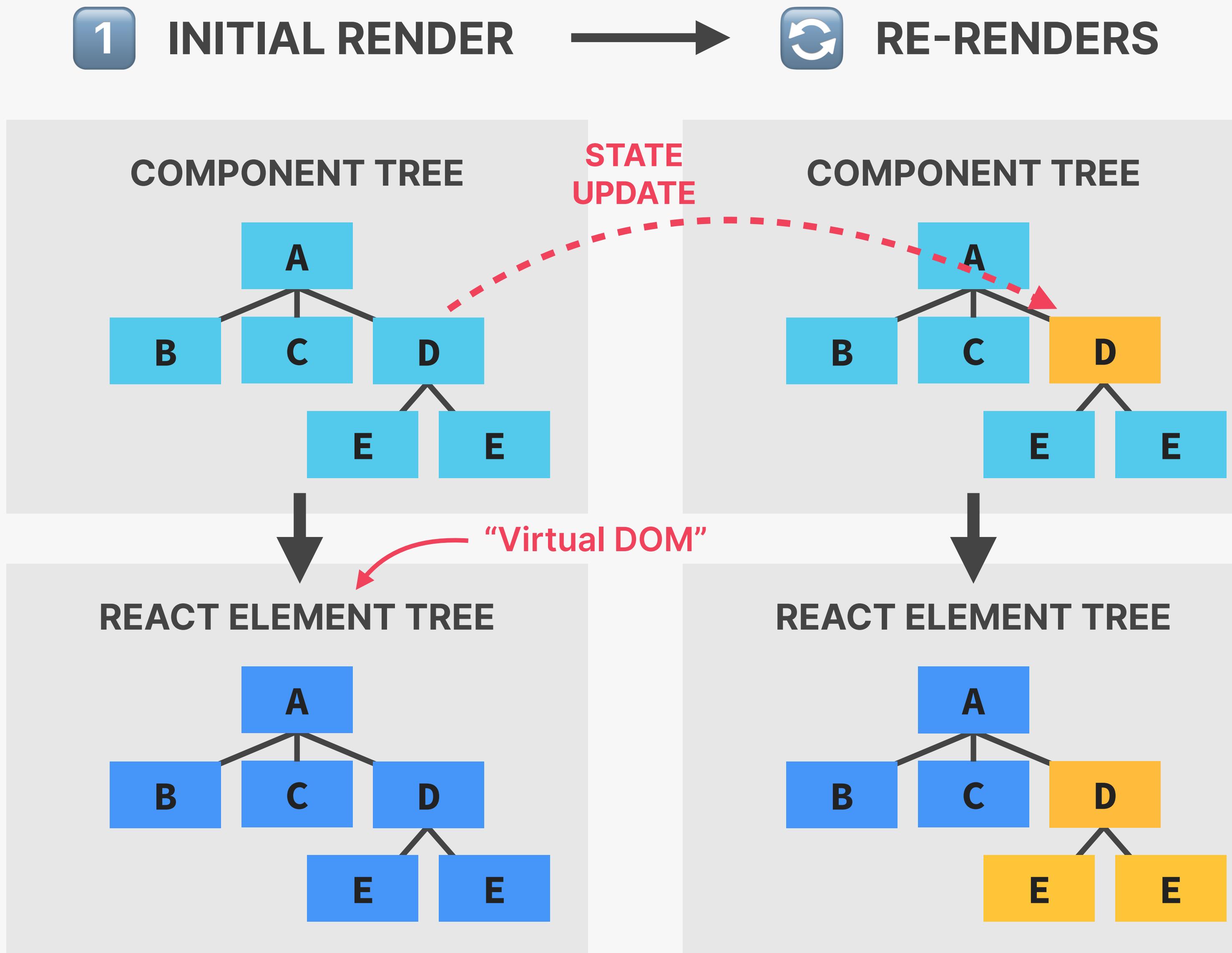
**NOT TRUE #1: RENDERING IS UPDATING THE SCREEN / DOM**

**NOT TRUE #2: REACT COMPLETELY DISCARDS OLD VIEW (DOM) ON RE-RENDER**

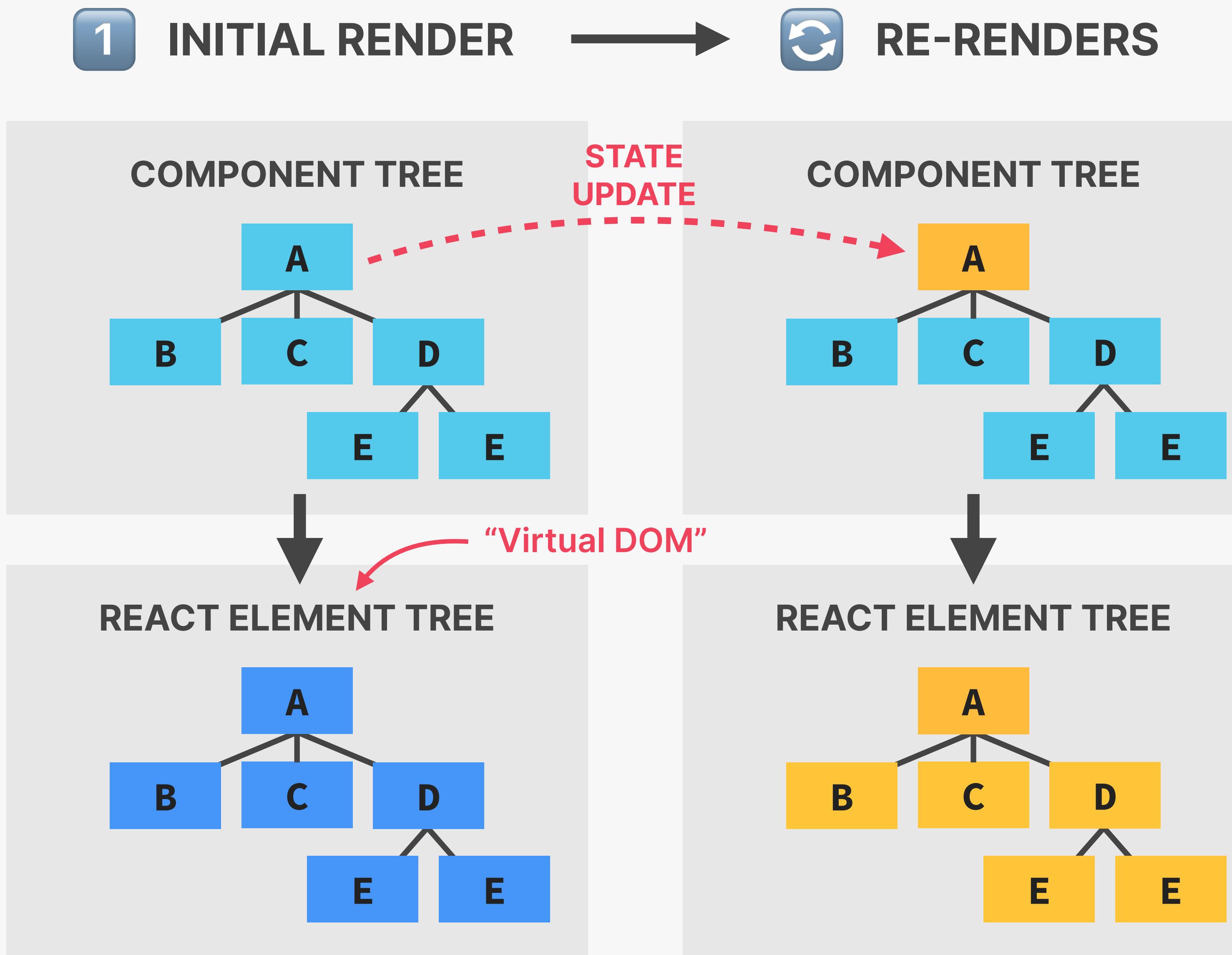
# THE RENDER PHASE



# THE VIRTUAL DOM (REACT ELEMENT TREE)

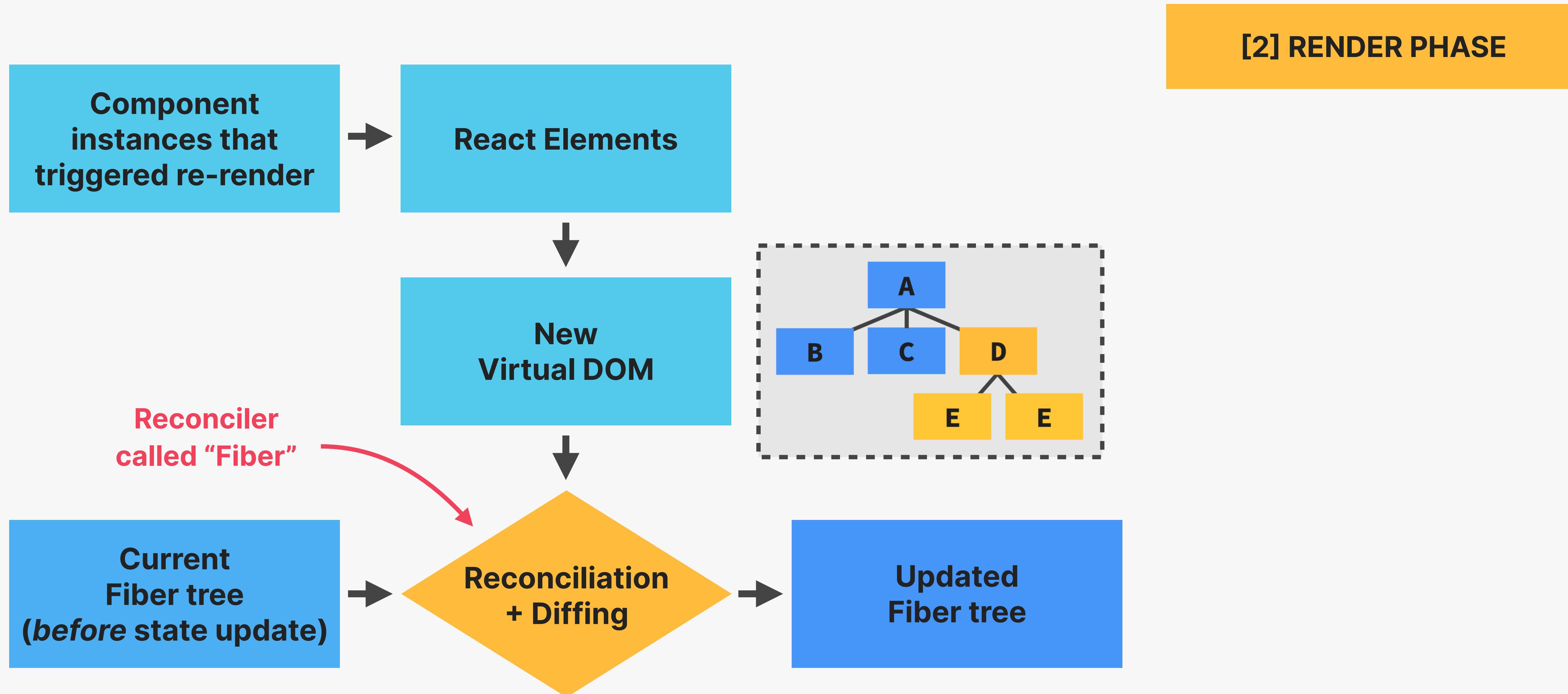


# THE VIRTUAL DOM (REACT ELEMENT TREE)



- 👉 **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

# THE RENDER PHASE



# WHAT IS RECONCILIATION AND WHY DO WE NEED IT?



Why not update the entire DOM whenever state changes somewhere in the app?



**BECAUSE**

👉 That would be inefficient and wasteful:

1

Writing to the DOM is (relatively) **slow**

2

Usually only a **small part of the DOM** needs to be updated

👉 React **reuses** as much of the existing DOM as possible



**HOW?**



**Reconciliation:** Deciding which DOM elements actually need to be inserted, deleted, or updated, in order to reflect the latest state changes

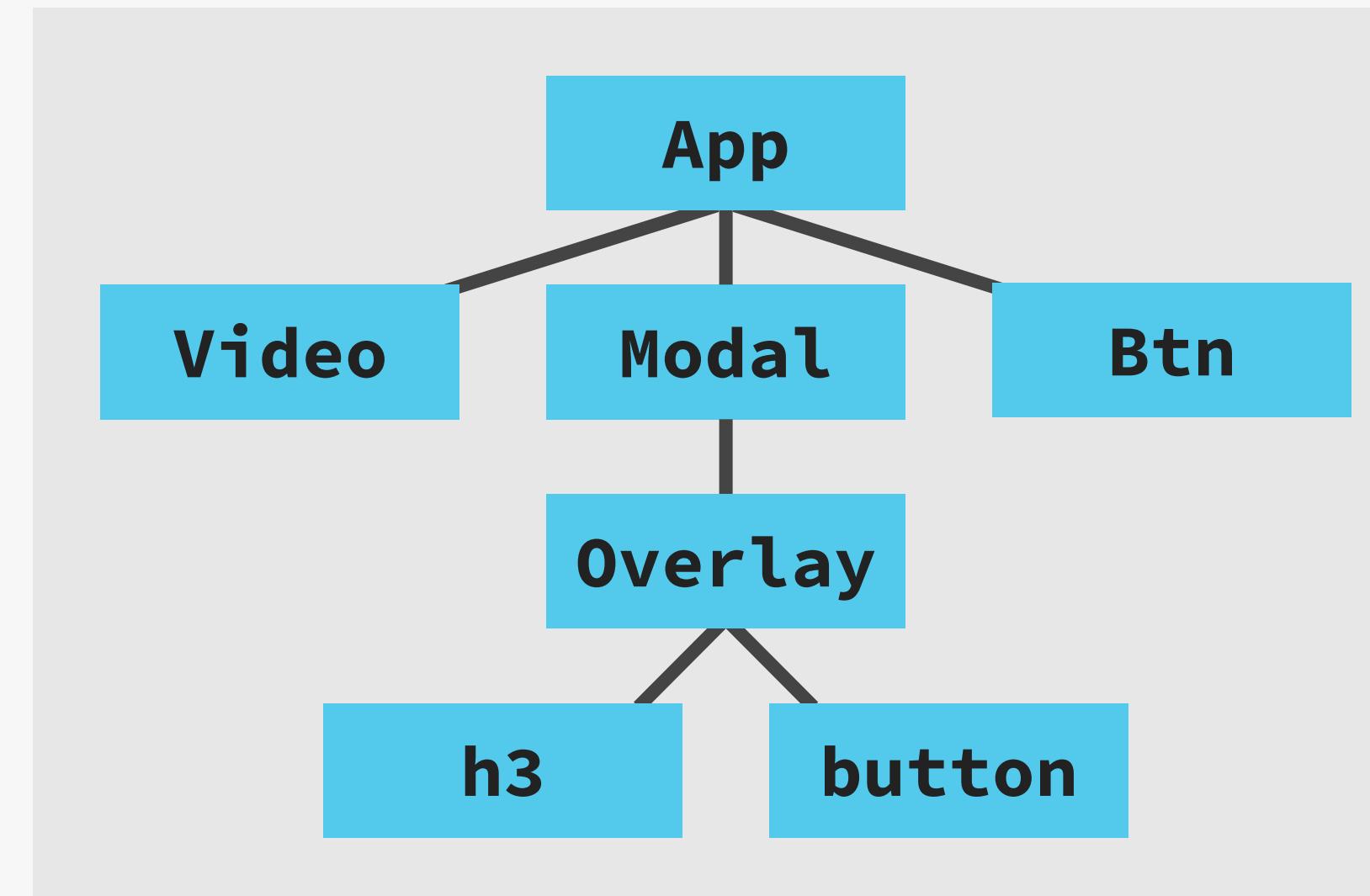
A screenshot of a Udemy course page titled 'Build Responsive Real-World Websites with HTML and CSS'. The sidebar on the right lists course content with titles like 'Working With Colors', 'Pseudo-classes', 'Styling Hyperlinks', etc., each with a duration indicator. A hand cursor is hovering over the 'Share' button at the top of the sidebar.

A screenshot of the same Udemy course page, but now a modal window is open in the center asking 'How would you rate this course?'. It has a 'Select Rating' section with five yellow stars. A red arrow points from the text 'Only these new DOM elements are created' down to the fourth star of the rating scale.

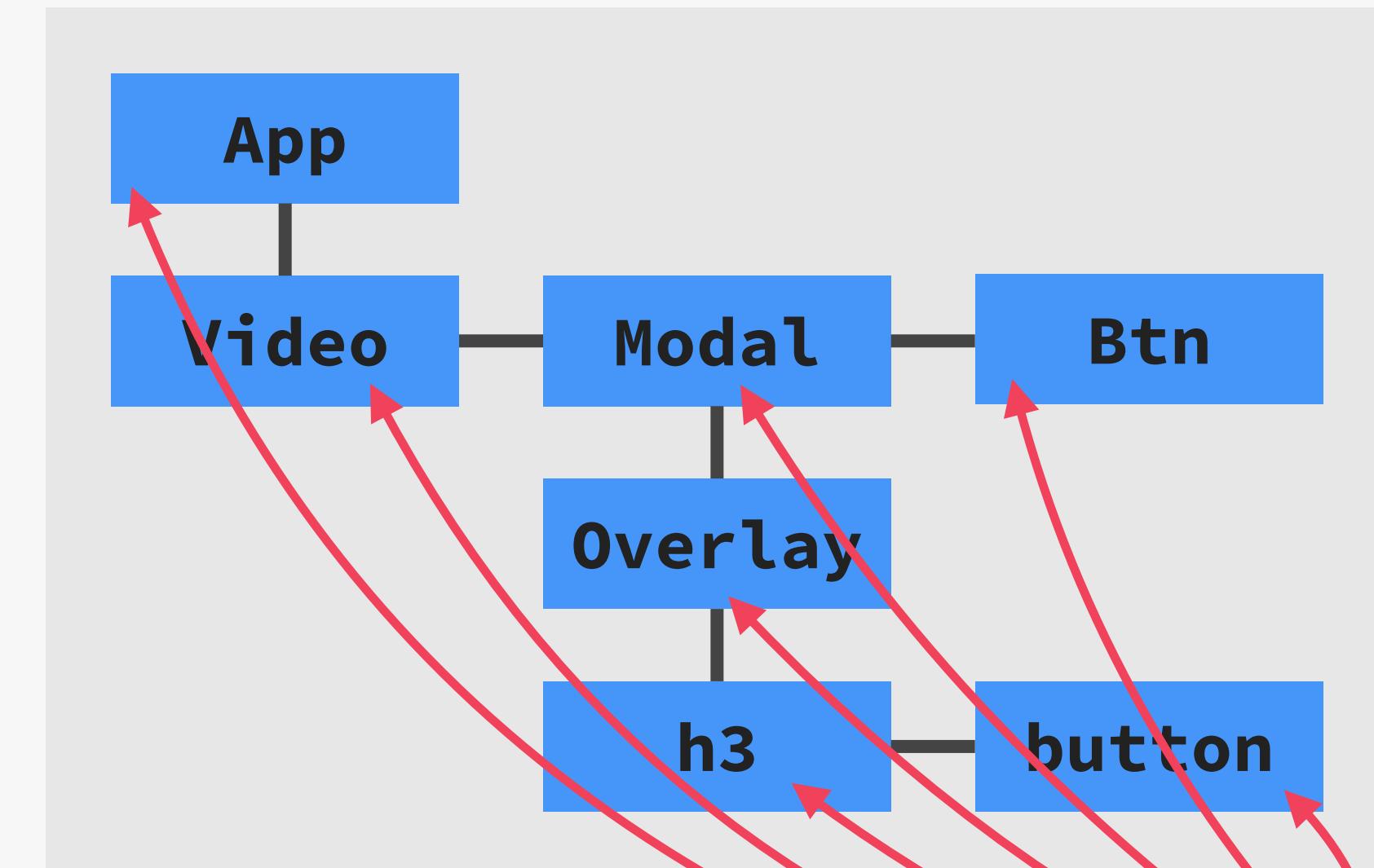
Only these new DOM elements are created

# THE RECONCILER: FIBER

REACT  
ELEMENT  
TREE  
(VIRTUAL  
DOM)



ON INITIAL  
RENDER



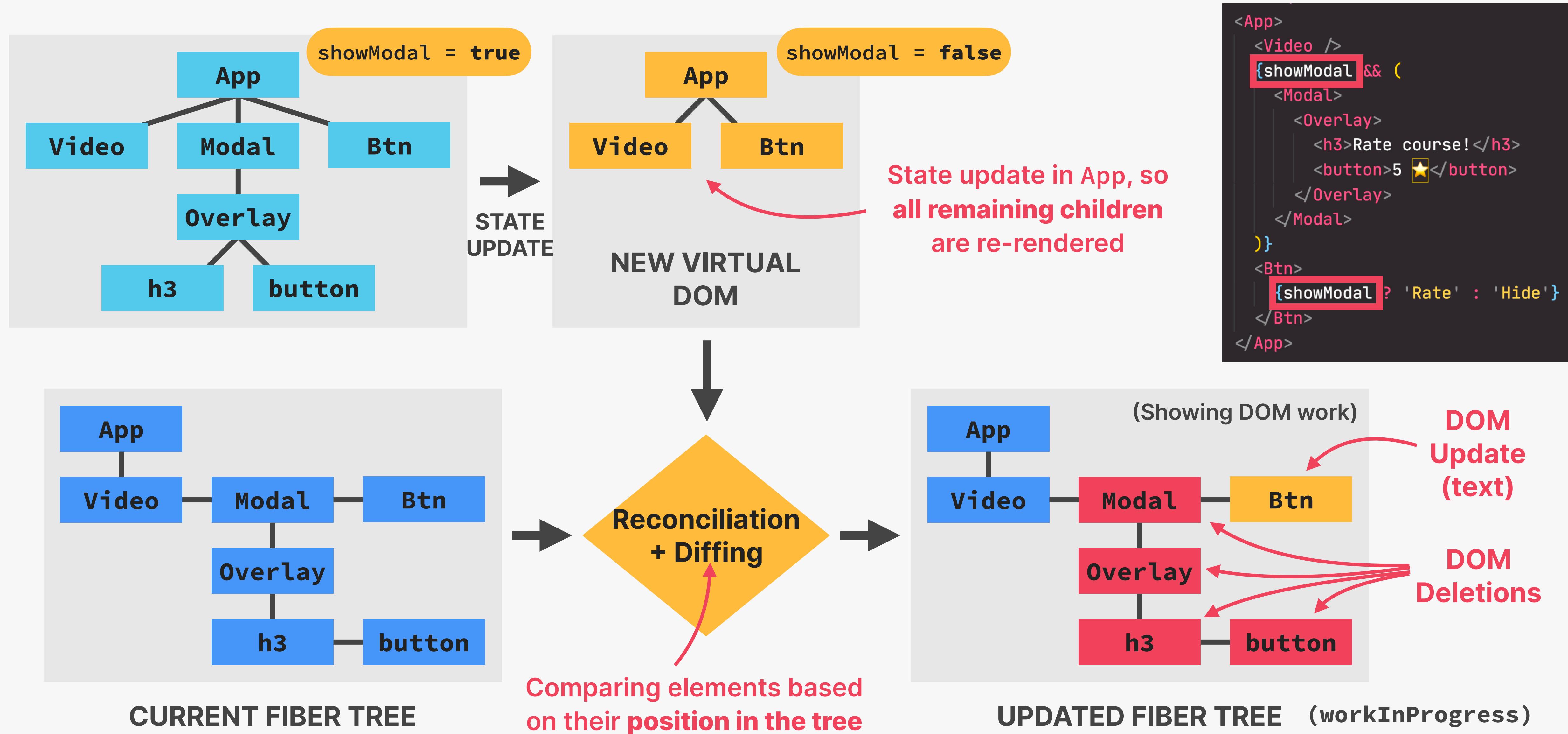
FIBER  
TREE

- 👉 Fiber tree: internal tree that has a “fiber” for each component instance and DOM element
- 👉 Fibers are **NOT** re-created on every render
- 👉 Work can be done **asynchronously**

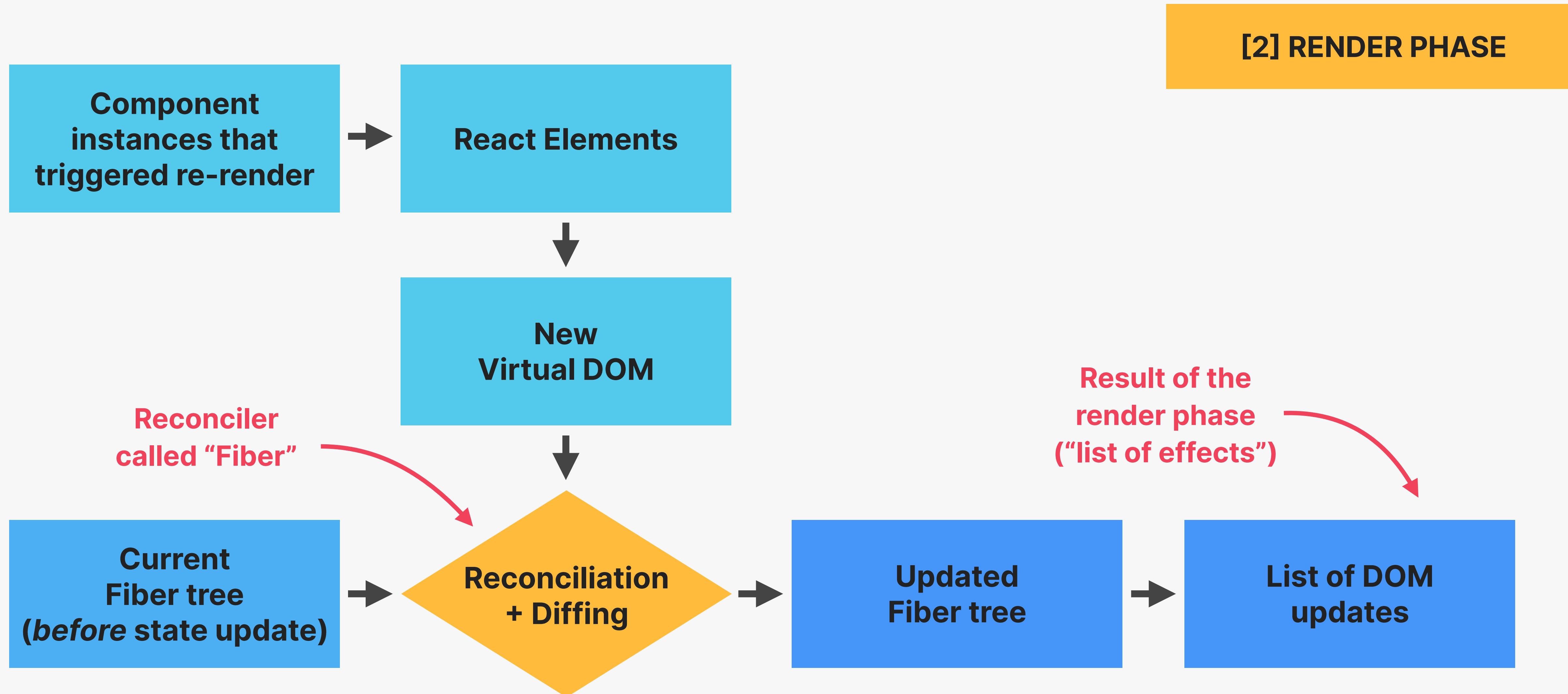
- 👉 Rendering process can be split into chunks, tasks can be prioritized, and work can be **paused, reused, or thrown away**
- 👉 Enables **concurrent features** like Suspense or transitions
- 👉 Long renders **won't block** JS engine

Current state
Props
Side effects
Used hooks
Queue of work

# RECONCILIATION IN ACTION

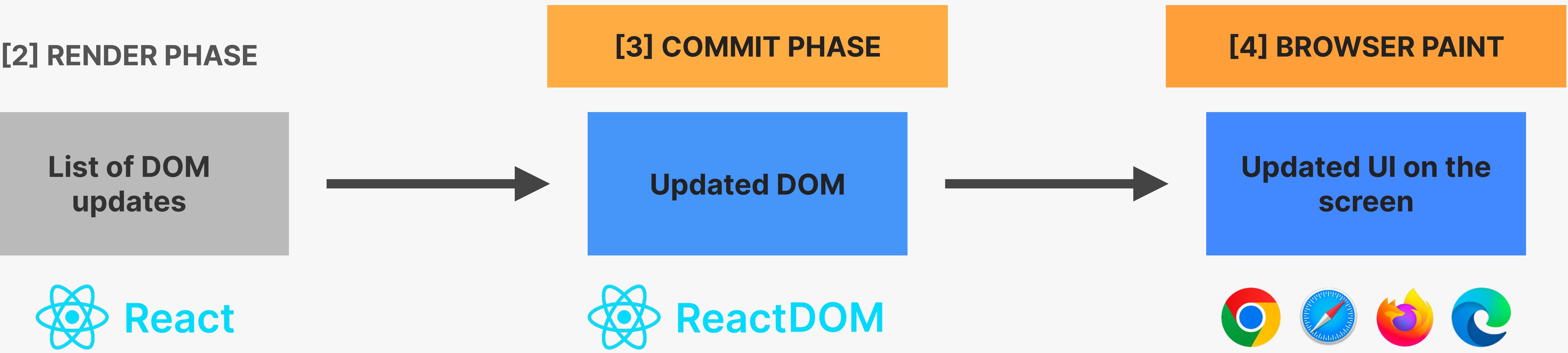


# THE RENDER PHASE



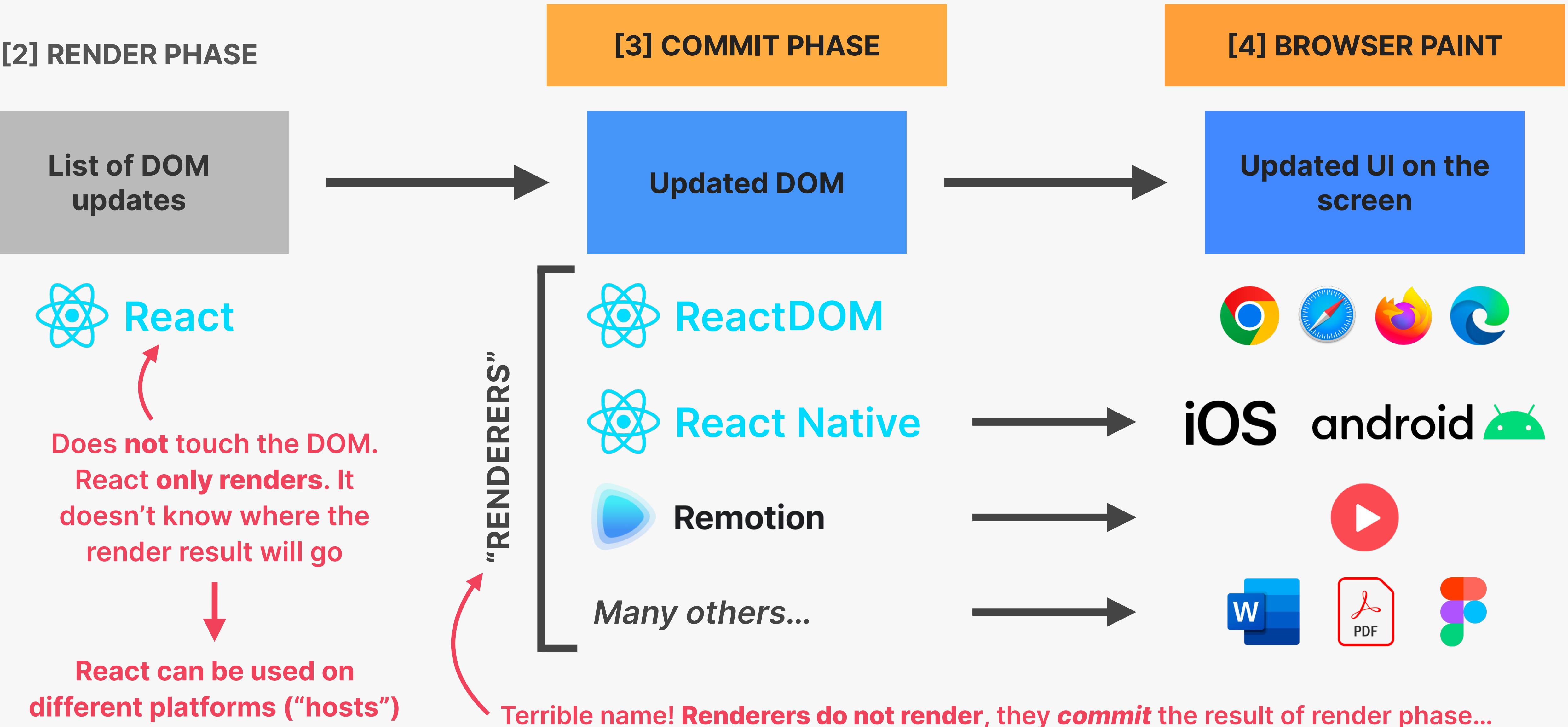


# THE COMMIT PHASE AND BROWSER PAINT

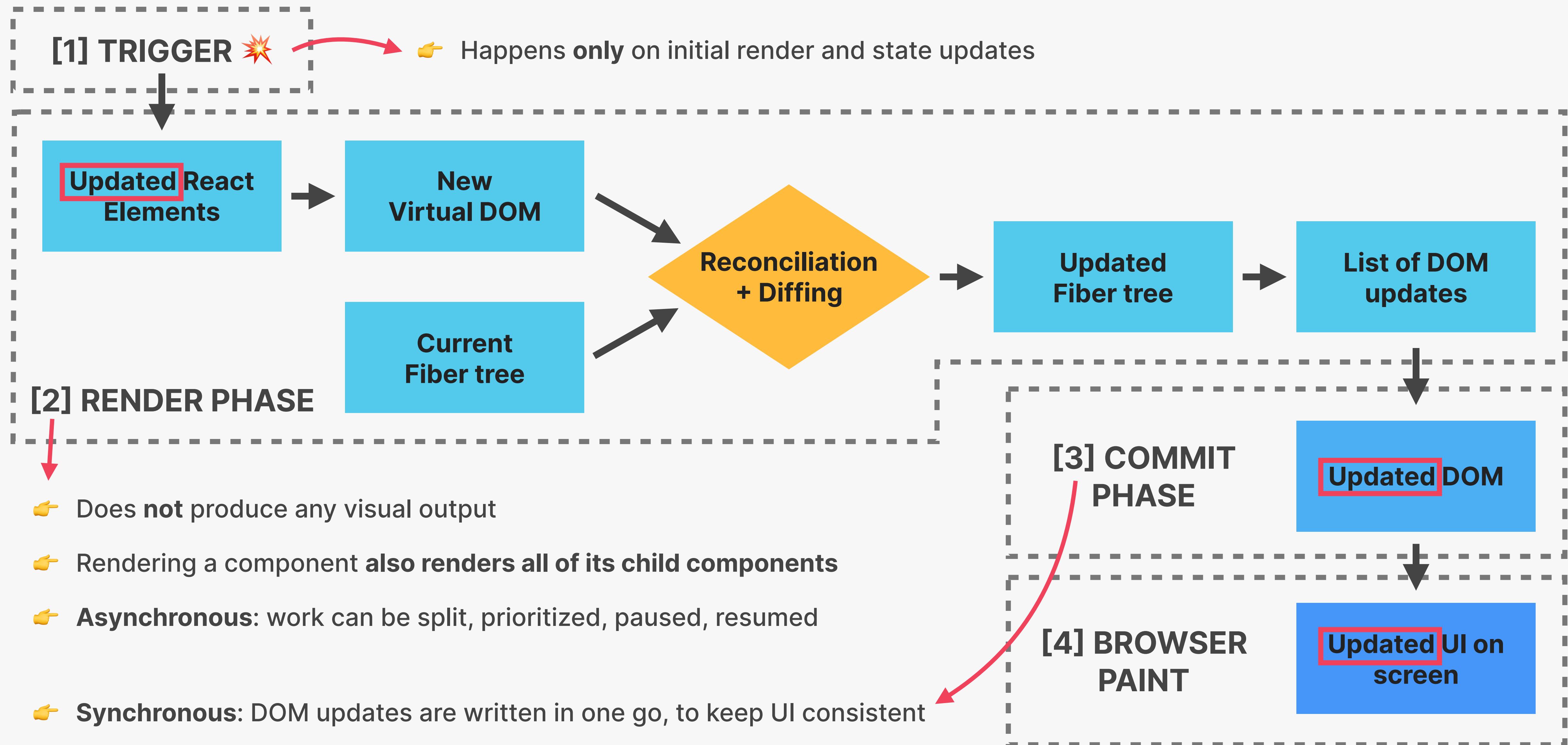


- 👉 **React writes to the DOM:** insertions, deletions, and updates (list of DOM updates are “flushed” to the DOM)
- 👉 **Committing is synchronous:** DOM is updated in one go, it can’t be interrupted. This is necessary so that the DOM never shows partial results, ensuring a consistent UI (in sync with state at all times)
- 👉 After the commit phase completes, the `workInProgress` fiber tree becomes the current tree for the next render cycle

# THE COMMIT PHASE AND BROWSER PAINT

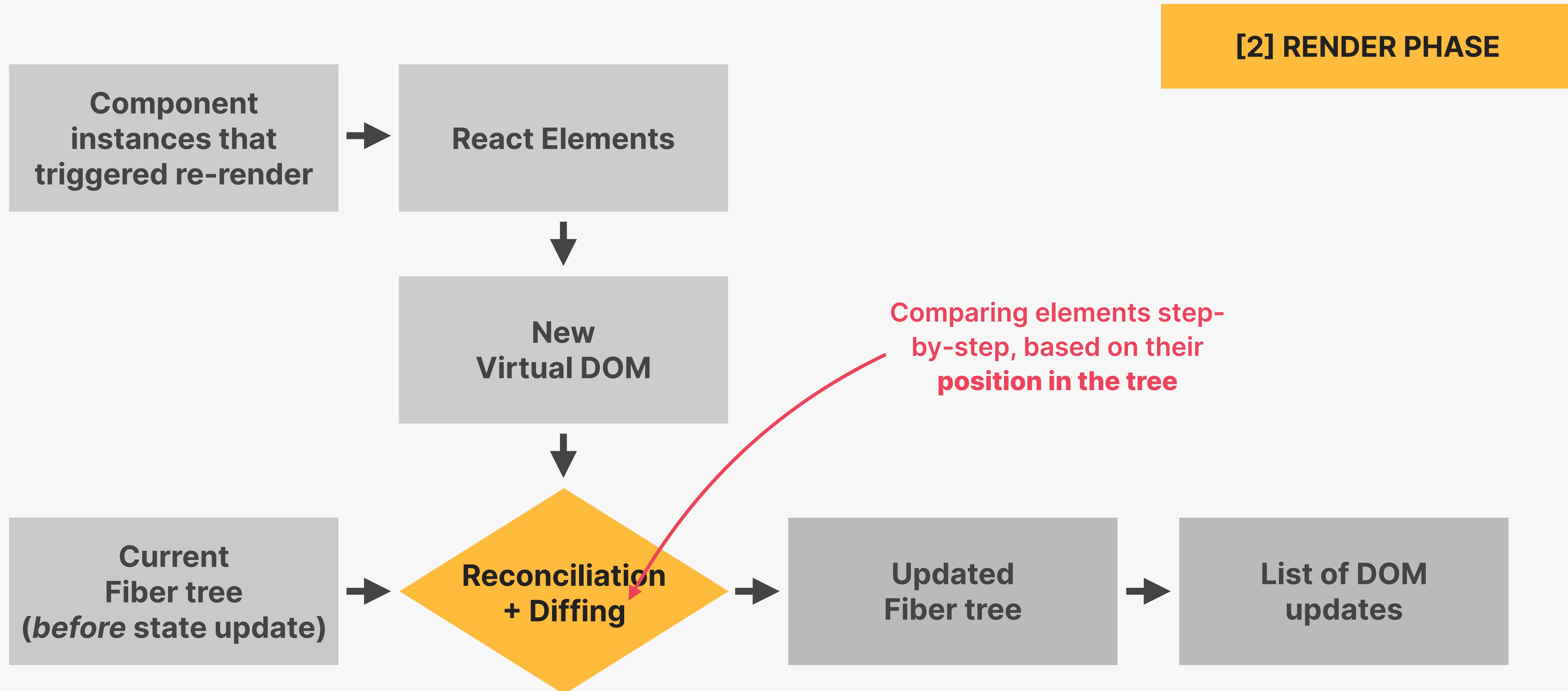


# RECAP: PUTTING IT ALL TOGETHER





# THE RENDER PHASE



# HOW DIFFING WORKS

- 👉 Diffing uses 2 fundamental assumptions (rules):

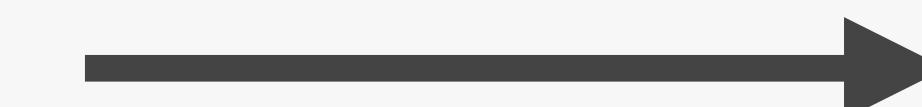
1 Two elements of different types will produce different trees

2 Elements with a stable key prop stay the same across renders

👉 This allows React to go from 1,000,000,000  $[O(n^3)]$  to 1000  $[O(n)]$  operations per 1000 elements

## 1. SAME POSITION, DIFFERENT ELEMENT

```
<div>  
| <SearchBar />  
</div>  
<main> ... </main>
```



Different DOM element

```
<header>  
| <SearchBar />  
</header>  
<main> ... </main>
```

```
<div>  
| <SearchBar />  
</div>  
<main> ... </main>
```



Different React element (component instance)

```
<div>  
| <ProfileMenu />  
</div>  
<main> ... </main>
```

👉 React assumes entire sub-tree is no longer valid

👉 Old components are destroyed and removed from DOM, including state

👉 Tree might be rebuilt if children stayed the same (state is reset)

# HOW DIFFING WORKS

👉 Diffing uses 2 fundamental assumptions (rules):

1 Two elements of different types will produce different trees

2 Elements with a stable key prop stay the same across renders

👉 This allows React to go from 1,000,000,000  $[O(n^3)]$  to 1000  $[O(n)]$  operations per 1000 elements

## 2. SAME POSITION, SAME ELEMENT

```
<div className="hidden">  
| <SearchBar />  
</div>  
<main> ... </main>
```



Same DOM element

```
<div className="active">  
| <SearchBar />  
</div>  
<main> ... </main>
```

```
<div>  
| <SearchBar wait={1} />  
</div>  
<main> ... </main>
```



Same React element  
(component instance)

```
<div>  
| <SearchBar wait={5} />  
</div>  
<main> ... </main>
```

- 👉 Element will be kept (as well as child elements), **including state**
- 👉 **New props / attributes** are passed if they changed between renders
- 👉 Sometimes this is **not** what we want... Then we can use the key prop



# WHAT IS THE KEY PROP?

## KEY PROP

- 👉 Special prop that we use to tell the diffing algorithm that an element is **unique**
- 👉 Allows React to **distinguish** between multiple instances of the same component type
- 👉 When a key **stays the same across renders**, the element will be kept in the DOM  
*(even if the position in the tree changes)*

### 1 Using keys in lists

- 👉 When a key **changes between renders**, the element will be destroyed and a new one will be created *(even if the position in the tree is the same as before)*

### 2 Using keys to reset state

# 1. KEYS IN LISTS [STABLE KEY]



NO KEYS

```
<ul>
  <Question question={q[1]} />
  <Question question={q[2]} />
</ul>
```



ADDING NEW LIST ITEM

```
<ul>
  <Question question={q[0]} />
  <Question question={q[1]} />
  <Question question={q[2]} />
</ul>
```



👉 Same elements, but different position in tree, so they are removed and recreated in the DOM (*bad for performance*)



WITH KEYS

```
<ul>
  <Question key='q1' question={q[1]} />
  <Question key='q2' question={q[2]} />
</ul>
```



ADDING NEW LIST ITEM

```
<ul>
  <Question key='q0' question={q[0]} />
  <Question key='q1' question={q[1]} />
  <Question key='q2' question={q[2]} />
</ul>
```



👉 Different position in the tree, but the key stays the same, so the elements will be kept in the DOM   👉 **Always use keys!**

## 2. KEY PROP TO RESET STATE [CHANGING KEY]

👉 If we have the same element at the same position in the tree, the **DOM element and state** will be kept

```
<QuestionBox>
  <Question
    question={{
      title: 'React vs JS',
      body: 'Why should we use React?',
    }}
    key="q23"
  />
</QuestionBox>
```



**NEW QUESTION IN  
SAME POSITION**

```
<QuestionBox>
  <Question
    question={{
      title: 'Best course ever :D',
      body: 'This is THE React course!',
    }}
  />
</QuestionBox>
```

**Question state (answer):**

React allows us to build apps faster |

**State was  
preserved. NOT  
what we want**

**Question state (answer):**

React allows us to build apps faster |

## 2. KEY PROP TO RESET STATE [CHANGING KEY]

👍 WITH KEY

👉 If we have the same element at the same position in the tree, the **DOM element and state will be kept**

```
<QuestionBox>
  <Question
    question={{
      title: 'React vs JS',
      body: 'Why should we use React?',
    }}
    key="q23"
  />
</QuestionBox>
```

NEW QUESTION IN  
SAME POSITION

```
<QuestionBox>
  <Question
    question={{
      title: 'Best course ever :D',
      body: 'This is THE React course!',
    }}
    key="q89"
  />
</QuestionBox>
```

Question state (answer):

React allows us to build apps faster |

State was  
RESET

Question state (answer):





# THE TWO TYPES OF LOGIC IN REACT COMPONENTS

## 1. RENDER LOGIC

- 👉 Code that lives at the **top level** of the component function
- 👉 Participates in **describing** how the component view looks like
- 👉 Executed **every time** the component renders

## 2. EVENT HANDLER FUNCTIONS

- 👉 Executed as a **consequence of the event** that the handler is listening for (change event in this example)
- 👉 Code that actually **does things**: update state, perform an HTTP request, read an input field, navigate to another page, etc.

```
function Question({ question }) {  
  const [newAnswer, setNewAnswer] = useState('');  
  const numAnswers = question.answers.length ?? 0;  
  
  const handleNewAnswer = function (e) {  
    if (question.closed) return;  
    setNewAnswer(e.target.value);  
  };  
  
  const createList = function () {  
    return (  
      <ul>  
        {question.answers.map((q) => (  
          <li>{q}</li>  
        ))}  
      </ul>  
    );  
  };  
  
  return (  
    <div>  
      <h3>{question.title}</h3>  
      <p>{question.body}</p>  
      {question.hasAnswer ? (  
        createList()  
      ) : (  
        <input  
          value={newAnswer}  
          onChange={handleNewAnswer}  
        />  
      )}  
    </div>  
  );  
}
```

# REFRESHER: FUNCTIONAL PROGRAMMING PRINCIPLES

- 👉 **Side effect:** dependency on or modification of any data outside the function scope. “*Interaction with the outside world*”. Examples: mutating external variables, HTTP requests, writing to DOM.

👋 Side effects are not bad! A program can only be useful if it has some interaction with the outside world

Side effect: Outside variable mutation

- 👉 **Pure function:** a function that has **no side effects**.

- 👉 Does **not** change any variables outside its scope
- 👉 Given the **same input**, a pure function always returns the **same output**

Unpredictable output (date changes)

## ✓ Pure function

```
function circleArea(r) {  
  return 3.14 * r * r;  
}
```

## 👉 Impure function

```
const areas = {};  
  
function circleArea(r) {  
  areas.circle = 3.14 * r * r;  
}
```

## 👉 Impure function

```
function circleArea(r) {  
  const date = Date.now();  
  const area = 3.14 * r * r;  
  return `${date}: ${area}`;  
}
```

# RULES FOR RENDER LOGIC

- 👉 **Components must be pure when it comes to render logic:** given the same props (input), a component instance should always return the same JSX (output)
- 👉 **Render logic must produce no side effects:** no interaction with the “outside world” is allowed. So, in render logic:
  - 👉 Do NOT perform **network requests** (API calls)
  - 👉 Do NOT start **timers**
  - 👉 Do NOT directly **use the DOM API**
  - 👉 Do NOT **mutate objects or variables** outside of the function scope
  - 👉 Do NOT **update state (or refs)**: this will create an infinite loop!

This is why we can't  
mutate props!

👉 Side effects are allowed (and encouraged) in **event handler functions!**  
There is also a special hook to **register side effects** (`useEffect`)



# HOW STATE UPDATES ARE BATCHED

- 👉 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

```
const [answer, setAnswer] = useState('');
const [best, setBest] = useState(true);
const [solved, setSolved] = useState(false);

const reset = function () {
  setAnswer('');
  console.log(answer);
  setBest(true);
  setSolved(false);
};

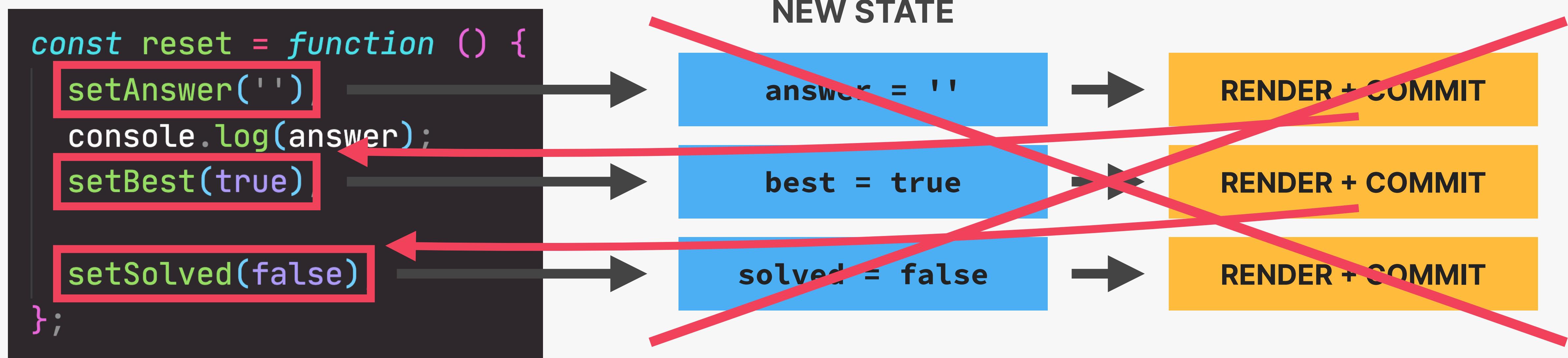
return (
  <div>
    <button onClick={reset}>Reset</button>
    {/* ... */}
  </div>
);
```

**Event  
handler  
function**

A red arrow points from the text "Event handler function" to the `onClick={reset}` prop of the `<button>` element in the JSX code.

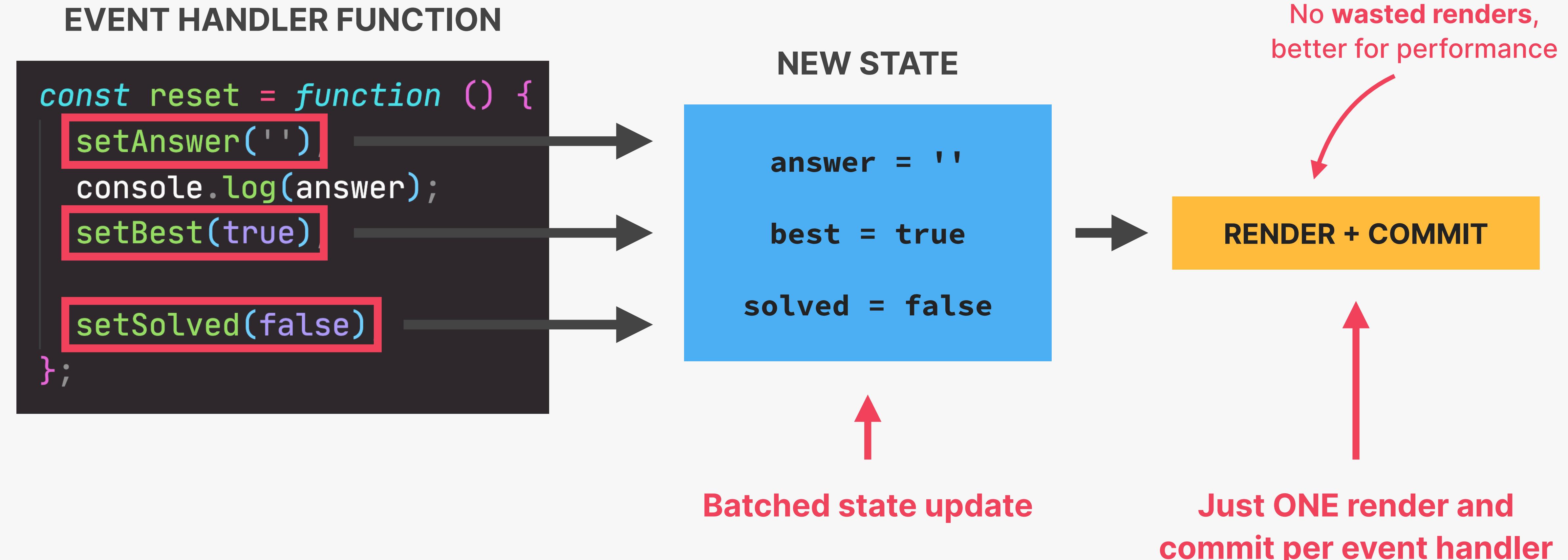
# HOW STATE UPDATES ARE BATCHED

## EVENT HANDLER FUNCTION



This is NOT how React updates  
multiple pieces of state in the  
same event handler

# HOW STATE UPDATES ARE BATCHED



# UPDATING STATE IS ASYNCHRONOUS

## EVENT HANDLER FUNCTION

```
const reset = function () {  
  setAnswer('')  
  console.log(answer)  
  setBest(true)  
  
  setSolved(false)  
};
```



What will the value of answer be at this point?

State is stored in the Fiber tree during render phase

At this point, re-render has not happened yet

Therefore, answer still contains current state, not the updated state ('')

“Stale state”

- 👉 Updated state variables are **not** immediately available after setState call, but only after the re-render
- 👉 This also applies when **only one** state variable is updated
- 👉 If we need to update state based on previous update, we use setState with callback (setAnswer(answer=>...))

**UPDATING STATE IN REACT IS ASYNCHRONOUS**

# BATCHING BEYOND EVENT HANDLER FUNCTIONS

- 👉 We can **opt out** of automatic batching by wrapping a state update in `ReactDOM.flushSync()` (*but you will never need this*)

```
const reset = function () {  
  setAnswer('');  
  console.log(answer);  
  setBest(true);  
  
  setSolved(false);  
};
```

We now get automatic batching at all times, everywhere

👉 **AUTOMATIC BATCHING IN...**

**REACT 17**

**REACT 18+**

## EVENT HANDLERS

```
<button onClick={reset}>Reset</button>
```



## TIMEOUTS

```
setTimeout(reset, 1000);
```



## PROMISES

```
fetchStuff().then(reset);
```



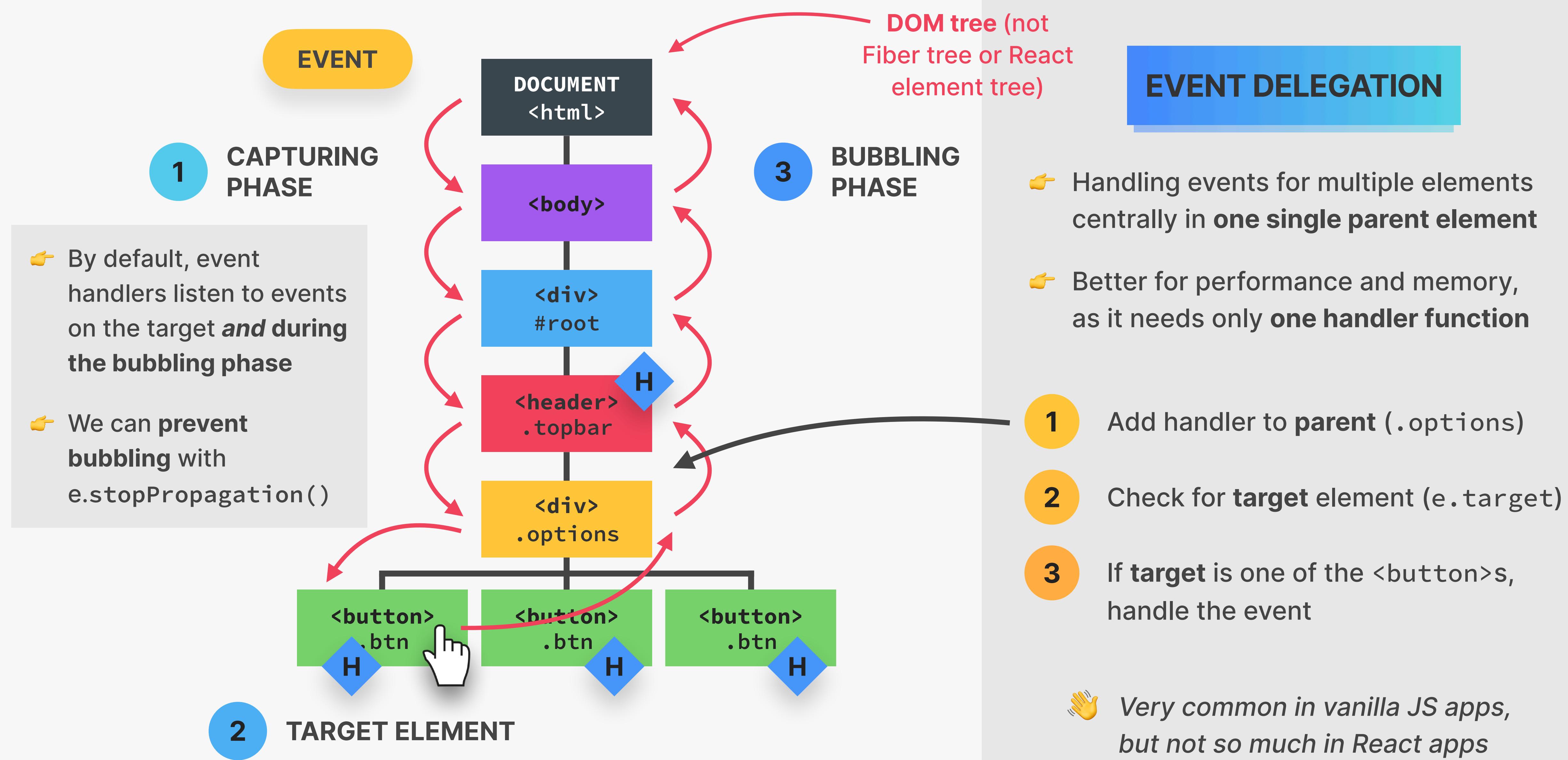
## NATIVE EVENTS

```
el.addEventListener('click', reset);
```

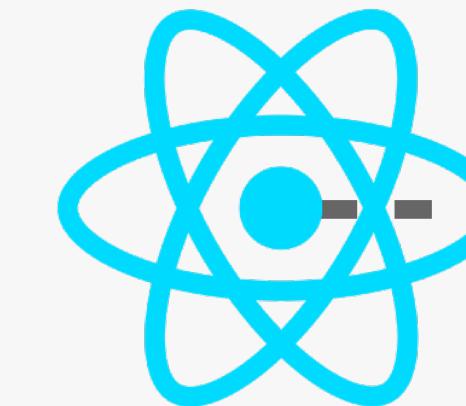




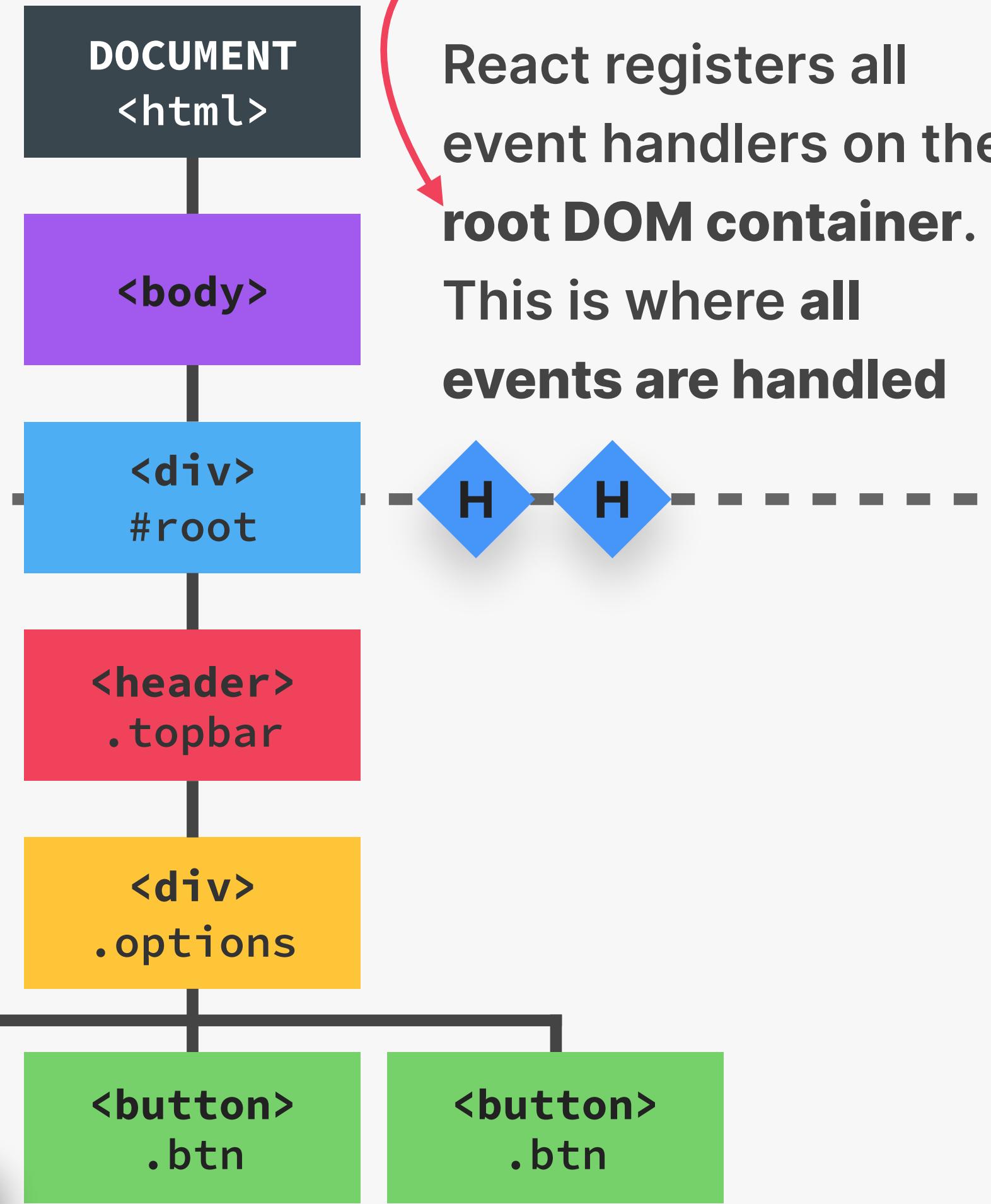
# DOM REFRESHER: EVENT PROPAGATION AND DELEGATION



# HOW REACT HANDLES EVENTS



INSIDE REACT  
FIBER TREE



Usually div#root, but  
can be any DOM element

React registers all  
event handlers on the  
**root DOM container**.  
This is where **all**  
**events are handled**

WHEN WE ATTACH AN EVENT HANDLER...

```
<button  
  className="btn"  
  onClick={() => setLoading(true)}  
/>
```

~~document  
.querySelector('.btn')  
.addEventListener(  
'click',  
(() => setLoading(true))  
)~~

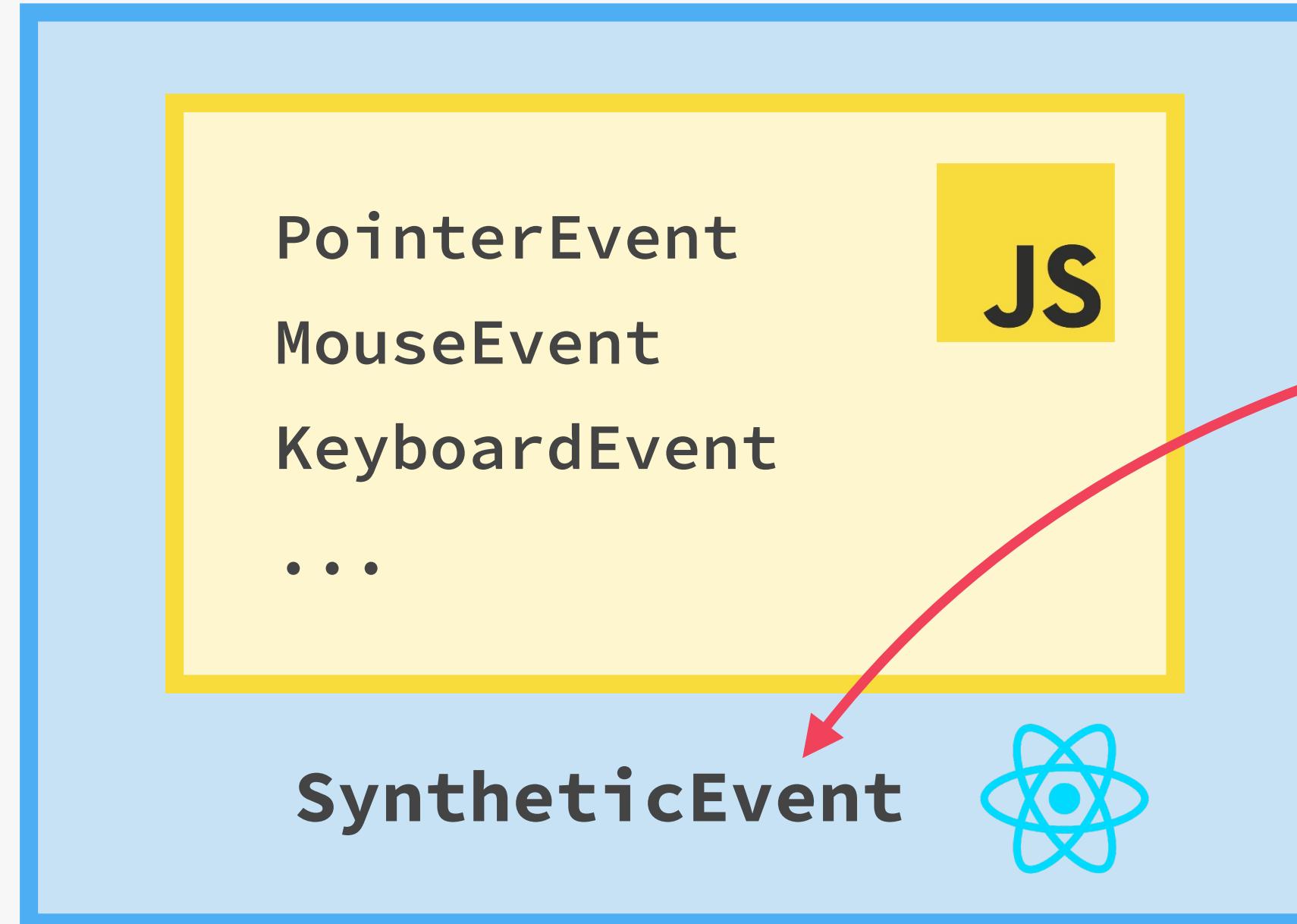
🚫 ... **WHAT APPEARS  
TO BE HAPPENING**

document  
.querySelector('#root')  
.addEventListener(  
'click',  
(() => setLoading(true))  
)

✓ ... **WHAT ACTUALLY  
HAPPENS INTERNALLY**

👉 Behind the scenes, React performs **event delegation** for all events in our applications

# SYNTHETIC EVENTS



```
<input onChange={e => setText(e.target.value)} />
```

- 👉 Wrapper around the DOM's native event object
- 👉 Has **same interface** as native event objects, like `stopPropagation()` and `preventDefault()`
- 👉 Fixes browser inconsistencies, so that events work in the exact **same way in all browsers**
- 👉 **Most synthetic events bubble** (including focus, blur, and change), except for scroll

## EVENT

- 👉 Attributes for event handlers are named using **camelCase** (`onClick` instead of `onclick` or `click`)

## HANDLERS IN



- 👉 Default behavior can **not** be prevented by returning `false` (only by using `preventDefault()`)

- 👉 Attach “Capture” if you need to handle during **capture phase** (example: `onClickCapture`)



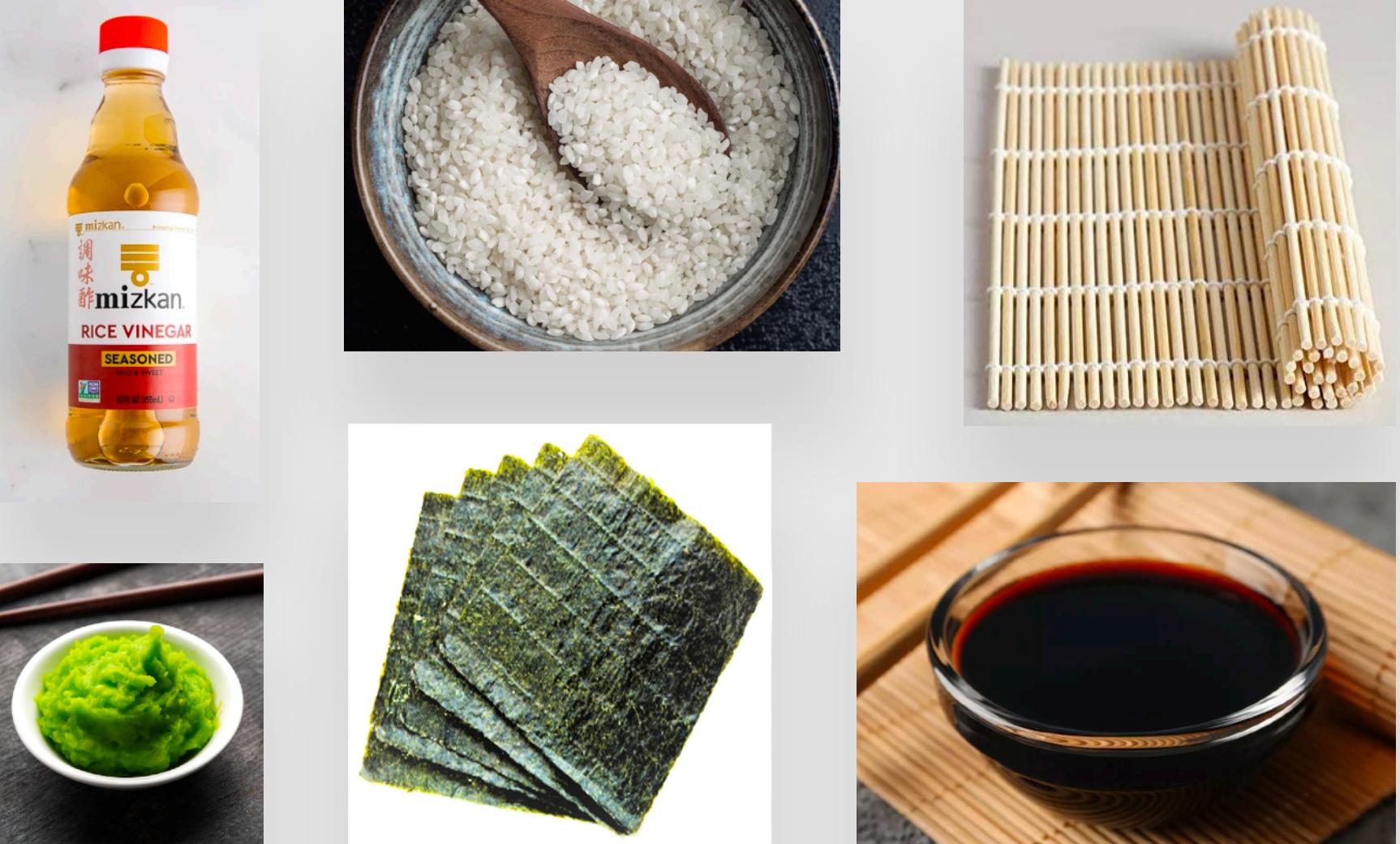
# FIRST, AN ANALOGY



## ALL-IN-ONE KIT



## SEPARATE INGREDIENTS



👍 **Ease of mind:** All ingredients are included

👎 **No choice:** You're stuck with the kit's ingredients

👍 **Freedom:** You can choose the best ingredients

👎 **Decision fatigue:** You need to research and buy all ingredients separately

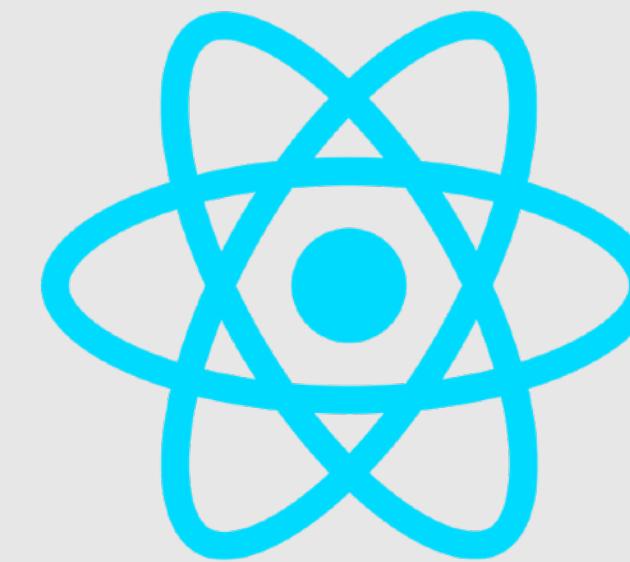
# FIRST, AN ANALOGY



## ALL-IN-ONE KIT



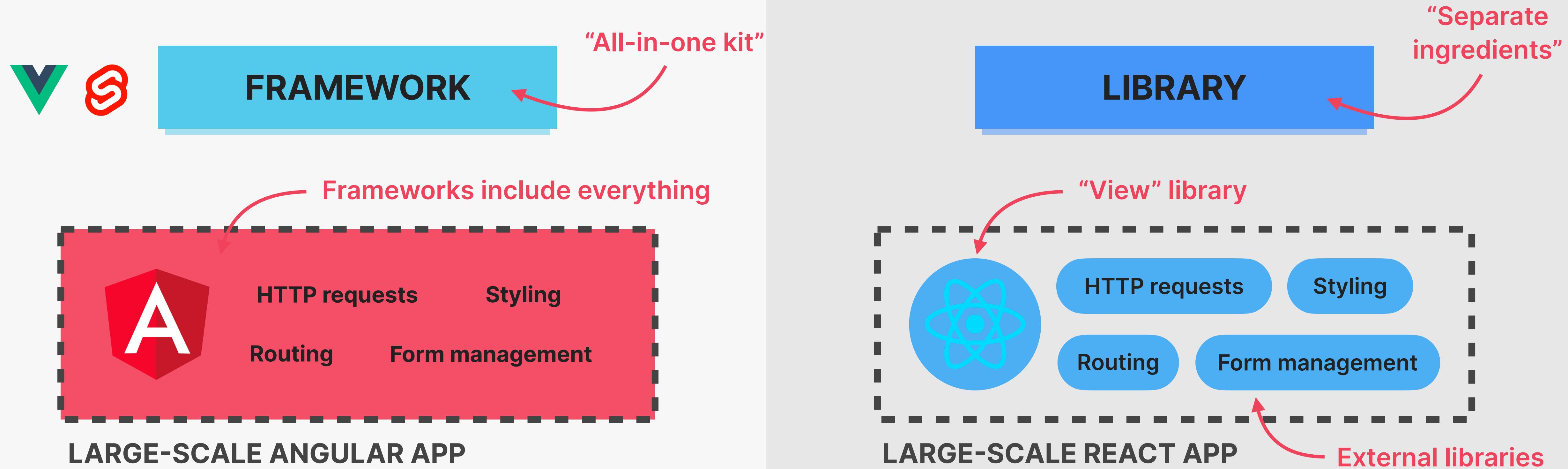
## SEPARATE INGREDIENTS



- 👍 **Ease of mind:** All ingredients are included
- 👎 **No choice:** You're stuck with the kit's ingredients

- 👍 **Freedom:** You can choose the best ingredients
- 👎 **Decision fatigue:** You need to research and buy all ingredients separately

# FRAMEWORK VS. LIBRARY



- 👍 **Ease of mind:** Everything you need to build a complete application **is included** in the framework ("batteries included")
- 👎 **No choice:** You're stuck with the framework's tools and conventions (which is not always bad!)

- 👍 **Freedom:** You can (or *need to*) **choose multiple 3rd-party libraries** to build a complete application
- 👎 **Decision fatigue:** You need to **research, download, learn, and stay up-to-date** with multiple external libraries

# REACT 3RD-PARTY LIBRARY ECOSYSTEM

1 Routing (for SPAs)



**React Router**



**React Location**

👉 Library options  
for different React  
application needs

2 HTTP requests



**JS fetch()**

**A X I O S**

3 Remote state management

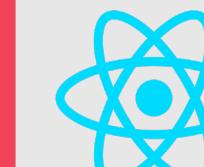


**React Query**

**S W R**

**A P O L L O**

4 Global state management



**Context API**



**Redux**



**Zustand**

5 Styling



**CSS  
MODULES**



**< >  
styled  
components**



**tailwindcss**

6 Form management



**React Hook Form**



**FORMIK**

7 Animations/transitions



**Motion**



**react-spring**

8 UI components



**chakra**



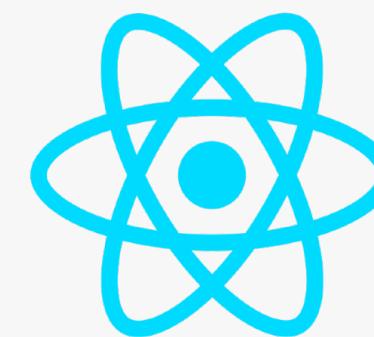
**Mantine**

# FRAMEWORKS BUILT ON TOP OF REACT

NEXT.js

Remix

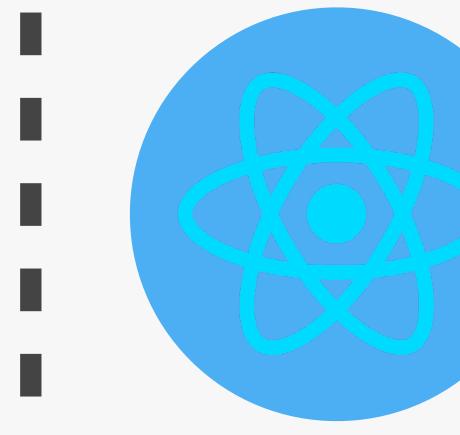
Gatsby



Full-stack frameworks!

“Opinionated”  
React frameworks

“VANILLA” REACT APP



HTTP requests

Styling

Routing

Form management

NEXT.js Remix

External libraries

Included out of the box

HTTP requests

Styling

Routing

Form management

👉 React frameworks offer many other features: server-side rendering (SSR), static site generation (SSG), better developer experience (DX), etc.





# PRACTICAL SUMMARY



A **component** is like a blueprint for a piece of UI that will eventually exist on the screen. When we “use” a component, React creates a **component instance**, which is like an actual physical manifestation of a component, containing props, state, and more. A component instance, when rendered, will return a **React element**

<Question />

*function Question()*



“Rendering” only means **calling component functions** and calculating what DOM elements need to be inserted, deleted, or updated. It has nothing to do with writing to the DOM. Therefore, **each time a component instance is rendered and re-rendered, the function is called again**



Only the **initial app render** and **state updates** can cause a render, which happens for the **entire application**, not just one single component



When a component instance gets re-rendered, **all its children will get re-rendered as well**. This doesn’t mean that all children will get updated in the DOM, thanks to reconciliation, which checks which elements have actually changed between two renders. But all this re-rendering can still have an impact on performance (more on that later in the course ➡)



# PRACTICAL SUMMARY



Diffing is how React decides which DOM elements need to be added or modified. If, between renders, a certain React element **stays at the same position in the element tree**, the corresponding DOM element and component state will stay the same. If the element **changed to a different position**, or if it's a **different element type**, the DOM element and state will be destroyed



Giving elements a key prop allows React to distinguish between multiple component instances. **When a key stays the same across renders**, the element is kept in the DOM. This is why we need to use keys in lists. **When we change the key between renders**, the DOM element will be destroyed and rebuilt. We use this as a **trick to reset state**



**Never declare a new component inside another component!** Doing so will re-create the nested component every time the parent component re-renders. React will always see the nested component as **new**, and therefore **reset its state** each time the parent state is updated



The logic that produces JSX output for a component instance ("render logic") is **not allowed to produce any side effects**: no API calls, no timers, no object or variable mutations, no state updates. **Side effects are allowed in event handlers and useEffect** (next section ➡)



# PRACTICAL SUMMARY



The DOM is updated in the commit phase, **but not by React, but by a “renderer” called ReactDOM**. That’s why we always need to include both libraries in a React web app project. We can use other renderers to use React on different platforms, for example to build mobile or native apps



Multiple state updates inside an event handler function are **batched**, so they happen all at once, **causing only one re-render**. This means we can **not access a state variable immediately after updating it**: state updates are **asynchronous**. Since React 18, batching also happens in timeouts, promises, and native event handlers.



When using events in event handlers, we get access to a **synthetic event object**, not the browser’s native object, so that **events work the same way across all browsers**. The difference is that **most synthetic events bubble**, including focus, blur, and change, which do not bubble as native browser events. Only the scroll event does not bubble

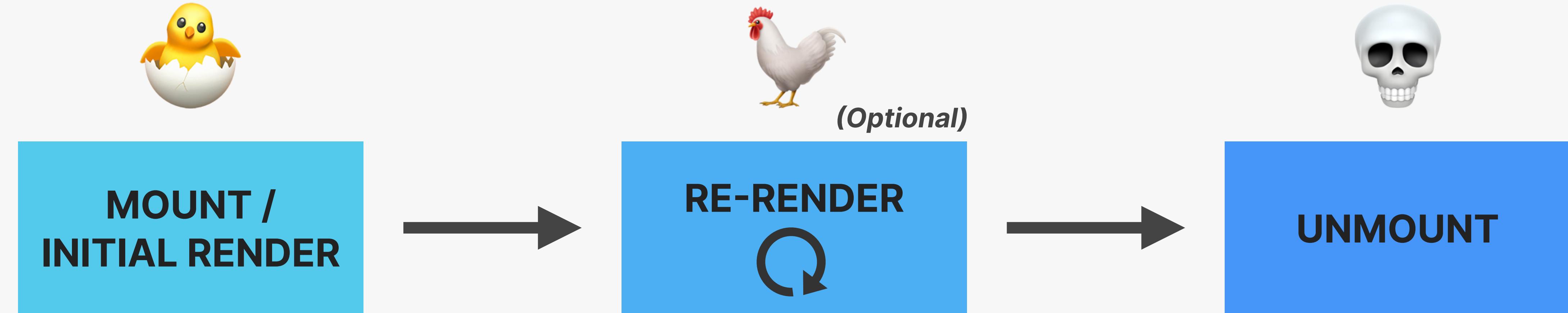


**React is a library, not a framework**. This means that you can assemble your application using your favorite third-party libraries. The downside is that you need to find and learn all these additional libraries. No problem, as you will learn about the most commonly used libraries in this course



# EFFECTS AND DATA FETCHING

# COMPONENT (INSTANCE) LIFECYCLE



- 👉 Component instance is rendered for the **first time**
- 👉 Fresh state and props are created

**HAPPENS WHEN:**

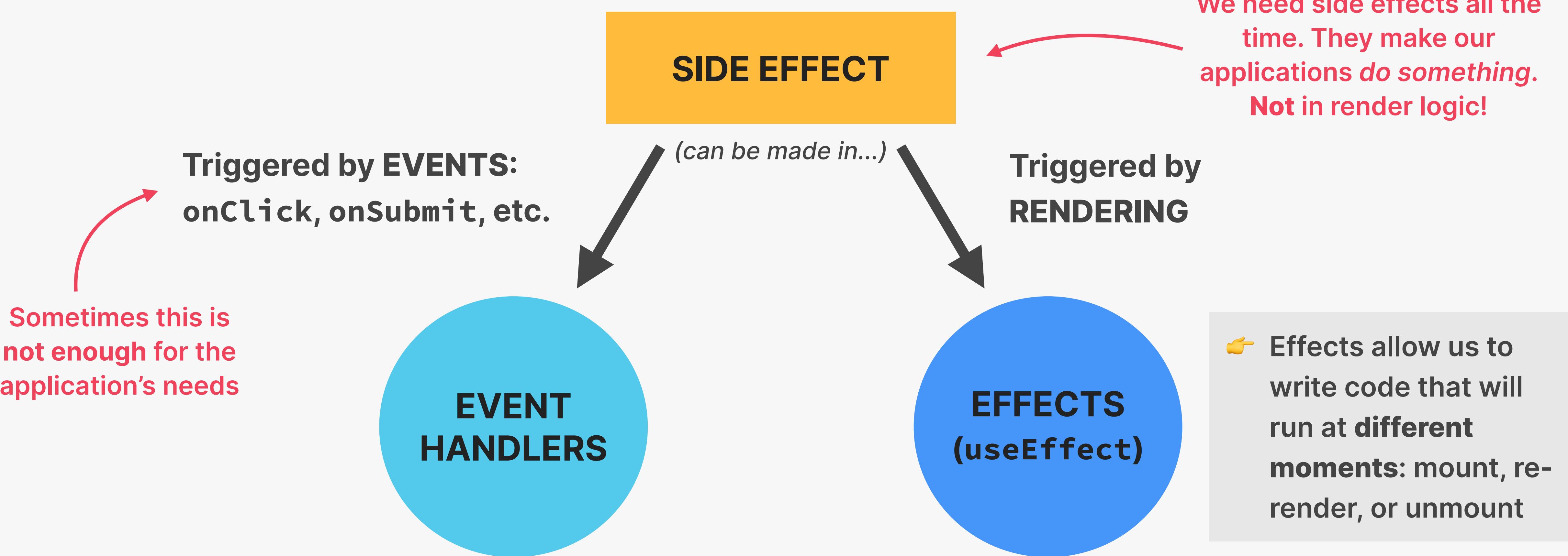
- 👉 State changes
- 👉 Props change
- 👉 Parent re-renders
- 👉 Context changes

- 👉 Component instance is **destroyed and removed**
- 👉 State and props are **destroyed**

👉 We can define code to run at these specific **points in time**



# WHERE TO CREATE SIDE EFFECTS



👉 **REVIEW:** A **side effect** is basically any “*interaction between a React component and the world outside the component*”. We can also think of a side as “*code that actually does something*”. Examples: Data fetching, setting up subscriptions, setting up timers, manually accessing the DOM, etc.

# EVENT HANDLERS VS. EFFECTS

## EVENT HANDLERS

```
function handleClick() {  
  fetch(`http://www.omdbapi.com/?s=inception`)  
    .then((res) => res.json())  
    .then((data) => setMovies(data.Search));  
}
```

- 👉 Executed when the **corresponding event** happens
- 👉 Used to **react** to an event
- 👉 Preferred way of creating side effects!

Produce the same result,  
but at **different moments**

## EFFECTS (useEffect)

```
useEffect(function () {  
  fetch(`http://www.omdbapi.com/?s=inception`)  
    .then((res) => res.json())  
    .then((data) => setMovies(data.Search));  
  
  return () => console.log('Cleanup');  
}, []);
```

Effect

Cleanup  
function

When?

Thinking about  
synchronization,  
not lifecycles

Dependency array

- 👉 Executed **after the component mounts** (initial render), and **after subsequent re-renders** (according to dependency array)

- 👉 Used to keep a component **synchronized with some external system** (in this example, with the API movie data)

(We'll come back to all this after using `useEffect` in practice...)



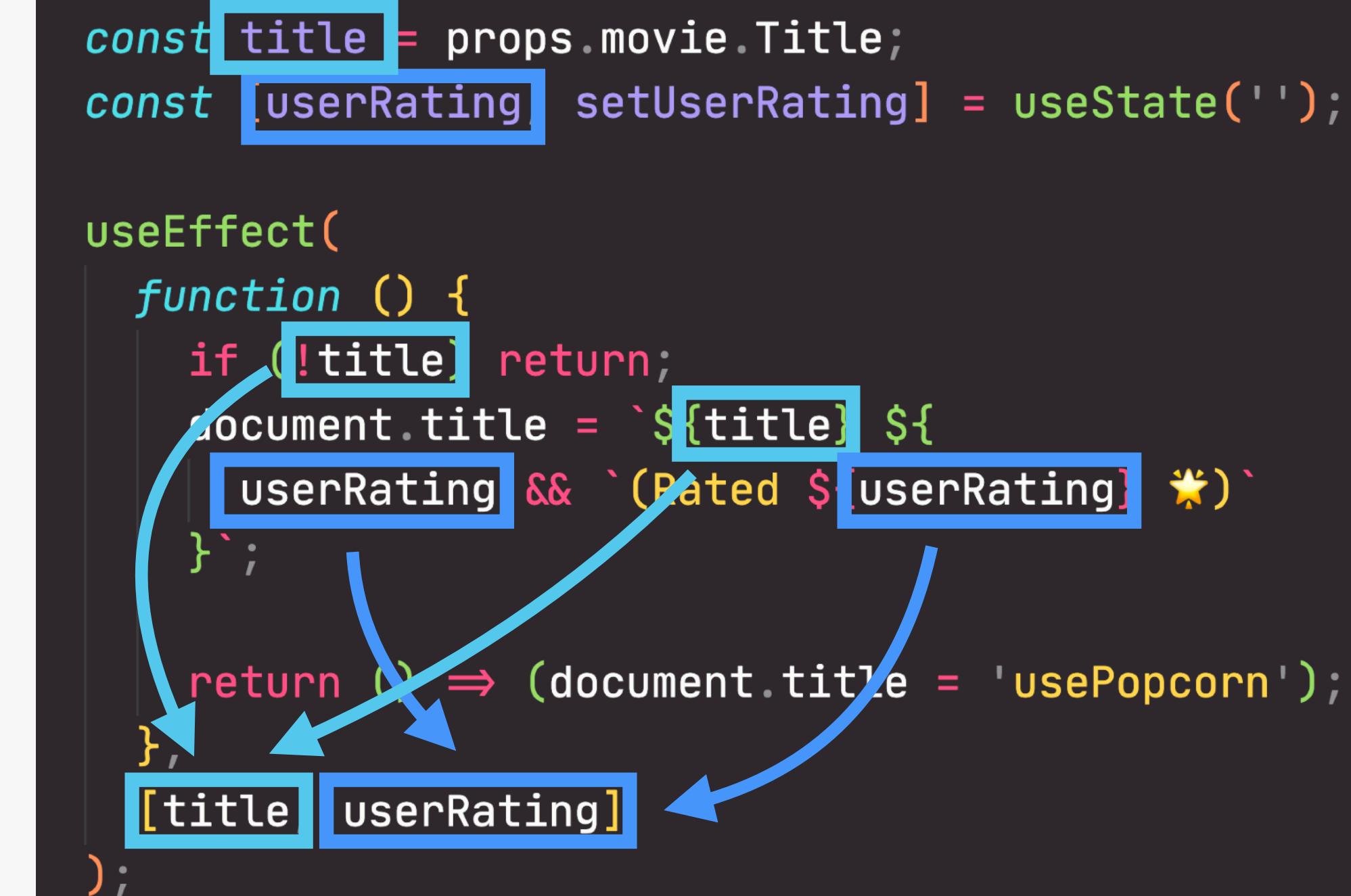
# WHAT'S THE USEEFFECT DEPENDENCY ARRAY?

## THE DEPENDENCY ARRAY

- 👉 By default, effects run **after every render**. We can prevent that by passing a **dependency array**
- 👉 Without the dependency array, React doesn't know **when** to run the effect
- 👉 ***Each time one of the dependencies changes, the effect will be executed again***
- 👉 Every **state variable** and **prop** used inside the effect **MUST** be included in the dependency array

```
const title = props.movie.Title;
const [userRating, setUserRating] = useState('');

useEffect(
  function () {
    if (!title) return;
    document.title = `${title} ${userRating} && `Rated ${userRating} ⭐`;
  },
  [title, userRating]
);
```

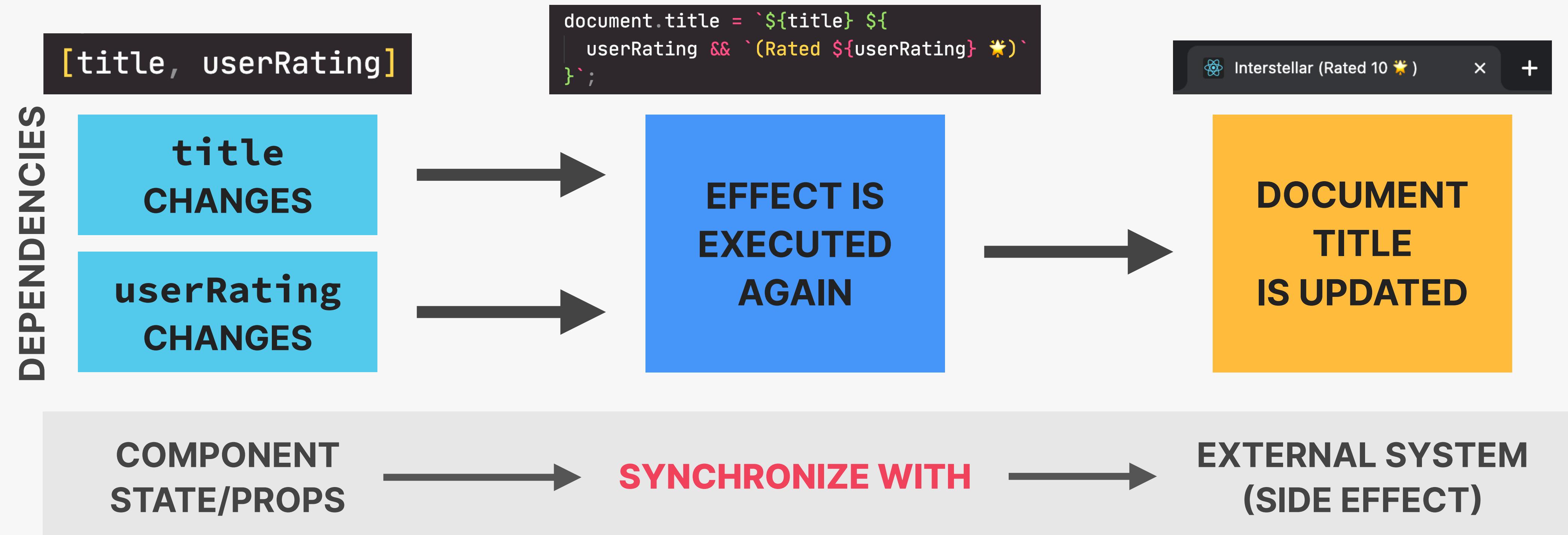


Otherwise, we get a “**stale closure**”. We will go more into depth in a future section 👉

# USEEFFECT IS A SYNCHRONIZATION MECHANISM

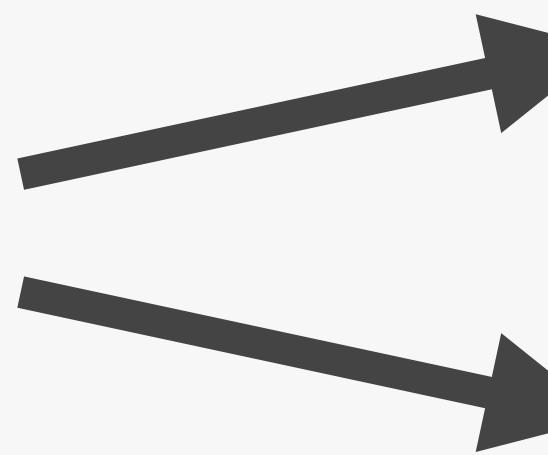
## THE MECHANICS OF EFFECTS

- 👉 **useEffect** is like an **event listener** that is listening for one dependency to change. **Whenever a dependency changes, it will execute the effect again**
- 👉 Effects **react** to updates to state and props used inside the effect (the dependencies). So **effects are “reactive”** (like state updates re-rendering the UI)



# SYNCHRONIZATION AND LIFECYCLE

DEPENDENCY  
(STATE OR PROPS)  
CHANGES



EFFECT IS  
EXECUTED AGAIN

COMPONENT IS  
RE-RNDERED

Effects and  
component lifecycle  
are deeply connected

👉 We can use the dependency array to run effects when the component renders or re-renders



## SYNCHRONIZATION

`useEffect(fn, [x, y, z]);`



Effect synchronizes with x, y, and z

`useEffect(fn, []);`



Effect synchronizes with no state/props

`useEffect(fn);`



Effect synchronizes with everything



## LIFECYCLE

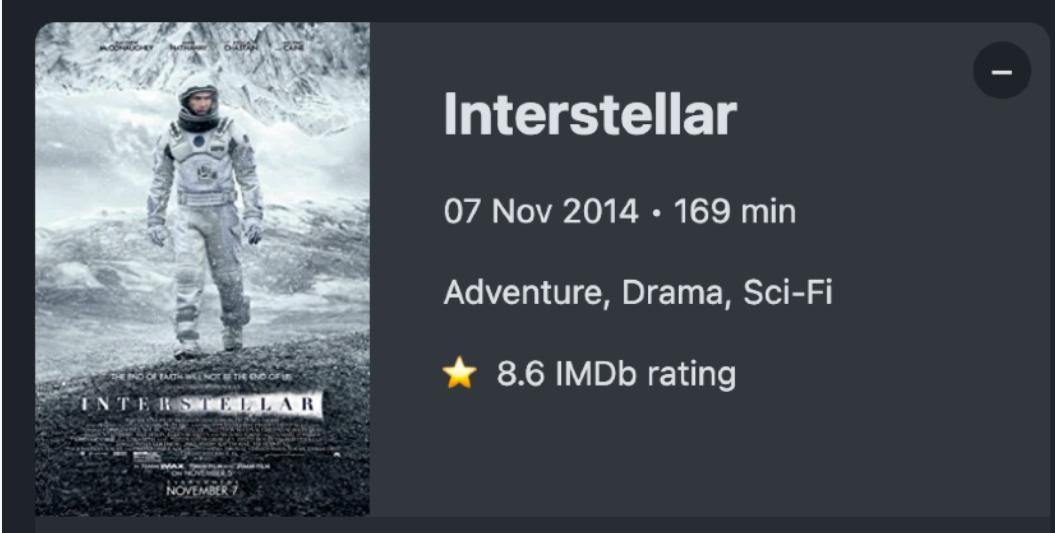
Runs on mount and re-renders triggered by updating x, y, or z

Runs only on mount (initial render)

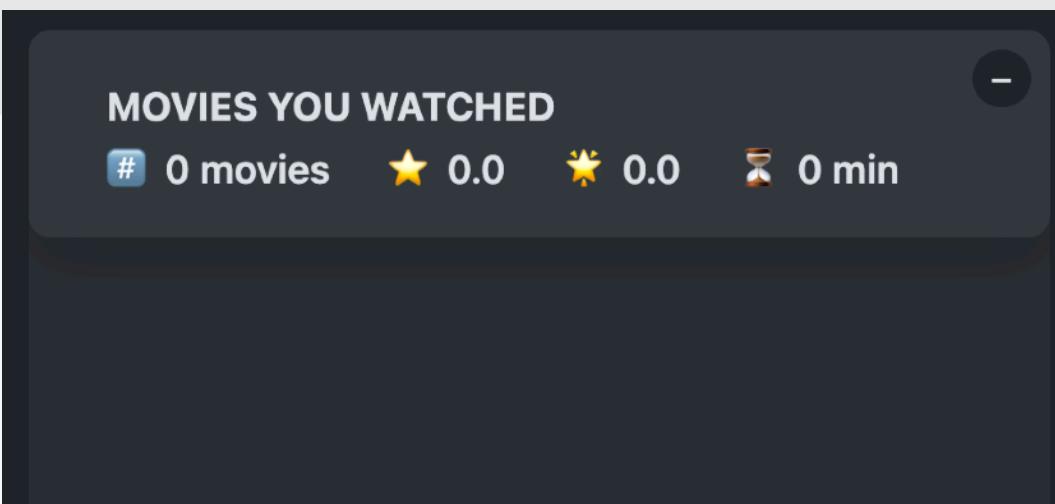
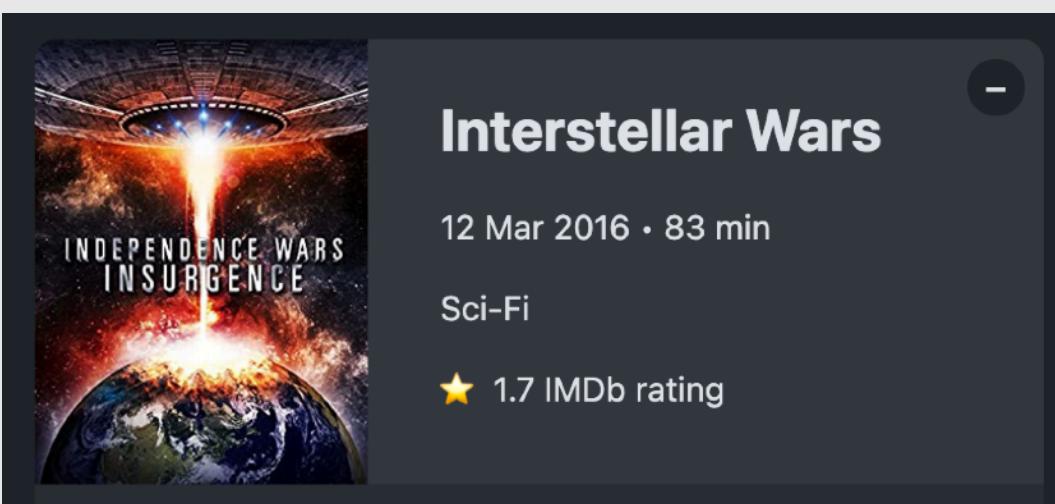
Runs on every render (usually bad 🚫)

# WHEN ARE EFFECTS EXECUTED?

`title = 'Interstellar'`



`title = 'Interstellar Wars'`



time

MOUNT (INITIAL RENDER)

COMMIT

BROWSER PAINT

EFFECT ✨

title CHANGES

RE-RENDER

COMMIT

LAYOUT EFFECT

BROWSER PAINT



EFFECT ✨

UNMOUNT



<MovieDetails />

If an effect sets state, an additional render will be required

```
document.title = `${title} ${userRating && `Rated ${userRating} ★`};
```

Interstellar

Another type of effect that is very rarely necessary (useLayoutEffect)

[title, userRating]

```
document.title = `${title} ${userRating && `Rated ${userRating} ★`};
```

Interstellar Wars

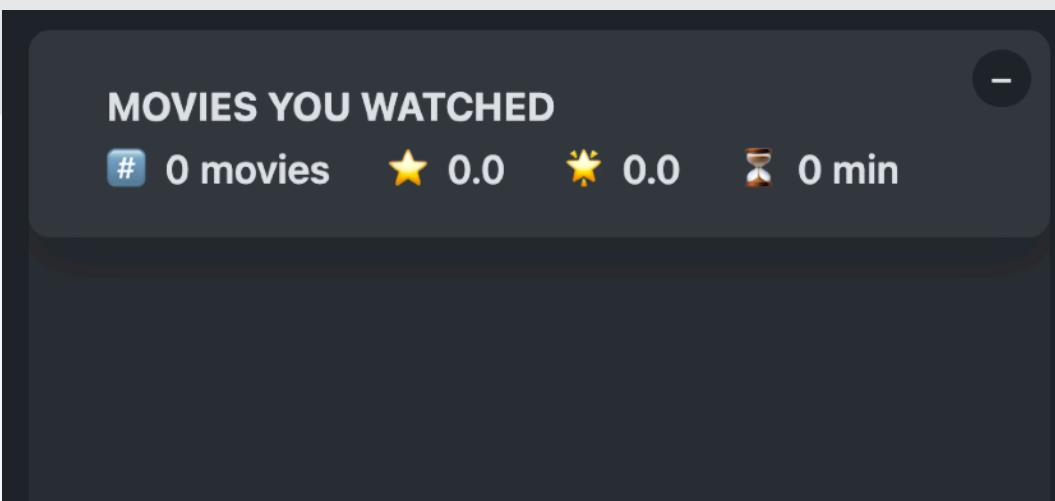
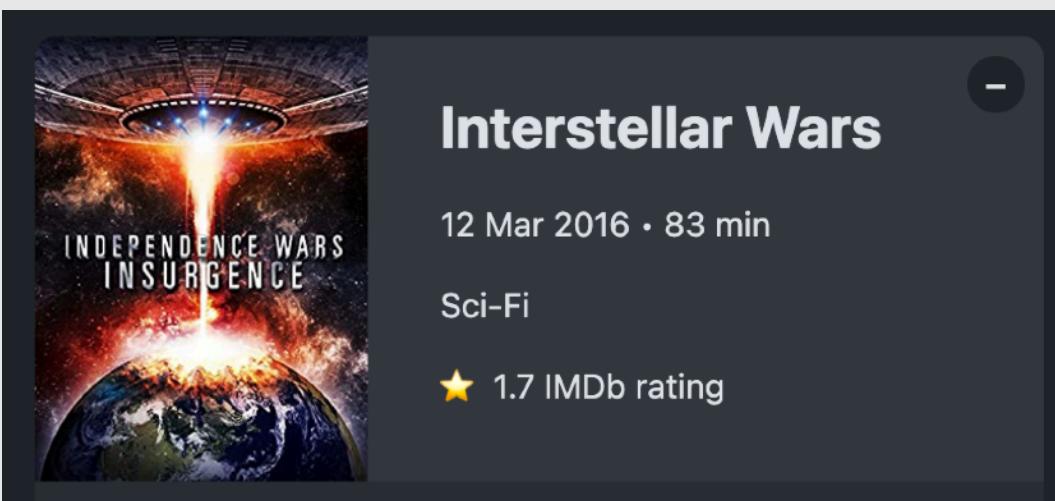


# WHEN ARE EFFECTS EXECUTED?

`title = 'Interstellar'`



`title = 'Interstellar Wars'`



time

MOUNT (INITIAL RENDER)

COMMIT

BROWSER PAINT

EFFECT ✨

title CHANGES

RE-RENDER

COMMIT

LAYOUT EFFECT

BROWSER PAINT

CLEANUP 🪢

EFFECT ✨

UNMOUNT

CLEANUP 🪢

<MovieDetails />

```
document.title = `${title} ${  
  userRating && `Rated ${userRating} ★`}  
};
```

Interstellar

```
() => (document.title = 'usePopcorn');
```

usePopcorn

```
document.title = `${title} ${  
  userRating && `Rated ${userRating} ★`}  
};
```

Interstellar Wars

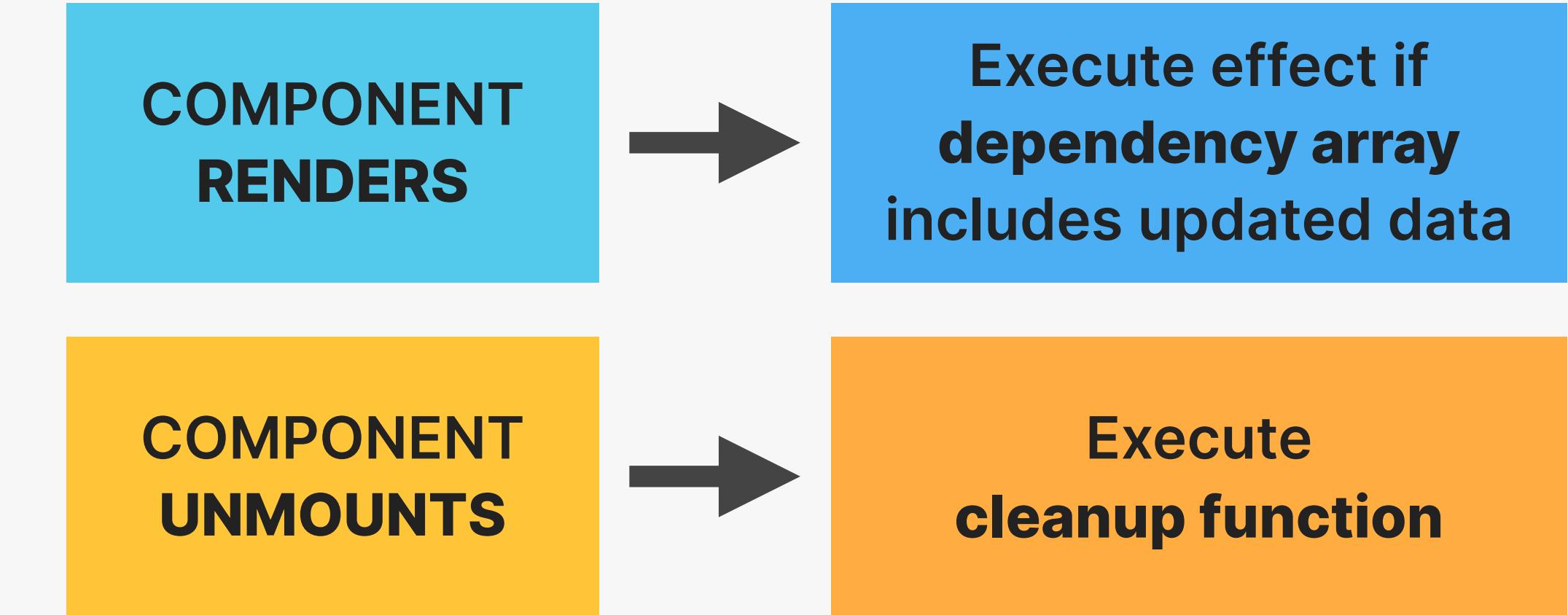
```
() => (document.title = 'usePopcorn');
```

usePopcorn

# THE CLEANUP FUNCTION

## USEEFFECT CLEANUP FUNCTION

- 👉 Function that we can return from an effect (*optional*)
- 👉 Runs on two different occasions:
  - 1 Before the effect is **executed again**
  - 2 After a component has **unmounted**
- 👉 Necessary whenever the side effect **keeps happening after the component has been re-rendered or unmounted**
- 👉 Each effect should do **only one thing!** Use **one useEffect hook for each side effect.** This makes effects easier to clean up



Examples →

### EFFECT

- 👉 HTTP request → Cancel request
- 👉 API subscription → Cancel subscription
- 👉 Start timer → Stop timer
- 👉 Add event listener → Remove listener



### POTENTIAL CLEANUP



CUSTOM HOOKS,  
REFS, AND MORE  
STATE

# WHAT ARE REACT HOOKS?

## REACT HOOKS

- 👉 Special built-in functions that allow us to “hook” into React internals:
  - 👉 Creating and accessing **state** from Fiber tree
  - 👉 Registering **side effects** in Fiber tree
  - 👉 Manual **DOM selections**
  - 👉 Many more...
- 👉 Always start with “use” (`useState`, `useEffect`, etc.)
- 👉 Enable easy **reusing of non-visual logic**: we can compose multiple hooks into our own **custom hooks**
- 👉 Give **function components** the ability to own state and run side effects at different lifecycle points (before v16.8 only available in **class components**)

# OVERVIEW OF ALL BUILT-IN HOOKS



## MOST USED

✓ useState

✓ useEffect

👉 useReducer

👉 useContext



## LESS USED

👉 useRef

👉 useCallback

👉 useMemo

👉 useTransition

👉 useDeferredValue

✗ useLayoutEffect

✗ useDebugValue

✗ useImperativeHandle

✗ useId



## ONLY FOR LIBRARIES

✗ useSyncExternalStore

✗ useInsertionEffect

✓ Have learned

👉 Will learn

✗ Will not learn

👋 As of React v18.x

# THE RULES OF HOOKS

## RULES OF HOOKS

1

### Only call hooks at the top level

- 👉 Do NOT call hooks inside **conditionals**, **loops**, **nested functions**, or after an **early return**
- 👉 This is necessary to ensure that hooks are always called in the **same order** (hooks rely on this)

2

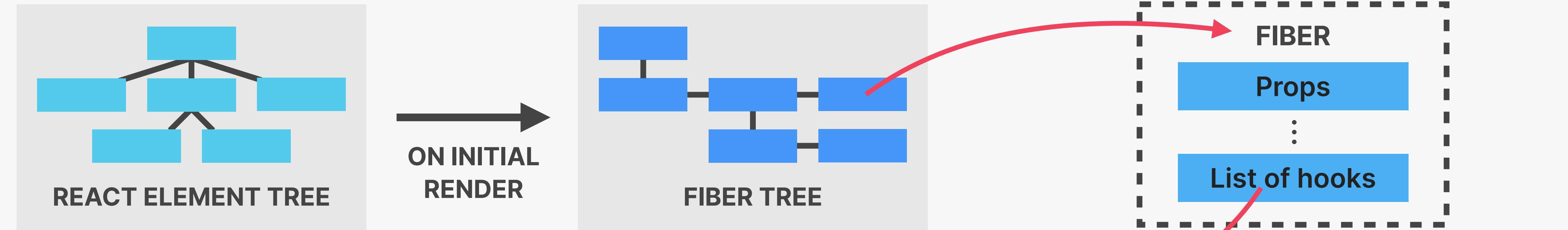
### Only call hooks from React functions

- 👉 Only call hooks inside a **function component** or a **custom hook**



*These rules are automatically enforced by React's ESLint rules*

# HOOKS RELY ON CALL ORDER

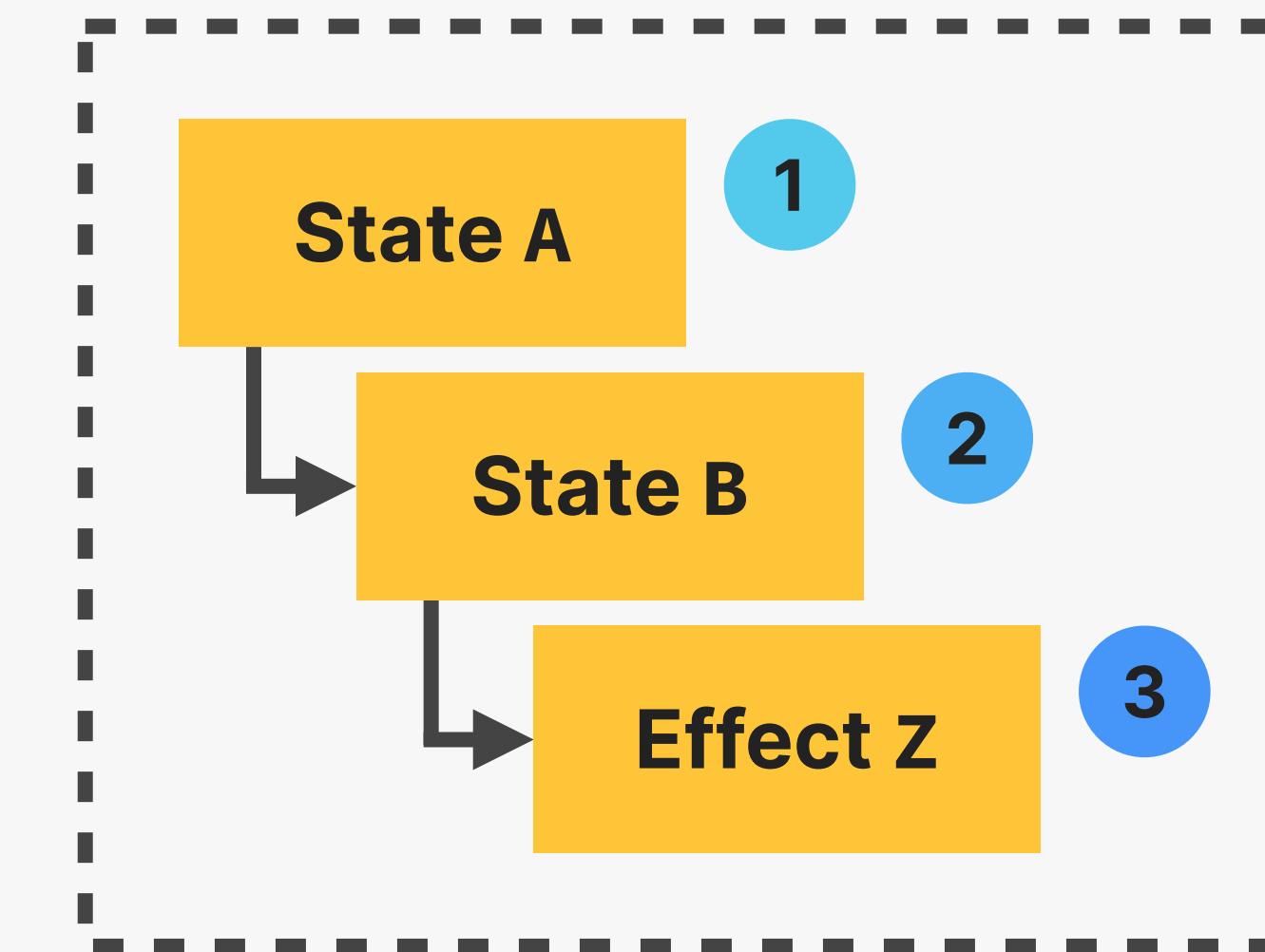


👉 Hypothetical example! This code does **NOT** work

```
const [A, setA] = useState(23) 1
if (A === 23) false
const [B, setB] = useState('') 2
useEffect(fnZ, []) 3
```

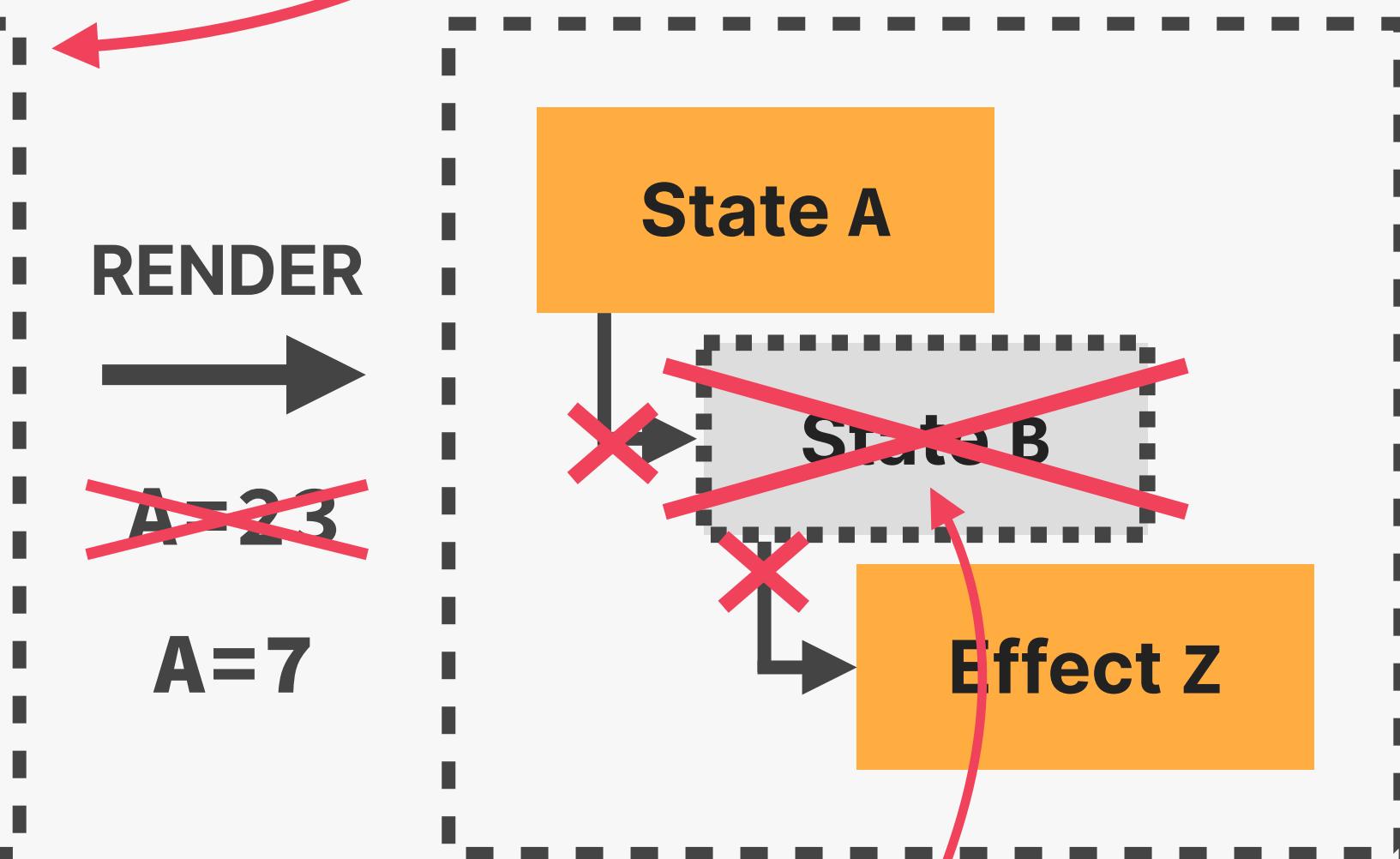
Violates Rule #1

List built based on hooks call order



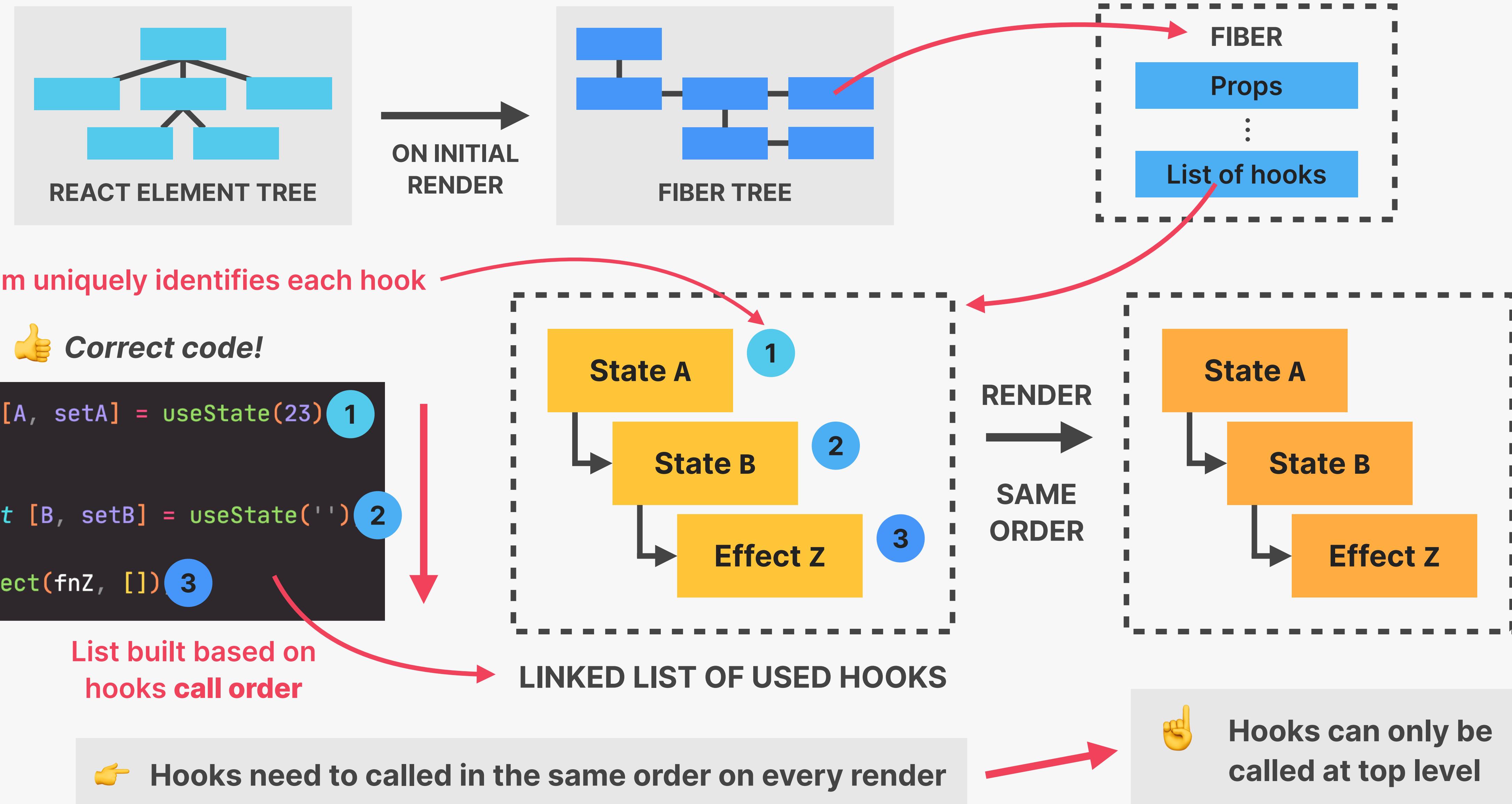
LINKED LIST OF USED HOOKS

👉 Hooks need to called in the same order on every render



A==23 is now false, so after re-render, this hook would no longer exist, destroying the linked list 😢

# HOOKS RELY ON CALL ORDER





# SUMMARY OF DEFINING AND UPDATING STATE

1

## CREATING STATE

Simple

Based on function  
(lazy evaluation)

Make sure to **NOT** mutate objects  
or arrays, but to **replace** them

2

## UPDATING STATE

Simple

Based on current state

```
const [count, setCount] = useState(23);
```

```
const [count, setCount] = useState(  
  () => localStorage.getItem('count')  
)
```

- 👉 Function must be **pure** and accept **no arguments**. Called only on **initial render**

```
setCount(1000);
```

```
setCount((c) => c + 1)
```

- 👉 Function must be **pure** and return **next state**

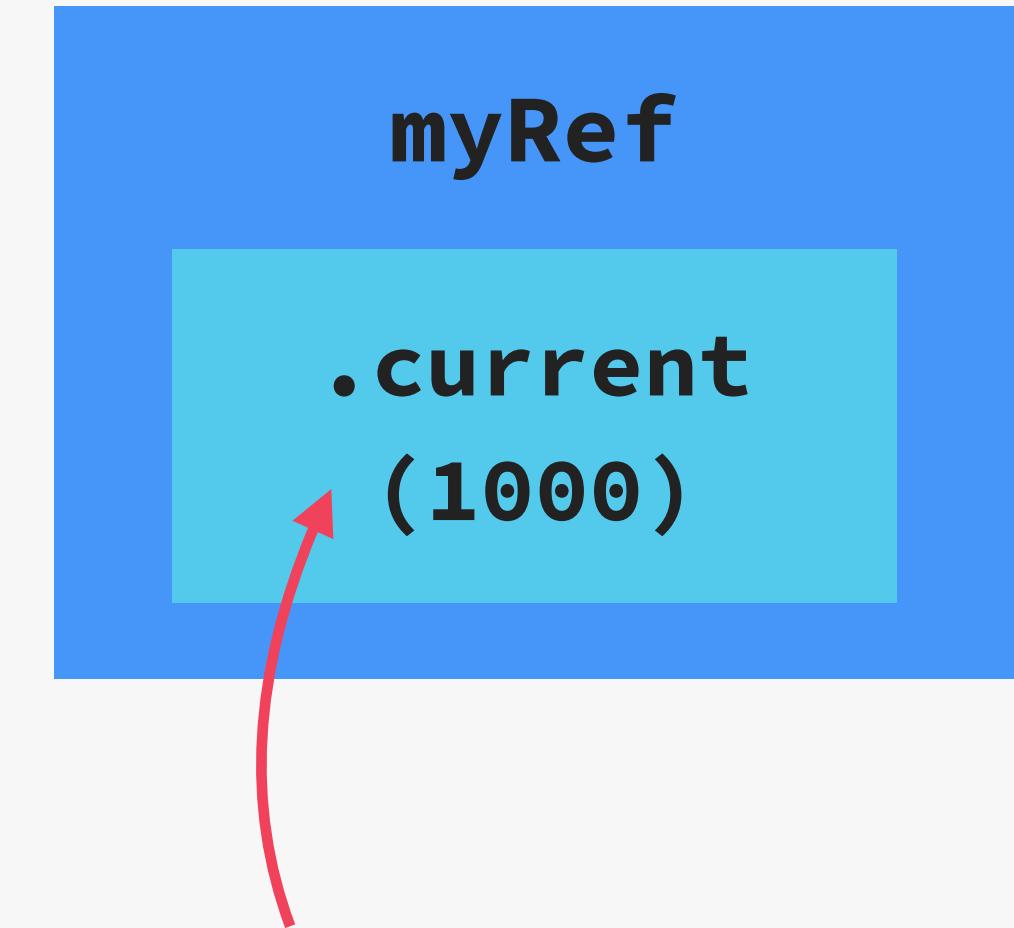


# WHAT ARE REFS?

## REF WITH `useRef`

- 👉 “Box” (object) with a **mutable** `.current` property that is **persisted across renders** (“normal” variables are always reset)
- 👉 Two big use cases:
  - 1 Creating a variable that stays the same between renders (e.g. previous state, setTimeout id, etc.)
  - 2 Selecting and storing DOM elements
- 👉 Refs are for **data that is NOT rendered**: usually only appear in event handlers or effects, not in JSX (otherwise use state)
- 👉 Do **NOT** read write or read `.current` in render logic (like state)

```
const myRef = useRef(23);
```

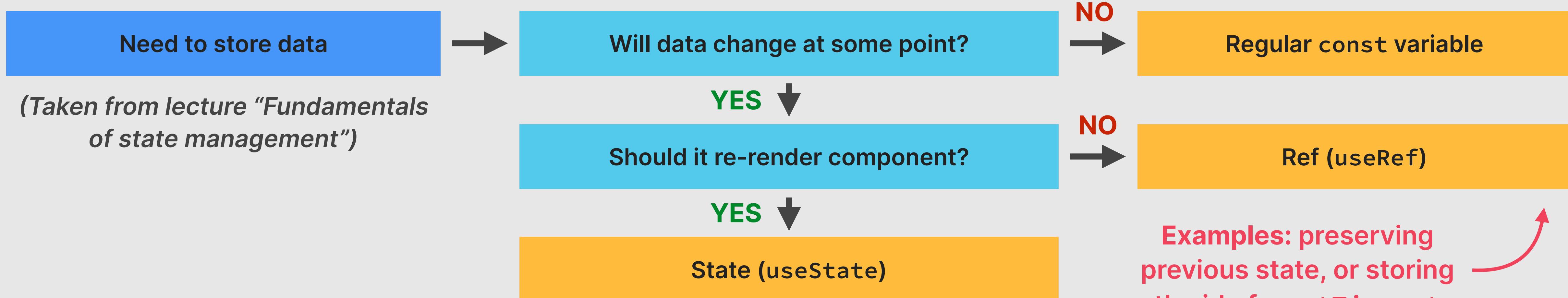


We can write to and  
read from the ref  
using `.current`

```
myRef.current = 1000;
```

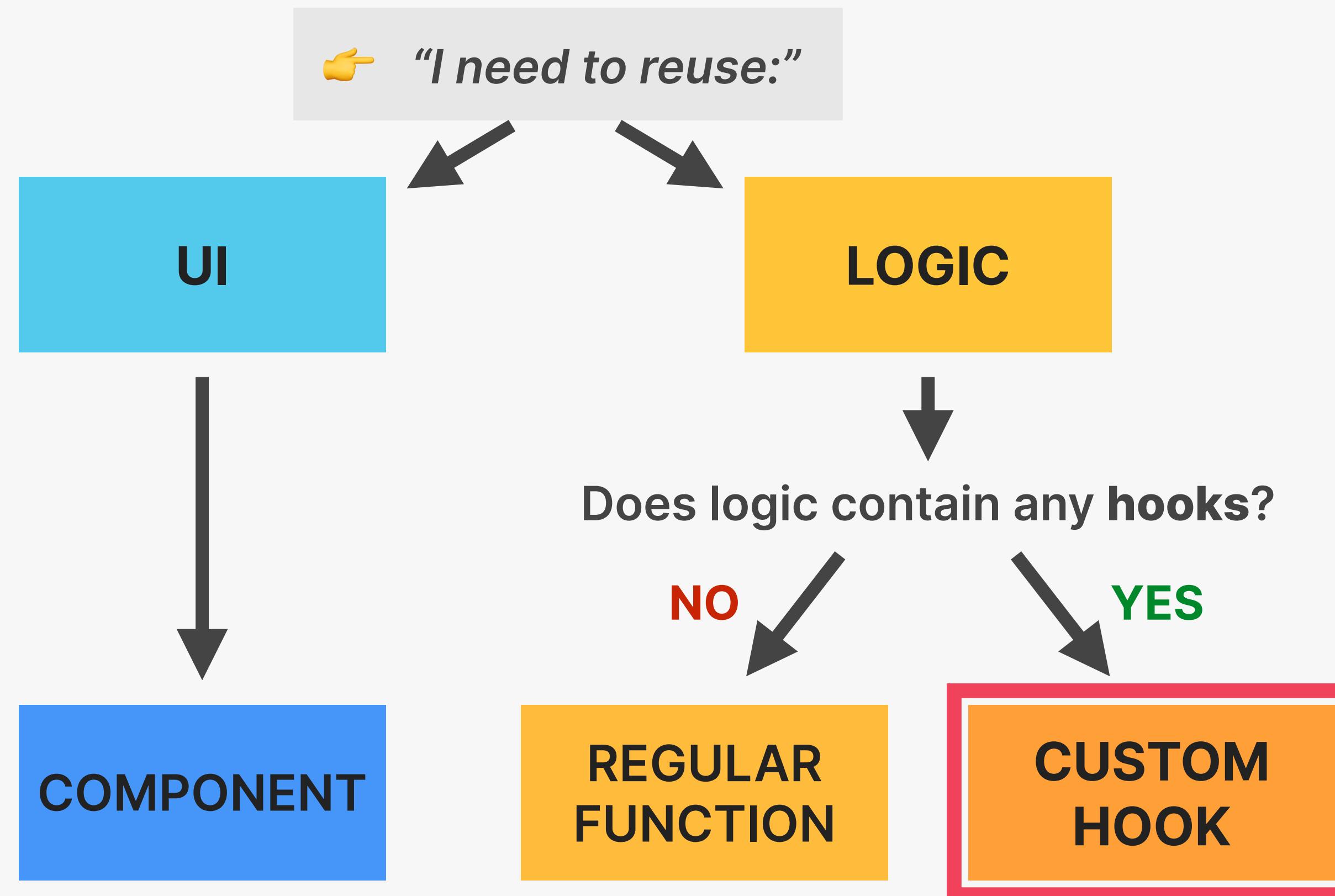
# STATE VS. REFS

	PERSISTS ACROSS RENDERS	UPDATING CAUSES RE-RENDER	IMMUTABLE	ASYNCHRONOUS UPDATES
STATE	✓	✓	✓	✓
REFS	✓	✗	✗	✗





# REUSING LOGIC WITH CUSTOM HOOKS



- 👉 Allow us to reuse **non-visual logic** in multiple components
- 👉 One custom hook should have **one purpose**, to make it **reusable** and **portable** (even across multiple projects)
- 👉 Rules of hooks apply to custom hooks too

```
function useFetch(url) {  
  const [data, setData] = useState([]);  
  const [isLoading, setIsLoading] = useState(false);  
  
  useEffect(function () {  
    fetch(url)  
      .then(res => res.json())  
      .then(res => setData(res));  
  }, []);  
  
  return [data, isLoading]  
}
```

The code shows a custom hook named `useFetch`. It uses the `useState` hook to manage state. It also uses the `useEffect` hook to fetch data from a URL. The returned value is an array containing the fetched data and the loading status.

Needs to use **one or more hooks**

Function name needs to start with **use**

Unlike components, can receive and return **any relevant data** (usually `[]` or `{}`)



**REACT BEFORE  
HOOKS: CLASS-  
BASED REACT**

# FUNCTION COMPONENTS VS. CLASS COMPONENTS

Existed since beginning,  
but without hooks

## FUNCTION COMPONENTS

## CLASS COMPONENTS

JUL 17	Introduced in	v16.8 (2019, with hooks)	v0.13 (2015)
💎	How to create	JavaScript function (any type)	ES6 class, extending React.Component
👀	Reading props	Destructuring or props.X	this.props.X
🚀	Local state	useState hook	this.setState()
✨	Side effects/lifecycle	useEffect hook	Lifecycle methods
☎️	Event handlers	Functions	Class methods
🎨	Returning JSX	Return JSX from function	Return JSX from render method
🏆	Advantages	<ul style="list-style-type: none"><li>👍 Easier to build (less boilerplate code)</li><li>👍 Cleaner code: useEffect combines all lifecycle-related code in a single place</li><li>👍 Easier to share stateful logic</li><li>👍 We don't need this keyword anymore</li></ul>	<ul style="list-style-type: none"><li>👍 Lifecycle might be easier to understand for beginners</li></ul>



# PART 03

---

# ADVANCED REACT + REDUX

# THE ADVANCED USERREDUCER HOOK

# WHY USEREDUCER?

## 👉 STATE MANAGEMENT WITH useState IS NOT ENOUGH IN CERTAIN SITUATIONS:

1

When components have **a lot of state variables and state updates**, spread across many event handlers **all over the component**

2

When **multiple state updates** need to happen **at the same time** (as a reaction to the same event, like “starting a game”)

3

When updating one piece of state **depends on one or multiple other pieces of state**

## 👉 IN ALL THESE SITUATIONS, useReducer CAN BE OF GREAT HELP

# MANAGING STATE WITH USERREDUCER

## STATE WITH useReducer

- 👉 An alternative way of setting state, ideal for **complex state and related pieces of state**
- 👉 Stores related pieces of state in a **state** object Like `setState()` with superpowers
- 👉 `useReducer` needs **reducer:** function containing **all logic to update state**. Decouples state logic from component
- 👉 **reducer:** pure function (*no side effects!*) that takes current state and action, and returns the next state
- 👉 **action:** object that describes **how to update state**
- 👉 **dispatch:** function to trigger state updates, by “**sending**” actions from event handlers to the reducer

Instead of `setState()`

```
const [state, dispatch] =  
  useReducer(reducer, initialState);
```

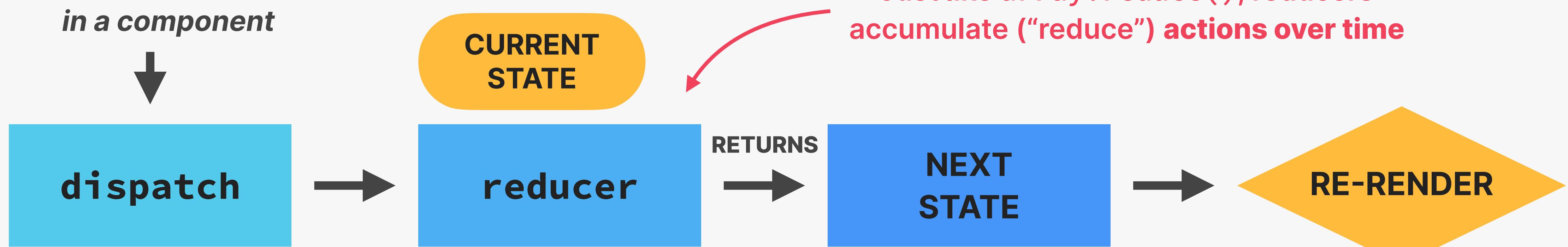
```
function reducer(state, action) {  
  switch (action.type) {  
    case 'dec':  
      return state - 1;  
    case 'inc':  
      return state + 1;  
    case 'setCount':  
      return action.payload;  
    default:  
      throw new Error('Unknown');  
  }  
}
```

# HOW REDUCERS UPDATE STATE

```
const [state, dispatch] = useReducer(reducer, initialState);
```

useReducer

👉 Updating state  
in a component



action  
`type = 'updateDay'`  
`payload = 23`

Object that contains information on  
how the reducer should update state

useState

UPDATED STATE

UPDATE

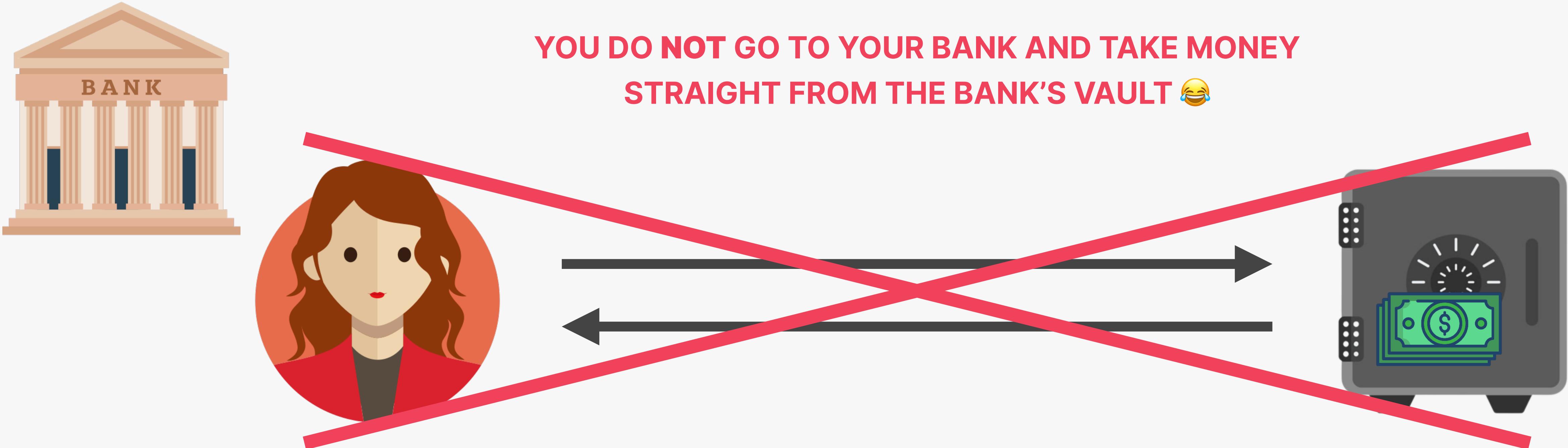
NEXT (UPDATED)  
STATE

useState

RE-RENDER

# A MENTAL MODEL FOR REDUCERS

👉 REAL-WORLD TASK: WITHDRAWING \$5,000 FROM YOUR BANK ACCOUNT



# A MENTAL MODEL FOR REDUCERS



## REAL-WORLD TASK: WITHDRAWING \$5,000 FROM YOUR BANK ACCOUNT



**DISPATCHER**

(Who **requests** the update)

I would like to withdraw  
\$5,000 from account 923577



**REDUCER**

(Who **makes** the update)

(How to make the update)

**ACTION**

```
type: 'withdraw',
payload: {
  amount: 5000,
  account: 923577,
},
```



**STATE**

(What needs to be updated)



# USESTATE VS. USEREDUCER

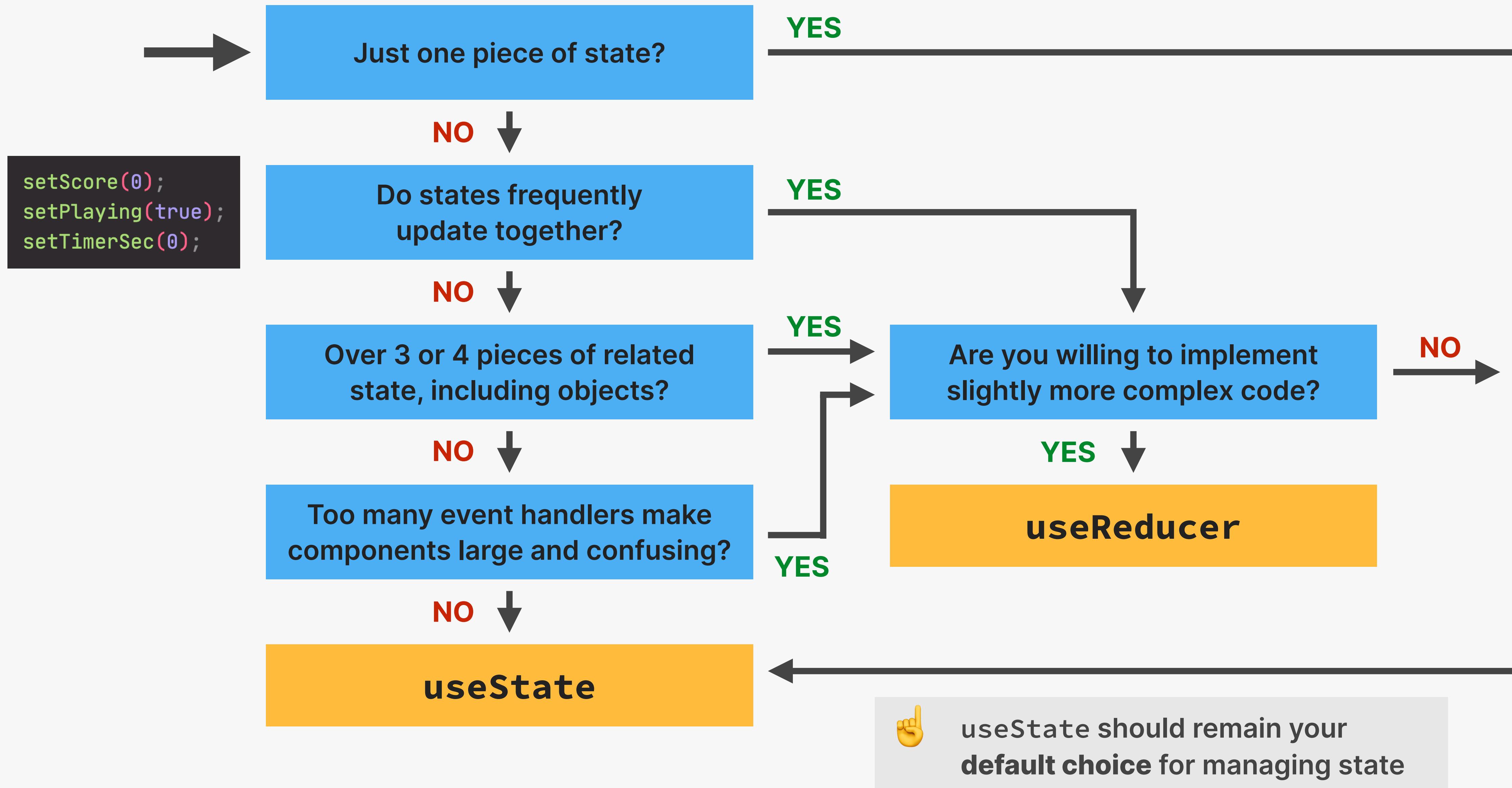
## useState

- 👉 Ideal for **single, independent pieces of state** (numbers, strings, single arrays, etc.)
- 👉 Logic to update state is placed directly in event handlers or effects, **spread all over one or multiple components**
- 👉 State is updated by **calling setState** (setter returned from useState)
- 👉 **Imperative** state updates
  - `setScore(0);  
setPlaying(true);  
setTimerSec(0);`
- 👉 Easy to understand and to use

## useReducer

- 👉 Ideal for **multiple related pieces of state** and **complex state** (e.g. object with many values and nested objects or arrays)
- 👉 Logic to update state lives in **one central place, decoupled from components**: the reducer
- 👉 State is updated by **dispatching an action** to a reducer
- 👉 **Declarative** state updates: complex state transitions are **mapped to actions**
  - `dispatch({ type: 'startGame' })`
- 👉 More **difficult** to understand and implement

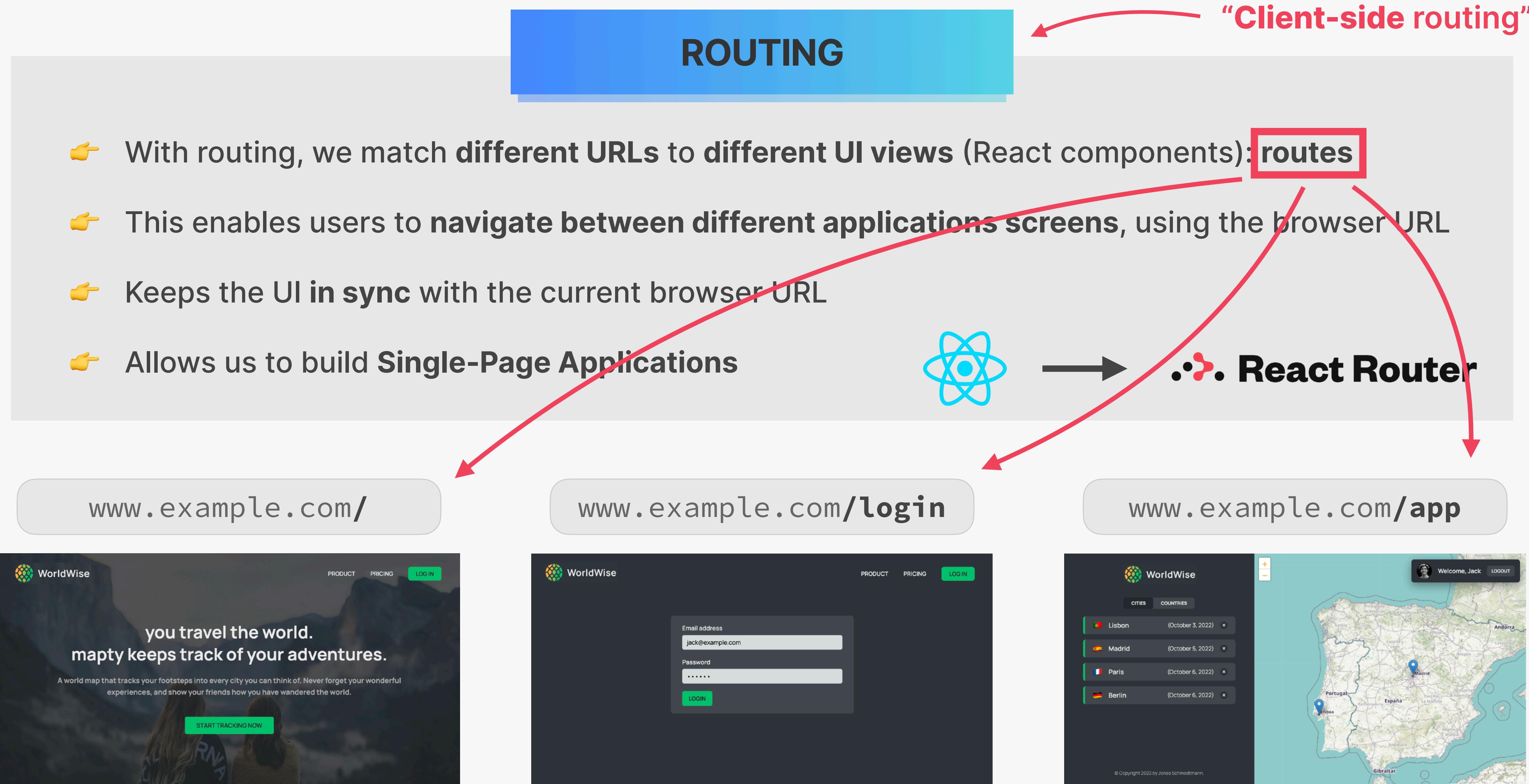
# WHEN TO USE USEREDUCER?



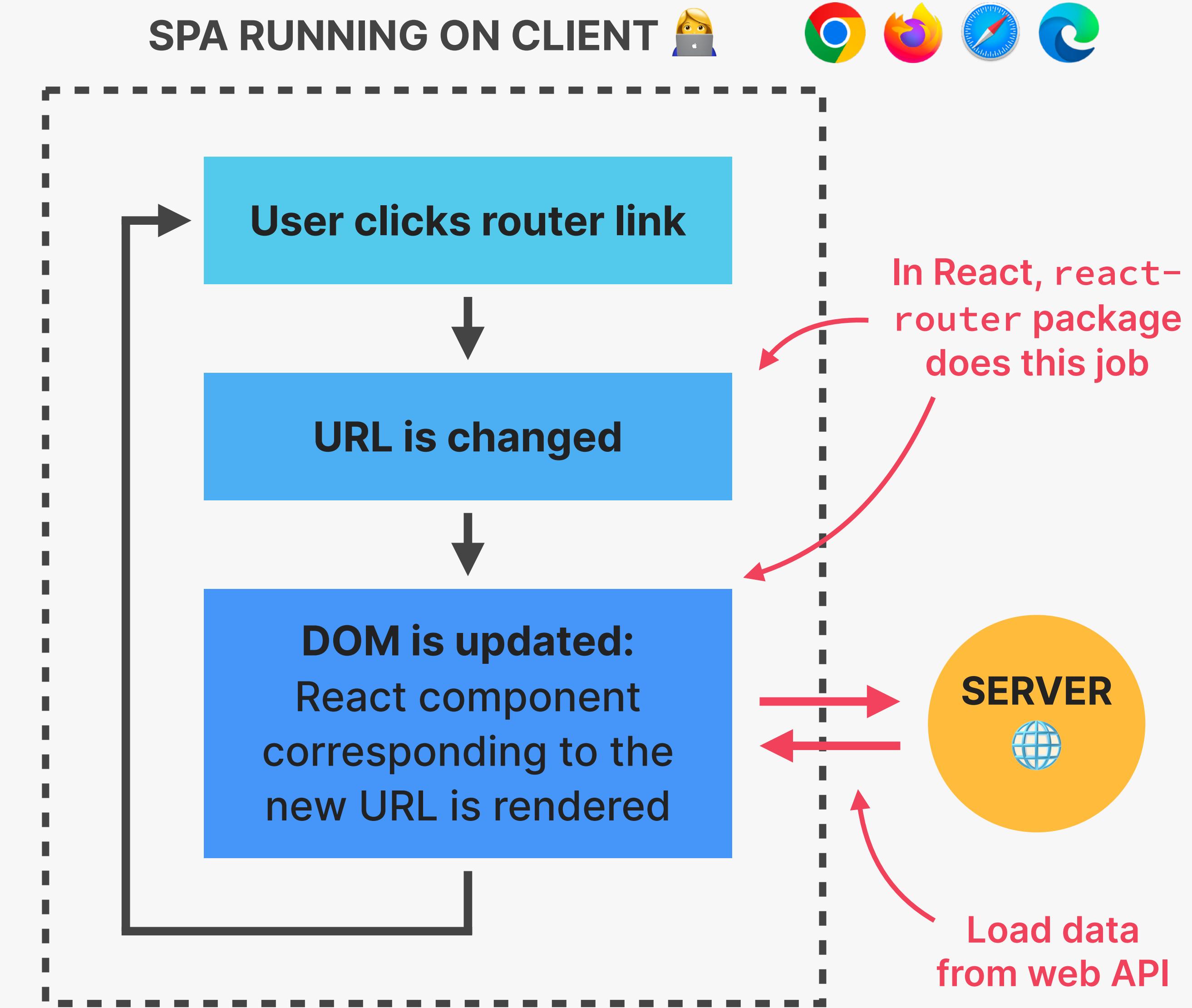
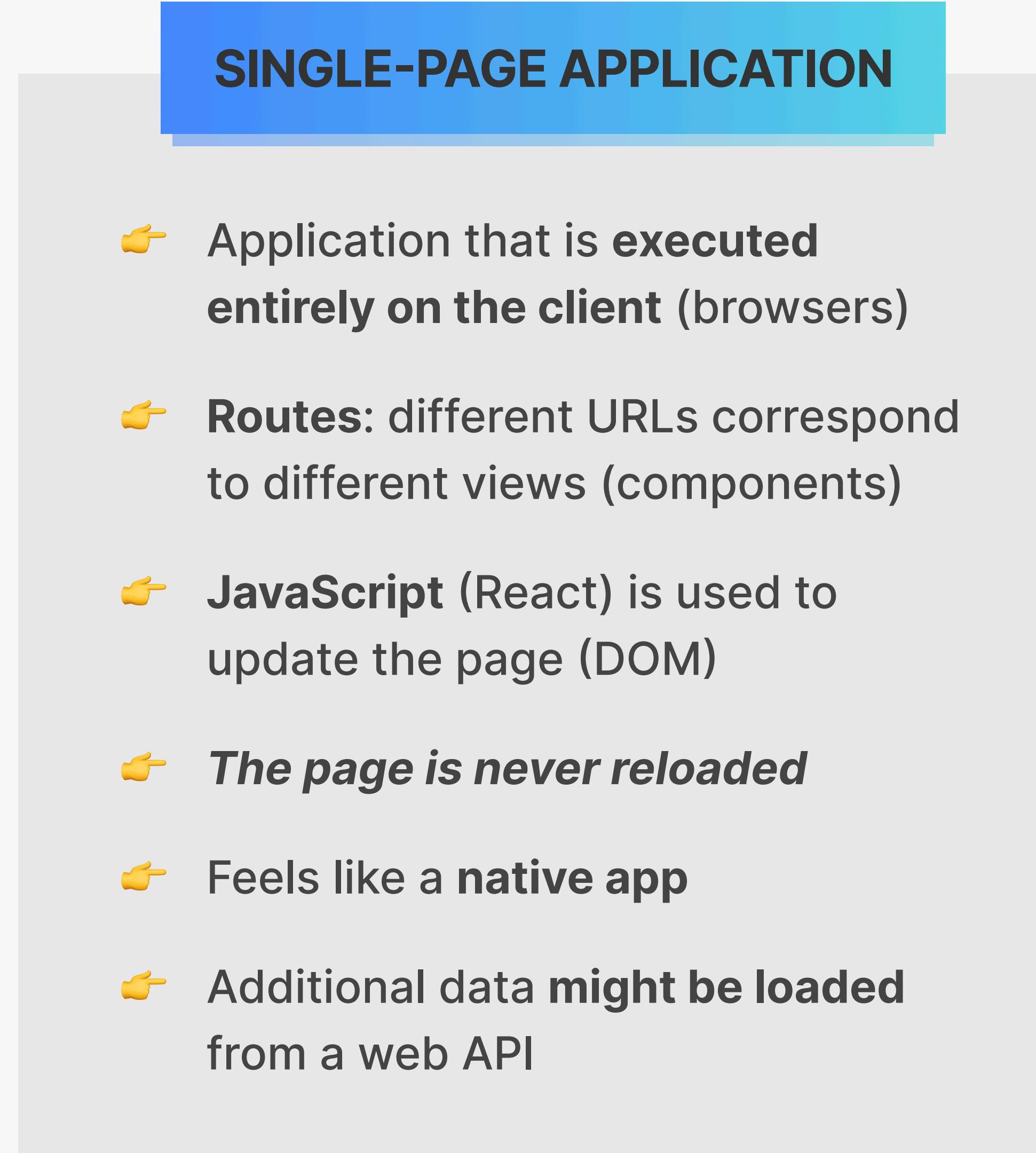


# REACT ROUTER: BUILDING SINGLE- PAGE APPLICATIONS (SPA)

# WHAT IS ROUTING?



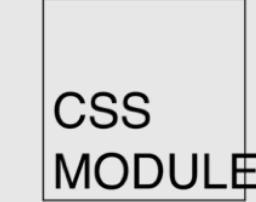
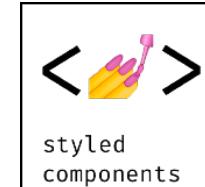
# SINGLE-PAGE APPLICATIONS (SPA)





# STYLING OPTIONS IN REACT

React doesn't care about styling

STYLING OPTION	WHERE?	HOW?	SCOPE	BASED ON
👉 <b>Inline CSS</b> 	JSX elements	style prop	<b>JSX element</b>	CSS
👉 <b>CSS or Sass file</b>  	External file	className prop	<b>Entire app</b>	CSS
👉 <b>CSS Modules</b> 	One external file per component	className prop	<b>Component</b>	CSS
👉 <b>CSS-in-JS</b> 	External file or component file	Creates new component	<b>Component</b>	JavaScript
👉 <b>Utility-first CSS</b> 	JSX elements	className prop	<b>JSX element</b>	CSS

👉 Alternative to styling with CSS: UI libraries like MUI, Chakra UI, Mantine, etc.   



# THE URL FOR STATE MANAGEMENT

- 👉 The URL is an excellent place to store UI state and an alternative to useState in some situations!  
Examples: open/closed panels, currently selected list item, list sorting order, applied list filters

- 1 Easy way to store state in a **global place**, accessible to **all components** in the app
- 2 Good way to “pass” data from one page into the next page
- 3 Makes it possible to **bookmark and share** the page with the exact UI state it had at the time

www.example.com /app/cities/lisbon?lat=38.728&lng=-9.141

• React Router tools:

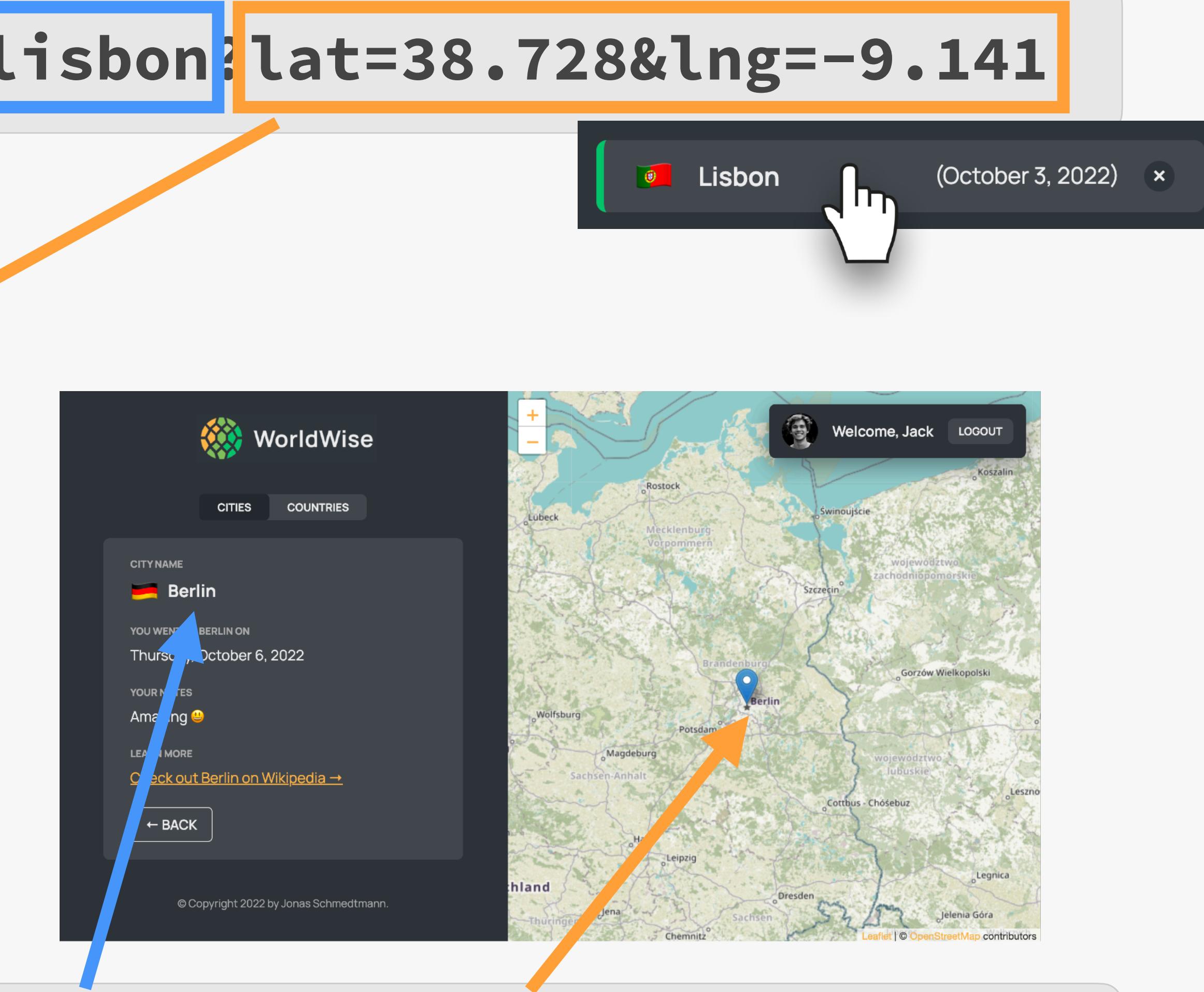
↑  
**path**

↑  
**params**

↑  
**query string**

# EXAMPLE: PARAMS AND QUERY STRING

www.example.com/app/cities/lisbon?lat=38.728&lng=-9.141



- 👉 City name and GPS location were retrieved from the URL instead of application state!

www.example.com/app/cities/berlin?lat=52.536&lng=13.377



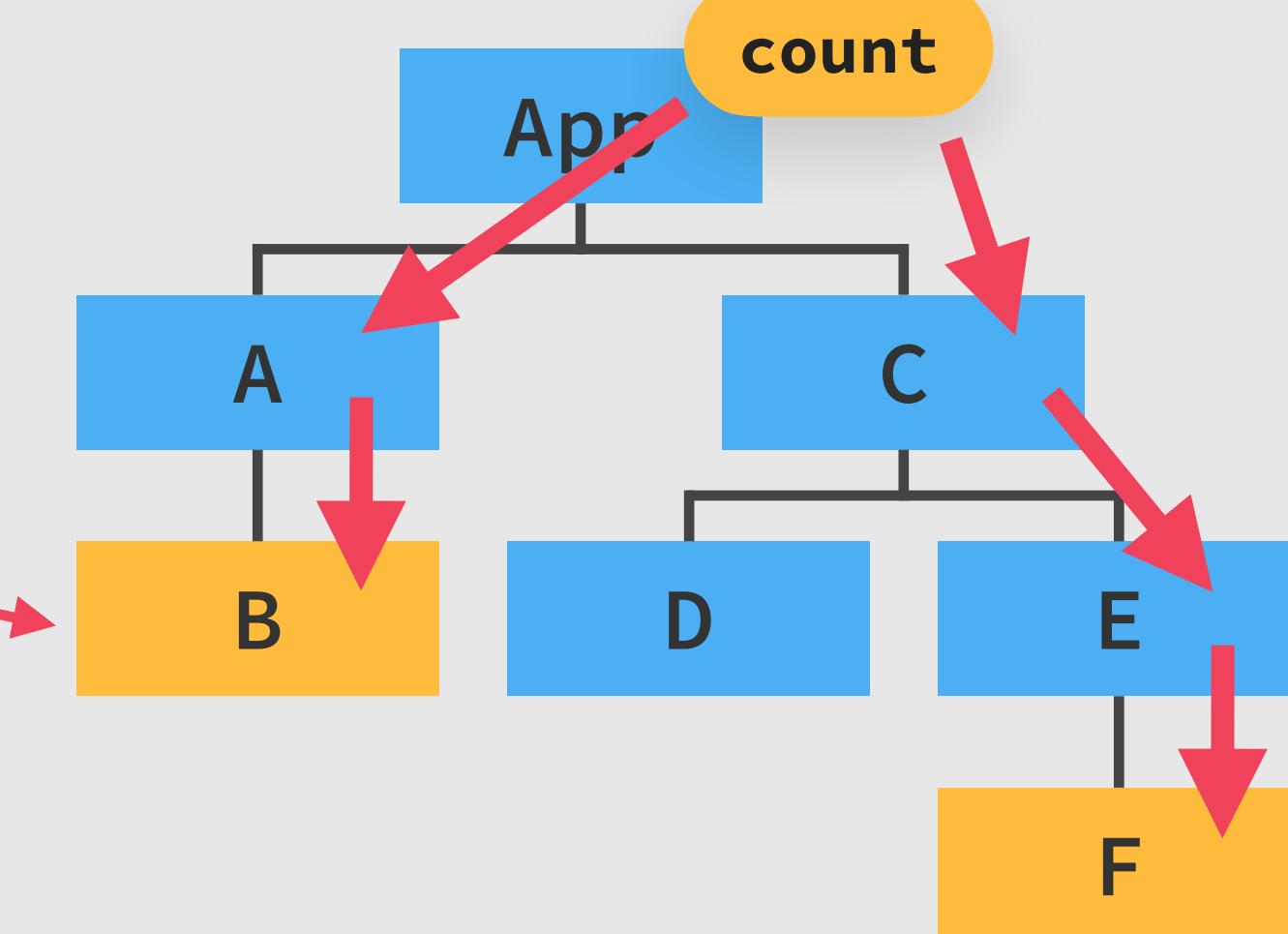
# ADVANCED STATE MANAGEMENT: THE CONTEXT API

# A SOLUTION TO PROP DRILLING

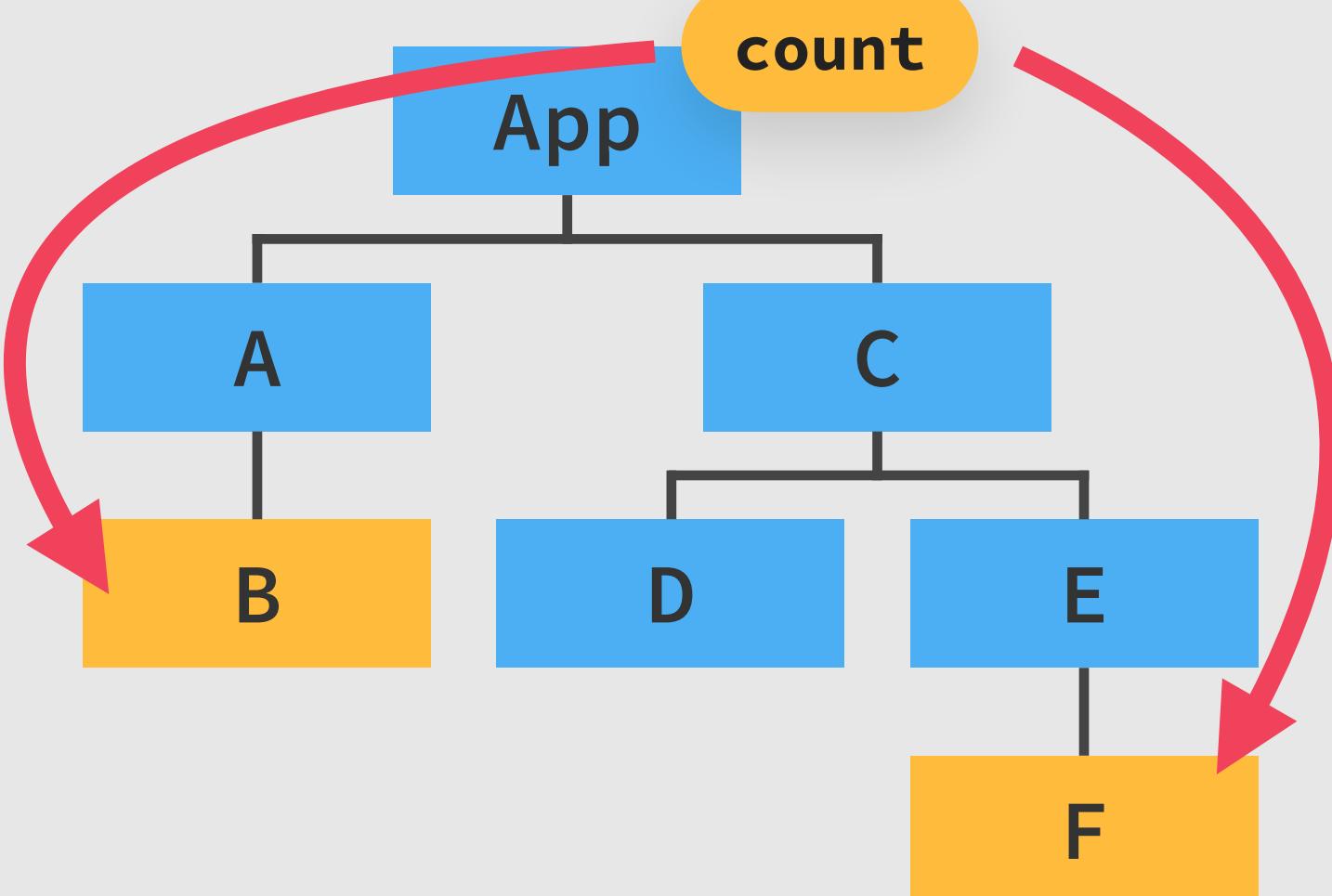
👉 TASK: Passing state into multiple deeply nested child components

## SOLUTION 1: PASSING PROPS

Components  
that need  
count state



## SOLUTION 2: CONTEXT API



🚫 PROBLEM: “PROP DRILLING”

👍 READ STATE FROM EVERYWHERE

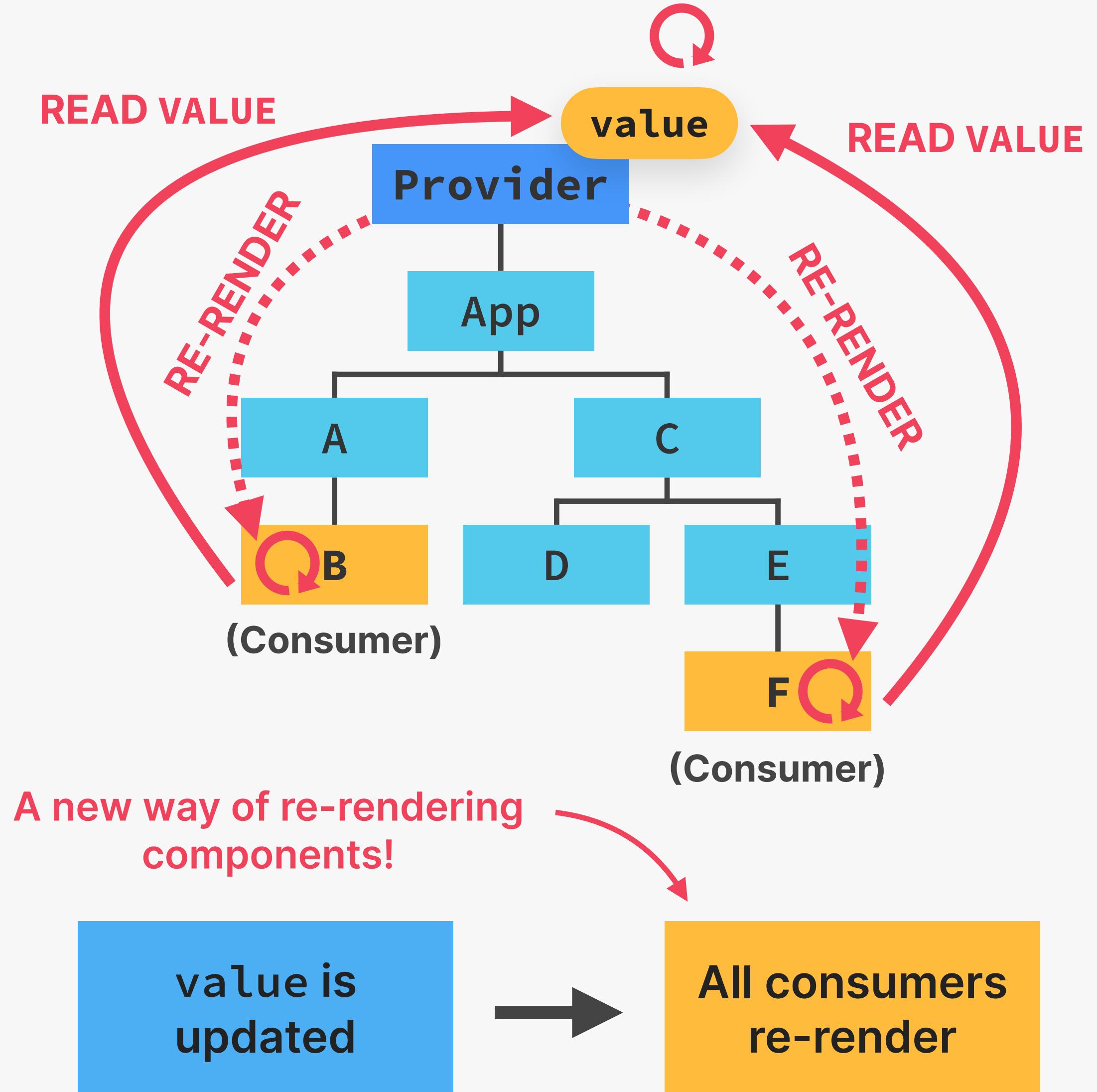


Remember that a good solution to “prop drilling” is better component composition (see “Thinking in React” section)

# WHAT IS THE CONTEXT API?

## CONTEXT API

- 👉 System to pass data throughout the app **without manually passing props down the tree**
  - 👉 Allows us to “broadcast” global state to the entire app
- 1 **Provider:** gives all child components access to value
- 2 **value:** data that we want to make available (usually state and functions)
- 3 **Consumers:** all components that read the provided context value





# REVIEW: WHAT IS STATE MANAGEMENT?



**State management:** Giving each piece of state the right **home**

## ✓ When to use state

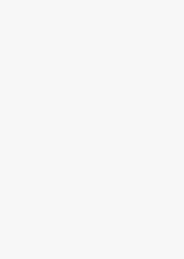
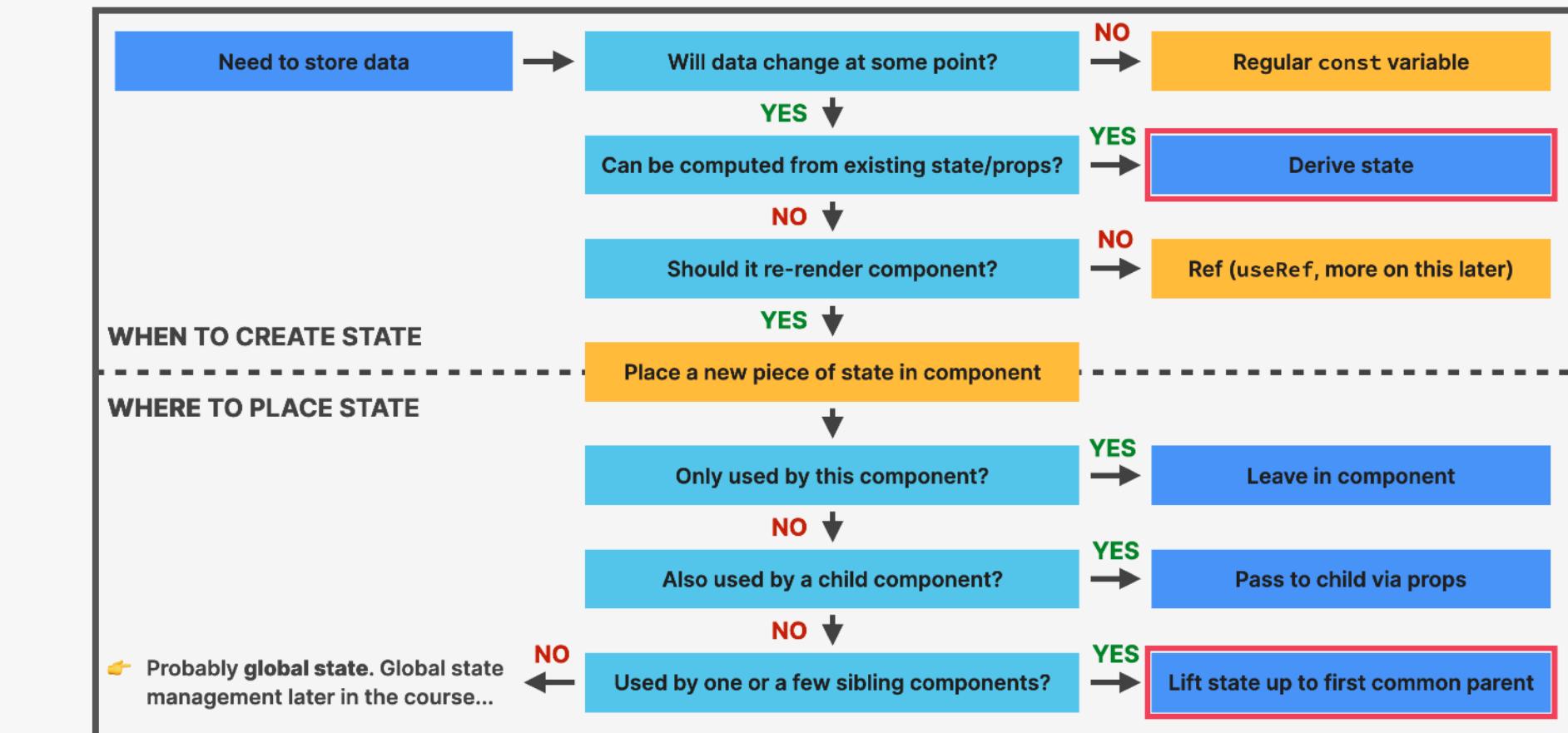
## ✓ Types of state (**accessibility**): local vs. global

## 👉 Types of state (**domain**): UI vs. remote

## 👉 Where to place each piece of state

## 👉 Tools to manage all types of state

This lecture!



From Lecture “Fundamentals of State Management”. You can keep using this 😊

# TYPES OF STATE

1

## STATE ACCESSIBILITY

*"If this component was rendered twice, should a state update in one of them reflect in the other one?"*



### LOCAL STATE

VS.



### GLOBAL STATE

- 👉 Needed only by **one or few components**
- 👉 Only accessible in **component and child components**

- 👉 Might be needed by **many components**
- 👉 Accessible to **every component** in the application

2

## STATE DOMAIN



### REMOTE STATE

VS.



### UI STATE

- 👉 All application data **loaded from a remote server (API)**
- 👉 Usually **asynchronous**
- 👉 Needs re-fetching + updating

- 👉 **Everything else 😅**
- 👉 Theme, list filters, form data, etc.
- 👉 Usually **synchronous** and stored in the application

# STATE PLACEMENT OPTIONS

Where to place state?	TOOLS	WHEN TO USE?
 Local component	useState, useReducer, or useRef	 Local state
 Parent component	useState, useReducer, or useRef	Lifting up state
 Context	Context API + useState or useReducer	Global state (preferably UI state)
 3rd-party library	Redux, React Query, SWR, Zustand, etc.	Global state (remote or UI)
 URL	React Router	Global state, passing between pages
 Browser	Local storage, session storage, etc.	Storing data in user's browser

# STATE MANAGEMENT TOOL OPTIONS



*How to manage different types of state in practice?*

1 STATE DOMAIN

UI STATE

2

REMOTE STATE

Mostly in small applications

1

## STATE ACCESSIBILITY



LOCAL STATE



GLOBAL STATE

- 👉 useState
- 👉 useReducer
- 👉 useRef

- 👉 fetch + useEffect + useState/useReducer

- 👉 Context API + useState/useReducer
- 👉 Redux, Zustand, Recoil, etc.
- 👉 React Router

- 👉 Context API + useState/useReducer
- 👉 Redux, Zustand, Recoil, etc.

- 👉 React Query
- 👉 SWR
- 👉 RTK Query

Tools highly specialized in handling remote state



**PERFORMANCE  
OPTIMIZATION AND  
ADVANCED  
USEFFECT**

# PERFORMANCE OPTIMIZATION TOOLS

1

## PREVENT WASTED RENDERS

- 👉 memo
- 👉 useMemo
- 👉 useCallback
- 👉 Passing elements as children or regular prop

2

## IMPROVE APP SPEED/ RESPONSIVENESS

- 👉 useMemo
- 👉 useCallback
- 👉 useTransition

3

## REDUCE BUNDLE SIZE

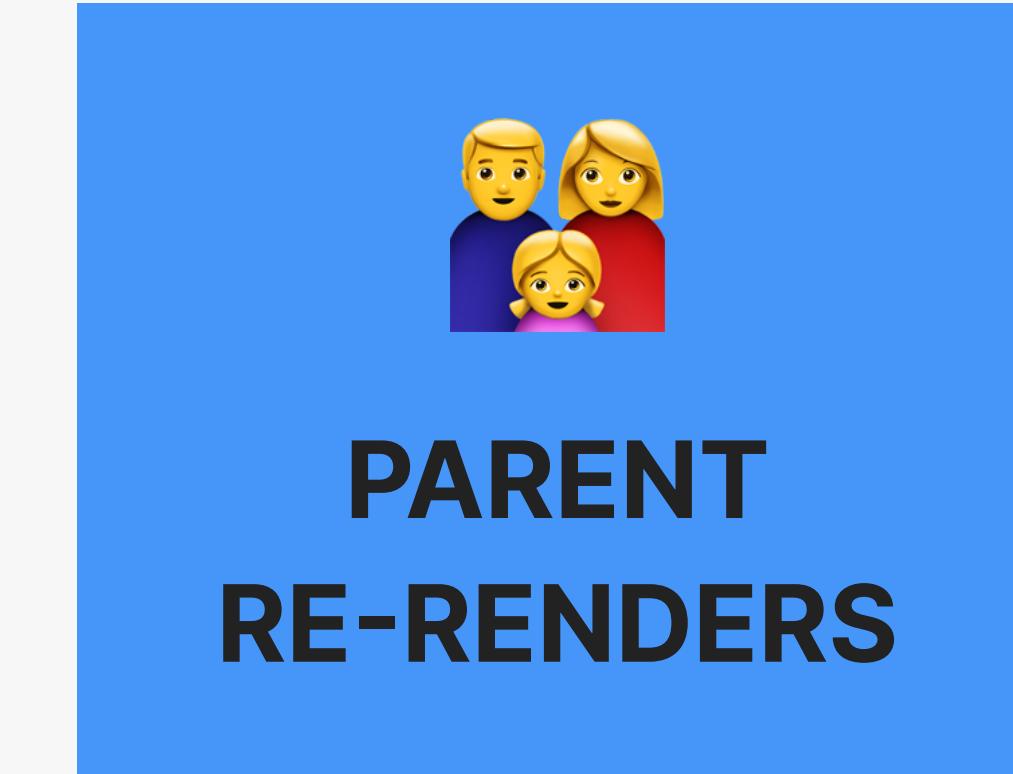
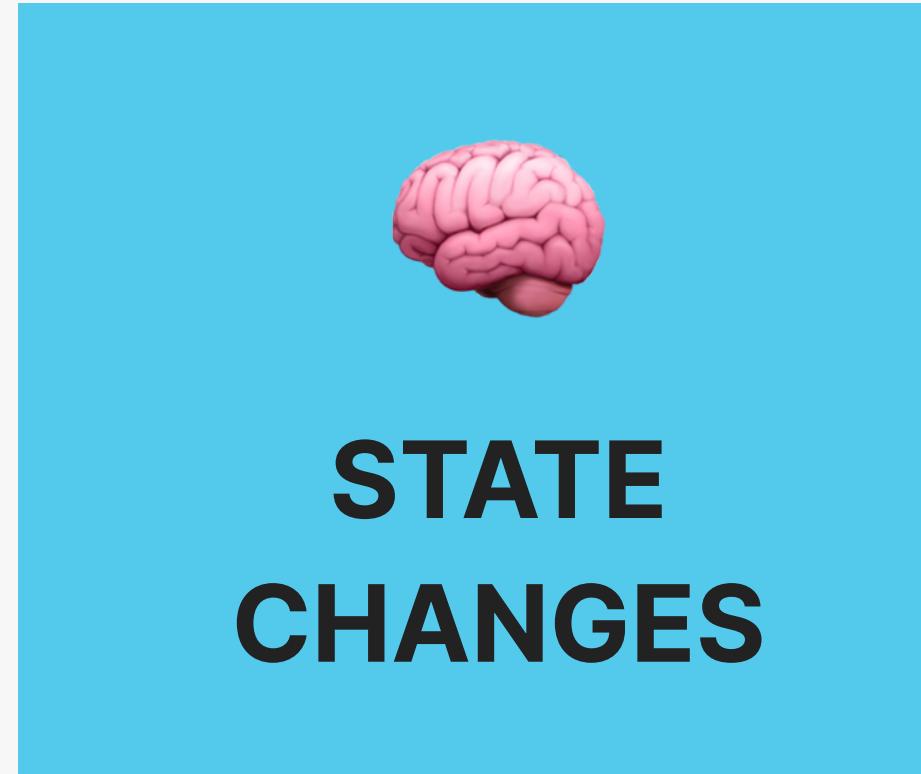
- 👉 Using fewer 3rd-party packages
- 👉 Code splitting and lazy loading



This list of tools and techniques is, by no means, exhaustive. You're already doing many optimizations by following the best practices I have been showing you 🤘

# WHEN DOES A COMPONENTS INSTANCE RE-RENDER?

👉 A component instance only gets re-rendered in 3 different situations:



Creates the false impression that changing props re-renders a component. This is NOT true.



👉 Remember: a render does *not* mean that the DOM actually gets updated, it just means the component function gets called. But this can be an expensive operation.



Usually no problem, as React is very fast!

👉 Wasted render: a render that didn't produce any change in the DOM

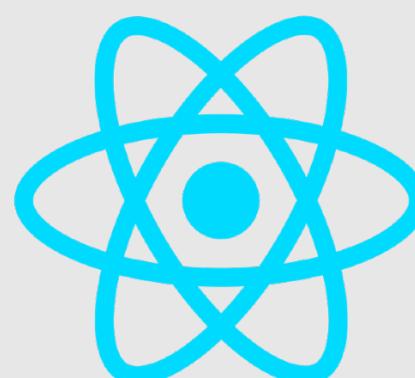
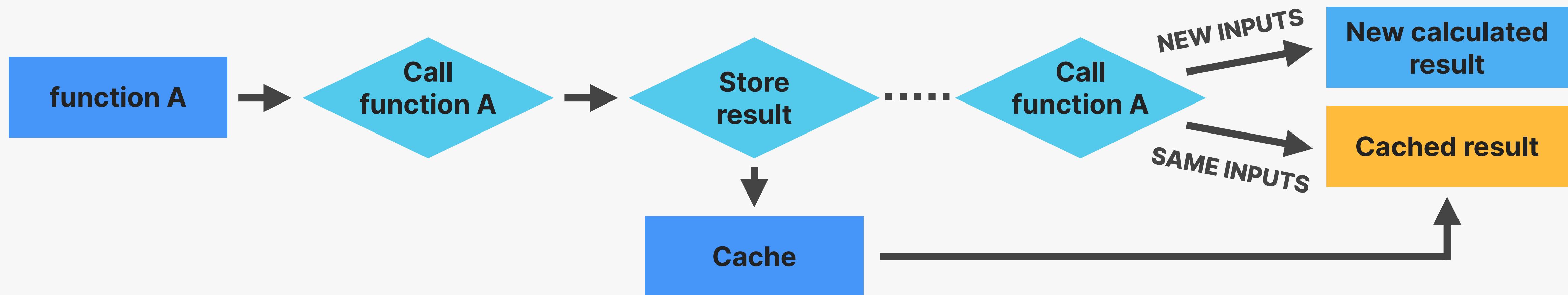


👉 Only a problem when they happen too frequently or when the component is very slow



# WHAT IS MEMOIZATION?

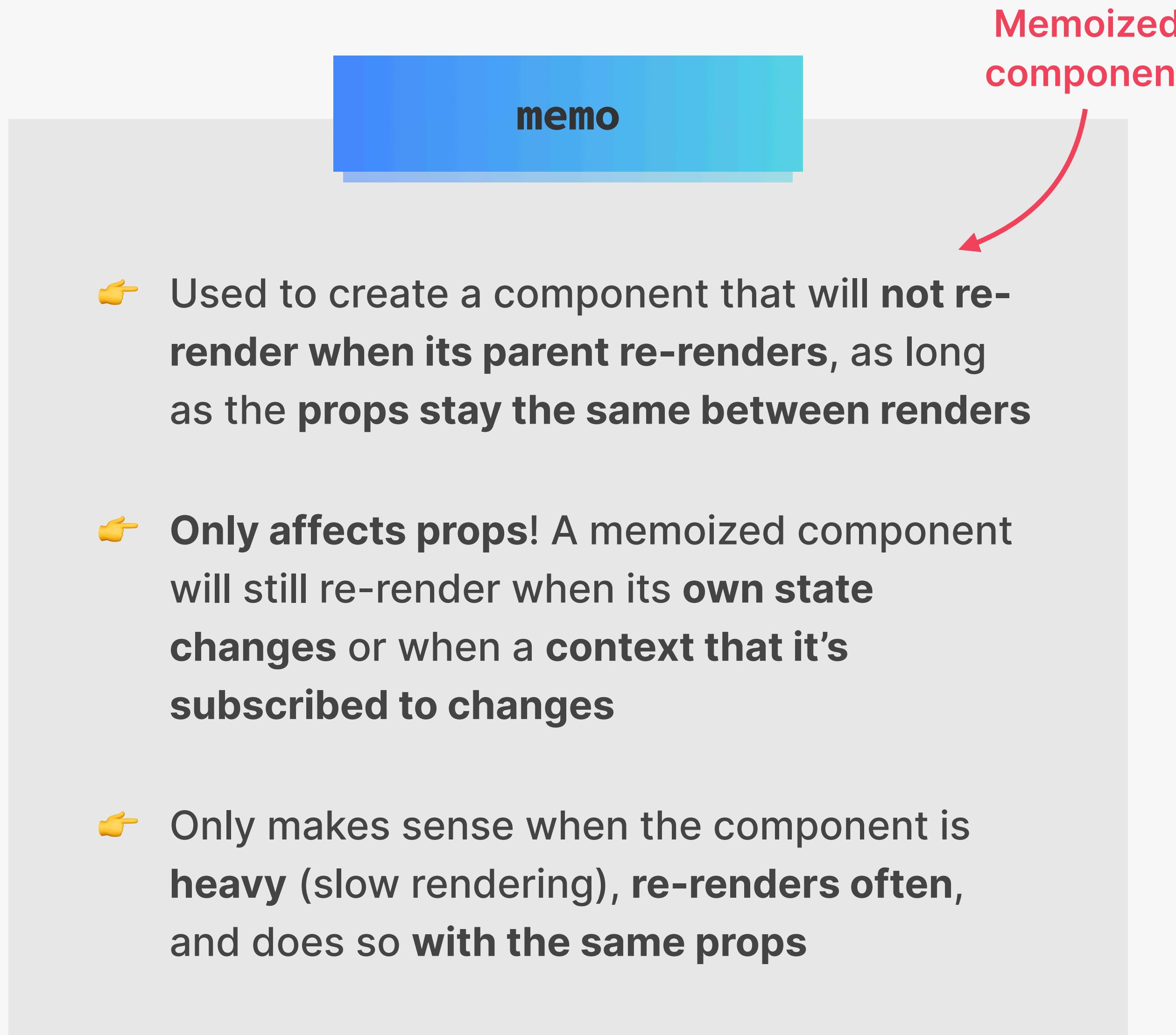
👉 **Memoization:** Optimization technique that executes a pure function once, and saves the result in memory. If we try to execute the function again with the **same arguments as before**, the previously saved result will be returned, **without executing the function again**.



- 👉 Memoize **components** with `memo`
- 👉 Memoize **objects** with `useMemo`
- 👉 Memoize **functions** with `useCallback`

- 1 Prevent wasted renders
- 2 Improve app speed/responsiveness

# THE MEMO FUNCTION



## REGULAR BEHAVIOR (NO MEMO)

Components  
re-renders



Child  
re-renders

## MEMOIZED CHILD WITH MEMO

Components  
re-renders



Memoized child  
does **NOT**  
re-render



Memoized child  
re-renders



# AN ISSUE WITH MEMO

In React, everything is **re-created on every render** (including objects and functions)



In JavaScript, two objects or functions that look the same, are **actually different** (`{ } != { }`)

**THEREFORE** A dark gray downward-pointing arrow.

If objects or functions are passed as props, the child component will always see them as **new props on each re-render**



**If props are different between re-renders, *memo* will not work**

**SOLUTION** A dark gray downward-pointing arrow.

We need to **memoize objects and functions, to make them stable (preserve) between re-renders** (`memoized {} == memoized {}`)

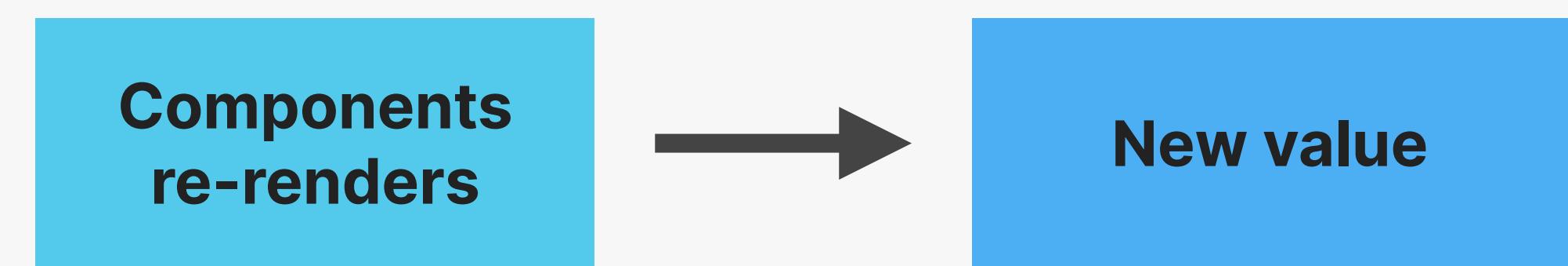
# TWO NEW HOOKS: USEMEMO AND USECALLBACK

## useMemo AND useCallback

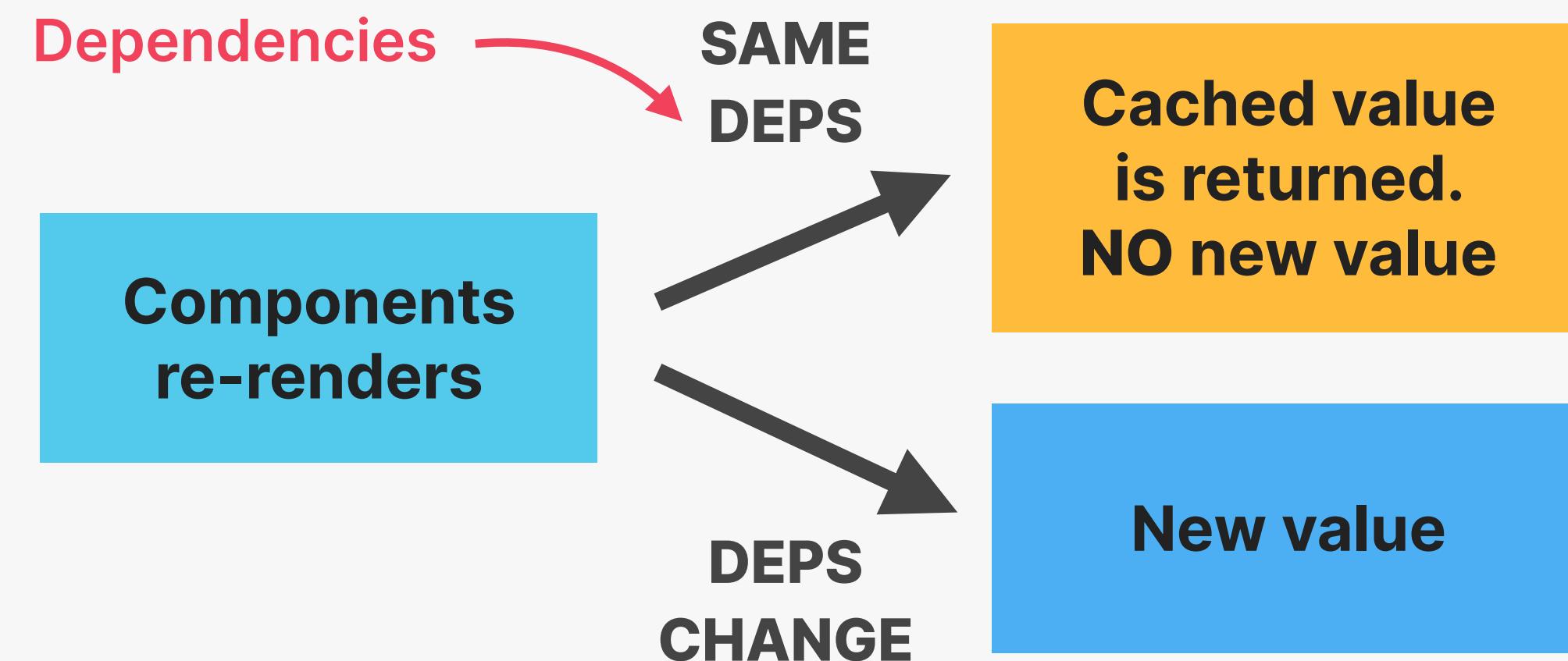
- 👉 Used to memoize values (`useMemo`) and functions (`useCallback`) between renders
- 👉 Values passed into `useMemo` and `useCallback` will be stored in memory ("cached") and **returned in subsequent re-renders, as long as dependencies ("inputs") stay the same**
- 👉 `useMemo` and `useCallback` have a **dependency array** (like `useEffect`): whenever one **dependency changes**, the value will be re-created



### REGULAR BEHAVIOR (NO USEMEMO)



### MEMOIZING A VALUE WITH USEMEMO



# TWO NEW HOOKS: USEMEMO AND USECALLBACK

## useMemo AND useCallback

- 👉 Used to memoize values (`useMemo`) and functions (`useCallback`) between renders
- 👉 Values passed into `useMemo` and `useCallback` will be stored in memory ("cached") and **returned in subsequent re-renders, as long as dependencies ("inputs") stay the same**
- 👉 `useMemo` and `useCallback` have a **dependency array** (like `useEffect`): whenever one **dependency changes**, the value will be **re-created**
- 👉 Only use them for one of the three **use cases**!

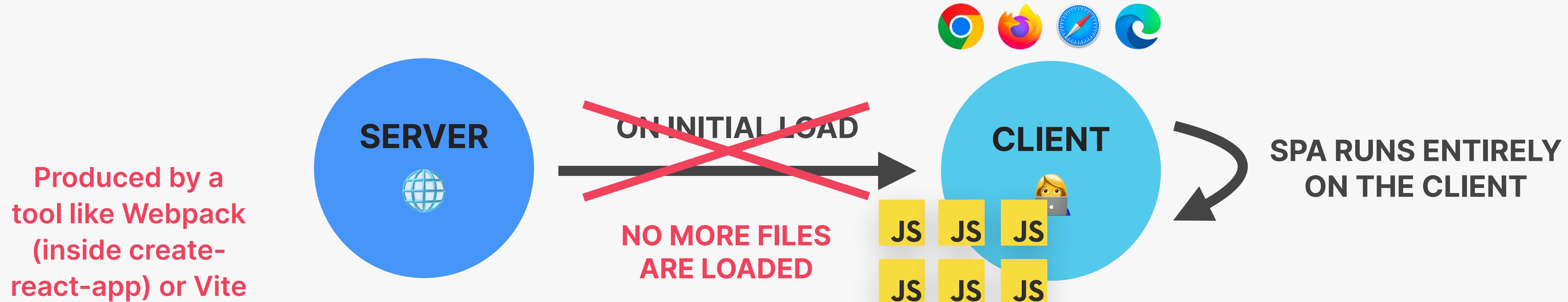
## THREE BIG USES CASES:

- 1 Memoizing props to prevent wasted renders (together with `memo`)
- 2 Memoizing values to avoid expensive re-calculations on every render
- 3 Memoizing values that are used in dependency array of another hook

For example to  
avoid infinite  
`useEffect` loops



# THE BUNDLE AND CODE SPLITTING



- 👉 **Bundle:** JavaScript file containing the **entire application code**. Downloading the bundle will load **the entire app at once**, turning it into a SPA
- 👉 **Bundle size:** Amount of JavaScript users have to download to start using the app. One of the most important things to be optimized, so that the bundle takes **less time to download**
- 👉 **Code splitting:** Splitting bundle into multiple parts that can be **downloaded over time** ("lazy loading")



# DON'T OPTIMIZE PREMATURELY!

DO

- ✓ Find performance bottlenecks using the Profiler and visual inspection (laggy UI)
- ✓ Fix those real performance issues
- ✓ Memoize expensive re-renders
- ✓ Memoize expensive calculations
- ✓ Optimize context if it has many consumers and changes often
- ✓ Memoize context value + child components
- ✓ Implement code splitting + lazy loading for SPA routes

DON'T!

- 🚫 Don't optimize prematurely!
- 🚫 Don't optimize anything if there is nothing to optimize...
- 🚫 Don't wrap all components in memo()
- 🚫 Don't wrap all values in useMemo()
- 🚫 Don't wrap all functions in useCallback()
- 🚫 Don't optimize context if it's not slow and doesn't have many consumers



# USEEFFECT DEPENDENCY ARRAY RULES

## DEPENDENCY ARRAY RULES

- 👉 Every state variable, prop used inside the effect **MUST** be included in the dependency array
- 👉 All “reactive values” must be included! That means any function or variable that reference any other reactive value
- 👉 Dependencies choose themselves: **NEVER** ignore the exhaustive-deps ESLint rule!
- 👉 Do NOT use objects or arrays as dependencies (objects are recreated on each render, and React sees new objects as different, `{}` `!==` `{}`)

Reactive value: state, prop, or context value, or any other value that *references* a reactive value

```
const [number, setNumber] = useState(5);
const [duration, setDuration] = useState(0);
const mins = Math.floor(duration);
const secs = (duration - mins) * 60;

const formatDur = function () {
  return `${mins}:${secs < 10 ? '0' : ''}${secs}`;
};

useEffect(
  function () {
    document.title =
      `${number}-exercise workout ${formatDur()}`;
  },
  [number, formatDur]
);
```

All reactive values used in effect

- 👉 The same rules apply to the dependency arrays of other hooks: useMemo and useCallback

# REMOVING UNNECESSARY DEPENDENCIES



## MOVING FUNCTION DEPENDENCIES

- 👉 Move function **into the effect**
- 👉 If you need the function in multiple places, **memoize it** (`useCallback`)
- 👉 If the function doesn't reference any reactive values, move it **out of the component**



## MOVING OBJECT DEPENDENCIES

- 👉 Instead of including the entire object, include **only the properties you need** (primitive values)
- 👉 If that doesn't work, use the same strategies as for functions (**moving** or **memoizing** object)



## OTHER STRATEGIES

- 👉 If you have **multiple related reactive values** as dependencies, try using a **reducer** (`useReducer`)
- 👉 You don't need to include `setState` (from `useState`) and `dispatch` (from `useReducer`) in the dependencies, as **React guarantees them to be stable** across renders

# WHEN NOT TO USE AN EFFECT



Effects should be used as a **last resort**, when no other solution makes sense. React calls them an “escape hatch” to step outside of React

## THREE CASES WHERE EFFECTS ARE OVERUSED:

1

**Responding to a user event.** An event handler function should be used instead

2

**Fetching data on component mount.** This is fine in small apps, but in real-world app, a library like React Query should be used

3

**Synchronizing state changes with one another** (setting state based on another state variable). Try to use derived state and event handlers

Avoid these as a beginner

We actually do this in the current project, but for a good reason 😊



# **REDUX AND MODERN REDUX TOOLKIT (WITH THUNKS)**

# WHAT IS REDUX?

REDUX

- 👉 3rd-party library to manage **global state**
- 👉 **Standalone** library, but easy to integrate with React apps using `react-redux` library
- 👉 All global state is stored in one **globally accessible store**, which is easy to update using “actions” (like `useReducer`)
- 👉 It’s conceptually similar to using the Context API + `useReducer`
- 👉 Two “versions”: (1) Classic Redux, (2) Modern Redux Toolkit

We will learn both 😎



Redux

Global store is updated



All consuming components re-render



You need to have a really good understanding of the `useReducer` hook in order to understand Redux!

# DO YOU NEED TO LEARN REDUX?

👉 Historically, Redux was used in most React apps for all global state. Today, that has changed, because there are many alternatives. **Many apps don't need Redux anymore**, unless they need a lot of global UI state.

*You might not need to learn Redux...*

## 🤔 WHY LEARN REDUX IN THIS COURSE?

1

Redux can be hard to learn, and this course teaches it well 😅

2

You will encounter Redux code in your job, so you should understand it

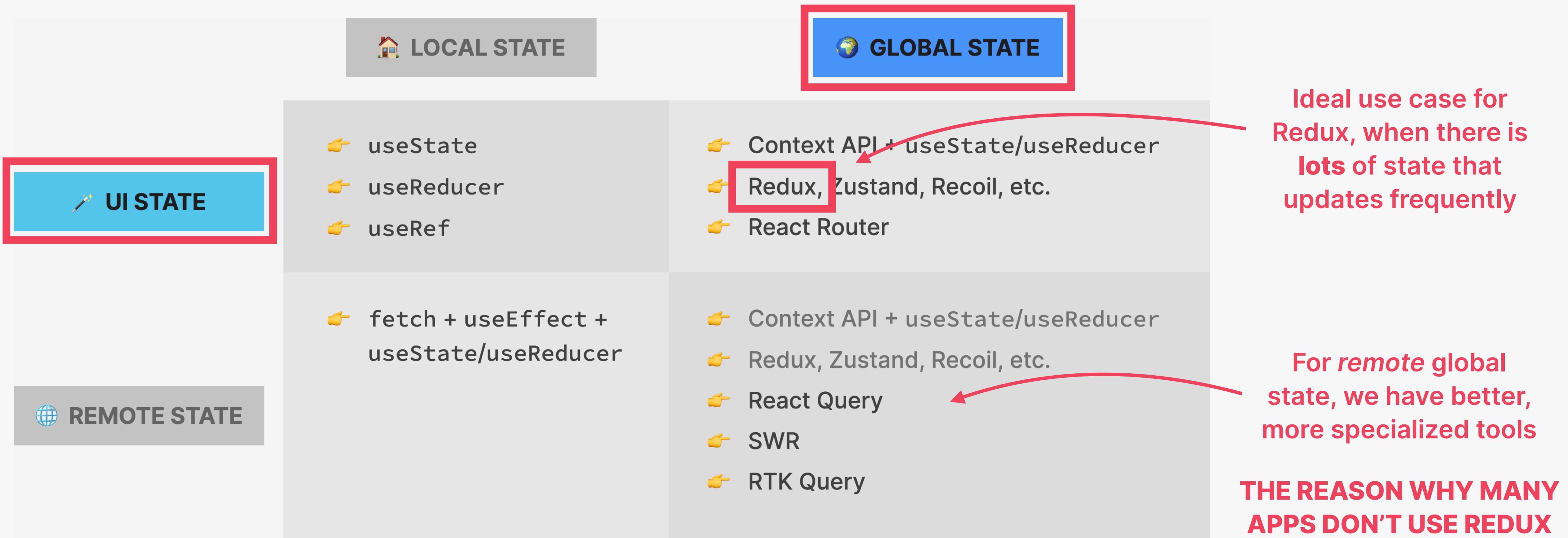
3

Some apps do require Redux (or a similar library)

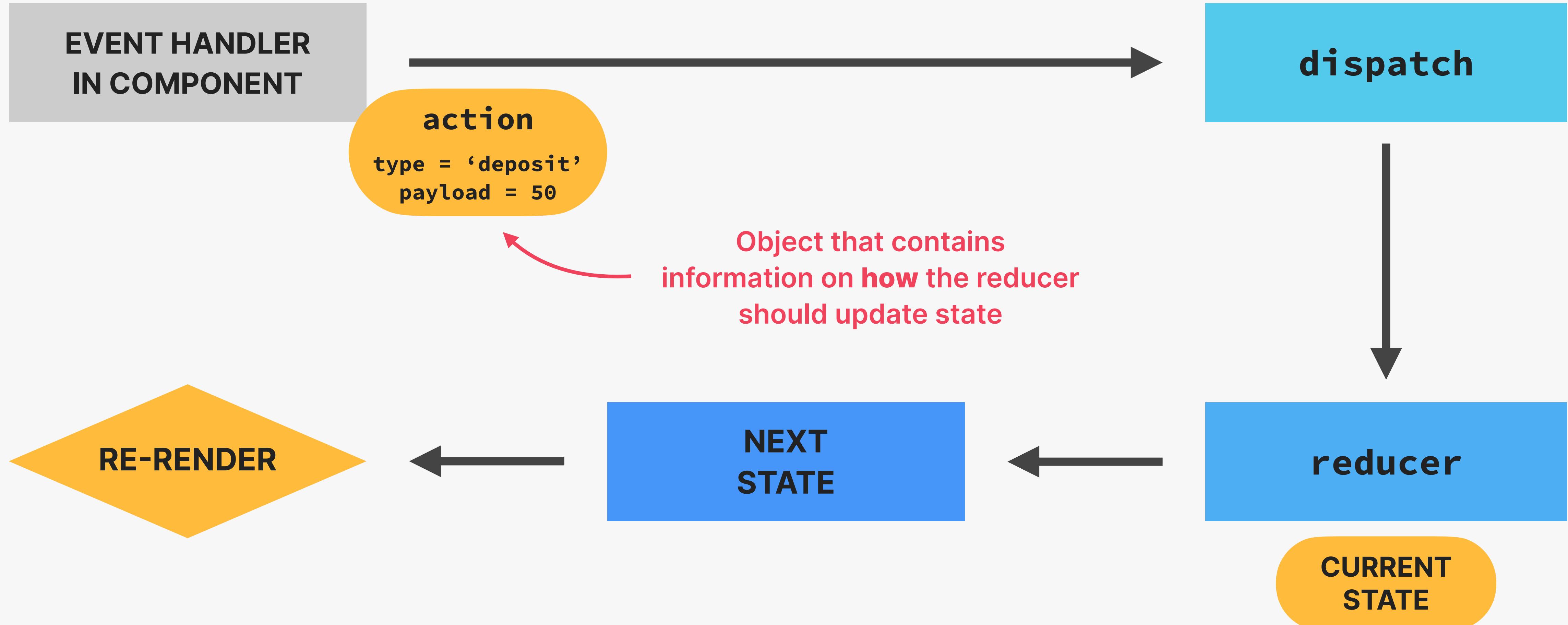
# REDUX USE CASES



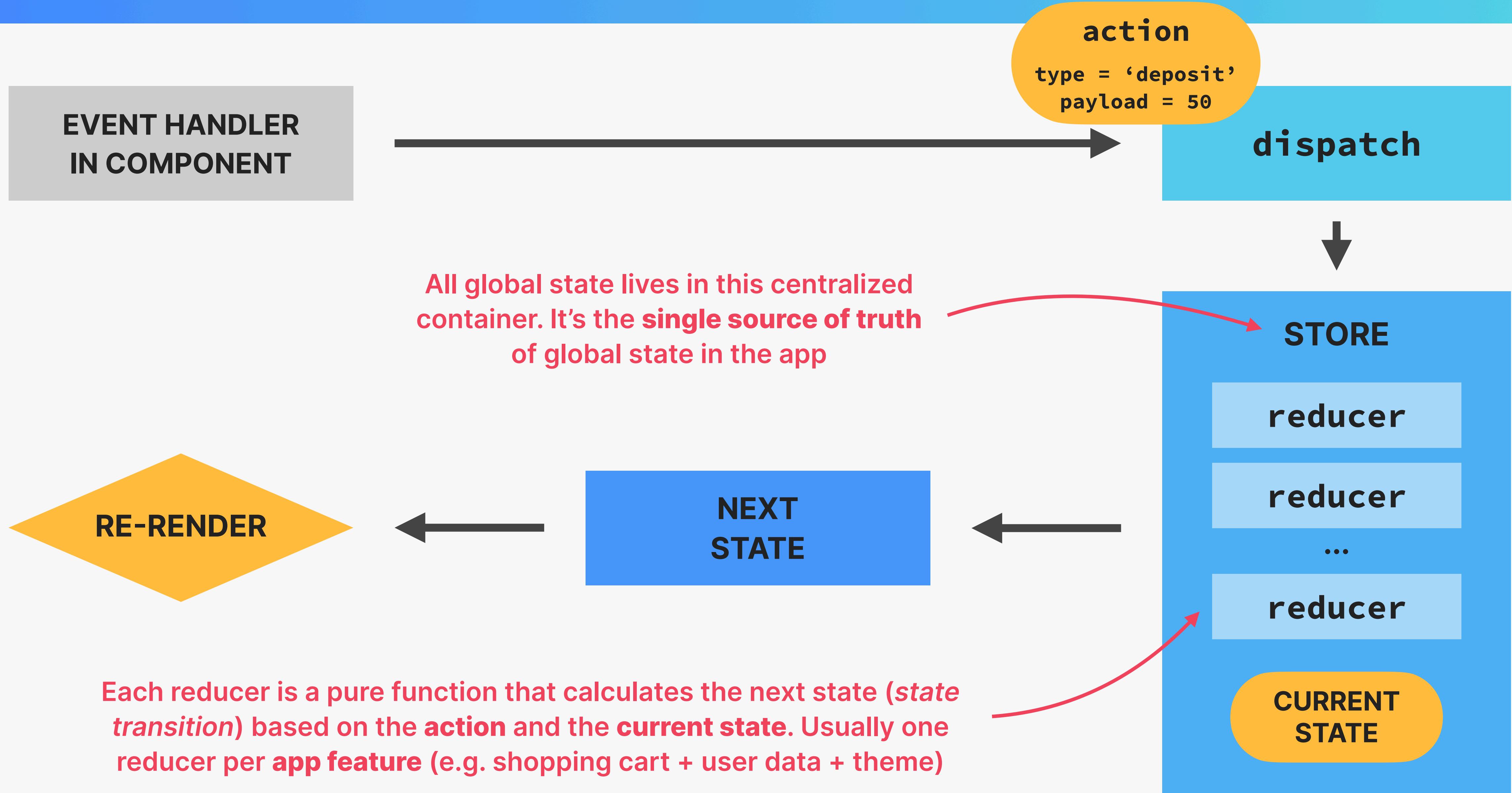
Historically, Redux was used in most React apps for all global state. Today, that has changed, because there are many alternatives. Many apps don't need Redux anymore, unless they need a lot of global UI state.



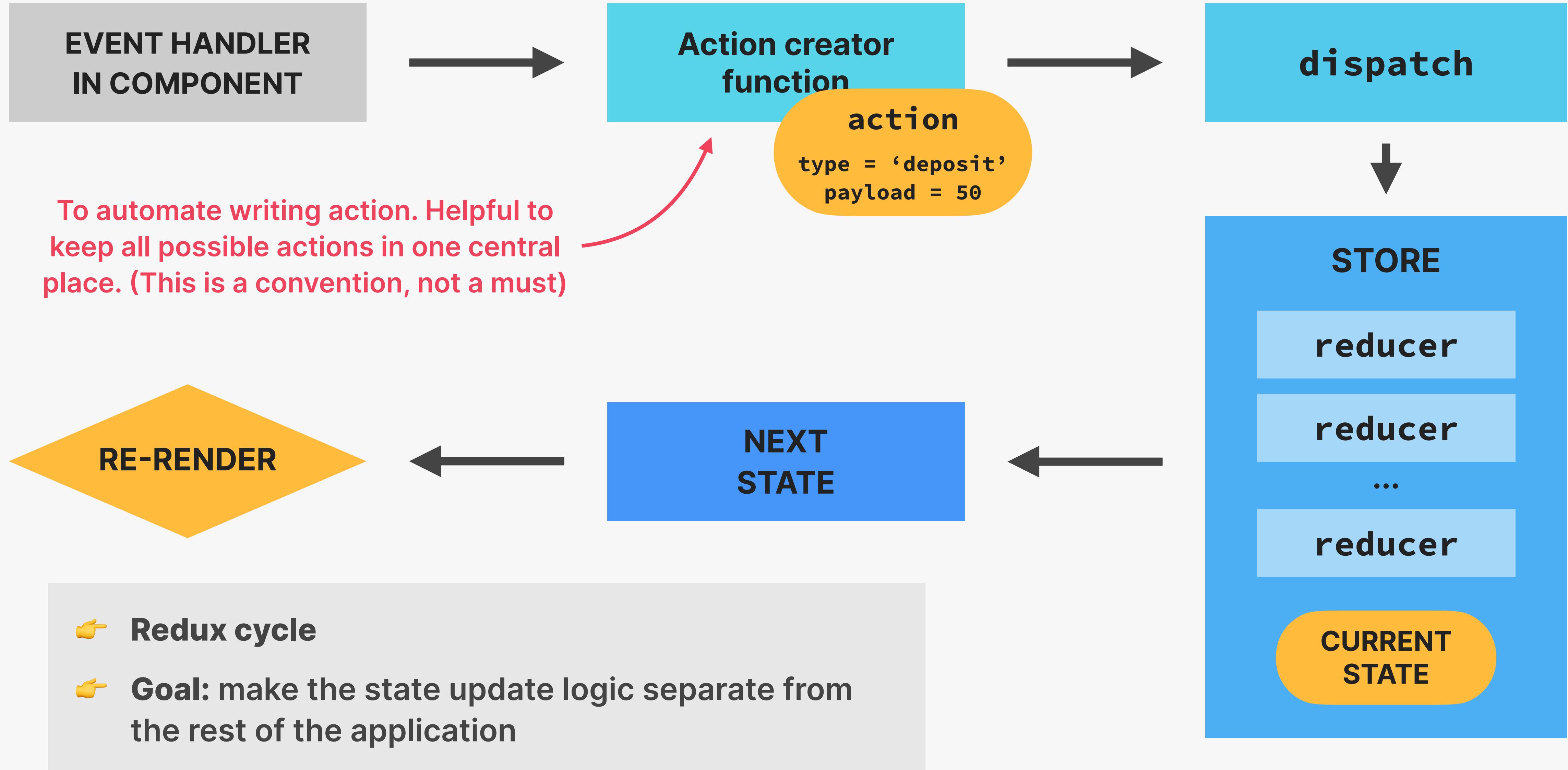
# THE MECHANISM OF THE USERREDUCER HOOK



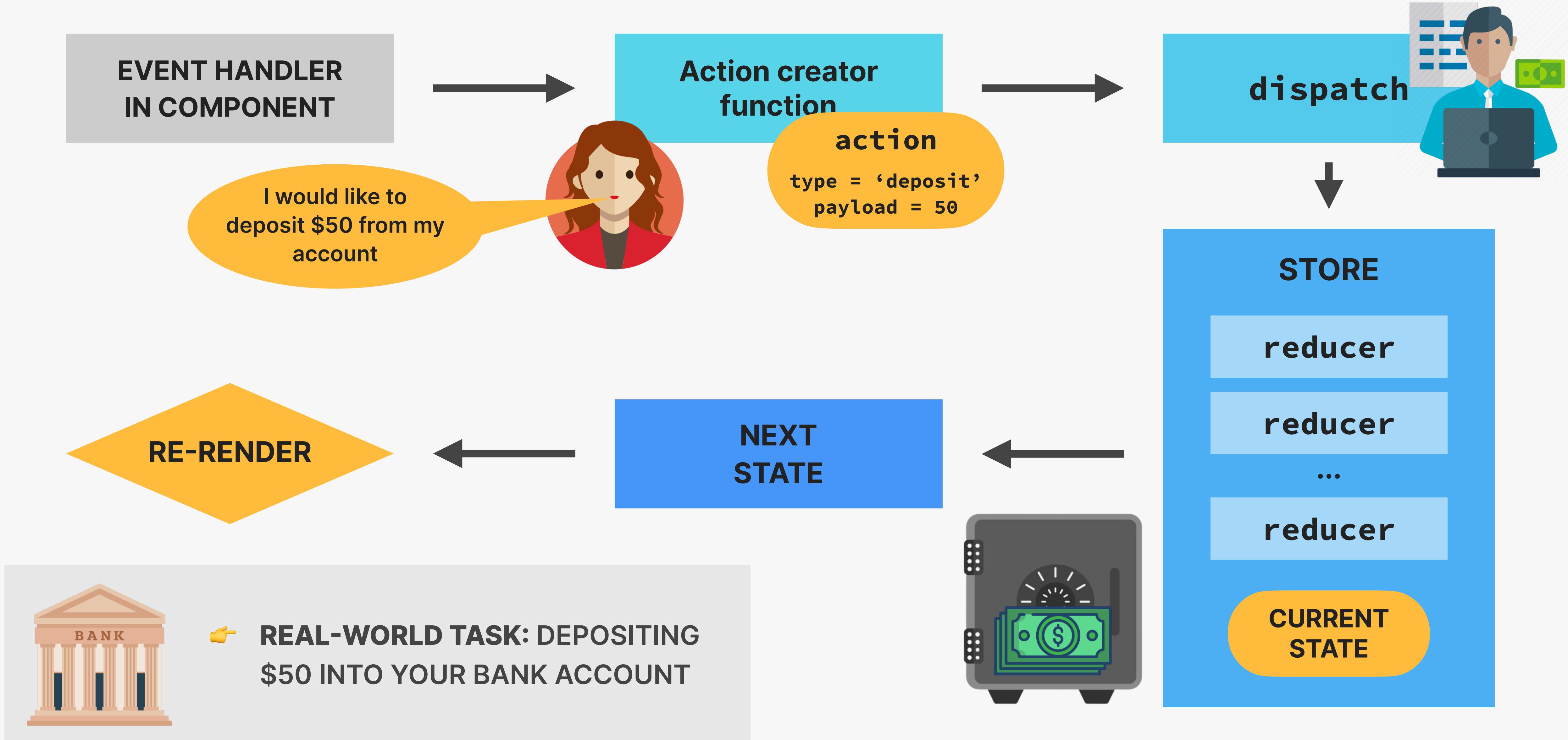
# THE MECHANISM OF REDUX



# THE MECHANISM OF REDUX



# THE MECHANISM OF REDUX



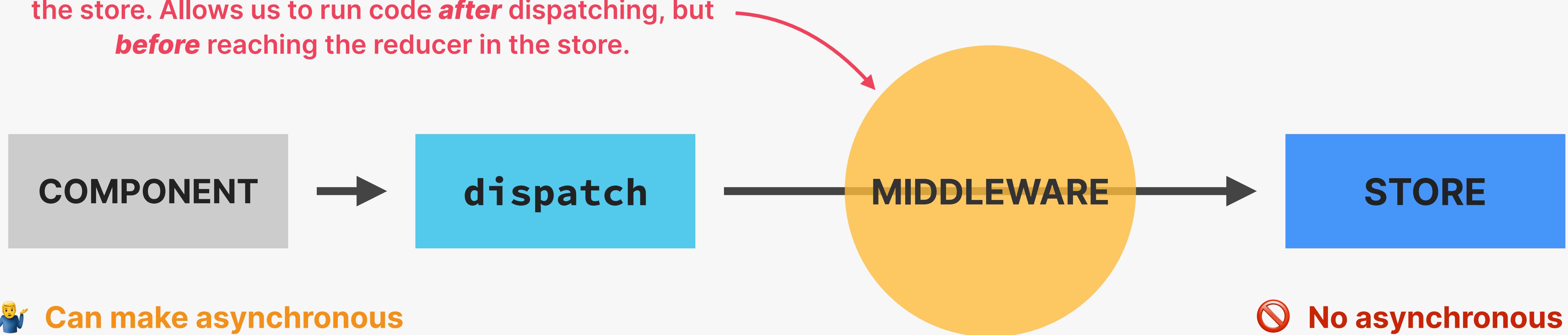


# WHAT IS REDUX MIDDLEWARE?



Where to make an **asynchronous API call** (or any other async operation) in Redux?

A function that sits between dispatching the action and the store. Allows us to run code **after** dispatching, but **before** reaching the reducer in the store.



💡 Can make asynchronous operations and then dispatch

💡 Fetching data in components is not ideal

👍 Perfect for asynchronous code

👍 API calls, timers, logging, etc.

👍 The place for side effects

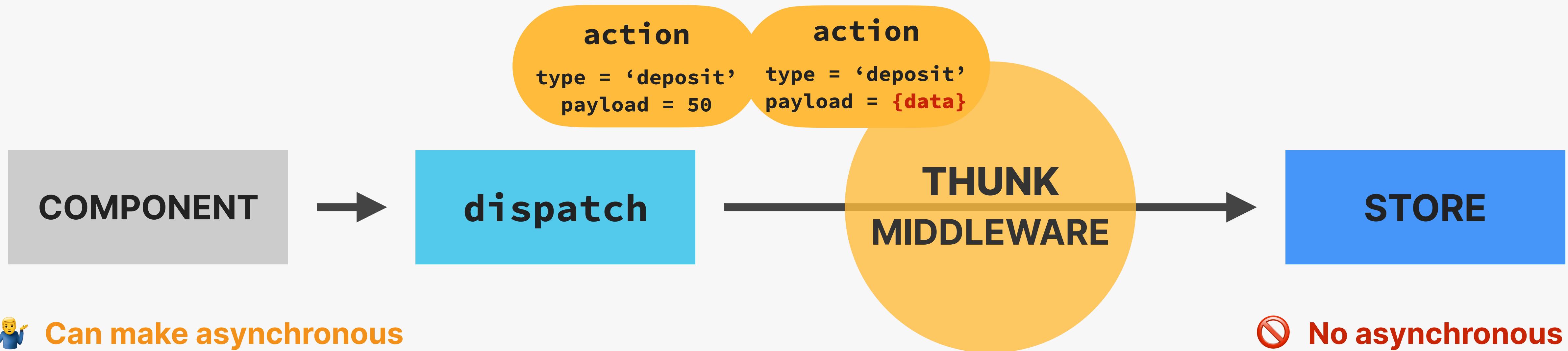
🚫 No asynchronous operations

🚫 Reducers need to be pure functions

# REDUX THUNKS



Where to make an **asynchronous API call** (or any other async operation) in Redux?



💡 **Can make asynchronous operations and then dispatch**

💡 **Fetching data in components is not ideal**

👍 **Perfect for asynchronous code**

👍 **API calls, timers, logging, etc.**

👍 **The place for side effects**

🚫 **No asynchronous operations**

🚫 **Reducers need to be pure functions**



# WHAT IS REDUX TOOLKIT?

## REDUX TOOLKIT

- 👉 The **modern and preferred** way of writing Redux code
- 👉 An **opinionated** approach, forcing us to use Redux best practices
- 👉 100% compatible with “classic” Redux, allowing us to **use them together**
- 👉 Allows us to **write a lot less code** to achieve the same result (less “boilerplate”)
- 👉 Gives us 3 big things (but there are many more...):
  - 1 We can write code that “**mutates**” state inside reducers (will be converted to **immutable** logic behind the scenes by “Immer” library)
  - 2 Action creators are **automatically** created
  - 3 **Automatic** setup of thunk middleware and DevTools



# CONTEXT API VS. REDUX

## CONTEXT API + useReducer

- 👍 Built into React
- 👍 Easy to set up a **single context**
- 👎 Additional state “slide” requires new context **set up from scratch** (“provider hell” in App.js)
- 👎 No mechanism for async operations
- 👎 Performance optimization is a **pain**
- 👎 Only React DevTools

## REDUX

- 👎 Requires additional package (larger bundle size)
- 👎 More work to set up **initially**
- 👍 Once set up, it’s easy to create **additional state “slices”**
- 👍 Supports **middleware** for async operations
- 👍 Performance is optimized **out of the box**
- 👍 Excellent DevTools

Keep in mind that we should **not** use  
these solutions for **remote state**

# WHEN TO USE CONTEXT API OR REDUX?

## CONTEXT API + useReducer

*“Use the Context API for global state management in small apps”*

- 👉 When you just need to share a value that **doesn't change often** [Color theme, preferred language, authenticated user, ...]
- 👉 When you need to solve a simple **prop drilling** problem
- 👉 When you need to manage state in a **local sub-tree** of the app

These are not super common in UI state

- 👉 When you have lots of global UI state that needs to be **updated frequently** (because *Redux is optimized for this*) [Shopping cart, current tabs, complex filters or search, ...]
- 👉 When you have **complex state** with nested objects and arrays (*because you can mutate state with Redux Toolkit*)

For example in the compound component pattern



There is no right answer that fits every project. It all depends on the project needs!



# PART 04

---

# PROFESSIONAL REACT DEVELOPMENT

# REACT ROUTER WITH DATA LOADING (v6.4+)

# THE PROJECT: 🍕 FAST REACT PIZZA CO.

REMEMBER OUR VERY FIRST PROJECT?

— FAST REACT PIZZA CO. —

**OUR MENU**

Authentic Italian cuisine. 6 creative dishes to choose from. All from our stone oven, all organic, all delicious.

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

We're open until 22:00. Come visit us or order online.

[Order now](#)



🍕 FAST REACT PIZZA CO.

👉 Now the same restaurant (business) needs a simple way of allowing customers to **order pizzas and get them delivered to their home**

👉 We were hired to build the application front-end 



# HOW TO PLAN AND BUILD A REACT APPLICATION

FROM THE EARLIER “THINKING IN REACT” LECTURE:

- 1 Break the desired UI into **components**
- 2 Build a **static** version (no state yet)
- 3 Think about **state management + data flow**



- 👉 This works well for small apps with **one page and a few features**
- 👉 In **real-world apps**, we need to adapt this process



# HOW TO PLAN AND BUILD A REACT APPLICATION

1

Gather application **requirements and features**

2

Divide the application into **pages**

👉 Think about the **overall** and **page-level UI**

👉 Break the desired UI into **components** ← **From earlier**

👉 Design and build a **static** version (no state yet) ← **From earlier**

3

Divide the application into **feature categories**

👉 Think about **state management + data flow** ← **From earlier**

4

Decide on what **libraries** to use (technology decisions)

This is just a rough overview. In the real-world, things are never this linear



# PROJECT REQUIREMENTS FROM THE BUSINESS

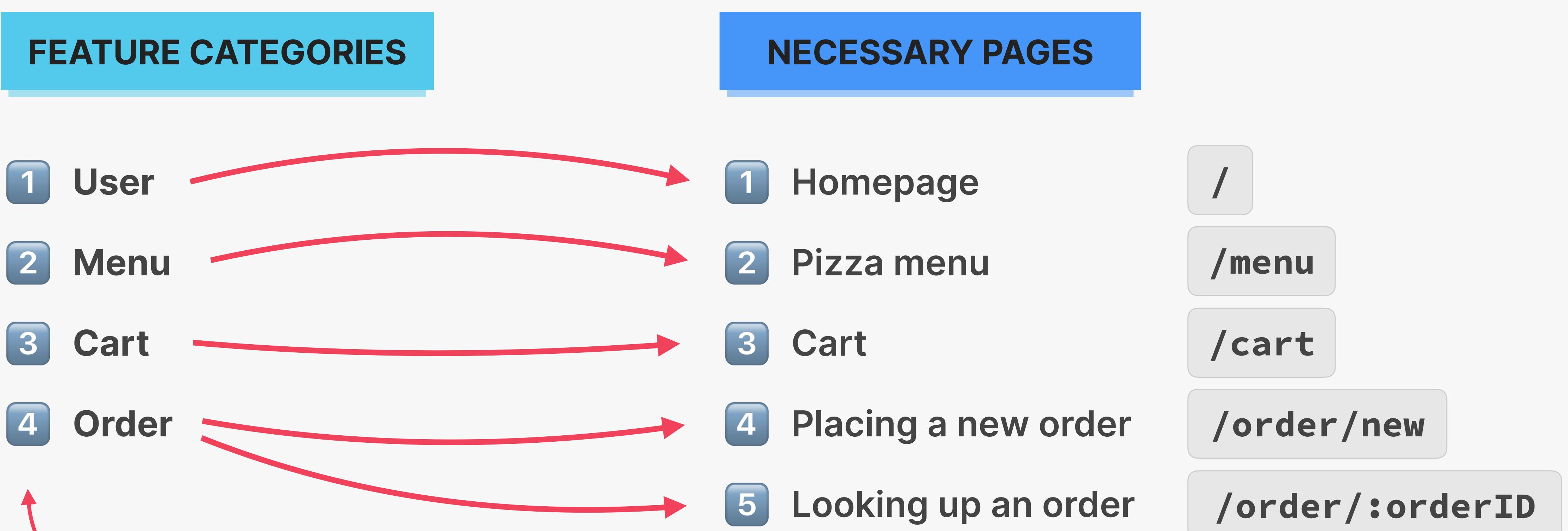
STEP 1

- 👉 Very simple application, where users can order **one or more pizzas from a menu**
- 👉 Requires **no user accounts** and no login: users just input their names before using the app
- 👉 The pizza menu can change, so it should be **loaded from an API** ✓ DONE
- 👉 Users can add multiple pizzas to a **cart** before ordering
- 👉 Ordering requires just the **user's name, phone number, and address**
- 👉 If possible, **GPS location** should also be provided, to make delivery easier
- 👉 User's can mark their order as “**priority**” for an additional 20% of the cart price
- 👉 Orders are made by **sending a POST request** with the order data (user data + selected pizzas) to the API
- 👉 Payments are made on delivery, so **no payment processing** is necessary in the app
- 👉 Each order will get a **unique ID** that should be displayed, so the **user can later look up their order** based on the ID
- 👉 Users should be able to mark their order as “**priority**” order **even after it has been placed**

From these requirements, we can understand the features we need to implement

# FEATURES + PAGES

STEP 2 + 3



All features can be placed into one of these. So this is what the app will essentially be about

# STATE MANAGEMENT + TECHNOLOGY DECISIONS

STEP 3 + 4

STATE  
“DOMAINS” /  
“SLICES”

These usually map  
quite nicely to the  
app features

👉 Routing

👉 Styling

👉 Remote state  
management

👉 UI State  
management

- 1 User → Global UI state (*no accounts, so stays in app*)
- 2 Menu → Global remote state (*menu is fetched from API*)
- 3 Cart → Global UI state (*no need for API, just stored in app*)
- 4 Order → Global remote state (*fetched and submitted to API*)

TYPES OF  
STATE

This is just one of many tech  
stacks we could have chosen

• React Router

*The standard for React SPAs*

tailwindcss

*Trendy way of styling applications that we want to learn*

• React Router

*New way of fetching data right inside React Router (v6.4+) that is worth exploring (“render-as-you-fetch” instead of “fetch-on-render”). Not really state management, as it doesn’t persist state.*

Redux

*State is fairly complex. Redux has many advantages for UI state. Also, we want to practice Redux a bit more*



# TAILWIND CSS CRASH COURSE: STYLING THE APP

# WHAT IS TAILWIND CSS?

## TAILWIND CSS

- 👉 “A *utility-first CSS framework packed with utility classes like flex, text-center and rotate-90 that can be composed to build any design, directly in your markup (HTML or JSX)*”
- 👉 **Utility-first CSS approach:** writing tiny classes with one single purpose, and then combining them to build entire layouts
- 👉 In tailwind, **these classes are already written for us.** So we’re not gonna write any new CSS, but instead use some of tailwind’s hundreds of classes



# THE GOOD AND BAD ABOUT TAILWIND

## THE GOOD

These two are enough  
to give tailwind a try!



- 👍 You don't need to think about class names
- 👍 No jumping between files to write markup and styles
- 👍 Immediately understand styling in any project that uses tailwind
- 👍 Tailwind is a design system: many design decisions have been taken for you, which makes UIs look better and more consistent
- 👍 Saves a lot of time, e.g. on responsive design
- 👍 Docs and VS Code integration are great

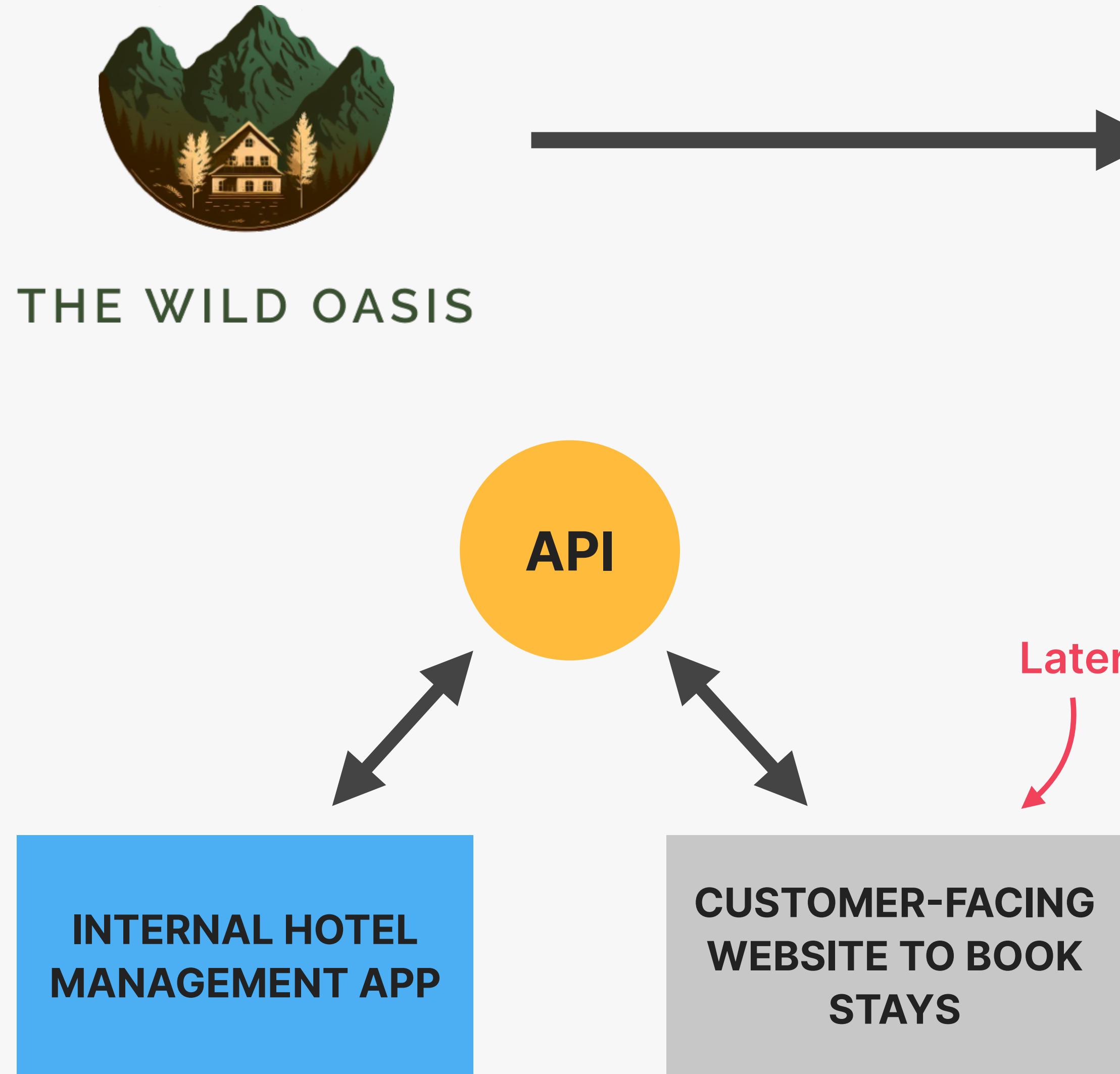
## THE BAD

- 👎 Markup (HTML or JSX) looks very unreadable, with lots of class names (*you get used to it*)
  - 👎 You have to learn a lot of class names (*but after a day of usage you know fundamentals*)
  - 👎 You need to install and set up tailwind on each new project
  - 👎 You're giving up on "vanilla CSS" 😢
- 👏 Many people love to hate on tailwind for no reason. Please don't be that person! Try it before judging 🙏



**SETTING UP OUR  
BIGGEST PROJECT +  
STYLED  
COMPONENTS**

# THE PROJECT: THE WILD OASIS



- 👉 “The Wild Oasis” is a small boutique **hotel** with 8 luxurious wooden cabins
- 👉 They need a custom-built application to manage everything about the hotel: **bookings, cabins and guests**
- 👉 This is the **internal application** used inside the hotel to **check in guests as they arrive**
- 👉 They have nothing right now, so they **also need the API**
- 👉 Later they will probably want a **customer-facing website** as well, where customers will be able to **book stays**, using the same API

# REVIEW: HOW TO PLAN A REACT APPLICATION

- 1 Gather application **requirements and features**
- 2 Divide the application into **pages**
- 3 Divide the application into **feature categories**
- 4 Decide on what **libraries** to use (technology decisions)



# PROJECT REQUIREMENTS FROM THE BUSINESS

- 👉 Users of the app are hotel employees. They need to be logged into the application to perform tasks
- 👉 New users can only be signed up inside the applications (to guarantee that only actual hotel employees can get accounts)
- 👉 Users should be able to upload an avatar, and change their name and password
- 👉 App needs a table view with all cabins, showing the cabin photo, name, capacity, price, and current discount
- 👉 Users should be able to update or delete a cabin, and to create new cabins (including uploading a photo)
- 👉 App needs a table view with all bookings, showing arrival and departure dates, status, and paid amount, as well as cabin and guest data
- 👉 The booking status can be “unconfirmed” (booked but not yet checked in), “checked in”, or “checked out”. The table should be filterable by this important status
- 👉 Other booking data includes: number of guests, number of nights, guest observations, whether they booked breakfast, breakfast price
- 👉 Users should be able to delete, check in, or check out a booking as the guest arrives (no editing necessary for now)
- 👉 Bookings may not have been paid yet on guest arrival. Therefore, on check in, users need to accept payment (outside the app), and then confirm that payment has been received (inside the app)
- 👉 On check in, the guest should have the ability to add breakfast for the entire stay, if they hadn't already
- 👉 Guest data should contain: full name, email, national ID, nationality, and a country flag for easy identification
- 👉 The initial app screen should be a dashboard, to display important information for the last 7, 30, or 90 days:
  - 👉 A list of guests checking in and out on the current day. Users should be able to perform these tasks from here
  - 👉 Statistics on recent bookings, sales, check ins, and occupancy rate
  - 👉 A chart showing all daily hotel sales, showing both “total” sales and “extras” sales (only breakfast at the moment)
  - 👉 A chart showing statistics on stay durations, as this is an important metric for the hotel
- 👉 Users should be able to define a few application-wide settings: breakfast price, min and max nights/booking, max guests/booking
- 👉 App needs a dark mode

STEP 1



# PROJECT REQUIREMENTS FROM THE BUSINESS

- 👉 Users of the app are hotel employees. They need to be logged into the application to perform tasks
  - 👉 New users can only be signed up inside the applications (to guarantee that only actual hotel employees can get in)
  - 👉 Users should be able to upload an avatar, and change their name and password
- AUTHENTICATION**
- 👉 App needs a table view with all cabins, showing the cabin photo, name, capacity, price, and current discount
  - 👉 Users should be able to update or delete a cabin, and to create new cabins (including uploading a photo)
- CABINS**
- 👉 App needs a table view with all bookings, showing arrival and departure dates, status, and paid amount, as well as cabin and guest data
  - 👉 The booking status can be “unconfirmed” (booked but not yet checked in), “checked in”, or “checked out”. The table needs to be sorted by this important status
  - 👉 Other booking data includes: number of guests, number of nights, guest observations, whether they booked breakfast, breakfast price
- BOOKINGS**
- 👉 Users should be able to delete, check in, or check out a booking as the guest arrives (no editing necessary for now)
  - 👉 Bookings may not have been paid yet on guest arrival. Therefore, on check in, users need to accept payment (on behalf of the guest) and then confirm that payment has been received (inside the app)
  - 👉 On check in, the guest should have the ability to add breakfast for the entire stay, if they hadn't already
- CHECK IN / OUT**
- 👉 Guest data should contain: full name, email, national ID, nationality, and a country flag for easy identification
- GUESTS**
- 👉 The initial app screen should be a dashboard, to display important information for the last 7, 30, or 90 days:
    - 👉 A list of guests checking in and out on the current day. Users should be able to perform these tasks from here
    - 👉 Statistics on recent bookings, sales, check ins, and occupancy rate
    - 👉 A chart showing all daily hotel sales, showing both “total” sales and “extras” sales (only breakfast at the moment)
    - 👉 A chart showing statistics on stay durations, as this is an important metric for the hotel
- DASHBOARD**
- 👉 Users should be able to define a few application-wide settings: breakfast price, min and max nights/booking, maximum guests per cabin
  - 👉 App needs a dark mode
- SETTINGS**

# FEATURES + PAGES

STEP 2 + 3

## FEATURE CATEGORIES

1 Bookings

2 Cabins

3 Guests

4 Dashboard

5 Check in and out

6 App settings

7 Authentication

## NECESSARY PAGES

1 Dashboard

2 Bookings

3 Cabins

4 Booking check in

5 App settings

6 User sign up

7 Login

8 Account settings

/dashboard

/bookings

/cabins

/checkin/:bookingID

/settings

/users

/login

/account

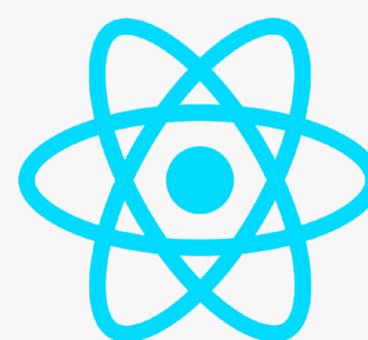


We will discuss state later.  
Most of it will be global

# CLIENT-SIDE RENDERING (CSR) OR SERVER-SIDE RENDERING (SSR)?

## CSR WITH PLAIN REACT

- 👉 Used to build **Single-Page Applications (SPAs)**
- 👉 All HTML is rendered on the **client**
- 👉 All JavaScript needs to be downloaded before apps start running: **bad for performance**
- 👉 **One perfect use case:** apps that are used “internally” as tools inside companies, that are entirely hidden behind a login



This is exactly what  
we want to build in  
this project

## SSR WITH FRAMEWORK

- 👉 Used to build **Multi-Page Applications (MPAs)**
- 👉 Some HTML is rendered in the **server**
- 👉 **More performant**, as less JavaScript needs to be downloaded
- 👉 The **React team** is moving more and more in this direction

NEXT.JS

Remix

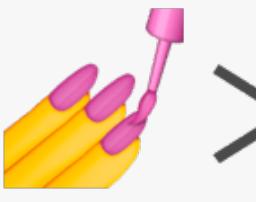
# TECHNOLOGY DECISIONS

👉 Routing

 **React Router**

STEP 4

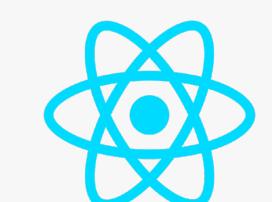
👉 Styling

<  > styled components

👉 Remote state management

 **React Query**

👉 UI State management

 **Context API**

👉 Form management

 **React Hook Form**

👉 Other tools

React icons / React hot toast / Recharts / date-fns / Supabase

*The standard for React SPAs*

*Very popular way of writing component-scoped CSS, right inside JavaScript. A technology worth learning*

*The best way of managing remote state, with features like caching, automatic re-fetching, pre-fetching, offline support, etc. Alternatives are SWR and RTK Query, but this is the most popular*

*There is almost no UI state needed in this app, so one simple context with useState will be enough. No need for Redux*

*Handling bigger forms can be a lot of work, such as manual state creation and error handling. A library can simplify all this*

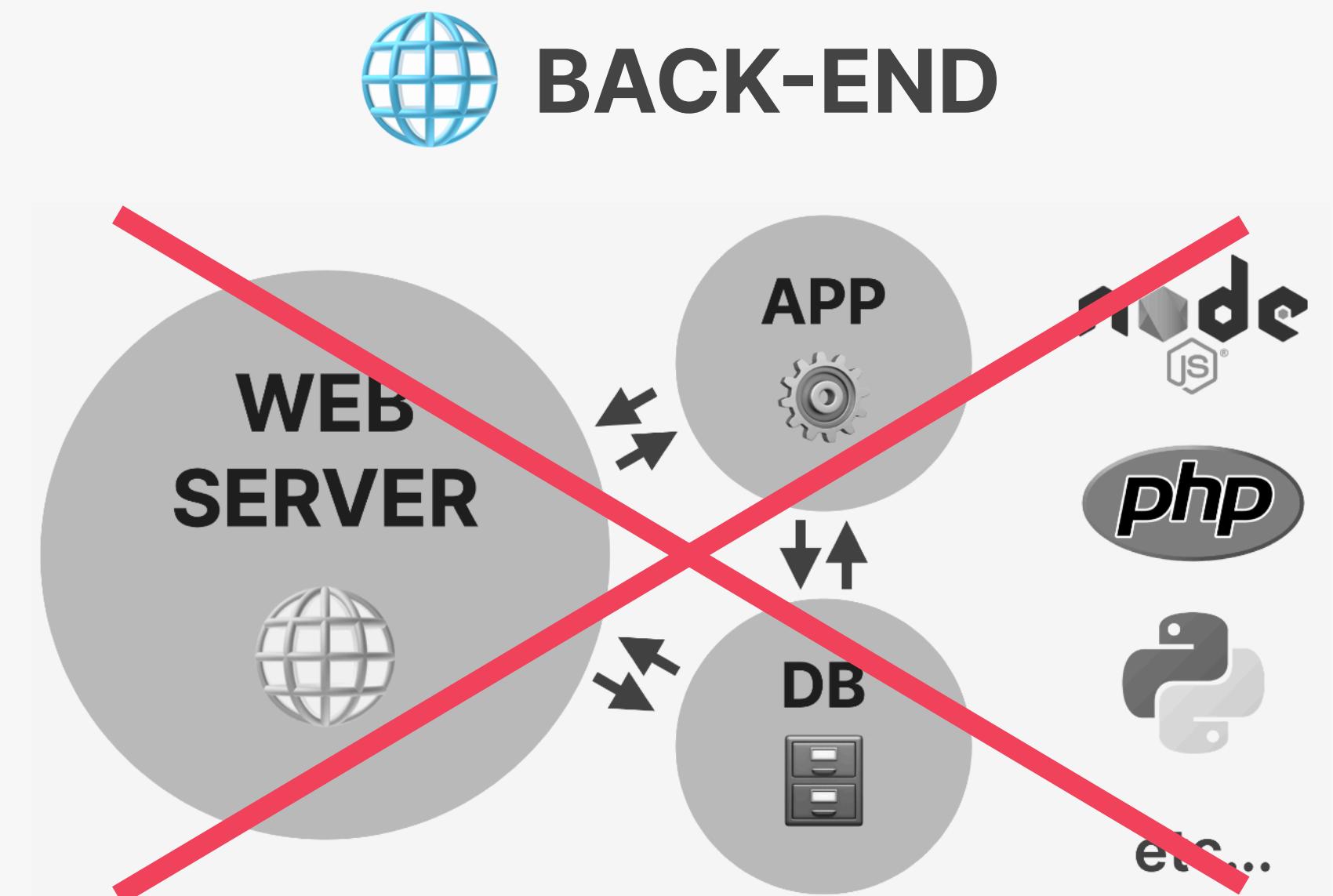


**SUPABASE CRASH  
COURSE: BUILDING A  
BACK-END!**

# WHAT IS SUPABASE?

## SUPABASE

- 👉 Service that allows developers to easily **create a back-end with a Postgres database**
- 👉 Automatically creates a **database** and **API** so we can easily request and receive data from the server
- 👉 No back-end development needed 😊
- 👉 Perfect to get up and running **quickly!**
- 👉 Not just an API: Supabase also comes with easy-to-use **user authentication** and **file storage**

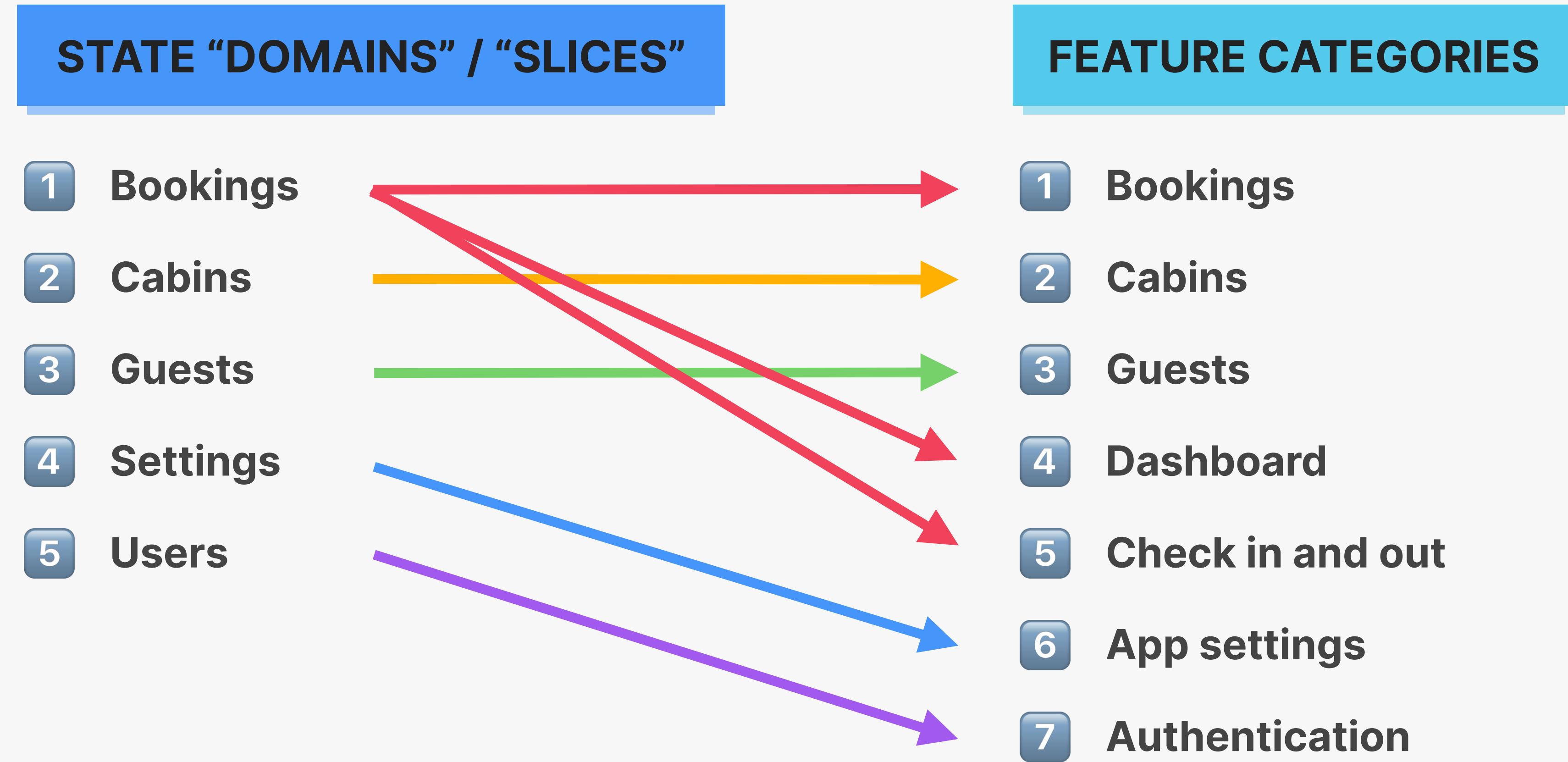


**WITH SUPABASE, WE DON'T NEED  
TO DO ANY OF THIS MANUALLY!  
IT'S ALL INCLUDED**

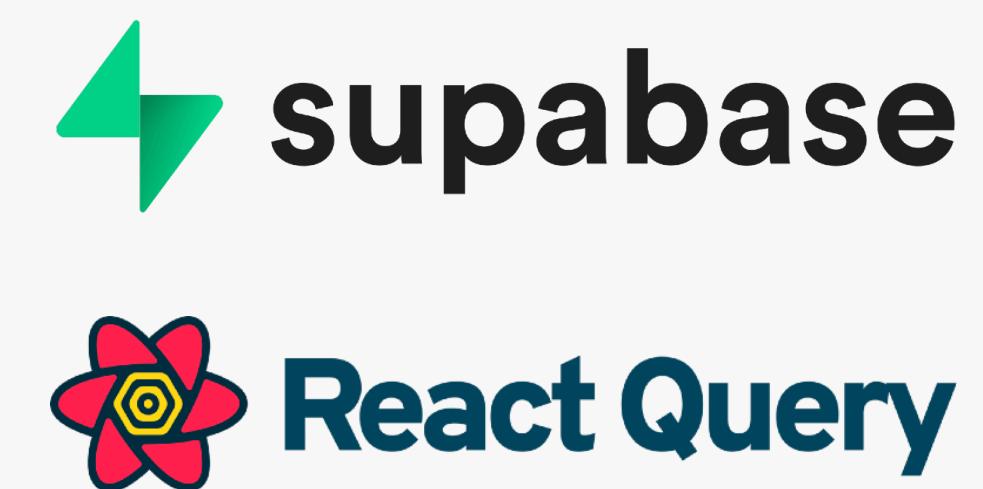




# MODELING STATE

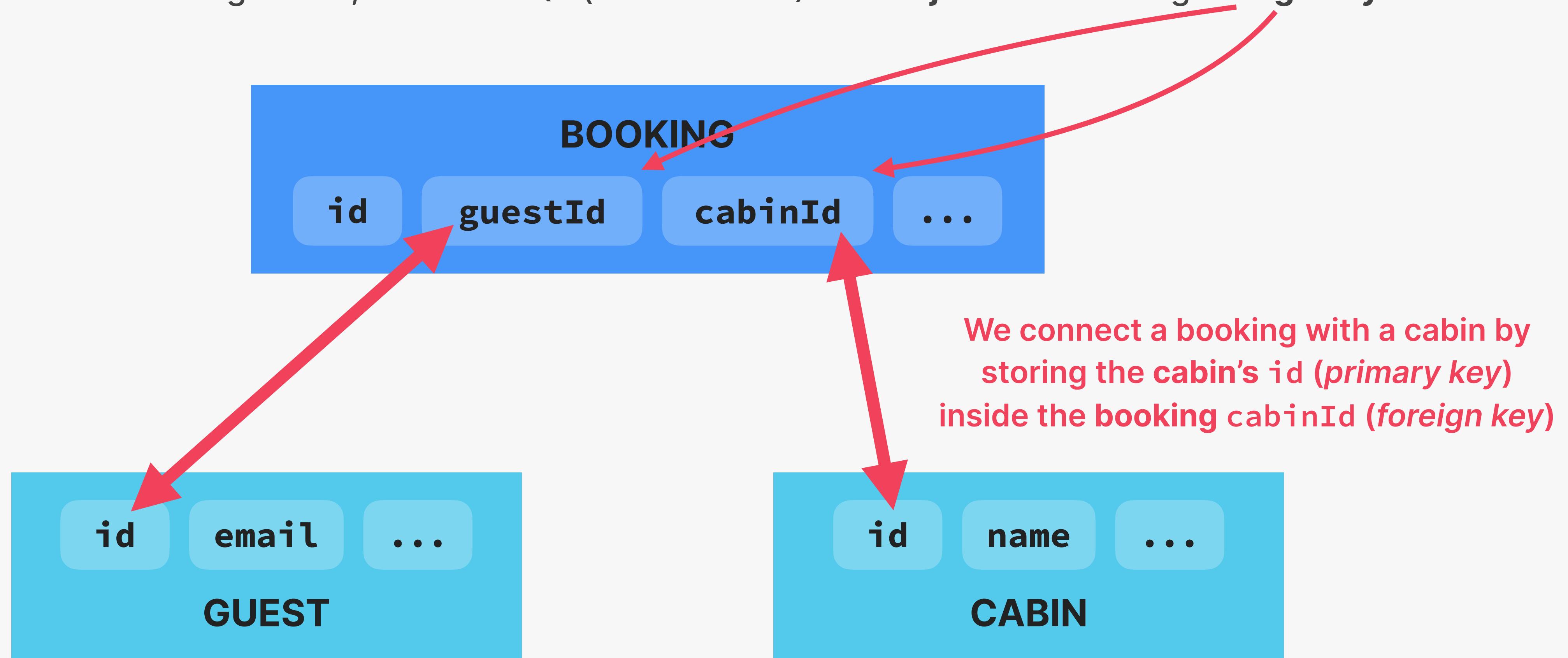


- 👉 All this state will be **global remote state**, stored within Supabase
- 👉 There will be one **table** for each state “slice” in the database



# THE BOOKINGS TABLE

- 👉 Bookings are about a **guest** renting a **cabin**
- 👉 So a booking needs information about what **guest** is booking which **cabin**: we need to **connect** them
- 👉 Supabase uses a Postgres DB, which is SQL (relational DB). So we **join** tables using **foreign keys**



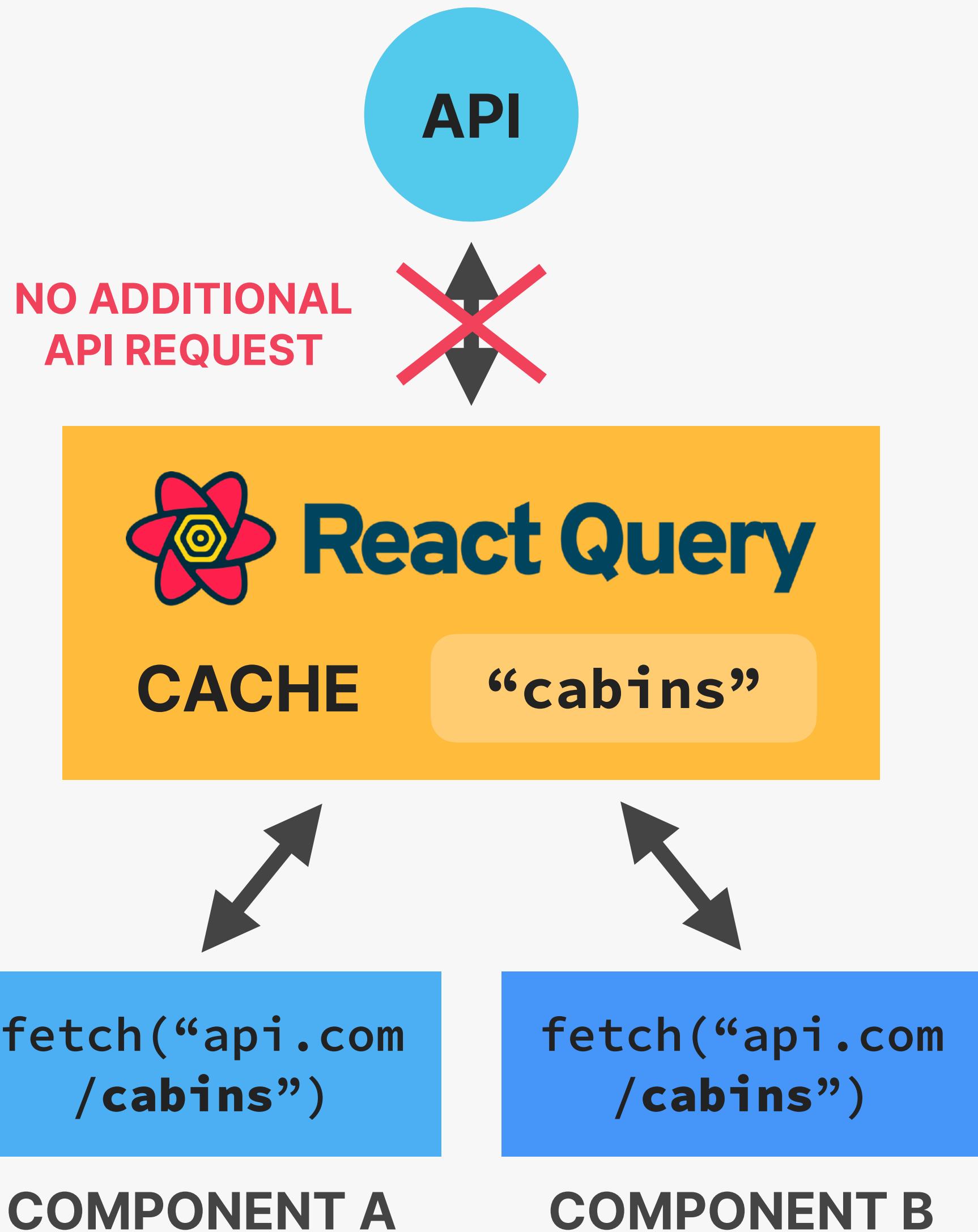


# REACT QUERY: MANAGING REMOTE STATE

# WHAT IS REACT QUERY?

## REACT QUERY

- 👉 Powerful library for managing **remote (server) state**
- 👉 Many features that allow us to write a **lot less code**, while also **making the UX a lot better**:
  - 👉 Data is stored in a cache
  - 👉 Automatic loading and error states
  - 👉 Automatic re-fetching to keep state synched
  - 👉 Pre-fetching
  - 👉 Easy remote state mutation (updating)
  - 👉 Offline support
- 👉 Needed because remote state is **fundamentally different** from regular (UI) state





# ADVANCED REACT PATTERNS

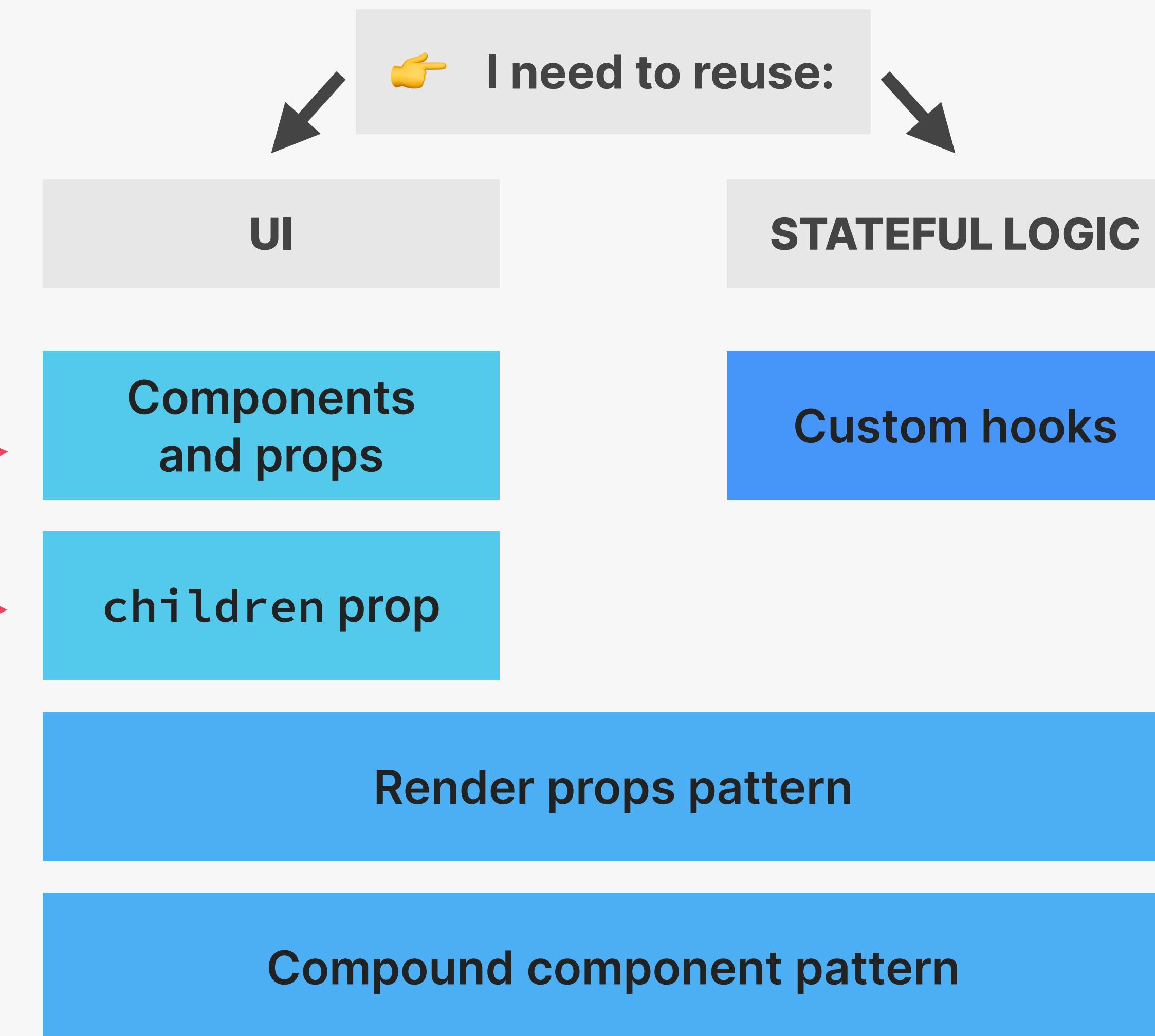
# HOW TO REUSE CODE IN REACT?

Use props as a component API, to enable custom behavior. Can be stateless, stateful, or structural components

👉 I need to reuse:

To customize the component's content

👉 There are even more patterns, but these ones matter most



Logic with hooks

For complete control over what the component renders, by passing in a function that tells the component what to render. Was more common before hooks, but still useful

For very self-contained components that need/want to manage their own state. Compound components are like fancy super-components