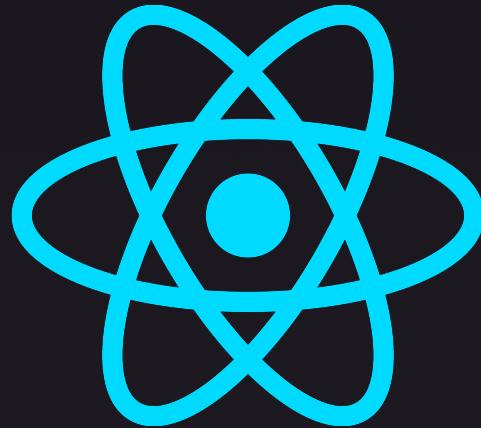
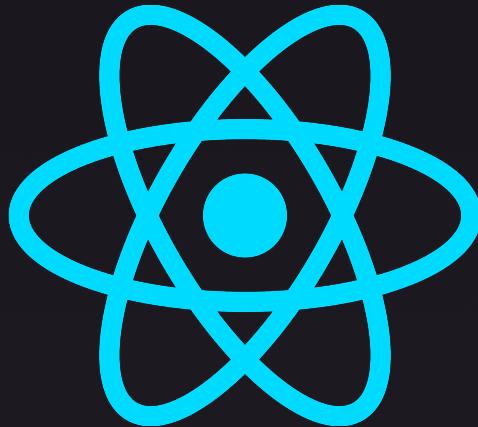


What is React?

And why exactly would you use it?



React is a library for building user interfaces

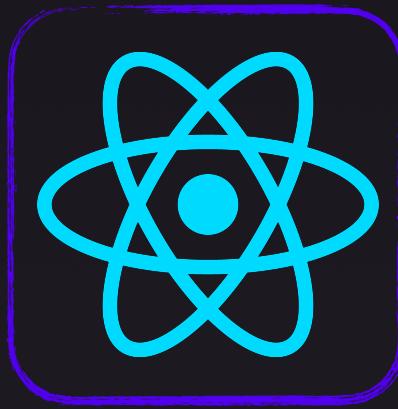


A **JavaScript library for building user interfaces**

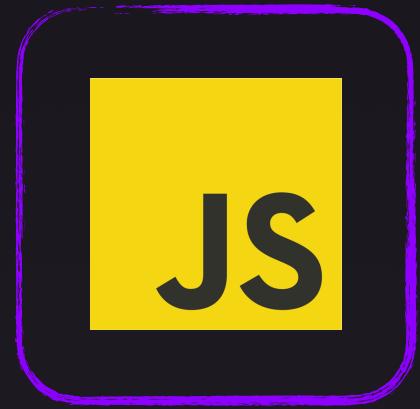
But why would you use it?



Displays &
manages website
content & UI



React builds up
on JavaScript (it's
a library!)

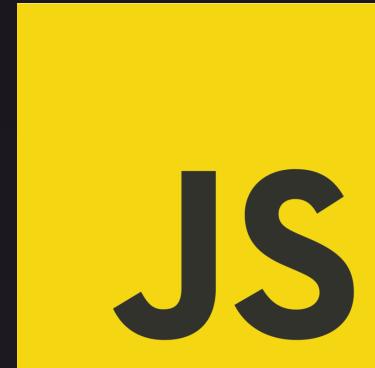


As a React
developer, you'll
write React-specific
JavaScript code



**In the end, it's JavaScript
doing "the magic"**

React just uses JavaScript features

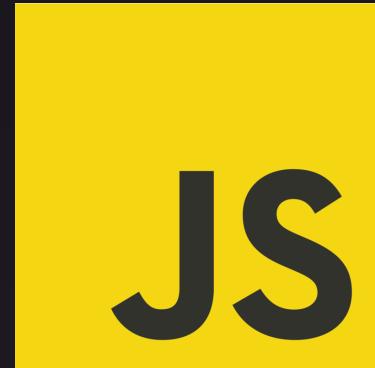


Why not just JavaScript?

JS

Using “Just JavaScript” Typically Isn’t A Great Option

- ▶ Writing complex JavaScript code quickly becomes **cumbersome**
- ▶ Complex JavaScript code quickly becomes **error-prone**
- ▶ Complex JavaScript code often is **hard to maintain or edit**
- ▶ React offers a **simpler mental model**

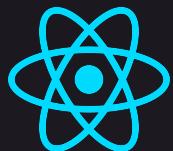


Basic Web Dev & JavaScript
Knowledge is Assumed!

React = Declarative UI Programming

With React, you **define the target UI state(s)** – not the steps to get there!

Instead, React will figure out & perform the necessary steps



Declarative

Define the goal, not the steps

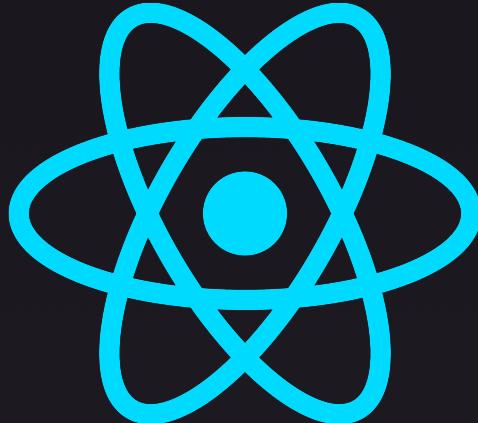
```
let content;  
  
if (user.isLoggedIn) {  
  content = <button>Continue</button>  
} else {  
  content = <button>Log In</button>  
}  
  
return content;
```



Imperative

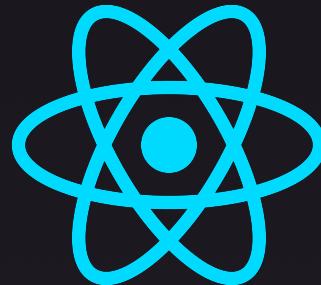
Define the steps, not the goal

```
let btn = document.querySelector('button');  
  
if (user.isLoggedIn) {  
  button.textContent = 'Continue';  
} else {  
  button.textContent = 'Log In';  
}  
  
document.body.append(btn);
```



Our First React App

Time to get your hands dirty!



Our First React App

Time to get your hands dirty!

Use the attached (updated!) demo app

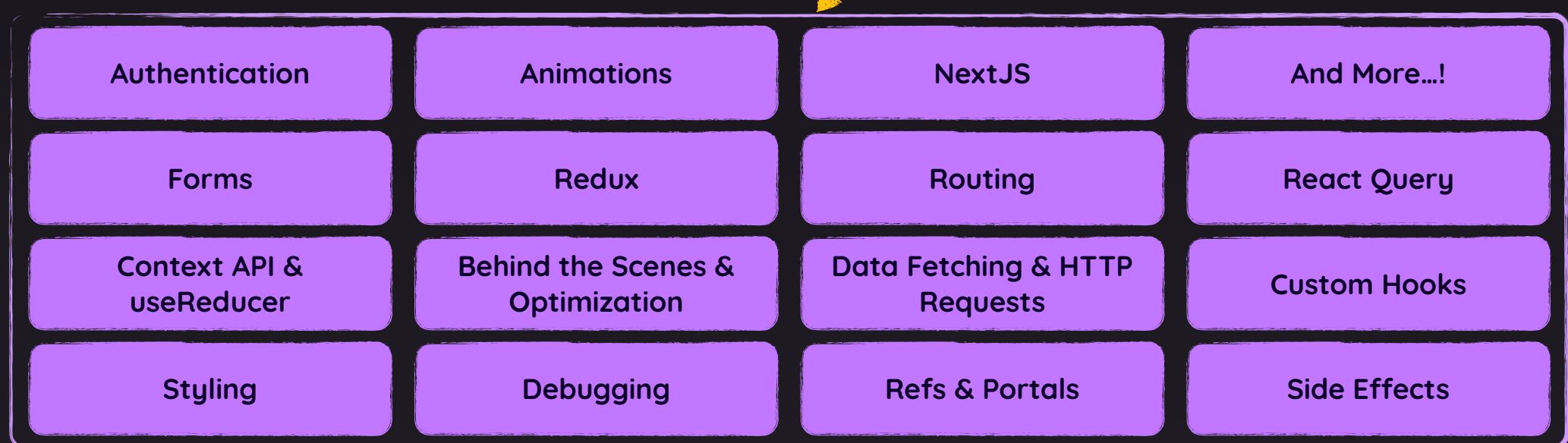
Add a fourth button

Load the content for this button from the “content” array



It's a Modular Course!

Deep dives & advanced concepts



React Essentials

Essentials - Deep Dive

Essentials - Practice

Solid React Foundation: Core concepts every React developer must know

JavaScript Refresher

Getting Started

Refresher module, in case it's been some time since you last worked with JS



One Course, Two Paths



Standard Path

Recommended

Start with lecture 1 in section 1

Complete the course lecture
by lecture & section by section

You'll learn React step-by-step, from the ground up & in great detail!



Summary Path

If you got limited time

Complete the “Summary” section

You'll get a good overview of basic & advanced React concepts

It's also a great refresher after finishing the course



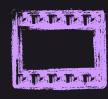
How To Get The Most Out Of This Course



Meet the Prerequisites

Basic web dev & JavaScript knowledge is required

Use the JavaScript refresher if needed



Watch the Videos

Watch them at your pace

Slow me down or speed me up

Pause & rewind

Repeat sections



Practice!

Complete coding exercises

Pause & practice on your own

Build dummy demo projects



Help each other

Use code attachments if stuck

Ask & answer in the Q&A section

Find fellow developers on our Discord



React Code Must Be Transformed!



React Code

JavaScript code that typically uses JSX (“HTML in JavaScript”)



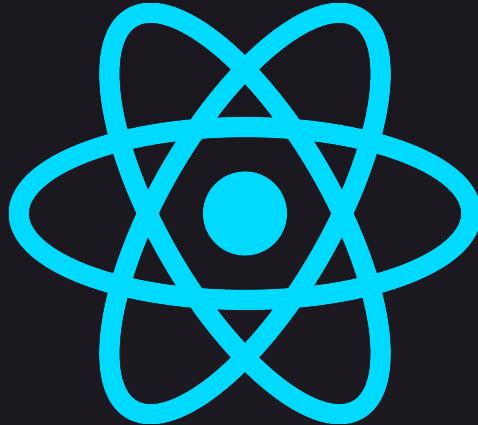
Code that runs in the browser

JavaScript code without JSX

Code is **transformed** & **optimized** (e.g., unnecessary whitespace is removed)

Handled by build tool (e.g., Vite)

→ That's why a “more complex” project setup (which includes such a build tool) is needed



You Need A React Project

If you want to write React code



JavaScript Refresher

In case it's been some time...



This course section is optional!

It's recommended if you haven't used JavaScript in a while or if you don't have a lot of JavaScript experience



This section **does not replace** a
JavaScript course!

But it will revisit **crucial JavaScript concepts** needed
for building React apps

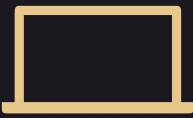


JavaScript Refresher

In case it's been some time since you last worked with JavaScript

- ▶ Core Syntax & Rules
- ▶ Essential, Modern JavaScript Features
- ▶ Key JavaScript Features Used In React Apps

JavaScript Can Be Executed In Many Environments



In the Browser
(i.e., as part of websites)

JavaScript code can be included in any website
The code then executes inside the browser (i.e., on the machine of the website visitor)



On any Computer
(e.g., server-side code)

Thanks to Node.js or Deno, JavaScript code can be executed outside of the browser, too
The code then executes directly on the machine



On mobile Devices
(e.g., via embedded websites)

With extra technologies like Capacitor or React Native, you can build mobile apps based on JavaScript
The code then executes on the mobile device

Adding JavaScript Code To A Website

Between <script> Tags

```
<script>  
alert('Hello')  
</script>
```

Can quickly lead to unmaintainable & complex HTML files

Typically only used for very short scripts

Via <script> Import

```
<script src="script.js"></script>
```

Separates HTML & JavaScript code

Maintaining complex JS-powered apps becomes easier

JavaScript code is just plain **text!**

How Code Is Executed



Code is read top to bottom,
left to right



JavaScript Code Consists Of “Statements”

Statement

The “thing” that gets executed



Keywords

Built into JavaScript

Required to “tell”
JavaScript that a
certain feature is used

`let, if, for, ...`



Identifiers

Defined by developers

Used to identify
commands (functions),
variables (values) etc.

`alert(), age, ...`



Values / Expressions

Defined by developers

Hardcoded values or
expressions that
produce new values

`‘Hello world’, 5 - 3, ...`

Keywords

Keywords enable language features



let, const, if, for, function, ...

Identifiers

Identifiers identify “things”



Variables



Functions
("Commands")



Parameters



Property

JavaScript code is **case-sensitive!**

Identifiers Must Follow Certain Rules & Recommendations

#1

Must not contain whitespace or special characters (except \$ and _)

Valid: \$userName, age, user_name, data\$, ...

Invalid: %userName, age/, user name, ...

#2

May contain numbers but must not start with a number

Valid: user3, us3r, ...

Invalid: 3user, 11players, ...

#3

Must not clash with reserved keywords

Valid: user, age, data, ...

Invalid: let, const, if, ...

#4

Should use camelCasing

Recommended: userName, isCorrect, ...

Uncommon: user_name, iscorrect, ...

#5

Should describe what the “thing” it identifies contains or does

Recommended: userName, isCorrect, loadData, ...

Uncommon: userDataPoint, correctness, dataLoader, ...

**Semicolons are optional
(in most cases)!**

Whitespace is ignored in many cases!

Use it to format your code & improve readability
But avoid adding too much whitespace

**React projects use a
build process**

The code **you write** is **not** the
code that gets **executed** (like
this) in the browser

Your code is **transformed**
before it's handed off to the
browser

React Projects Use A Build Process

1

Raw, unprocessed React code **won't execute** in the browser



JSX

JSX is not a default JavaScript feature

2

In addition, the code would **not be optimized for production** (e.g., not minified)



React projects require a build process that transforms your code

create-react-app, vite etc. give you such a build process (no custom setup or tweaking needed)

There Are Different Types Of Values

String

Text values

Wrapped with single or double quotes

Can also be created with backticks (`)

```
"Hello World"  
'Max'  
`Hi there`
```

Number

Positive or negative

With decimal point (float) or without it (integer)

```
5  
-23  
3.14  
-8.12
```

Boolean

True or false

A simple “Yes” or “No” value type

Typically used in conditions

```
true  
false
```

Null & undefined

“There is no value”

undefined: Default if no value was assigned yet

null: Explicitly assigned by developer (reset value)

```
undefined  
null
```

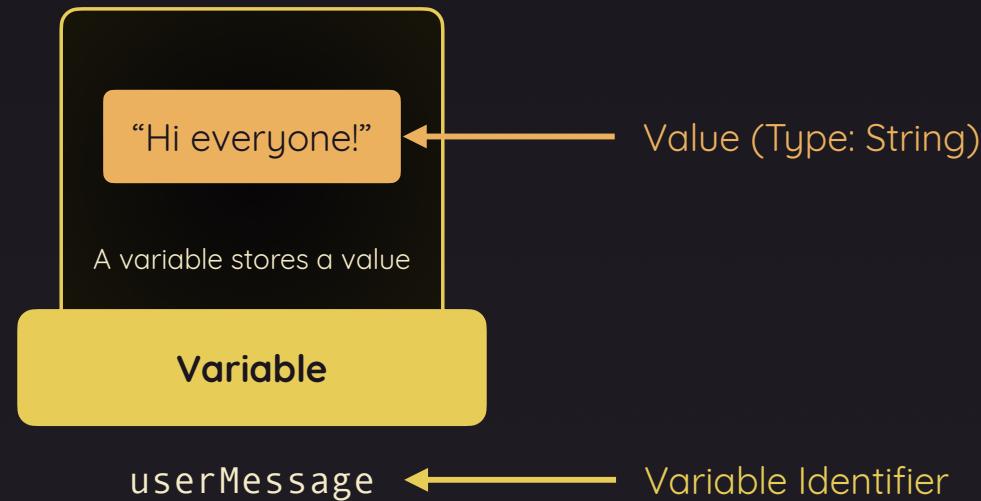
Additionally



Objects

Variables store Values

Variables Are Data Containers



Why Use Variables?

1

Reusability

Store a value in a variable once and use it as often and in as many places as needed

2

Readability

Organize your code over several lines rather than cramming everything into a single line

Variables vs Constants

Variables

Defined via `let`

Can be re-assigned

(i.e., the stored value can be overwritten)

```
let age = 34;
```

Allowed

age = 29;



Constants

Defined via `const`

Cannot be re-assigned

(i.e., the stored value can't be overwritten)

```
const age = 34;
```

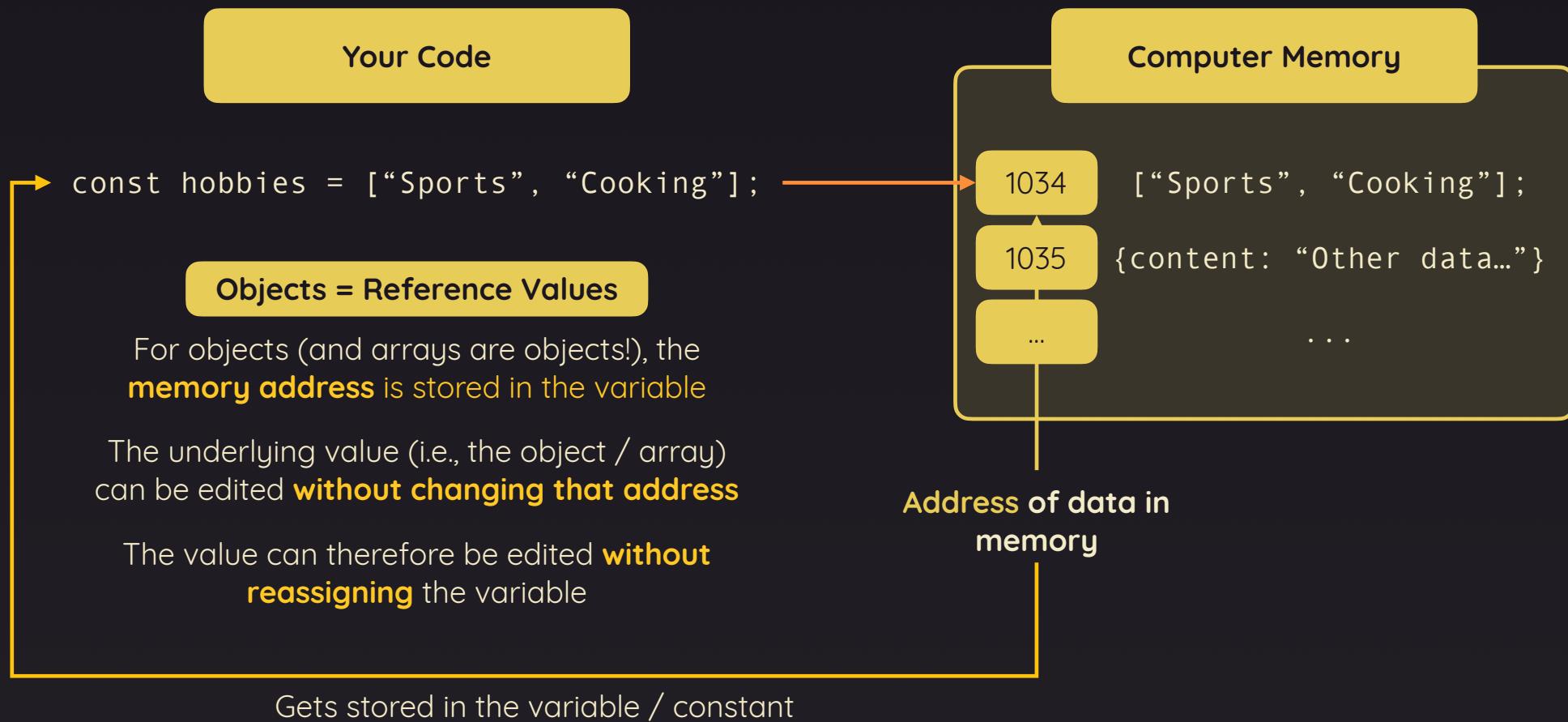


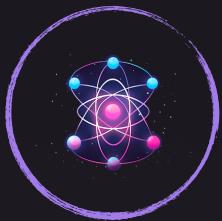
age = 29;

Error

Values can be hardcoded
But they can also be derived
via Expressions & Operators

Reference Values





React Essentials

Components, JSX & State

- ▶ Building User Interfaces with **Components**
- ▶ Using, Sharing & Outputting **Data**
- ▶ Handling User **Events**
- ▶ Building Interactive UIs with **State**



About This Section

What you'll learn in this section

- How to build web user interfaces with React Components
- How to make those user interfaces interactive with State
- How to output static, dynamic & conditional data

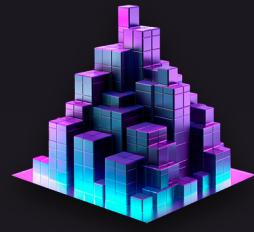
What you'll be able to do after finishing this section

- Build basic, interactive React web applications
- Use React Components to build user interfaces with ease
- Handle user events & output data

What you should know before starting this section

- You must have basic JavaScript knowledge (variables, objects, arrays, functions, control structures, DOM)
- NO prior React knowledge is required!





Components



Components Are UI Building Blocks





React Apps Are
Built By Combining
Components



Components Are The Foundation



Components



JSX

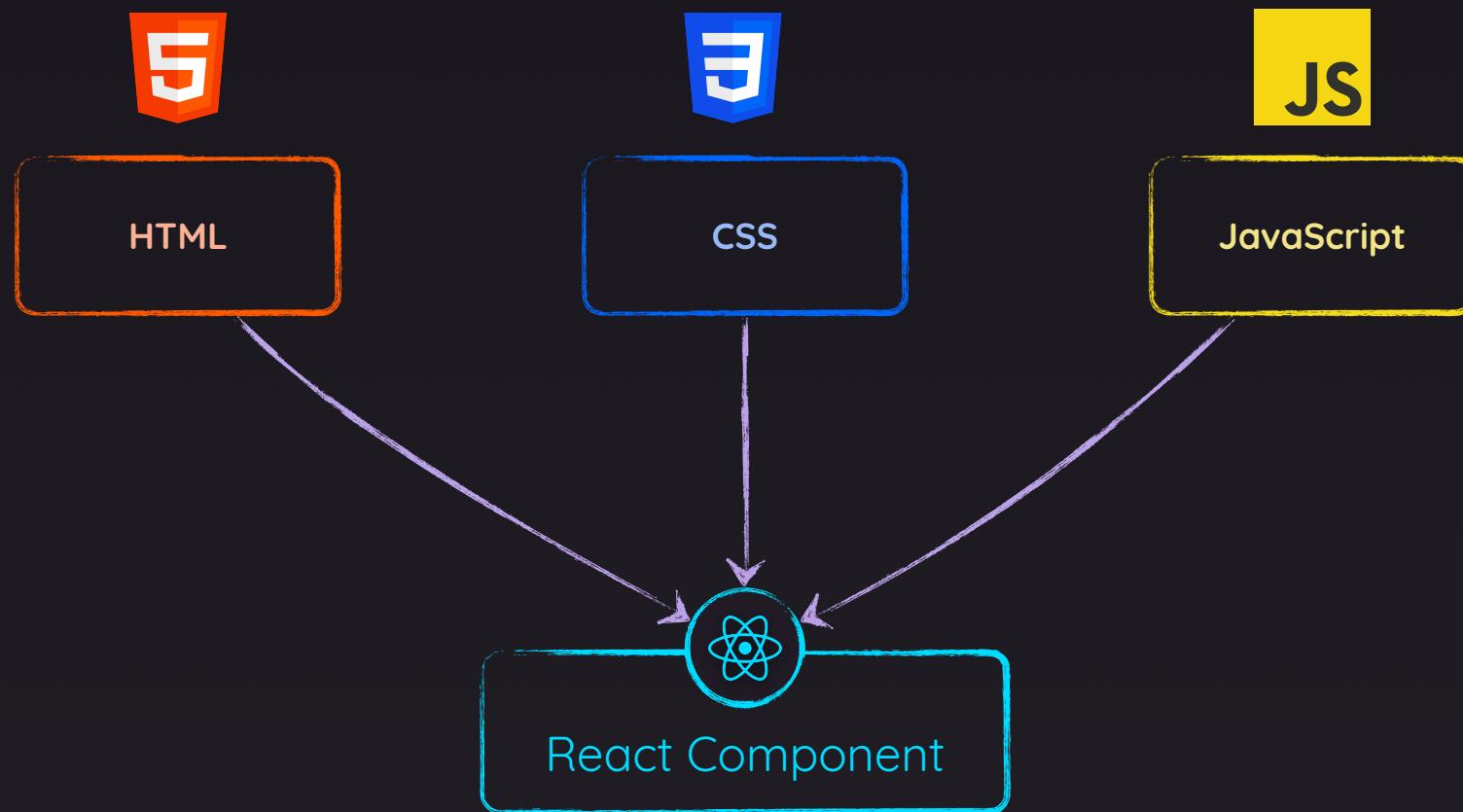


State



Props

Creating Components



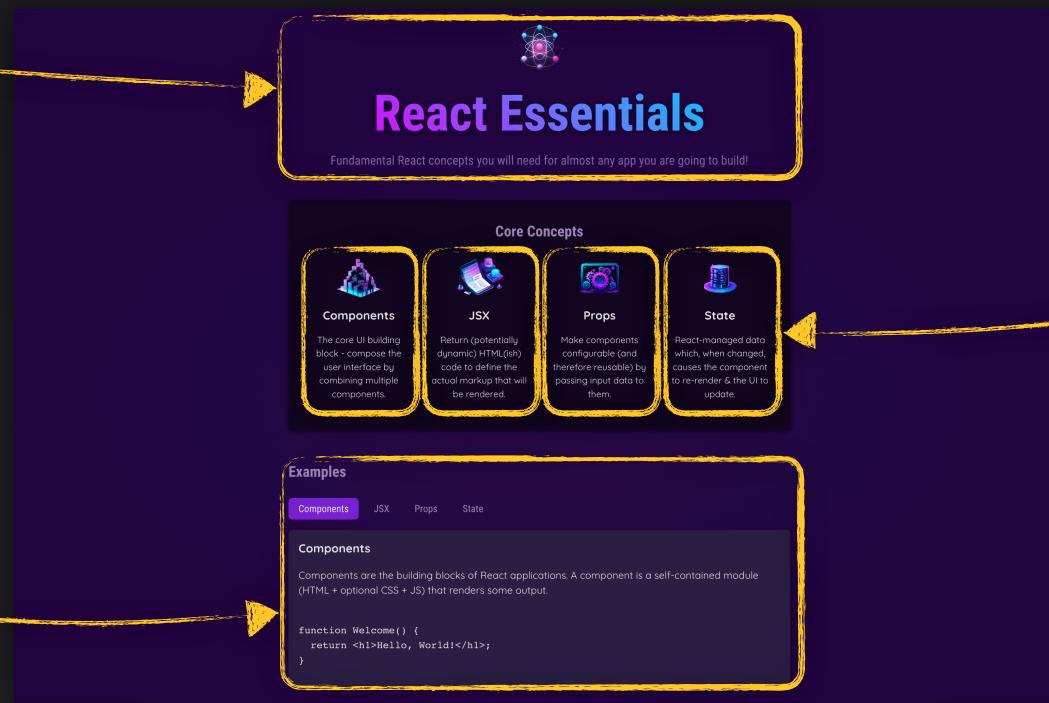
Build User Interfaces With Components

Any website or app can be broken down into smaller building blocks: **Components**

Component Header

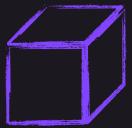
It can therefore also be built by creating & combining such components

Component Interactive Tabs



Components
Core Concept Items

Why Components?



Reusable Building Blocks

Create **small building blocks** & **compose** the UI from them

If needed: **Reuse** a building block in different parts of the UI (e.g., a reusable button)



Related Code Lives Together

Related HTML & JS (and possibly CSS) code is **stored together**

Since JS influences the output, storing JS + HTML together makes sense

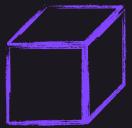


Separation of Concerns

Different components handle different data & logic

Vastly **simplifies** the process of working on complex apps

Why Components?



Reusable Building Blocks

Create **small building blocks** & **compose** the UI from them

If needed: **Reuse** a building block in different parts of the UI (e.g., a reusable button)



Related Code Lives Together

Related HTML & JS (and possibly CSS) code is **stored together**

Since JS influences the output, storing JS + HTML together makes sense



Separation of Concerns

Different components handle different data & logic

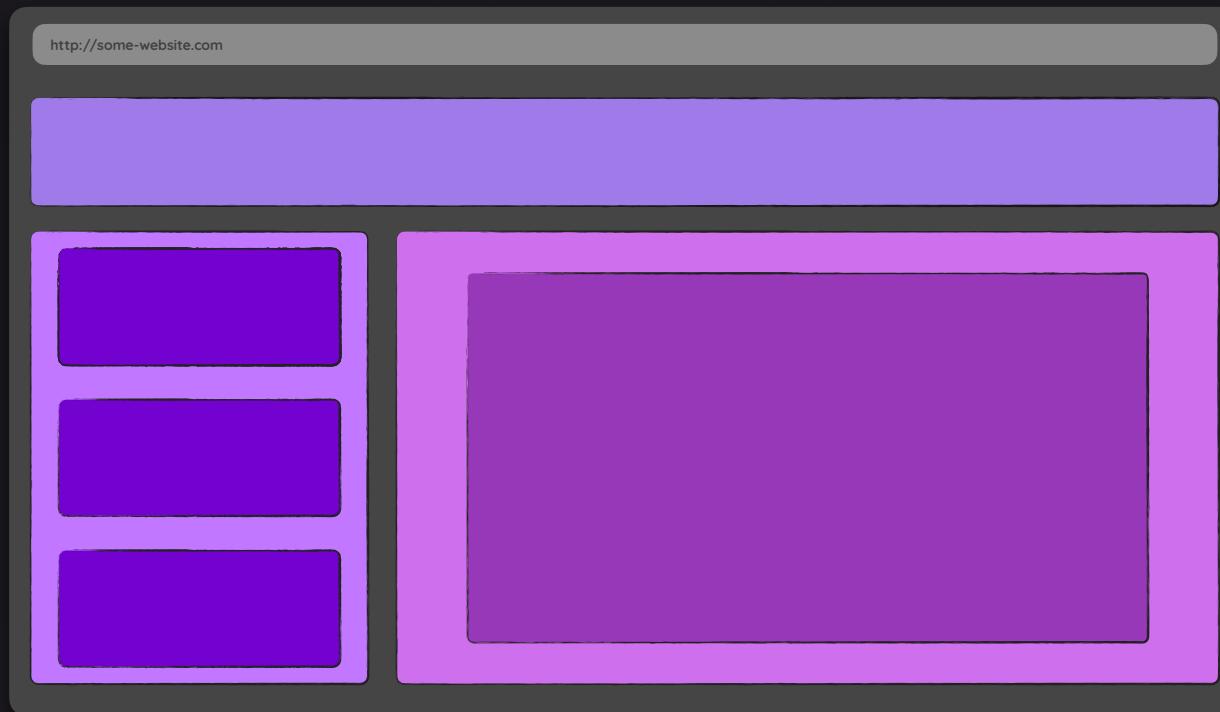
Vastly **simplifies** the process of working on complex apps

Components Can Potentially Be Reused



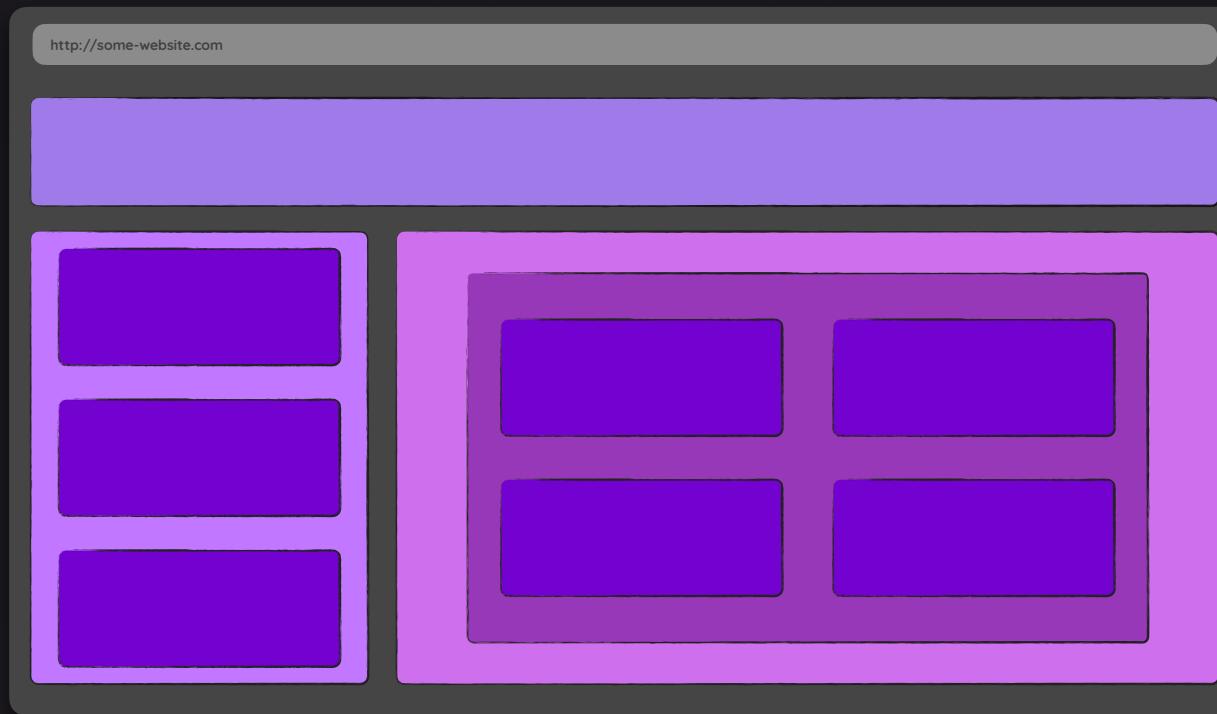


Components Can Potentially Be Reused





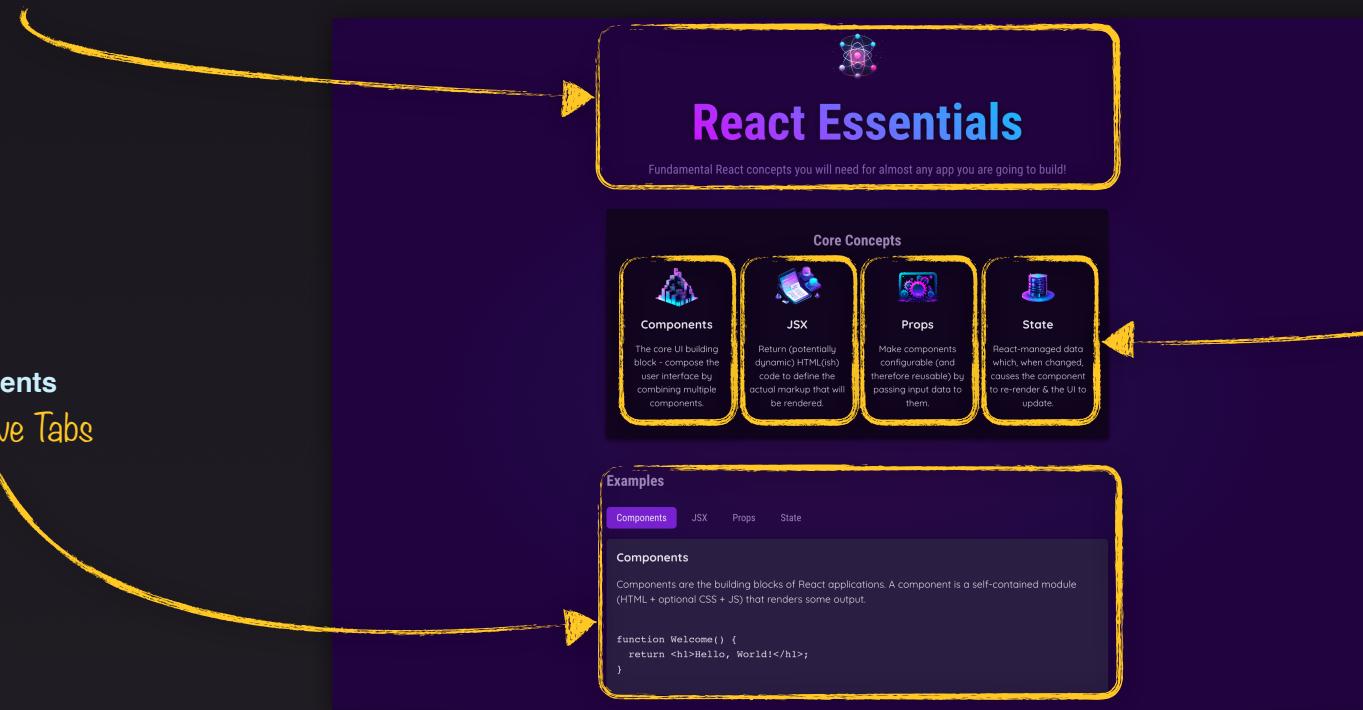
Components Can Potentially Be Reused



Components Can Potentially Be Reused

Components

Header



Components

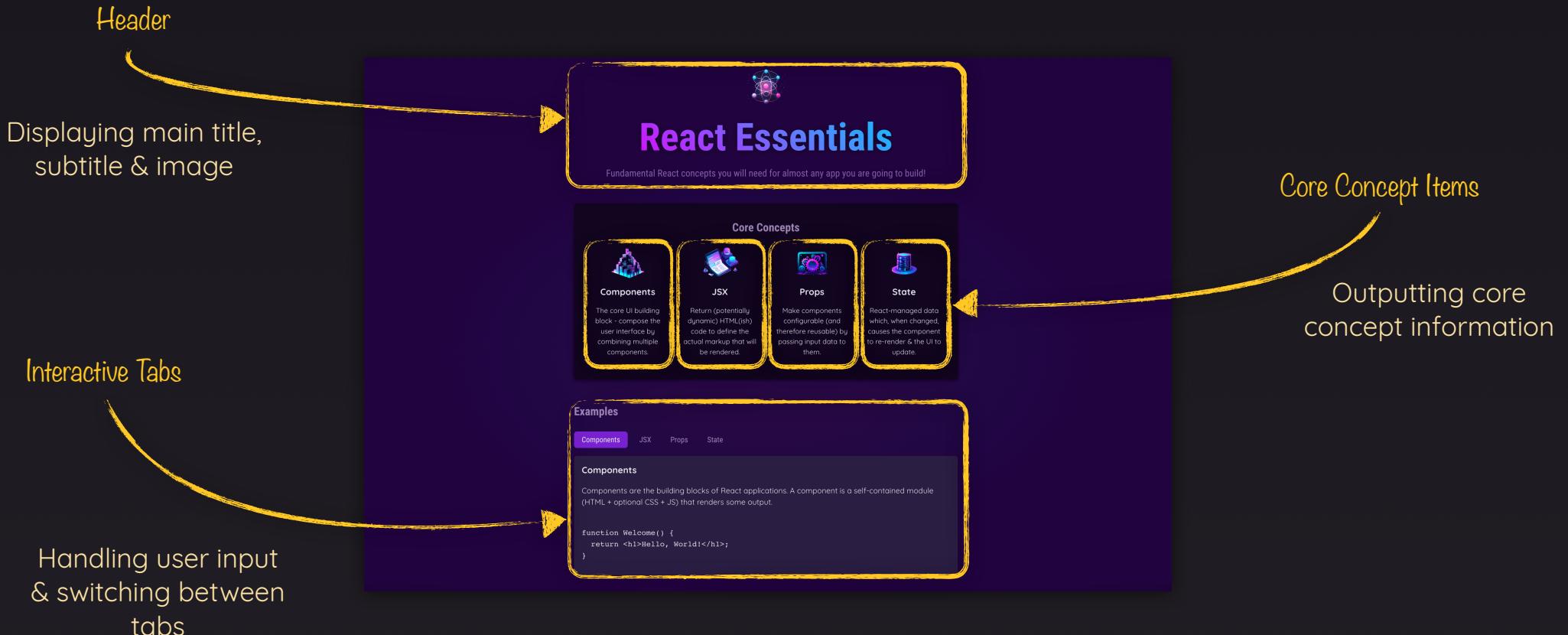
Interactive Tabs

Components

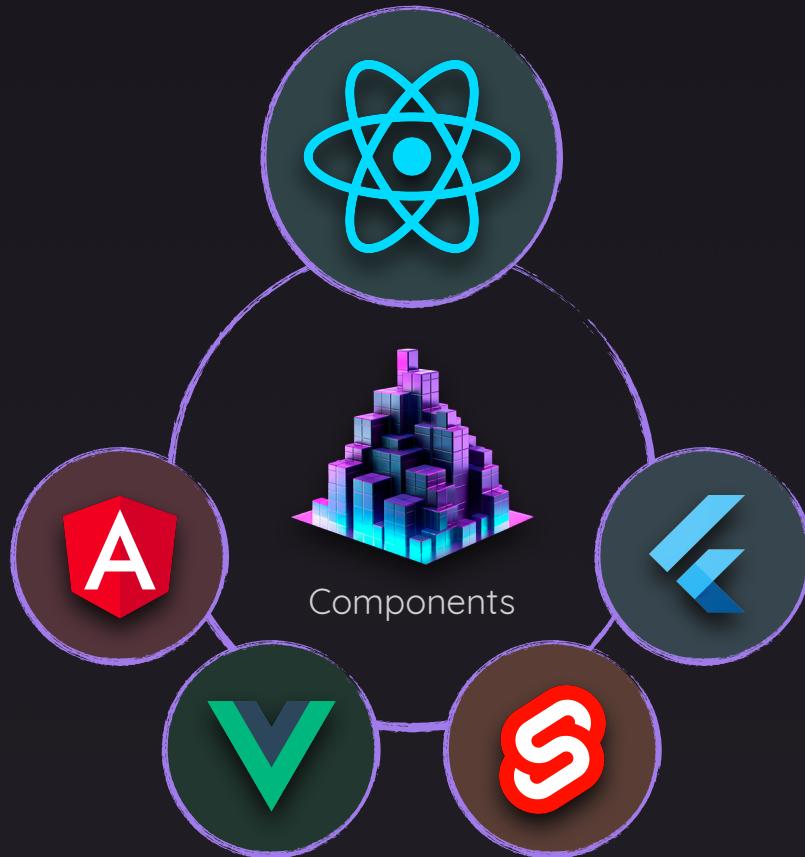
Core Concept Items

The same component is used multiple times with different input data

Different Components, Different Responsibilities



Components Are A Fundamental Concept



Describe The Target UI With JSX

JavaScript Syntax eXtension

JSX

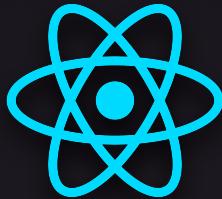
```
<div>  
  
  <h1>Hello World!</h1>  
  
  <p>JSX code is awesome!</p>  
  
</div>
```

Used to describe & create HTML elements in JavaScript in a **declarative** way

BUT

Browsers do not support JSX!

React projects come with a **build process** that **transforms** JSX code (behind the scenes) to code that **does work** in browsers



With React, You Write Declarative Code

You define the target HTML structure & UI – not the steps to get there!

Component Functions Must Follow Two Rules



Name Starts With Uppercase Character

The function name **must start** with an **uppercase** character

Multi-word names should be written in **PascalCase** (e.g., “MyHeader”)

It's **recommended** to pick a name that **describes** the UI building block (e.g., “Header” or “MyHeader”)

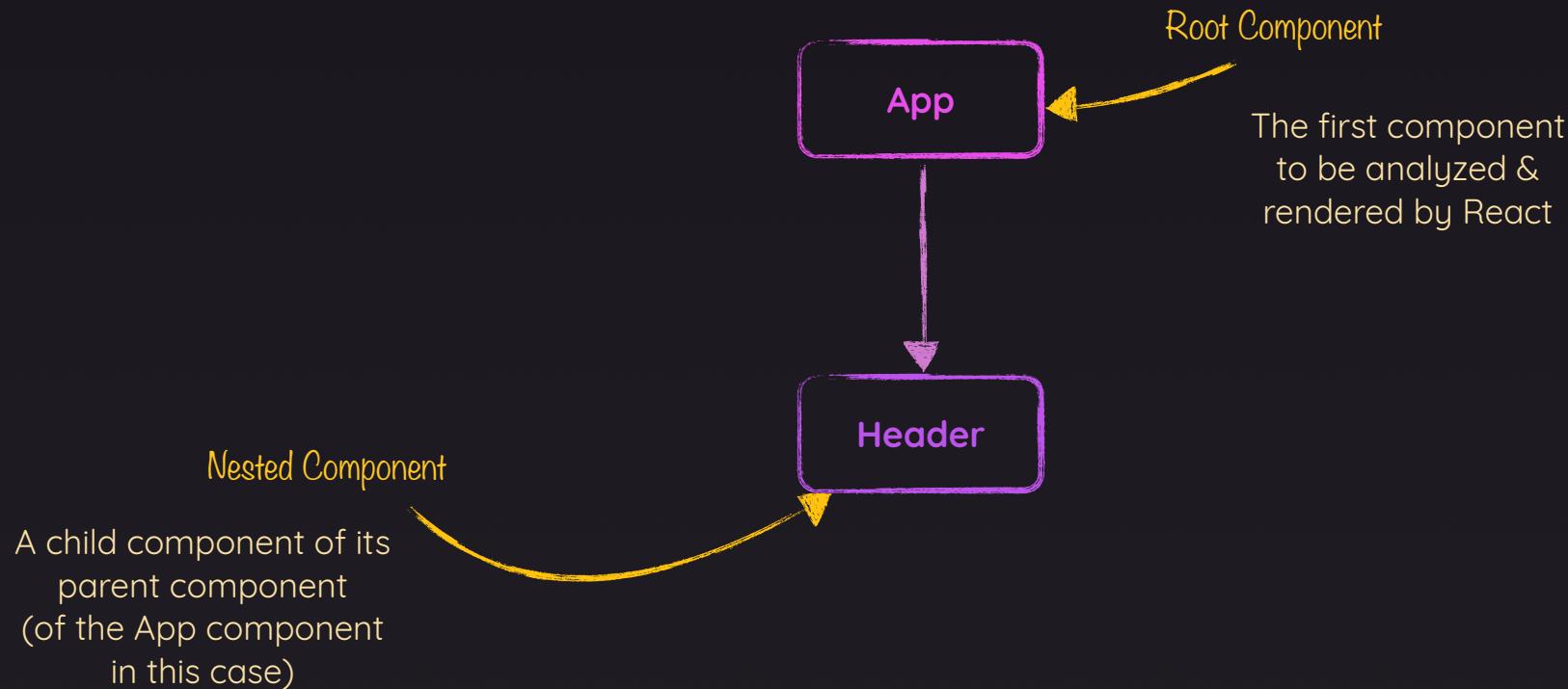


Returns “Renderable” Content

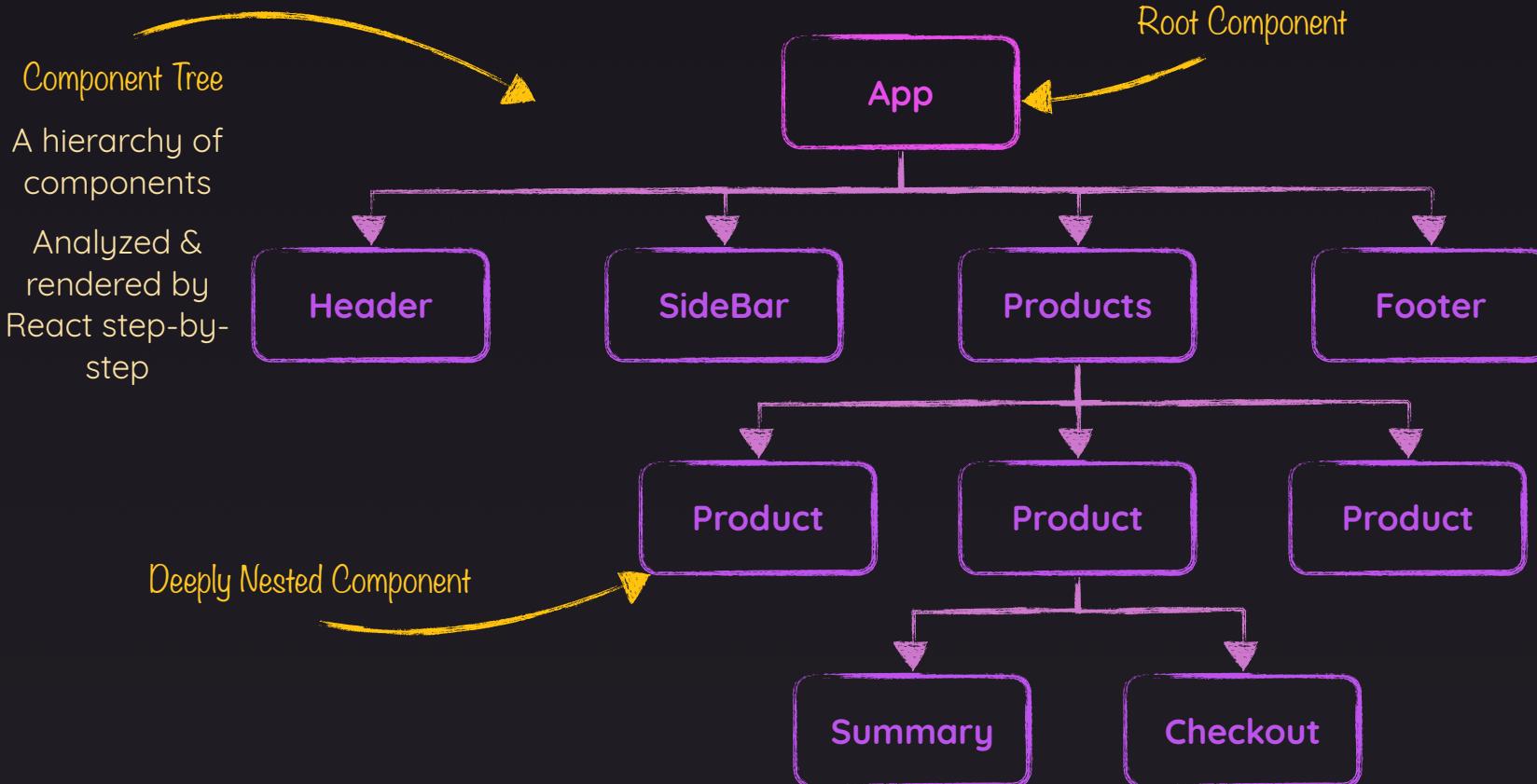
The function must **return** a value that can be **rendered** (“displayed on screen”) by React

In **most** cases: Return **JSX**
Also allowed: string, number, boolean, null,
array of allowed values

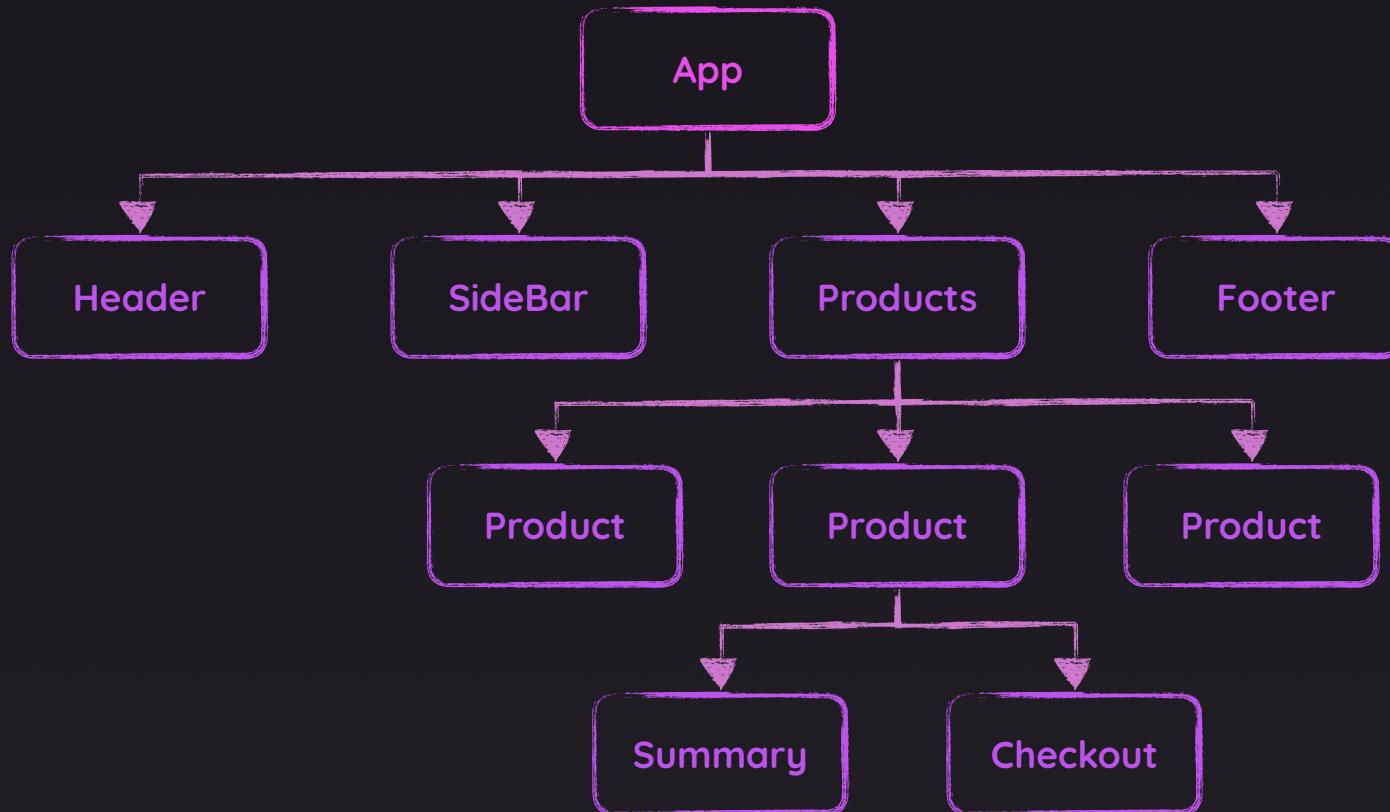
How Components Get Rendered



Building a Component Hierarchy



Building a Component Hierarchy



From Component Tree To DOM Node Tree

```
<div>
  <header>
    
    <h1>React Essentials</h1>
    <p>Learn about all the core React concepts!</p>
  </header>
  <main>
    <p>Time to get started!</p>
  </main>
</div>
```

Built-in Components vs Custom Components



Built-in Components

Name starts with a **lowercase** character

Only **valid, officially defined** HTML elements are allowed

Are **rendered as DOM nodes** by React (i.e., displayed on the screen)



Custom Components

Name starts with **uppercase** character

Defined by you, “wraps” built-in or other custom components

React “**traverses**” the component tree until it has only built-in components left

Outputting Dynamic Content in JSX



Static Content

Content that's **hardcoded** into the JSX code

Can't change at runtime

Example

```
<h1>Hello World!</h1>
```



Dynamic Content

Logic that **produces the actual value** is added to JSX

Content / value is **derived** at runtime

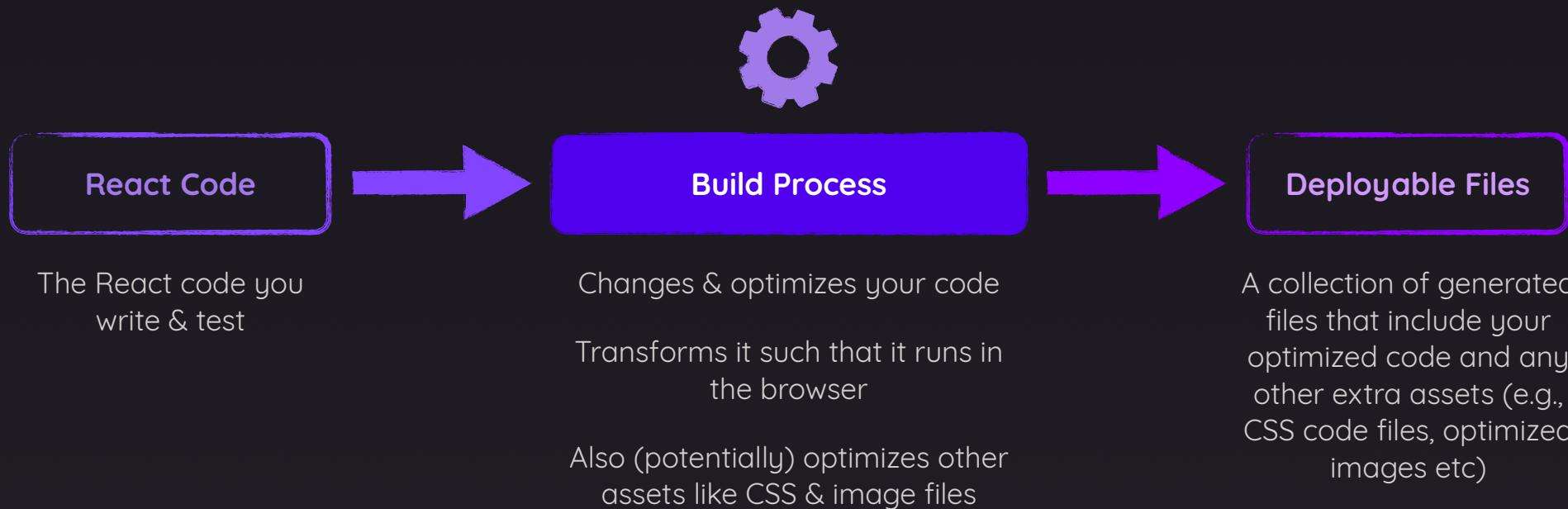
Example

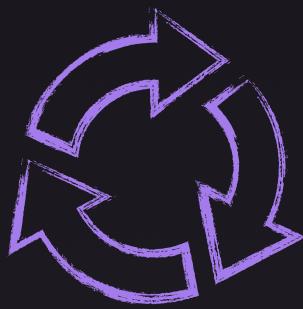
```
<h1>{user_name}</h1>
```



React Projects & “The Build Process”

React projects must be “built” (via a build process) before deployment

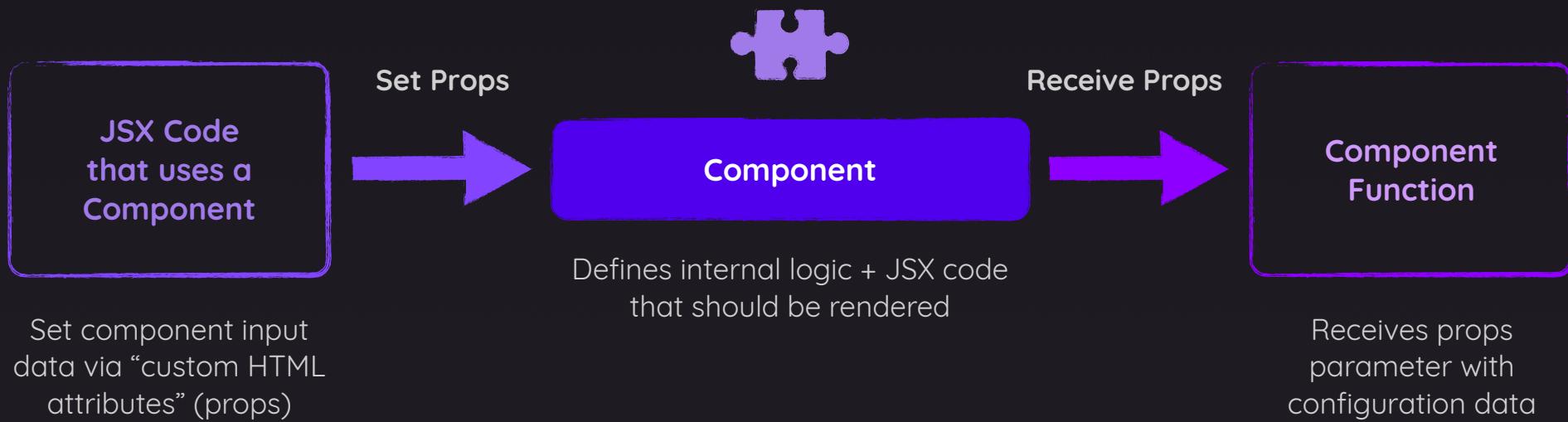




You Can Reuse React Components
But You Don't Have To!

Configuring Components With “Props”

React allows you to **pass data to components** via a concept called **“Props”**



Props Accept All Value Types

You're not limited to text values!

String value

Number value

Curly braces are required to pass the value as a number.

If quotes were used, it would be considered a string.

```
<UserInfo
  name="Max"
  age={34}
  details={{userName: 'Max'}}
  hobbies={['Cooking', 'Reading']}
/>
```

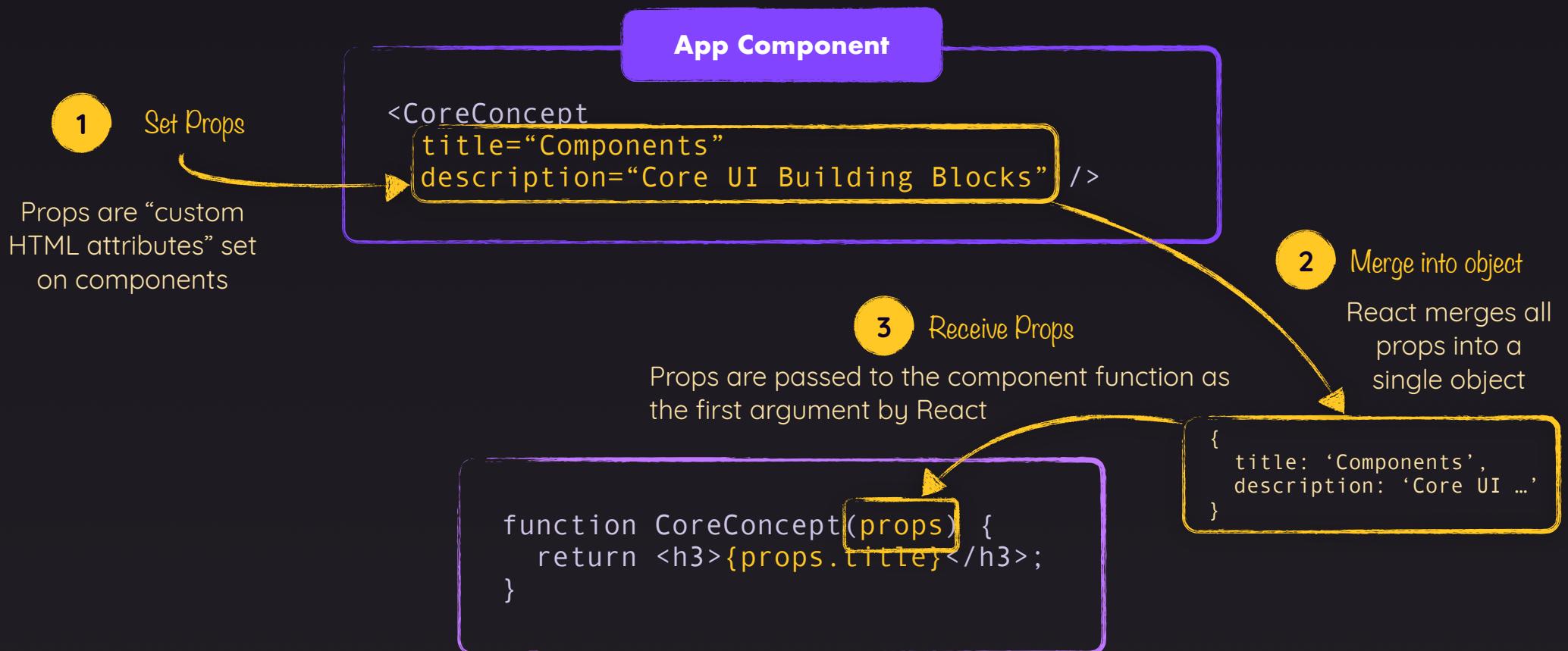
Object value

This is **no special “double curly braces” syntax!**

It's simply a JS object passed as a value.

Array value

Understanding Props



The Special “children” Prop

The “children” Prop

React automatically passes a special prop named “children” to every custom component

```
<Modal>
  <h2>Warning</h2>
  <p>Do you want to delete this file?</p>
</Modal>
```

App Component

Content for “children”

The content between component opening and closing tags is used as a value for the special “children” prop

Modal Component

```
function Modal(props) {
  return <div id="modal">{props.children}</div>;
}
```

props.children



“children” Prop vs “Attribute Props”

Using “children”

```
<TabButton>Components</TabButton>
```

```
function TabButton({ children }) {  
  return <button>{children}</button>;  
}
```

Using Attributes

```
<TabButton label="Components"></TabButton>
```

```
function TabButton({ label }) {  
  return <button>{label}</button>;  
}
```

For components that take a **single piece of renderable content**, this approach is closer to “normal HTML usage”

This approach is **especially convenient** when passing **JSX code as a value** to another component

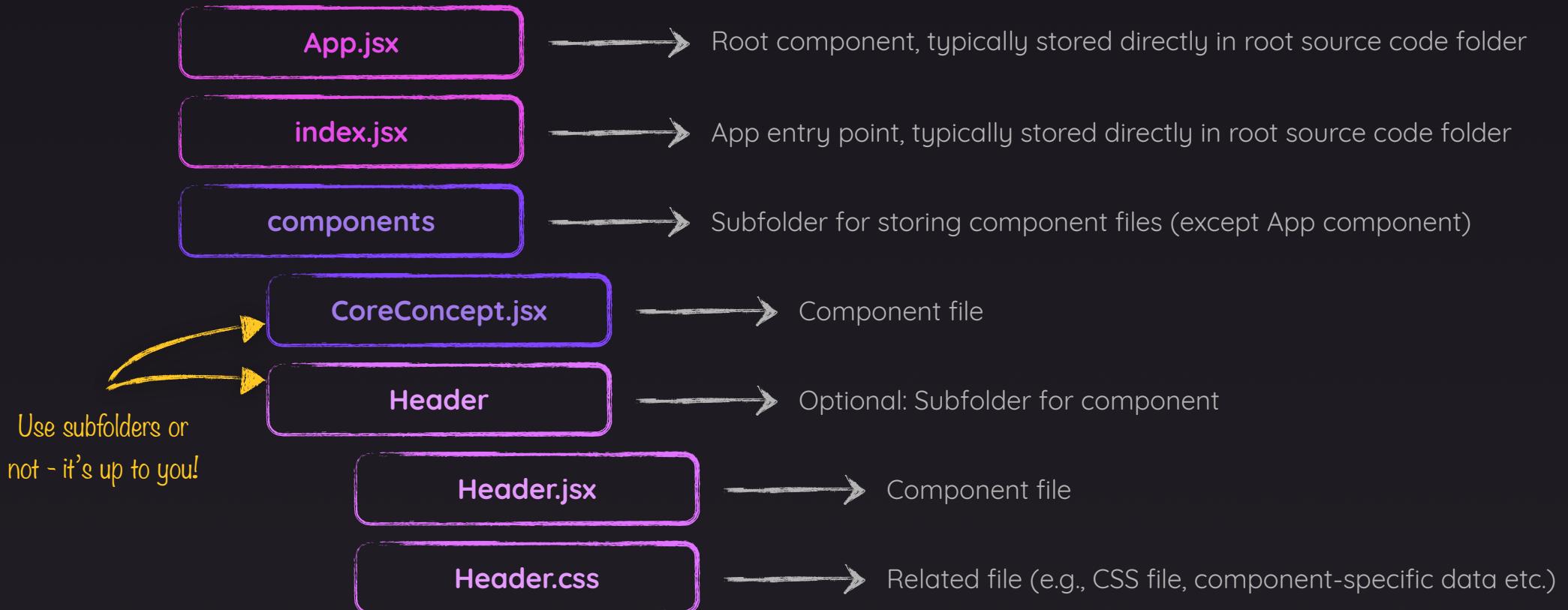
This approach makes sense if you got **multiple smaller pieces of information** that must be passed to a component

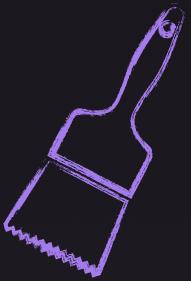
Adding **extra props** instead of just wrapping the content with the component tags mean **extra work**

Ultimately, it comes down to your use-case and personal preferences.



Structuring React Projects - The “src” Folder





By Default, React Components Execute Only Once

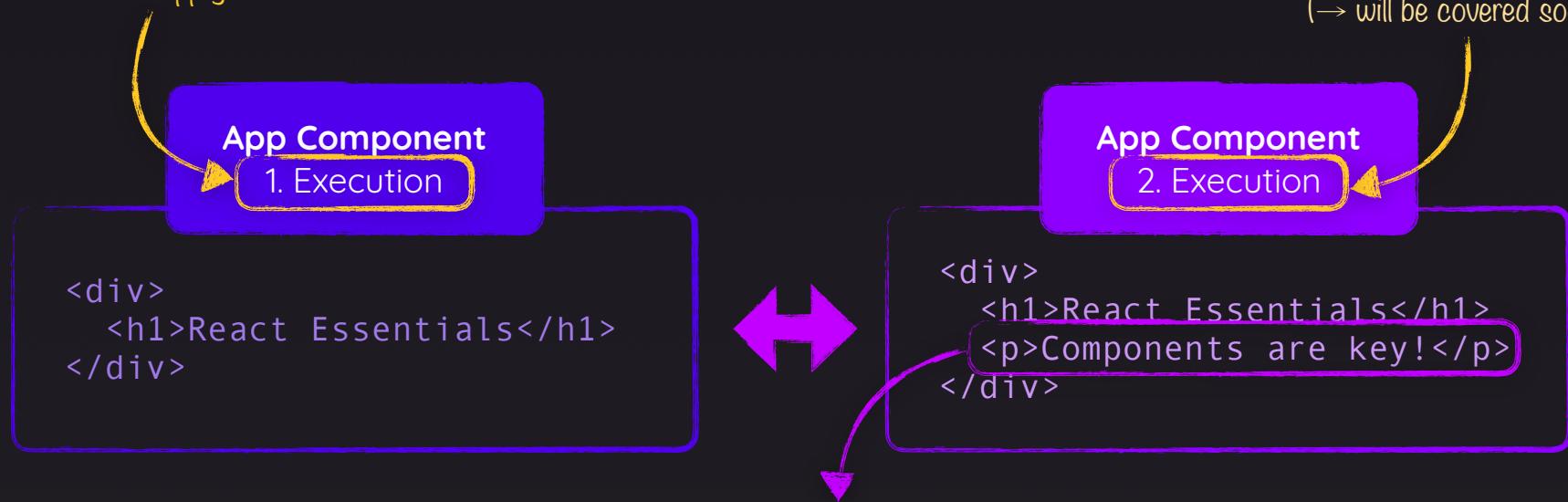
**You have to “tell” React If
A Component Should be
Executed Again**

How React Checks If UI Updates Are Needed

React **compares** the **old output** ("old JSX code") of your component function to the **new output** ("new JSX code") and **applies any differences** to the actual website UI

Because web app got loaded

Because of a state update
(→ will be covered soon)



Identified difference

Necessary updates will be applied to the real DOM,
ensuring that the visible UI matches the expected output.



useState() Yields An Array With Two Elements

And it will **always** be **exactly** two elements

```
const stateArray = useState('Please click a button');
```



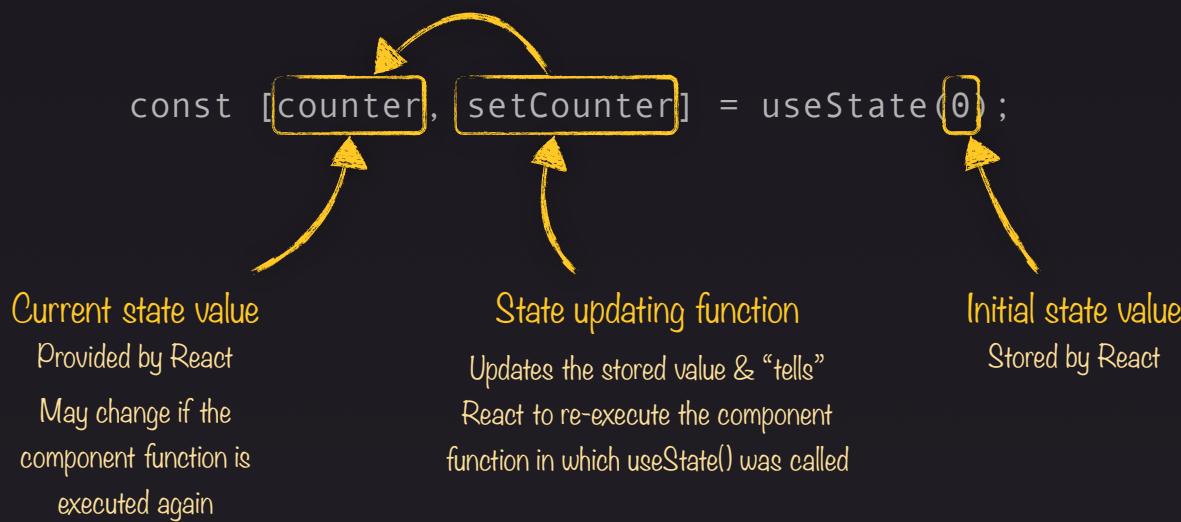
Array produced and returned by React's useState() function

Contains exactly two elements

Manage State

Manage data & “tell” React to **re-execute** a component function via React’s **useState()** Hook

State updates lead to new state values
(as the component function executes again)



Rules of Hooks

1

Only call Hooks inside of Component Functions

React Hooks must not be called outside of React component functions



```
function App() {  
  const [val, setVal] = useState(0);  
}
```



```
const [val, setVal] = useState(0);  
  
function App() { ... }
```

2

Only call Hooks on the top level

React Hooks must not be called in nested code statements (e.g., inside of if-statements)



```
function App() {  
  const [val, setVal] = useState(0);  
}
```



```
function App() {  
  if (someCondition)  
    const [val, setVal] = useState(0);  
}
```

Transforming Data To JSX

```
[  
  { userName: 'Max', age: 34 },  
  { userName: 'Manuel', age: 35 },  
  { userName: 'Marie', age: 38 },  
]
```

Transform

```
[  
  <User {...user[0]} />,  
  <User {...user[1]} />,  
  <User {...user[2]} />,  
]
```

```
<ul>  
  <User {...user[0]} />  
  <User {...user[1]} />  
  <User {...user[2]} />  
</ul>
```

Output in JSX

Essential React Concepts



Components

Reusable building blocks which are used to build the overall app UI



Props

Component “attributes” (input data) used to configure components



JSX

JS syntax extension to describe HTML in JavaScript



State

React-managed data which, when changed, causes React to re-execute the related component function



React Essentials - Deep Dive

Beyond The Basics

- ▶ Behind The Scenes of **JSX**
- ▶ Structuring **Components** and **State**
- ▶ **Advanced State** Usage
- ▶ **Patterns & Best Practices**

You Don't Need JSX (But It's Convenient)

```
<div id="content">  
  <p>Hello World!</p>  
</div>
```

Requires build process &
code transformation

Easy to read & understand

```
React.createElement(  
  'div',  
  { id: 'content' },  
  React.createElement(  
    'p',  
    null,  
    'Hello World'  
)  
)
```

Works without special build
process & transformation

Pretty verbose & not
necessarily intuitive

Component Type

Identifies the to-be-
rendered component

Props Object

Sets component
props

Child Content

The content
passed between
the component
tags

Additional Key Component & Props Concepts



Forwarded Props



Multiple Component Slots

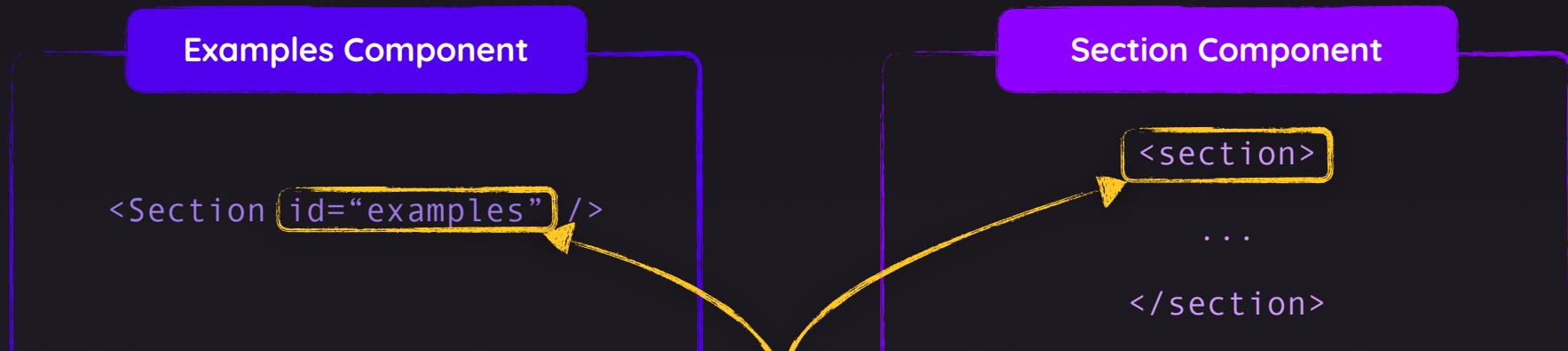


Element Identifiers as Props



Default Prop Values

Props Are Not Forwarded Automatically



"id" is ignored

Props must be used & set explicitly





New Project, New Concepts



Multiple State Values



Nested Lists



Lifting State Up



Array & Object States



Derived State



**Component Functions vs
“Normal Functions”**



Updating State Based On Old State



```
setIsEditing(!isEditing);
```

If your **new state depends on your previous state** value, you should **not** update the state like this



```
setIsEditing(wasEditing => !wasEditing);
```

Instead, **pass a function** to your state updating function

This function will **automatically be called** by React and will receive the **guaranteed latest state value**



React Is Scheduling State Updates

Somewhere in your code

Code snippets in different places in your app's code

```
setIsEditing(true); -----> 1 → Update state to true  
...  
setIsEditing(false);-----> 2 → Update state to false  
...  
setIsEditing(true); -----> 3 → Update state to true  
...  
setIsEditing(false);-----> 4 → Update state to false
```

React's State Updating Schedule

Managed behind the scenes by React

State updates are **not performed instantly** but at some point in the future
(when React has time for it)

In most cases, those state updates of course still are executed **almost instantly**

Update Object-State Immutable



Objects & arrays (which technically are objects) are reference values in JavaScript



NOT creating a copy
(because user is an object = a
reference value)



```
const updatedUser = user;  
updatedUser.name = 'Max'
```

Editing the user object in memory

You should therefore **not mutate** them directly – instead
create a **(deep) copy** first!

Creating a copy via JavaScript's
“Spread” operator

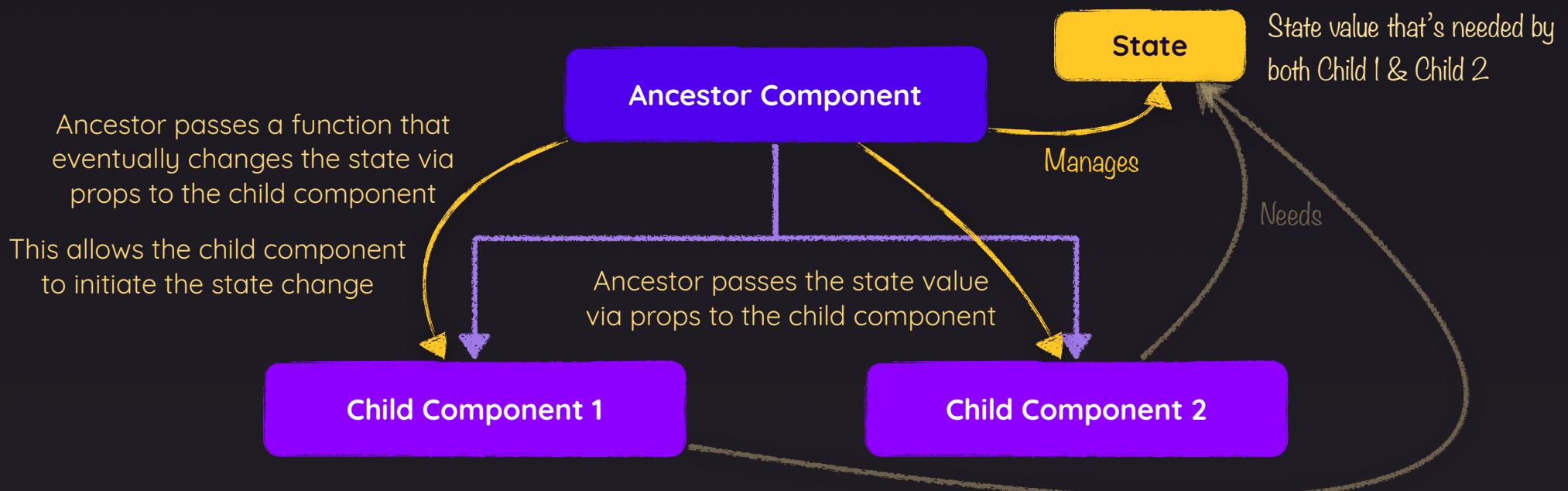


```
const updatedUser = { ...user }  
updatedUser.name = 'Max'
```

Editing the copy, not the original

Lifting State Up

Lift the state up to the **closest ancestor component** that has access to all components that need to work with that state



You Don't Always Have Just One State Value

Player data and game state must be managed



Combined State Object

Data from different app features is managed in a **single, shared state object**

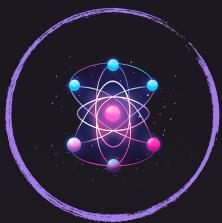
You must ensure that you don't **accidentally lose data** related to feature B when feature A's data changes



Separate State Slices

Data from different app features is managed in **different state values**

Suboptimal if states from different features depend on each other



React Essentials - Practice Project

Apply Your Knowledge & Practice What You Learned

- ▶ Build an “Investment Calculator” Web App
- ▶ Build, Configure & Combine **Components**
- ▶ Manage Application **State**
- ▶ Output **List** & **Conditional** Content

Time To Practice!



Your Task

Build an “Investment Calculator” web app

Use the **starting project** attached to this lecture

Add **components** for displaying a **header**, fetching **user input** & outputting the **results table**

Fetch & store user input (i.e., the entered investment parameters)

Derive investment results with help of the provided utility function (in the starting project)

Display investment results in a HTML **table** (use `<table>`, `<thead>`, `<tbody>` `<tr>`, `<th>`, `<td>`)

Conditionally display an info message if an **invalid duration** (< 1) was entered

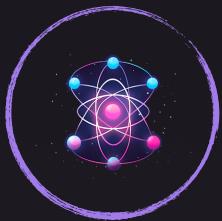


You Can't Fail!

There's More Than One Way Of Building This!

My solution (shown in the next lectures) is **not the only correct solution!**

And even if you can't build the entire app – simply **try to get as close as possible!**



Deploying React Projects

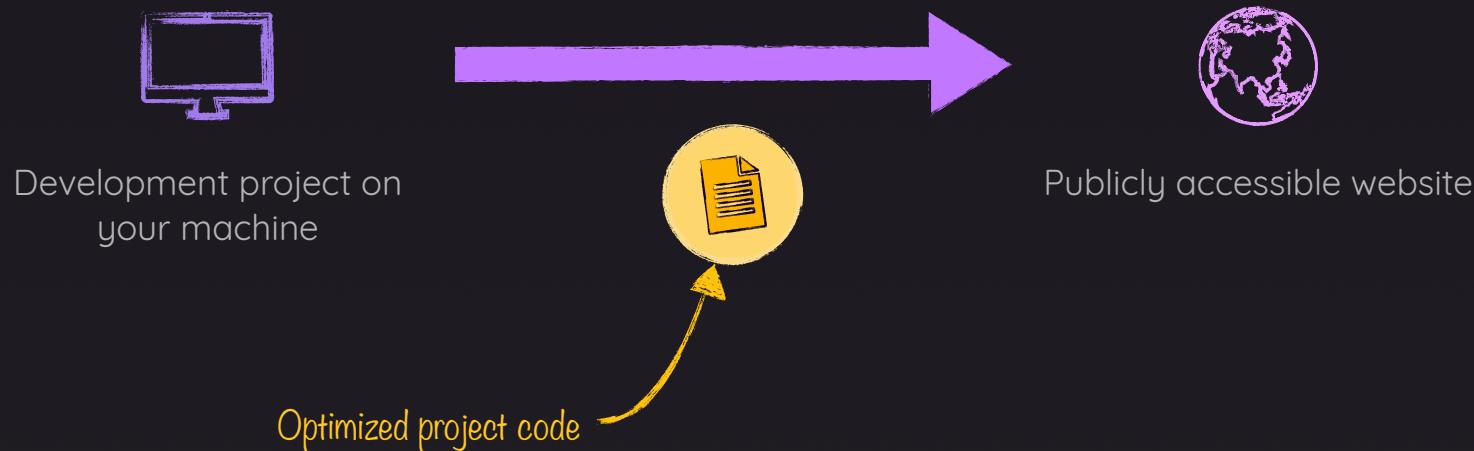
From Development To Production

- ▶ What Does “**Deployment**” Mean?
- ▶ React Project **Hosting Requirements**
- ▶ **Example** Deployment



What Does “Deployment” Mean?

Deployment: The process of **publishing** your project

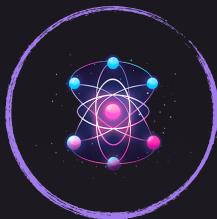




Debugging React Apps

Finding & Fixing Errors

- ▶ Making Sense of **React Error Messages**
- ▶ Finding **Logical Errors** via the Browser **DevTools & Debugger**
- ▶ Enabling React's **Strict Mode**
- ▶ Using the **React DevTools** for Application Analysis & Manipulation



Styling React Apps

Static & Dynamic Styling for Pretty Apps

- ▶ Styling with **Vanilla CSS**
- ▶ **Scoping** Styles with **CSS Modules**
- ▶ **CSS-in-JS** Styling with “**Styled Components**”
- ▶ Styling with **Tailwind CSS**
- ▶ Static & **Dynamic (Conditional)** Styling



This is not a CSS course

But you should know how to apply
CSS to your React components



Inline Styles: Advantages & Disadvantages



Advantages

Quick & easy to add to JSX

Styles only affect the element to which
you add them

Dynamic (conditional) styling is simple



Disadvantages

You need to know CSS

You need to style every element
individually

No separation between CSS & JSX code

Vanilla CSS: Advantages & Disadvantages



Advantages

CSS code is decoupled from JSX code

You write CSS code as you (maybe) know
and (maybe) love it

CSS code can be written by another
developer who needs only a minimal
amount of access to your JSX code



Disadvantages

You need to know CSS

CSS code is not scoped to components →
CSS rules may clash across components
(e.g., same CSS class name used in
different components for different
purposes)

CSS Modules

Vanilla CSS with file-specific scoping



CSS Modules: Advantages & Disadvantages



Advantages

CSS code is decoupled from JSX code

You write CSS code as you (maybe) know
and (maybe) love it

CSS code can be written by another
developer who needs only a minimal
amount of access to your JSX code

CSS classes are scoped to the component
(files) which import them → No CSS class
name clashes



Disadvantages

You need to know CSS

You may end up with many relatively
small CSS files in your project

Styled Components: Advantages & Disadvantages



Advantages

- Quick & easy to add
- You continue “thinking in React” (→ configurable style functions)
- Styles are scoped to components → No CSS rule clashes



Disadvantages

- You need to know CSS
- No clear separation of React & CSS code
- You end up with many relatively small “wrapper” components

Tailwind CSS: Advantages & Disadvantages



Advantages

You don't need to know (a lot about) CSS

Rapid development

No style clashes between components
since you don't define any CSS rules

Highly configurable & extensible

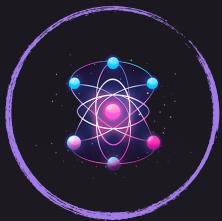


Disadvantages

Relatively long className values

Any style changes require editing JSX

You end up with many relatively small
“wrapper” components OR lots of copy &
pasting



Refs & Portals

Advanced DOM Access & Value Management

- ▶ Accessing **DOM Elements** with **Refs**
- ▶ **Managing Values** with Refs
- ▶ **Exposing API Functions** from Components
- ▶ **Detaching DOM Rendering** from JSX Structure with **Portals**

State vs Refs



State

Causes component re-evaluation (re-execution) when changed

Should be used for values that are directly reflected in the UI

Should not be used for “behind the scenes” values that have no direct UI impact



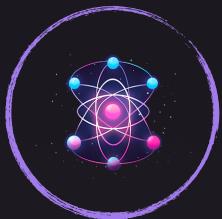
Refs

Do not cause component re-evaluation when changed

Can be used to gain direct DOM element access (→ great for reading values or accessing certain browser APIs)

Can be used to expose API functions from your components

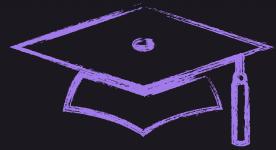
Should not be used for managing values that have a direct UI impact



Practice Project: Advanced Concepts

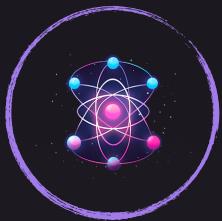
Working with Components, State, Styling, Refs & Portals

- ▶ Build a “Project Management” Web App
- ▶ Build, Style, Configure & Re-use **Components**
- ▶ Manage **State**
- ▶ Access DOM Elements & Browser APIs with **Refs**
- ▶ Manage JSX Rendering Positions with **Portals**



Challenge Time!

Try building this project on your own – or at least try to get as far as possible



Advanced State Management

Beyond Basic Apps & “Lifting Up State”

- ▶ The Problem Of Shared State: Prop Drilling
- ▶ Embracing **Component Composition**
- ▶ Sharing State with **Context**
- ▶ Managing Complex State with **Reducers**



Most React Apps Consist Of Multiple Components

App

Header

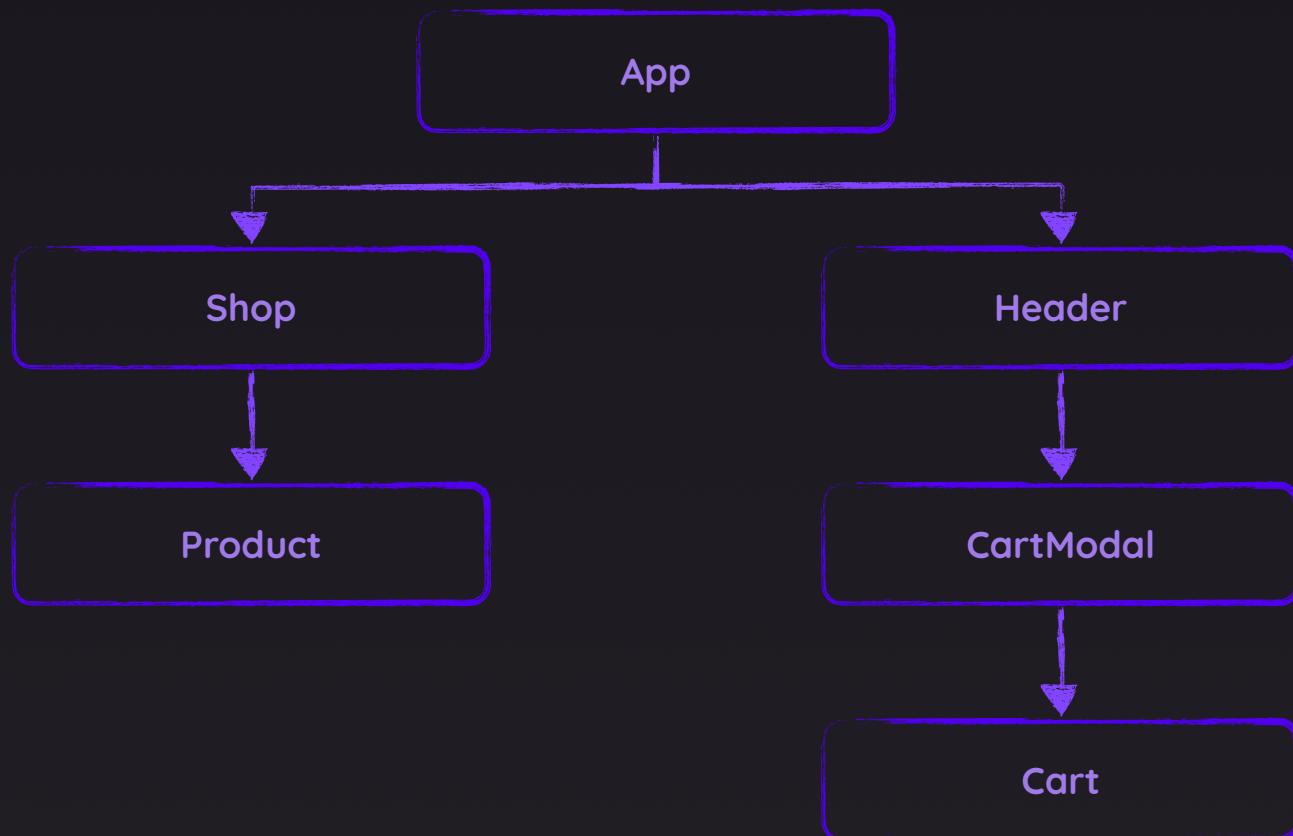
CartModal

Shop

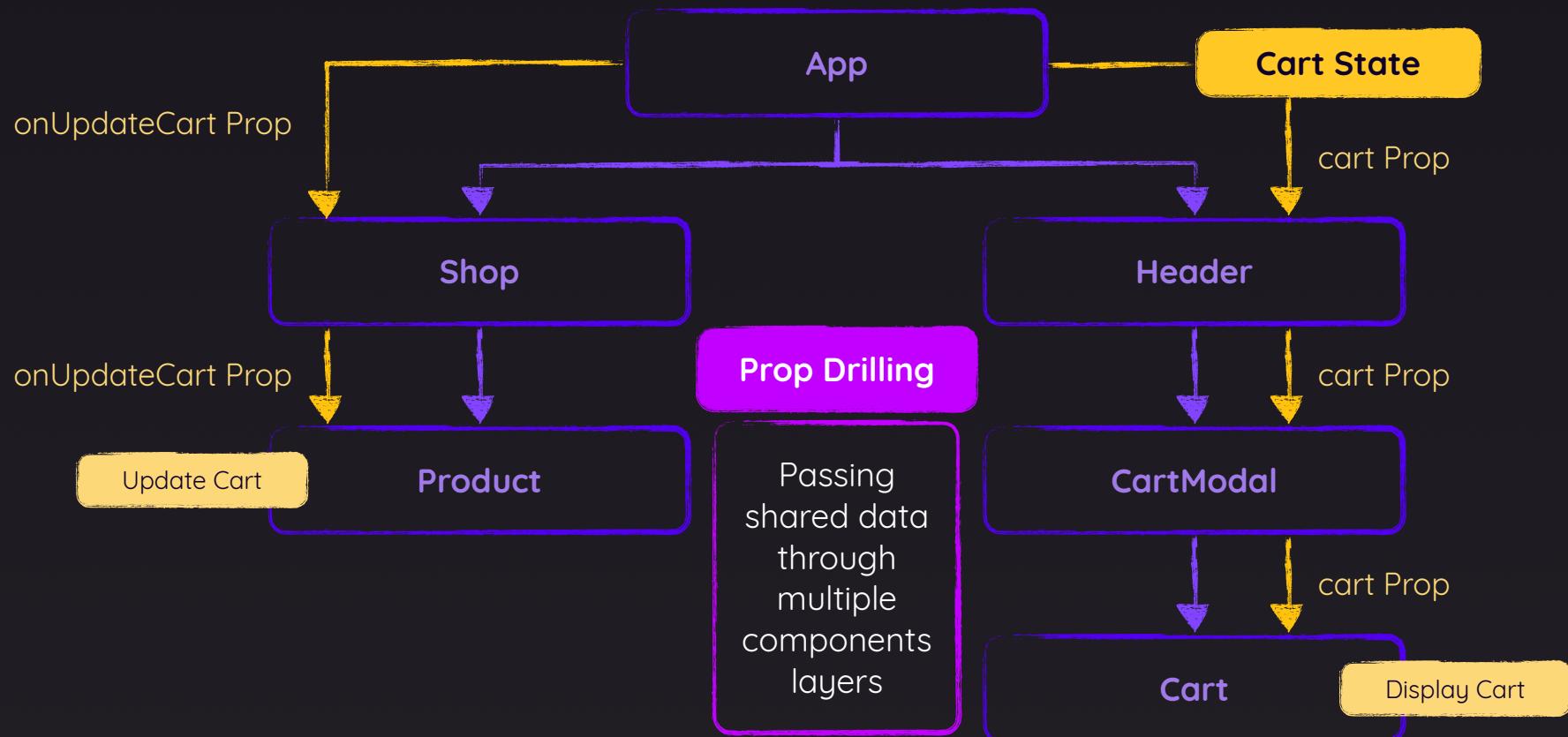
Product

Cart

Most React Apps Consist Of Multiple Components



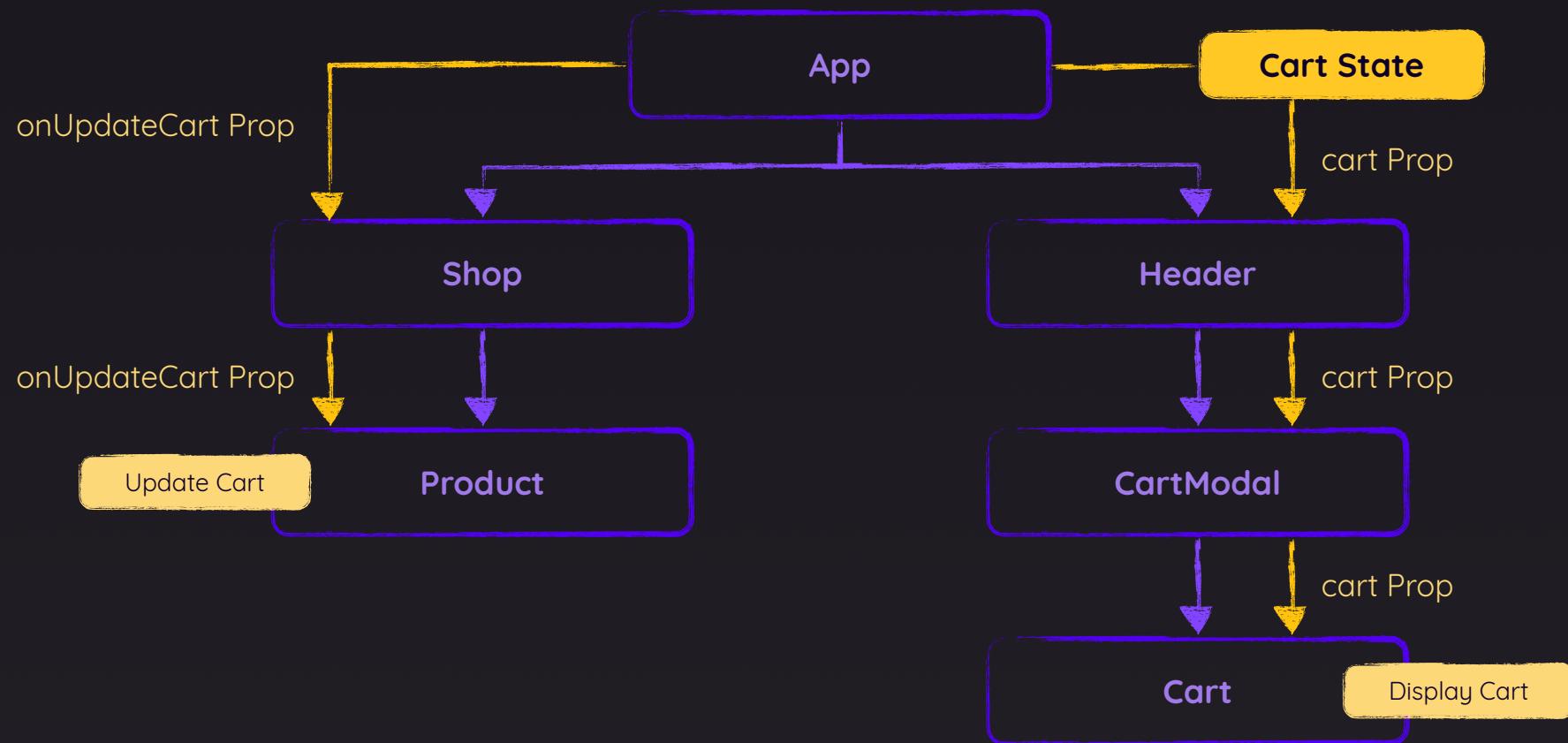
Most React Apps Consist Of Multiple Components



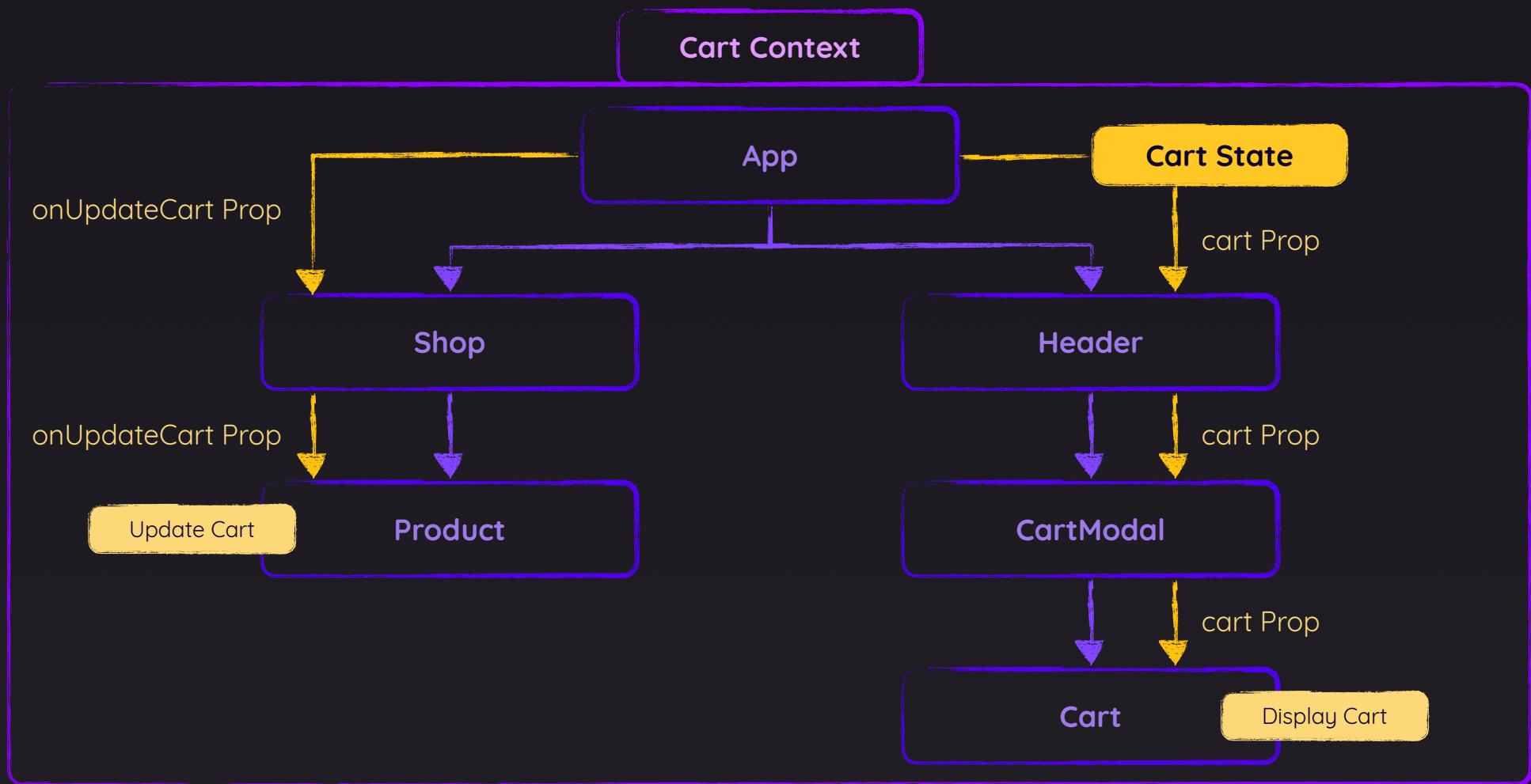
Component Composition

React's Context API

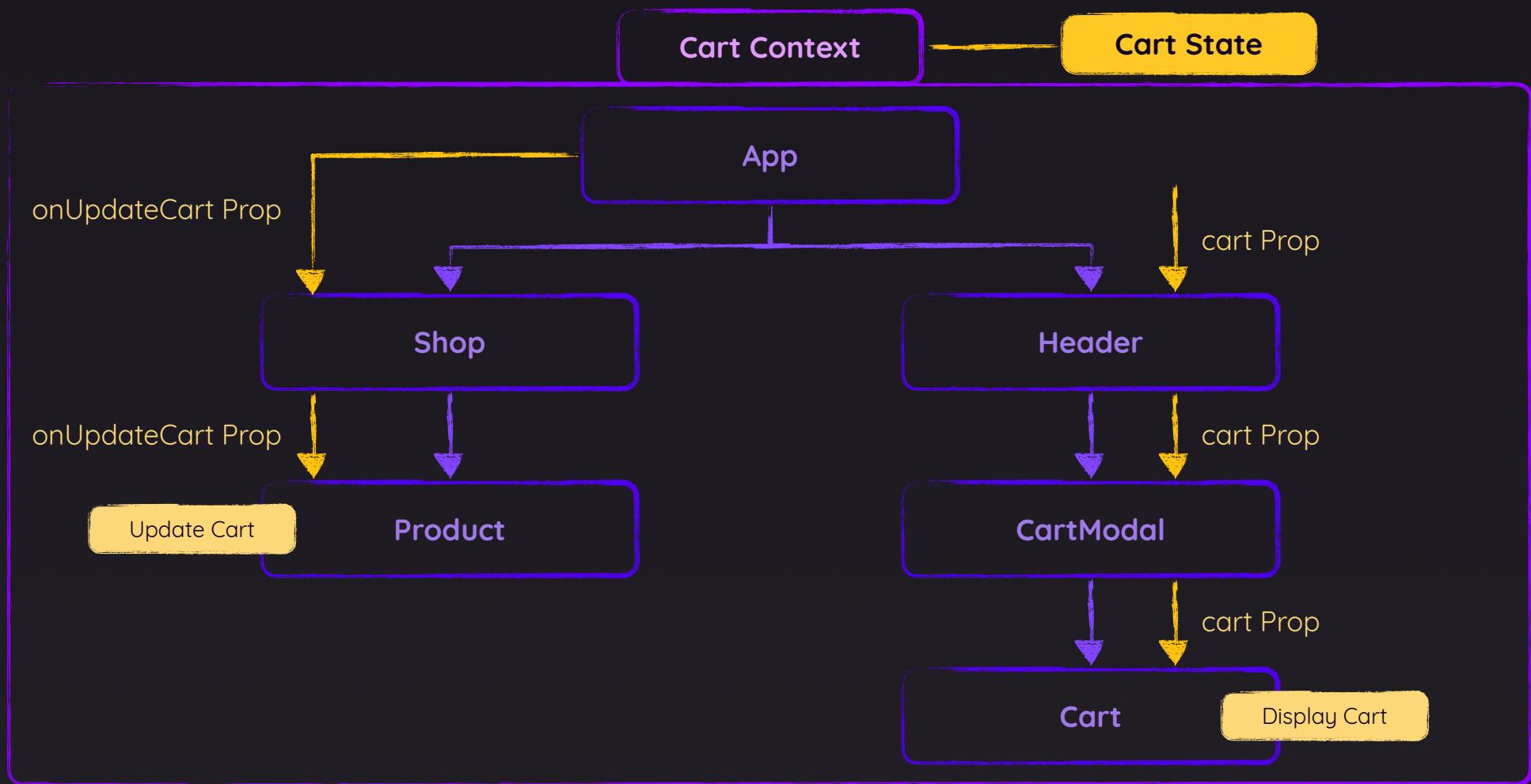
Most React Apps Consist Of Multiple Components



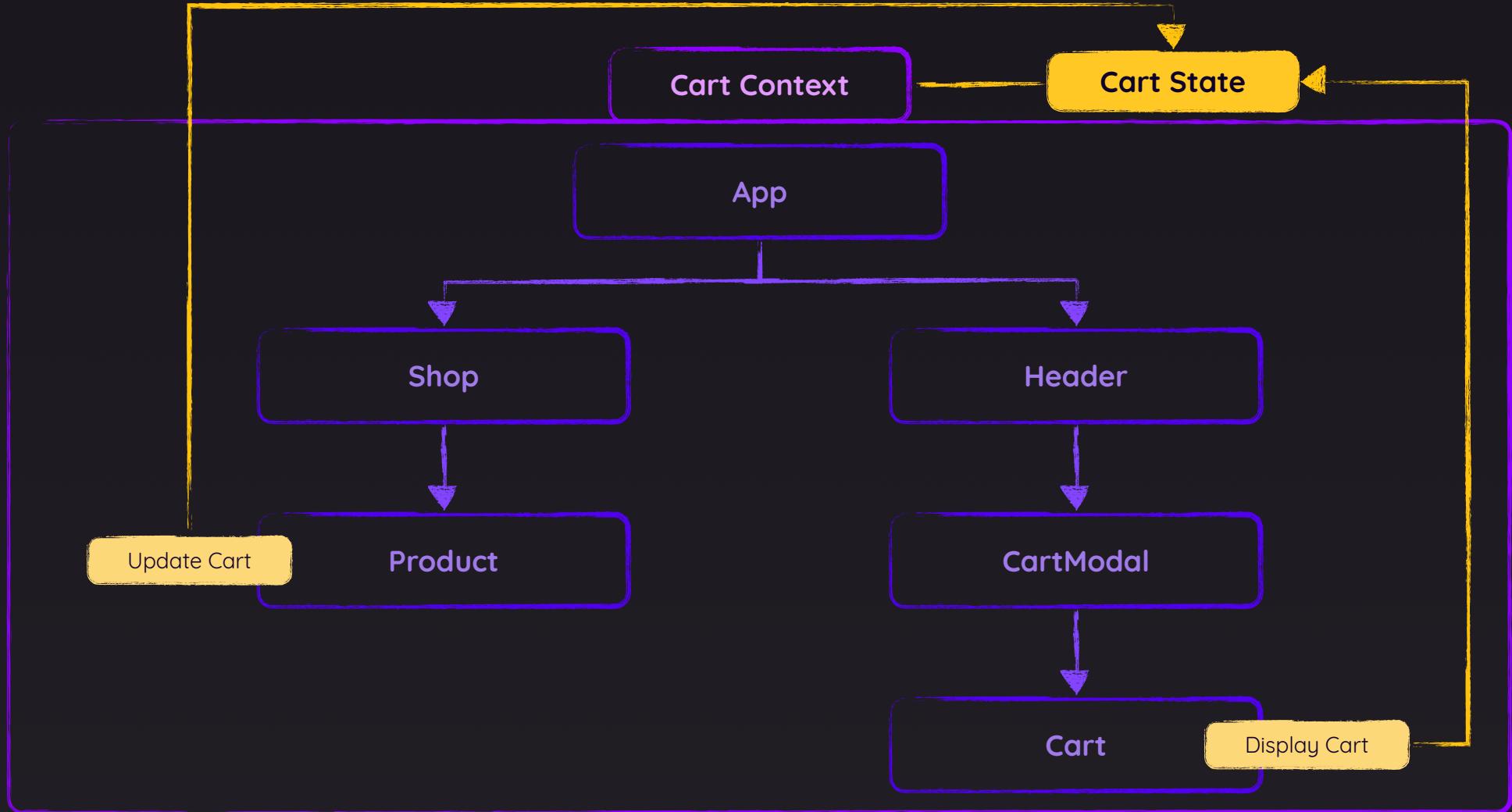
Most React Apps Consist Of Multiple Components



Most React Apps Consist Of Multiple Components

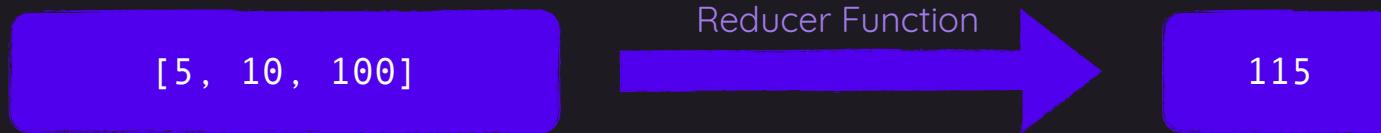


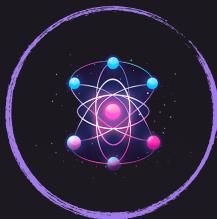
Most React Apps Consist Of Multiple Components



What's a “Reducer”?

A function that reduce one or more **complex values** to a **simpler one**





Dealing with Side Effects

Keeping the UI Synchronized

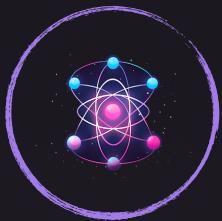
- ▶ Understanding **Side Effects & useEffect()**
- ▶ Effects & **Dependencies**
- ▶ **When NOT** to use useEffect()

What are “Side Effects”?

**Side effects are “tasks”
that don’t impact the
current component render
cycle**

You Might Not Need `useEffect()`!

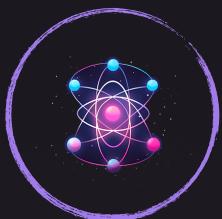
Use it as a last resort



Working with Effects

Practice & Dive Deeper

- ▶ Apply Your Knowledge
- ▶ Dealing with Effect **Dependencies & Cleanup**
- ▶ Combining Effects with Other React Concepts



Behind The Scenes

Understanding & Optimizing React

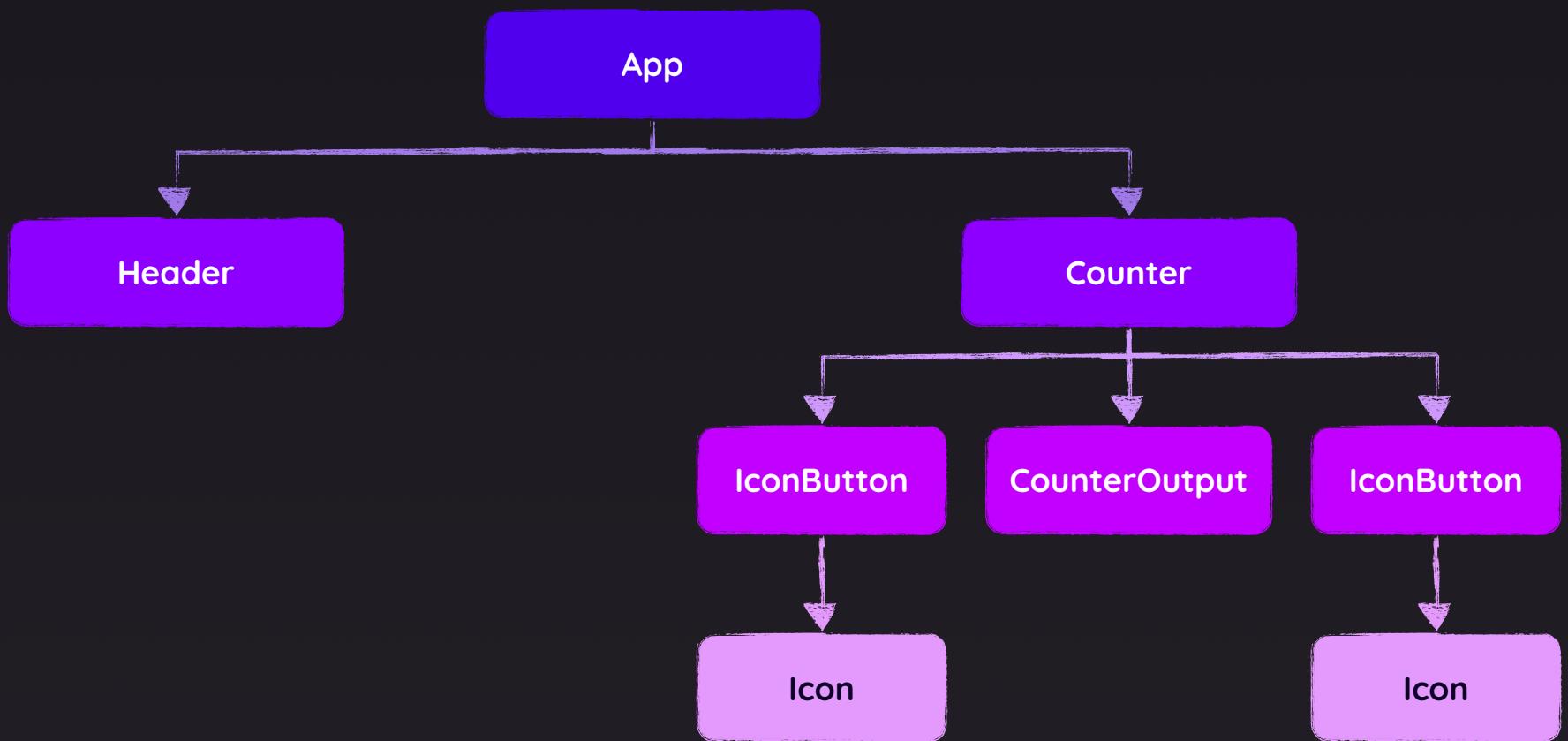
- ▶ How React **Updates The DOM**
- ▶ **Avoiding** Unnecessary Updates
- ▶ A Closer Look At **Keys**
- ▶ State **Scheduling** & State **Batching**

How Does React Update The DOM?

And how are component
functions executed?

React Builds A Component Tree

The **relation between components** is internally modelled as a tree structure



memo() compares prop values



Don't overuse memo()!

Use it **as high up in the component tree as possible**

→ blocking a component execution there will also block all child component executions

Checking props with memo() **costs performance!**

→ don't wrap it around all your components — it will just add a lot of unnecessary checks

Don't use it on components where **props will change frequently**

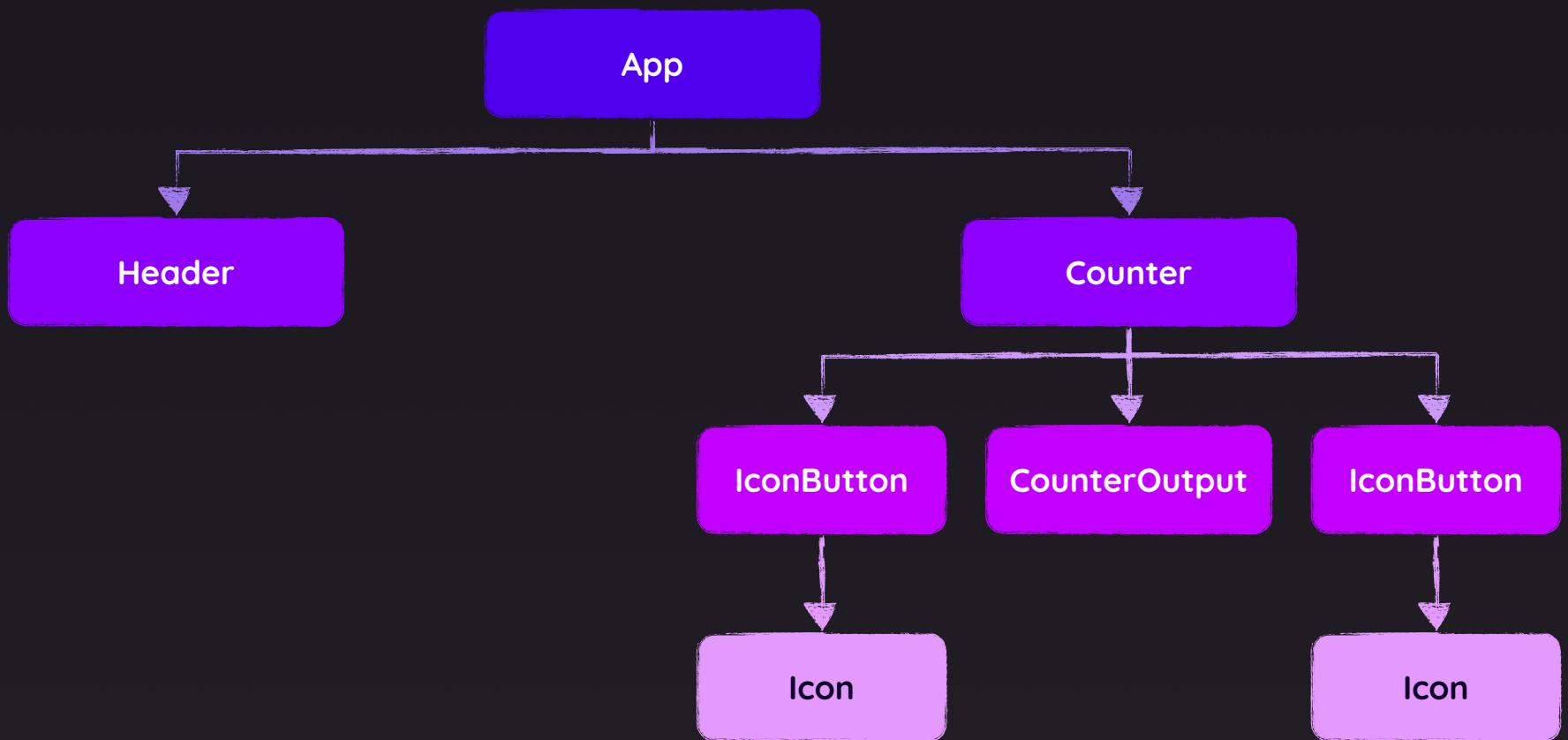
→ memo() would just perform a meaningless check in such cases (which costs performance)

React checks for necessary DOM updates via a “Virtual DOM”

It creates & compares virtual DOM snapshots to find out which parts of the rendered UI need to be updated

How React Updates The Website UI

Step 1: Creating a Component Tree



How React Updates The Website UI

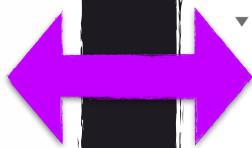
Step 2: Creating a Virtual Snapshot of the Target HTML Code

```
▼< <header id="main-header">
    
    <h1>React – Behind The Scenes</h1>
</header>
▼< <main>
    <section id="configure-counter"> flex
        <h2>Set Counter</h2>
        <input type="number" value="0">
        <button>Set</button>
    </section>
    <section class="counter">
        <p class="counter-info">
            "The initial counter value was "
            <strong>0</strong>
            ". It"
            ▶<strong>...</strong>
            " prime number."
        </p>
        <p> flex
            ▶<button class="button">...</button> flex
            <span class="counter-output">1</span>
            ▶<button class="button">...</button> flex
        </p>
    </section>
</main>
```

How React Updates The Website UI

Step 3: Compare New Virtual DOM Snapshot to Previous (Old) Virtual DOM Snapshot

Old Snapshot



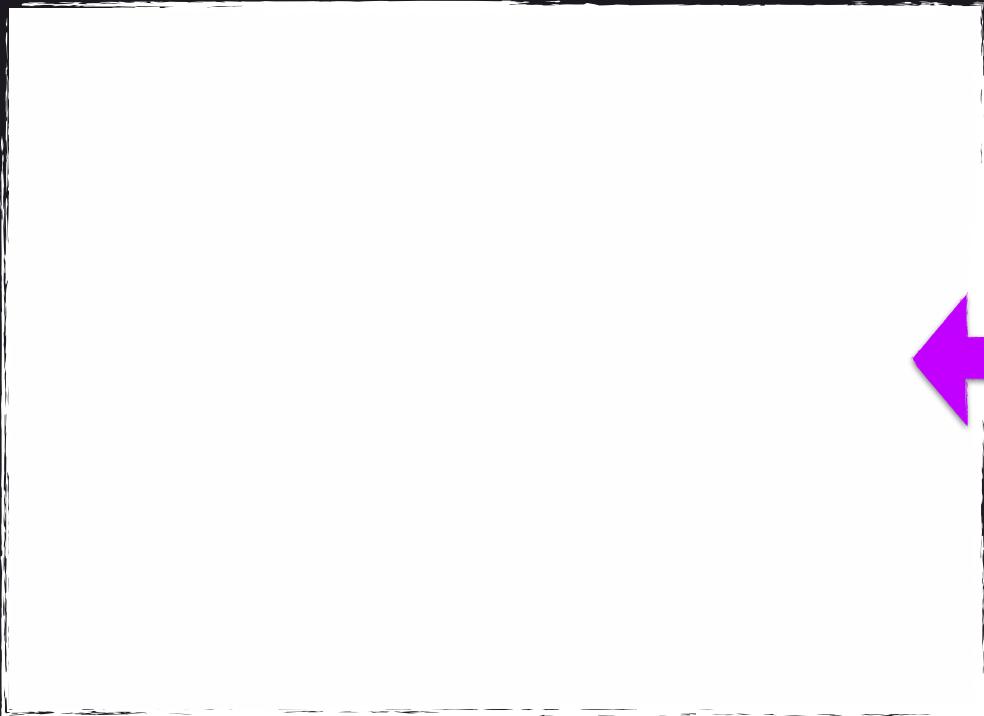
New Snapshot

```
<header id="main-header">
  
  <h1>React - Behind The Scenes</h1>
</header>
<main>
  <section id="configure-counter"> (flex
    <h2>Set Counter</h2>
    <input type="number" value="0">
    <button>Set</button>
  </section>
  <section class="counter">
    <p class="counter-info">
      "The initial counter value was "
      <strong>0</strong>
      ". It"
      ><strong>...</strong>
      " prime number."
    </p>
    <p> (flex
      ><button class="button">...</button> (flex
        <span class="counter-output">0</span>
      ><button class="button">...</button> (flex
    </p>
  </section>
</main>
```

How React Updates The Website UI

Step 4: Identify & Apply Changes to the “Real DOM”

Old Snapshot

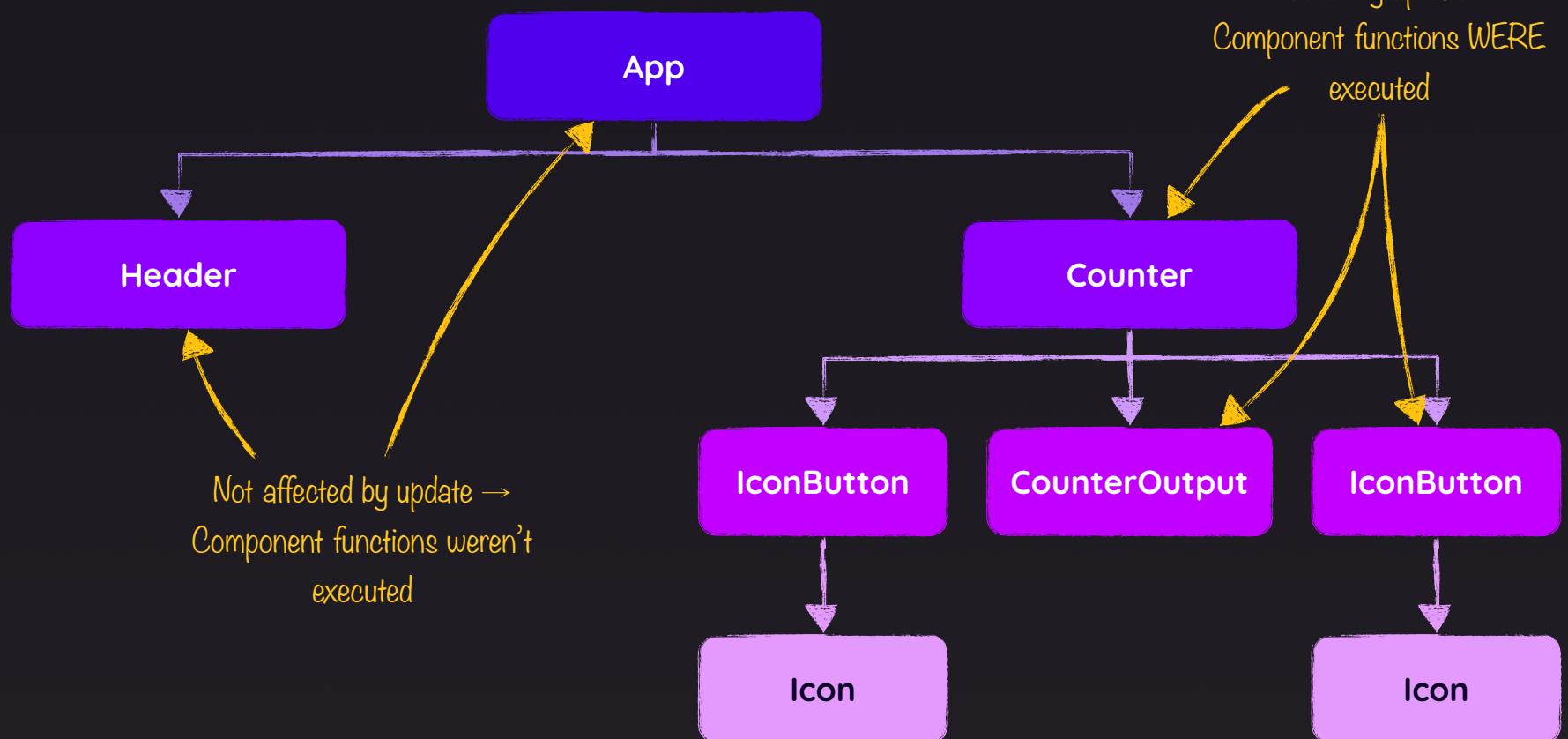


New Snapshot

```
<header id="main-header">
  
  <h1>React - Behind The Scenes</h1>
</header>
<main>
  <section id="configure-counter"> (flex)
    <h2>Set Counter</h2>
    <input type="number" value="0">
    <button>Set</button>
  </section>
  <section class="counter">
    <p class="counter-info">
      "The initial counter value was "
      <strong>0</strong>
      ". It"
      ><strong>...</strong>
      " prime number."
    </p>
    <p> (flex)
      ><button class="button">...</button> (flex)
        <span class="counter-output">0</span>
      ><button class="button">...</button> (flex)
    </p>
  </section>
</main>
```

Updating The Website UI After A Button Click

Step 1: Creating a Component Tree



How React Updates The Website UI

Step 2: Creating a Virtual Snapshot of the Target HTML Code

```
▼ <header id="main-header">
  
  <h1>React - Behind The Scenes</h1>
</header>
▼ <main>
  ▼ <section id="configure-counter"> flex
    <h2>Set Counter</h2>
    <input type="number" value="0">
    <button>Set</button>
  </section>
  ▼ <section class="counter">
    ▼ <p class="counter-info">
      "The initial counter value was "
      <strong>0</strong>
      ". It"
      ▶ <strong>...</strong>
      " prime number."
    </p>
    ▼ <p> flex
      ▶ <button class="button">...</button> flex
      <span class="counter-output">1</span>
      ▶ <button class="button">...</button> flex
    </p>
  </section>
</main>
```

How React Updates The Website UI

Step 3: Compare New Virtual DOM Snapshot to Previous (Old) Virtual DOM Snapshot

Old Snapshot

```
▼<header id="main-header">
  
  <h1>React - Behind The Scenes</h1>
</header>
▼<main>
  ▼<section id="configure-counter"> [flex]
    <h2>Set Counter</h2>
    <input type="number" value="0">
    <button>Set</button>
  </section>
  ▼<section class="counter">
    ▼<p class="counter-info">
      "The initial counter value was "
      <strong>0</strong>
      ". It"
      ▶<strong>...</strong>
      " prime number."
    </p>
    ▼<p> [flex]
      ▶<button class="button">...</button> [flex]
      <span class="counter-output">0</span>
      ▶<button class="button">...</button> [flex]
    </p>
  </section>
</main>
```

New Snapshot

```
▼<header id="main-header">
  
  <h1>React - Behind The Scenes</h1>
</header>
▼<main>
  ▼<section id="configure-counter"> [flex]
    <h2>Set Counter</h2>
    <input type="number" value="0">
    <button>Set</button>
  </section>
  ▼<section class="counter">
    ▼<p class="counter-info">
      "The initial counter value was "
      <strong>0</strong>
      ". It"
      ▶<strong>...</strong>
      " prime number."
    </p>
    ▼<p> [flex]
      ▶<button class="button">...</button> [flex]
      <span class="counter-output">1</span>
      ▶<button class="button">...</button> [flex]
    </p>
  </section>
</main>
```



How React Updates The Website UI

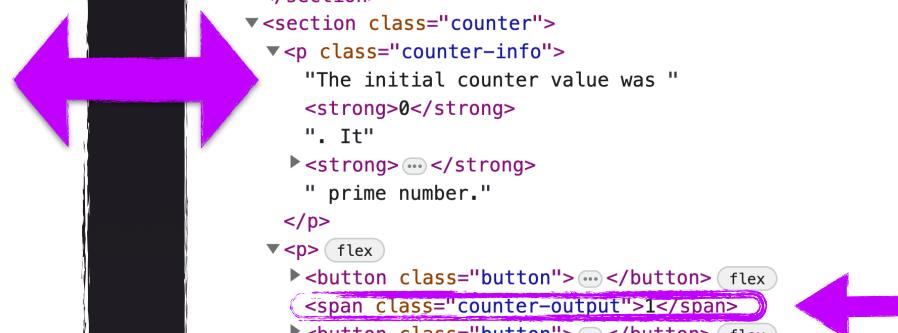
Step 4: Identify & Apply Changes to the “Real DOM”

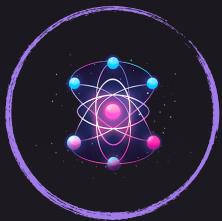
Old Snapshot

```
▼<header id="main-header">
  
  <h1>React - Behind The Scenes</h1>
</header>
▼<main>
  ▼<section id="configure-counter"> [flex]
    <h2>Set Counter</h2>
    <input type="number" value="0">
    <button>Set</button>
  </section>
  ▼<section class="counter">
    ▼<p class="counter-info">
      "The initial counter value was "
      <strong>0</strong>
      ". It"
      ▶<strong>...</strong>
      " prime number."
    </p>
    ▼<p> [flex]
      ▶<button class="button">...</button> [flex]
      <span class="counter-output">0</span>
      ▶<button class="button">...</button> [flex]
    </p>
  </section>
</main>
```

New Snapshot

```
▼<header id="main-header">
  
  <h1>React - Behind The Scenes</h1>
</header>
▼<main>
  ▼<section id="configure-counter"> [flex]
    <h2>Set Counter</h2>
    <input type="number" value="0">
    <button>Set</button>
  </section>
  ▼<section class="counter">
    ▼<p class="counter-info">
      "The initial counter value was "
      <strong>0</strong>
      ". It"
      ▶<strong>...</strong>
      " prime number."
    </p>
    ▼<p> [flex]
      ▶<button class="button">...</button> [flex]
      <span class="counter-output">1</span>
      ▶<button class="button">...</button> [flex]
    </p>
  </section>
</main>
```





Class-Based Components

An Alternative Way Of Building Components

- ▶ What & Why?
- ▶ Working with Class-based Components
- ▶ Error Boundaries

Class Components vs Functional Components

Functional Components

```
function Product(props) {  
  return <h2>A Product!</h2>  
}
```

Components are regular JavaScript functions which return renderable results (typically JSX)

The default & most modern approach!

Class-based Components

```
class Product extends Component {  
  render() {  
    return <h2>A Product!</h2>  
  }  
}
```

Components can also be defined as JS classes where a render() method defines the to-be-rendered output

Was required in the past

when using React prior to version 16.8



**Traditionally, you had to use
class-based components to
manage “State”**

React 16.8 introduced
“React Hooks” for
functional components

**Class-based components
can't use React Hooks**

Class Components Lifecycle

Side Effects in Functional Components: `useEffect()`



Class-based components can't use Hooks!

`componentDidMount()`

Called once a component **mounted**
→ evaluated & rendered by React

→ `useEffect(..., [])`

`componentDidUpdate()`

Called once a component **updated**
→ re-evaluated & re-rendered by React

→ `useEffect(..., [someValue])`

`componentWillUnmount()`

Called right before component is **unmounted**
→ right before removed from DOM

→ `useEffect(() => {
 return () => { ... }
}, [])`

You Don't Have To Use Functional Components!

You can use class-based components if you prefer them (though it's really not necessarily recommended...)



Which Component Type Should You Use?

Strong Recommendation: You should prefer **functional components!**

Use class-based if ...



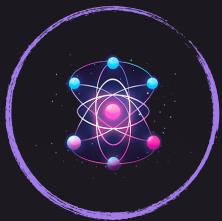
...you **prefer** them



...you're working on an existing
project or in a team where
they're getting used



...you're building an Error
Boundary



Understanding Redux

Managing App-Wide State with Redux

- ▶ **What** Is Redux? And **Why** Would You Use It?
- ▶ Redux **Basics** & **Using Redux** with React
- ▶ Working with **Redux Toolkit**

**A state management
system for cross-component
or app-wide state**

State

Data which, when
changed, should
affect the UI

State?

useState()!
or useReducer



What Is Cross-Component & App-Wide State?



Local State

State belongs to a single component

E.g., listening to user input on an input field or toggling a “show more details” field



Should be **managed inside the component** via `useState()` / `useReducer()`



Cross-component State

State affecting multiple components

E.g., open / closed state of a modal overlay



Requires “**prop drilling**”



App-wide State

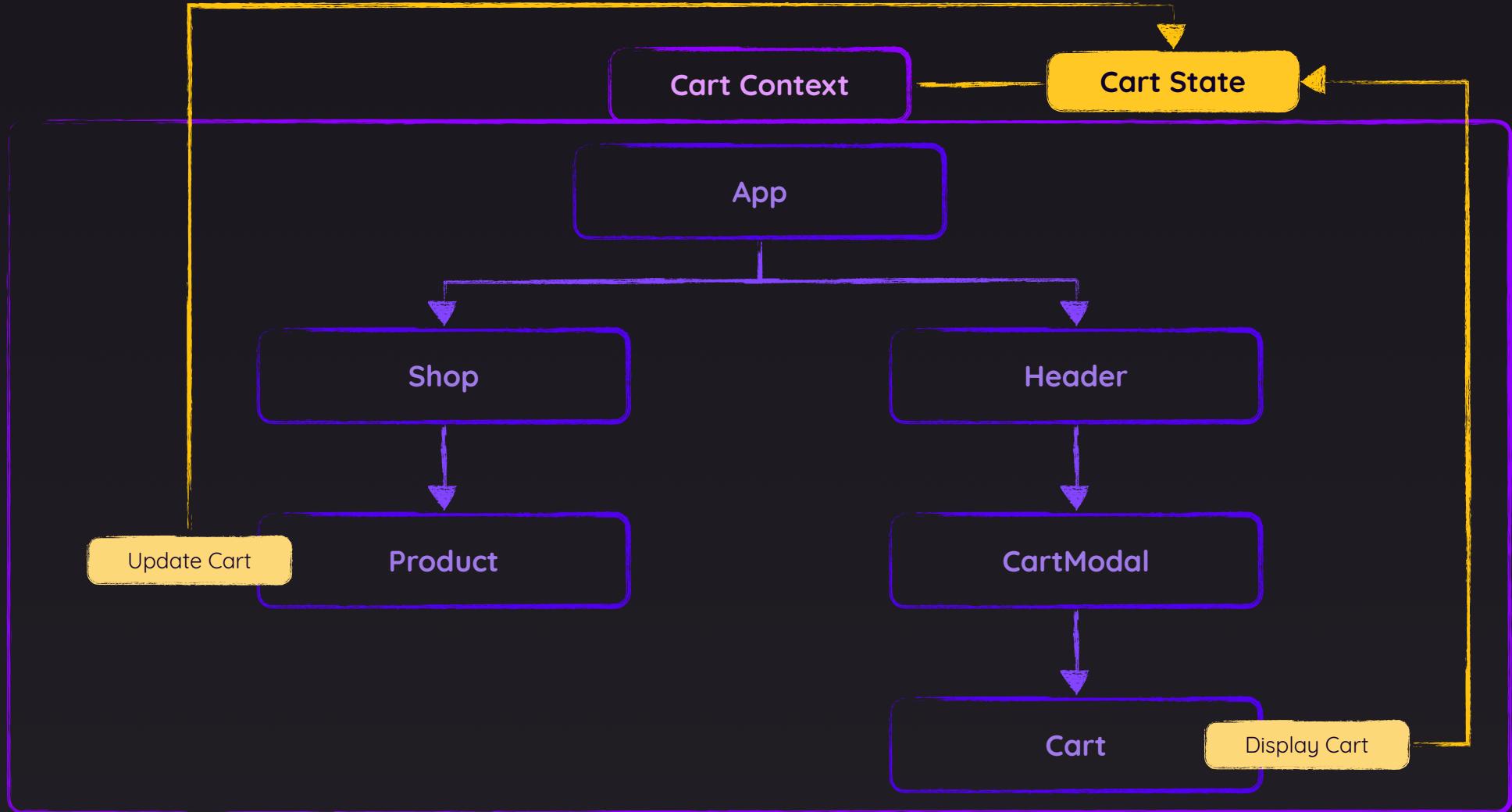
State affecting the entire app

E.g., user authentication status or chosen theme



Requires “**prop drilling**”
Or: **React Context** or **Redux**

React Context Manages Multi-Component State



**A state management
system for cross-component
or app-wide state**

A state management system for cross-component or app-wide state

Don't we have React Context already?

React Context Has Some Potential Disadvantages



Complex Setup & Management

In more complex apps, using React Context can lead to deeply nested or “fat” “Context Provider” components



Performance

React Context is not optimized for high-frequency state changes

Potential Problem: Deeply Nested Providers



```
return (
  <AuthProvider>
    <ThemeContextProvider>
      <UIInteractionContextProvider>
        <MultiStepFormContextProvider>
          <UserRegistration />
        </MultiStepFormContextProvider>
      </UIInteractionContextProvider>
    </ThemeContextProvider>
  </AuthProvider>
);
```



Potential Problem: Complex Providers

```
● ● ●

function AllContextProvider() {
  const [isAuth, setIsAuth] = useState(false);
  const [isEvaluatingAuth, setIsEvaluatingAuth] = useState(false);
  const [activeTheme, setActiveTheme] = useState('default');
  const [ ... ] = useState(...);

  function loginHandler(email, password) { ... };
  function signupHandler(email, password) { ... };
  function changeThemeHandler(newTheme) { ... };

  ...

  return (
    <AllContext.Provider>
      ...
    </AllContext.Provider>
  )
}
```



Potential Problem: High-Frequency Changes



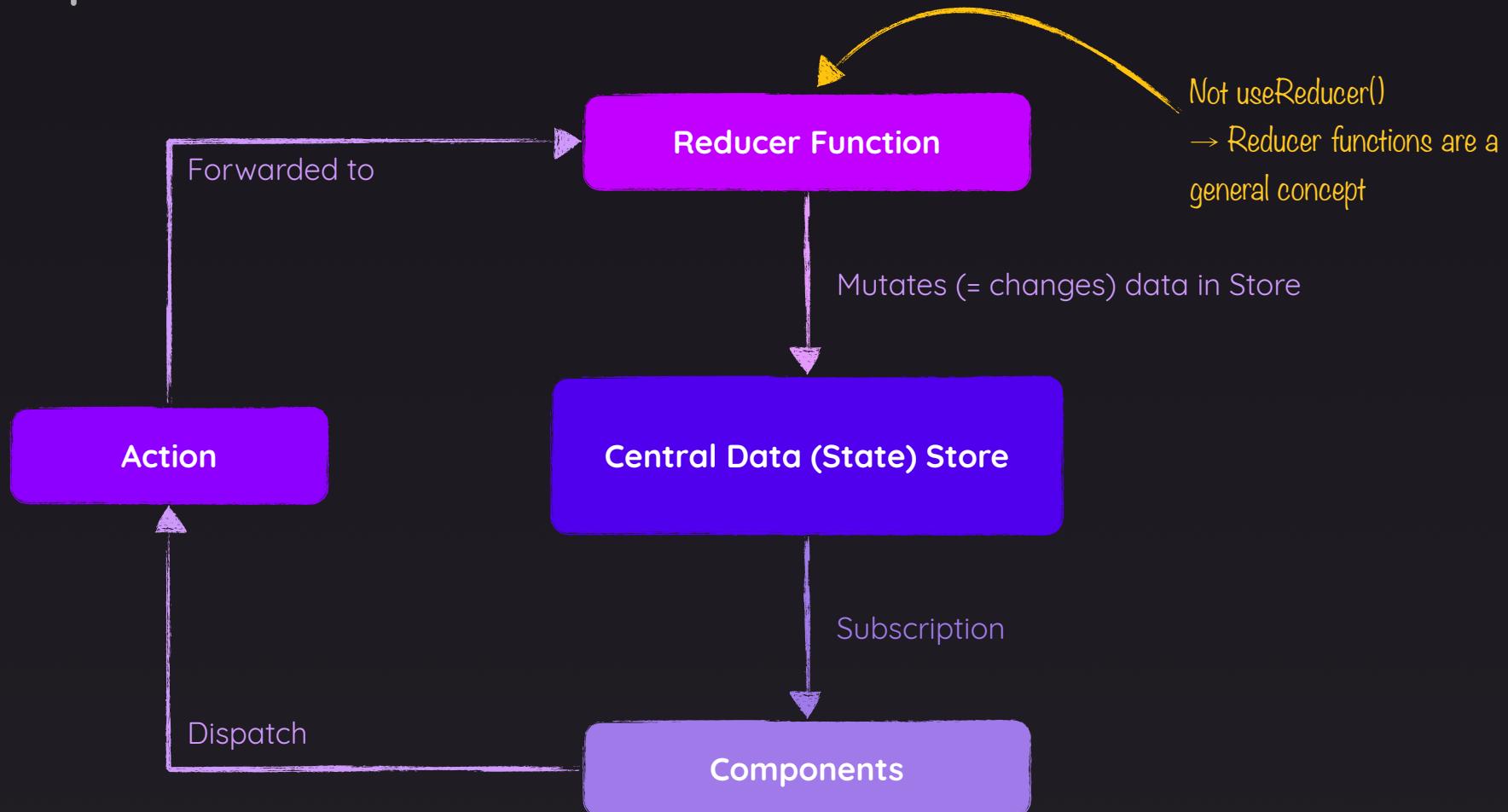
sebmarkbage commented on 18 Dec 2018

Member ...

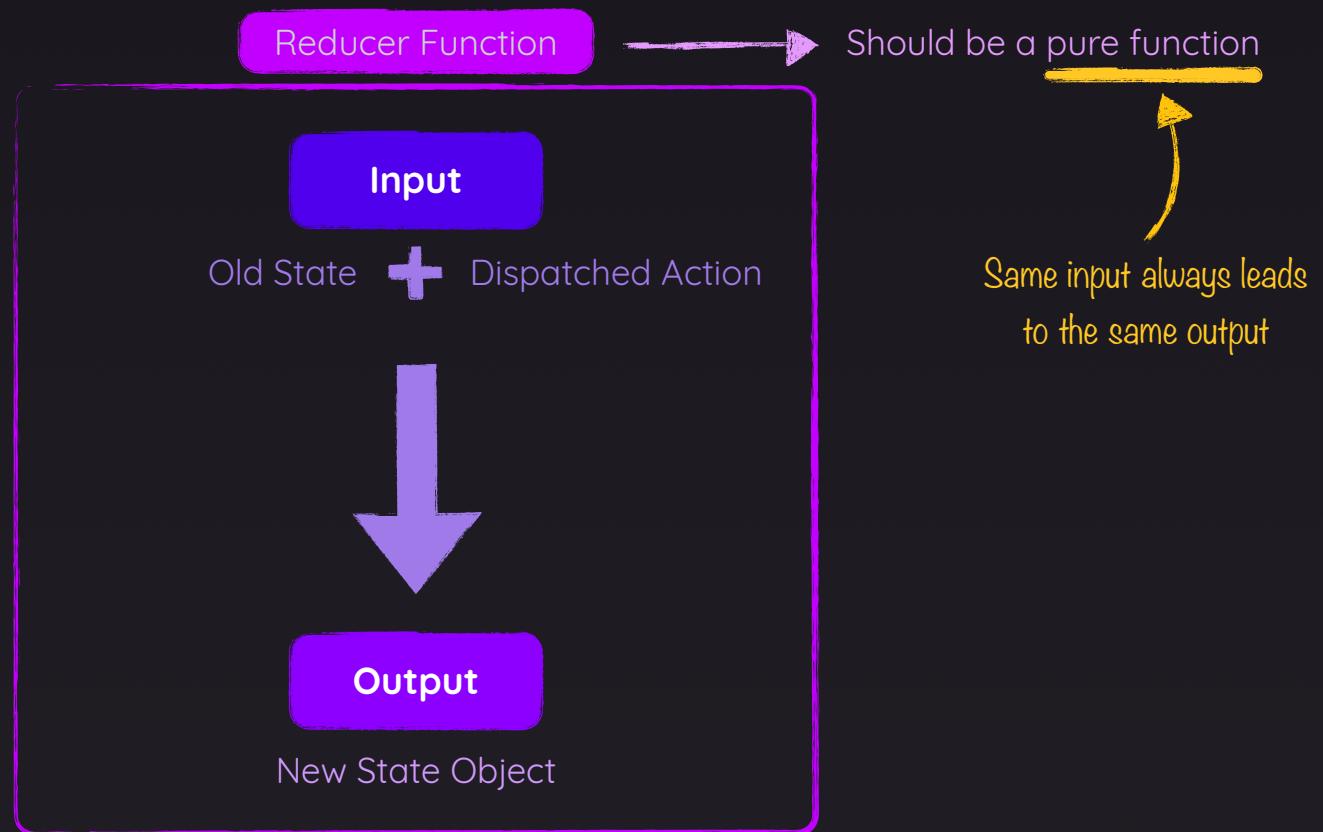
My personal summary is that new context is ready to be used for low frequency unlikely updates (like locale/theme). It's also good to use it in the same way as old context was used. I.e. for static values and then propagate updates through subscriptions. It's not ready to be used as a replacement for all Flux-like state propagation.

55 4

Core Redux Concepts



The Reducer Function



The Role Of Immutability

State updates must be done in an immutable way!

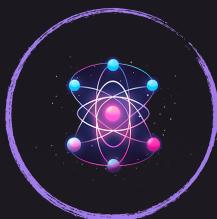


Can be tricky because **objects and arrays** are **reference values** in JavaScript

→ Changes to an object property **affect all places** where the object gets used



New object / array copies (also of nested objects / arrays) must be created when producing new state



Redux Deep Dive

Taking A Closer Look

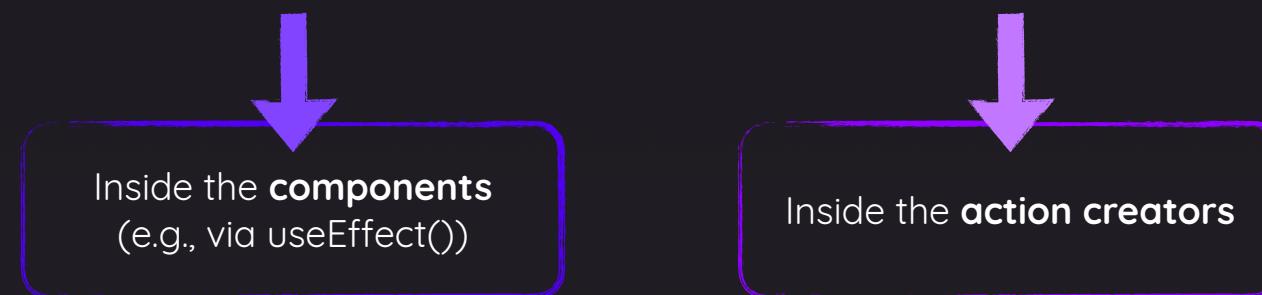
- ▶ Handling **Async Tasks** With Redux
- ▶ Where To **Put Your Code**
- ▶ The Redux **DevTools**

Side Effects, Async Tasks & Redux

Reducers must be **pure, side-effect free, synchronous** functions!



Where should side-effects & async tasks be executed?





Fat Reducers vs Fat Components vs Fat Actions

Where should your **logic** (= code) go?

Synchronous, side-effect free code
(i.e., data transformations)

Prefer reducers

Avoid action creators or components

Async code or code with side-effects

Prefer action creators or components

Never use reducers!

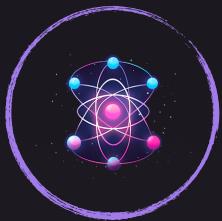


What is a “Thunk”?

A function that **delays an action** until later



An action creator function that does **NOT return the action itself** but instead **another function** which **eventually** returns the action

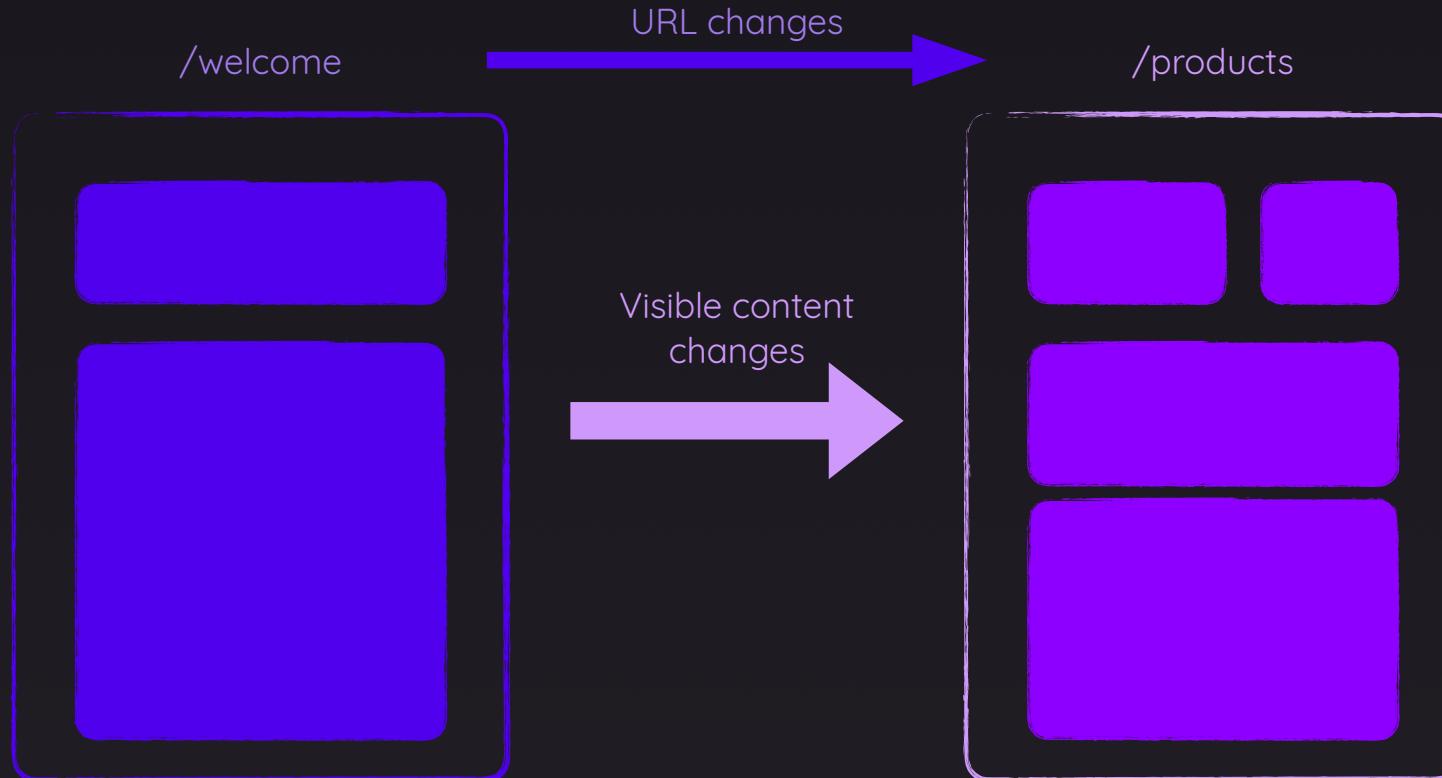


Single-Page Application Routing

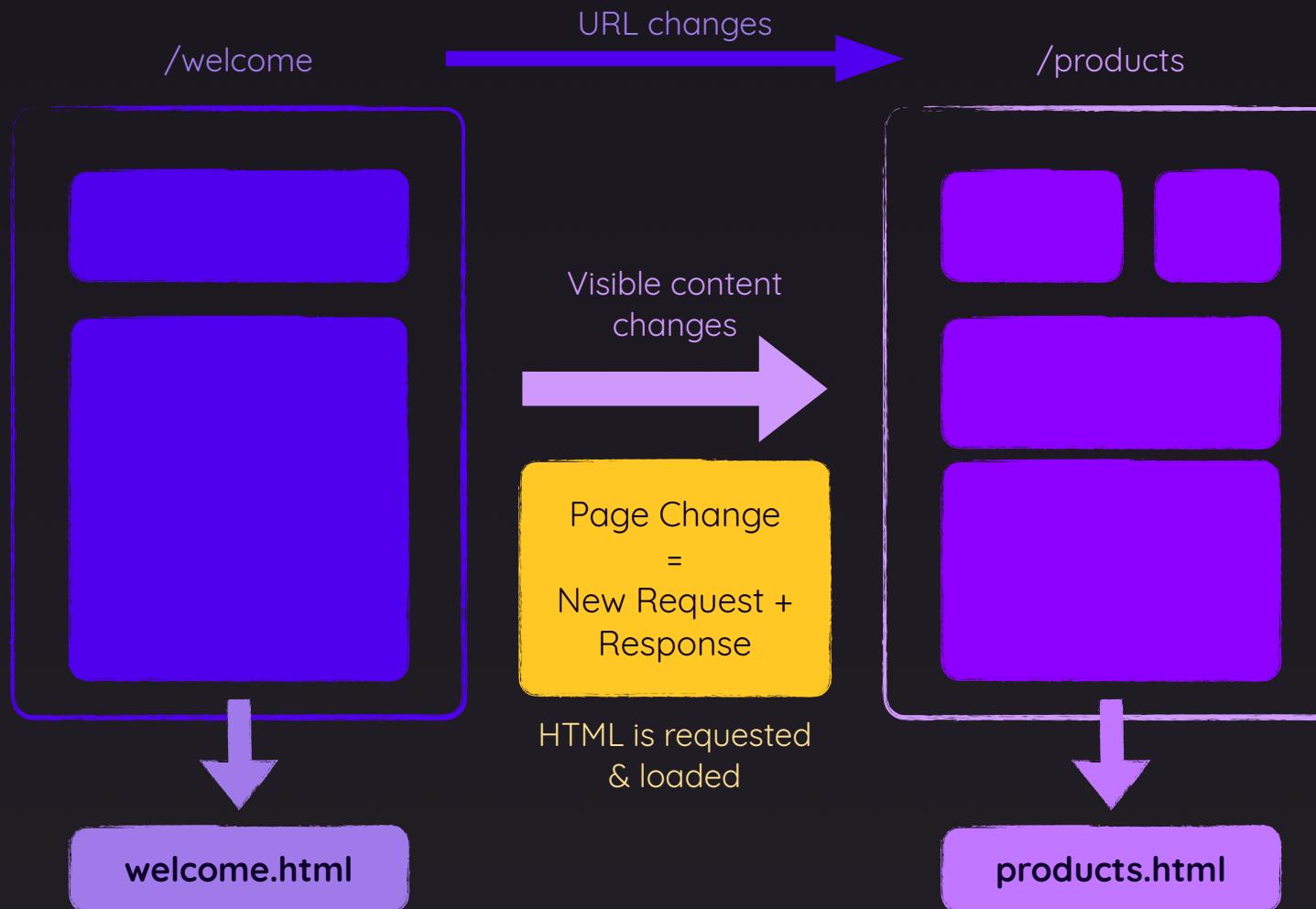
Multiple Pages In Single-Page Apps

- ▶ What & Why?
- ▶ Using **React Router**
- ▶ Data Fetching & Submission

What is a Routing?



Multi-Page Routing





Building SPAs

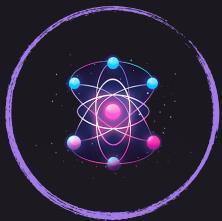
When building complex user interfaces, we typically build
Single Page Applications (SPAs)



Only one initial HTML request & response

Page (URL) changes are then
handled by client-side React code

Visible content is changed
without fetching a new HTML file



Authentication

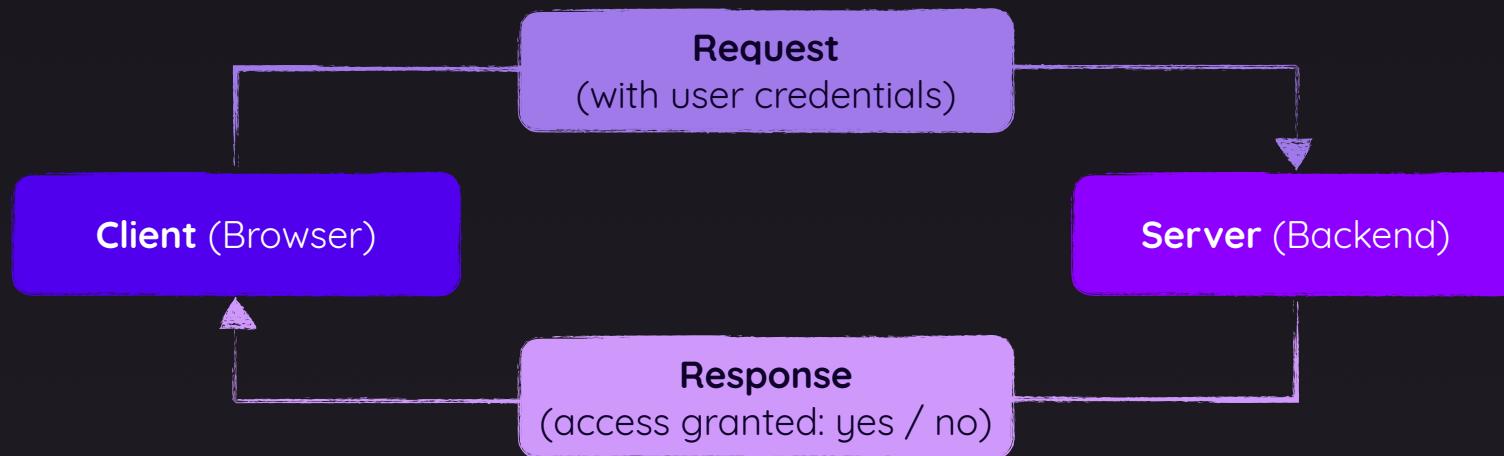
User Signup & Login

- ▶ How Authentication Works In React Apps
- ▶ Implementing User Authentication
- ▶ Adding Authentication Persistence & Auto-Logout

Authentication is needed if content should be protected

i.e., if content should not be
accessible by everyone

Getting Permission



Is that enough?

A “yes” alone is **not enough** to access protected resources /API endpoints)

Any client could simply send a request to our backend that “tells” the backend that we previously were granted access

How Does Authentication Work?

Client and server can't just exchange a "Yes"



Any client could simply send a request to our backend that "tells" the backend that we previously were granted access

Server-side Sessions

Store unique identifier on server,
send same identifier to client

Client sends identifier along with
requests to protected resources

Server can then check if the
identifier is valid (= previously
issued by server to client)

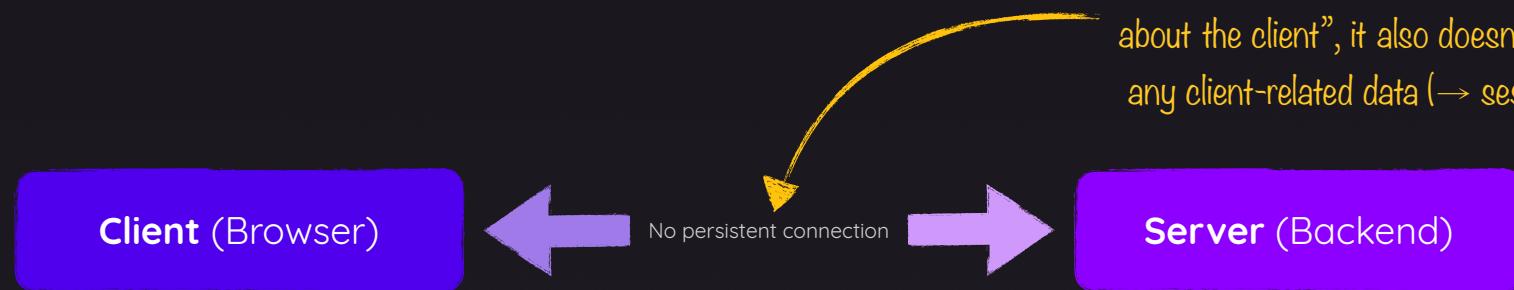
Authentication Tokens

Create (but not store) "permission"
token on server & send it to the client

Client attaches token to future
requests for protected resources

Server can then verify the
attached token

Working with Decoupled Backends



Since the backend “doesn’t care about the client”, it also doesn’t store any client-related data (→ sessions)

Most React apps are **SPAs** that are served by a server that’s **decoupled from the backend**

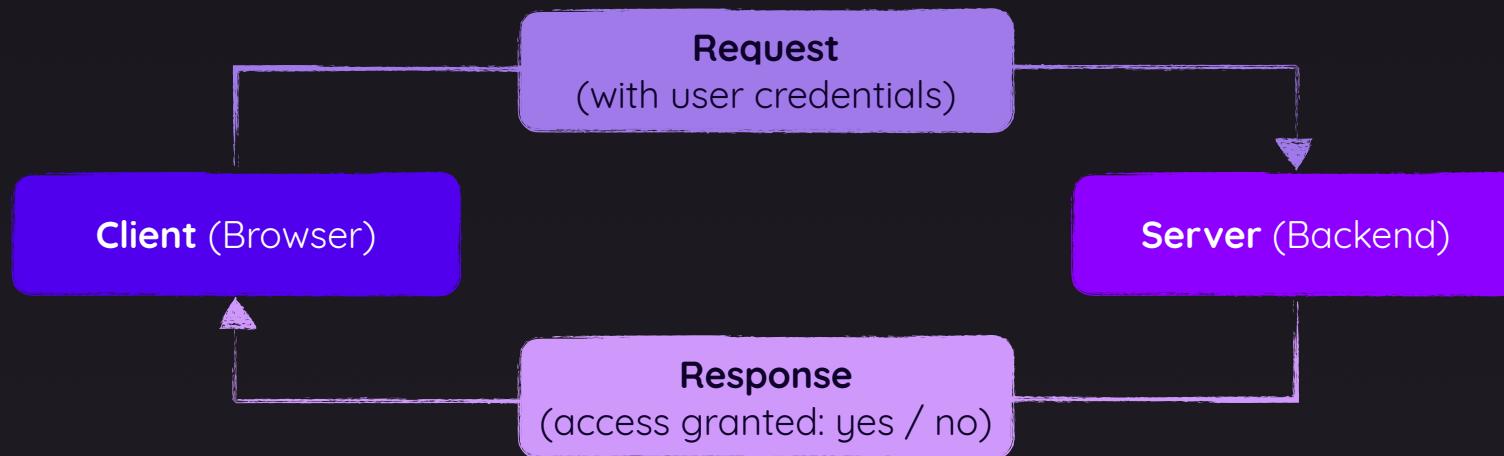
The SPA **handles routing** (on the client side) and only “talks” to the backend if it **needs data** (or needs to change data)

A **decoupled backend** is served by **different server** than the React frontend app

The backend **provides various resources routes** with which the client-side app may communicate

The backend does not register client-side routing actions or user interactions → it **may theoretically serve multiple, different client-side apps**
(React apps, mobile apps, etc)

Getting Permission

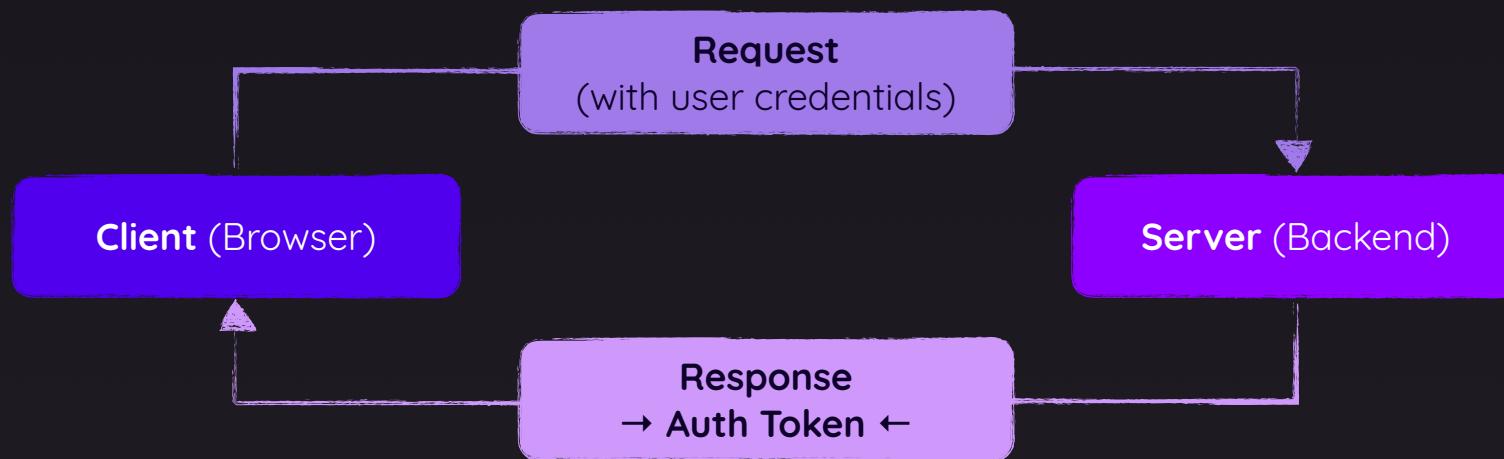


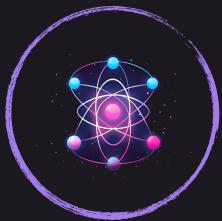
Is that enough?

A “yes” alone is **not enough** to access protected resources /API endpoints)

Any client could simply send a request to our backend that “tells” the backend that we previously were granted access

Getting Permission





Deploying React Apps

From Development To Production

- ▶ Deployment **Steps & Pitfalls**
- ▶ **Server-side** Routing vs **Client-side** Routing

Deployment Steps



Test Code: Manually & with automated tests



Optimize Code: Optimize user experience & performance



Build App: Run build process to parse, transform & optimize code



Upload App: Upload production code to hosting server



Configure Server: Ensure app is served securely & as intended

Lazy Loading

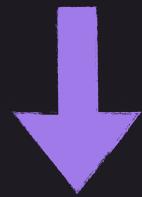
Load code only when it's needed

A React SPA is a “Static Website”

Only HTML, CSS & JavaScript

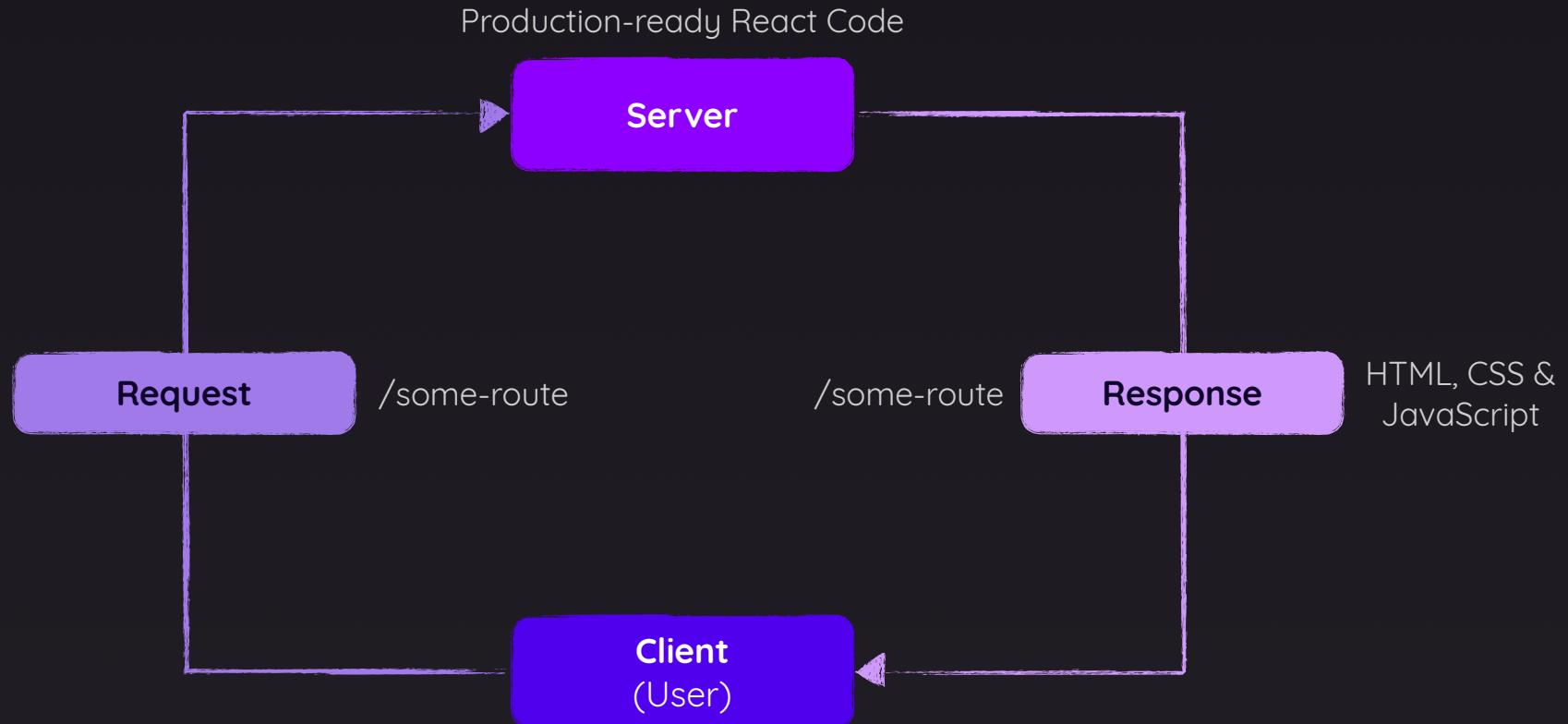
A React SPA is a “Static Website”

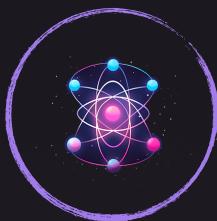
Only HTML, CSS & JavaScript



A static site host is needed

Server-side Routing vs Client-side Routing





Data Fetching & HTTP Requests

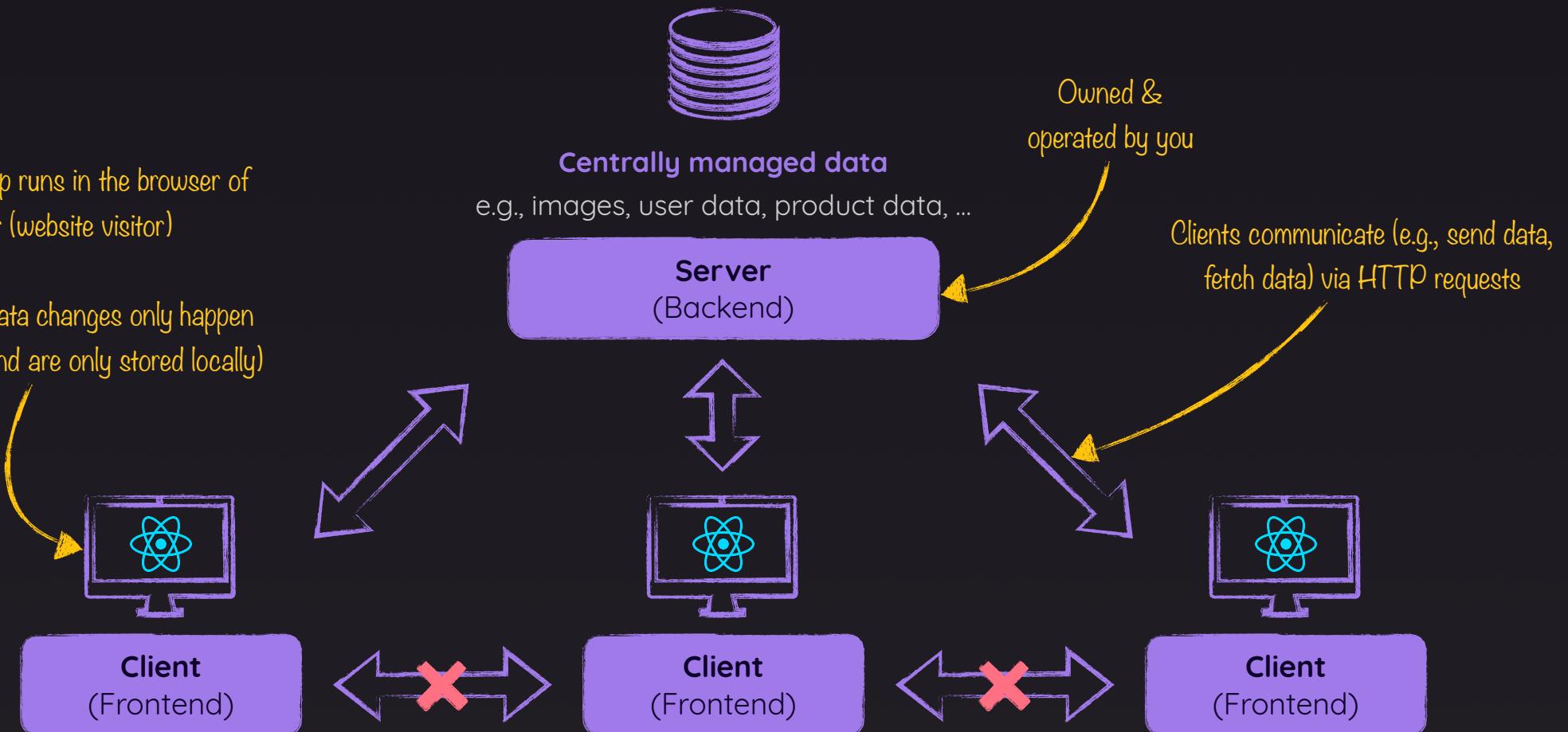
Sending & Receiving Data via HTTP

- ▶ How To **Connect a Backend** / Database
- ▶ **Fetching** Data
- ▶ **Sending** Data

Some Data Must Be Managed Centrally

React app runs in the browser of your user (website visitor)

→ any data changes only happen locally (and are only stored locally)





How Do You Connect Your React App To A Database?



You Don't!

At least not directly

Always keep in mind

**Your React code runs in the
browsers of your users**

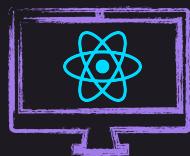
They can view that code (via the
browser developer tools) if they want to!

There also are technical restrictions & constraints

Not all operations can be performed in the browser

E.g, you can't access a (centrally managed) file system

Communicate with a Backend Server Instead



Frontend

Runs in the user's browser

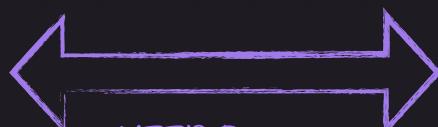
Users can theoretically view the entire code



Backend

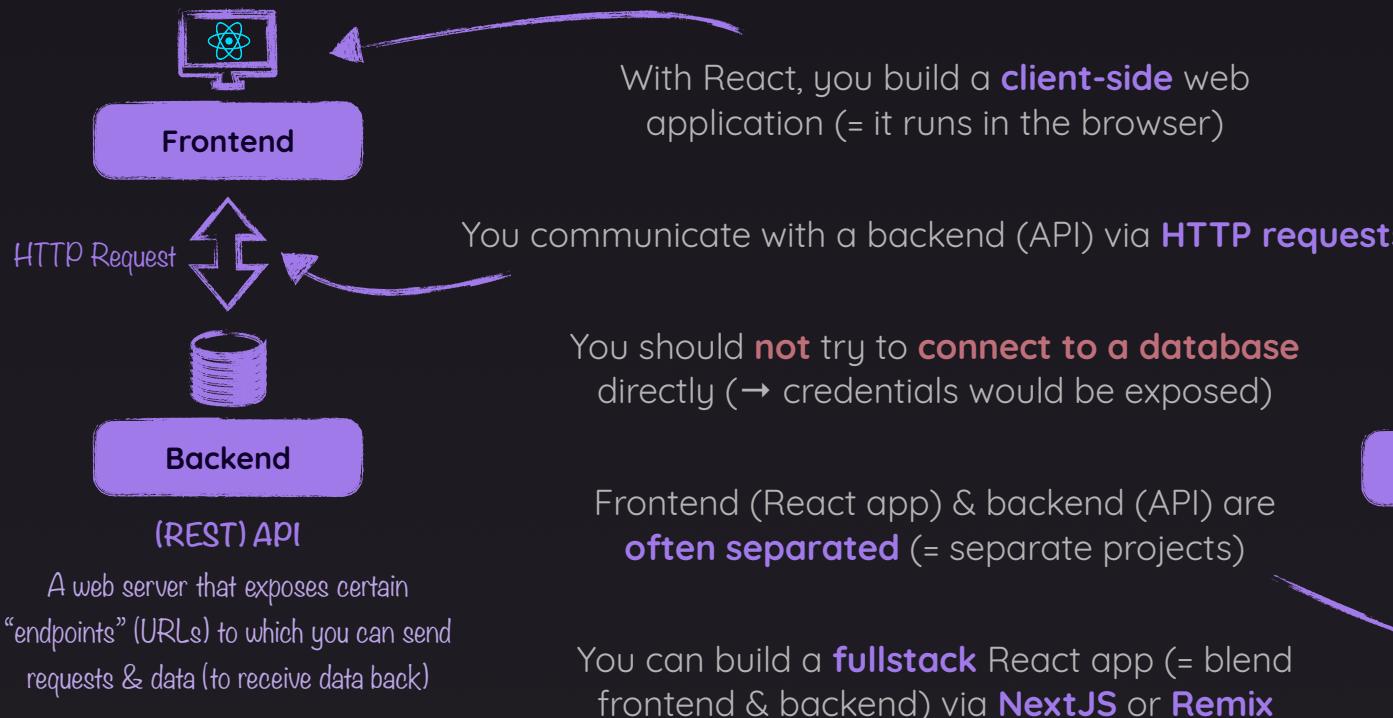
Runs on a separate (inaccessible) server

Backend code can interact with databases etc



HTTP Request

How To Connect A Backend / Database



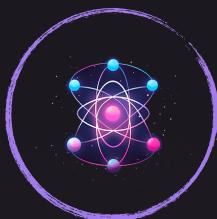
Frontend

React
(JavaScript)

Backend

PHP, C#, Java,
JavaScript, ...

You can build a **fullstack** React app (= blend frontend & backend) via **NextJS** or **Remix**



Custom Hooks

Creating & Using Custom React Hooks

- ▶ Repetition: **Rules of Hooks**
- ▶ **Why** Custom Hooks?
- ▶ **Creating** Custom Hooks
- ▶ **Using** Custom Hooks

Rules of Hooks

1

Only call Hooks inside of Component Functions

React Hooks must not be called outside of React component functions



```
function App() {  
  const [val, setVal] = useState(0);  
}
```



```
const [val, setVal] = useState(0);  
  
function App() { ... }
```

2

Only call Hooks on the top level

React Hooks must not be called in nested code statements (e.g., inside of if-statements)



```
function App() {  
  const [val, setVal] = useState(0);  
}
```



```
function App() {  
  if (someCondition)  
    const [val, setVal] = useState(0);  
}
```

Rules of Hooks – Updated

1

Only call Hooks in Component or Other Hook Functions



```
function App() {  
  const [val, setVal] = useState(0);  
}
```



```
const [val, setVal] = useState(0);  
  
function App() { ... }
```

2

Only call Hooks on the top level

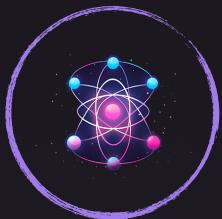
React Hooks must not be called in nested code statements (e.g., inside of if-statements)



```
function App() {  
  const [val, setVal] = useState(0);  
}
```



```
function App() {  
  if (someCondition)  
    const [val, setVal] = useState(0);  
}
```



Working with Forms & User Input

It's Trickier Than It Might Seem

- ▶ What's **Difficult** About Forms?
- ▶ Handling **Form Submission** & **Validating** User Input
- ▶ Using **Built-in** Form Features
- ▶ Building **Custom** Solutions

What's So Difficult?



Form Submission

Handling submission is relatively **easy**

Entered values can be managed via **state**

Alternatively, they can be extracted via **refs**

Or via **FormData** and native browser features



Input Validation

Providing a good user experience is **tricky**

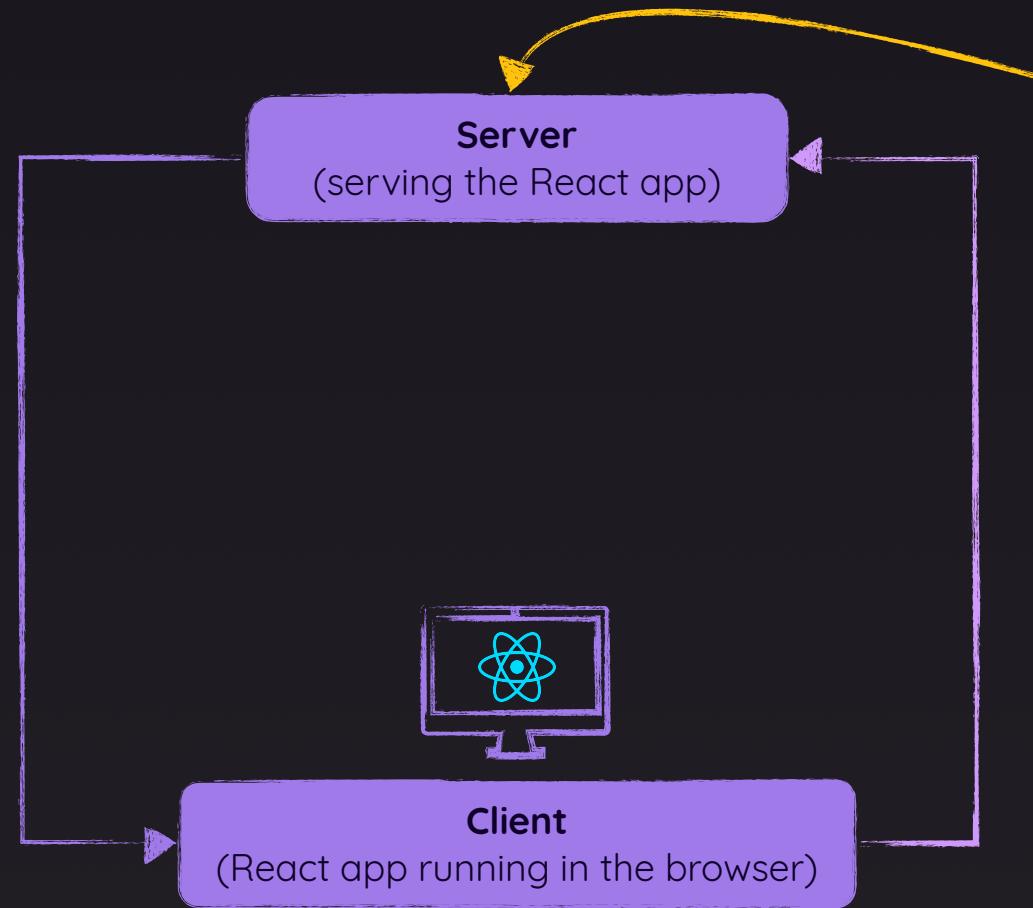
You can **validate** on every **keystroke** →
errors may be shown **too early**

You can **validate** on **lost focus** → errors
may be shown **too long**

You can **validate** on **form submission** →
errors may be shown **too late**

By Default The Browser Sends a HTTP Request

Serves the React app
JavaScript files + index.html
(+ any CSS needed) to
users visiting the website

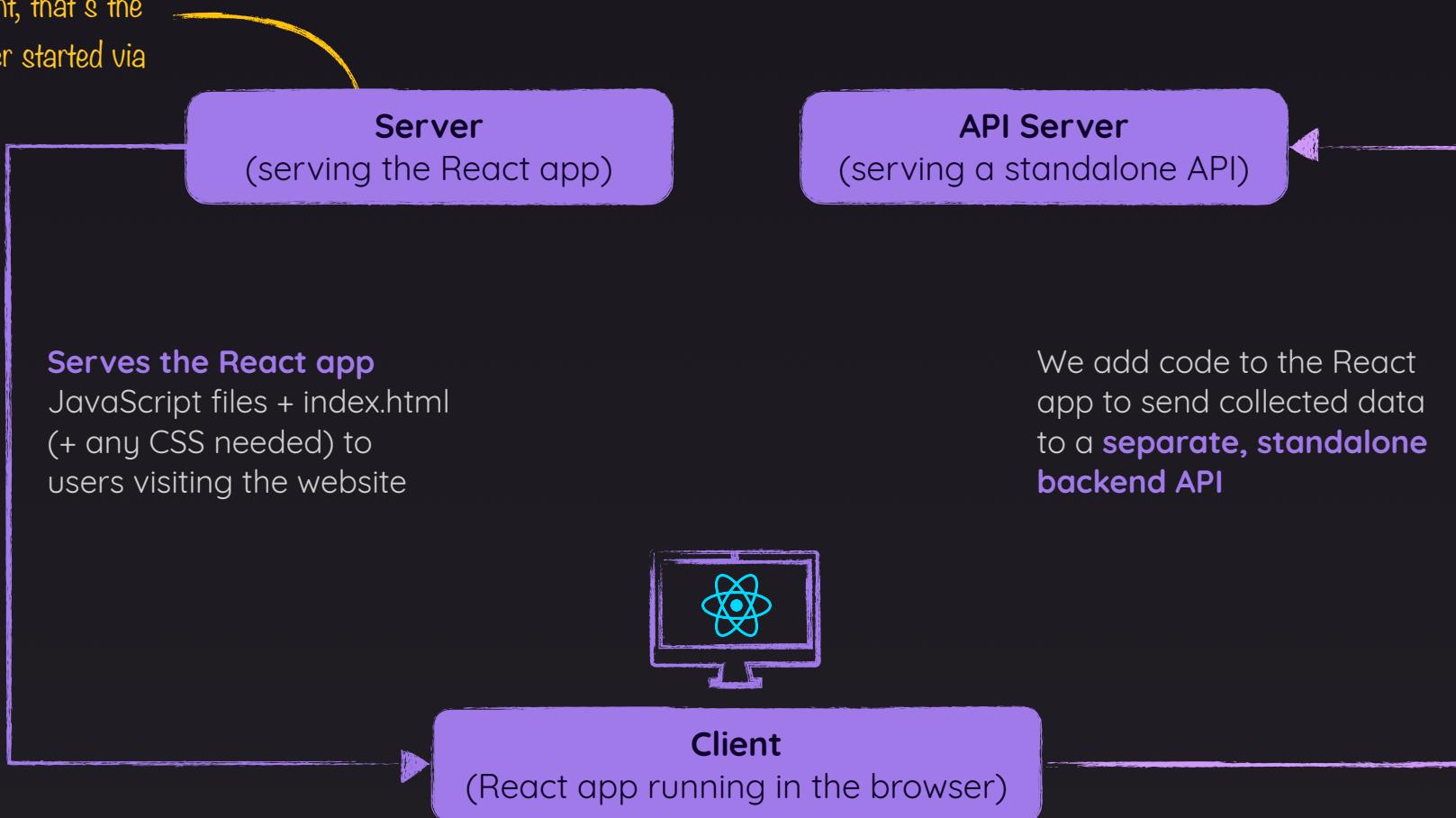


During development, that's the development server started via
“npm run dev”

Browser **automatically creates & sends a HTTP request** with entered **form data**

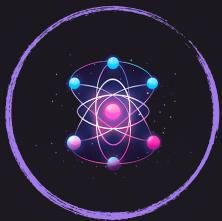
Often, Data Should Be Sent To A Backend

During development, that's the development server started via “npm run dev”





**Consider Using The
Native, Built-in Form
Handling Features!**



Time To Practice: Food Order App

Components, State, Context, Effects, HTTP Requests & More!

- ▶ Building a **Complete Project** From The Ground Up
- ▶ **Building & Configuring Components**
- ▶ Using **State & Context**
- ▶ Managing **HTTP Requests & Side Effects**

Time To Practice!



A Challenge For You

Build a “Food Order” web app

Use the **starting project** attached to this lecture

Add **components** for displaying **products**, the **cart** (in a **modal**) and a **checkout form** (also in a **modal**)

Fetch the (dummy) meals data from the **backend** & show it on the screen (GET /meals)

Allow users to **add & remove** products to / from the **cart**

Send cart data along with **user data** (full name, email, street, postal code, city) to the **backend** (POST /orders)

Handle **loading & error** states



We Need A Plan!

1

Add the **Header** component

2

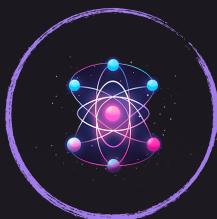
Add the **Meals-related** components & the logic to fetch meals data from a **backend**

3

Add **Cart** logic (add items to cart, edit cart items) & **Checkout** page logic



Feel free to **pause** this section & **try continuing on your own** anytime → It'll always be a good practice, even if you're not able to build everything



Animating React Apps

Using Framer Motion To Bring Things To Life

- ▶ “**Just CSS**” Might Be Enough!
- ▶ Building More **Complex Animations** with **Framer Motion**
- ▶ Animating Elements **In & Out**
- ▶ **Scroll-based** Animations

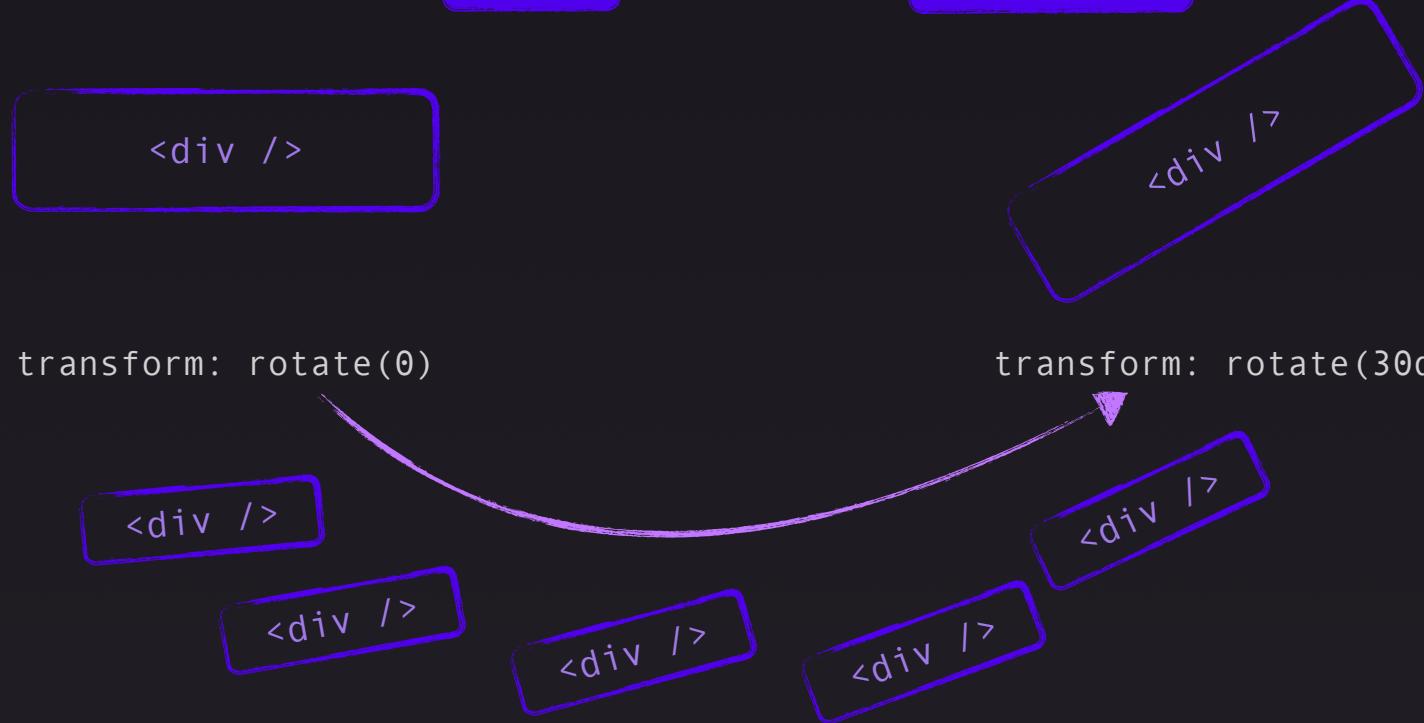


You Might Not Need An Animation Library

CSS Transitions & Animations
Might Be Enough

CSS Transitions

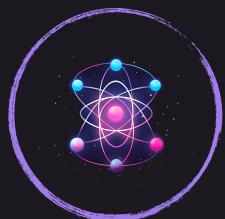
CSS Transitions allow you to “tell” CSS to smoothly transition between two CSS property values





**Framer Motion Allows You To
Build Good-Looking & Complex
Animations With Realistic Motion**

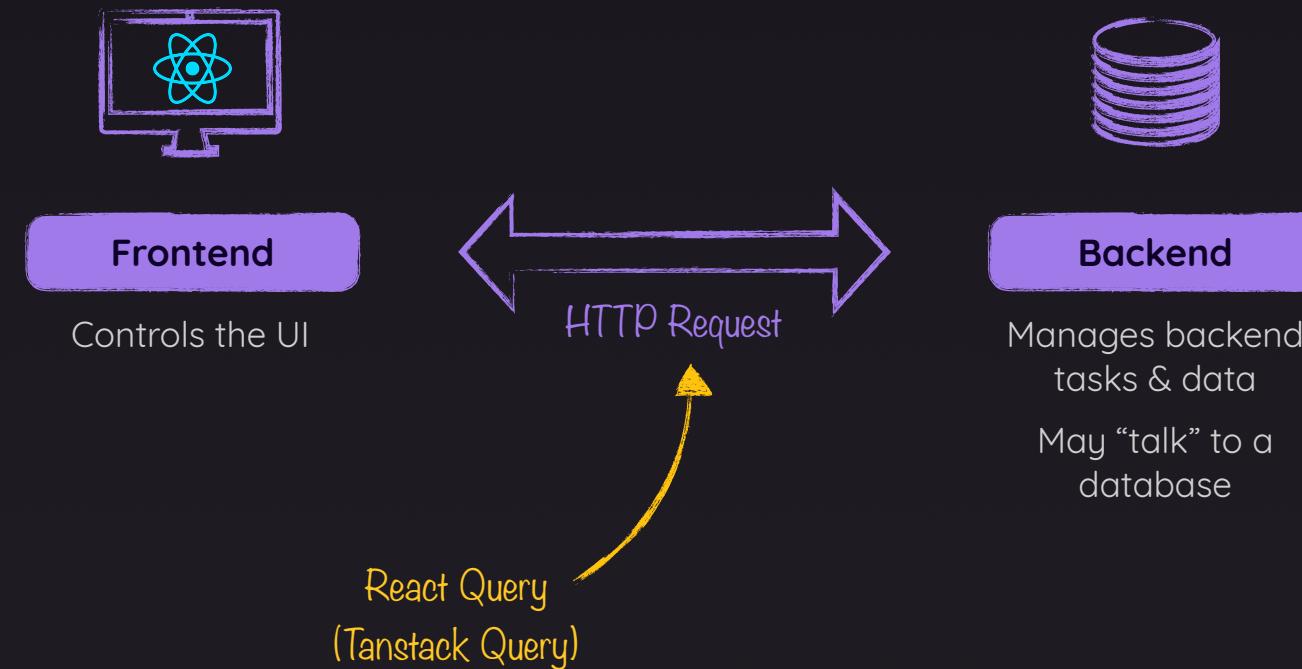
formerly React Query



Data Fetching with Tanstack Query

Sending HTTP Requests With Ease

- ▶ **What** Is Tanstack Query & **Why** Would You Use It?
- ▶ **Fetching & Mutating** Data
- ▶ **Configuring** Tanstack Query
- ▶ **Advanced Concepts:** Cache Invalidation, Optimistic Updating & More





What Is Tanstack Query?

What Is Tanstack Query?

A library that helps with sending
HTTP requests & keeping your
frontend UI in sync



You Don't Need Tanstack Query!

You Don't Need Tanstack Query!

But it can vastly simplify your code
(and your life as a developer)

Tanstack Query Does Not Send HTTP Requests

At least **not on its own**

You have to write the code that sends the actual HTTP request

Tanstack Query then manages the **data, errors, caching & much more!**

Tanstack Query Caches Query Data

```
useQuery({  
  queryKey: ['some-key'],  
  queryFn: fetchData  
});  
Called in Component A @ 10:32
```



fetchData() is executed
& HTTP request is sent



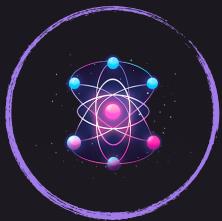
Data is **received**

```
{  
  id: 'd1',  
  title: 'Some data'  
}
```

```
useQuery({  
  queryKey: ['some-key'],  
  queryFn: fetchData  
});  
Called in Component B @ 10:34
```

Cached data is reused & shown
on the screen immediately

Cached (stored) by Tanstack
Query



React + TypeScript

Adding Type Safety To React Apps

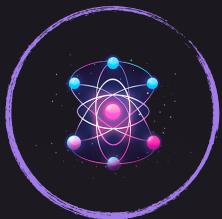
- ▶ What & Why?
- ▶ TypeScript Basics
- ▶ Combining React & TypeScript

What & Why?

TypeScript is a **superset**
to JavaScript

TypeScript adds static typing to JavaScript

**JavaScript on its own is
dynamically typed**



Testing React Apps

Automated Testing

- ▶ What is “Testing”? And why?
- ▶ Understanding Unit Tests
- ▶ Testing React Components & Building Blocks



What is “Testing”?



Manual Testing

Write Code → Preview & Test in Browser → Improve Code → Repeat

Very important: You see what your users will see



Error-prone: It's hard to always test all possible combinations & scenarios



Automated Testing

Write code that automatically tests your code

You test the individual building blocks of your app



Requires extra knowledge (→ how to write tests) but allows you to test all building blocks of your app



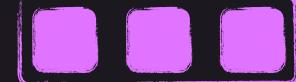
Different Kinds Of Automated Tests



Unit Tests



Integration Tests



End-to-End (E2E) Tests

Test the **individual building blocks** (functions, components) in **isolation**

Projects typically contain dozens or hundreds of unit tests

The most common / important kind of test

Test the **combination** of multiple building blocks

Projects typically contain a couple of integration tests

Also important, but focus on unit tests in most cases

Test **complete scenarios / user flows** in your app (as the user would experience them)

Projects typically contain only a few E2E tests

Important but can also be done manually (partially)



What Should You Test?

What?

+

How?



Test the different app building blocks

Unit tests: The smallest building blocks that make up your app

Test success and error cases,
also test rare (but possible)
results



Required Tools & Setup

We need a tool for running our tests and asserting the results



We need a tool for “simulating” (rendering) our React app / components



Jest



React Testing Library

Both tools are already set up for you when using create-react-app

For Vite & CodeSandbox, you find appropriate project setups attached!

Writing Tests — The Three “A”s

Arrange



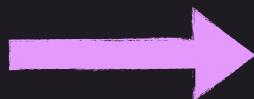
Set up the test data, test conditions and test environment

Act



Run logic that should be tested (e.g., execute function)

Assert



Compare execution results with expected results