A FIRST LOOK AT REACT

THE RISE OF **SINGLE-PAGE APPLICATIONS**

WordPress

NEXT.JS
Remix

The “old” way (but now getting modern again...)

SERVER-SIDE RENDERING

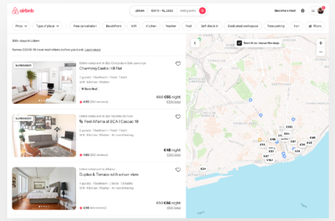
VIEW

DATA

RENDER WEBPAGE

HTML CSS JS

BROWSER



jQuery
write less, do more.

👉 *Developers started writing more and more JS code, which led to...*

SERVER / BACK-END

CLIENT / FRONT-END

The “modern” way

CLIENT-SIDE RENDERING

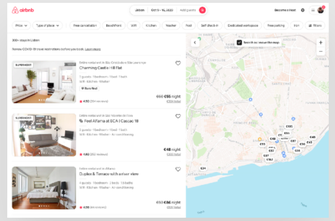
DATA

API

BROWSER

RENDER WEBPAGE

HTML CSS JS



VIEW

👉 **Single-Page Application (SPA)**

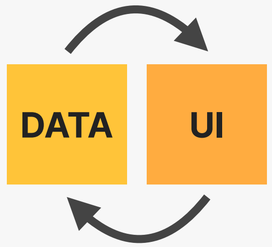
SINGLE-PAGE APPLICATIONS WITH **VANILLA JAVASCRIPT?**

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

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

↓

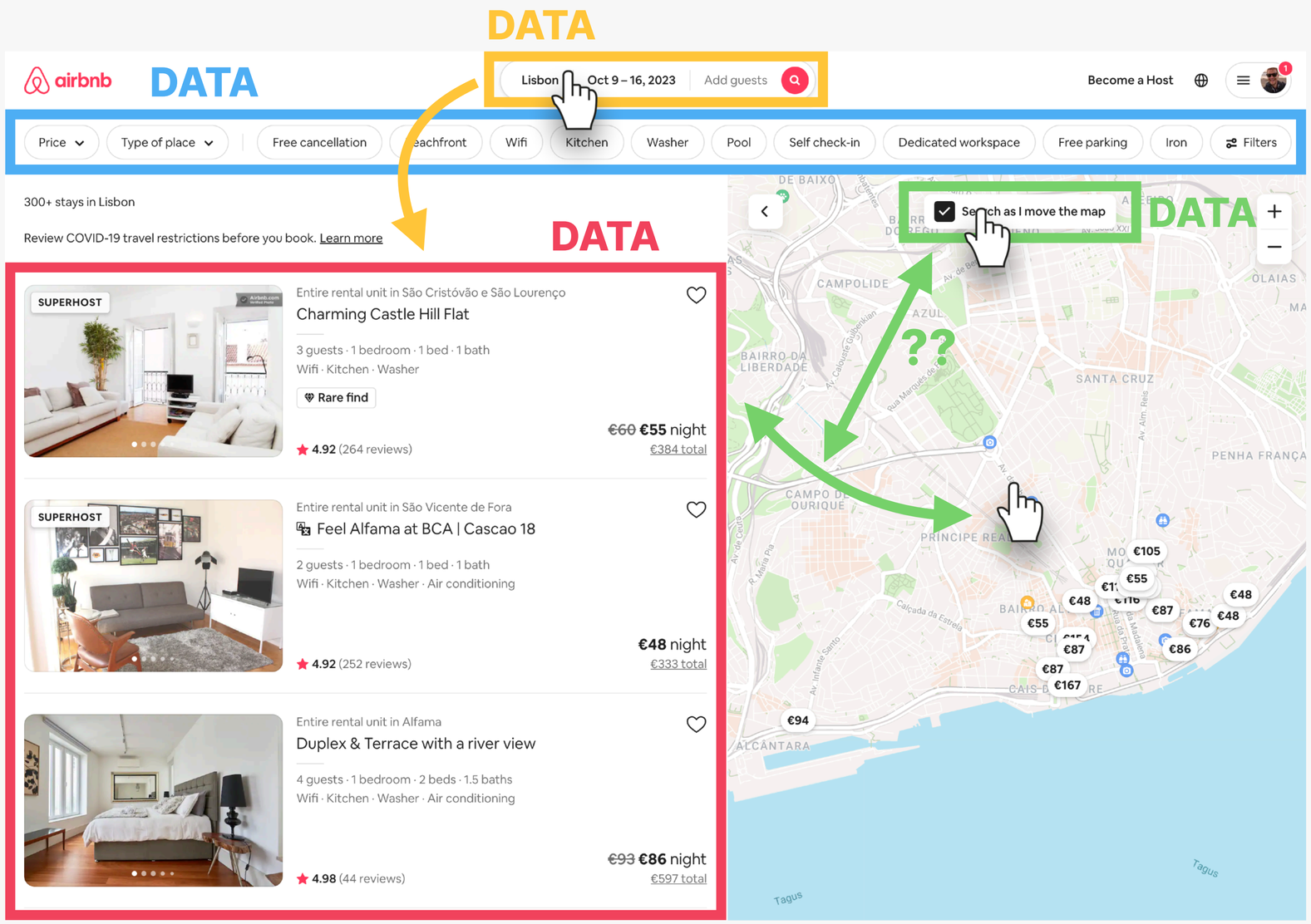
User interface needs to **stay in sync** with data

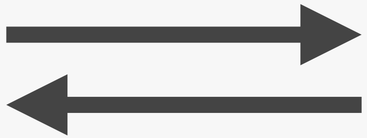


↓

Very hard problem to solve!

KEEPING UI **IN SYNC** WITH DATA





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

Piece of data
=
Piece of *state*

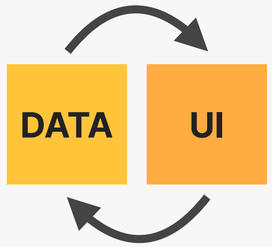
SINGLE-PAGE APPLICATIONS WITH **VANILLA JAVASCRIPT?**

👉 *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 JS jQuery

write less, do more.

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

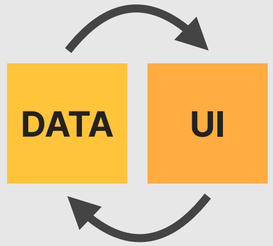
Data (state) is usually **stored in the DOM**, shared across entire app 👉 Hard to reason + bugs 🐛

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

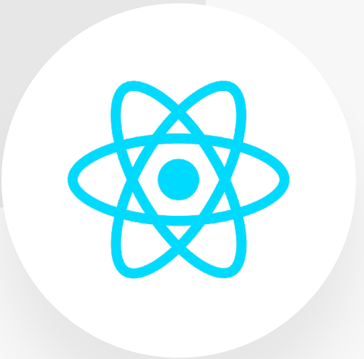
They give developers and teams a **consistent** way of building front-end applications

WHAT IS **REACT**?

REACT

JAVASCRIPT LIBRARY FOR BUILDING

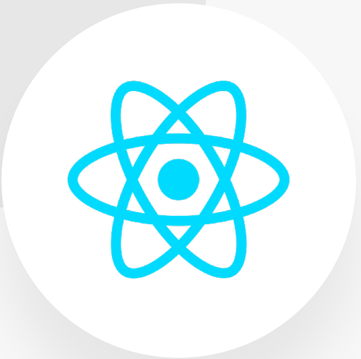
USER INTERFACES



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

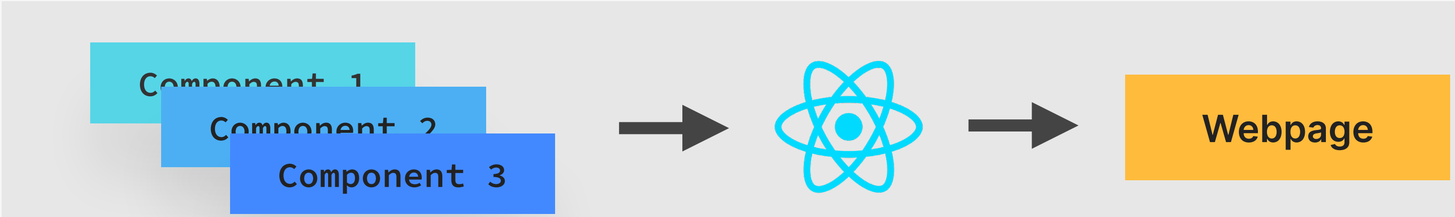
State-driven

JavaScript library

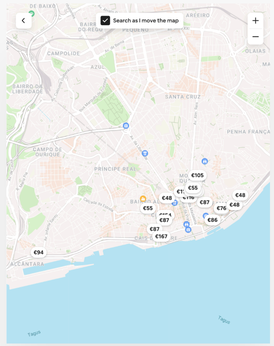
Extremely popular

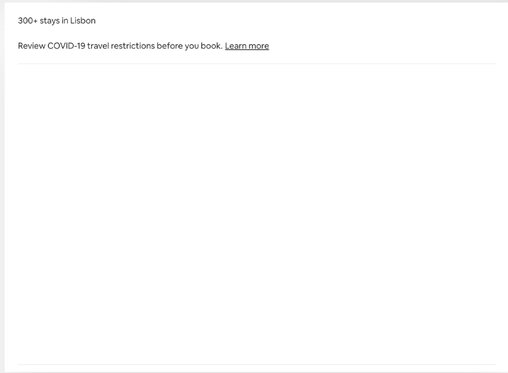
Created by facebook

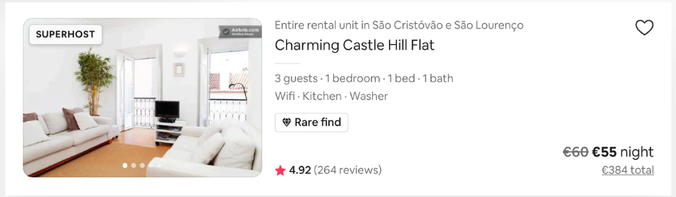
👉 Components are the **building blocks** of user interfaces in React



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

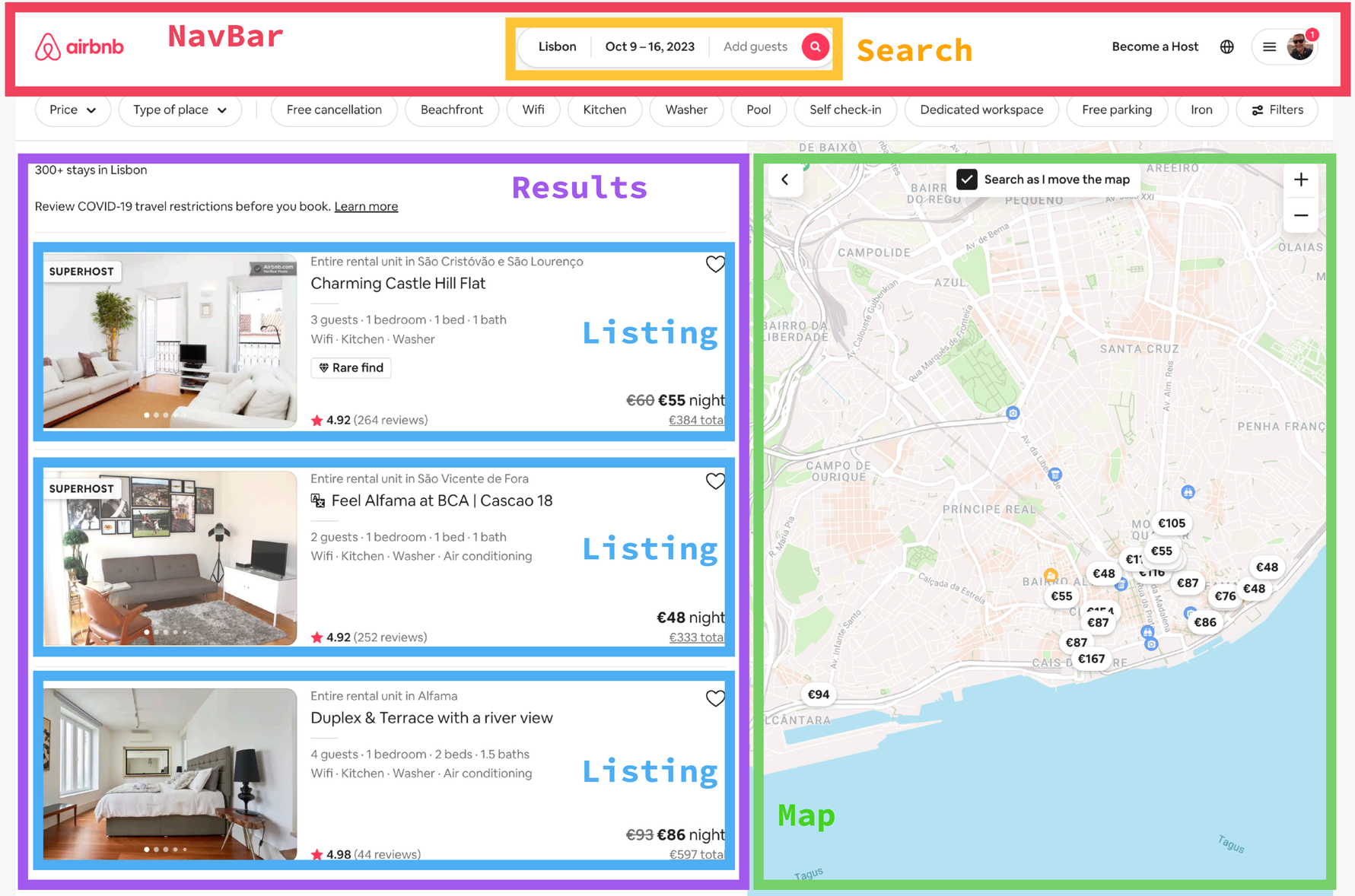






REACT IS **BASED ON COMPONENTS**

- **Based on components**
- Declarative
- State-driven
- JavaScript library
- Extremely popular
- Created by facebook



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**

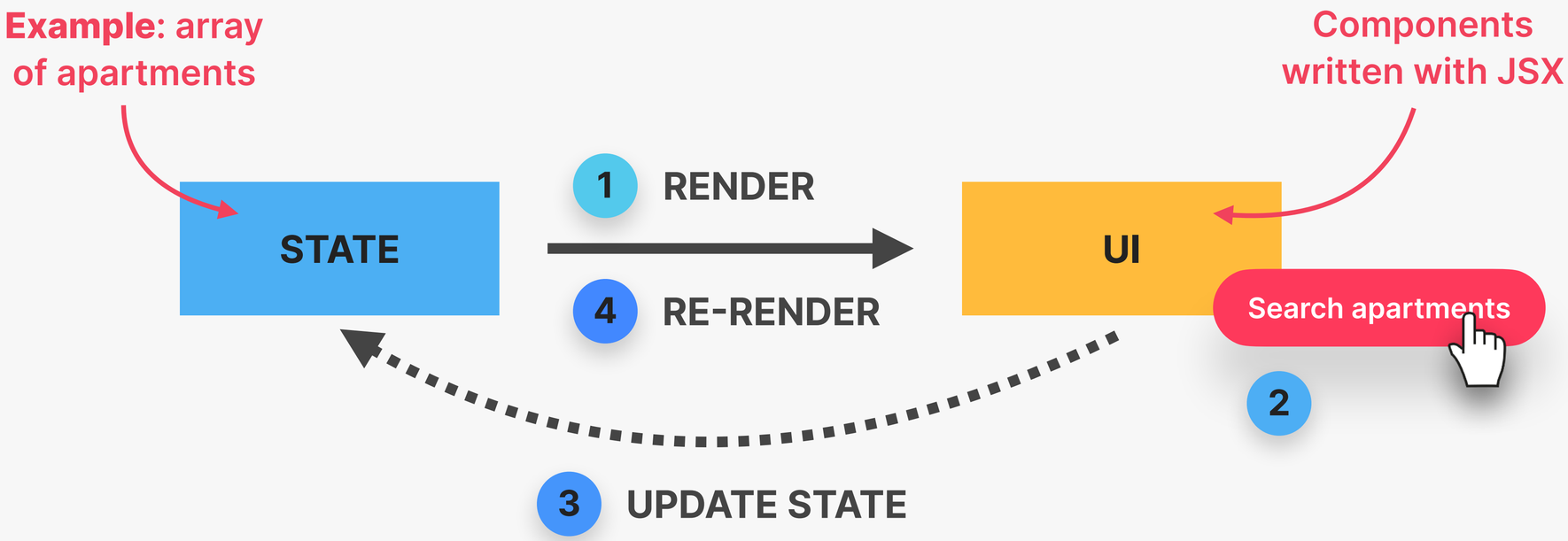
👉 **JSX**: a syntax that **combines** HTML CSS JavaScript as well as referencing **other components**

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**

- Based on components
- Declarative
- State-driven
- JavaScript library
- Extremely popular
- Created by facebook

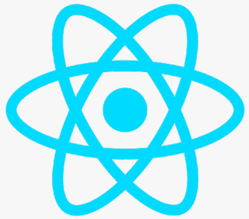


REACT *REACTS* TO STATE CHANGES
BY RE-RENDERING THE UI

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?



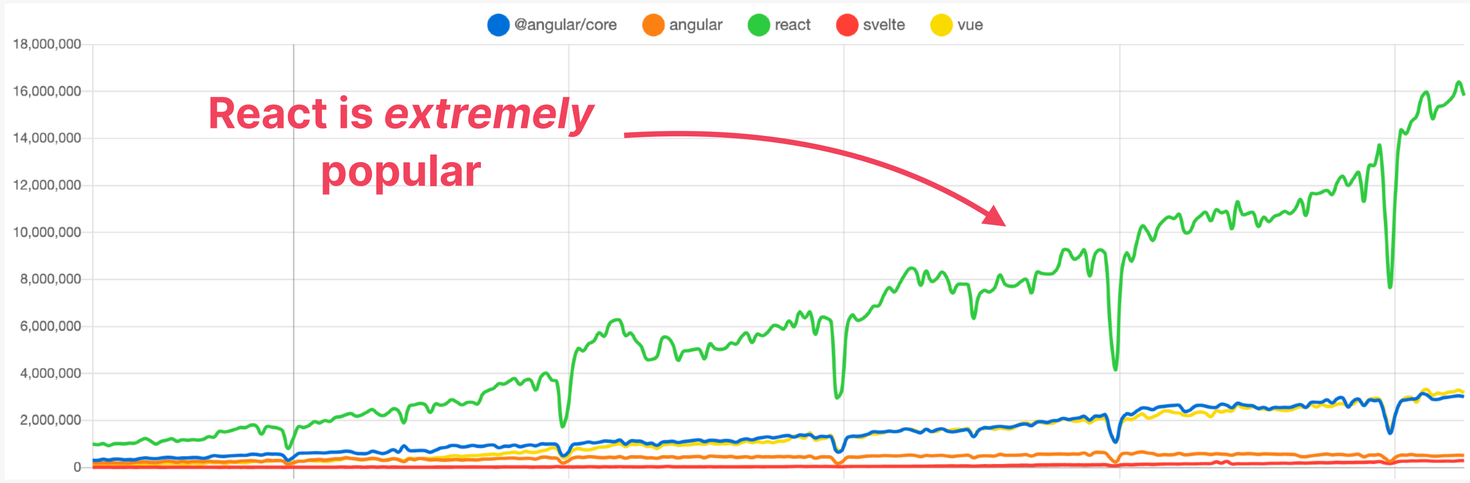
Because React is only the “view” layer. We need to pick multiple external libraries to build a complete application

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



(Weekly downloads from NPM)

- ✅ 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

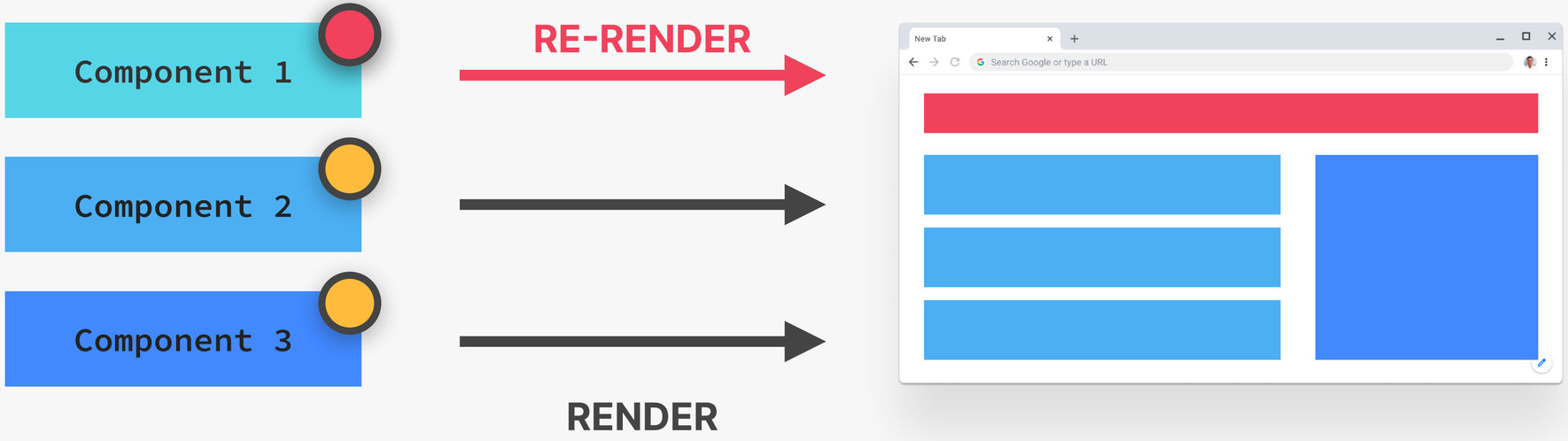
👉 React was open-sourced in **2013**, and has since then completely transformed front-end web development

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 RECT 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)

Most of the course

- ✅ Use for **tutorials** or experiments
- ⛔ Don’t use for a **real-world app**

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

- ✅ Use for modern **real-world apps**

By the end of the course

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 Copy

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

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

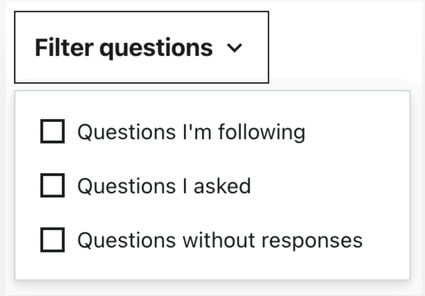
Remix is a full-stack React framework with nested routing. It lets you break your app into nested parts that can load data in parallel and refresh in response to the user actions. To create a new Remix project, run:

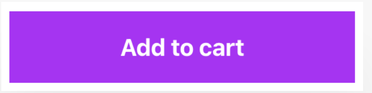
WORKING WITH COMPONENTS, PROPS, AND JSX

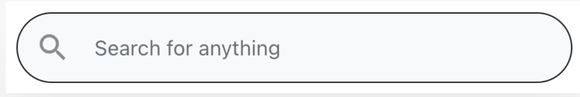
COMPONENTS AS **BUILDING BLOCKS**

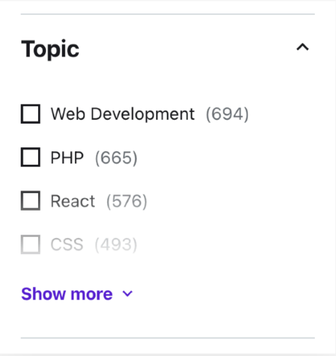
COMPONENTS

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





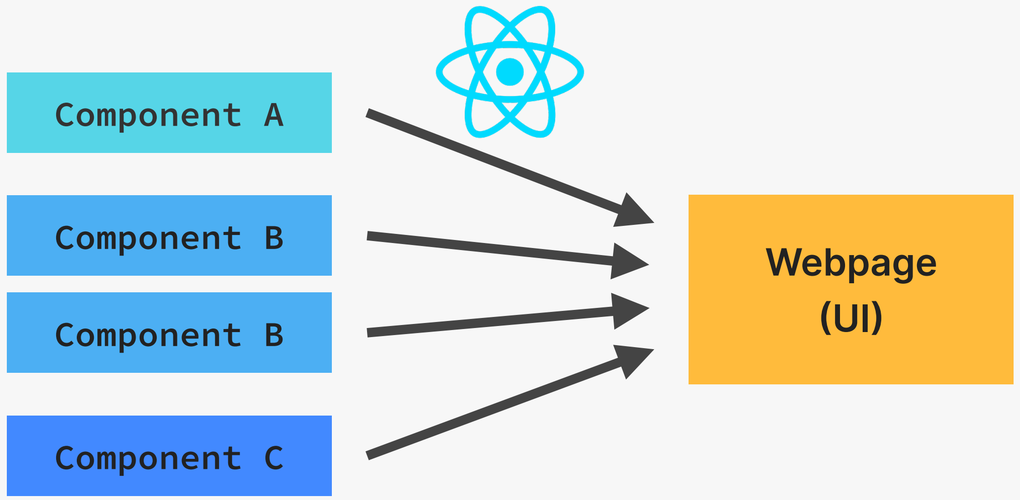




COMPONENTS AS **BUILDING BLOCKS**

COMPONENTS

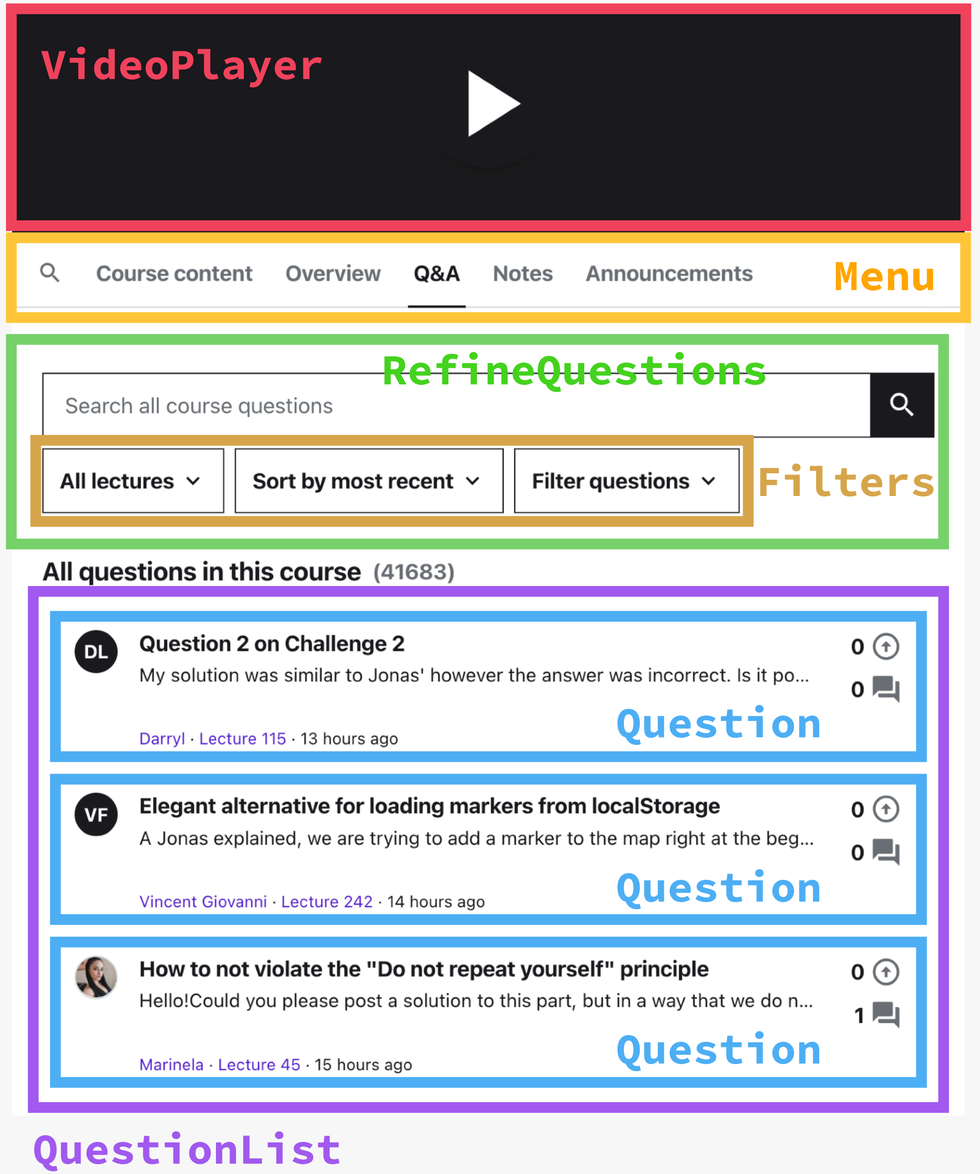
- 👉 React applications are entirely made out of components
- 👉 **Building blocks** of user interfaces in React
- 👉 Piece of UI that has its own **data**, **logic**, and **appearance** (*how it works and looks*)
- 👉 We build complex UIs by **building multiple components** and **combining** them



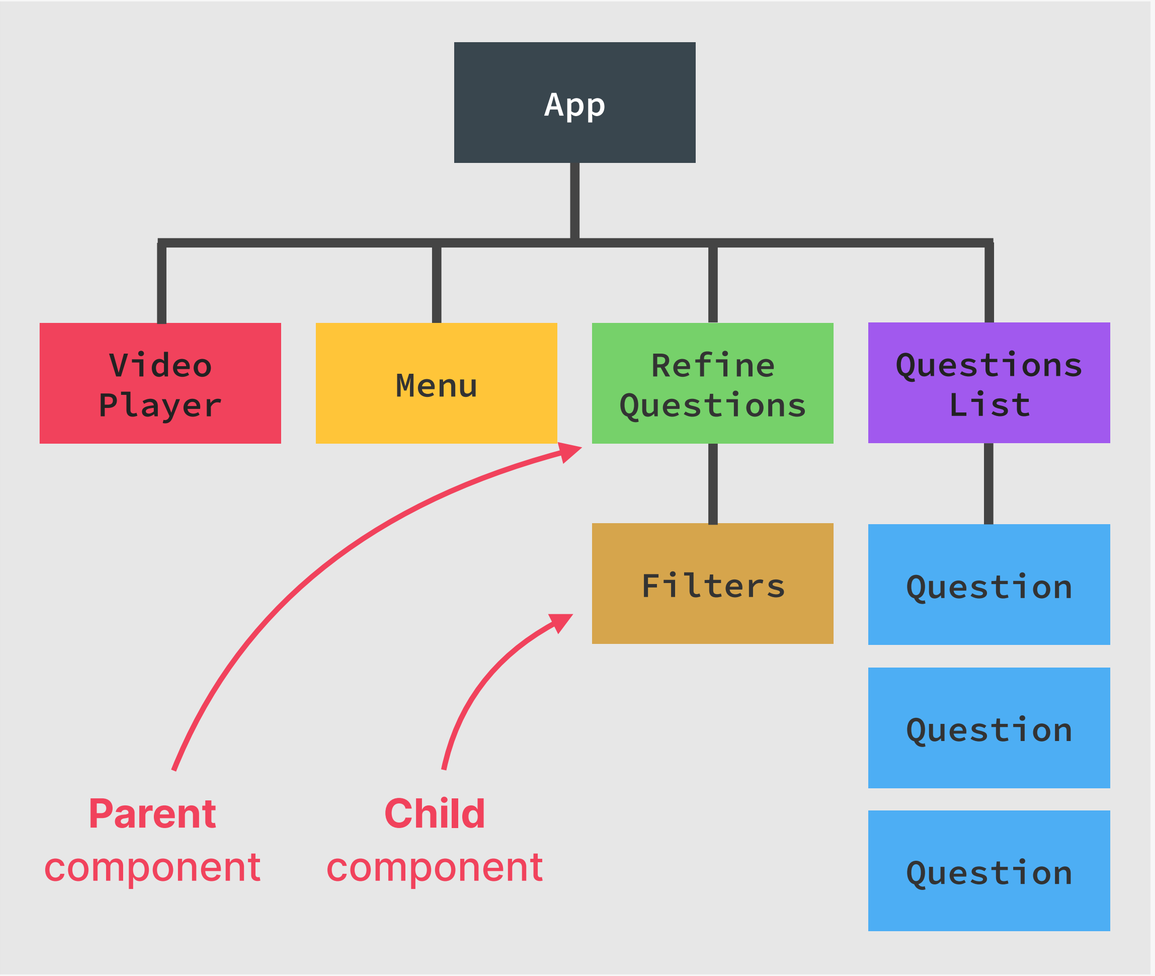
COMPONENTS AS **BUILDING BLOCKS**

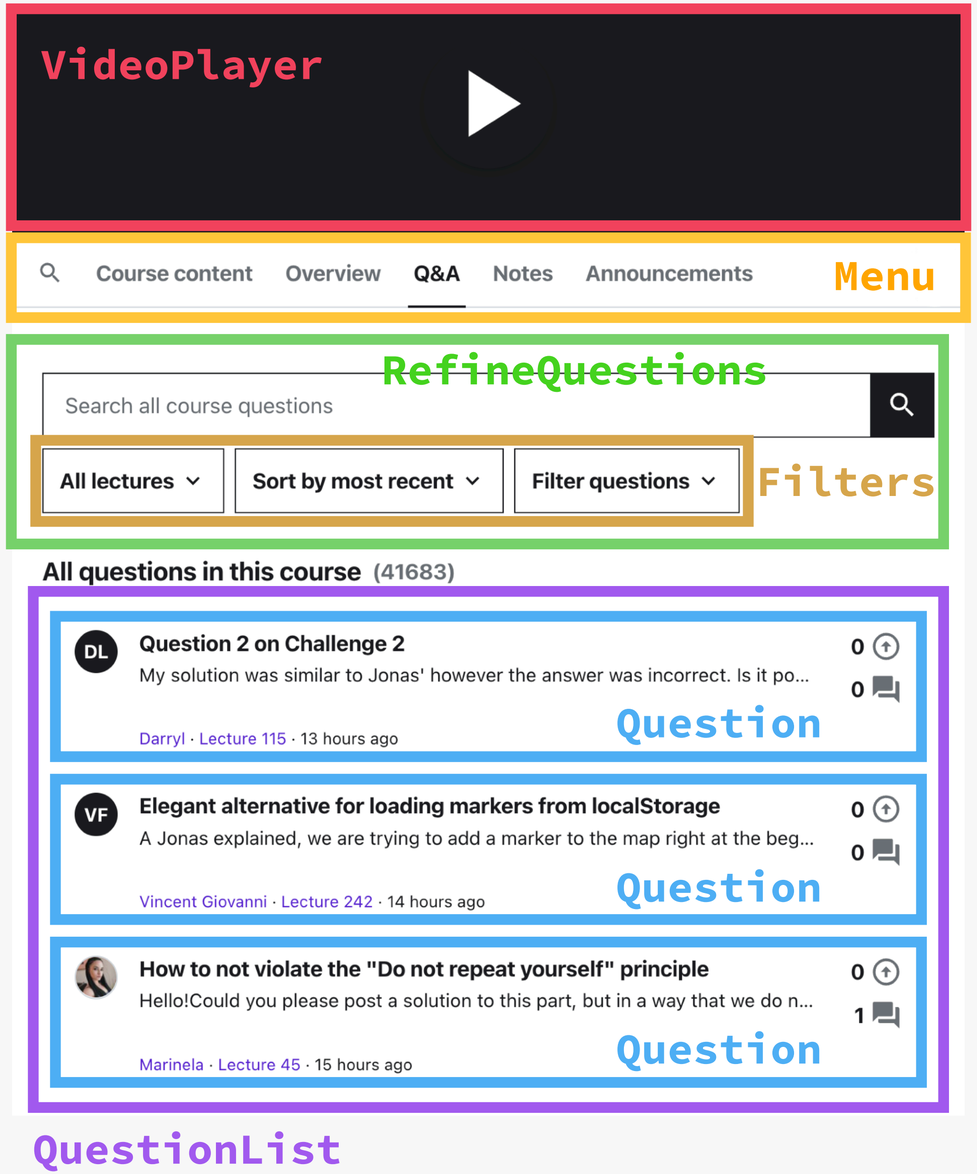
COMPONENTS

- 👉 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



COMPONENT TREES

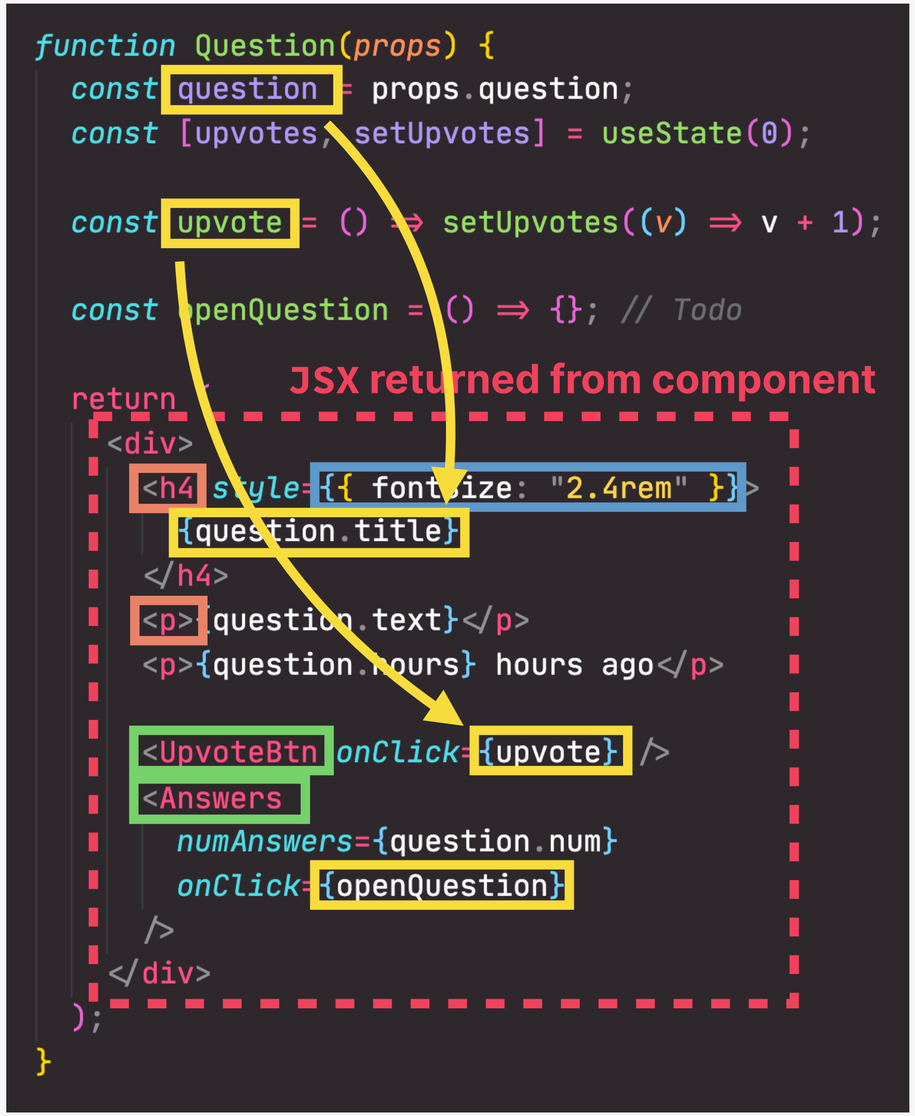




WHAT IS JSX?

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**



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**

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


BABEL

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


Hello React!

JSX IS **DECLARATIVE**

IMPERATIVE ← *"How to do things"*

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

JS

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

DECLARATIVE ← *"What we want"*

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

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

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

SEPARATION OF CONCERNS?

ONE *TECHNOLOGY* PER FILE

"Traditional" **separation of concerns**

JS

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

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

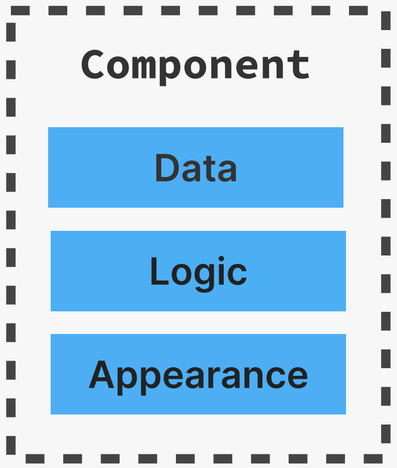
```
<div>
  <h4 class="title"></h4>
  <button class="btn">Upvote</button>

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

HTML

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

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} />
      ) : (
        <FirstAnswer />
      )}
    </div>
  );
}
```

JS

HTML

JS

Fundamental reason for **components**

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

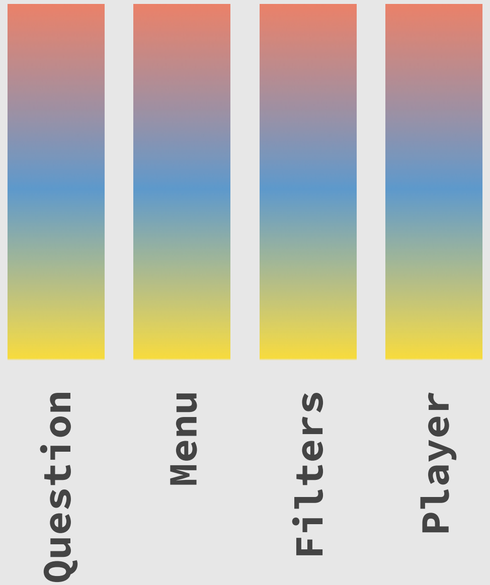
SEPARATION OF CONCERNS!

ONE *TECHNOLOGY* PER FILE

"Traditional" **separation of concerns**

COMPLETELY NEW PARADIGM

ONE *COMPONENT* PER FILE



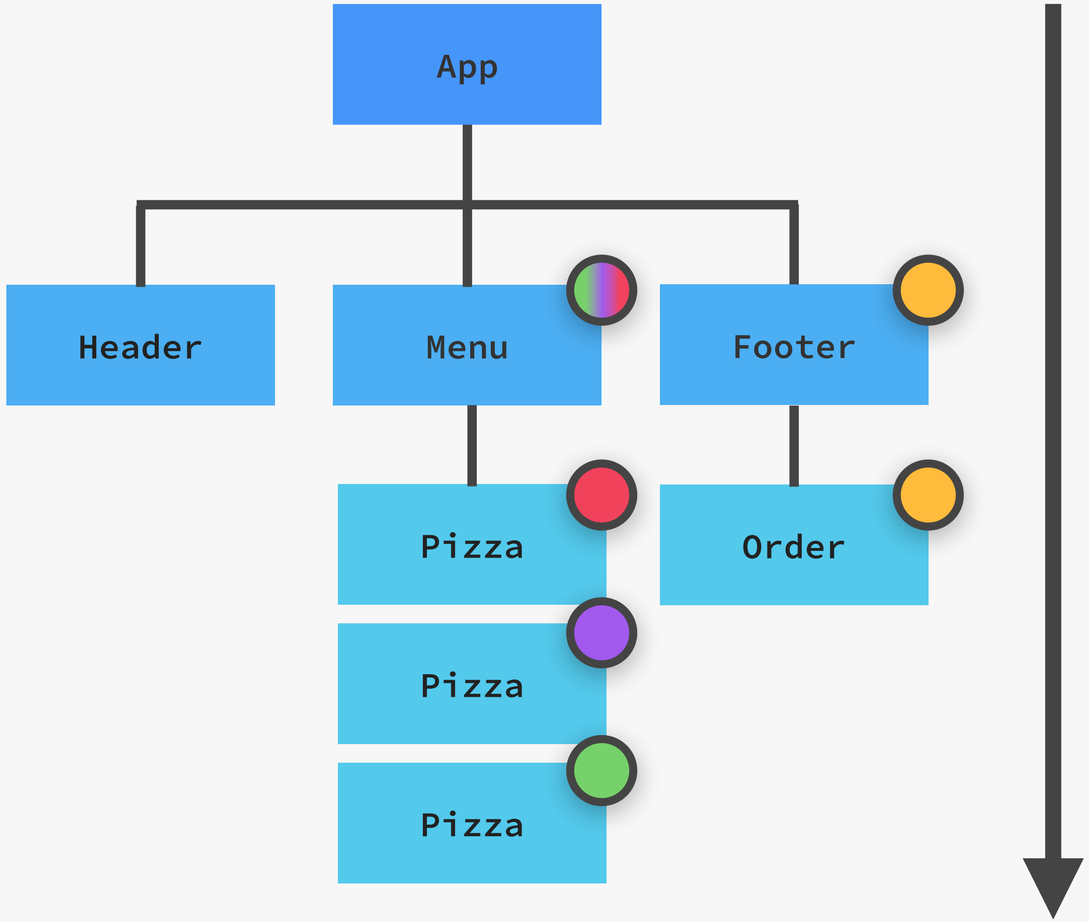
Each component is **concerned with one piece of the UI**

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

REVIEWING **PROPS**

PROPS

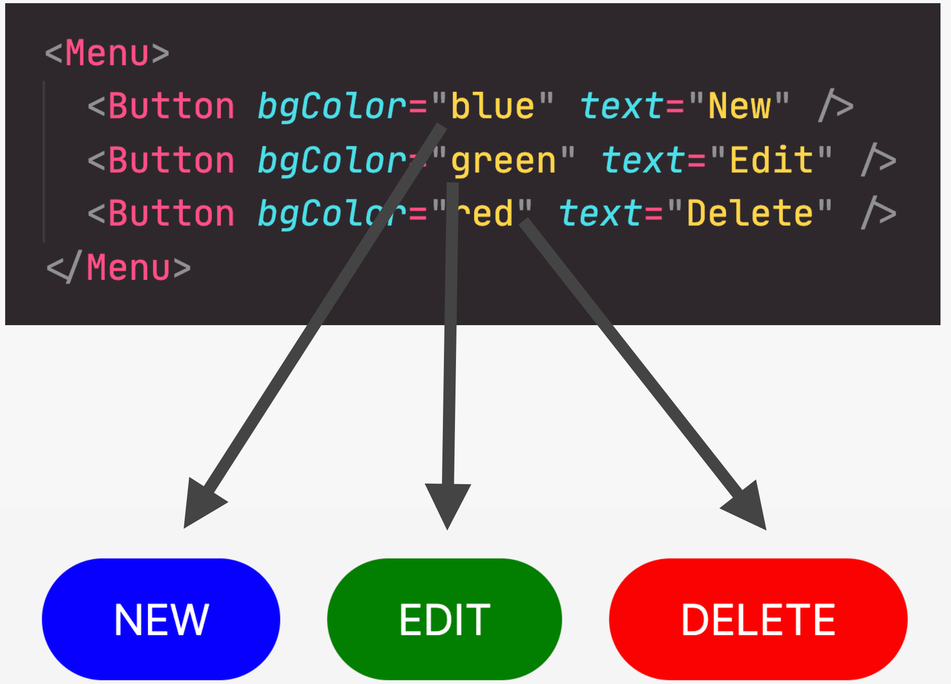
👉 Props are used to pass data from **parent components** to **child components (down** the component tree)



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



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

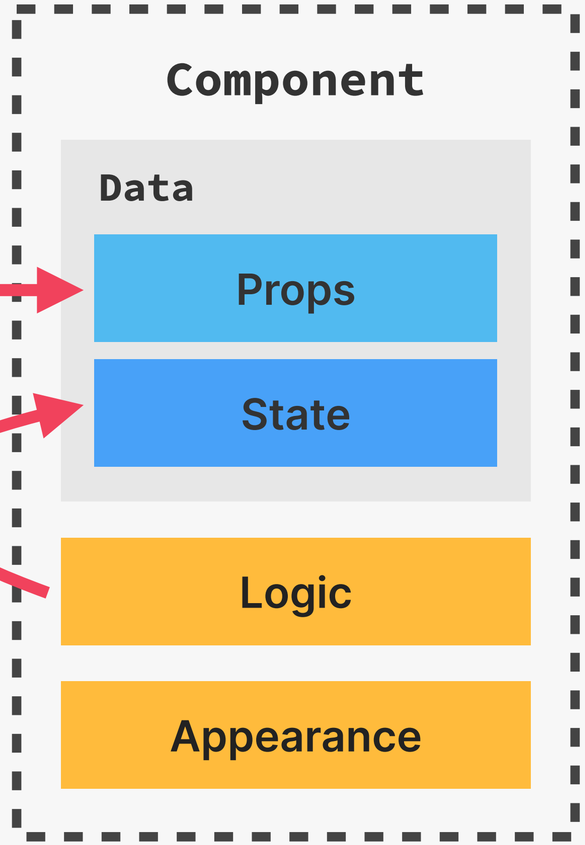
```jsx
function CourseRating() {
  const [rating, setRating] = useState(0);

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

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

PROPS ARE **READ-ONLY!**

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

Parent Component

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




```
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**

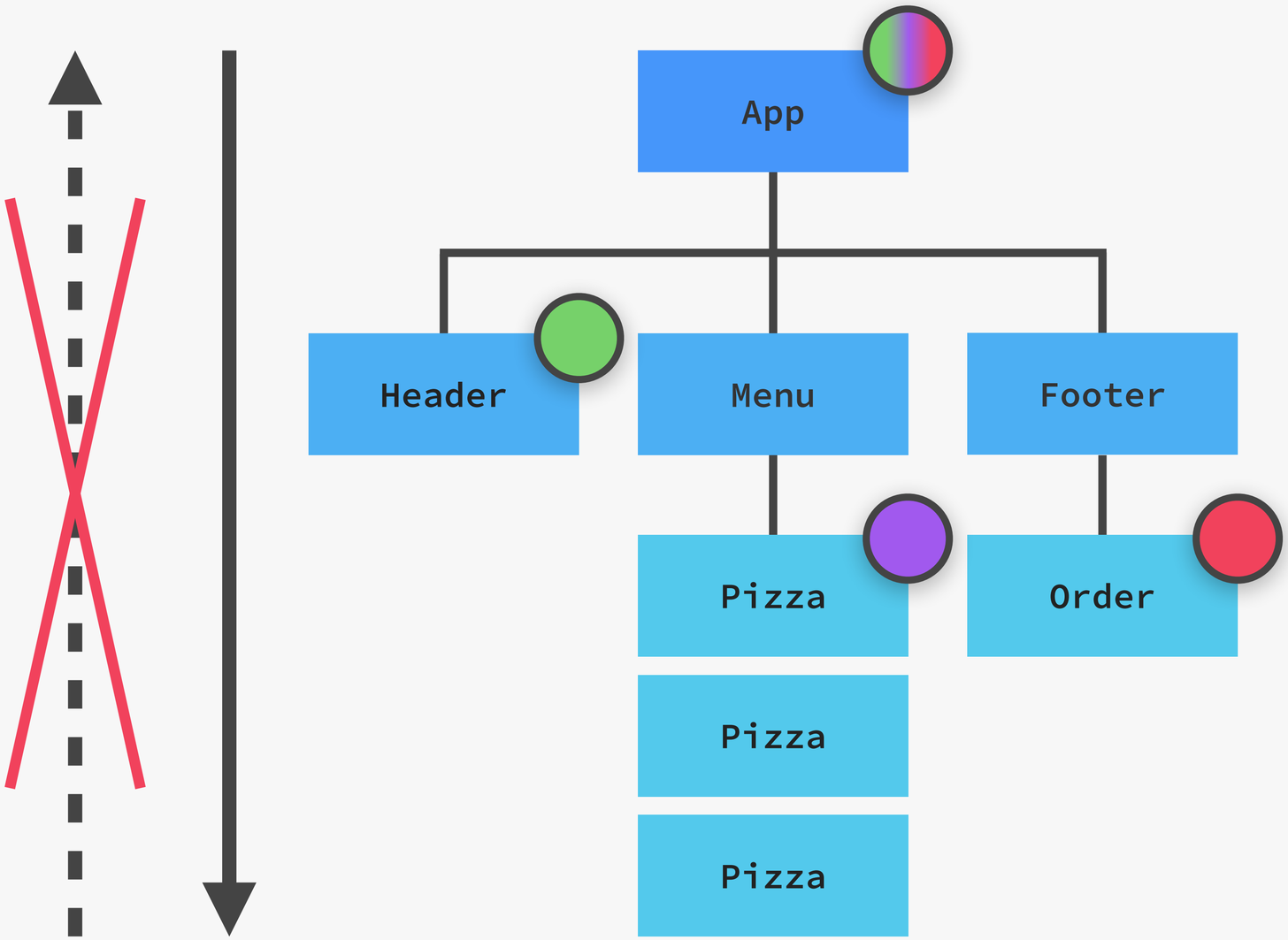
⬇️ WHY?

👉 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

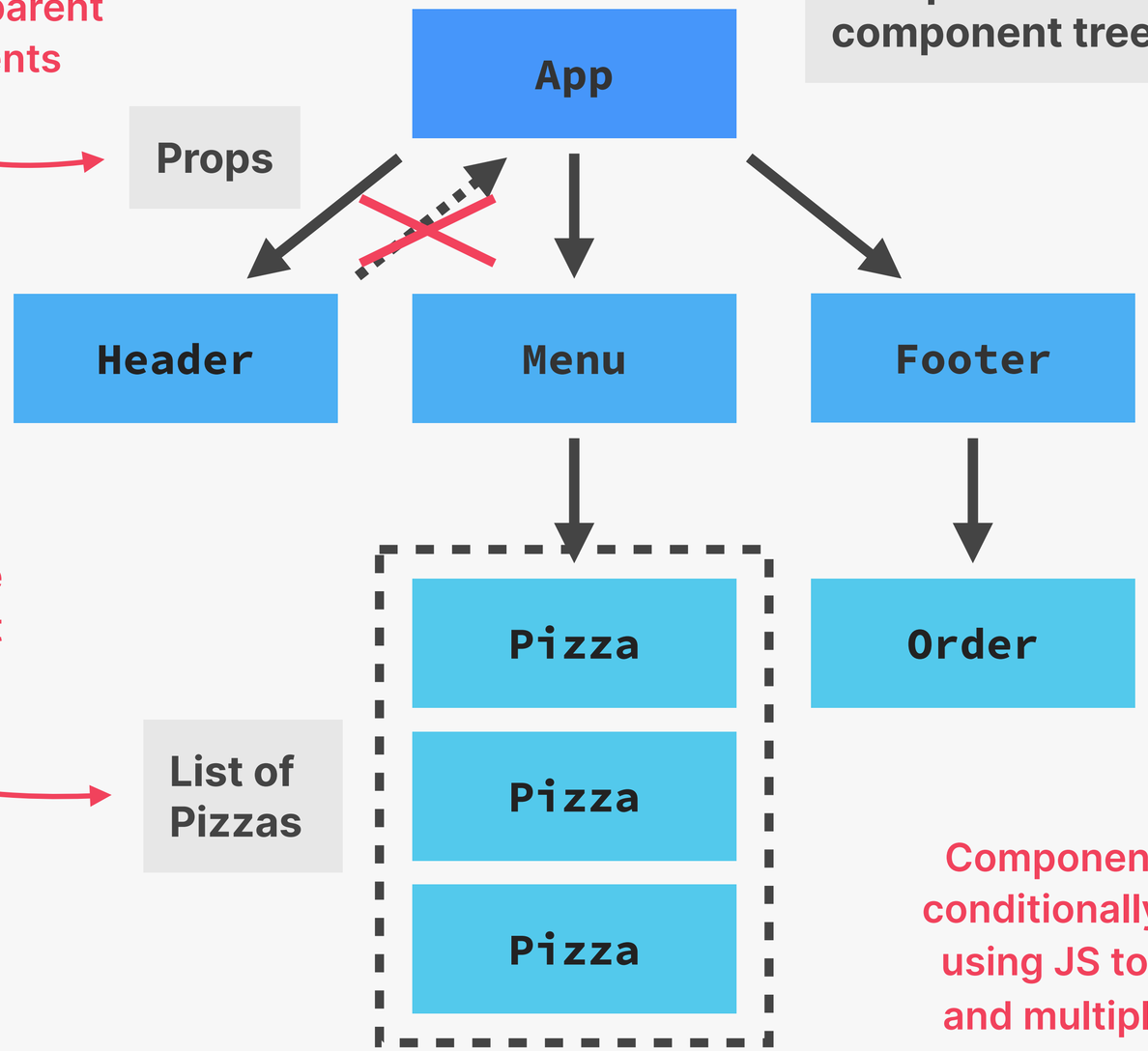
- 👉 `className` instead of HTML's `class`
- 👉 `htmlFor` instead of HTML's `for`
- 👉 Every tag needs to be **closed**. Examples: `<img />` or `<br />`
- 👉 All event handlers and other properties need to be **camelCased**. Examples: `onClick` or `onMouseOver`
- 👉 **Exception**: `aria-*` and `data-*` are 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**

To pass data from parent to child components

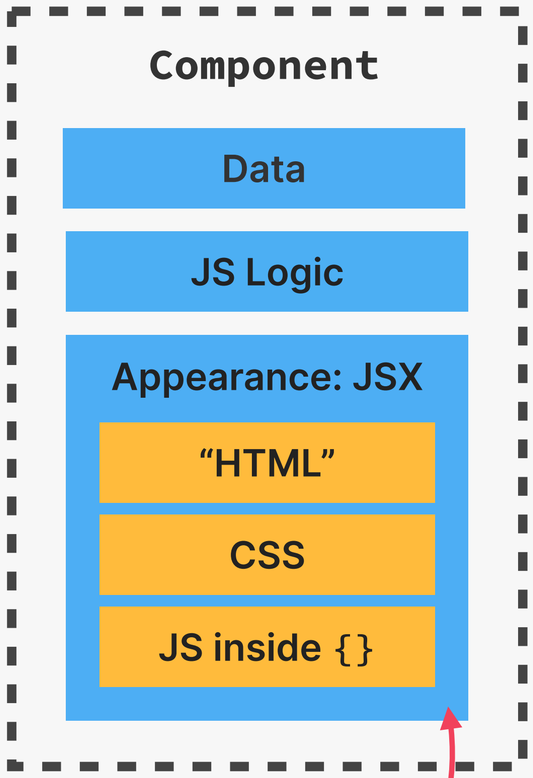
Props

Components in component tree



Rendering multiple components at once using the JavaScript `.map()` method

List of Pizzas



Components can be conditionally rendered using JS tools: `&&`, `?`, and multiple `return`

JSX block is what we return from a component

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...

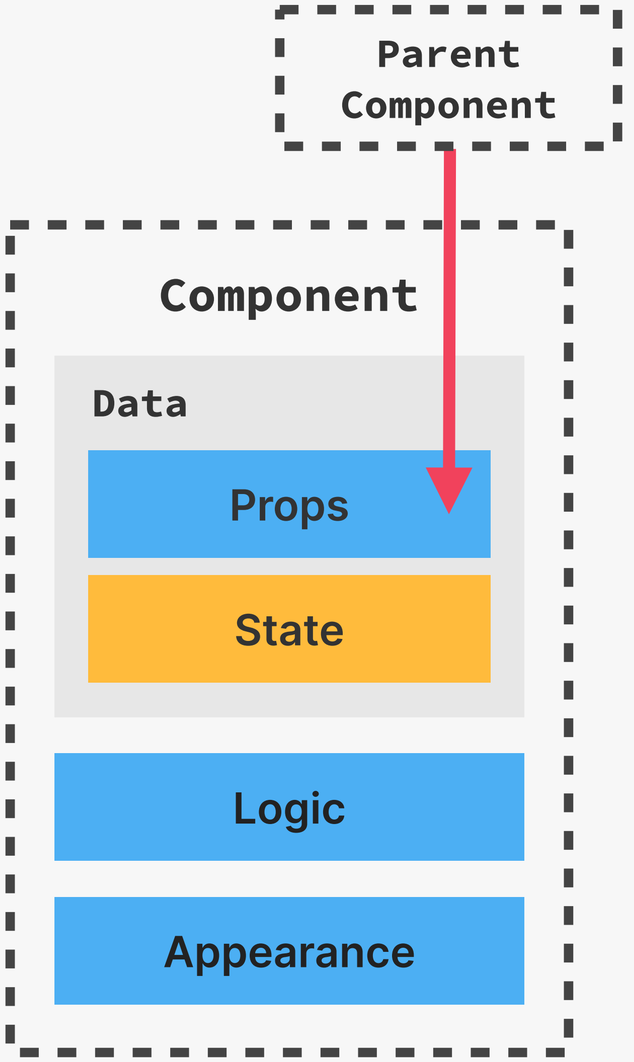
👋 **State** is the most important concept in React

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

WHAT IS **STATE**?

STATE

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

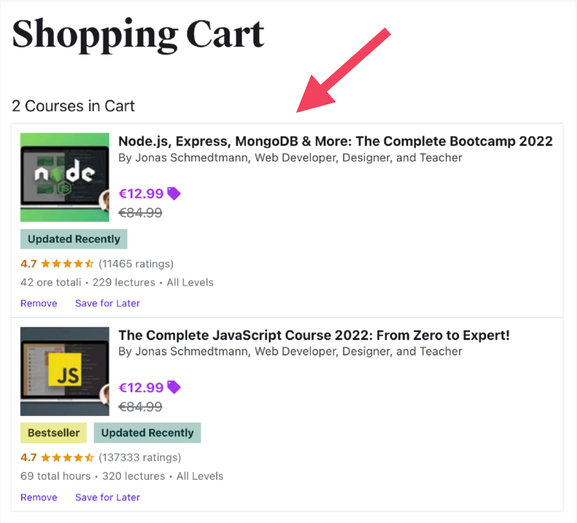


WHAT IS **STATE**?

STATE

- 👉 Data that a component **can hold over time**, necessary for information that it needs to **remember** throughout the app's lifecycle
- 👉 **"Component's memory"** 
- 👉 **"State variable" / "piece of state"**: A single variable in a component (component state)

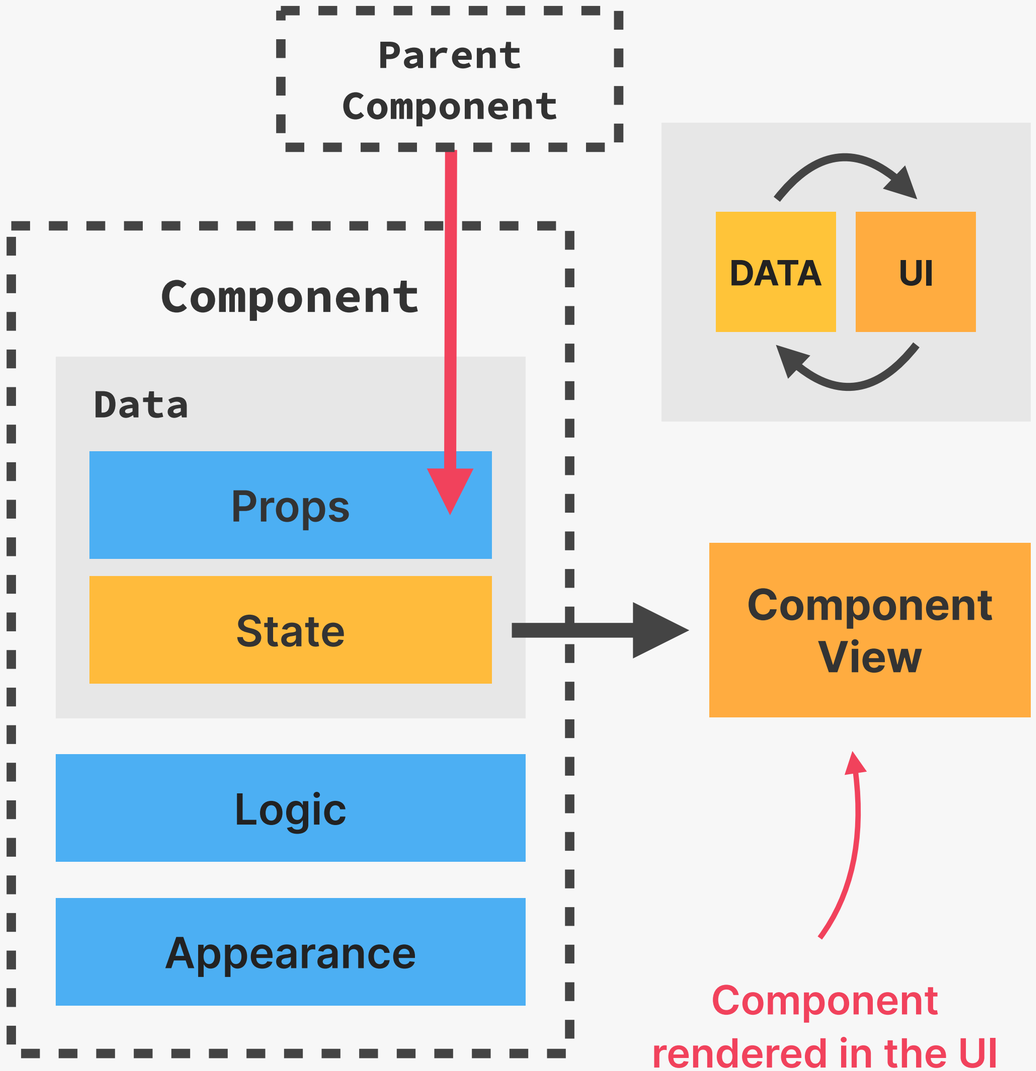
We use these terms interchangeably



WHAT IS **STATE**?

STATE

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



WHAT IS **STATE**?

STATE

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

STATE ALLOWS DEVELOPERS TO:

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

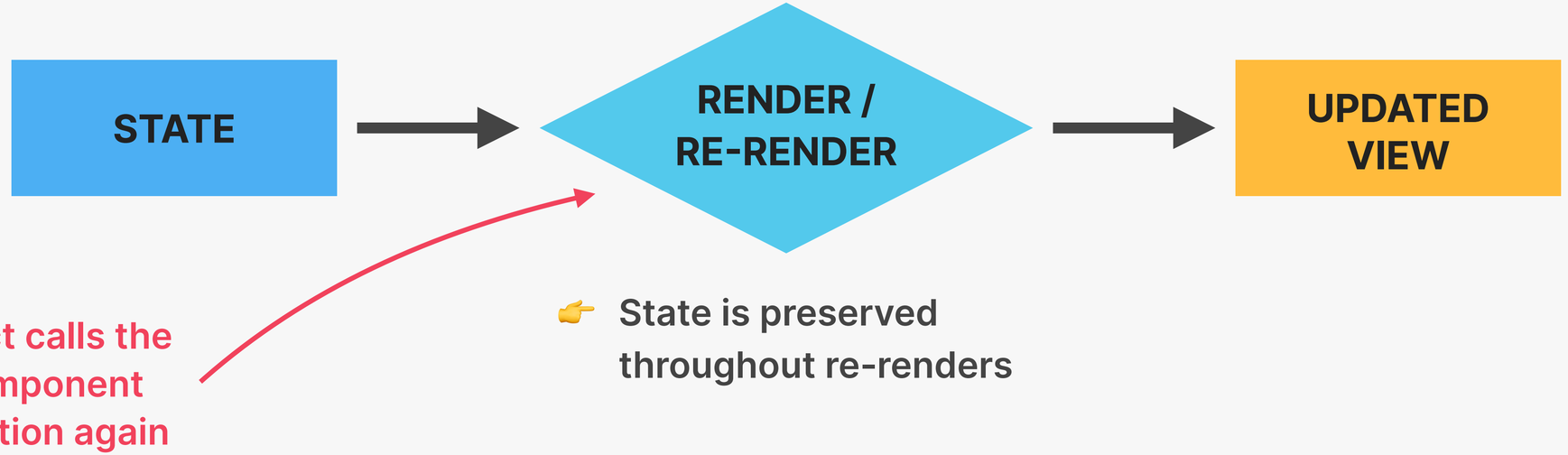
👋 *State is a **tool**. Mastering state will unlock the power of React development*

THE **MECHANICS** OF STATE IN REACT

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

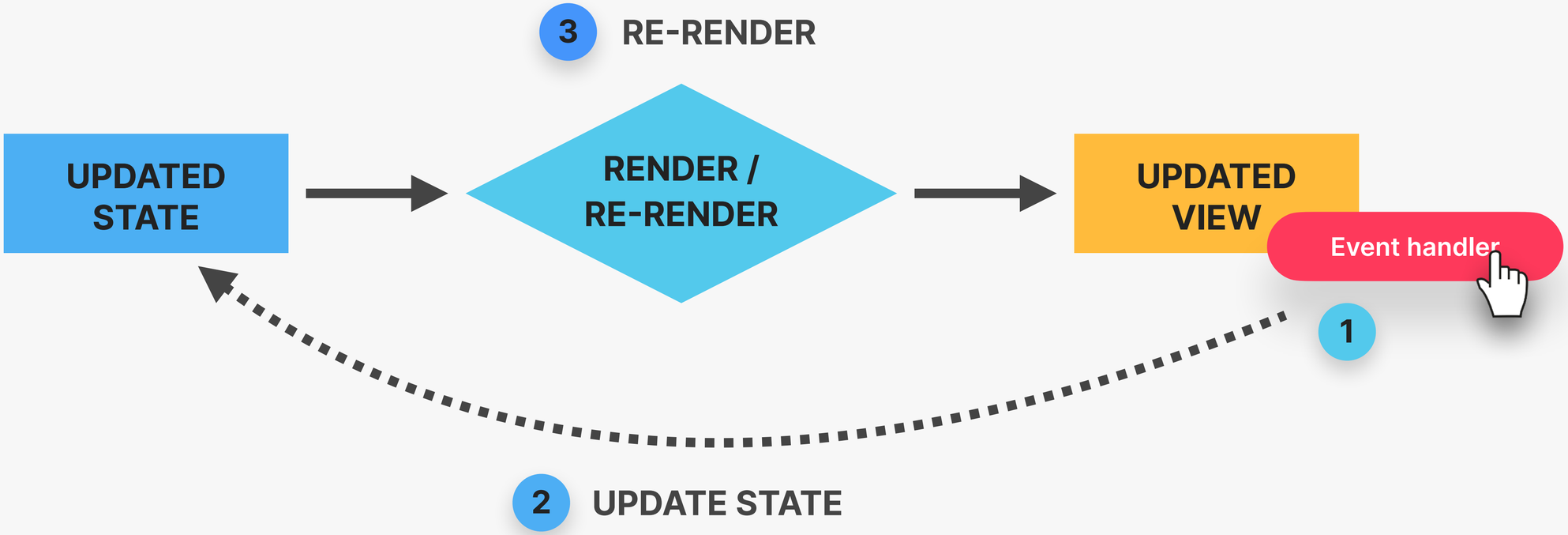
Because React is **declarative**

Important React principle



THE **MECHANICS** OF STATE IN REACT

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

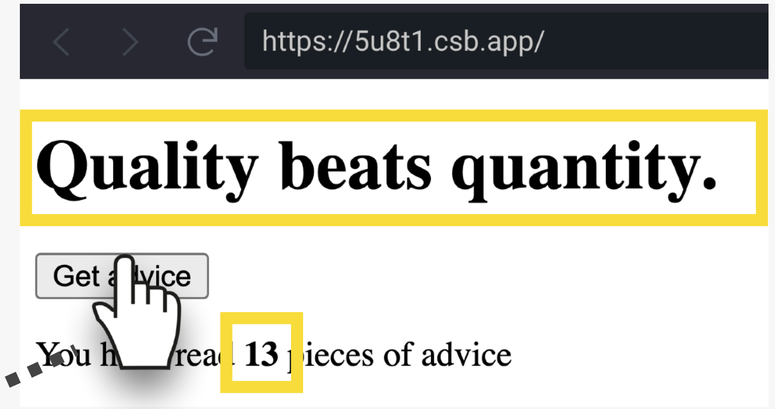


THE **MECHANICS** OF STATE IN REACT

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

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

RE-RENDER



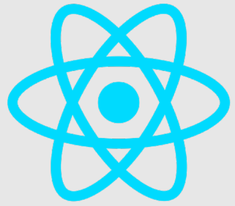
UPDATE STATE

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

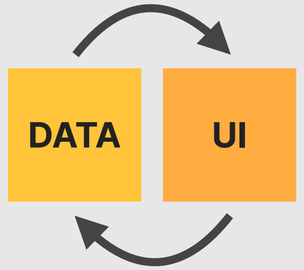
THE **MECHANICS** OF STATE IN REACT

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

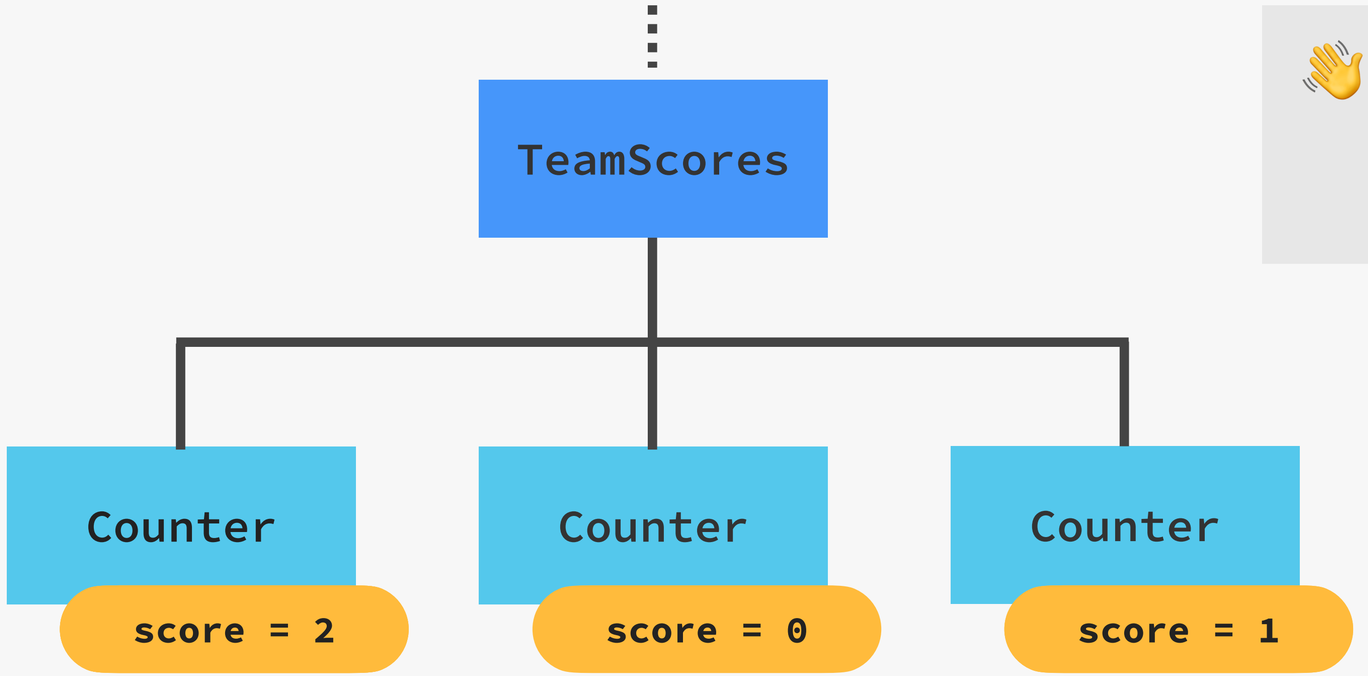
👉 *React is called "React" because...*



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



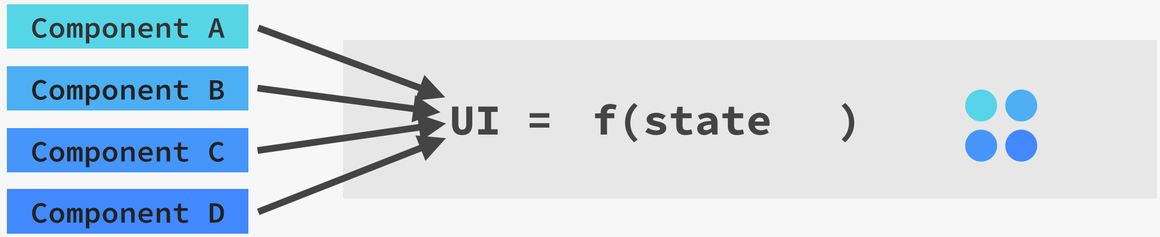
ONE COMPONENT, ONE **STATE**

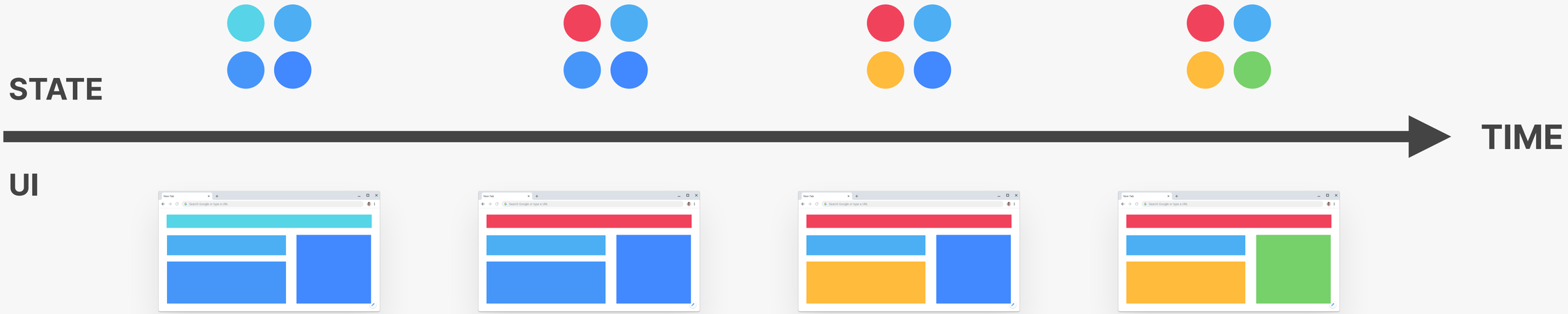


👋 Each component has and manages **its own state**, no matter how many times we render the same component

Multiple *instances* of the `Counter` component

UI AS A **FUNCTION** OF STATE





DECLARATIVE, *REVISITED*

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

🎯 IN **PRACTICAL** TERMS...

PRACTICAL GUIDELINES ABOUT STATE

- 👉 Use a state variable for any data that the component should keep track of ("remember") over time. **This is data that will change at some point**. In Vanilla JS, that's a `let` variable, or an `[]` or `{}`

- 👉 Whenever you want something in the component to be **dynamic**, create a piece of state related to that "thing", and update the state when the "thing" should change (aka "be dynamic")

  - 👉 ***Example:*** *A modal window can be open or closed. So we create a state variable* `isOpen` *that tracks whether the modal is open or not. On* `isOpen = true` *we display the window, on* `isOpen = false` *we hide it.*

- 👉 If you want to change the way a component looks, or the data it displays, **update its state.** This usually happens in an **event handler** function.

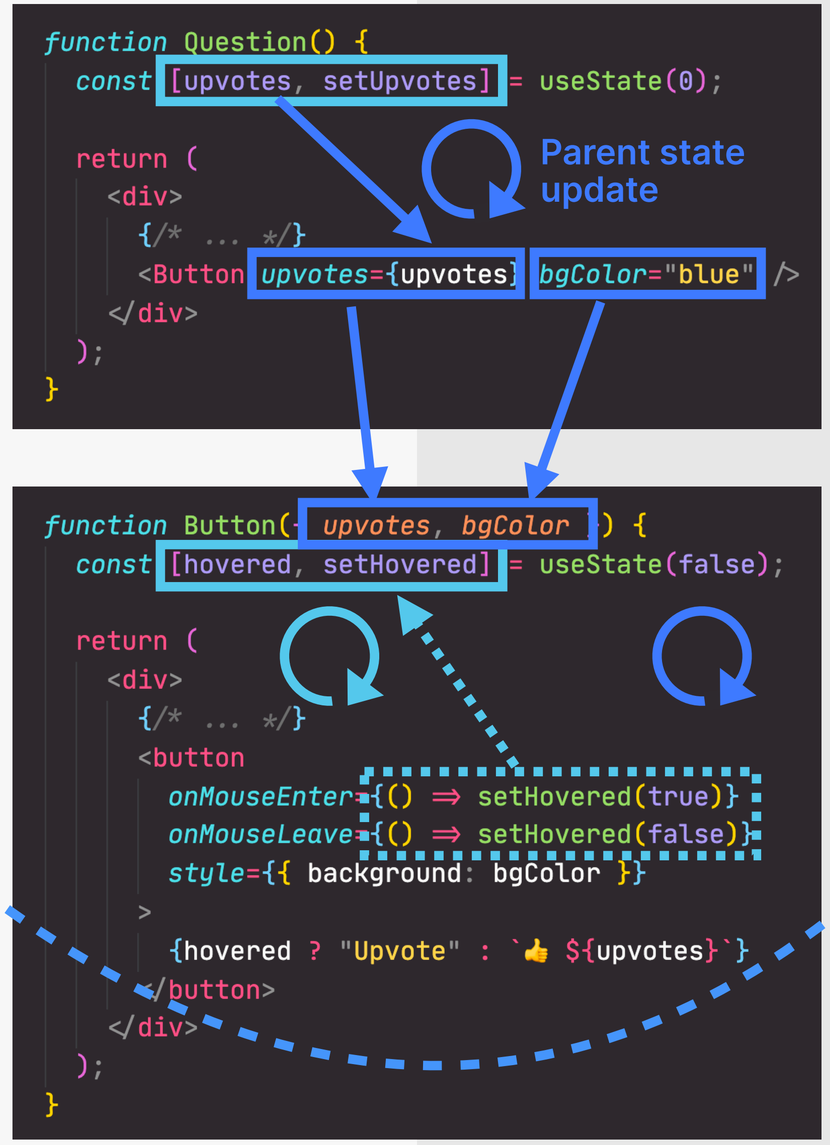
- 👉 When building a component, imagine its view as a **reflection of state changing over time**

- 👉 For data that should not trigger component re-renders, **don't use state**. Use a regular variable instead. This is a common **beginner mistake**.

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

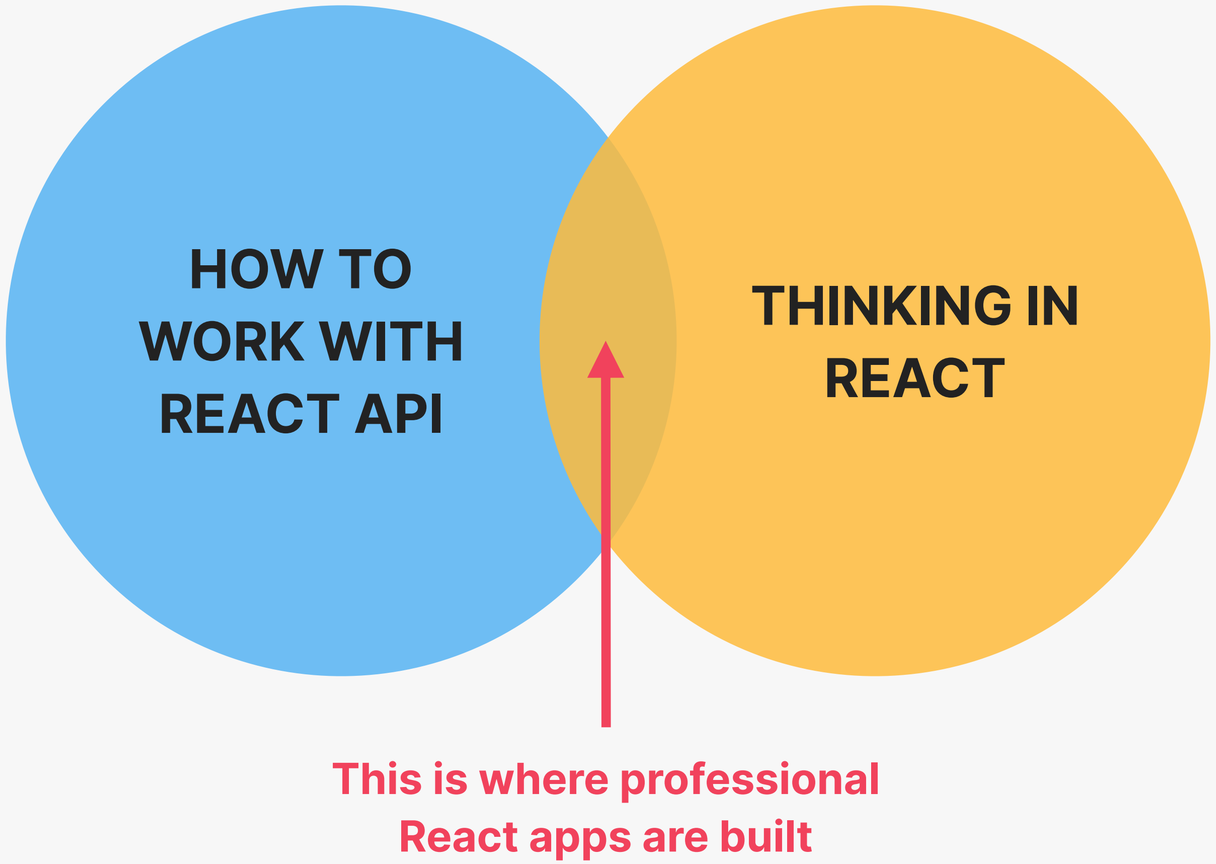


PROPS

- 👉 **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



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:

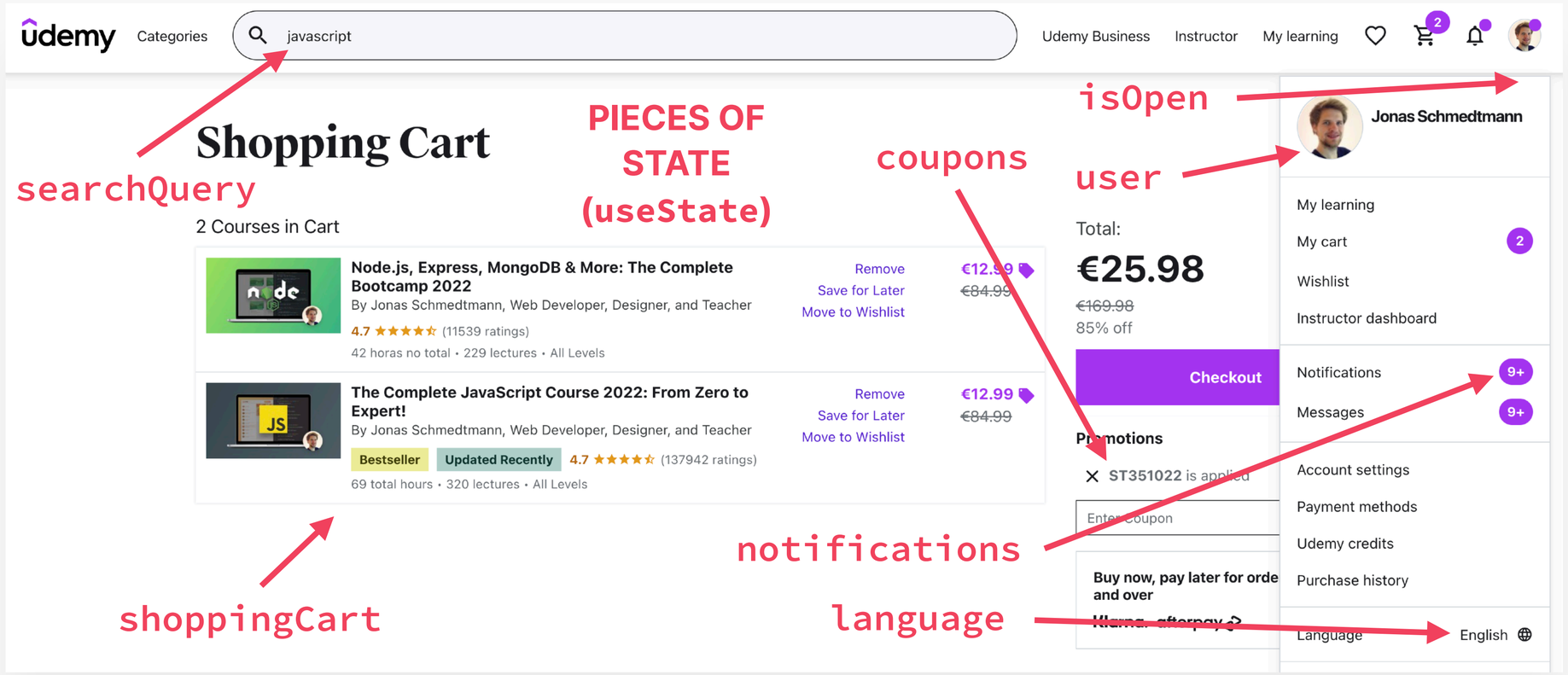
- 🤔 How to break up a UI design into components?
- 🤔 How to make some components reusable?
- 🤔 How to assemble UI from reusable components?
- 🤔 What pieces of state do I need for interactivity?
- 🤔 Where to place state? (What component should "own" each piece of state?)
- 🤔 What types of state can or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

→

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



TYPES OF STATE: LOCAL VS. GLOBAL STATE

LOCAL STATE

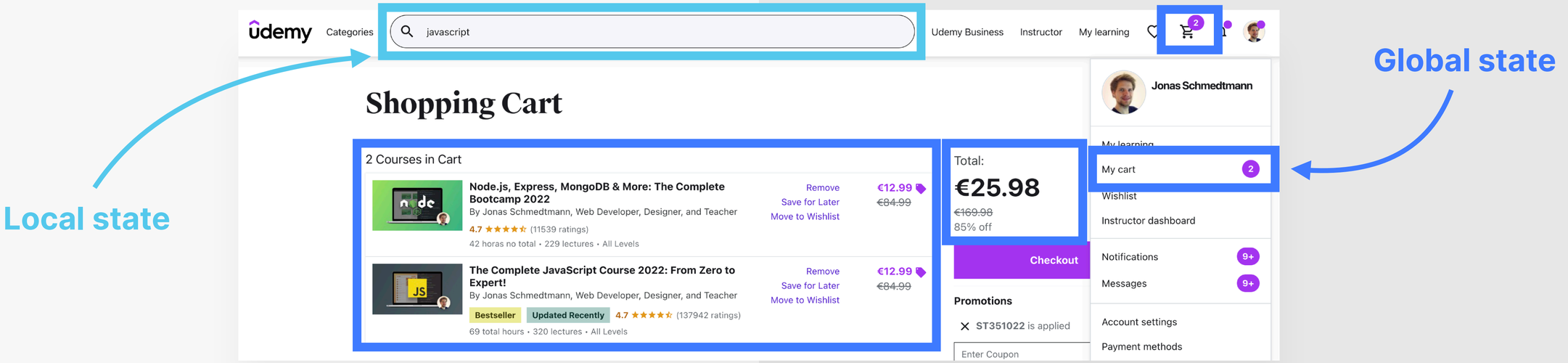
- 👉 State needed **only by one or few components**
- 👉 State that is defined in a component and **only that component and child components** have access to it (by passing via props)
- ☝️ ***We should always start with local state***

GLOBAL STATE

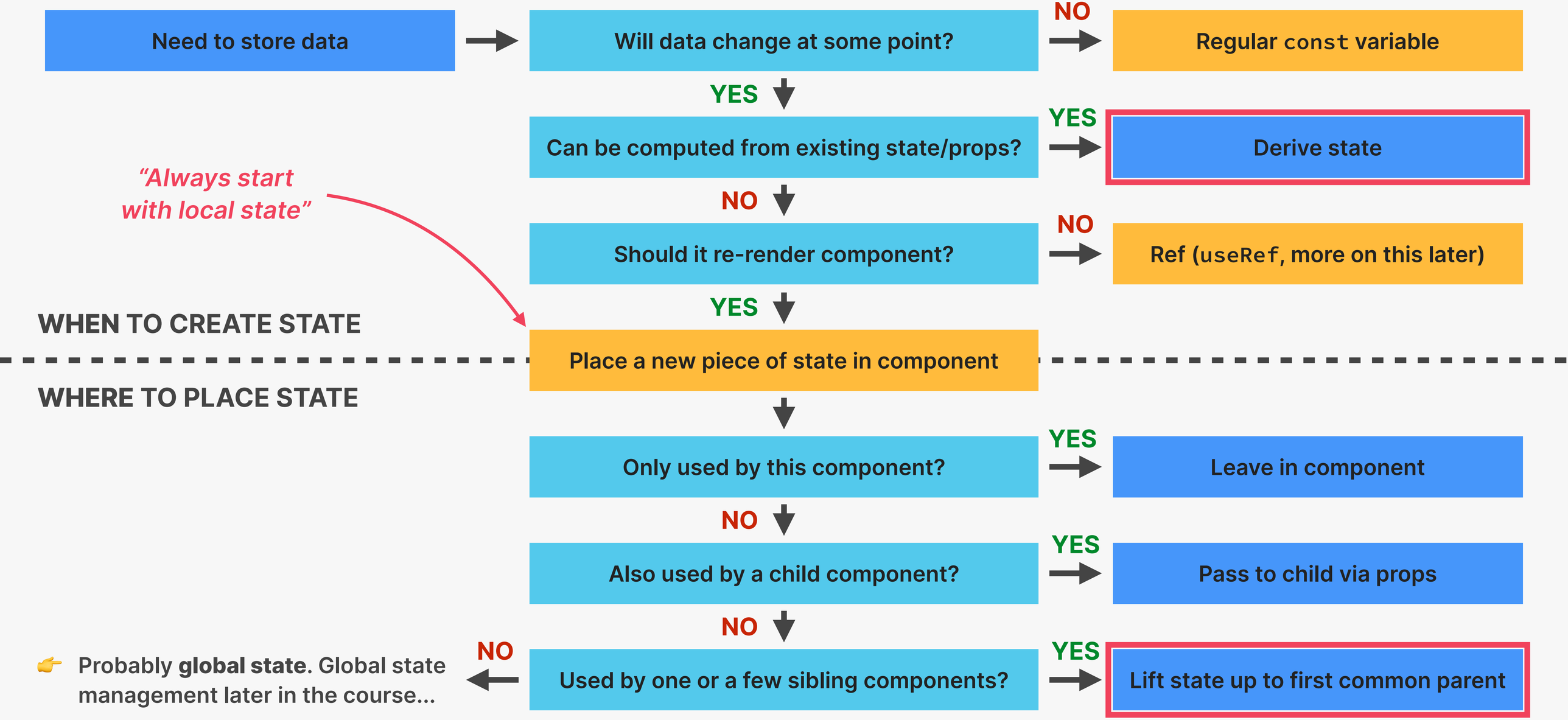
- 👉 State that **many components** might need
- 👉 **Shared** state that is accessible to **every component** in the entire application

Context API

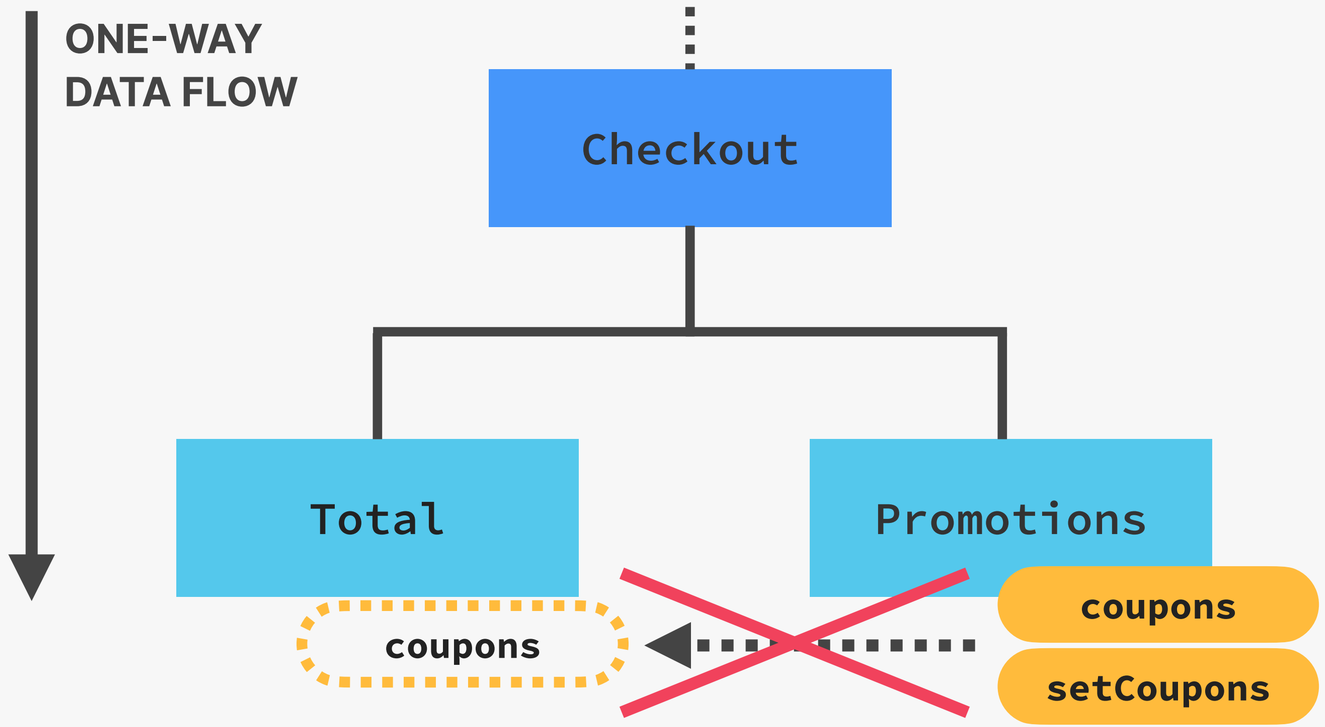
Redux



STATE: **WHEN** AND **WHERE**?



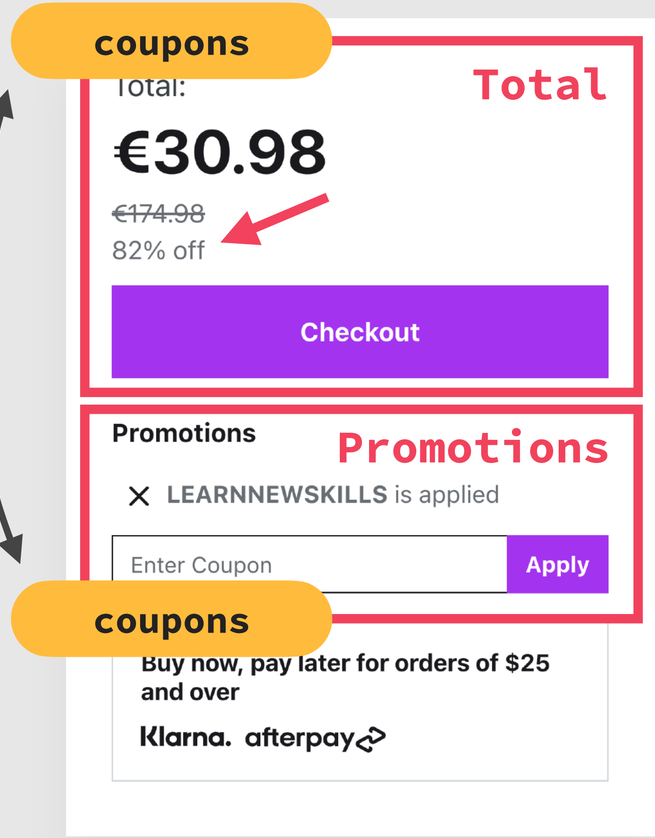
PROBLEM: SHARING STATE WITH SIBLING COMPONENT



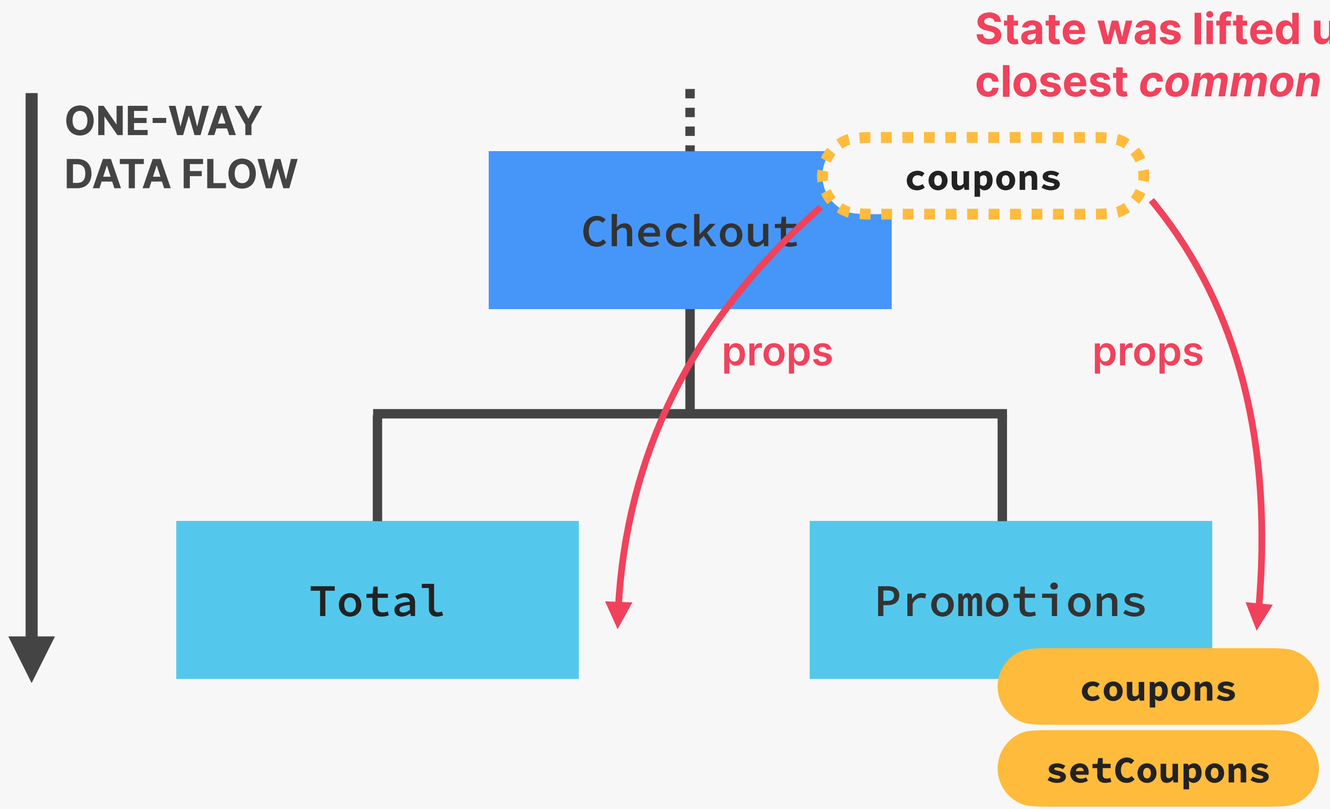
Data can only flow **down to children** (via props), not **sideways to siblings**

🤔 How do we share state with other components?

👉 `Total` component also needs access to `coupons` state

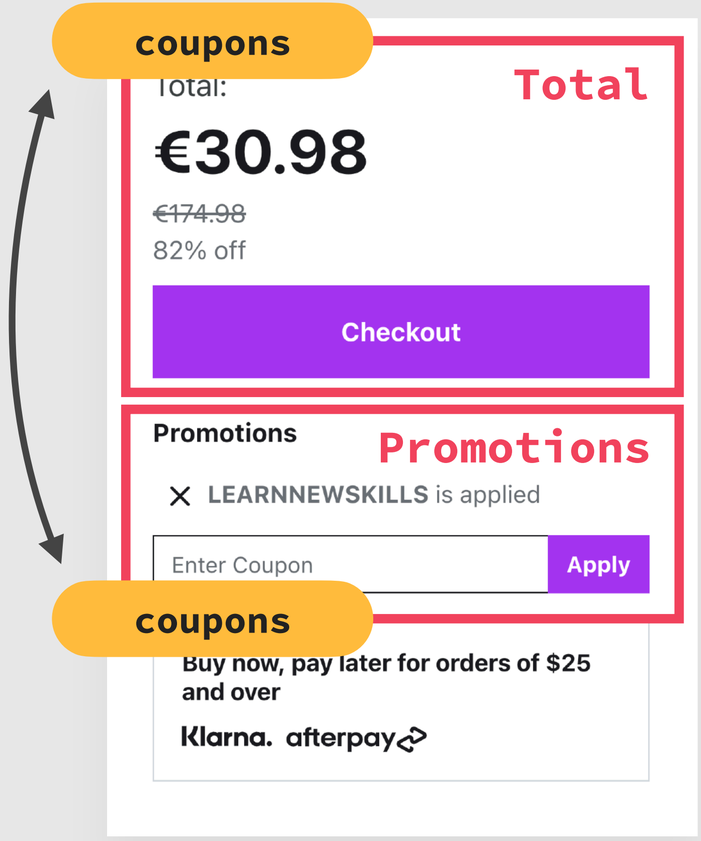


SOLUTION: LIFTING STATE UP



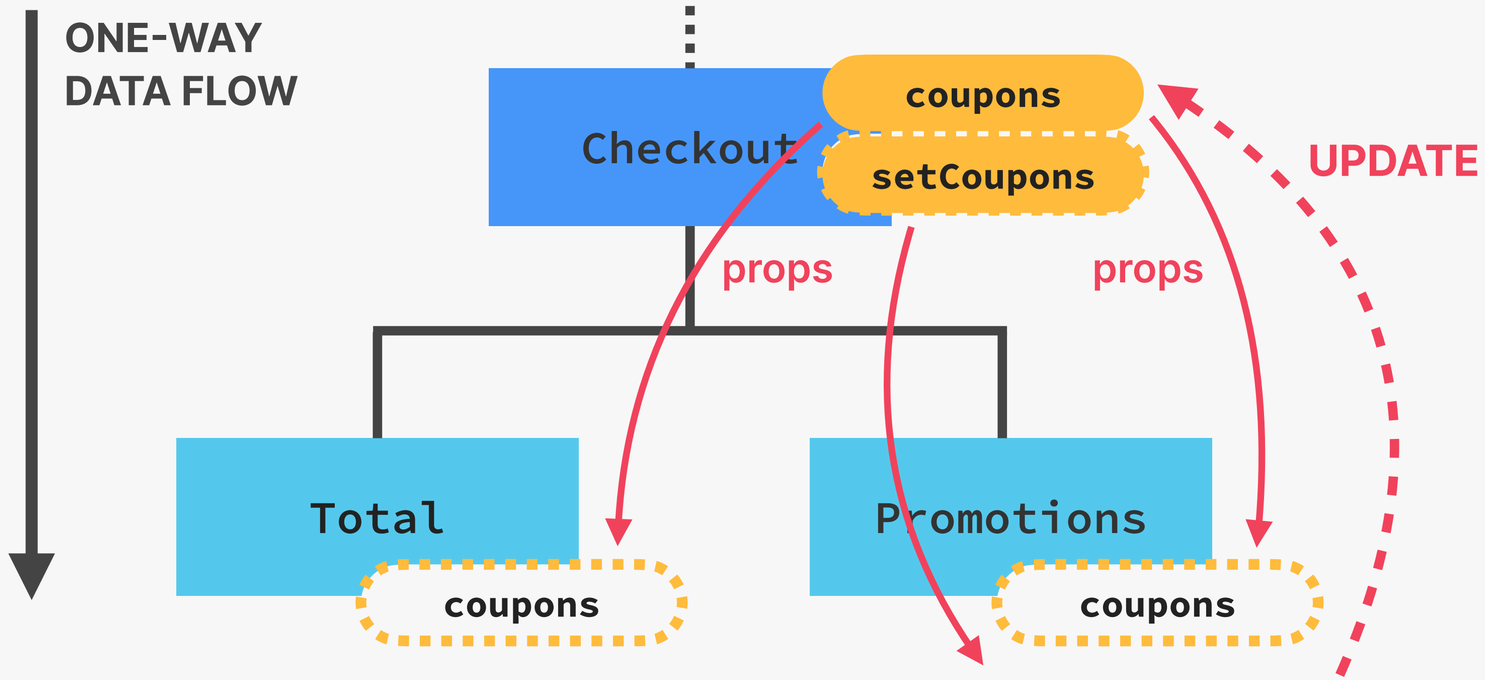
✌️ By **lifting state up**, we have successfully **shared** one piece of state with multiple components in **different positions** in the component tree

👉 `Total` component also needs access to `coupons` state



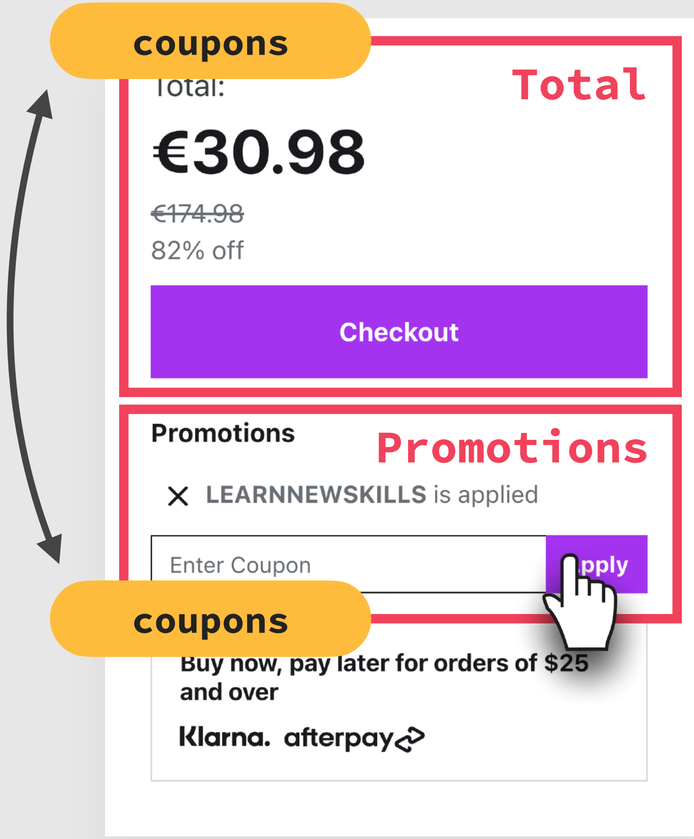
CHILD-TO-PARENT COMMUNICATION

👉 **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

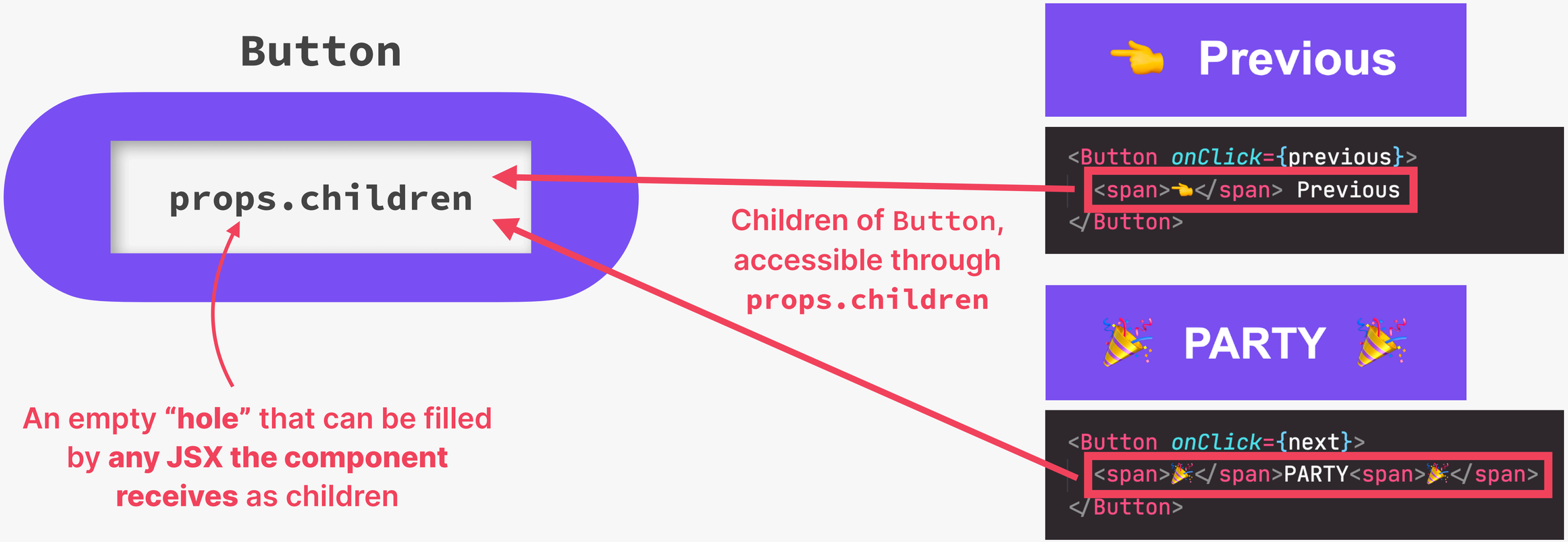
👍 Just regular variables, no `useState`

👍 `cart` state is the **single source of truth** for this related data

👍 Works because re-rendering component will **automatically re-calculate** derived state

```
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



- 👉 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**

Just one huge component

COMPONENT SIZE

SMALL

HUGE

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

COMPONENT SIZE **MATTERS**

Many small components

COMPONENT SIZE

SMALL

HUGE

Generally, we need to find the right balance between too specific and too broad

- 👉 We end up with 100s of mini-components
- 👉 Confusing codebase
- 👉 Too **abstracted**

Creating something new to hide the implementation details of that thing

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

HOW TO **SPLIT** A UI INTO COMPONENTS

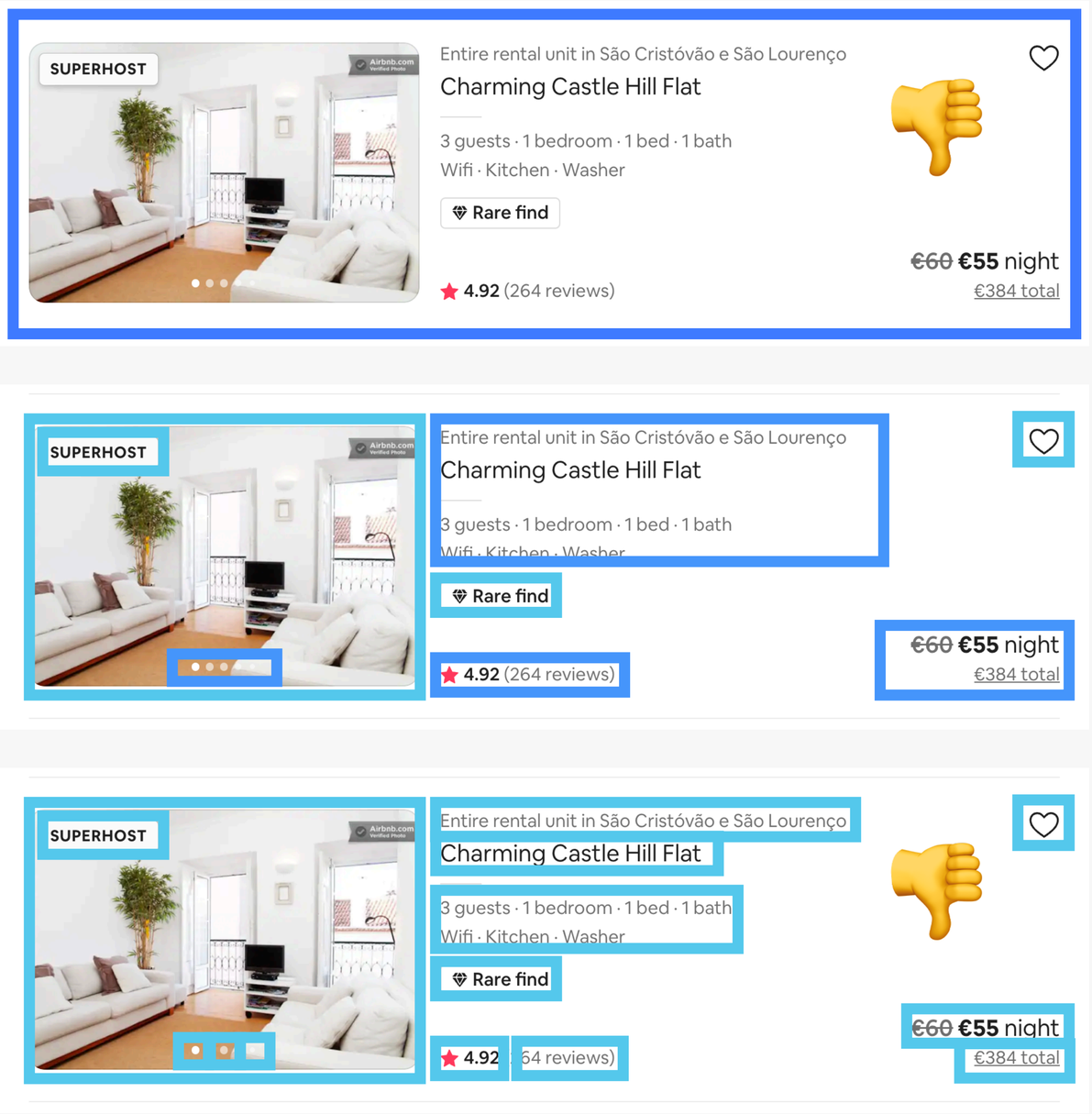
👉 **The 4 criteria for splitting a UI into components:**

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

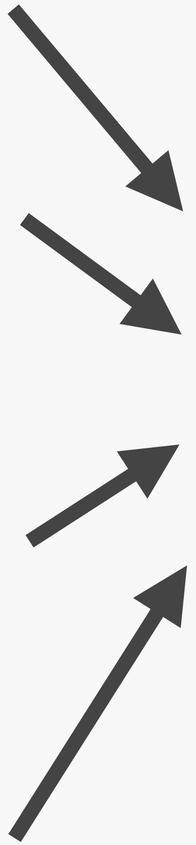
- 👉 Is it possible to reuse part of the component?
- 👉 Do you **want** or **need** to reuse it?

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

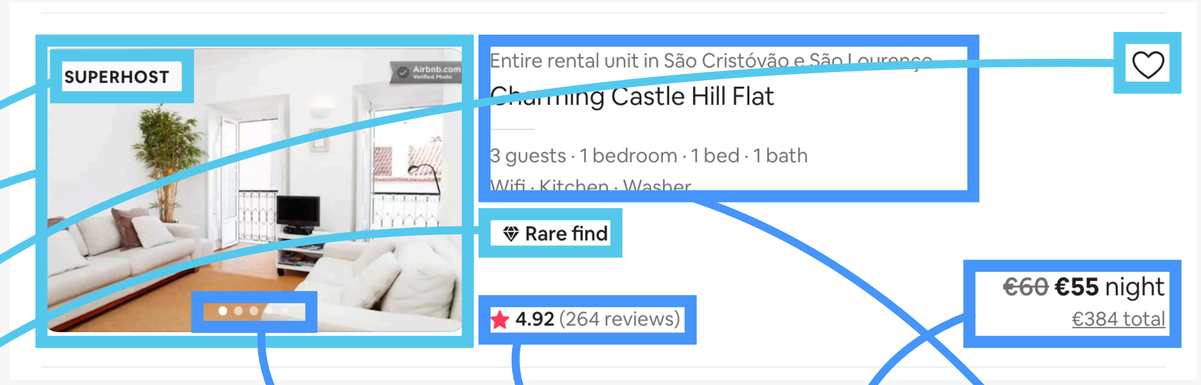
🏷️ Name a component according to **what it does** or **what it displays**. Don’t be afraid of using long component names

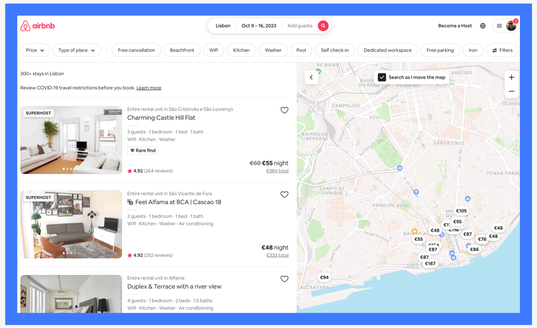
🪆 Never declare a new component **inside another component**!

🗃️ **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**





👉 **Page component**

SMALL

REUSABLE

Components with all different sizes

Different degrees of size, reusability, responsibility, and complexity

HUGE

NON-REUSABLE

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

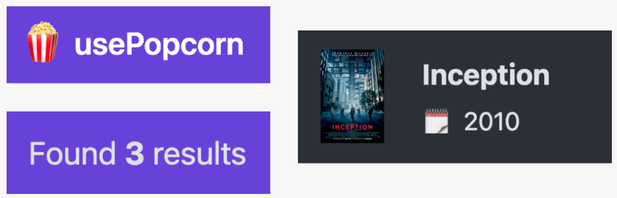
- 👉 **Most apps will have a few huge components**
- 👉 Not meant to be reused (**not a problem!**)

COMPONENT **CATEGORIES**

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

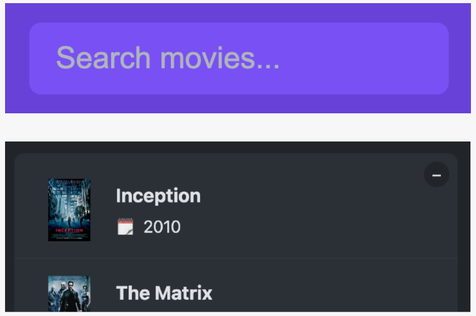
Stateless / presentational components

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



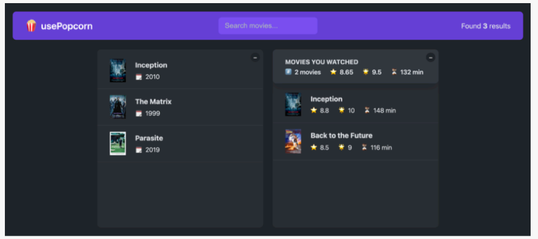
Stateful components

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



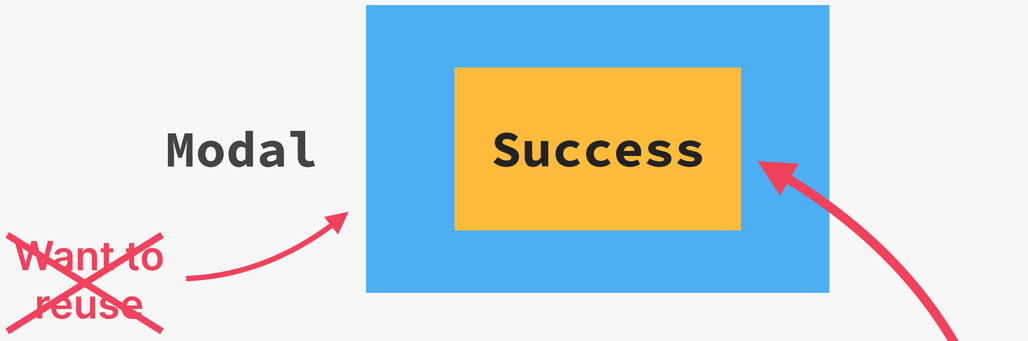
Structural components

- 👉 “**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




```
function Modal() {
  return (
    <div className="modal">
      <Success />
    </div>

function Success() {
  return <p>Well done!</p>;
```

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

COMPONENT COMPOSITION

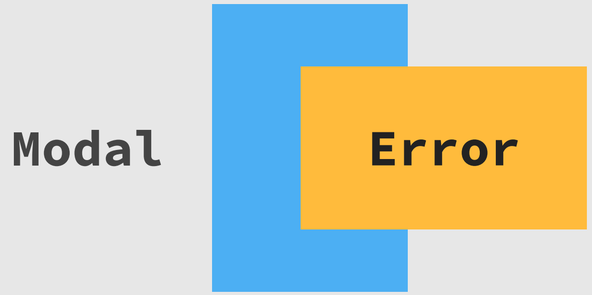

```
function Modal({ children }) {
  return (
    <div className="modal">
      {children}
    </div>

function Error() {
  return <p>This went wrong!</p>;
```

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

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

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

WHAT IS COMPONENT **COMPOSITION**?

COMPONENT COMPOSITION




```jsx
function Modal({ children }) {
  return (
    <div className="modal">
      {children}
    </div>
```

```jsx
function Error() {
  return <p>This went wrong!</p>;
```

```jsx
<Modal>
  <Success />
</Modal>
```

```jsx
<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

App

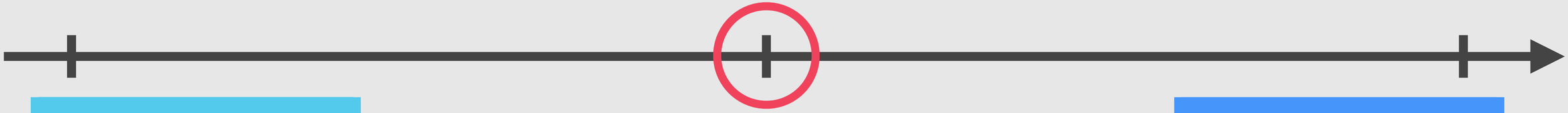
Component props = Public API

Consuming component

Abstraction that encapsulates UI and logic

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

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

- 👉 **Description** of a piece of UI
- 👉 A component is a function that **returns React elements** (element tree), usually written as JSX
- 👉 “Blueprint” or “Template”

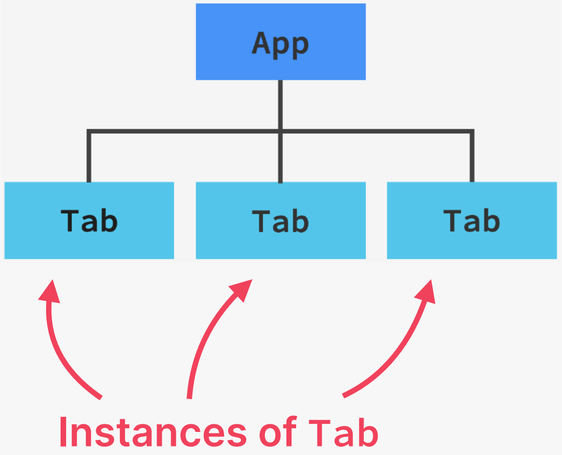
COMPONENT **VS.** INSTANCE **VS.** ELEMENT

Component

<Tab />

Component Instance

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



- 👉 Instances are created when we “**use**” components
- 👉 React internally calls `Tab()`
- 👉 Actual “**physical**” **manifestation** of a component
- 👉 Has its own state and props
- 👉 Has a **lifecycle** (can “be born”, “live”, and “die”)

COMPONENT **VS.** INSTANCE **VS.** ELEMENT

Component

↓ `<Tab />`

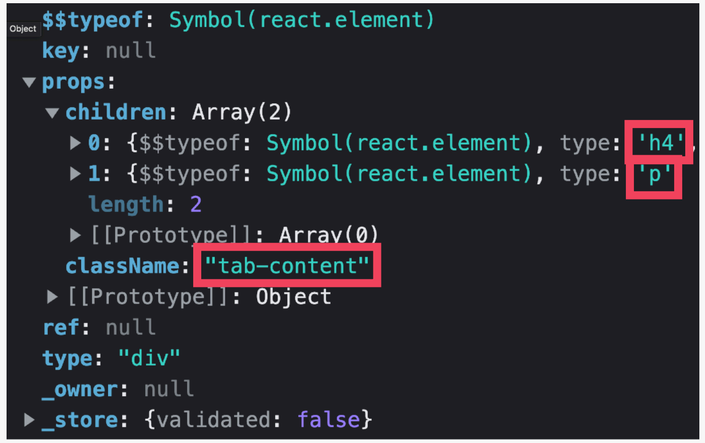
Component Instance

↓ **RETURNS**

React Element

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

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



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

Component

↓ `<Tab />`

Component Instance

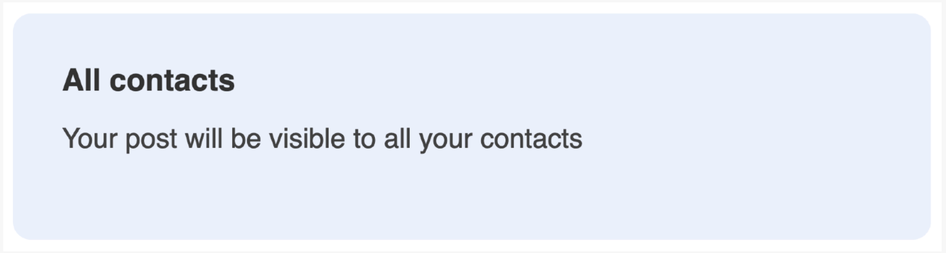
↓ **RETURNS**

React Element

↓ **INSERTED TO DOM**

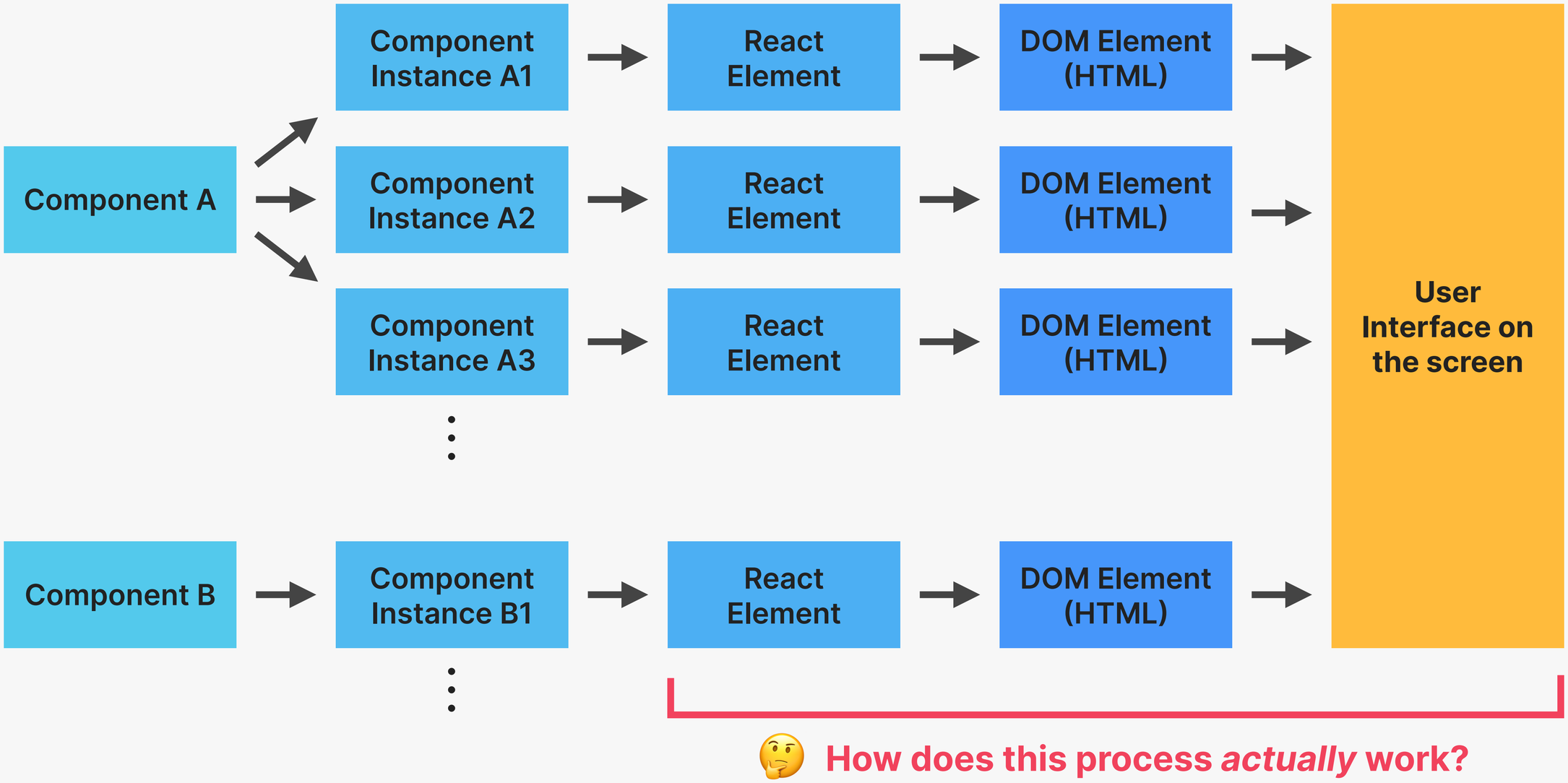
DOM Element (HTML)

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



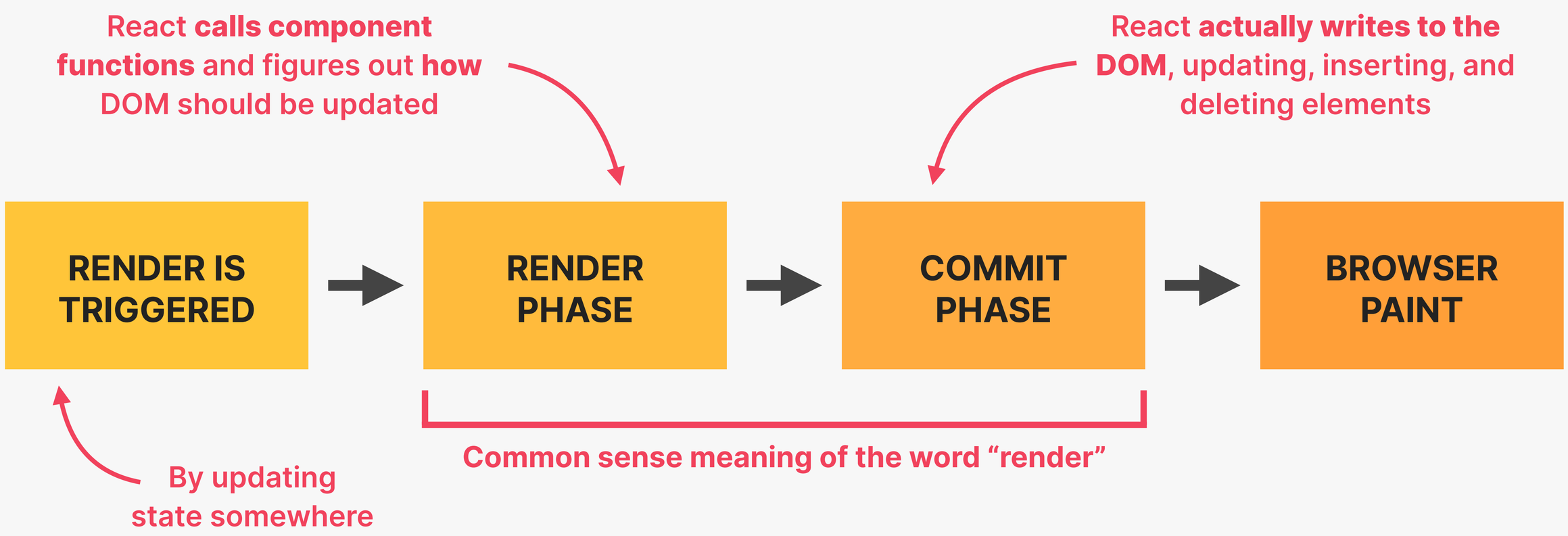
👉 Actual **visual representation** of the component instance in the browser

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:

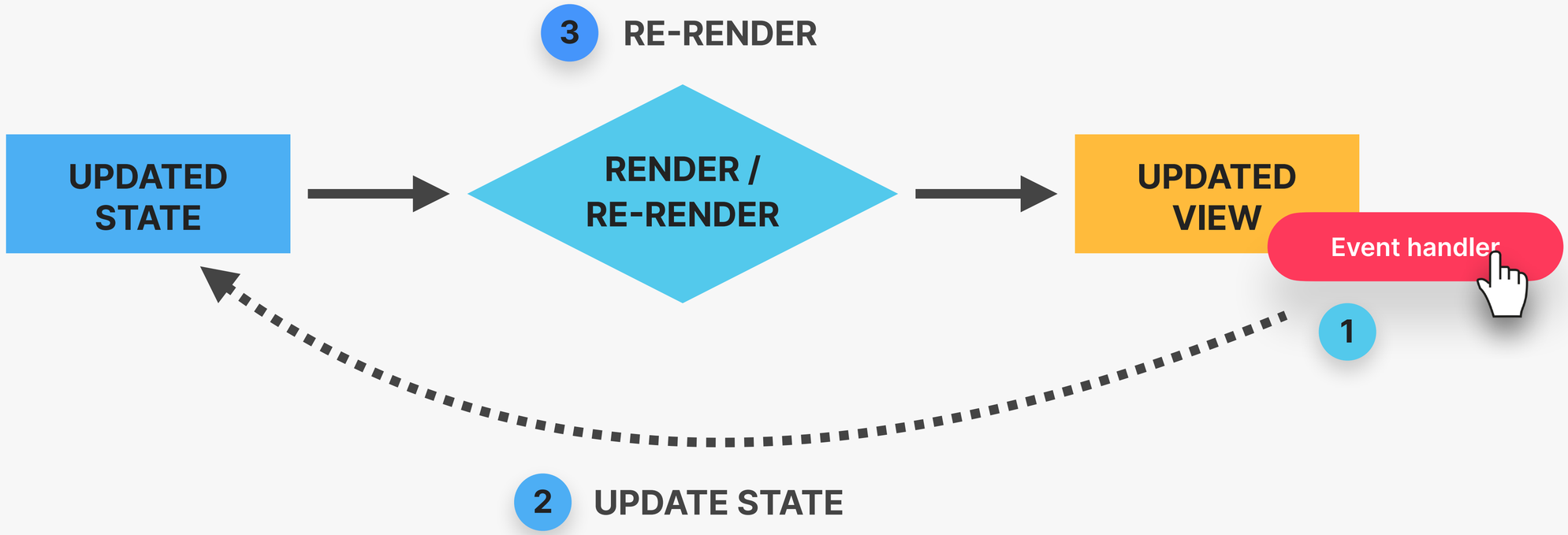
1. **Initial render** of the application
2. **State is updated** in one or more component instances (**re-render**)

👉 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

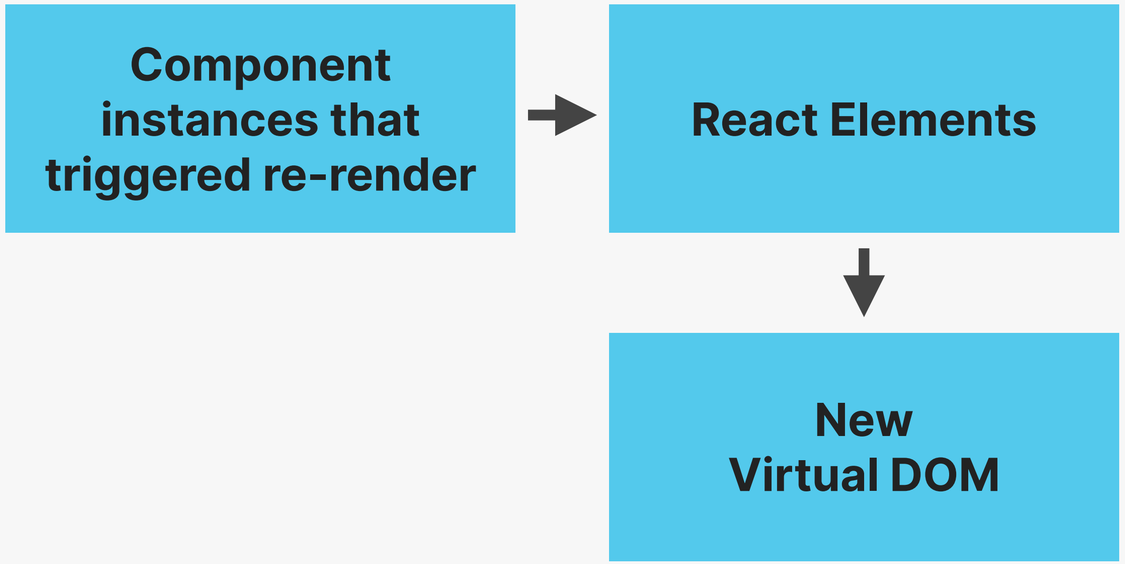


NOT TRUE #1: RENDERING IS UPDATING THE SCREEN / DOM

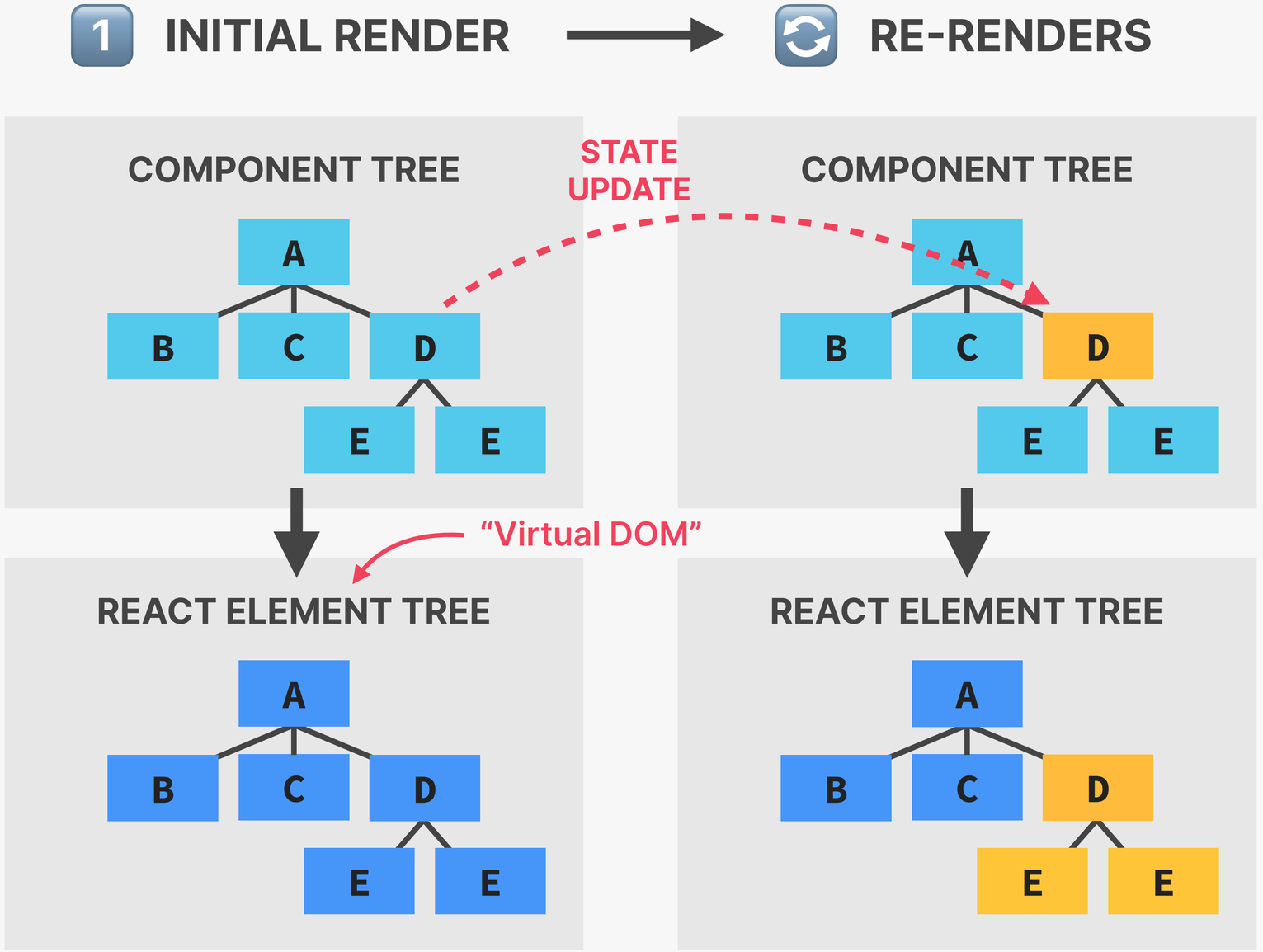
NOT TRUE #2: REACT COMPLETELY DISCARDS OLD VIEW (DOM) ON RE-RENDER

THE **RENDER** PHASE

[2] RENDER PHASE



THE **VIRTUAL DOM** (REACT ELEMENT TREE)

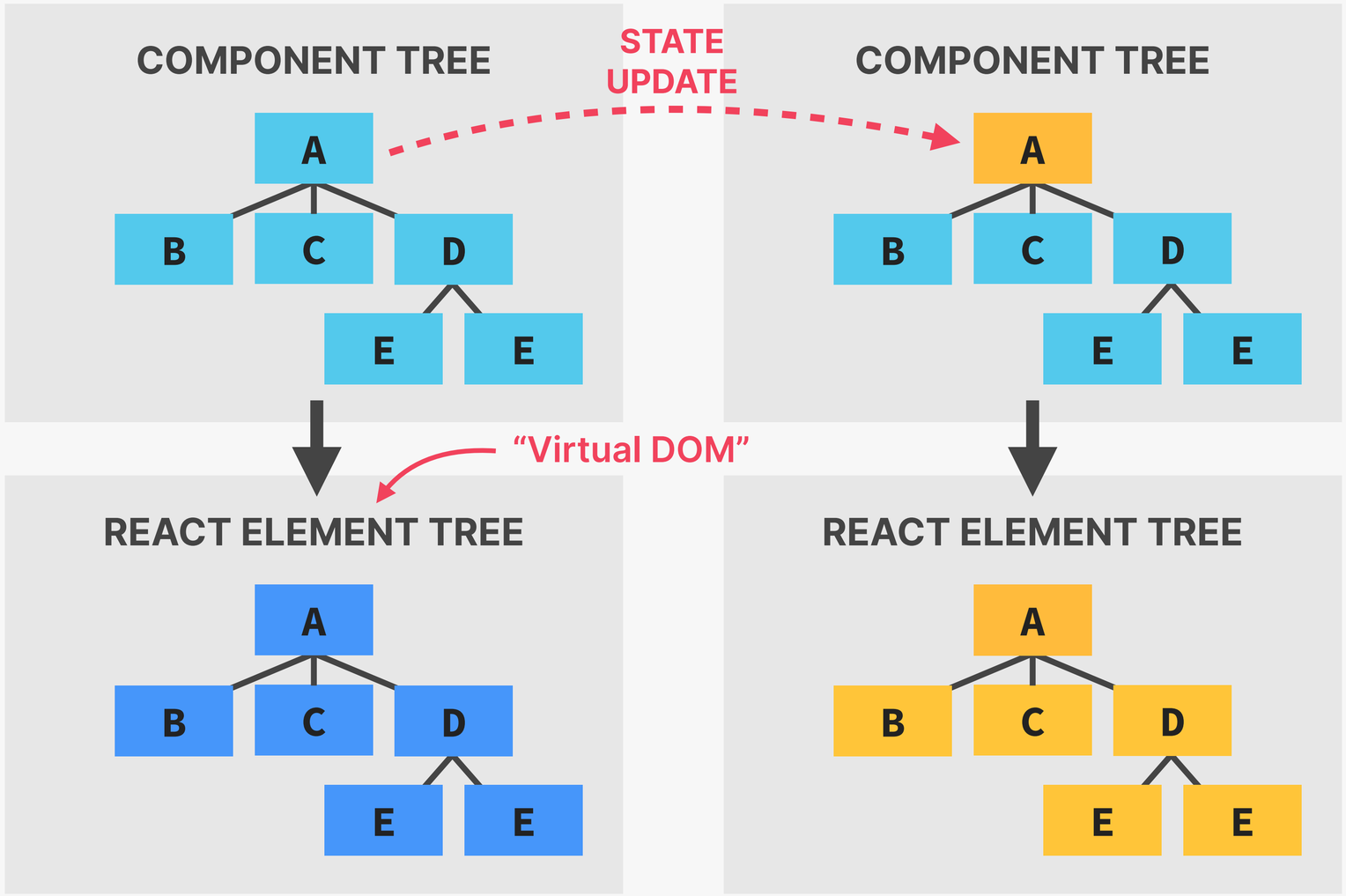


- 👉 **Virtual DOM:** Tree of all React elements created from all instances in the component tree
- 👉 Cheap and fast to create multiple trees
- 👉 Nothing to do with "shadow DOM"

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

THE **VIRTUAL DOM** (REACT ELEMENT TREE)

1 INITIAL RENDER → 🔄 RE-RENDERS



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

👉 Cheap and fast to create multiple trees

👉 Nothing to do with "shadow DOM"

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

⬇

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

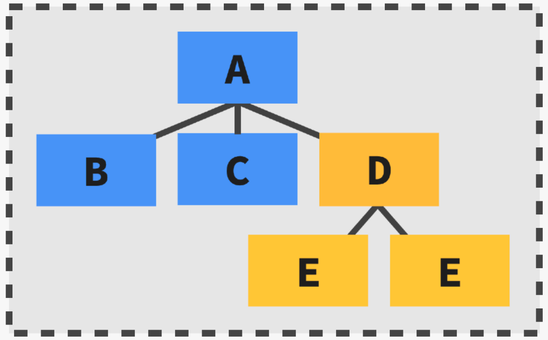
THE **RENDER** PHASE

[2] RENDER PHASE

Component instances that triggered re-render

React Elements

New Virtual DOM



Reconciler called “Fiber”

Current Fiber tree (*before* state update)

Reconciliation + Diffing

Updated Fiber tree

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

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

↓ **BECAUSE**

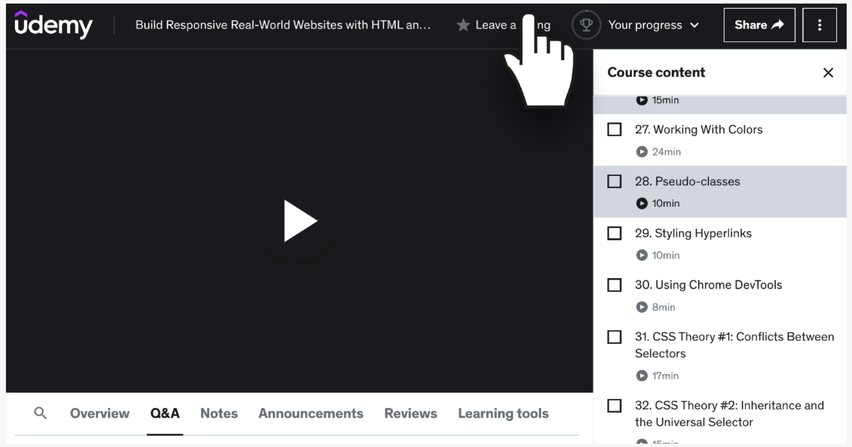
👉 That would be inefficient and wasteful:

1. Writing to the DOM is (relatively) **slow**
2. Usually only a **small part of the DOM** needs to be updated

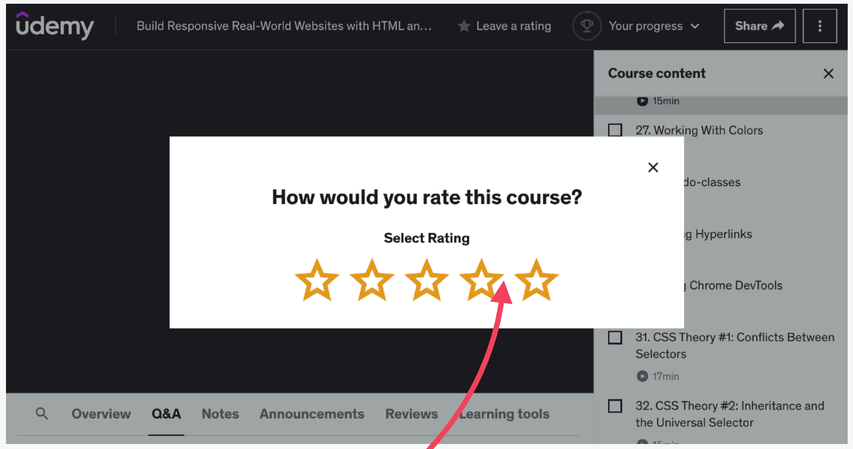
👉 React **reuses** as much of the existing DOM as possible

↓ **HOW?**

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



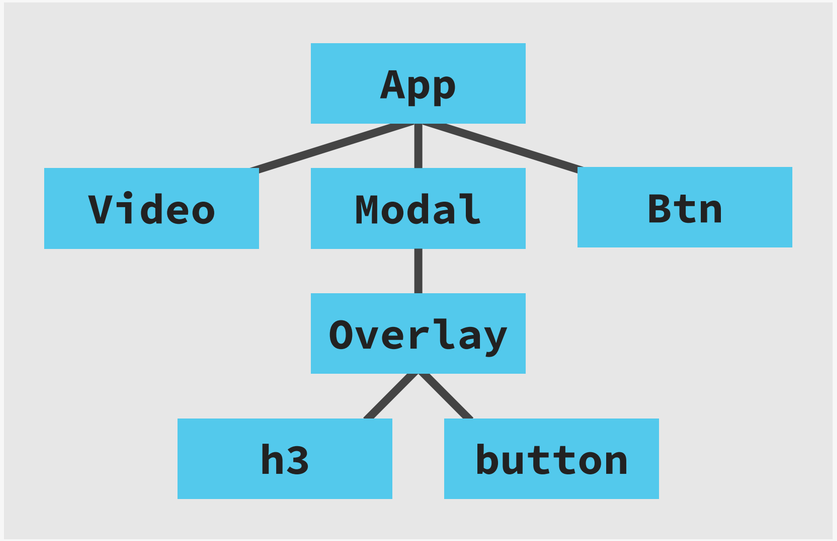
↓ `showModal = true`



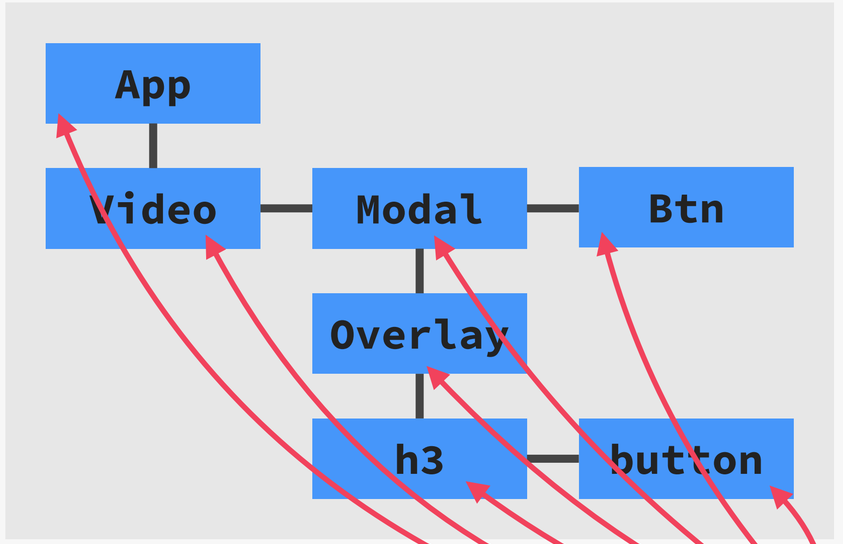
Only these new DOM elements are created

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**

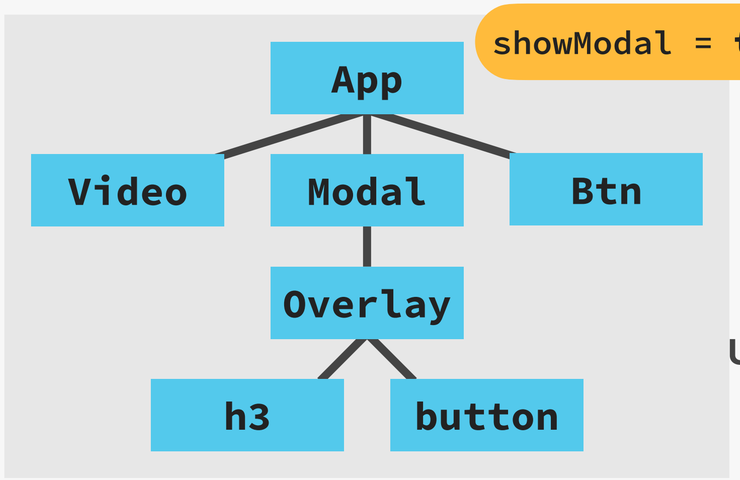
- 👉 Rendering process can be split into chunks, tasks can be prioritized, and work can be **paused**, **reused**, or **thrown away**
  - 👉 Enables **concurrent features** like Suspense or transitions
  - 👉 Long renders **won’t block** JS engine

FIBER

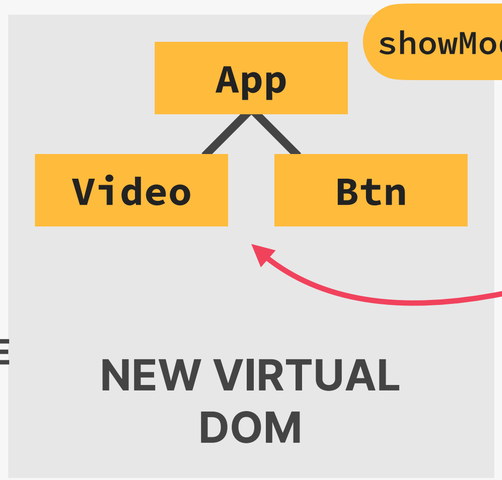
“Unit of work”

- Current state
- Props
- Side effects
- Used hooks
- Queue of work

RECONCILIATION IN ACTION



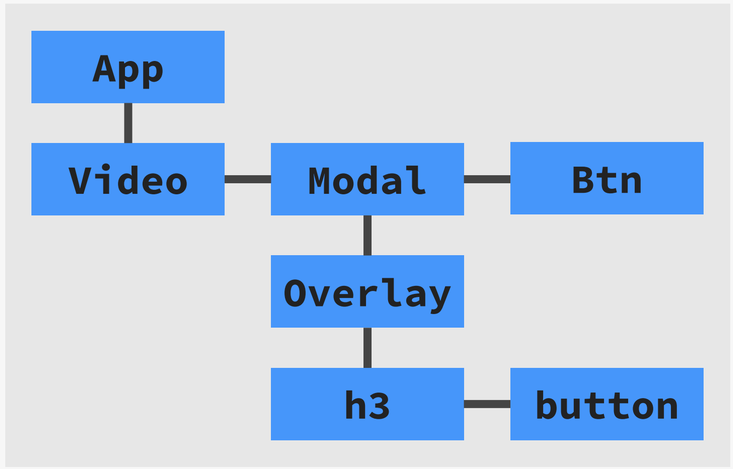
STATE UPDATE



NEW VIRTUAL DOM

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

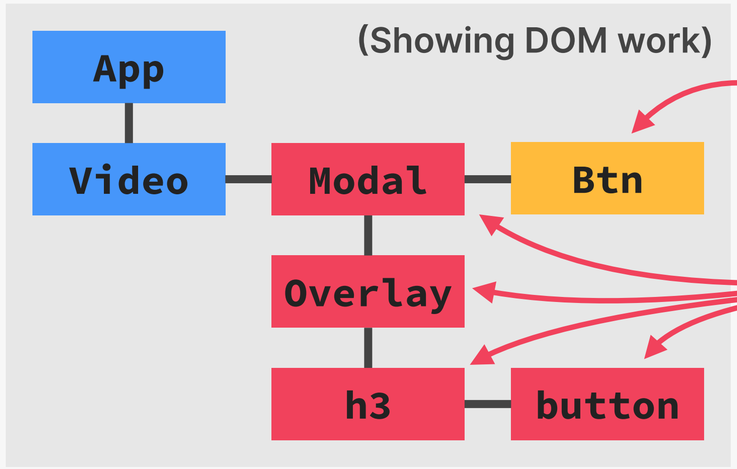
```
<App>
  <Video />
  {showModal && (
    <Modal>
      <Overlay>
        <h3>Rate course!</h3>
        <button>5 ⭐</button>
      </Overlay>
    </Modal>
  )}
  <Btn>
    {showModal ? 'Rate' : 'Hide'}
  </Btn>
</App>
```



CURRENT FIBER TREE

Reconciliation + Diffing

Comparing elements based on their **position in the tree**

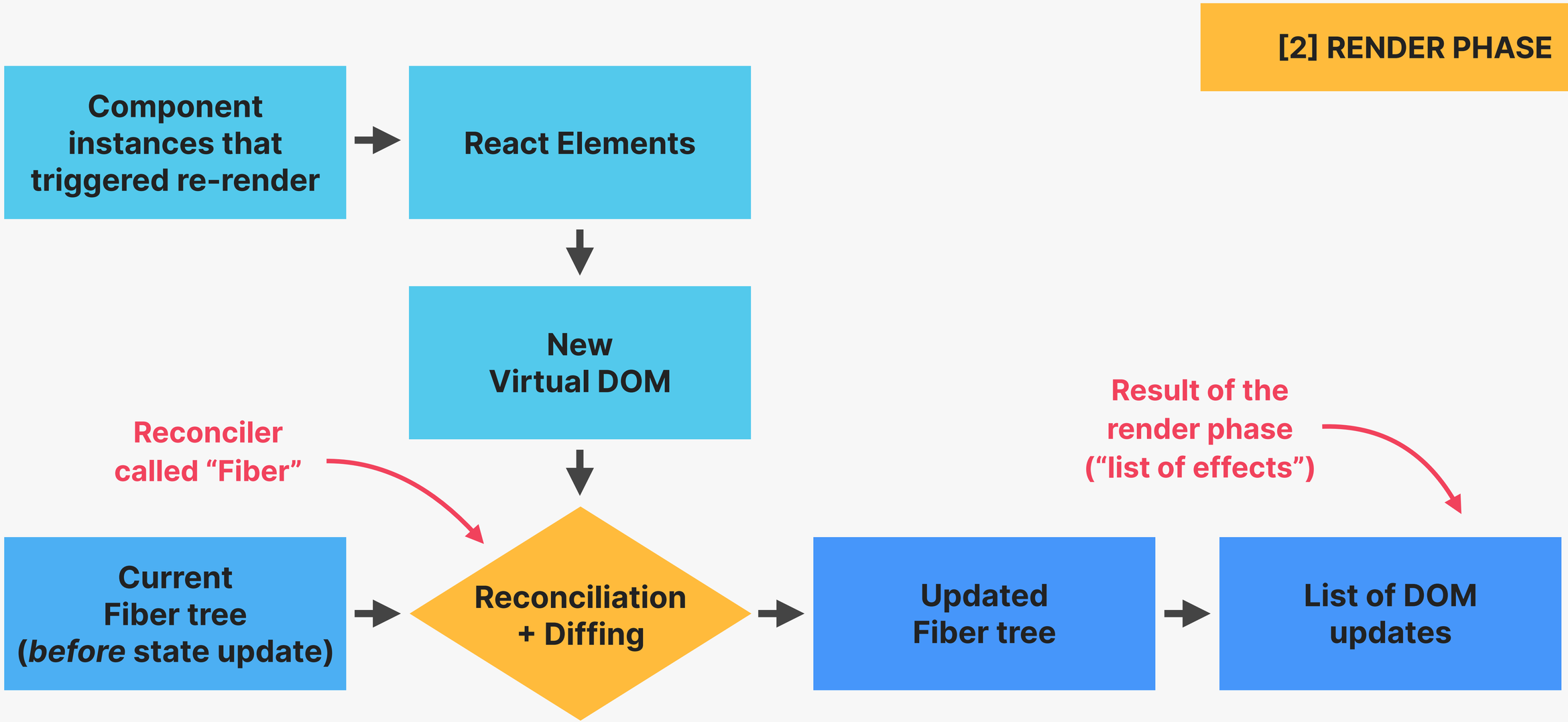


DOM Update (text)

DOM Deletions

UPDATED FIBER TREE (workInProgress)

THE **RENDER** PHASE



THE **COMMIT** PHASE AND BROWSER **PAINT**

[2] RENDER PHASE

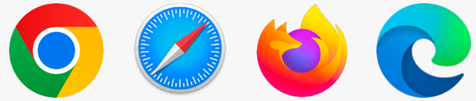
[3] COMMIT PHASE

[4] BROWSER PAINT

List of DOM updates

Updated DOM

Updated UI on the screen



- 👉 **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**

[2] RENDER PHASE

[3] COMMIT PHASE

[4] BROWSER PAINT

List of DOM updates

Updated DOM

Updated UI on the screen

React

Does **not** touch the DOM. React **only renders**. It doesn't know where the render result will go

React can be used on different platforms ("hosts")

"RENDERERS"

ReactDOM

React Native

Remotion

Many others...

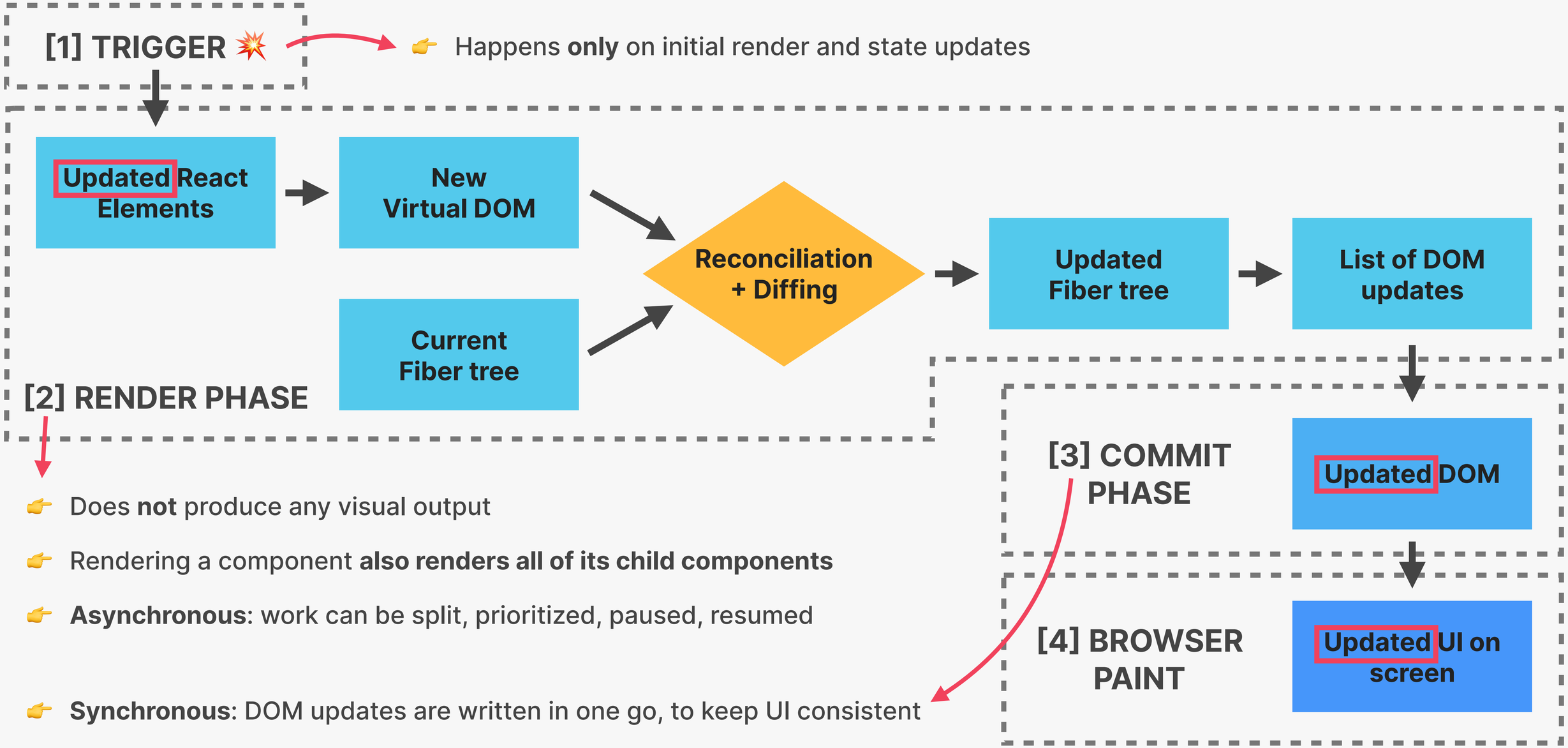
iOS

android

PDF

Terrible name! **Renderers do not render**, they ***commit*** the result of render phase...

RECAP: PUTTING IT ALL TOGETHER



👉 Does **not** produce any visual output

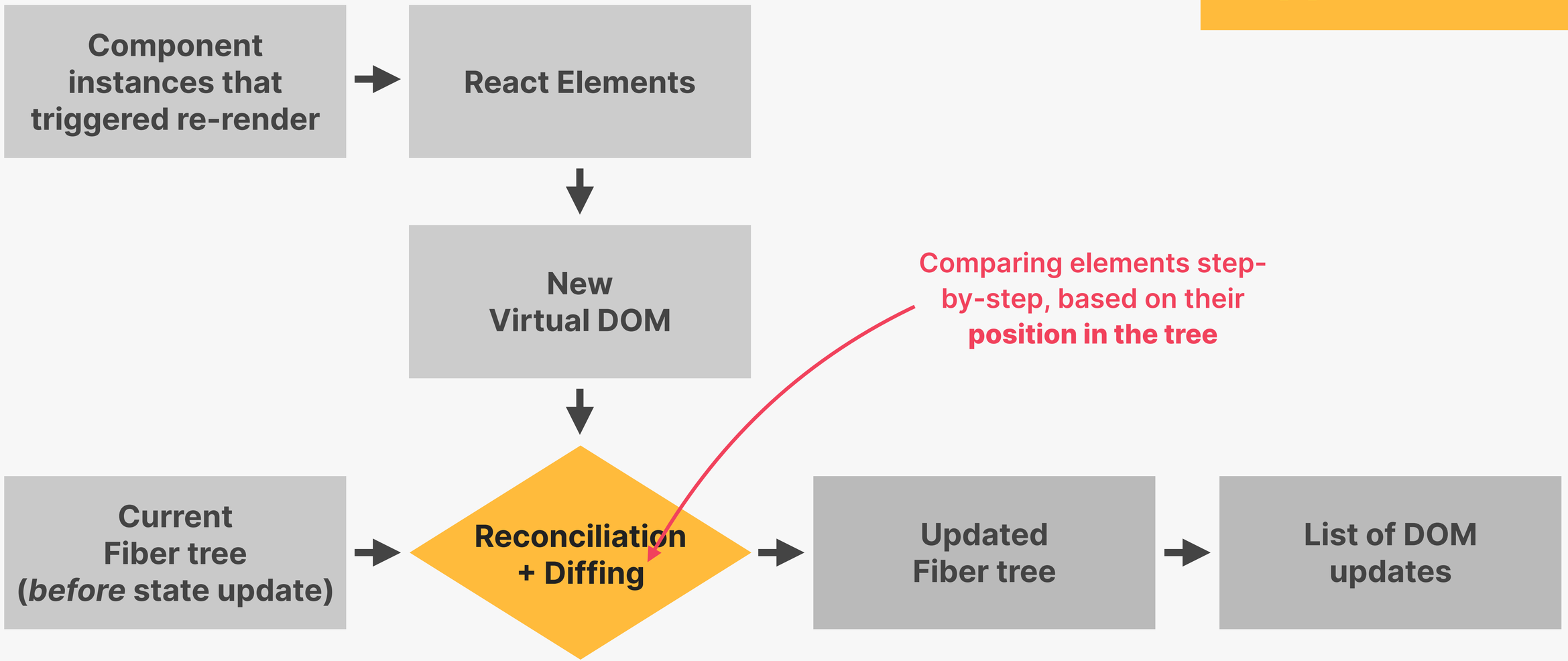
👉 Rendering a component **also renders all of its child components**

👉 **Asynchronous**: work can be split, prioritized, paused, resumed

👉 **Synchronous**: DOM updates are written in one go, to keep UI consistent

THE **RENDER** PHASE

[2] RENDER PHASE



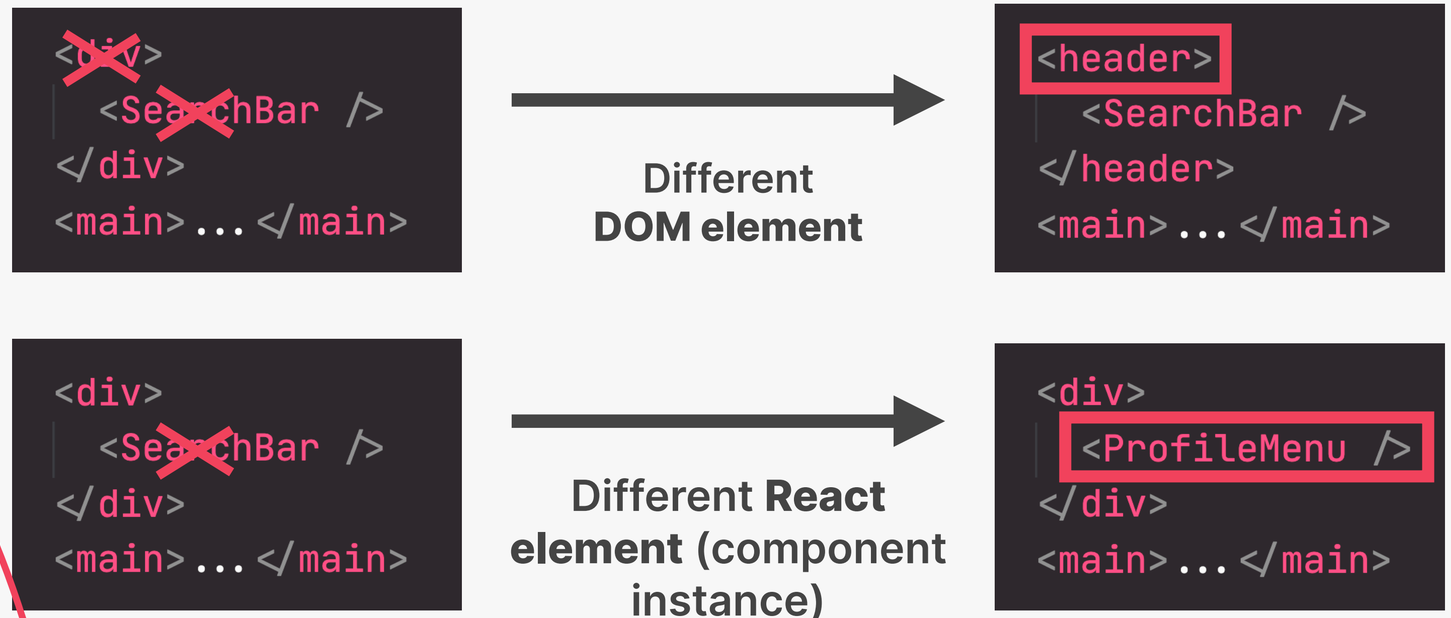
HOW **DIFFING** WORKS

👉 **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



👉 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

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

ADDING NEW LIST ITEM

```jsx
<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

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

ADDING NEW LIST ITEM

```jsx
<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

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

NEW QUESTION IN SAME POSITION

```jsx
<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):

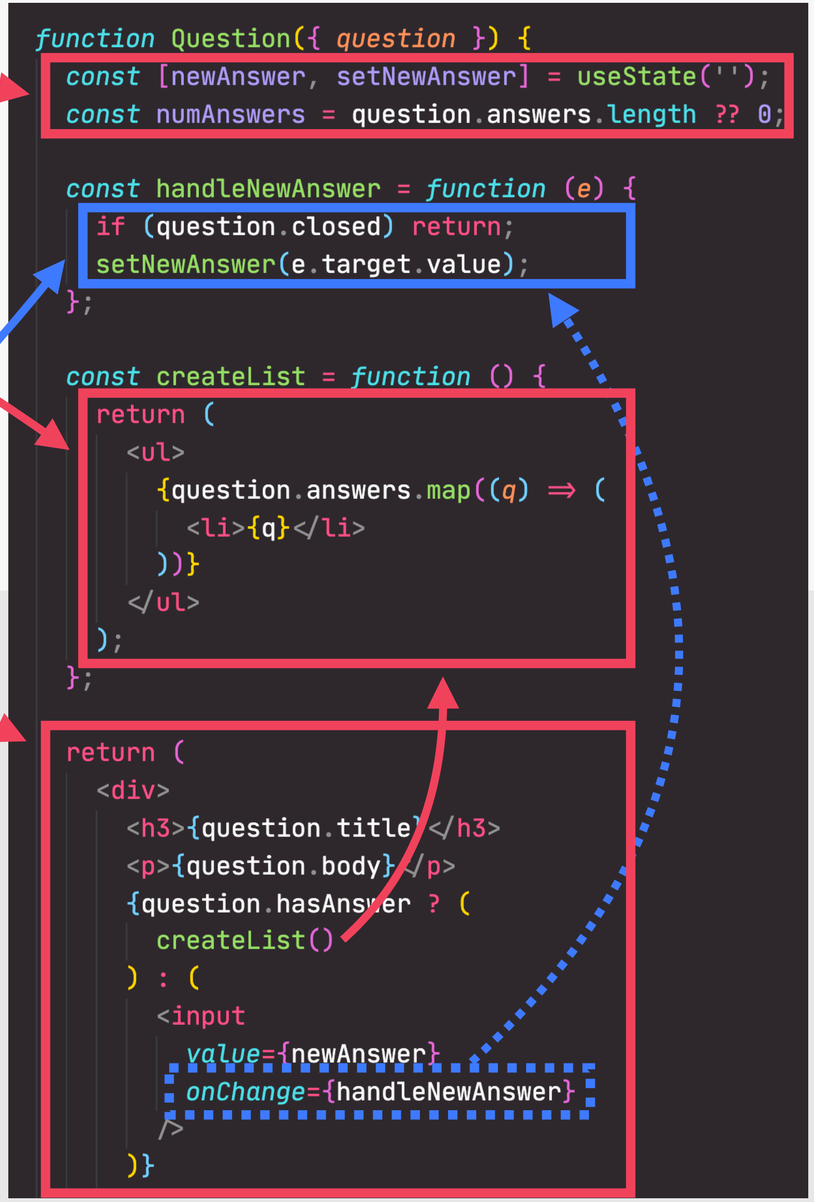
THE **TWO TYPES** OF LOGIC IN REACT COMPONENTS

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.



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

👉 **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**

✅ **Pure function**

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

☝️ **Impure function**

```
const areas = {};

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

Side effect: Outside variable mutation

☝️ **Impure function**

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

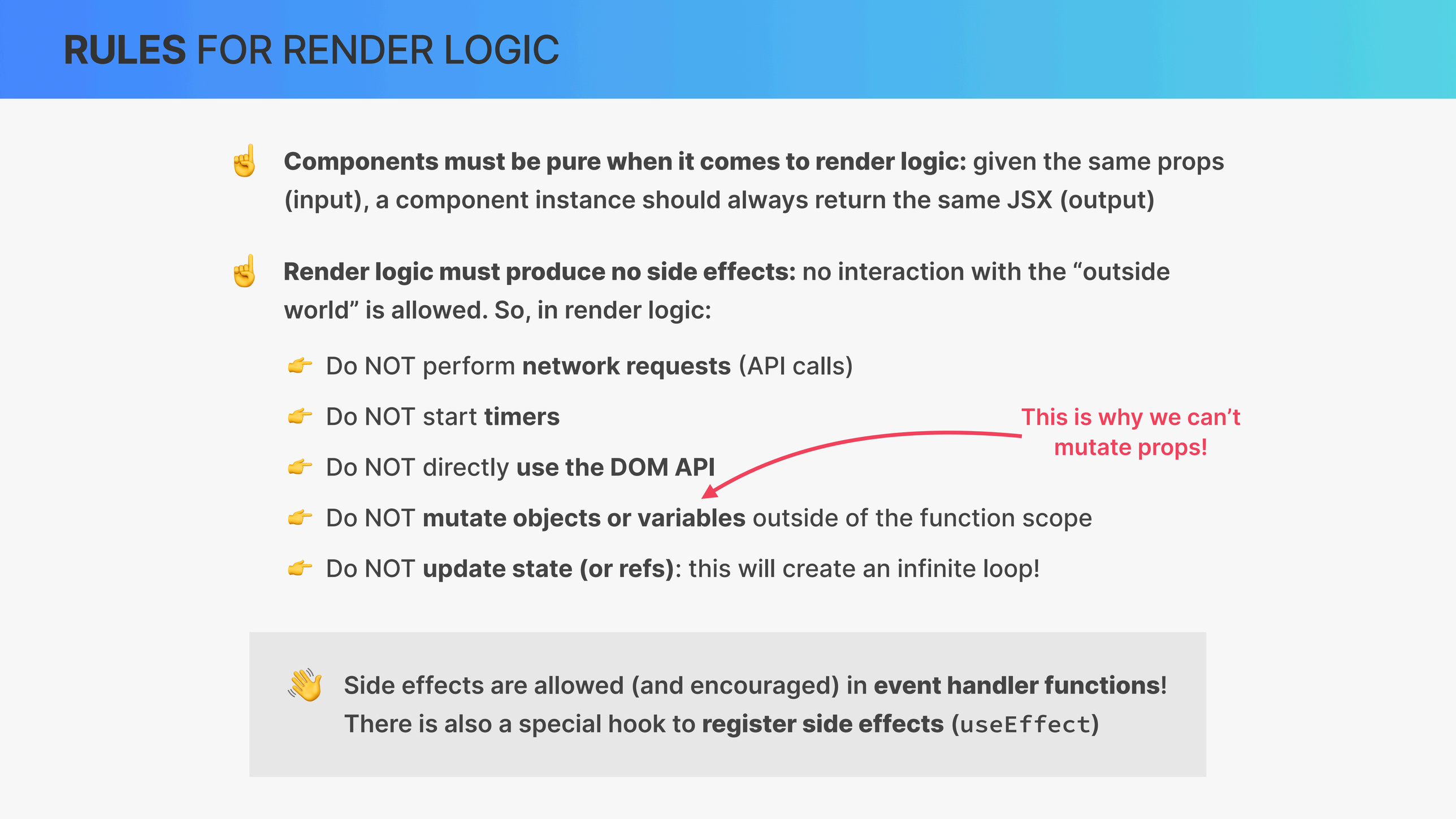
Unpredictable output (`date` changes)

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!



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

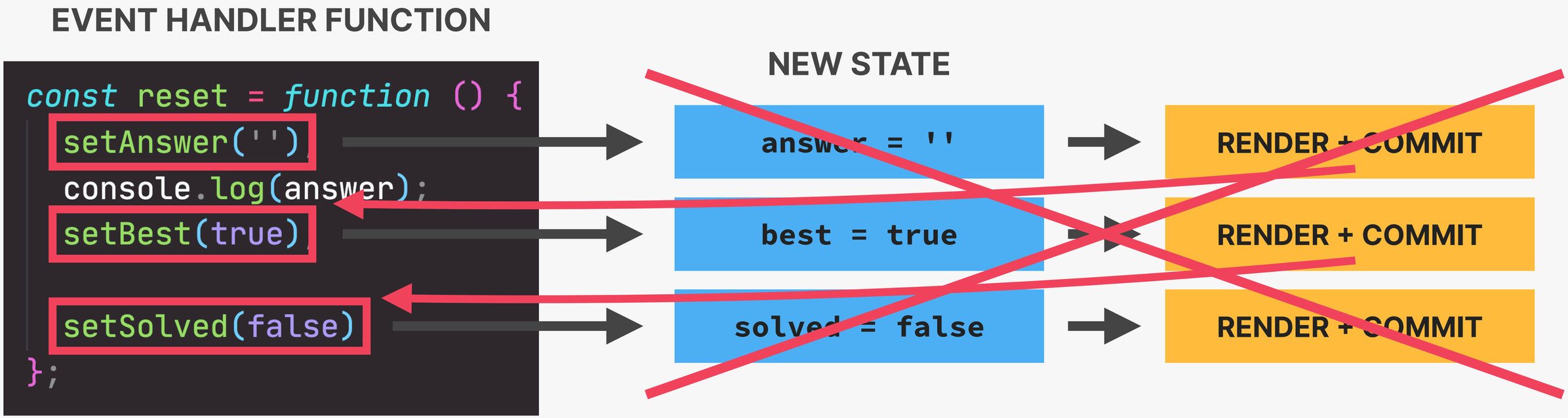
HOW STATE UPDATES ARE **BATCHED**

👉 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>
);
```

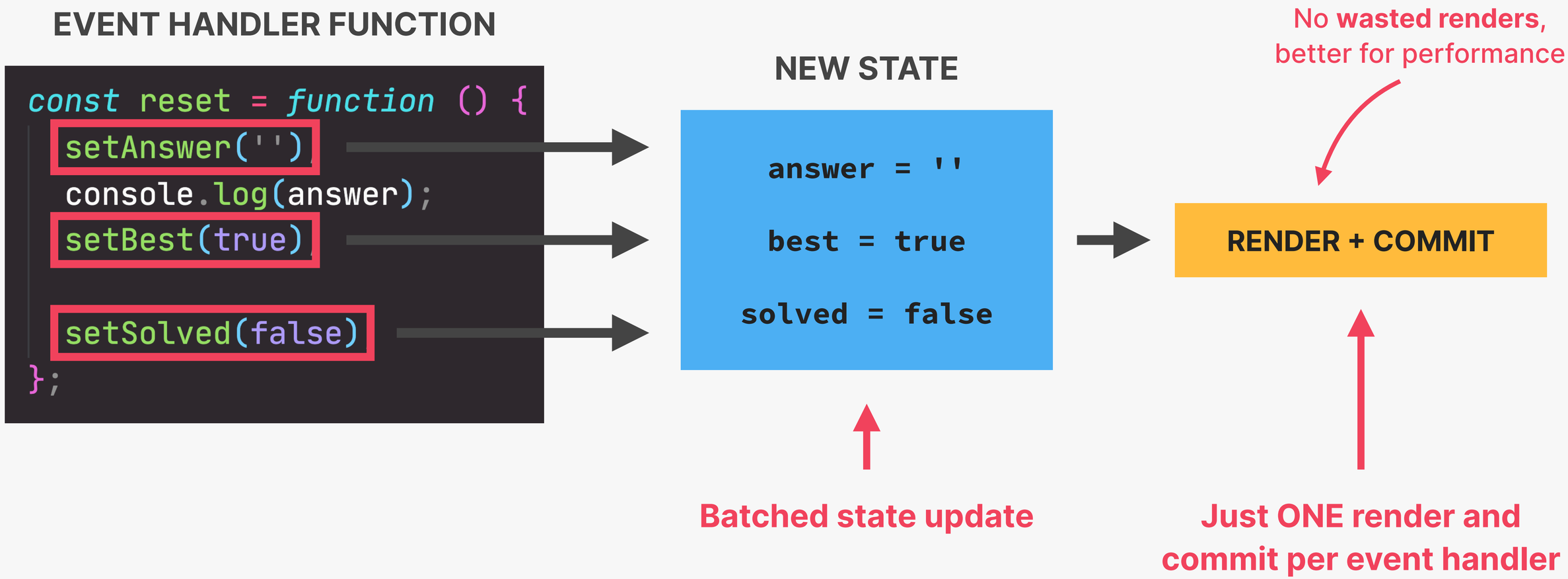
Event handler function

HOW STATE UPDATES ARE **BATCHED**



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 (`''`)

↓

UPDATING STATE IN REACT IS ASYNCHRONOUS

"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=>...)`)

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)*

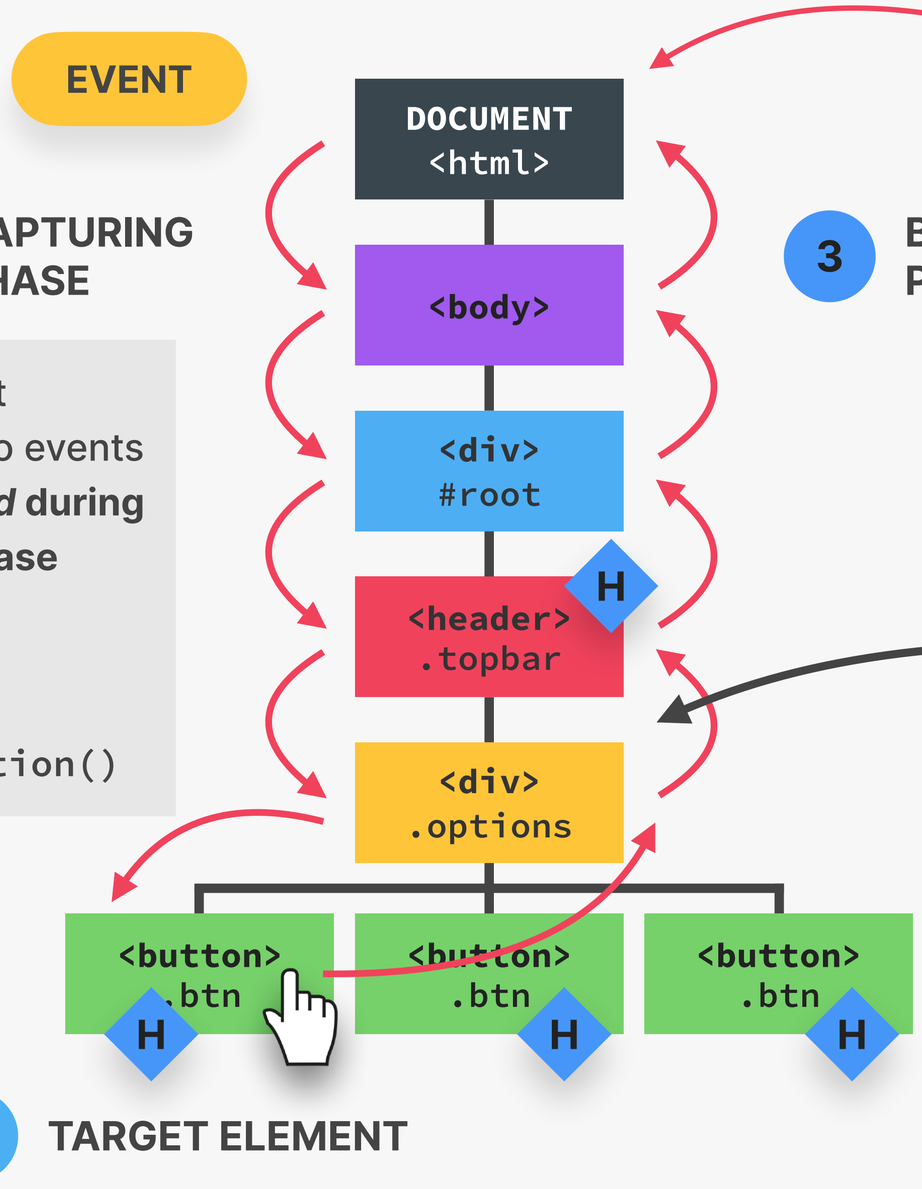
```
const reset = function () {
  setAnswer('');
  console.log(answer);
  setBest(true);

  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



1 **CAPTURING PHASE**

- 👉 By default, event handlers listen to events on the target ***and*** **during the bubbling phase**
- 👉 We can **prevent bubbling** with `e.stopPropagation()`

2 **TARGET ELEMENT**

3 **BUBBLING PHASE**

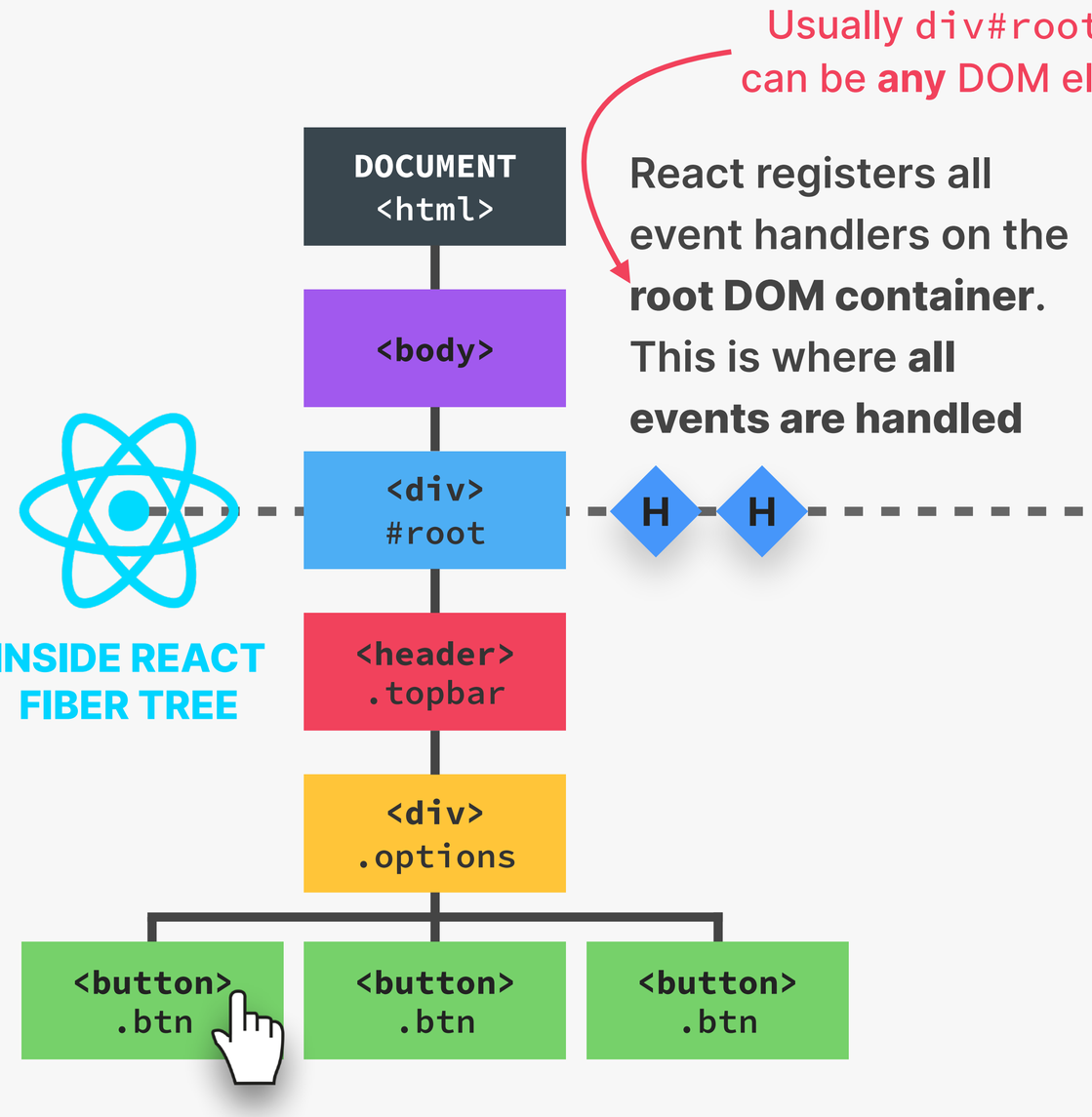
EVENT DELEGATION

- 👉 Handling events for multiple elements centrally in **one single parent element**
- 👉 Better for performance and memory, as it needs only **one handler function**

1. Add handler to **parent** (`.options`)
2. Check for **target** element (`e.target`)
3. If **target** is one of the `<button>`s, handle the event

👋 *Very common in vanilla JS apps, but not so much in React apps*

HOW **REACT** HANDLES EVENTS



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

WHEN WE ATTACH AN EVENT HANDLER...

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

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

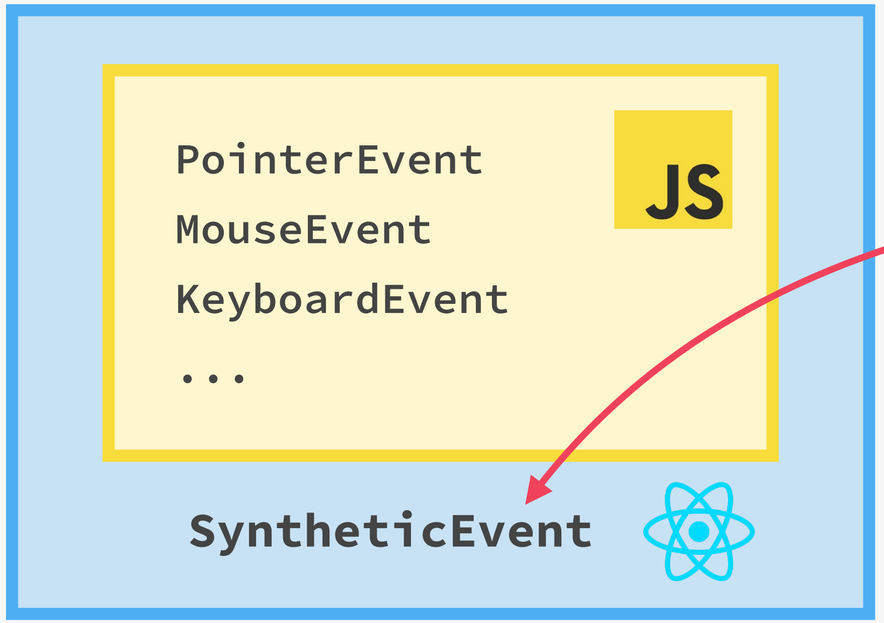
⛔ **... WHAT APPEARS TO BE HAPPENING**

```
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 HANDLERS IN VS. JS

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

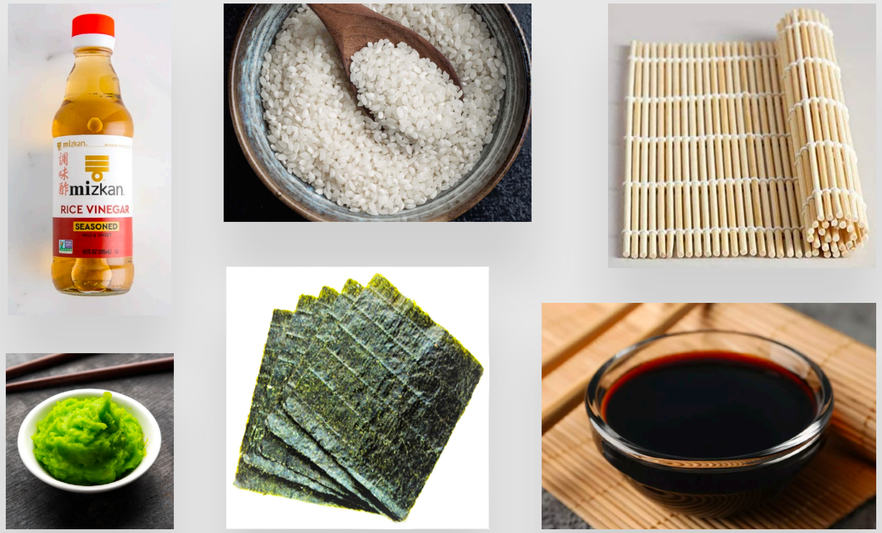
FIRST, AN **ANALOGY** 🍣

ALL-IN-ONE KIT

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

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

SEPARATE INGREDIENTS



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

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

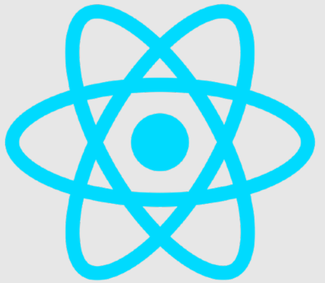
FIRST, AN **ANALOGY** 🍣

ALL-IN-ONE KIT

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

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

SEPARATE INGREDIENTS



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

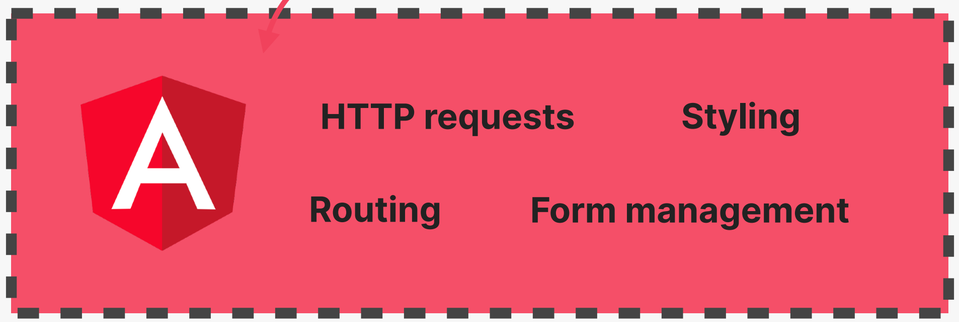
👎 **Decision fatigue:** You need to research and buy all ingredients separately

FRAMEWORK **VS.** LIBRARY

FRAMEWORK

"All-in-one kit"

Frameworks include everything



LARGE-SCALE ANGULAR APP

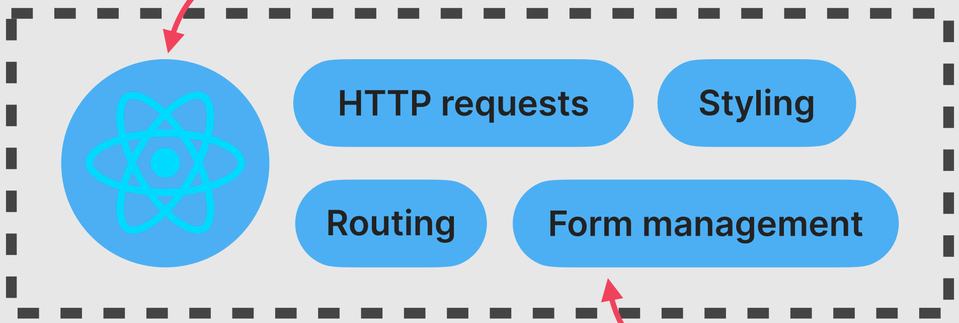
👍 **Ease of mind:** Everything you need to build a complete application **is included** in the framework ("batteries included")

👎 **No choice:** You're stuck with the framework's tools and conventions (which is not always bad!)

LIBRARY

"Separate ingredients"

"View" library



LARGE-SCALE REACT APP

External libraries

👍 **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
2	HTTP requests	fetch(), AXIOS
3	Remote state management	React Query, SWR, APOLLO
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	MUI, chakra, Mantine

👉 Library options for different React application needs

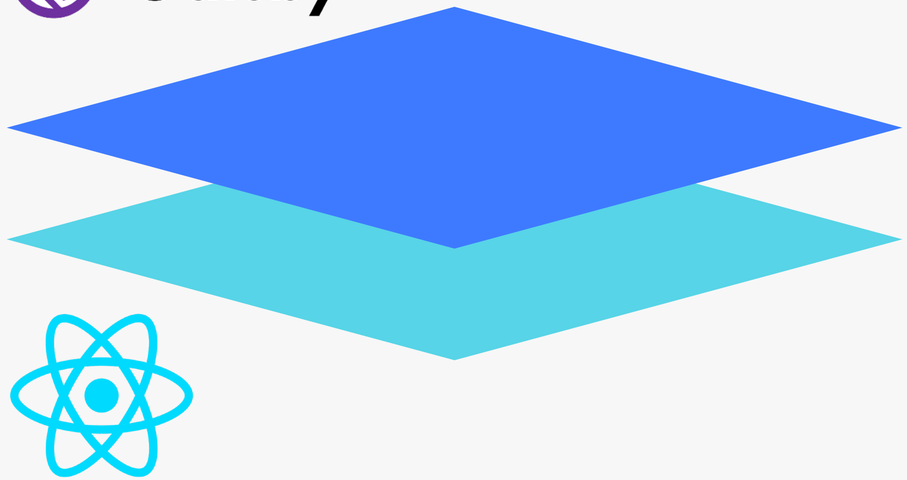
FRAMEWORKS BUILT **ON TOP** OF REACT

"Opinionated"
React frameworks

NEXT.JS

Remix

Gatsby



Full-stack frameworks!

"VANILLA" REACT APP

HTTP requests | Styling | Routing | Form management

External libraries

NEXT.JS Remix

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**

☎️ “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**

MOUNT / INITIAL RENDER

(Optional)

RE-RENDER

UNMOUNT

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

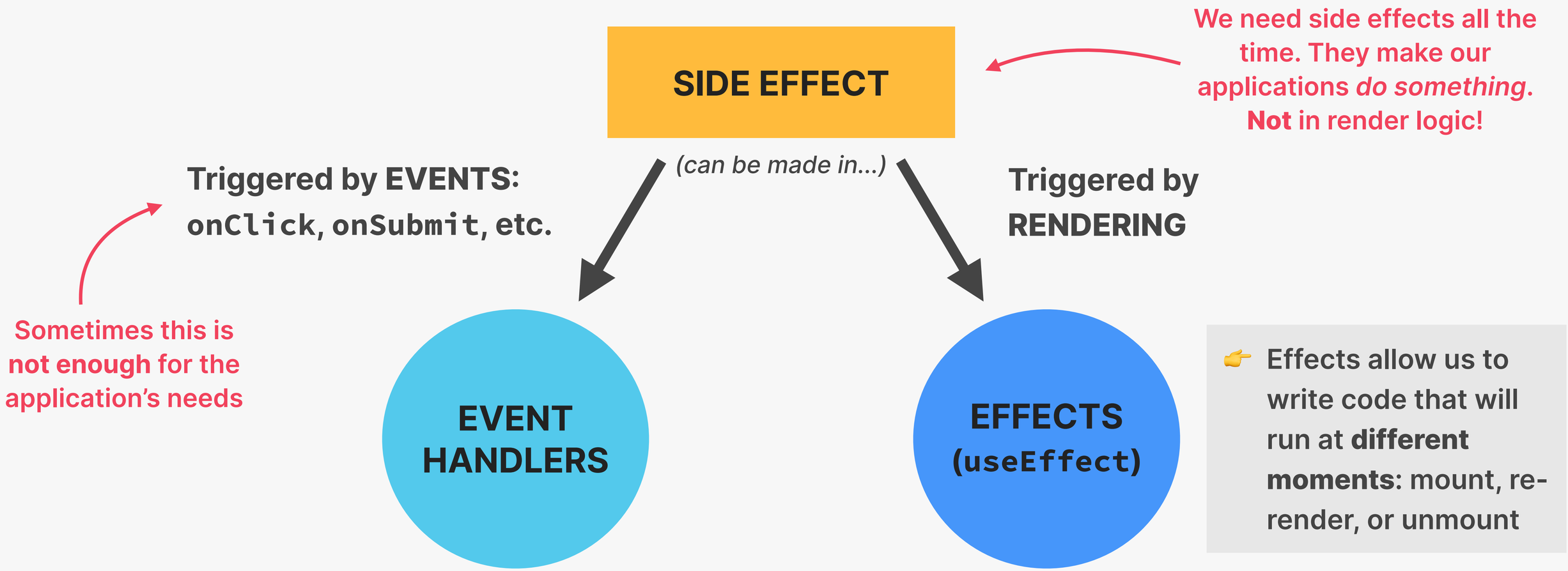
HAPPENS WHEN:

- 👉 **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

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

EFFECTS (`useEffect`)

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

Effect

Cleanup function

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

  return () => console.log('Cleanup');
}, []);
```

Dependency array

🤔 When?

👉 Executed when the **corresponding event happens**

👉 Used to **react** to an event

☝️ **Preferred way of creating side effects!**

Thinking about synchronization, **not** lifecycles

👉 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**

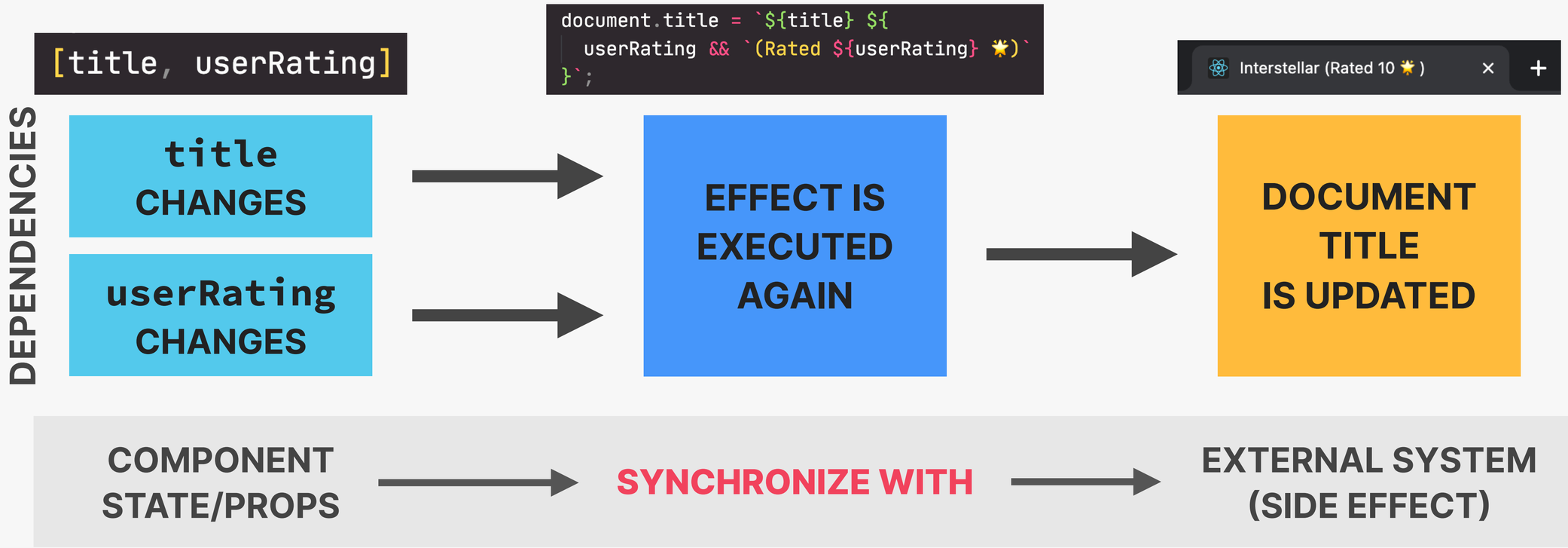
```
const title = props.movie.Title;
const [userRating, setUserRating] = useState('');

useEffect(
  function () {
    if (!title) return;
    document.title = `${title} ${
      userRating && `(Rated ${userRating} 🌟)`
    }`;

    return () => (document.title = 'usePopcorn');
  },
  [title, userRating]
);
```

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

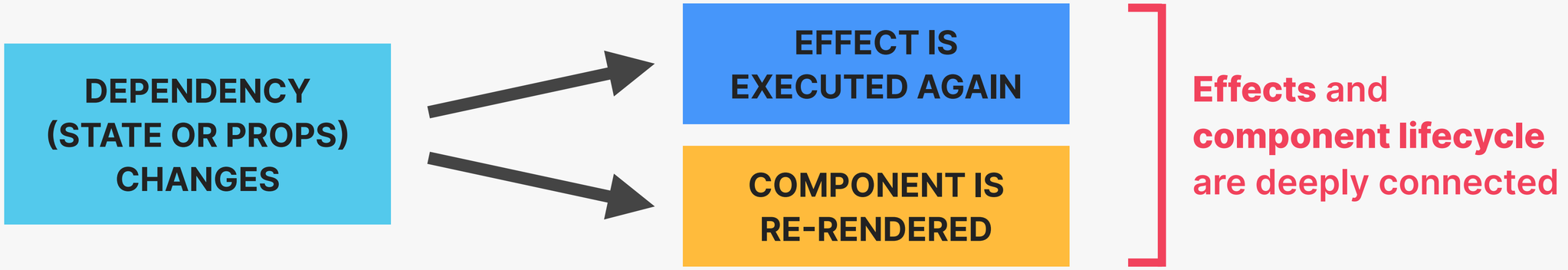
USEEFFECT IS A **SYNCHRONIZATION** MECHANISM

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



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

	🔄 SYNCHRONIZATION	🐣 LIFECYCLE
`useEffect(fn, [x, y, z]);`	Effect synchronizes with **x**, **y**, and **z**	Runs on **mount** and **re-renders** triggered by updating x, y, or z
`useEffect(fn, []);`	Effect synchronizes with **no state/props**	Runs only on **mount** (initial render)
`useEffect(fn);`	Effect synchronizes with **everything**	Runs on **every render** (usually bad ⛔)

WHEN ARE EFFECTS **EXECUTED?**

`title = 'Interstellar'`

MOUNT (INITIAL RENDER)

COMMIT

BROWSER PAINT

EFFECT ✨

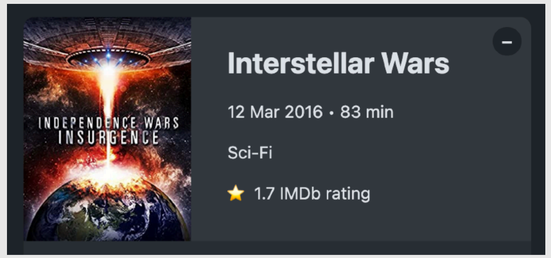
```
<MovieDetails />
```

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

```
document.title = `${title} ${
  userRating && `(Rated ${userRating} 🌟)`
}`;
```

Interstellar

`title = 'Interstellar Wars'`



`title` CHANGES

RE-RENDER

COMMIT

LAYOUT EFFECT

BROWSER PAINT

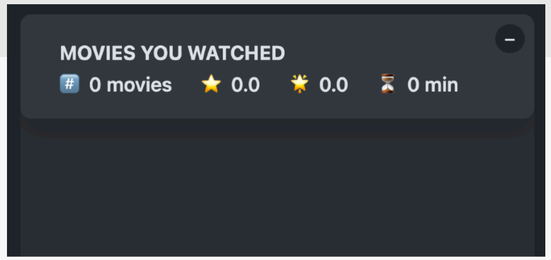
EFFECT ✨

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

```
[title, userRating]
```

```
document.title = `${title} ${
  userRating && `(Rated ${userRating} 🌟)`
}`;
```

Interstellar Wars



UNMOUNT

time

WHEN ARE EFFECTS **EXECUTED?**

`title = 'Interstellar'`

MOUNT (INITIAL RENDER)

```
<MovieDetails />
```

COMMIT

BROWSER PAINT

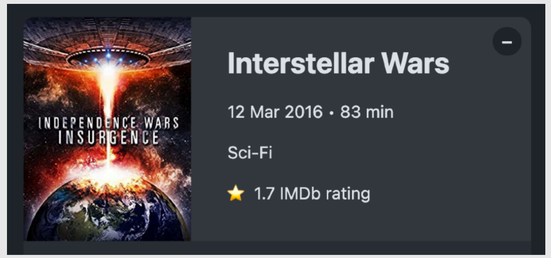
EFFECT ✨

```
document.title = `${title} ${
  userRating && `(Rated ${userRating} 🌟)`
}`;
```

Interstellar

`title CHANGES`

`title = 'Interstellar Wars'`



RE-RENDER

COMMIT

LAYOUT EFFECT

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

usePopcorn

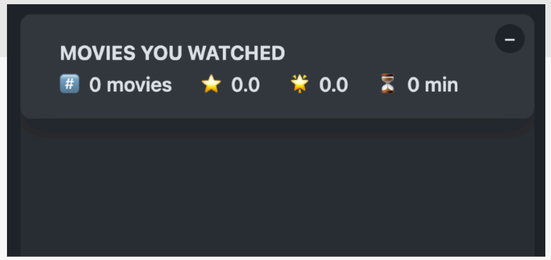
BROWSER PAINT

CLEANUP 🧹

EFFECT ✨

```
document.title = `${title} ${
  userRating && `(Rated ${userRating} 🌟)`
}`;
```

Interstellar Wars



UNMOUNT

CLEANUP 🧹

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

usePopcorn

time

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

COMPONENT RENDERS → **Execute effect if dependency array includes updated data**

COMPONENT UNMOUNTS → **Execute cleanup function**

Examples

✨ EFFECT	🧹 POTENTIAL CLEANUP
👉 HTTP request	→ Cancel request
👉 API subscription	→ Cancel subscription
👉 Start timer	→ Stop timer
👉 Add event listener	→ Remove listener

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`

👋 *As of React v18.x*

✅ Have learned
👉 Will learn
❌ Will **not** learn

THE **RULES** OF HOOKS

☝️ **RULES OF HOOKS**

1 **Only call hooks at the top level**

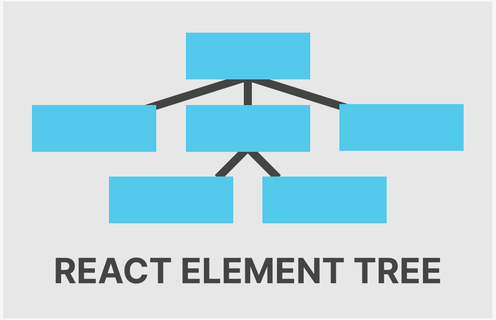
- 👉 Do **NOT** call hooks inside **conditionals**, **loops**, **nested functions**, or after an **early return**
- 👉 This is necessary to ensure that hooks are always called in the **same order** (hooks rely on this)

2 **Only call hooks from React functions**

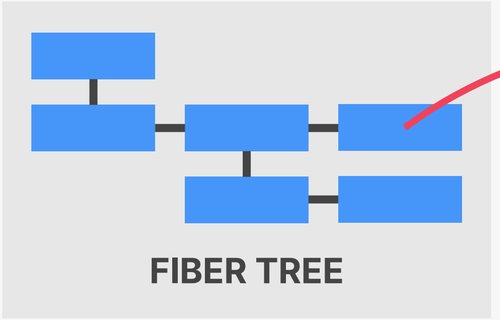
- 👉 Only call hooks inside a **function component** or a **custom hook**

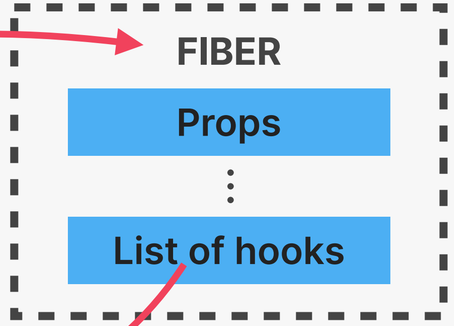
👋 *These rules are **automatically enforced** by React's ESLint rules*

HOOKS RELY ON **CALL ORDER**



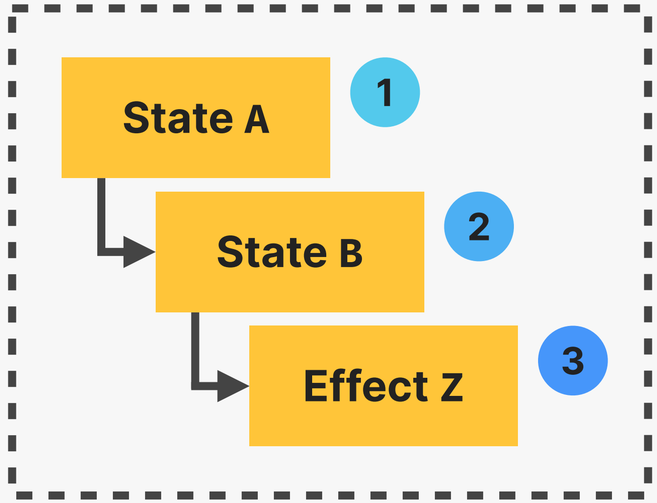
ON INITIAL RENDER





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

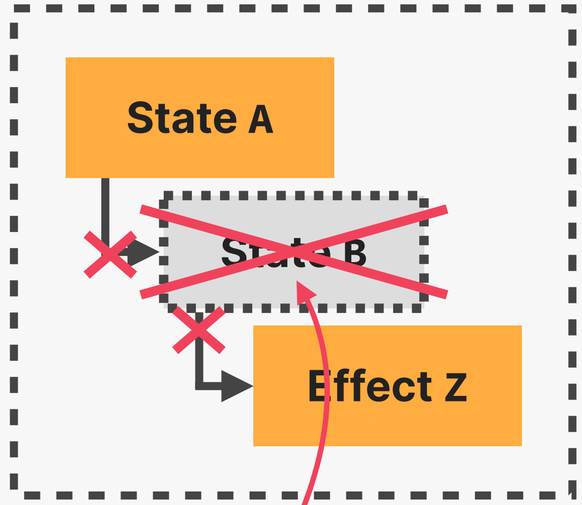
```
const [A, setA] = useState(23)   1
                          7
if (A === 23) false
  const [B, setB] = useState('')  2

useEffect(fnZ, [])                3
```



RENDER

~~A=23~~

A=7



Violates Rule #1

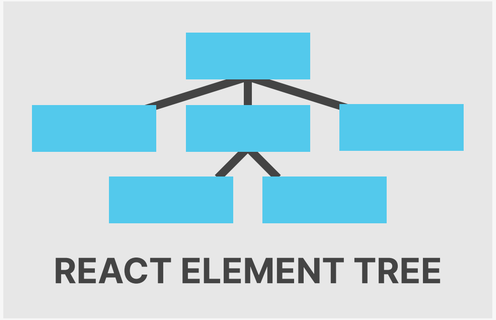
List built based on hooks **call order**

LINKED LIST OF USED HOOKS

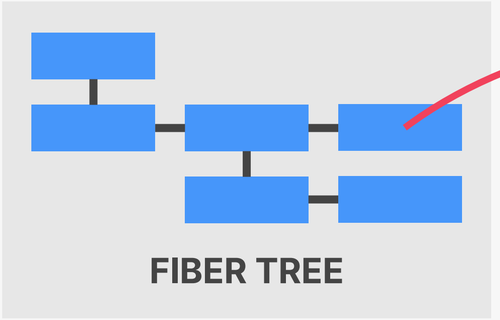
`A===23` is now `false`, so after re-render, this hook would no longer exist, **destroying the linked list** 😢

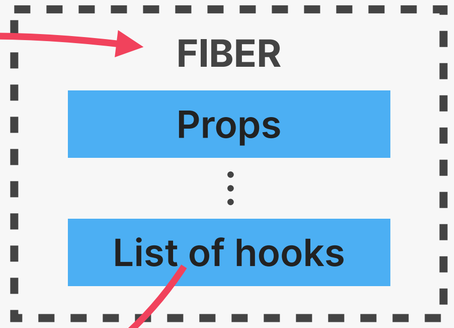
👉 **Hooks need to called in the same order on every render**

HOOKS RELY ON **CALL ORDER**



ON INITIAL RENDER



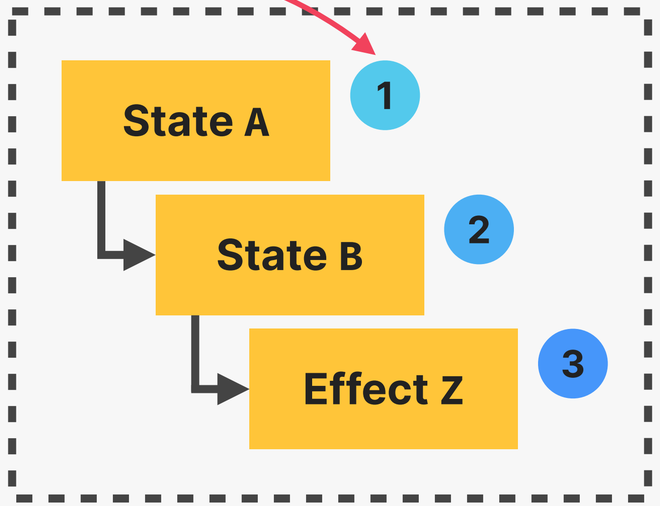


Order num uniquely identifies each hook

👍 *Correct code!*

```
const [A, setA] = useState(23)      1

  const [B, setB] = useState('')    2

useEffect(fnZ, [])                  3
```

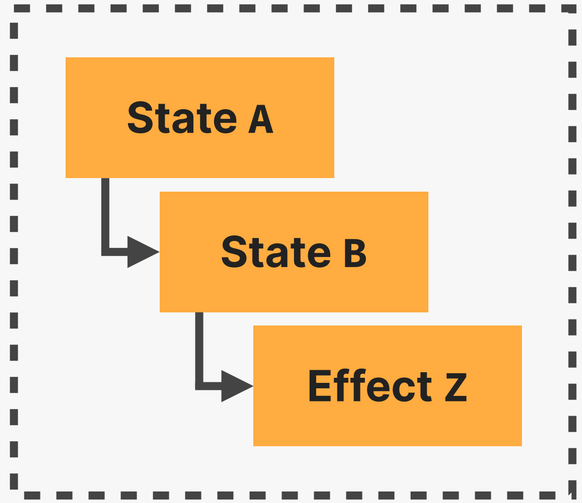
List built based on hooks **call order**



LINKED LIST OF USED HOOKS

RENDER

SAME ORDER



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

☝️ Hooks can only be called at top level

SUMMARY OF DEFINING AND UPDATING STATE

1 **CREATING STATE**

- **Simple**

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

- **Based on function (lazy evaluation)**

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

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

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

2 **UPDATING STATE**

- **Simple**

```
setCount(1000);
```

- **Based on current state**

```
setCount((c) ⇒ c + 1)
```

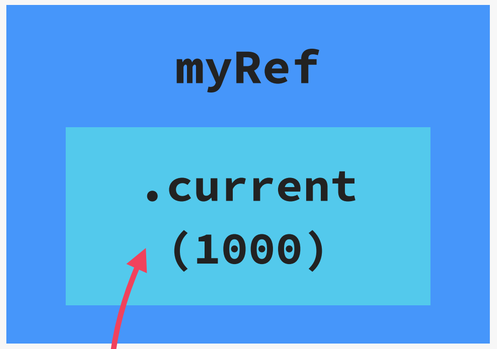
👉 Function must be **pure** and return next state

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);
```

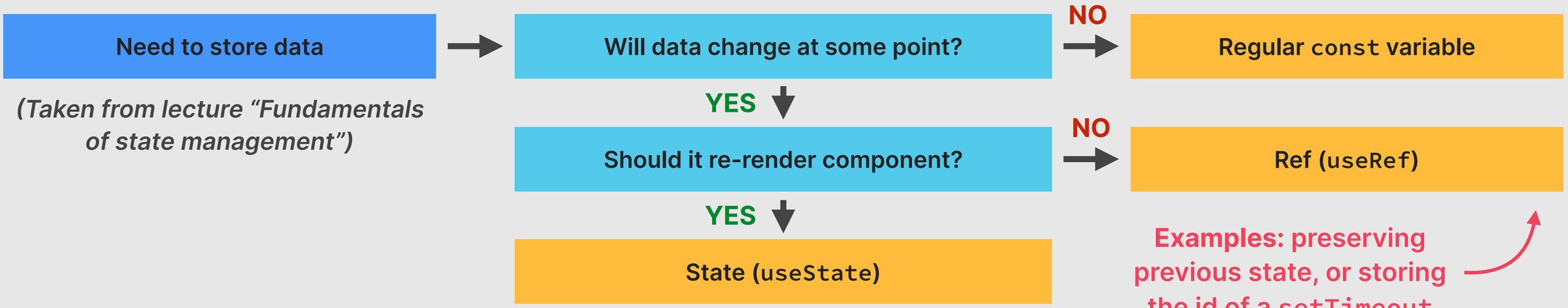


We can write to and read from the ref using `.current`

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

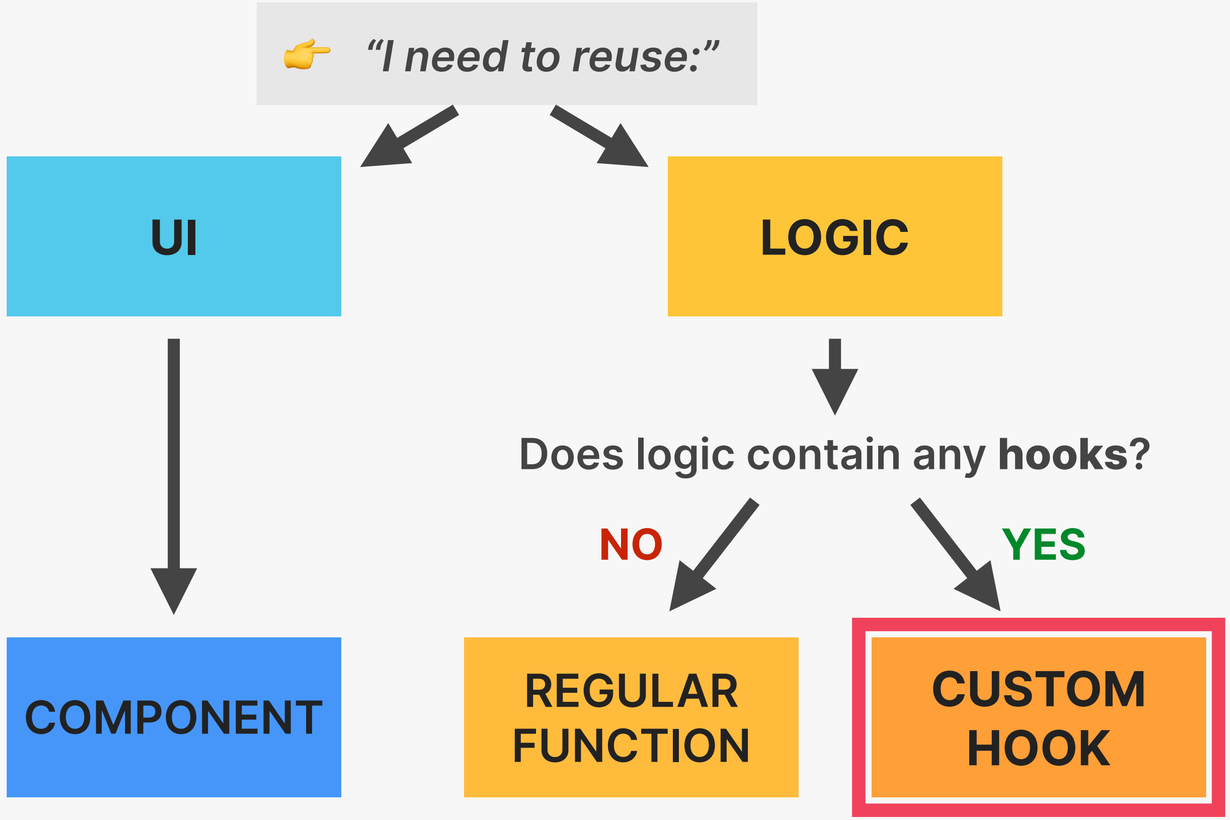
STATE **VS.** REFS

	PERSISTS ACROSS RENDERS	UPDATING CAUSES RE-RENDER	IMMUTABLE	ASYNCHRONOUS UPDATES
STATE	✅	✅	✅	✅
REFS	✅	❌	❌	❌



(Taken from lecture "Fundamentals of state management")

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 name needs to start with **use**

```
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];
}
```

Needs to use **one or more hooks**

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
📅 **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

Hooks are THE big difference

🏆 **Advantages**

FUNCTION COMPONENTS

- 👍 Easier to build (less boilerplate code)
- 👍 Cleaner code: `useEffect` combines all lifecycle-related code in a single place
- 👍 Easier to share stateful logic
- 👍 We don't need `this` keyword anymore

CLASS COMPONENTS

- 👍 Lifecycle might be easier to understand for beginners

PART 03

—

ADVANCED REACT + REDUX

THE ADVANCED USEREDUCER 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 USEREDUCER

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
- `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`

Like `setState()` with superpowers

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*

Just like `array.reduce()`, reducers accumulate ("reduce") **actions over time**

CURRENT STATE

dispatch → reducer → RETURNS → NEXT STATE → RE-RENDER

action

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

Object that contains information on how the reducer should update state

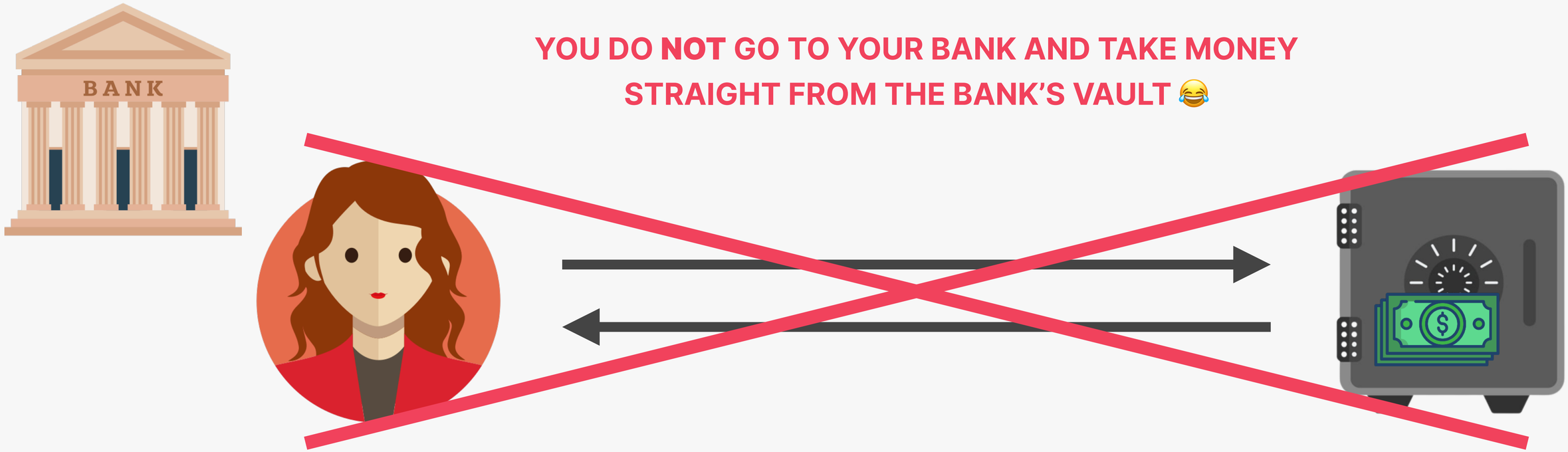
useState

setState → UPDATE → NEXT (UPDATED) STATE → RE-RENDER

UPDATED STATE

A **MENTAL MODEL** FOR REDUCERS

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



A **MENTAL MODEL** FOR REDUCERS

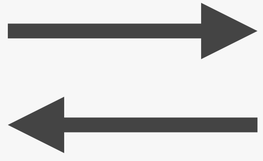
👉 **REAL-WORLD TASK:** WITHDRAWING $5,000 FROM YOUR BANK ACCOUNT

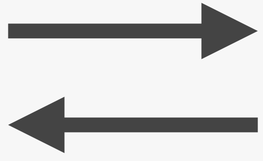
(**How** to make the update)

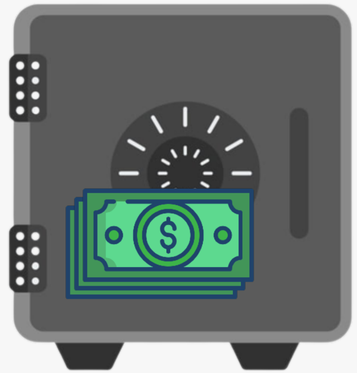
ACTION

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

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







DISPATCHER

(Who ***requests*** the update)

REDUCER

(Who ***makes*** the update)

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

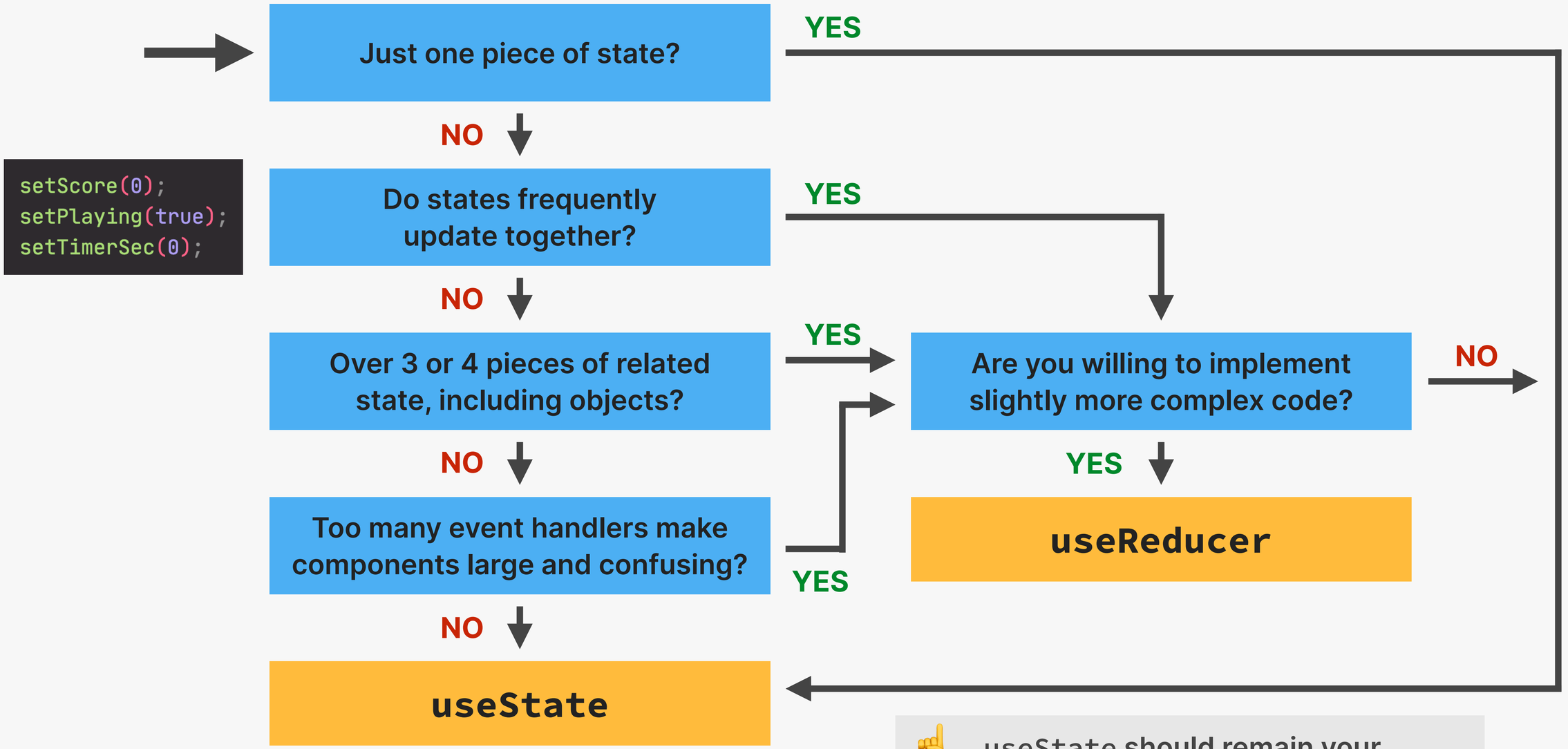
useReducer

- 👉 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?



☝️ `useState` should remain your **default choice** for managing state

REACT ROUTER: BUILDING SINGLE-PAGE APPLICATIONS (SPA)

WHAT IS **ROUTING**?

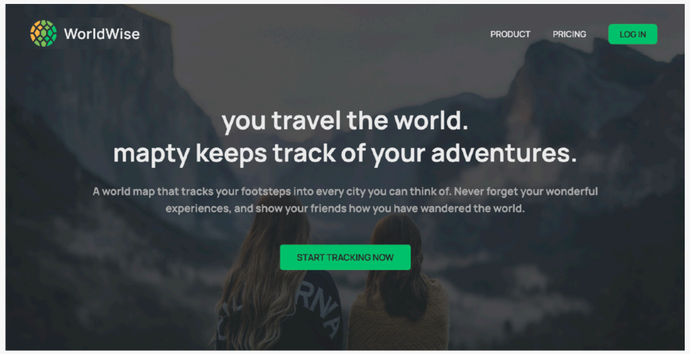
"Client-side routing"

ROUTING

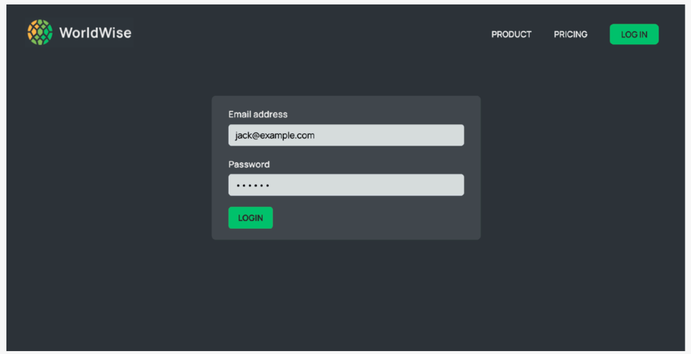
- 👉 With routing, we match **different URLs** to **different UI views** (React components): **routes**
- 👉 This enables users to **navigate between different applications screens**, using the browser URL
- 👉 Keeps the UI **in sync** with the current browser URL
- 👉 Allows us to build **Single-Page Applications**

React Router

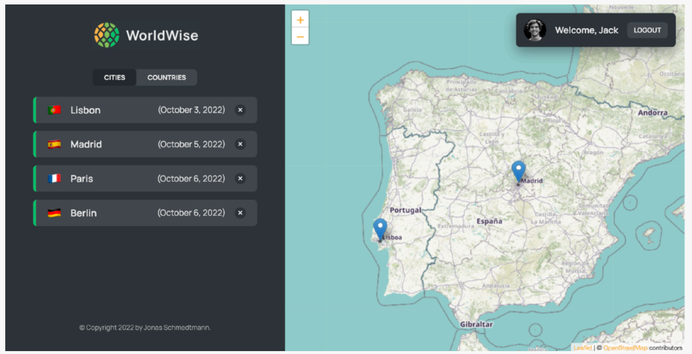
`www.example.com/`



`www.example.com/login`



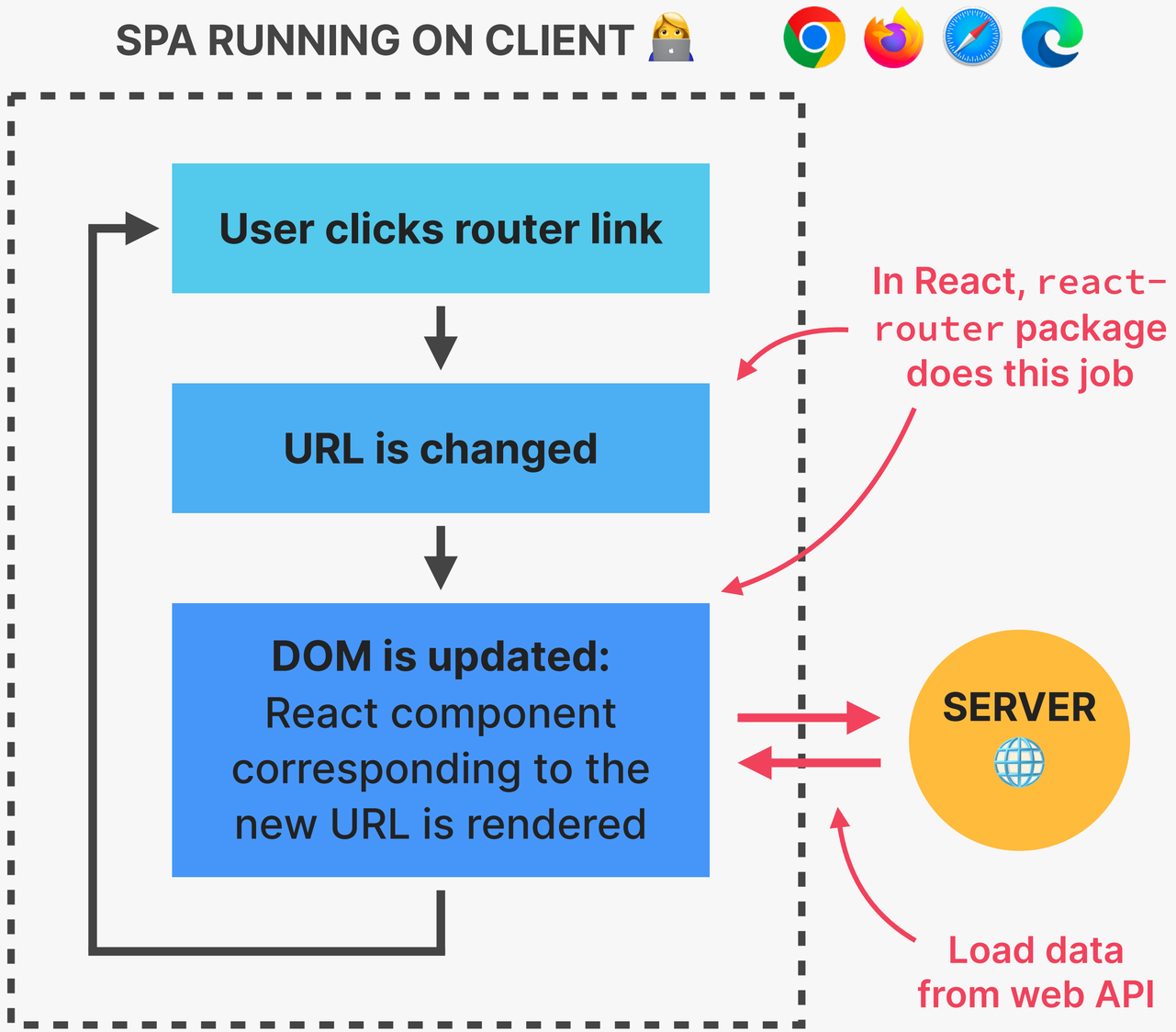
`www.example.com/app`



SINGLE-PAGE APPLICATIONS (SPA)

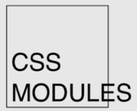
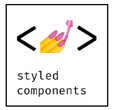
SINGLE-PAGE APPLICATION

- 👉 Application that is **executed entirely on the client** (browsers)
- 👉 **Routes**: different URLs correspond to different views (components)
- 👉 **JavaScript** (React) is used to update the page (DOM)
- 👉 ***The page is never reloaded***
- 👉 Feels like a **native app**
- 👉 Additional data **might be loaded** from a web API



STYLING OPTIONS IN REACT

React doesn't care about styling

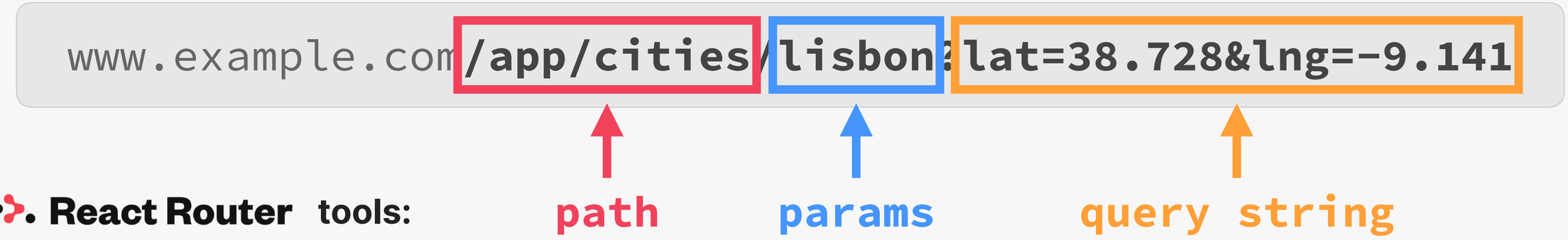
STYLING OPTION	WHERE?	HOW?	SCOPE	BASED ON
👉 **Inline CSS** 	JSX elements	`style` prop	Local **JSX element**	CSS
👉 **CSS or Sass file**  	External file	`className` prop	Global, causes problems **Entire app**	CSS
👉 **CSS Modules** 	One external file per component	`className` prop	**Component**	CSS
👉 **CSS-in-JS** 	External file or component file	Creates new component	**Component**	JavaScript
👉 **Utility-first CSS**  tailwindcss	JSX elements	`className` prop	**JSX element**	CSS

👋 **Alternative to styling with CSS:** UI libraries like MUI, Chakra UI, Mantine, etc. chakra  Mantine

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

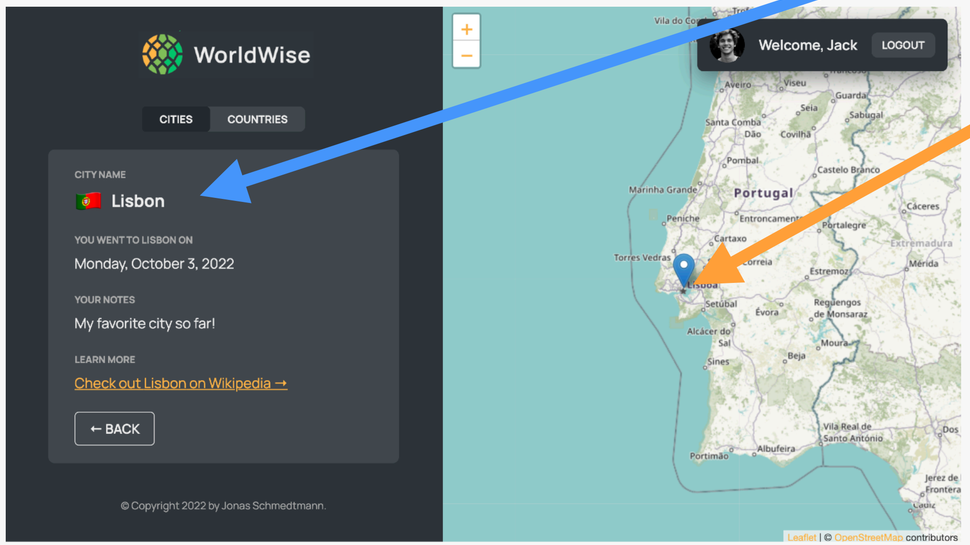
1. Easy way to store state in a **global place**, accessible to **all components** in the app
2. Good way to **“pass” data** from one page into the next page
3. Makes it possible to **bookmark and share** the page with the exact UI state it had at the time

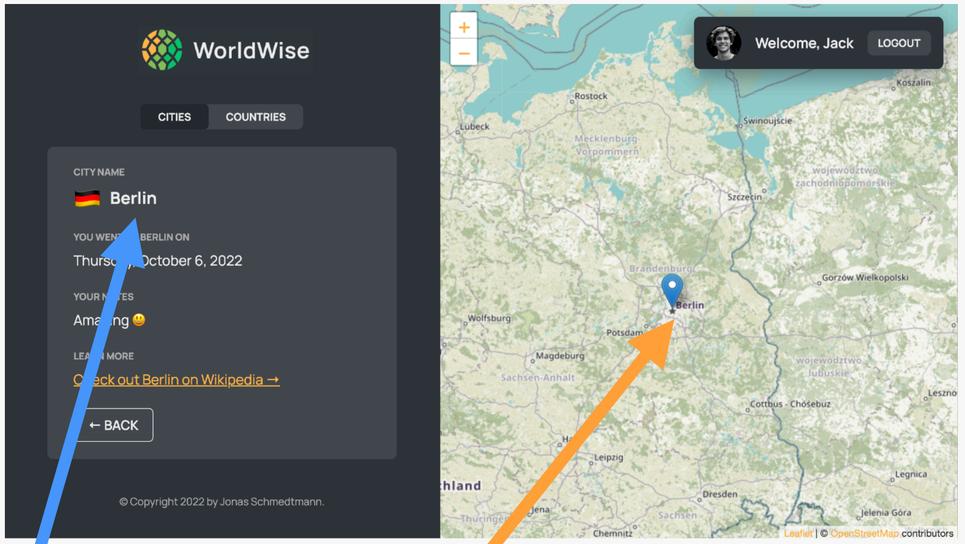


React Router tools:

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`

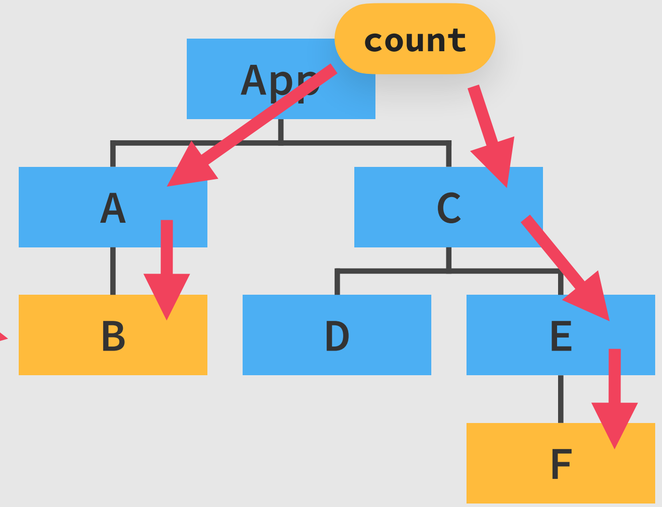
ADVANCED STATE MANAGEMENT: THE CONTEXT API

A SOLUTION TO **PROP DRILLING**

👉 **TASK: Passing state into multiple deeply nested child components**

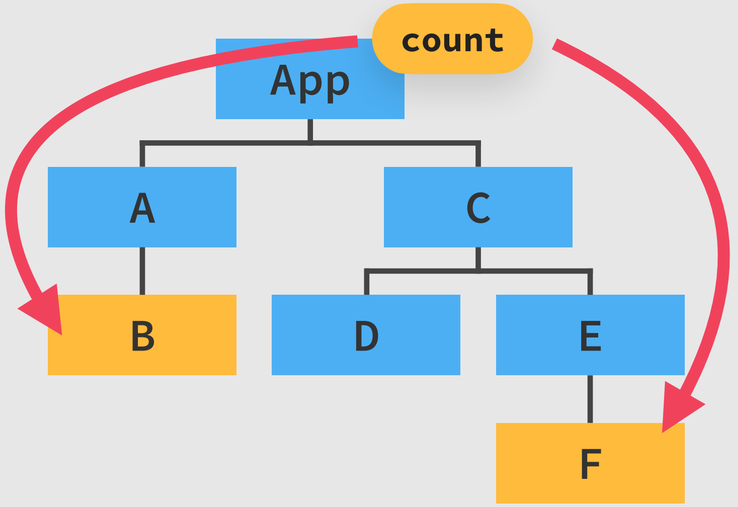
Components that need `count` state

💡 SOLUTION 1: PASSING PROPS



⛔ **PROBLEM: “PROP DRILLING”**

✅ SOLUTION 2: CONTEXT API



👍 **READ STATE FROM EVERYWHERE**

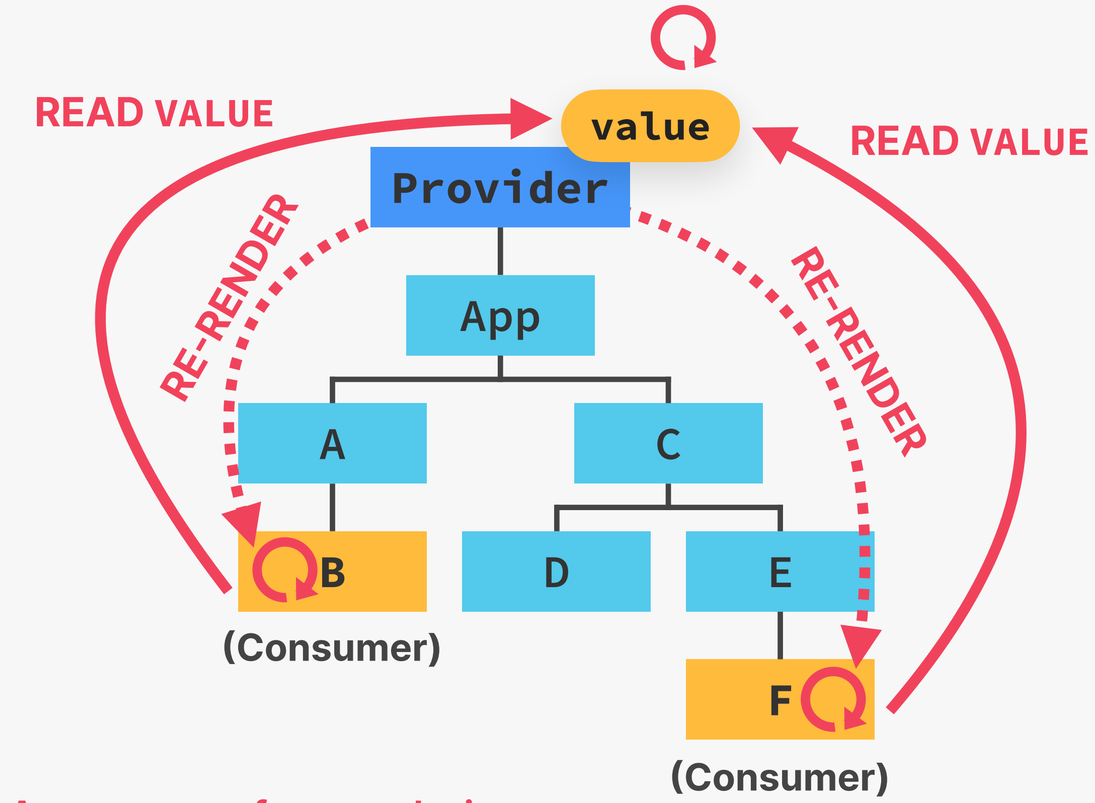
👋 Remember that a good solution to “prop drilling” is **better component composition** (see “Thinking in React” section)

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`



A new way of re-rendering components!

`value` is updated → All consumers re-render

REVIEW: WHAT IS STATE MANAGEMENT?

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

✅ **When** to use state

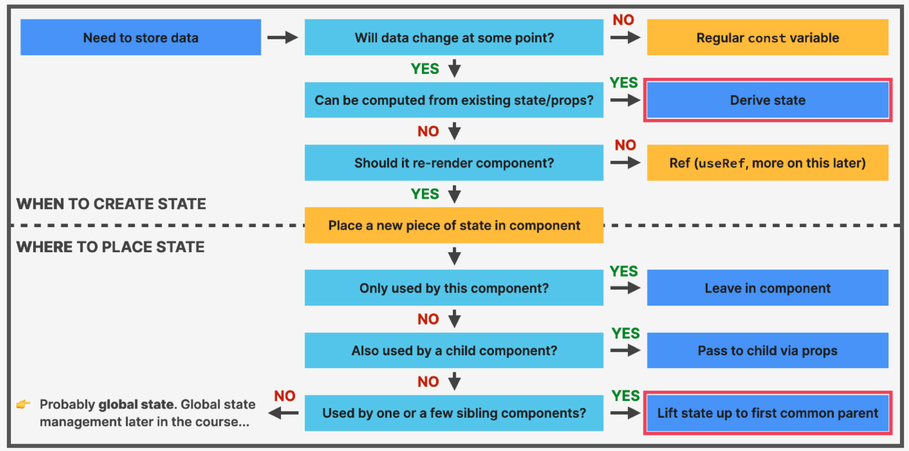
✅ **Types** of state (**accessibility**): local vs. global

👉 **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?"*

NO

🏠 LOCAL STATE

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

VS.

🌍 GLOBAL STATE

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

2 STATE DOMAIN

🌐 REMOTE STATE

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

VS.

🪄 UI STATE

- 👉 **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?***

1️⃣ STATE ACCESSIBILITY

2️⃣ STATE DOMAIN	🏠 LOCAL STATE	🌍 GLOBAL STATE
🪄 UI STATE	👉 `useState` 👉 `useReducer` 👉 `useRef`	👉 Context API + `useState/useReducer` 👉 Redux, Zustand, Recoil, etc. 👉 React Router
🌐 REMOTE STATE	👉 `fetch + useEffect + useState/useReducer`	👉 Context API + `useState/useReducer` 👉 Redux, Zustand, Recoil, etc. 👉 React Query 👉 SWR 👉 RTK Query

Mostly in small applications

Tools highly specialized in handling **remote** state

PERFORMANCE OPTIMIZATION AND ADVANCED USEEFFECT

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`

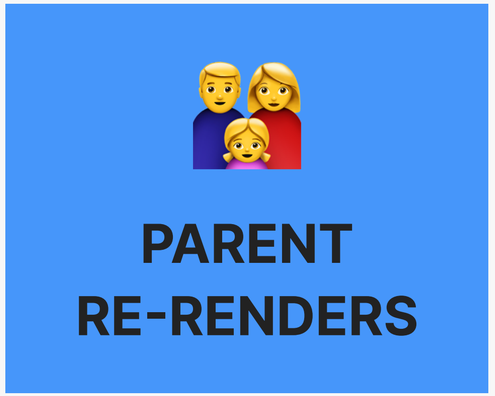
3 REDUCE BUNDLE SIZE

- 👉 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.

⬇

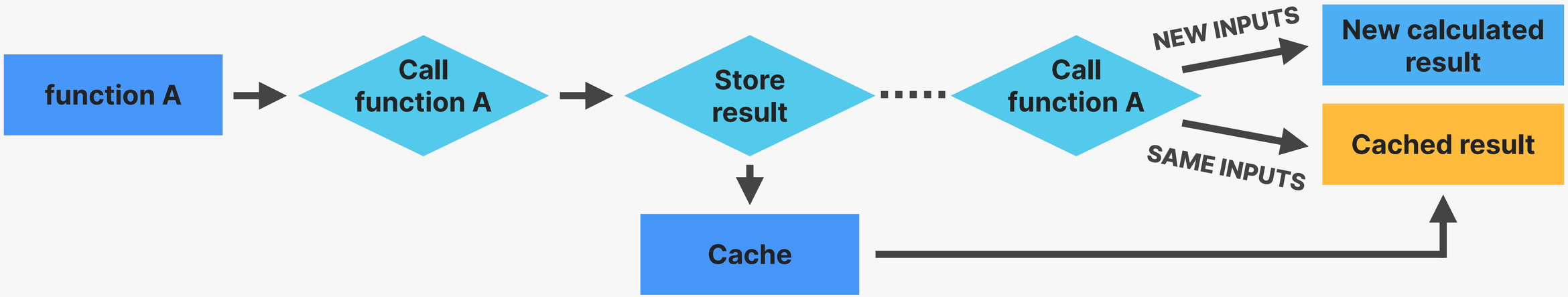
👉 **Wasted render:** a render that didn't produce any change in the DOM

Usually no problem, as React is very fast!

👉 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**

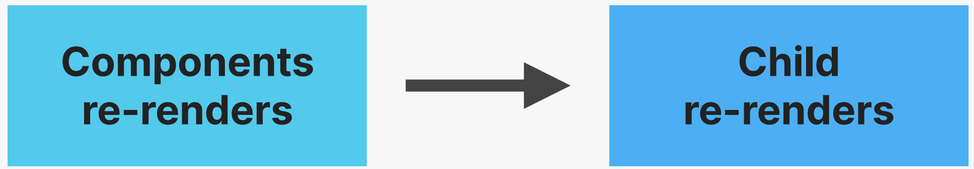
THE `MEMO` FUNCTION

`memo`

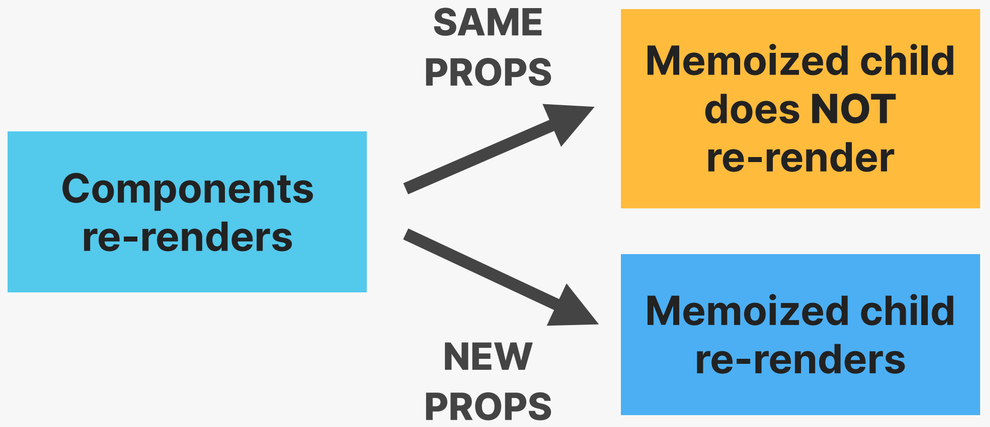
Memoized component

- 👉 Used to create a component that will **not re-render when its parent re-renders**, as long as the **props stay the same between renders**
- 👉 **Only affects props**! A memoized component will still re-render when its **own state changes** or when a **context that it's subscribed to changes**
- 👉 Only makes sense when the component is **heavy** (slow rendering), **re-renders often**, and does so **with the same props**

⛔ REGULAR BEHAVIOR (NO `MEMO`)



✅ MEMOIZED CHILD WITH `MEMO`



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 ↓

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 ↓

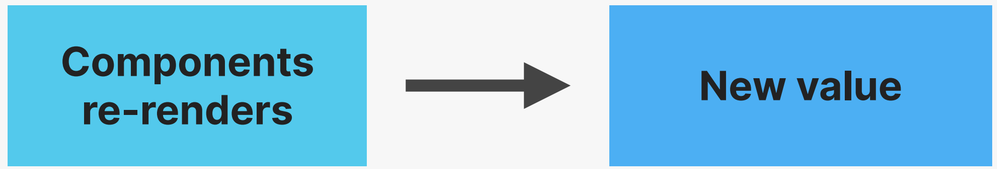
We need to memoize objects and functions, to make them stable (preserve) between re-renders (`memoized {} == memoized {}`)

TWO NEW HOOKS: `useMemo` AND `useCallback`

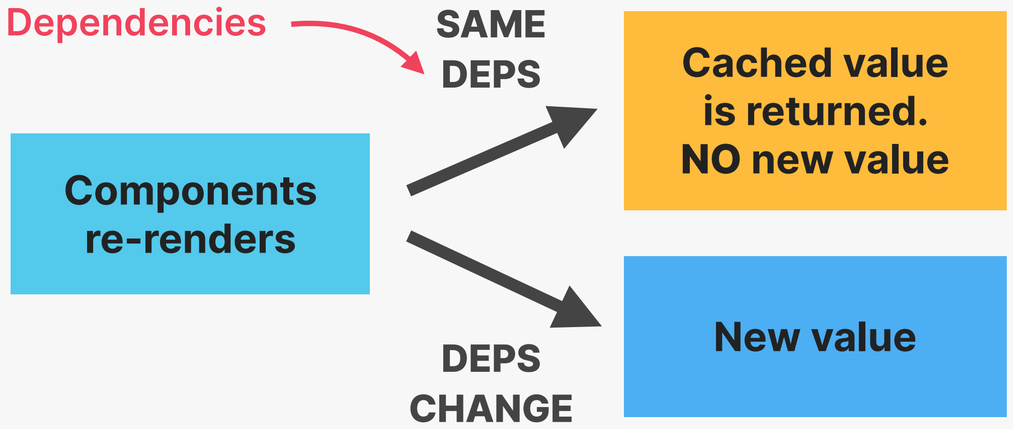
`useMemo` AND `useCallback`

- 👉 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`

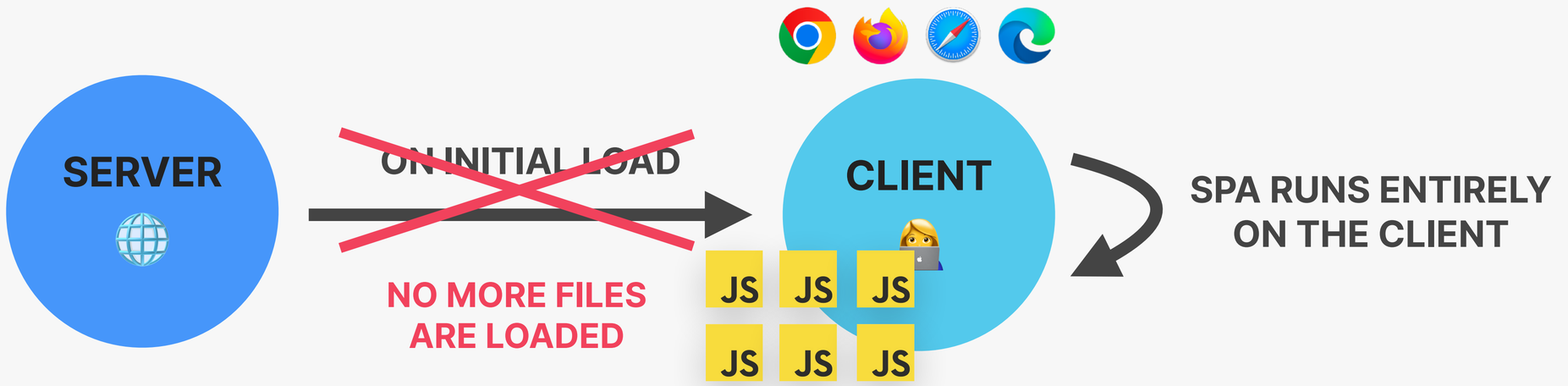
- 👉 Used to memoize values (**`useMemo`**) and functions (**`useCallback`**) **between renders**
- 👉 Values passed into `useMemo` and `useCallback` will be stored in memory ("cached") and **returned in subsequent re-renders**, **as long as dependencies** ("*inputs*") **stay the same**
- 👉 `useMemo` and `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**



Produced by a tool like Webpack (inside create-react-app) or Vite

- 👉 **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

🤖 REMOVING 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**

📦 REMOVING 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:

Avoid these as a beginner

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

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 😎

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**.

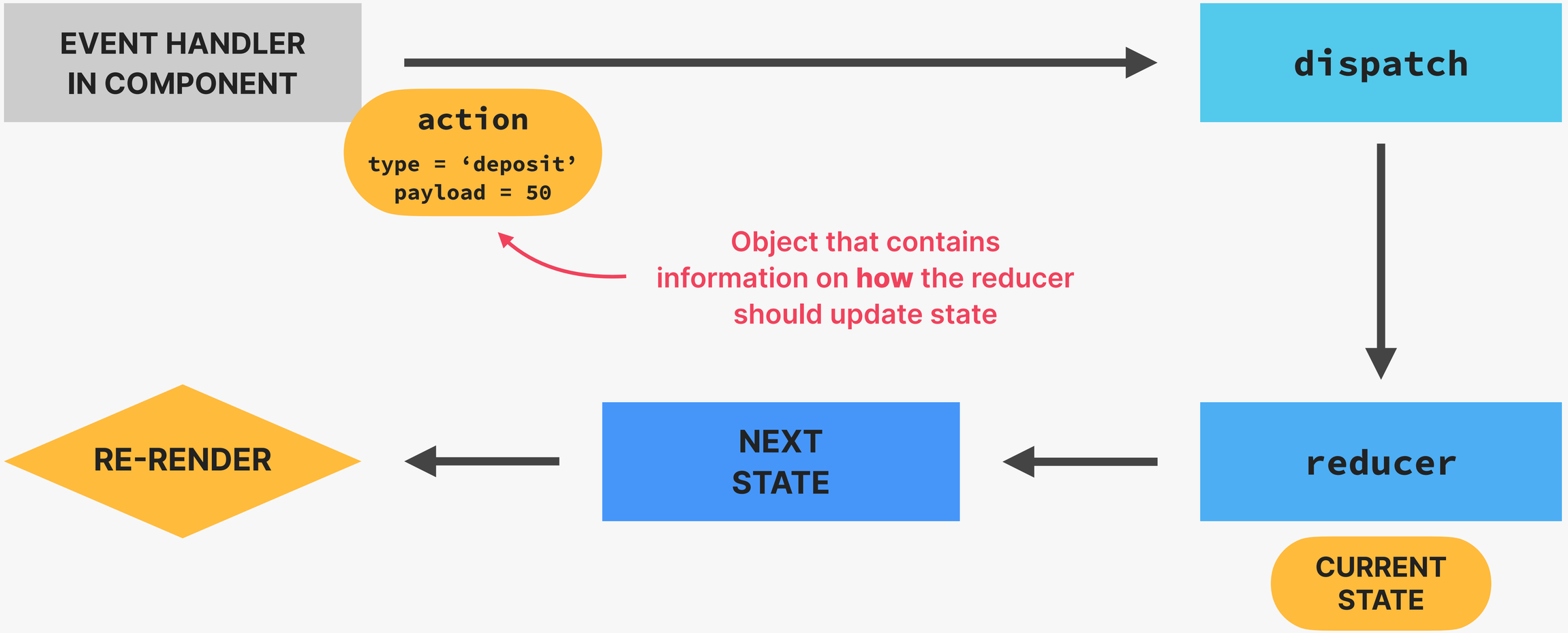
	🏠 LOCAL STATE	🌍 GLOBAL STATE
🪄 UI STATE	👉 `useState` 👉 `useReducer` 👉 `useRef`	👉 Context API + `useState/useReducer` 👉 Redux, Zustand, Recoil, etc. 👉 React Router
🌐 REMOTE STATE	👉 `fetch + useEffect + useState/useReducer`	👉 Context API + `useState/useReducer` 👉 Redux, Zustand, Recoil, etc. 👉 React Query 👉 SWR 👉 RTK Query

Ideal use case for Redux, when there is **lots** of state that updates frequently

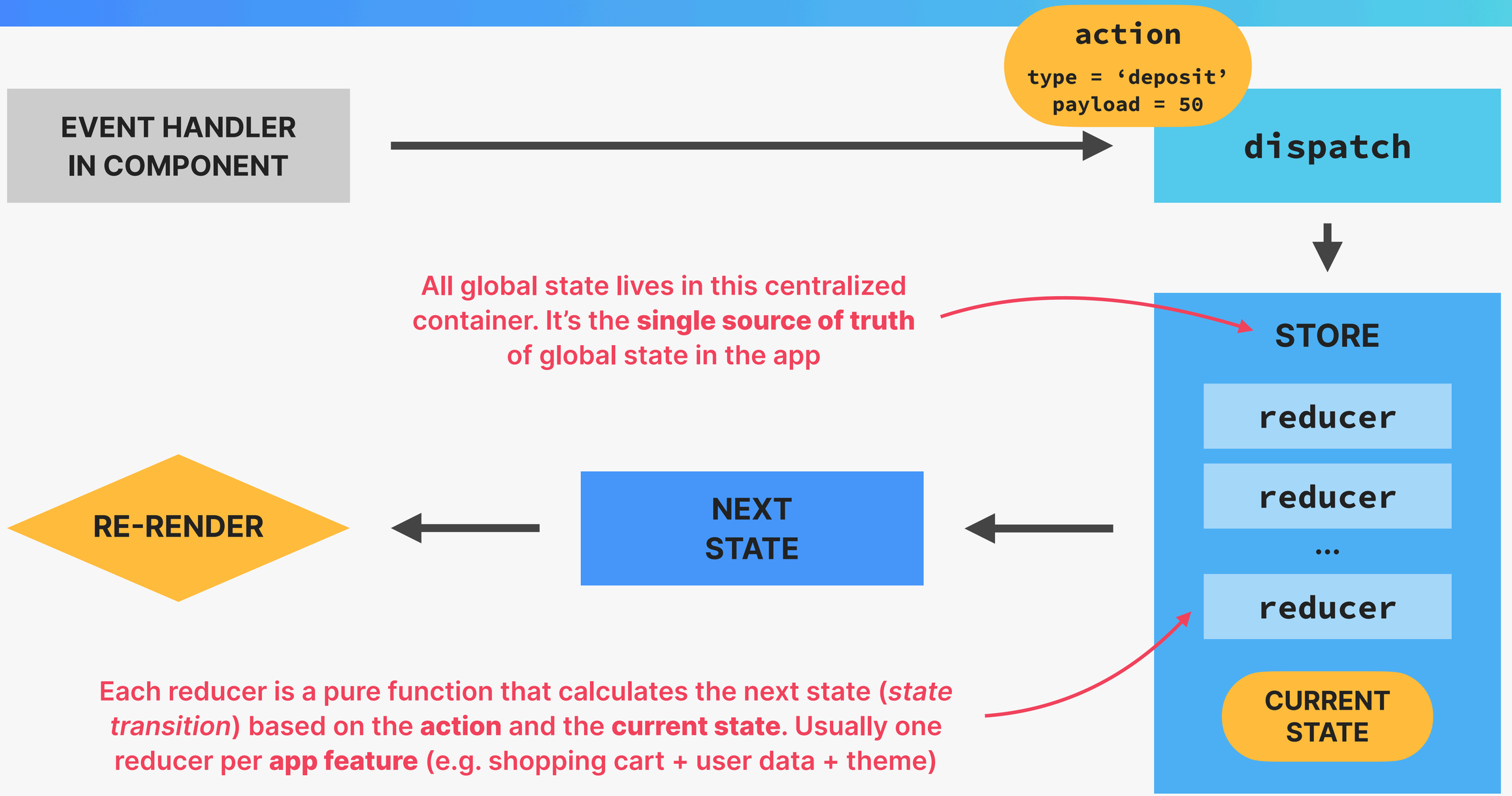
For *remote* global state, we have better, more specialized tools

THE REASON WHY MANY APPS DON'T USE REDUX

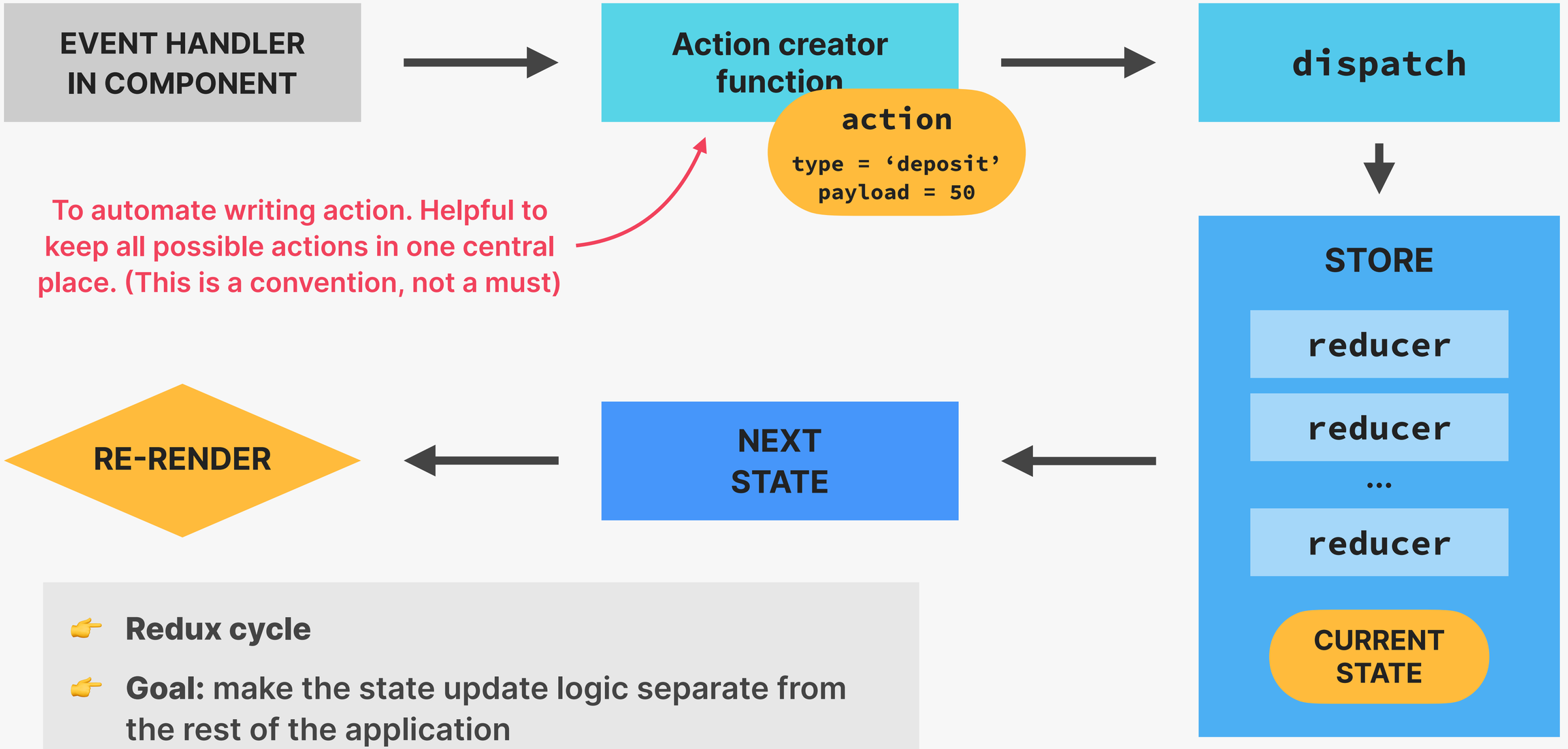
THE **MECHANISM** OF THE USEREDUCER HOOK



THE **MECHANISM** OF REDUX



THE **MECHANISM** OF REDUX



- 👉 **Redux cycle**
- 👉 **Goal:** make the state update logic separate from the rest of the application

THE **MECHANISM** OF REDUX

EVENT HANDLER IN COMPONENT

I would like to deposit $50 from my account

Action creator function

```
action
type = 'deposit'
payload = 50
```

dispatch

STORE

- reducer
- reducer
- ...
- reducer

CURRENT STATE

NEXT STATE

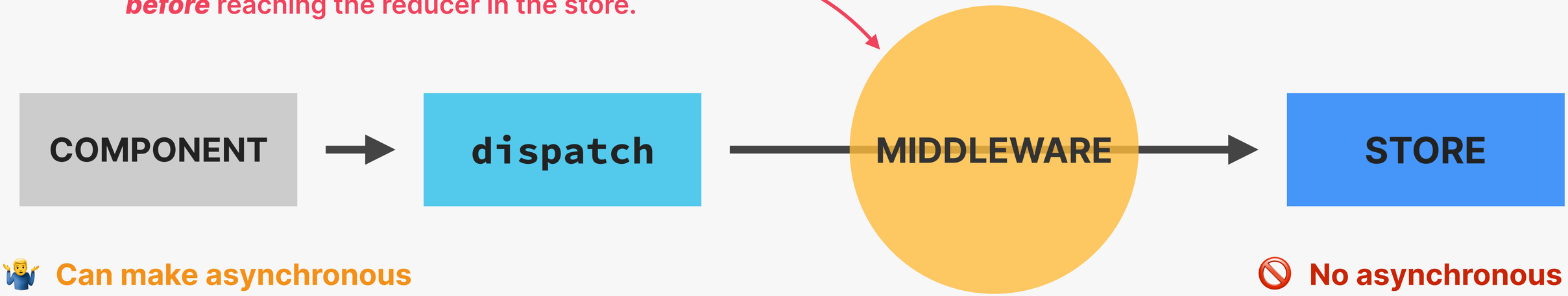
RE-RENDER

👉 **REAL-WORLD TASK:** DEPOSITING $50 INTO YOUR BANK ACCOUNT

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.

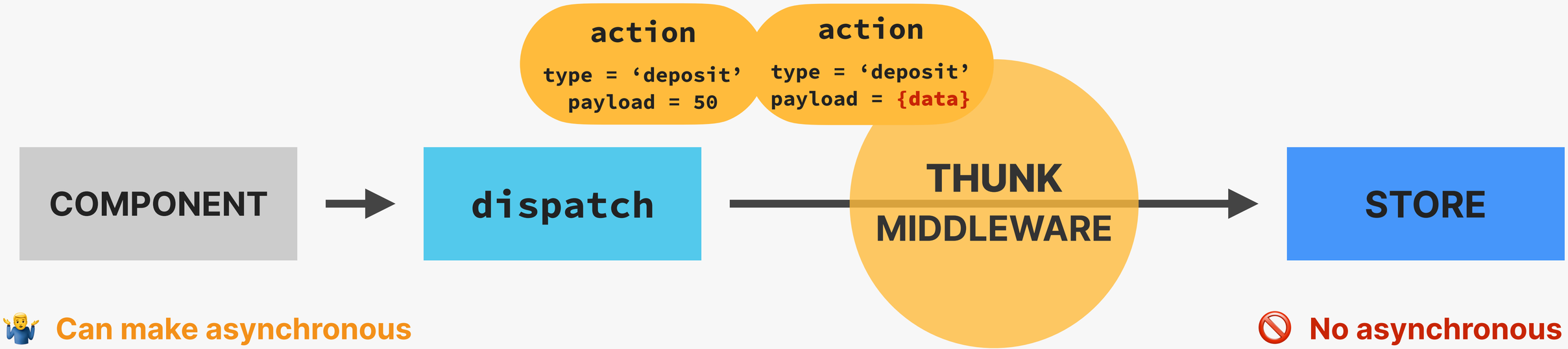
👍 The place for side effects

🚫 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

For example in the compound component pattern

These are not super common in UI state

REDUX

*"Use Redux for global state management in **large** apps"*

- 👉 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)*

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

PART 04

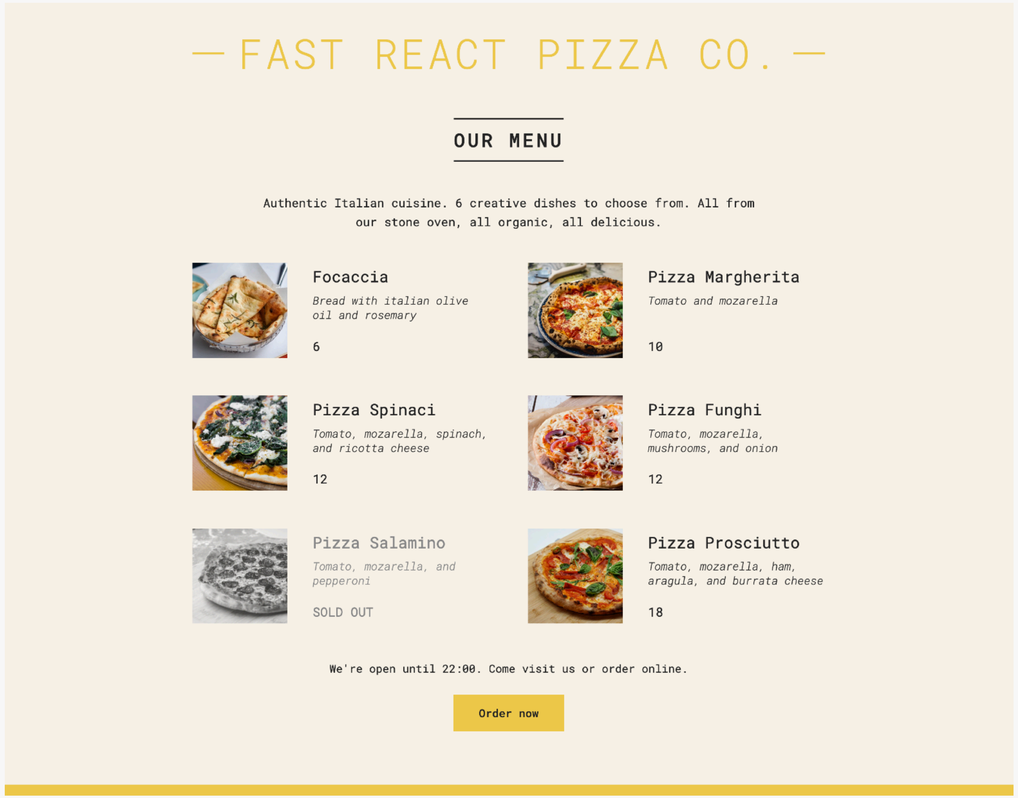
—

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.

- 👉 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

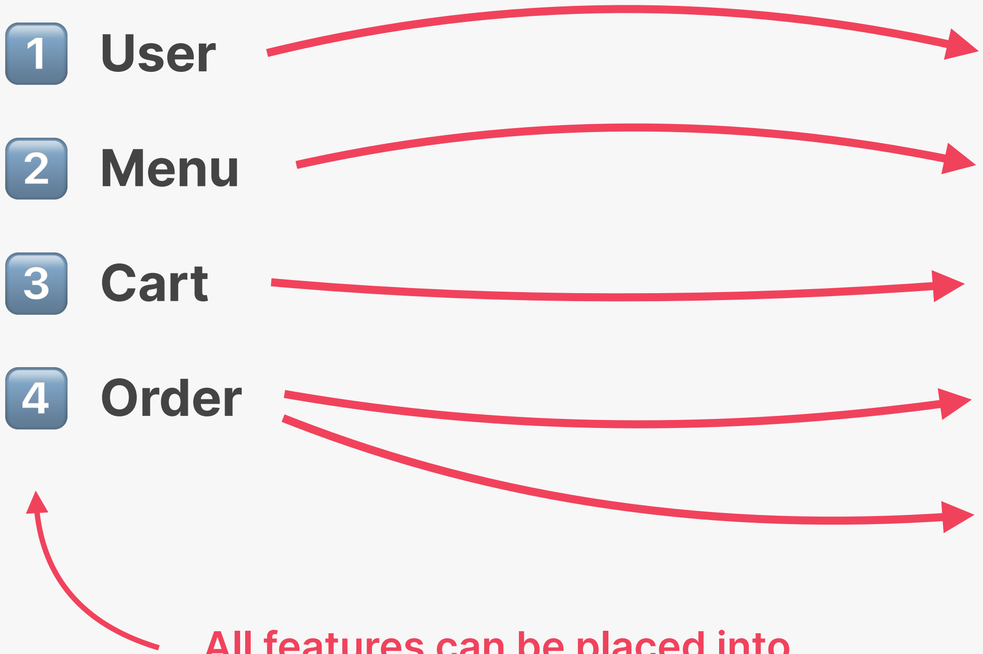
STEP 2 + 3

FEATURE CATEGORIES

1. User
2. Menu
3. Cart
4. Order

NECESSARY PAGES

1	Homepage	`/`
2	Pizza menu	`/menu`
3	Cart	`/cart`
4	Placing a new order	`/order/new`
5	Looking up an order	`/order/:orderID`



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

TYPES OF STATE

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)*

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

👉 **Routing**	**React Router**	*The standard for React SPAs*
👉 **Styling**	**tailwindcss**	*Trendy way of styling applications that we want to learn*
👉 **Remote state management**	**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.*
👉 **UI State management**	**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

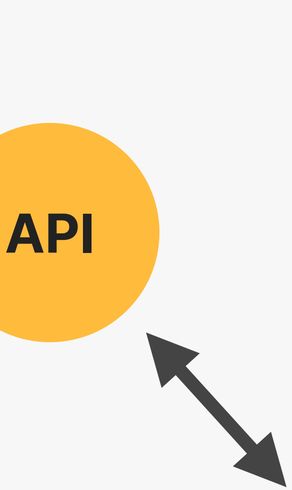
- 👎 **Markup (HTML or JSX) looks very unreadable**, with lots of class names *(you get used to it)*
- 👎 You have to learn a lot of class names *(but after a day of usage you know fundamentals)*
- 👎 You need to install and set up tailwind on each new project
- 👎 You’re giving up on “vanilla CSS” 😢

👋 **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



- 👉 “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

STEP 1

- 👉 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

👨‍💼 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
- 👉 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 ta
  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 (ou
  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, ma
- 👉 App needs a dark mode

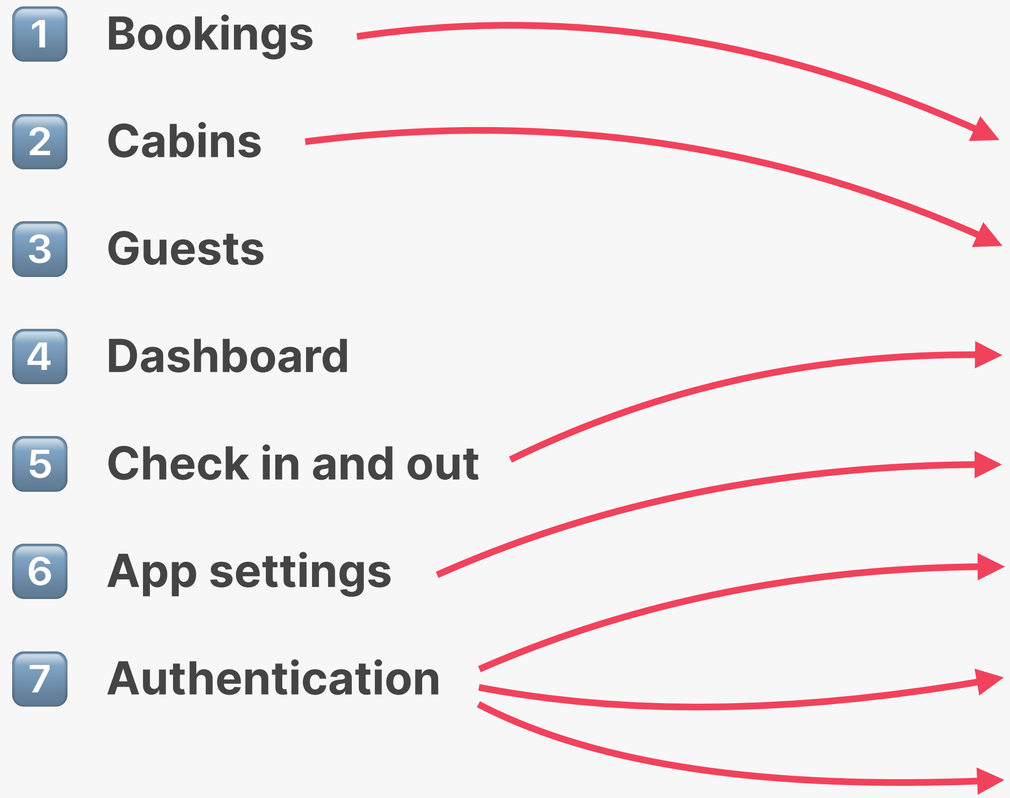
SETTINGS

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

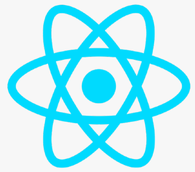
1	Dashboard	`/dashboard`
2	Bookings	`/bookings`
3	Cabins	`/cabins`
4	Booking check in	`/checkin/:bookingID`
5	App settings	`/settings`
6	User sign up	`/users`
7	Login	`/login`
8	Account settings	`/account`

👋 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**

STEP 4

👉 **Routing**	React Router	*The standard for React SPAs*
👉 **Styling**	styled components	*Very popular way of writing component-scoped CSS, right inside JavaScript. A technology worth learning*
👉 **Remote state management**	React Query	*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*
👉 **UI State management**	`Context API`	*There is almost no UI state needed in this app, so one simple context with `useState` will be enough. No need for Redux*
👉 **Form management**	React Hook Form	*Handling bigger forms can be a lot of work, such as manual state creation and error handling. A library can simplify all this*
👉 **Other tools**	React icons / React hot toast / Recharts / date-fns / Supabase	

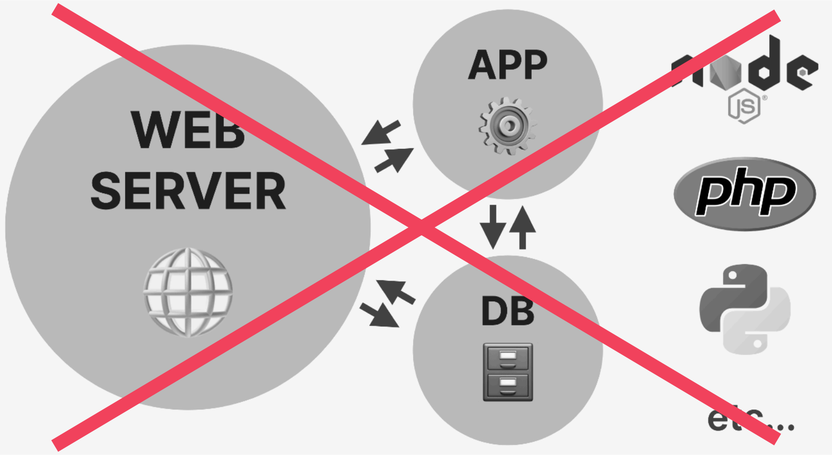
SUPABASE CRASH COURSE: BUILDING A BACK-END!

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**

BACK-END



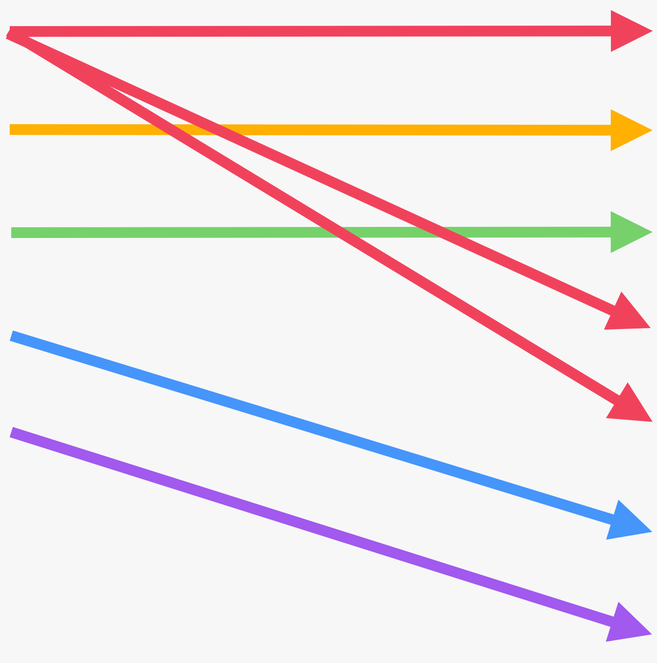
WITH SUPABASE, WE DON'T NEED TO DO ANY OF THIS MANUALLY! IT'S ALL INCLUDED

supabase

MODELING **STATE**

STATE “DOMAINS” / “SLICES”

1. Bookings
2. Cabins
3. Guests
4. Settings
5. Users



FEATURE CATEGORIES

1. Bookings
2. Cabins
3. Guests
4. Dashboard
5. Check in and out
6. App settings
7. Authentication

👉 All this state will be **global remote state**, stored within Supabase

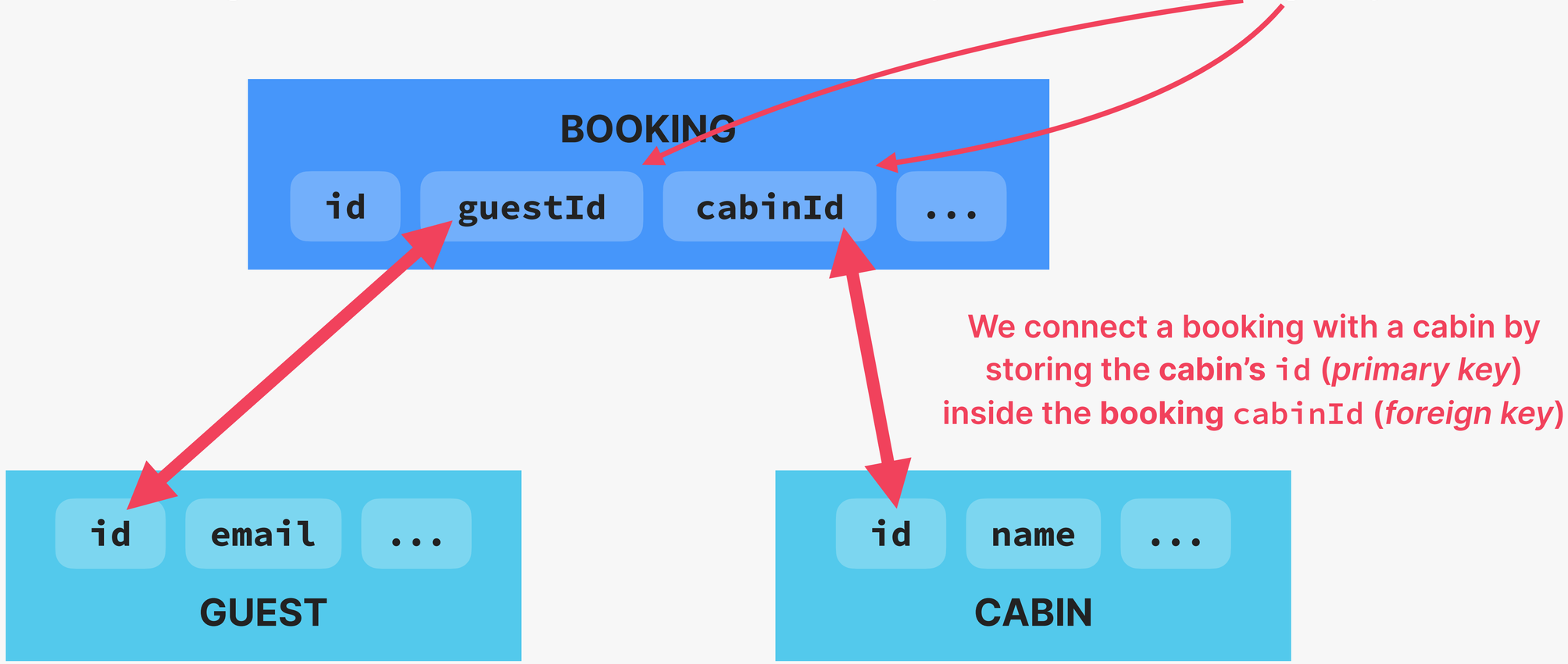
👉 There will be one **table** for each state “slice” in the **database**

supabase

React Query

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**

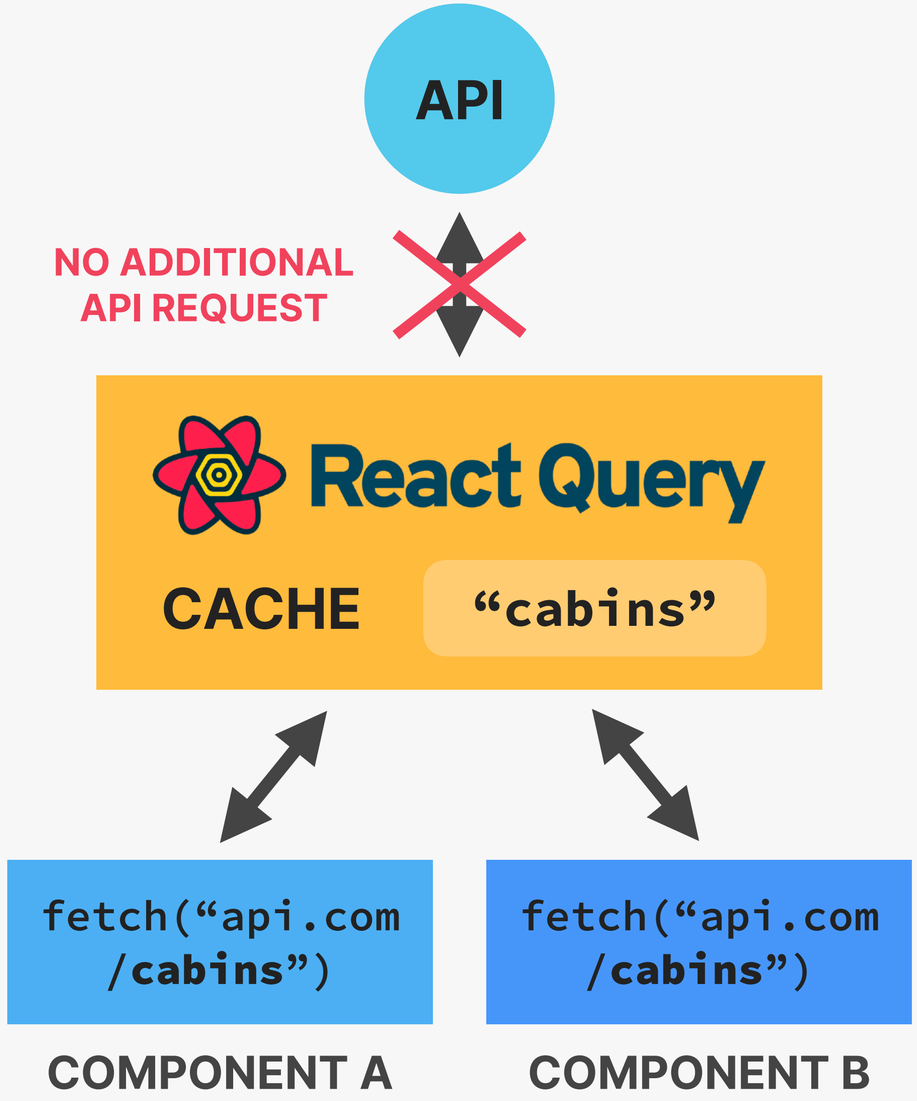


REACT QUERY: MANAGING REMOTE STATE

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?

👉 I need to reuse:

- UI
  - Components and props
  - `children` prop
  - Render props pattern
  - Compound component pattern
- STATEFUL LOGIC
  - Custom hooks

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

To customize the component's content

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

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