

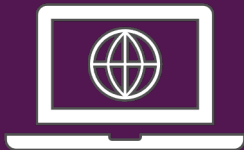
What Is React? And Why Would We Use It?

React is a JavaScript library for
building user interfaces

React makes building **complex**,
interactive and **reactive** user
interfaces **simpler**

What is React.js?

React.js



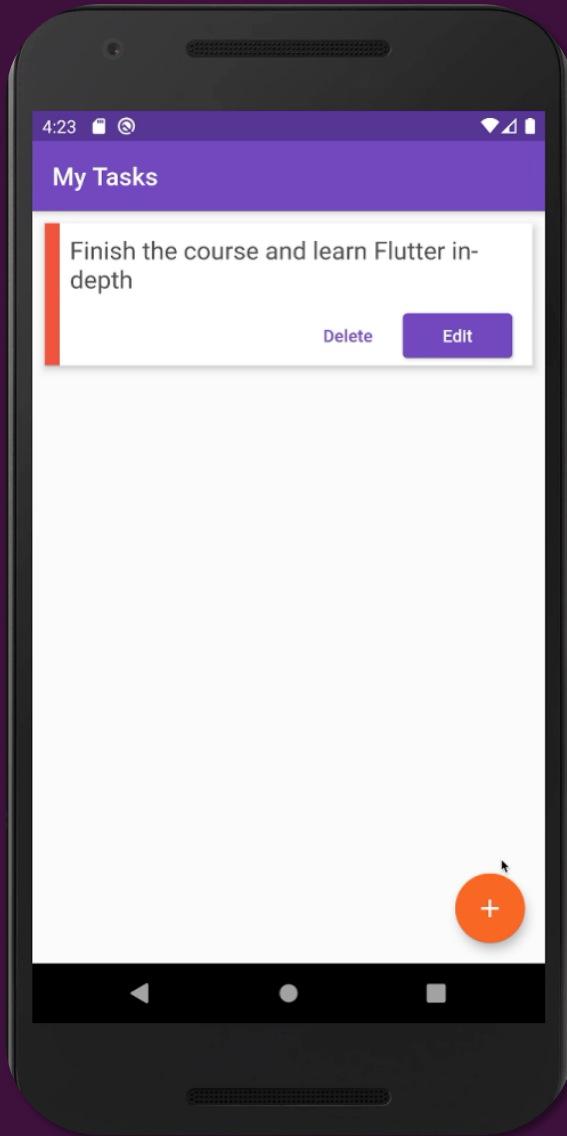
A client-side JavaScript
library



All about building modern,
reactive user interfaces for
the web



Declarative, component-
focused approach

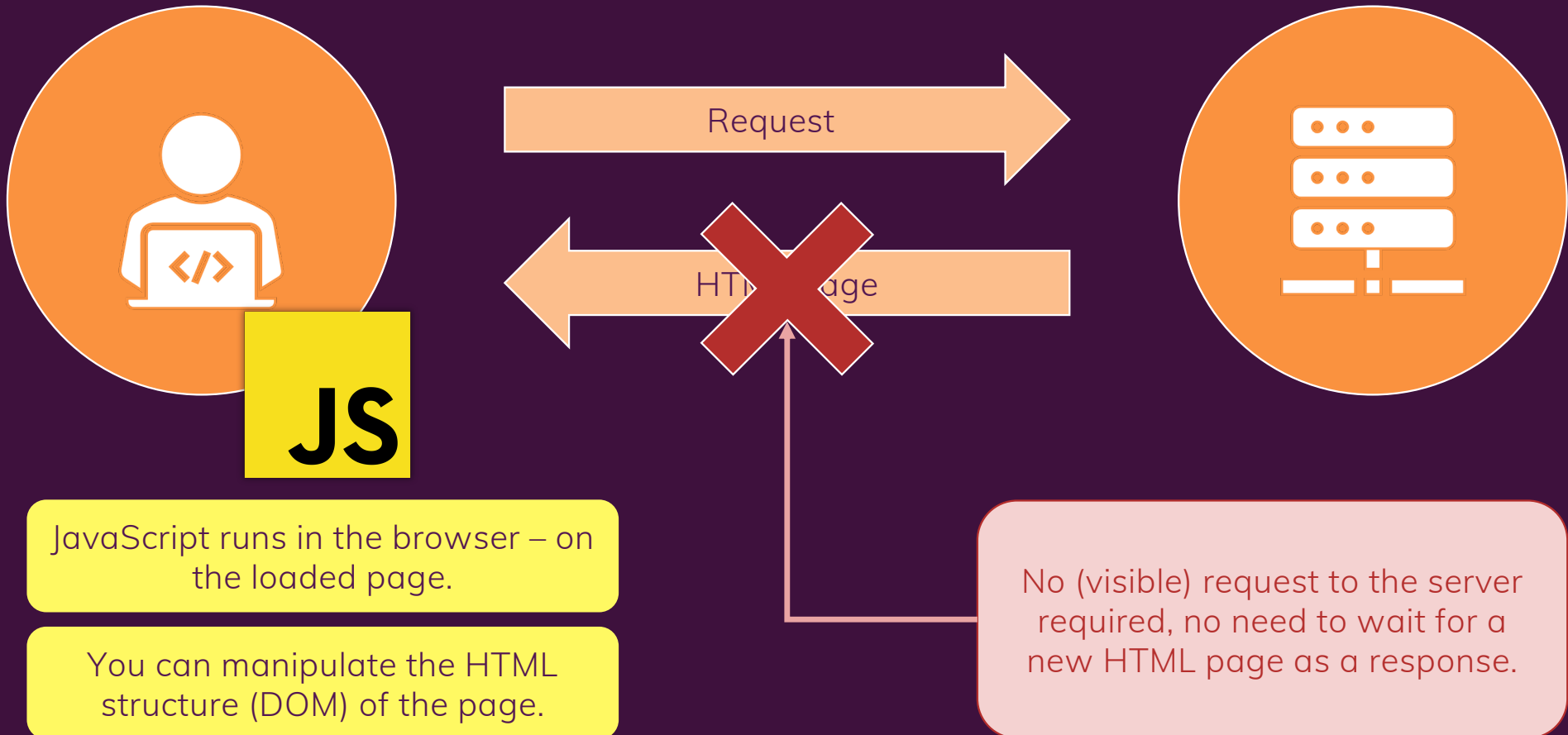


Mobile apps and desktop apps **feel** very “**reactive**”: Things happen instantly, you **don’t wait** for new pages to load or actions to start.

Traditionally, in web apps, you click a link and wait for a new page to load. You click a button and wait for some action to complete.

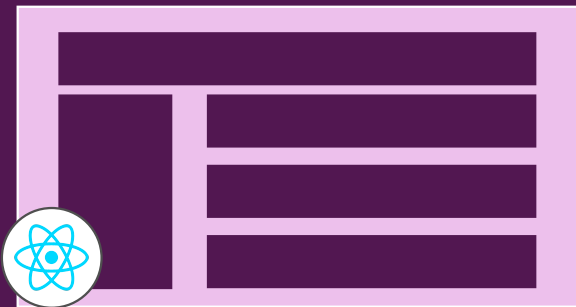


JavaScript To The Rescue!



Building Single-Page-Applications (SPAs)

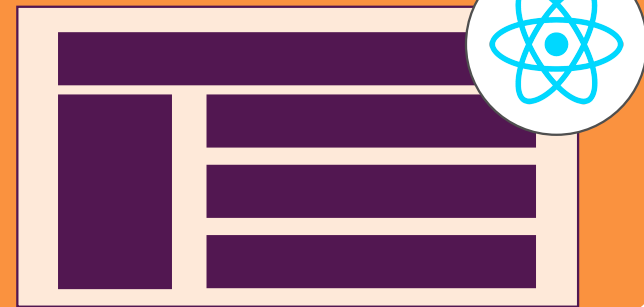
React can be used to **control parts** of HTML pages or entire pages.



“**Widget**” approach on a multi-page-application.
(Some) pages are still **rendered on and served by a backend server**.

In this course!

React can also be used to **control the entire frontend** of a web application



“Single-Page-Application” (SPA) approach. Server **only sends one HTML page**, thereafter, React takes over and controls the UI.

HTML, CSS & JavaScript are about
building user interfaces **as well**

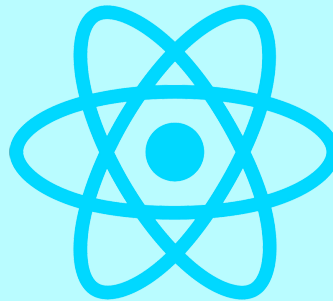
React.js Alternatives

Angular



Complete component-based UI framework, packed with features. Uses TypeScript. Can be overkill for smaller projects.

React.js



Lean and focused component-based UI library. Certain features (e.g. routing) are added via community packages.

Vue.js



Complete component-based UI framework, includes most core features. A bit less popular than React & Angular.

About This Course & Course Outline

Theory / Small
Demos & Examples



More Realistic
(Bigger) Example
Projects



Challenges &
Exercises

Components & Building
UIs

Working with Events &
Data: "props" and "state"

Styling React Apps &
Components

Introduction into "React
Hooks"

Basics & Foundation
(Introducing Key Features)

Side Effects, "Refs" & More
React Hooks

React's Context API &
Redux

Forms, Http Requests &
"Custom Hooks"

Routing, Deployment,
NextJS & More

Advanced Concepts
(Building for Production)

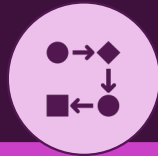
JavaScript Refresher

ReactJS Summary

React Hooks Summary

Summaries & Refreshers
(Optimizing your Time)

Taking This Course: Two Options



Standard Approach *(Recommended)*

Start with lecture 1 in section 1 and go through the course step by step

Skip JavaScript refresher module if you don't need it

Use React summary module at the end to summarize what you learned or to refresh knowledge in the future



Summary Approach *(If you're in a hurry)*

Skip forward to the React summary module

Optionally also take JavaScript refresher module if you need it

Go through the entire course after going through the summary module and / or if you got more time in the future

How To Get The Most Out Of The Course



Watch the Videos
(choose your pace)



Code Along & Practice
(also without me telling you)



Debug Errors & Explore Solutions
(also use code attachments)



Help Each Other & Learn Together
(Discord, Q&A Board)

React is a JavaScript library for
building user interfaces

HTML, CSS & JavaScript are about
building user interfaces **as well**

React makes building **complex**,
interactive and **reactive** user
interfaces **simpler**

React is all about “Components”

What is a “Component”?

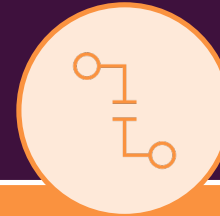
React is all about “**Components**”
Because all user interfaces in
the end are made up of
components

Why Components?



Reusability

Don't repeat yourself



Separation of Concerns

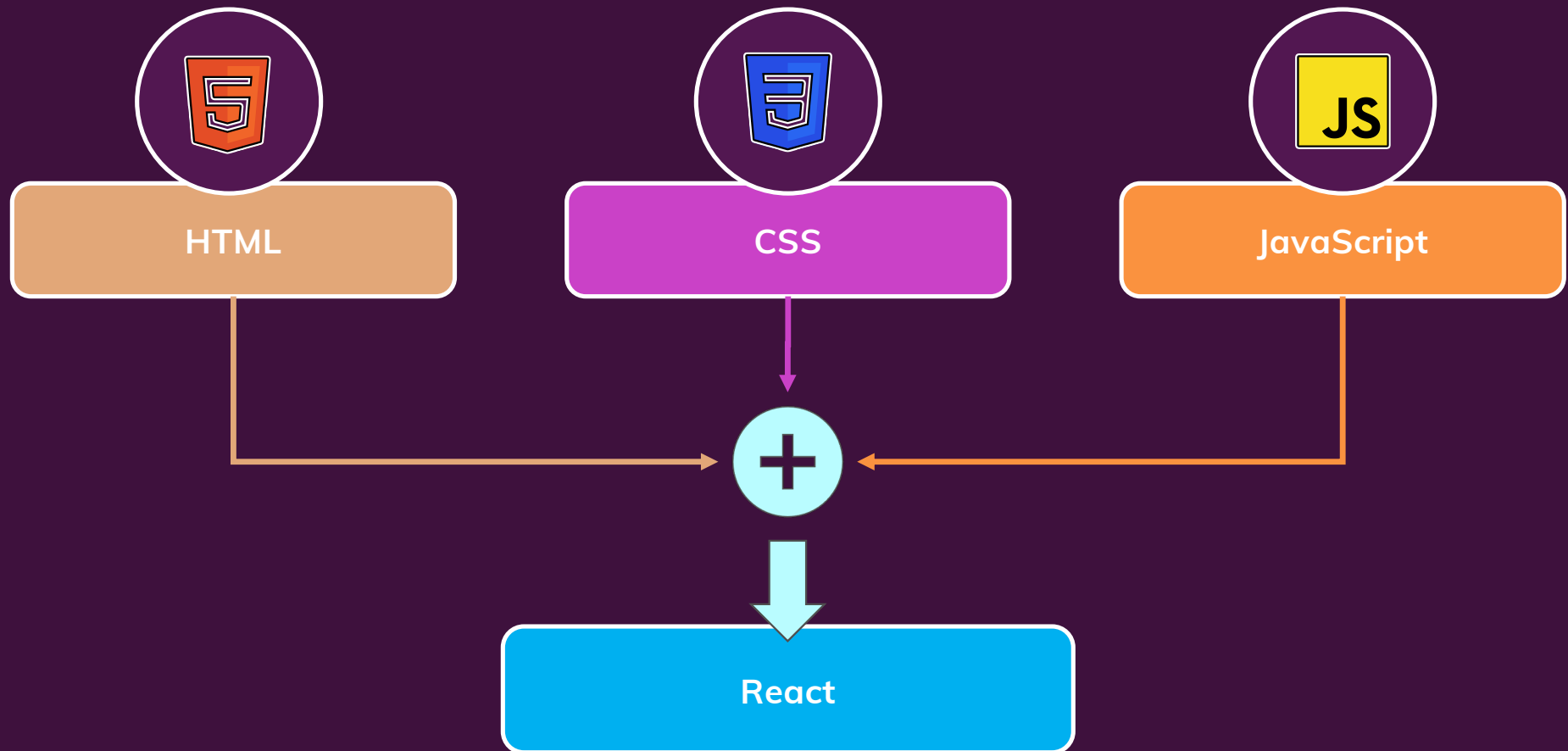
Don't do too many things in one
and the same place (function)



Split big chunks of code into
multiple smaller functions



How Is A Component Built?



React & Components

React allows you to create **re-usable and reactive components** consisting of **HTML and JavaScript** (and CSS)



Declarative Approach

Define the desired target state(s) and let React figure out the actual JavaScript DOM instructions

Build your own, custom HTML Elements

JSX = “HTML in JavaScript”

Understanding JSX

```
<p title="Intro text">  
React.js is a library for  
building user interfaces.  
</p>
```

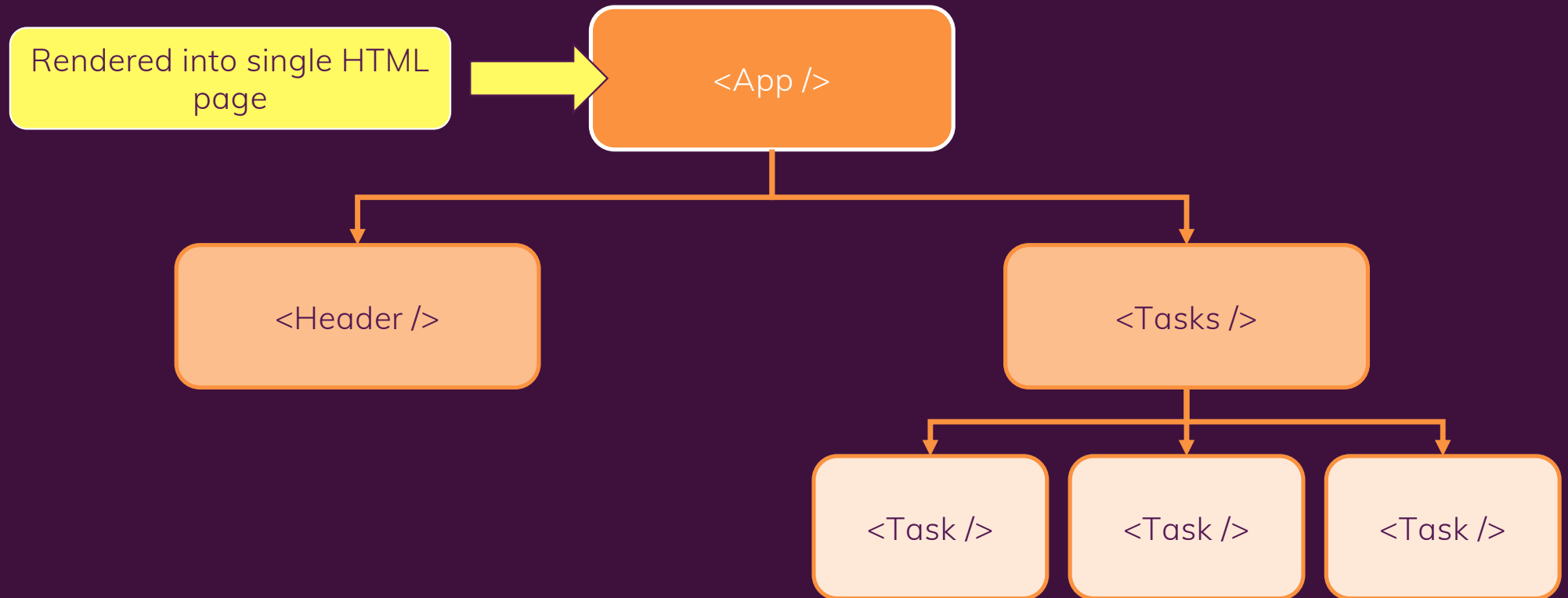


```
React.createElement(  
  'p',  
  { title: 'Intro text' },  
  'React.js is a library for  
  building user interfaces.'  
);
```

“**Syntactic sugar**”, does **not run** in the browser like this!

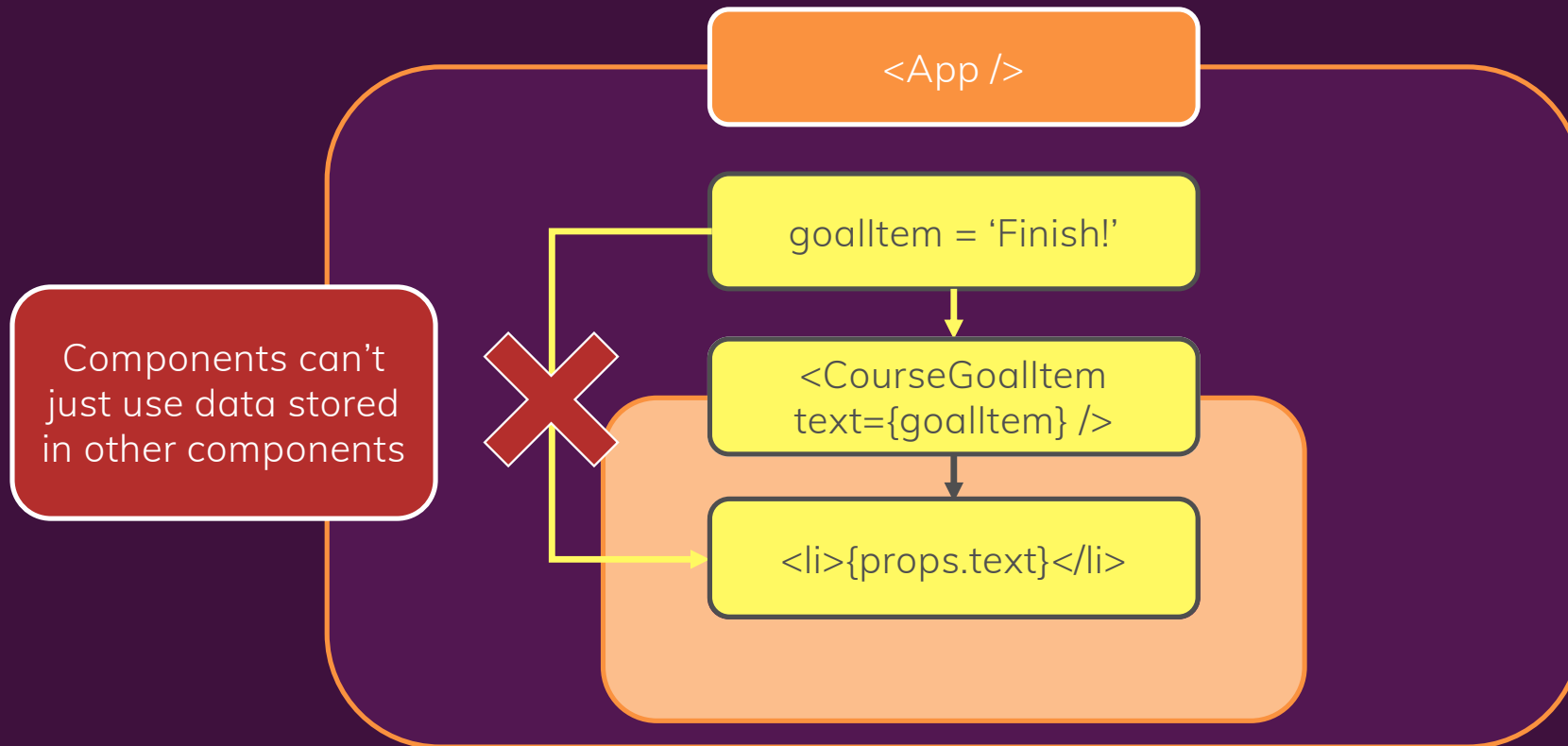
Real JavaScript code, would run in the browser like this. Not nice to use for more complex than “HTML code”.

You Build A Component Tree



Props are the “**attributes**” of your
“custom HTML elements” (Components)

Passing Data via "Props"



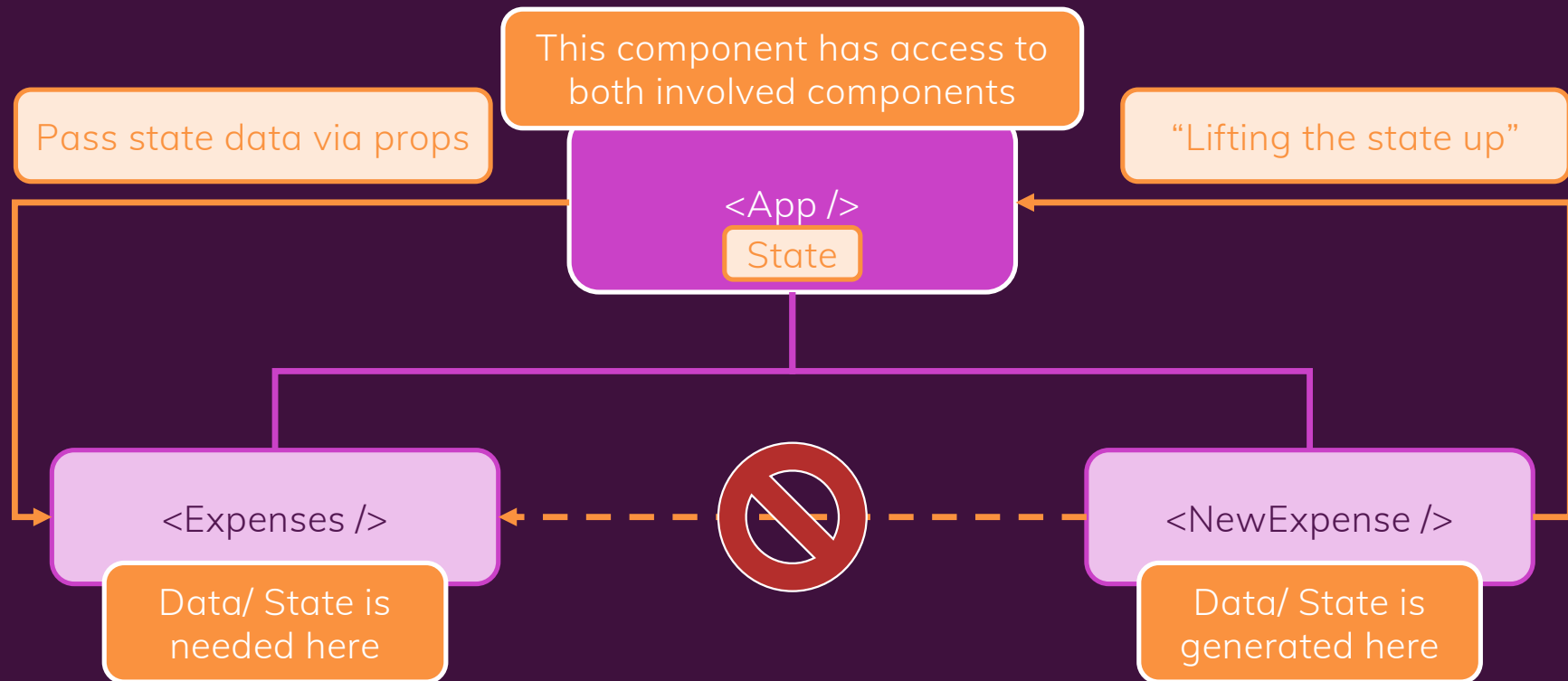
Updating Data via “State”

By default, React **does not care about changes** of variables inside of components. It does **not re-evaluate** the component's JSX markup.

“State” is data **managed by React**, where changes of the data do **force React to re-evaluate** (“re-render”) the component where the data changed.

Child components of components where state changed are **also re-evaluated**.

Lifting State Up



Stateful vs Stateless Components

Stateful Components

React components that manage internal state

Typically, you have only a couple of these

Also called “smart” components or “containers”.

Stateless Components

React components which only (possibly) use props, output JSX and add styling.

Typically, you have plenty of these.

Also called “dumb” or “presentational” components.

An Alternative Way Of Building Components

Functional Components

JavaScript Functions which return JSX

React executes them for you (initially and upon state changes)

Use "React Hooks" for state management

Class-based Components

JavaScript classes as blueprints for components.

render() method for outputting JSX (called by React)

Historically (React <16.8), the only way of managing state!

JSX Limitations

```
return (  
  <h2>Hi there!</h2>  
  <p>This does not work :-(</p>  
);
```

You **can't return more than one "root" JSX element** (you also can't store more than one "root" JSX element in a variable).

Because this also isn't valid JavaScript

```
return (  
  React.createElement('h2', {}, 'Hi there!')  
  React.createElement('p', {}, 'This does not work :-(  
);
```


The Solution: Always Wrap Adjacent Elements

```
return (  
  <div>  
    <h2>Hi there!</h2>  
    <p>This does not work :-(</p>  
  </div>  
);
```

Important: Doesn't have to be a <div> - ANY element will do the trick.

A New Problem: “<div> Soup”

```
<div>
  <div>
    <div>
      <div>
        <h2>Some content - yeah, this can really happen.</h2>
      </div>
    </div>
  </div>
</div>
```

In bigger apps, you can easily end up with **tons of unnecessary <div>s** (or other elements) which add **no semantic meaning or structure** to the page but **are only there because of React's/ JSX' requirement**.

Introducing Fragments

OR

```
return (  
  <React.Fragment>  
    <h2>Hi there!</h2>  
    <p>This does not work :-(</p>  
  </React.Fragment>  
);
```

```
return (  
  <>  
    <h2>Hi there!</h2>  
    <p>This does not work :-(</p>  
  </>  
);
```

It's an **empty wrapper component**: It **doesn't render** any real HTML element to the DOM. But it **fulfills React's/ JSX' requirement**.

Understanding React Portals

```
return (  
  <React.Fragment>  
    <MyModal />  
    <MyInputForm />  
  </React.Fragment>  
);
```



Real DOM

```
<section>  
  <h2>Some other content ... </h2>  
  <div class="my-modal">  
    <h2>A Modal Title!</h2>  
  </div>  
  <form>  
    <label>Username</label>  
    <input type="text" />  
  </form>  
</section>
```

Semantically and from a “clean HTML structure” perspective, having this nested modal isn’t ideal. It is an **overlay to the entire page** after all (that’s similar for side-drawers, other dialogs etc.).

Understanding React Portals

It's a bit like styling a `<div>` like a `<button>` and adding an event listener to it: It'll work, but it's not a good practice.

```
<div onClick={clickHandler}>Click me, I'm a bad button</div>
```

Understanding React Portals

```
return (  
  <React.Fragment>  
    <MyModal />  
    <MyInputForm />  
  </React.Fragment>  
);
```

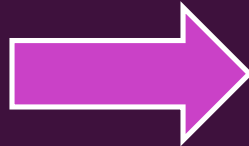


Real DOM

```
<section>  
  <h2>Some other content ... </h2>  
  <div class="my-modal">  
    <h2>A Modal Title!</h2>  
  </div>  
  <form>  
    <label>Username</label>  
    <input type="text" />  
  </form>  
</section>
```

Understanding React Portals

```
return (  
  <React.Fragment>  
    <MyModal />  
    <MyInputForm />  
  </React.Fragment>  
);
```



Real DOM

```
<div class="my-modal">  
  <h2>A Modal Title!</h2>  
</div>  
<section>  
  <h2>Some other content ... </h2>  
  <form>  
    <label>Username</label>  
    <input type="text" />  
  </form>  
</section>
```

What is an “Effect” (or a “Side Effect”)?

Main Job: Render UI & React to User Input

Evaluate & Render JSX
Manage State & Props
React to (User) Events & Input
Re-evaluate Component upon State &
Prop Changes

This all is “baked into” React via the “tools”
and features covered in this course (i.e.
useState() Hook, Props etc).

Side Effects: Anything Else

Store Data in Browser Storage
Send Http Requests to Backend Servers
Set & Manage Timers
...

These tasks **must happen outside of the normal component evaluation** and render cycle – especially since they might block/delay rendering (e.g. Http requests)

Handling Side Effects with the useEffect() Hook

```
useEffect(() => { ... }, [ dependencies ]);
```

A function that should be executed AFTER every component evaluation IF the specified dependencies changed

Your side effect code goes into this function.

Dependencies of this effect – the function only runs if the dependencies changed

Specify your dependencies of your function here

Introducing useReducer() for State Management

Sometimes, you have **more complex state** – for example if it got **multiple states, multiple ways of changing** it or **dependencies** to other states



useState() then often **becomes hard or error-prone to use** – it's easy to write bad, inefficient or buggy code in such scenarios



useReducer() can be used as a **replacement** for useState() if you need **“more powerful state management”**

Understanding useReducer()

```
const [state, dispatchFn] = useReducer(reducerFn, initialState, initFn);
```

The state snapshot used in the component re-render/ re-evaluation cycle

A function that can be used to dispatch a new action (i.e. trigger an update of the state)

The initial state

A function to set the initial state programmatically

`(prevState, action) => newState`

A function that is **triggered automatically** once an action is **dispatched** (via `dispatchFn()`) – it **receives the latest state snapshot** and **should return the new, updated state**.

useState() vs useReducer()

Generally, you'll know when you need useReducer() (→ when using useState() becomes cumbersome or you're getting a lot of bugs/ unintended behaviors)

useState()

The main state management "tool"

Great for independent pieces of state/ data

Great if state updates are easy and limited to a few kinds of updates

useReducer()

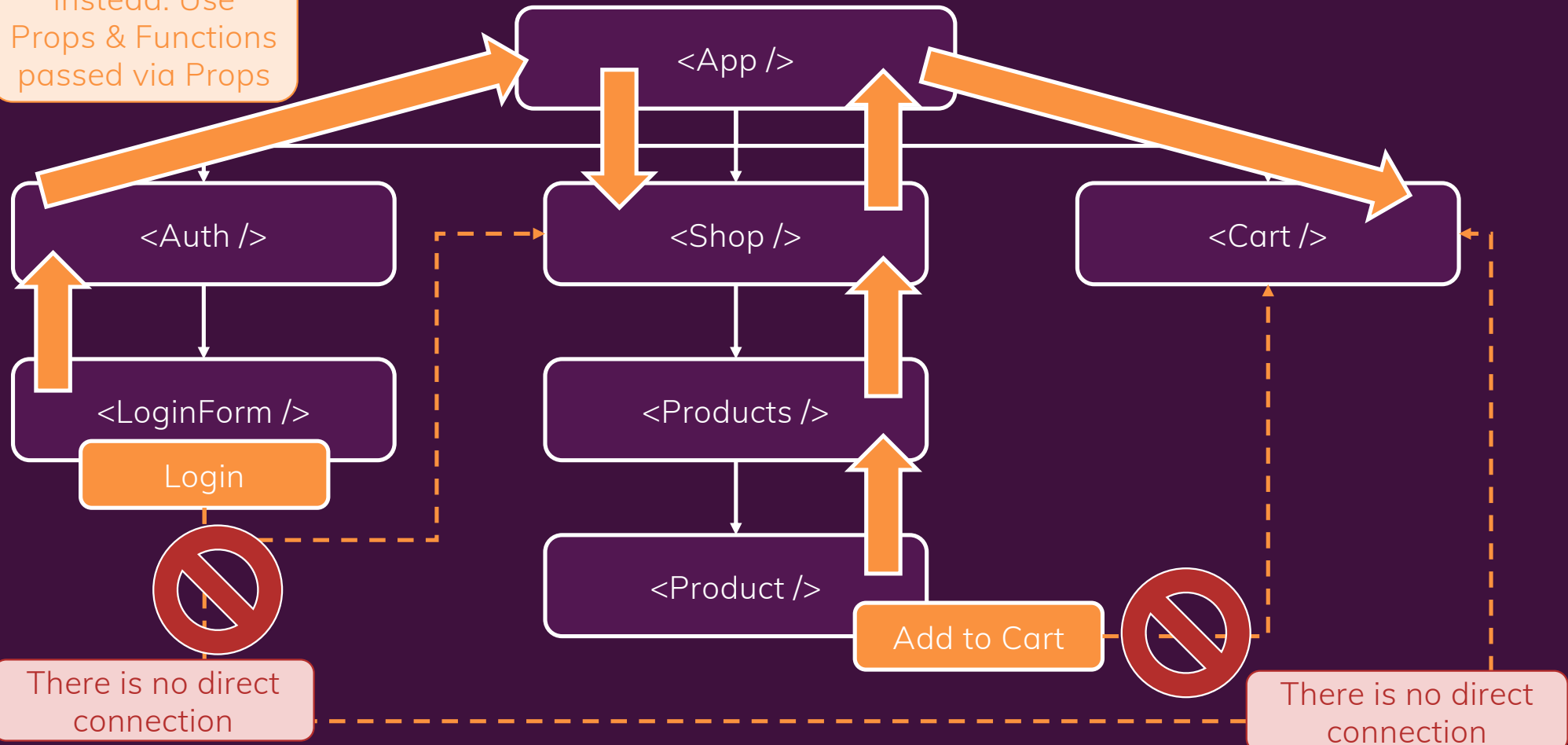
Great if you need "more power"

Should be considered if you have related pieces of state/ data

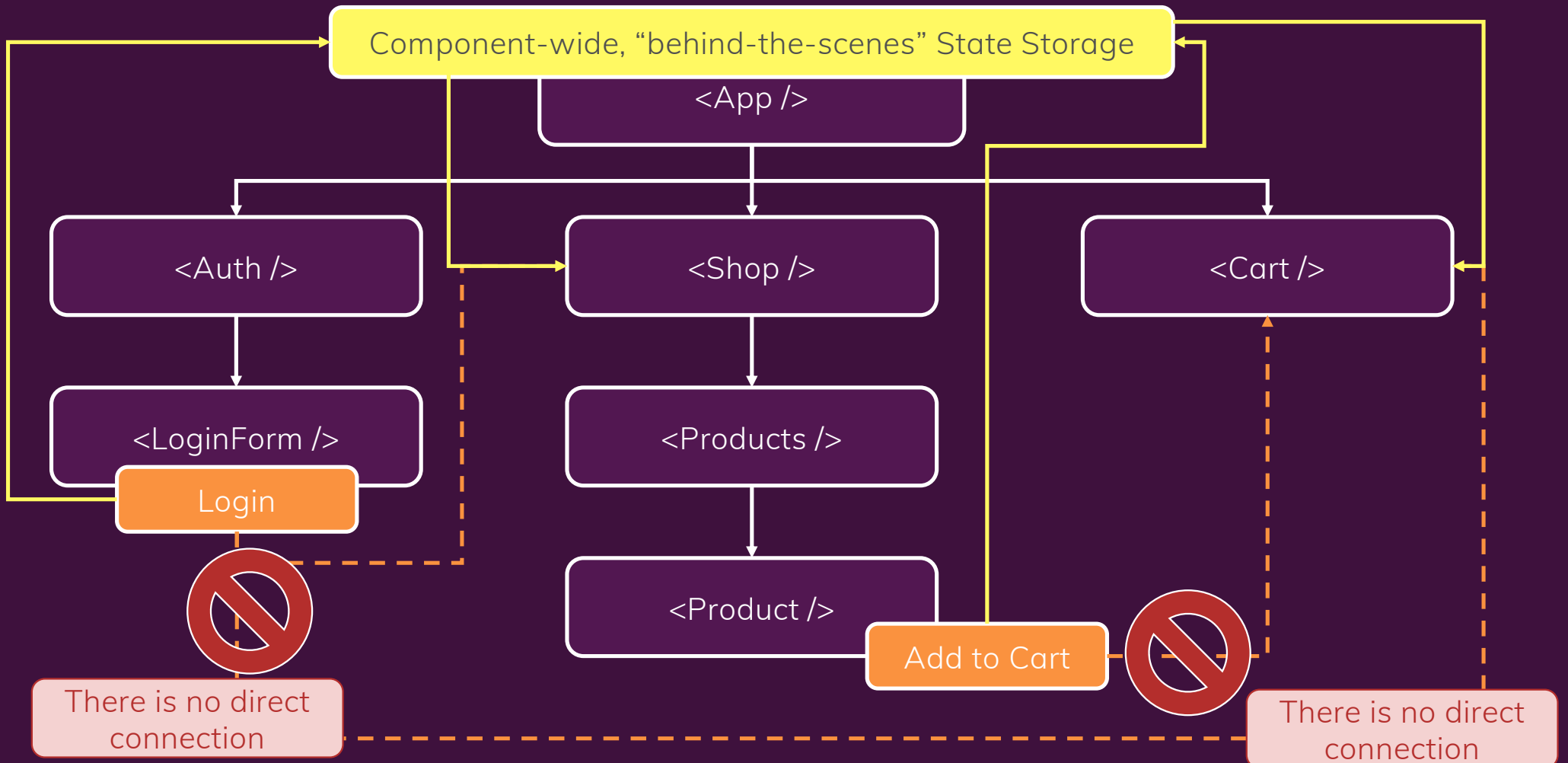
Can be helpful if you have more complex state updates

Component Trees & Component Dependencies

Instead: Use
Props & Functions
passed via Props



Context to the Rescue!



Context Limitations

React Context is **NOT optimized** for high frequency changes!



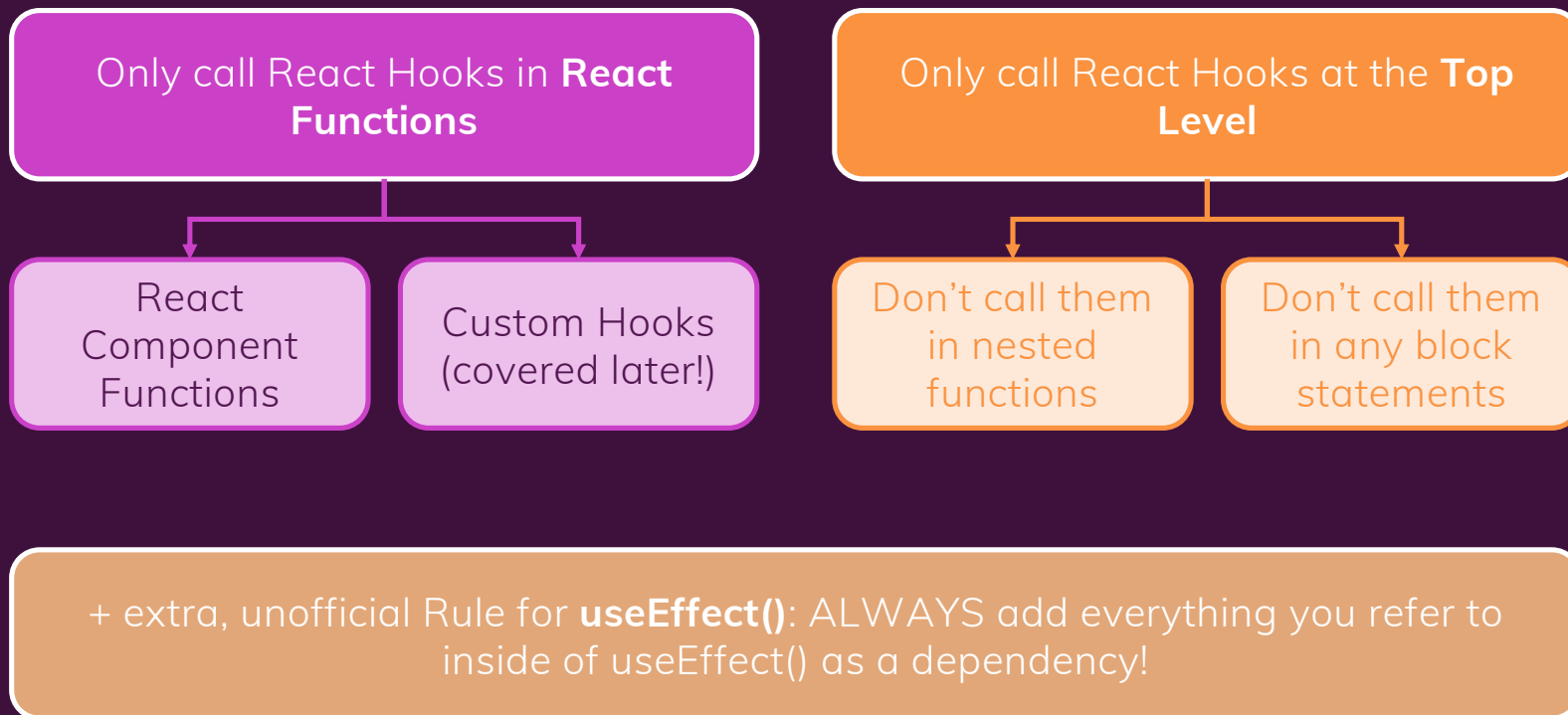
We'll explore a better tool for that, later

React Context also **shouldn't be used to replace ALL** component communications and props

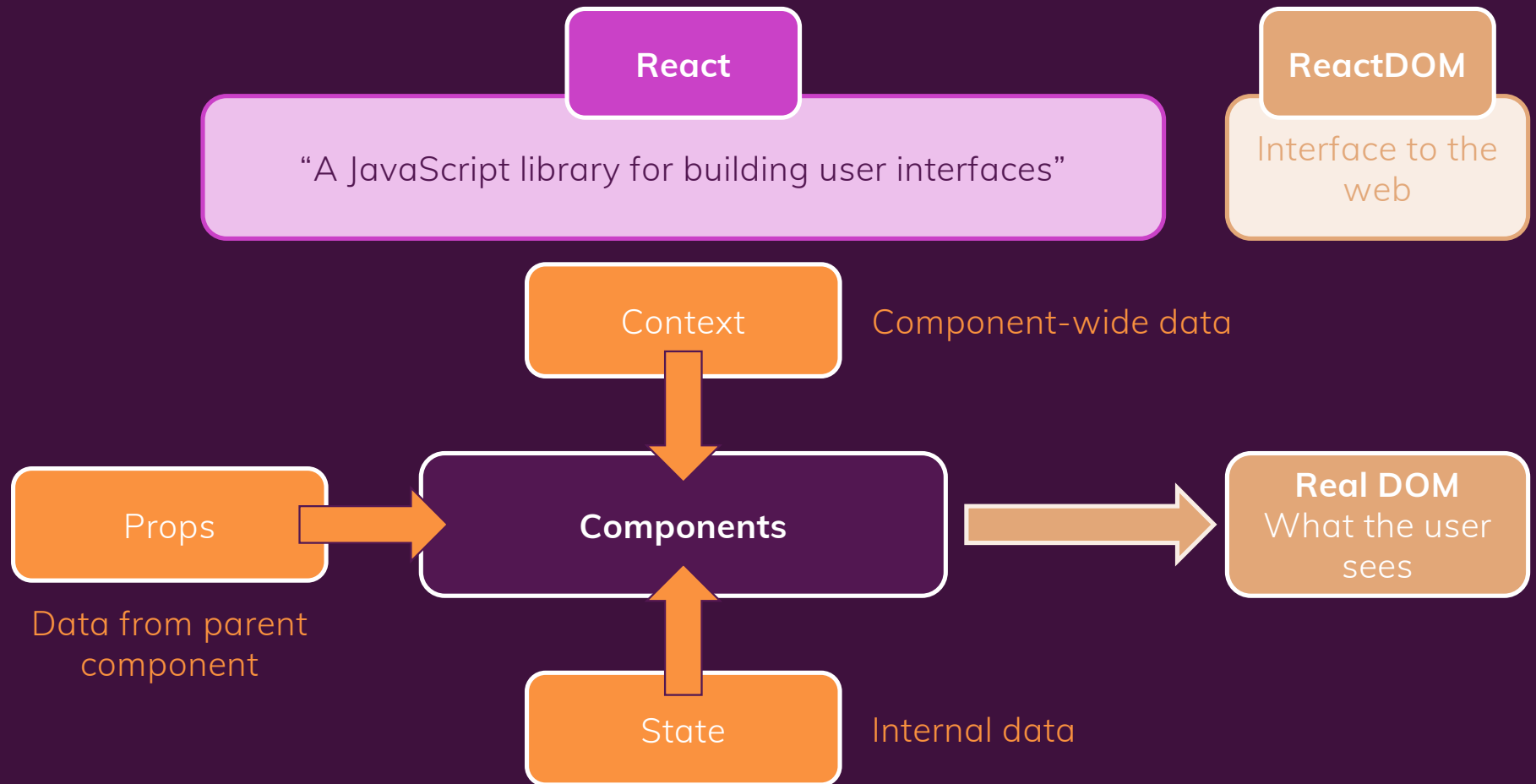


Component should still be configurable via props and short "prop chains" might not need any replacement

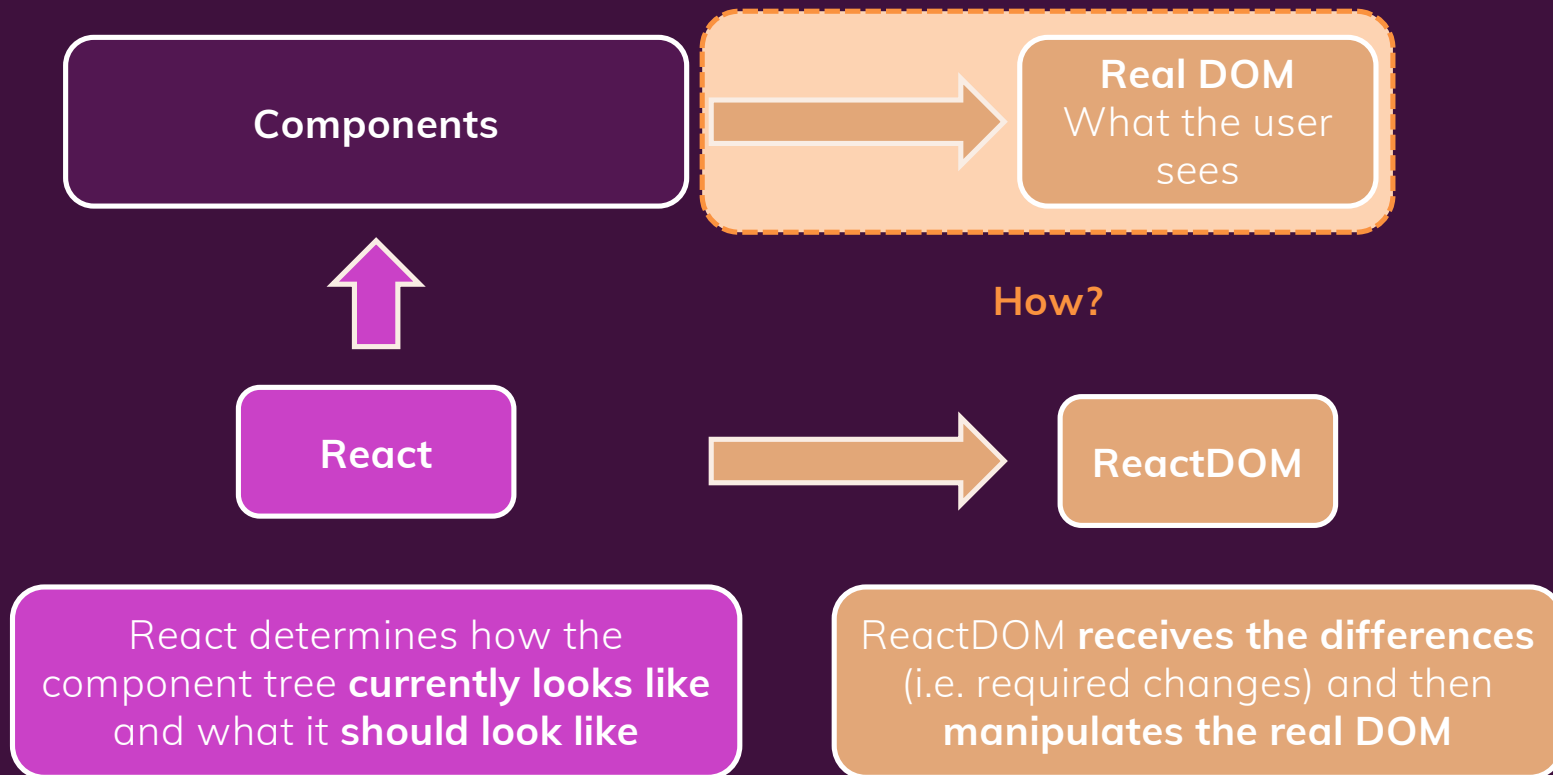
Rules of Hooks



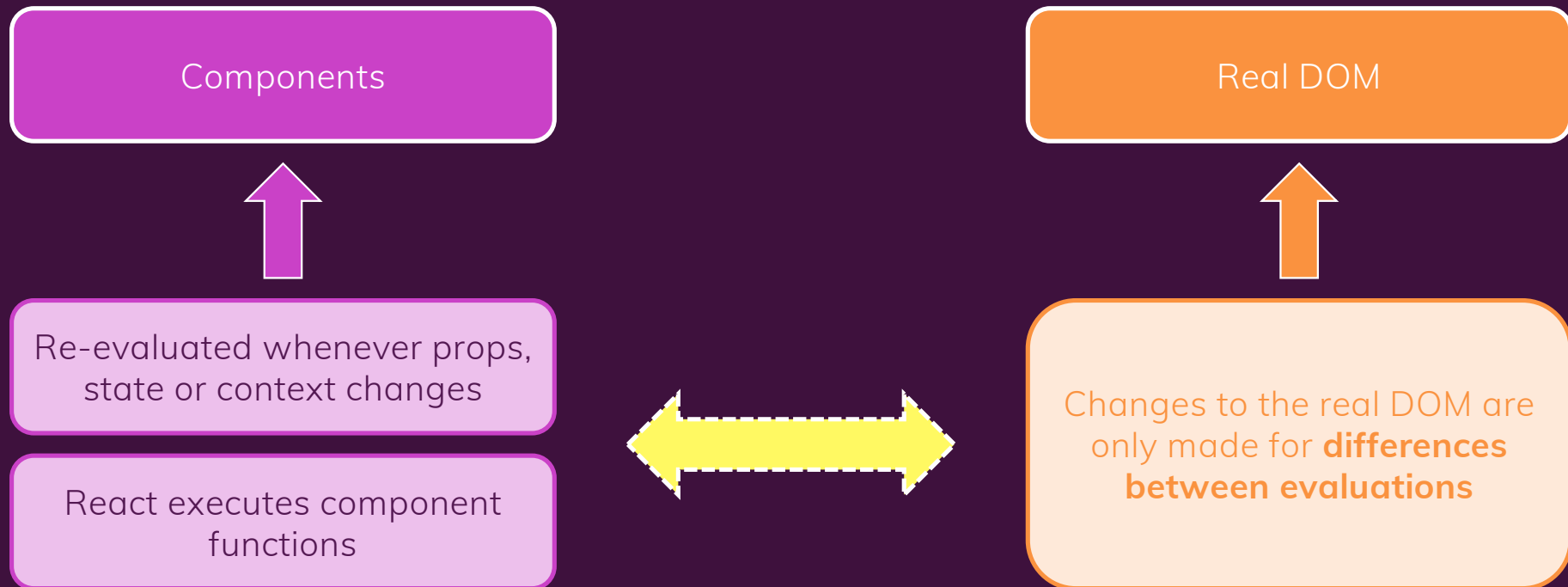
How Does React Work?



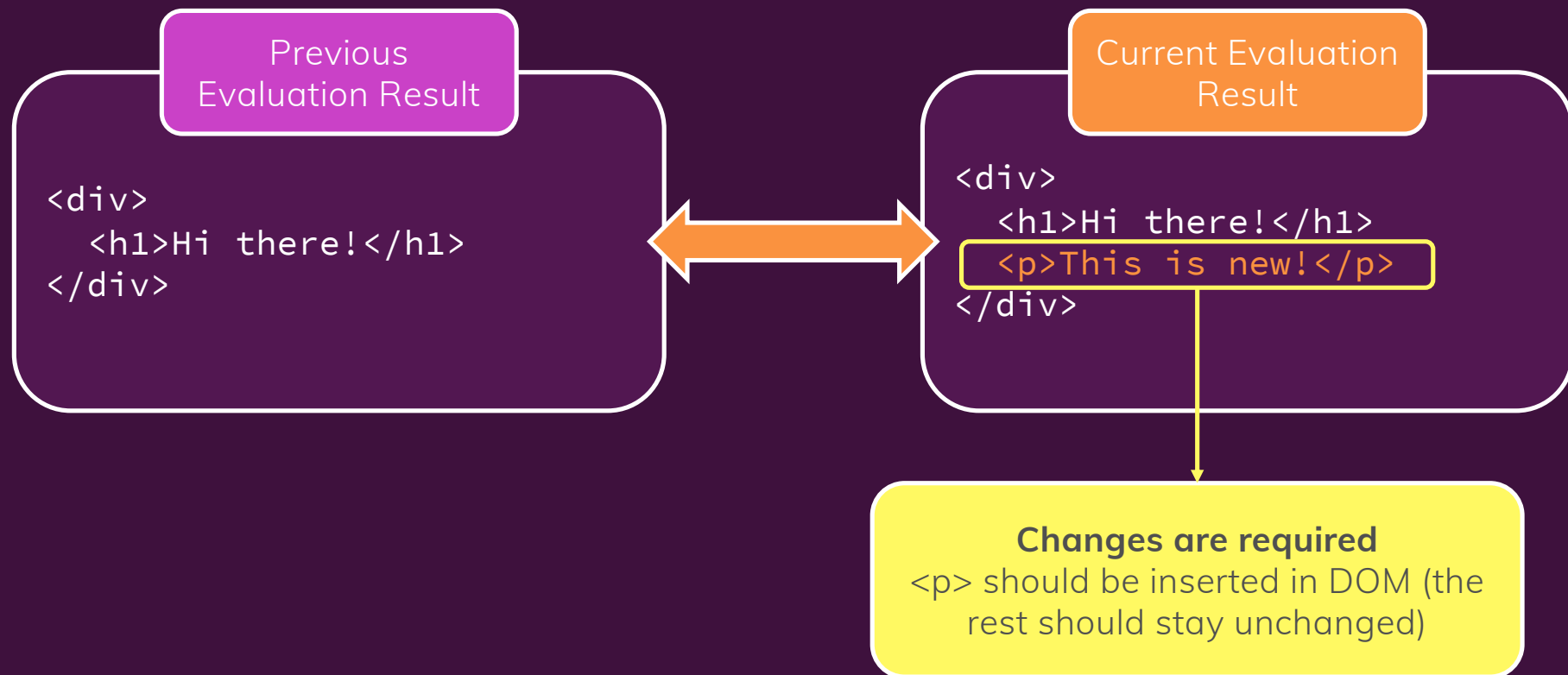
How Does React Work?



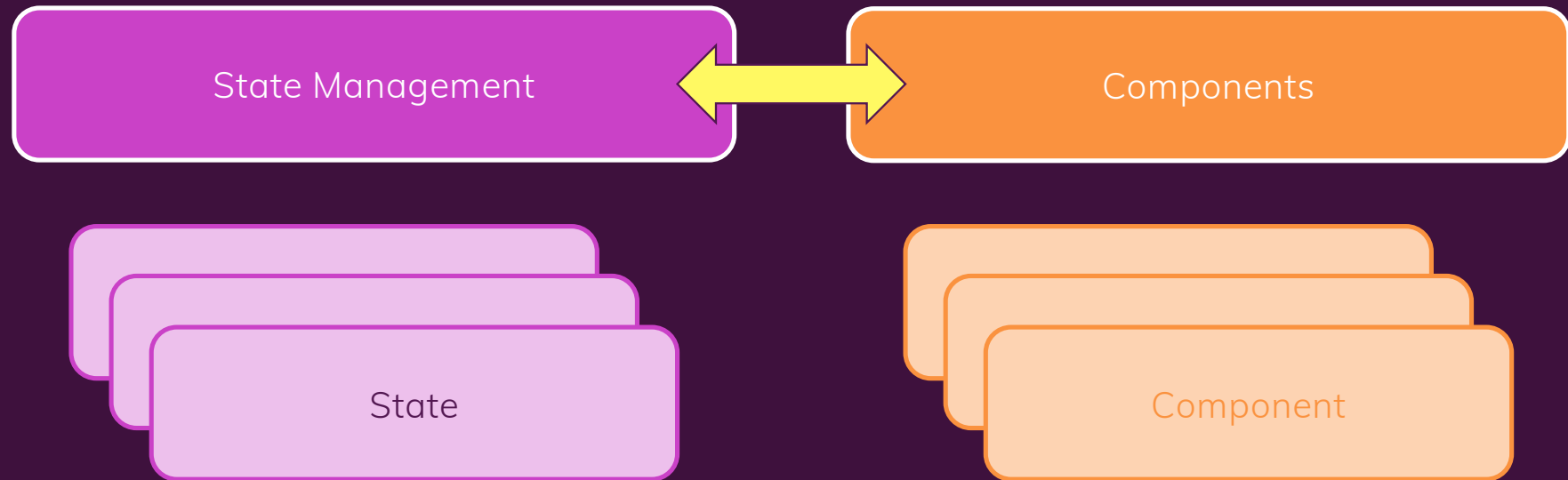
Re-Evaluating Components !== Re-Rendering the DOM



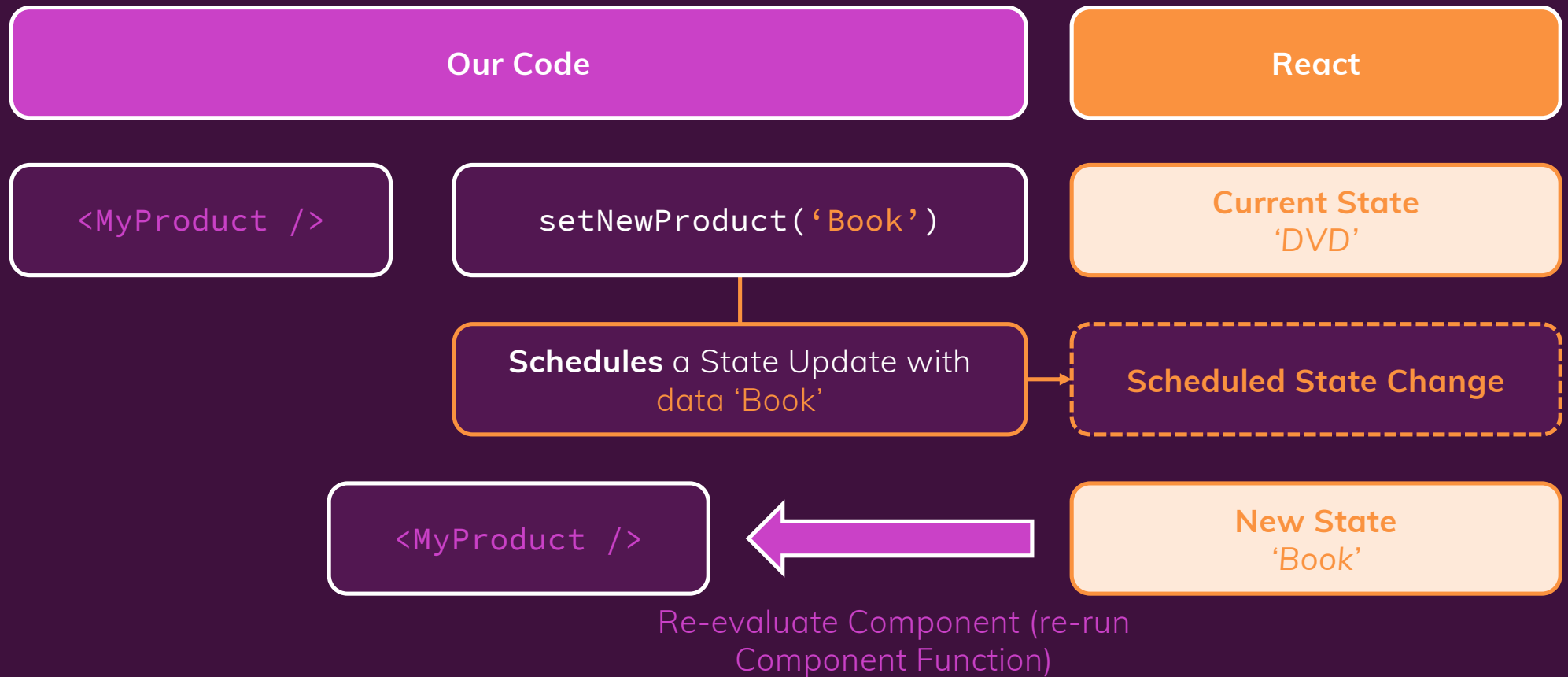
Virtual DOM Diffing



Components & State



State Updates & Scheduling



State Updates & Scheduling

React

Current State
'DVD'

Scheduled State Change

Scheduled State Change

New State
'Book'

Multiple updates can be
scheduled at the same time!

Class-based Components: An Alternative To Functions

Default & Most Modern Approach!

Functional Components

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

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

Was Required In The Past

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

Traditionally (**React < 16.8**), 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-based Component Lifecycle

Side-effects in Functional Components: **useEffect()**

Class-based Components can't use React Hooks!

`componentDidMount()`

Called once component mounted
(was evaluated & rendered)

`useEffect(..., [])`

`componentDidUpdate()`

Called once component updated
(was evaluated & rendered)

`useEffect(..., [someValue])`

`componentWillUnmount()`

Called right before component is
unmounted (removed from DOM)

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

You **don't have to use**
Functional Components
– it is fine to use Class-
based Ones instead

Class-based vs. Functional Components

Prefer functional components

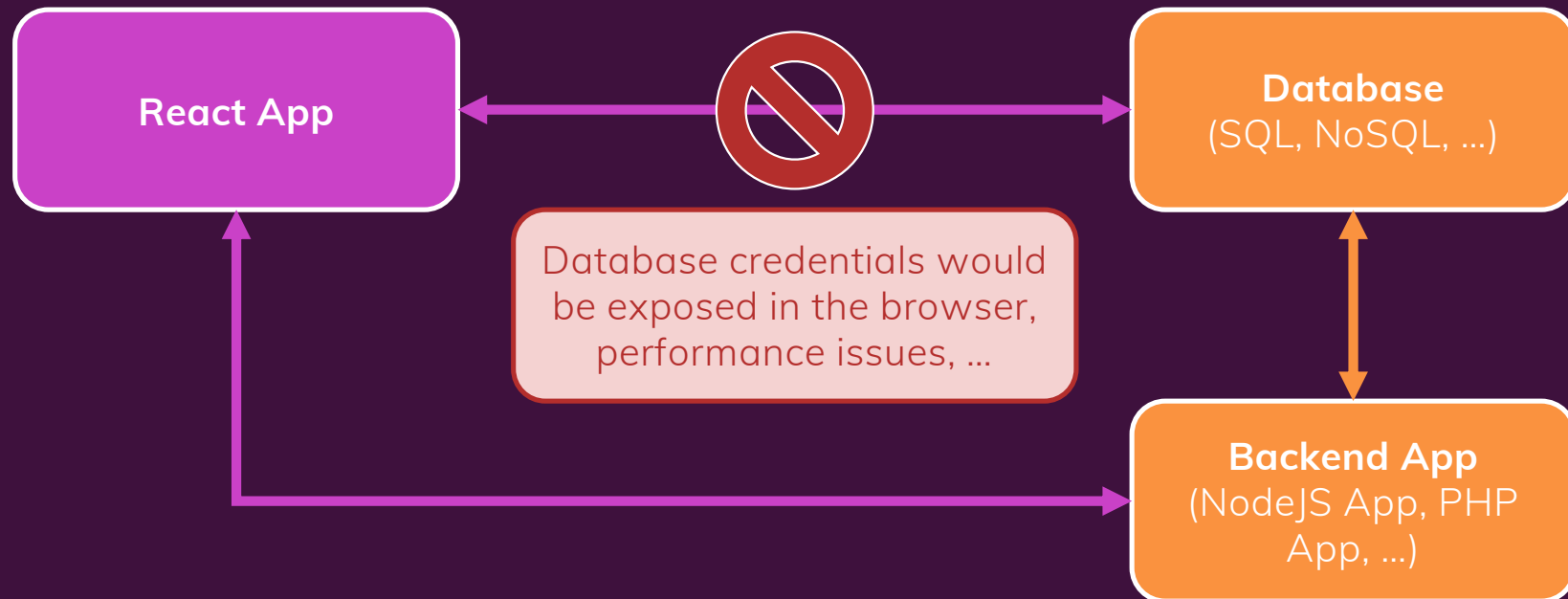
Use class-based if...

...you prefer them

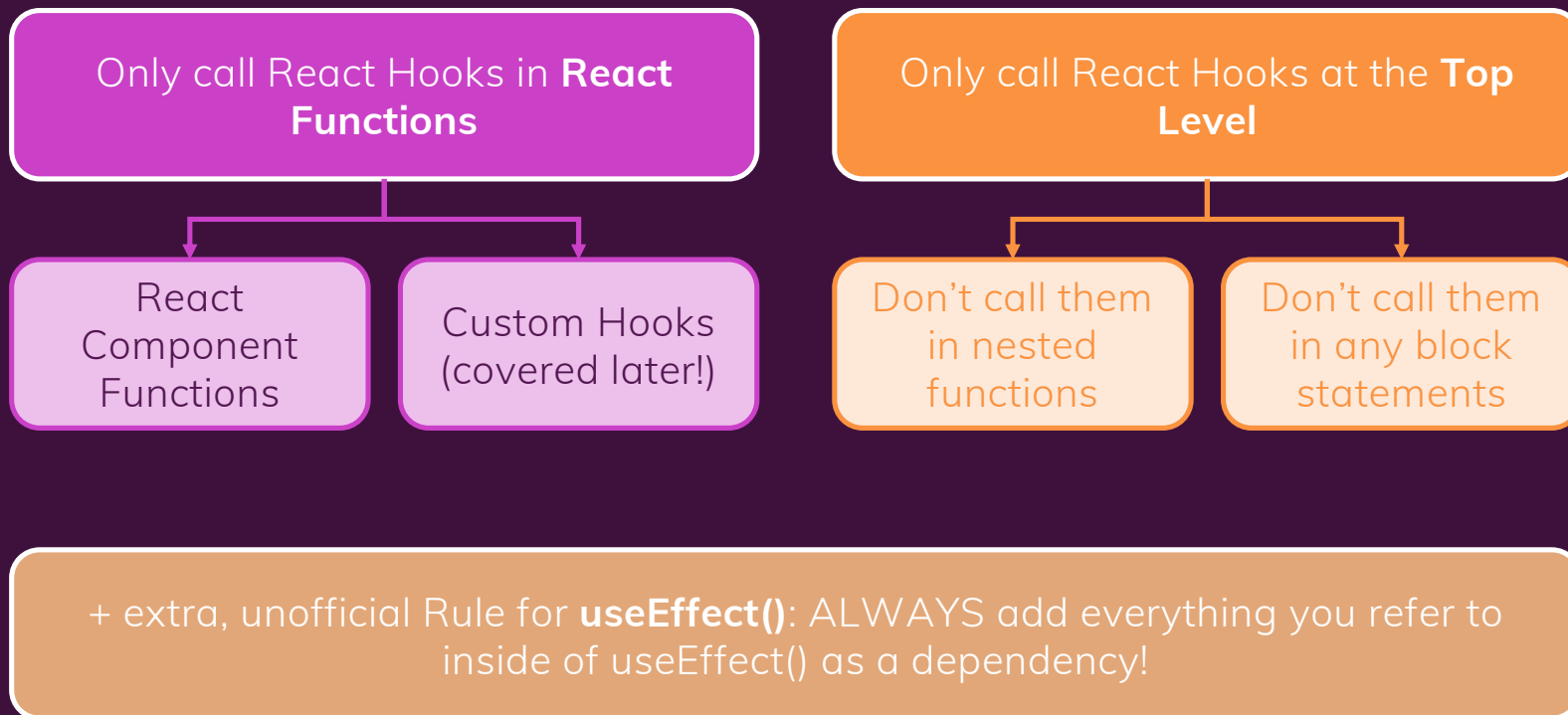
...you're working on an
existing project or in a team
where they're getting used

...you build an "Error
Boundary"

Browser-side Apps Don't Directly Talk To Databases



Rules of Hooks



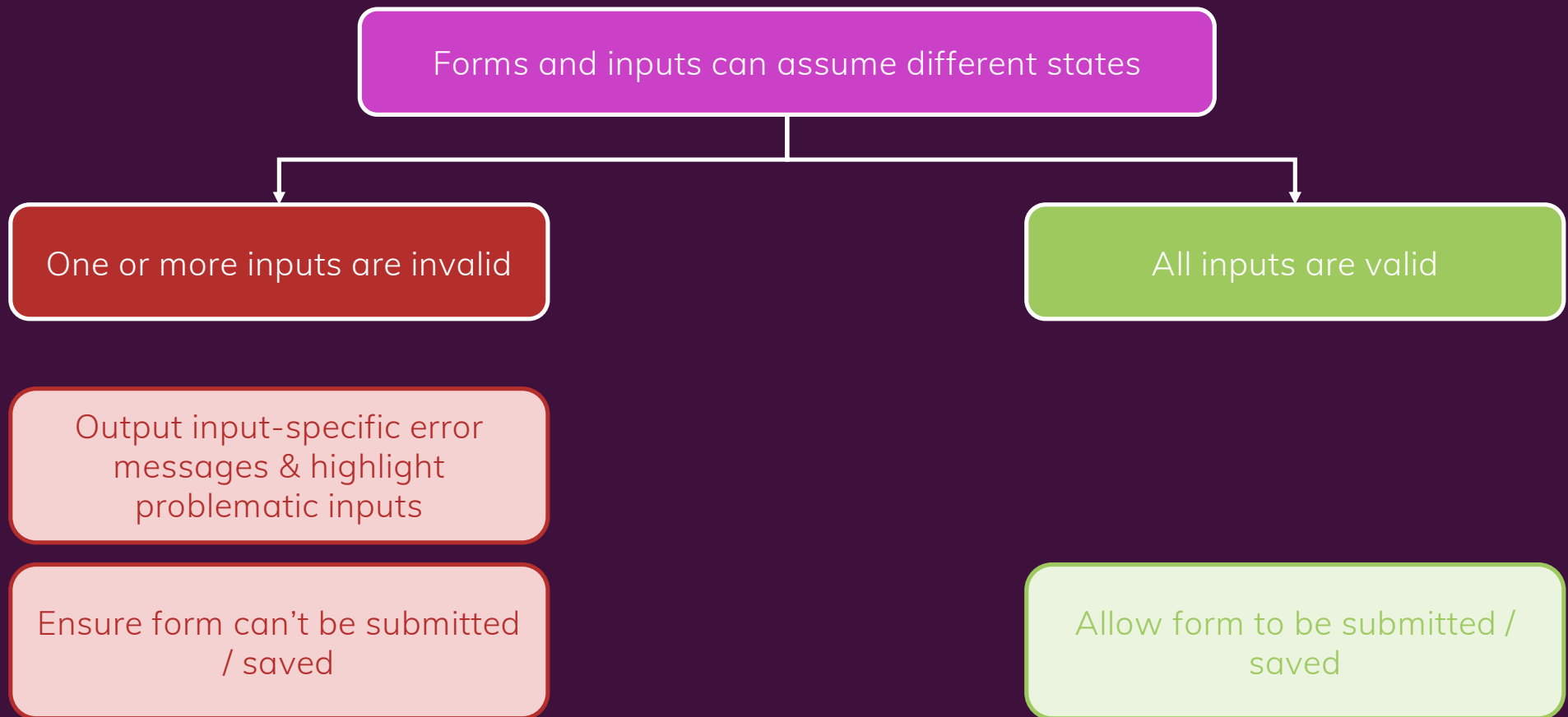
What are “Custom Hooks”?

Outsource **stateful** logic into **re-usable functions**



Unlike “regular functions”, custom hooks can use other React hooks and React state

What's Complex About Forms?



When To Validate?

When form is **submitted**

Allows the user to enter a valid value before warning him / her

Avoid unnecessary warnings but maybe present feedback "too late"

When a input is **losing focus**

Allows the user to enter a valid value before warning him / her

Very useful for untouched forms

On **every keystroke**

Warns user before he / she had a chance of entering valid values

If applied only on invalid inputs, has the potential of providing more direct feedback

What is “Redux”?

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

What Is Cross-Component / App-Wide State?

Local State

State that belongs to a single component

E.g. listening to user input in a input field; toggling a "show more" details field

Should be managed component-internal with `useState()` / `useReducer()`

Cross-Component State

State that affects multiple components

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

Requires "prop chains" / "prop drilling"

App-Wide State

State that affects the entire app (most/ all components)

E.g. user authentication status

Requires "prop chains" / "prop drilling"

OR: React Context or Redux

What is “Redux”?

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



Don't we have “React Context” already?

React Context – Potential Disadvantages

Complex Setup / Management

In more complex apps, managing React Context can lead to deeply nested JSX code and / or huge “Context Provider” components

Performance

React Context is not optimized for high-frequency state changes

React Context – Complex Setup

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


React Context – Complex Setup

```
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>  
  )  
}
```

React Context – Performance



sebookmarkage 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.

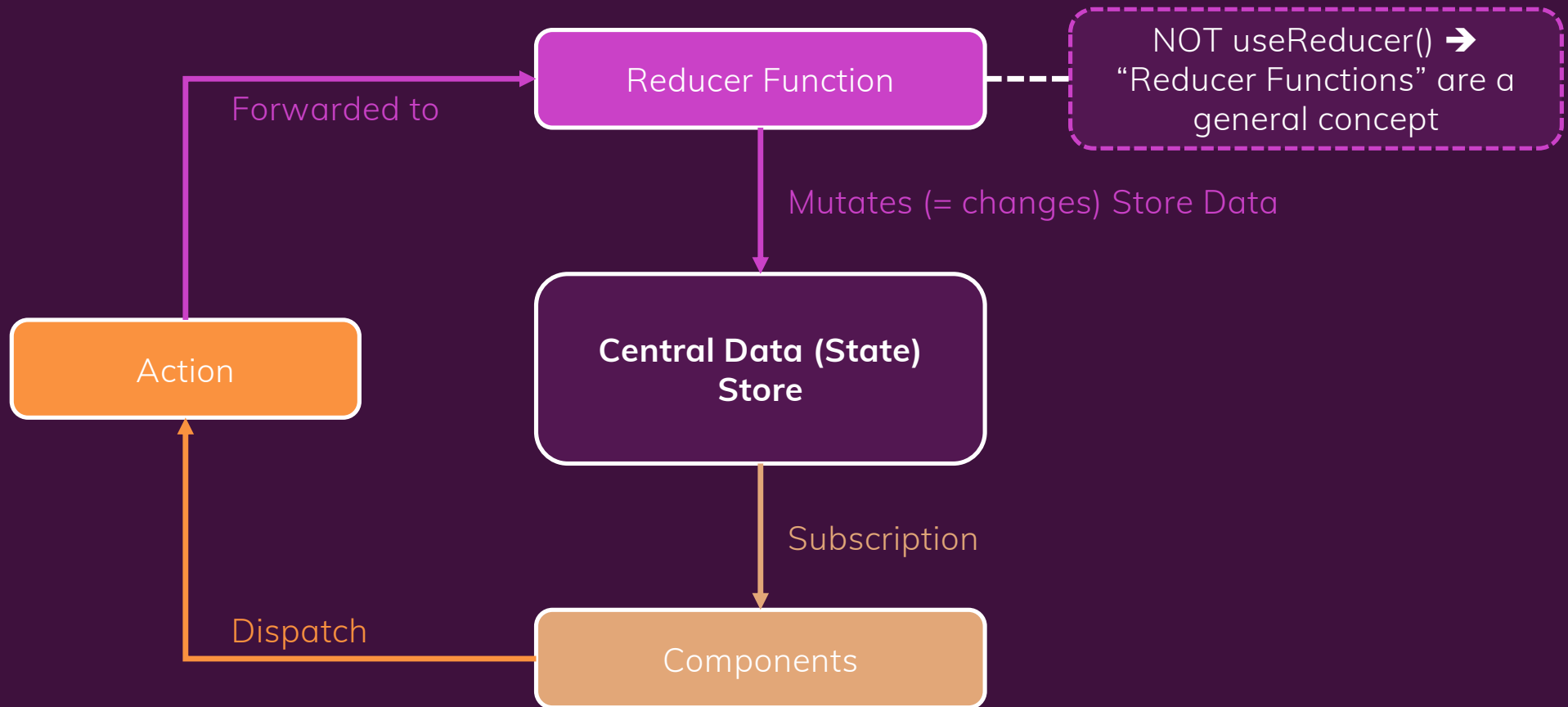


55

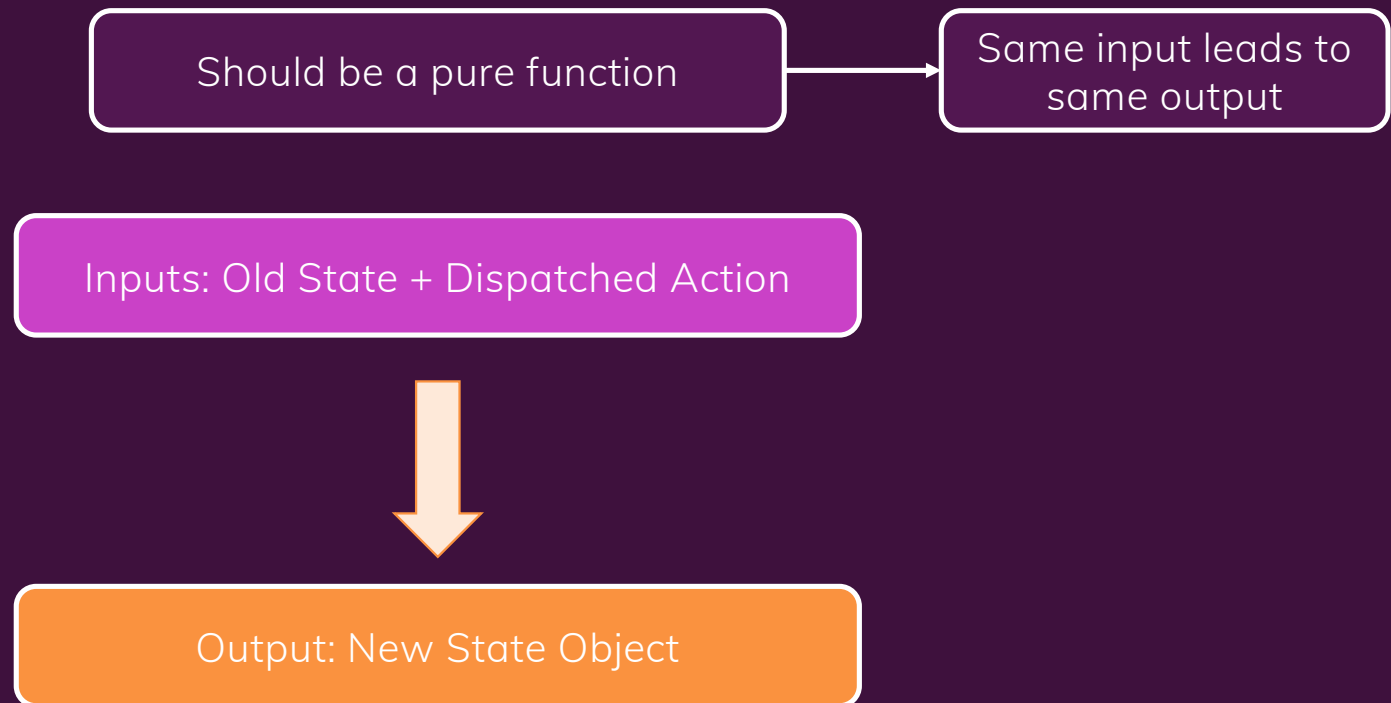


4

Core Redux Concepts



The Reducer Function



The Role Of Immutability

State updates must be done in an immutable way!

Objects and arrays are reference values in JavaScript



Changes made 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 a new state

Side Effects, Async Tasks & Redux

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

Input (Old State +
Action)



Output (New State)

Where should side-effects and async tasks be executed?

Inside the **components** (e.g. `useEffect()`)

Inside the **action creators**

Fat Reducers vs Fat Components vs Fat Actions

Where should our logic (code) go?

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

Prefer Reducers

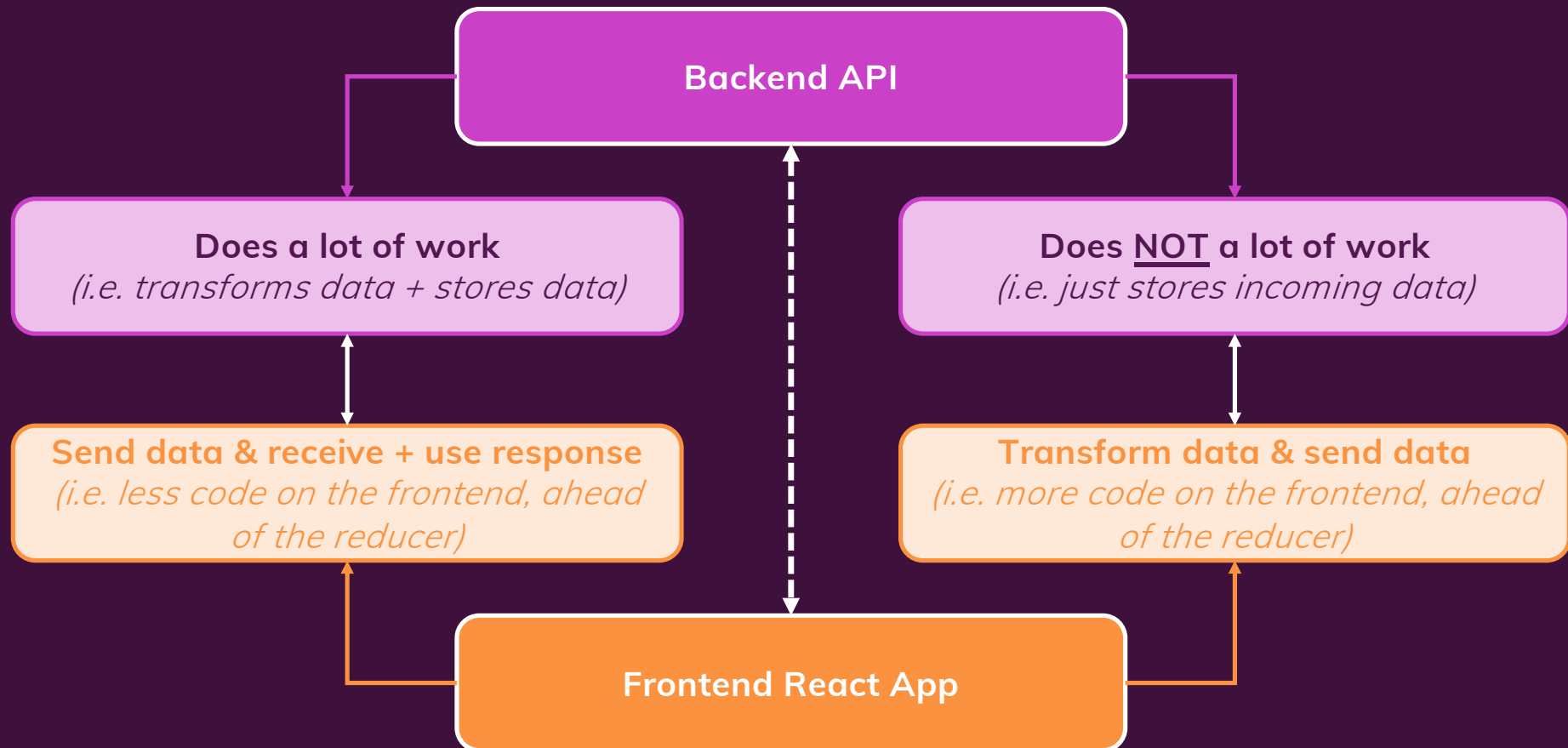
Avoid Action Creators or Components

Async code or code with side-effects

Prefer Action Creators or Components

Never use Reducers

Frontend Code Depends On Backend Code



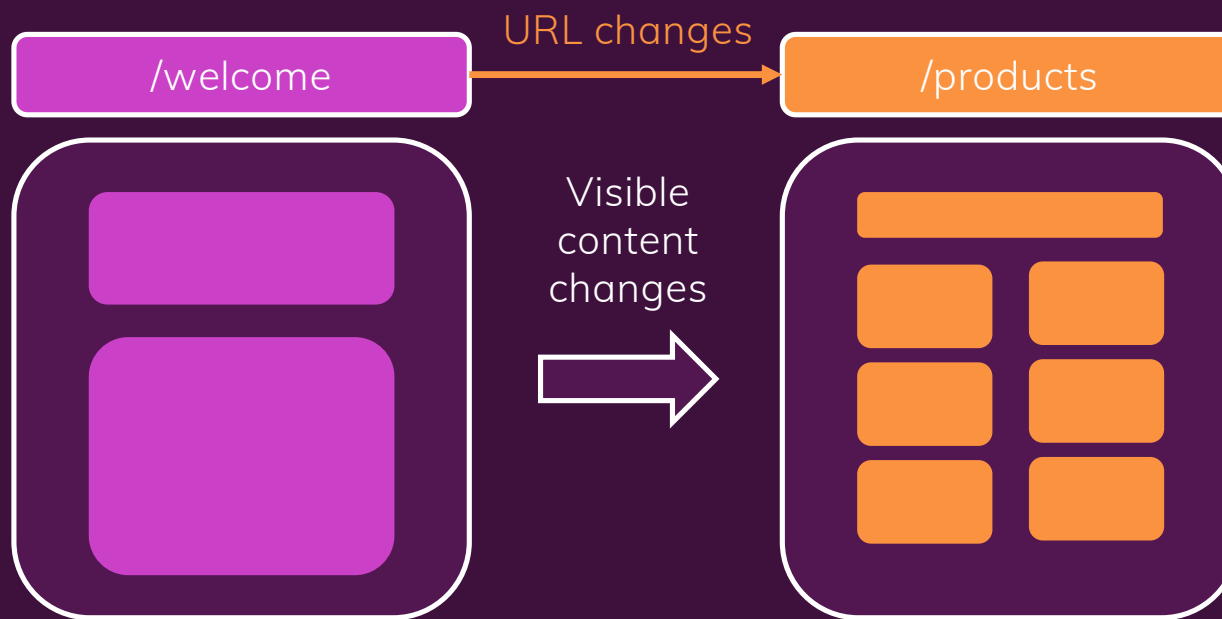
What is a “Thunk”?

A function that delays an action until later

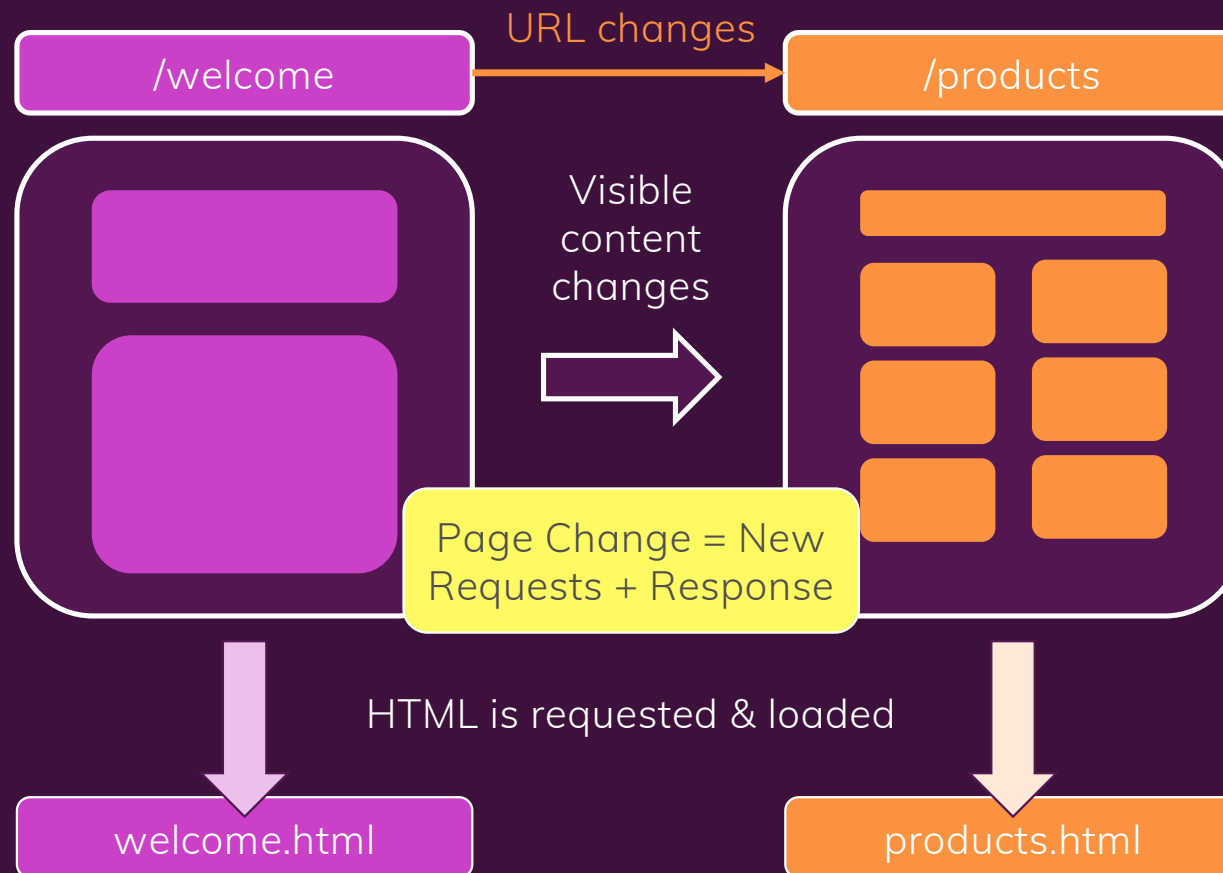


An action creator function that does NOT return the action itself but another function which eventually returns the action

What Is Routing?



Multi-Page Routing



Building SPAs

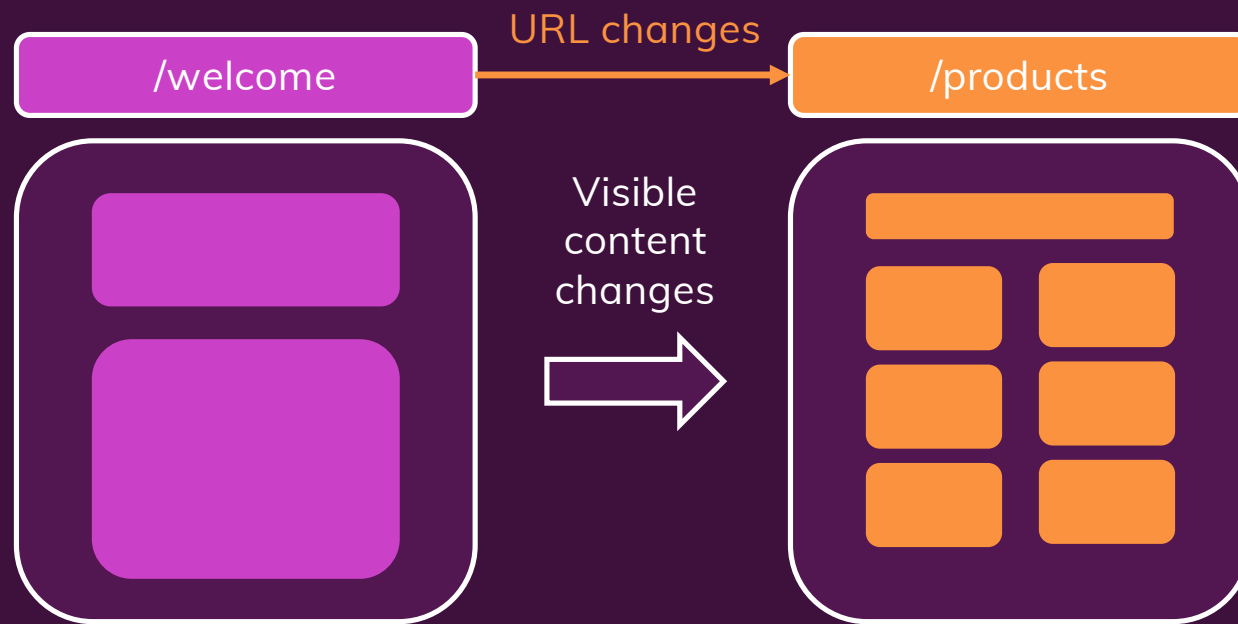
When building complex user interfaces, we typically build **Single Page Applications (SPAs)**

Only one initial HTML request & response

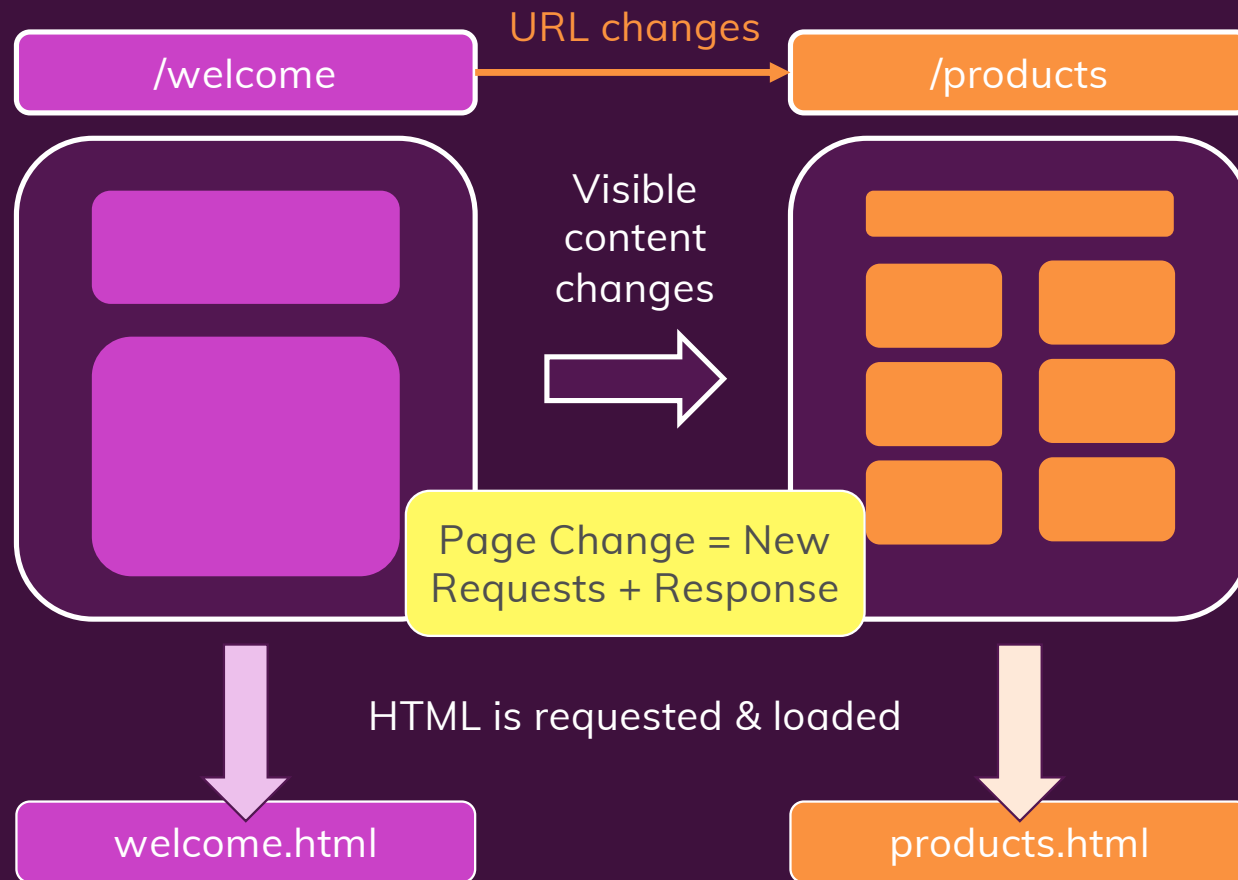
Page (URL) changes are then handled by client-side (React) code

Changes the visible content without fetching a new HTML file

What Is 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



Changes the visible content without fetching a new HTML file

Deployment Steps



Test Code



Optimize Code



Build App for Production



Upload Production Code to Server



Configure Server

Lazy Loading

Load code only when it's needed

A React SPA is a “Static Website”

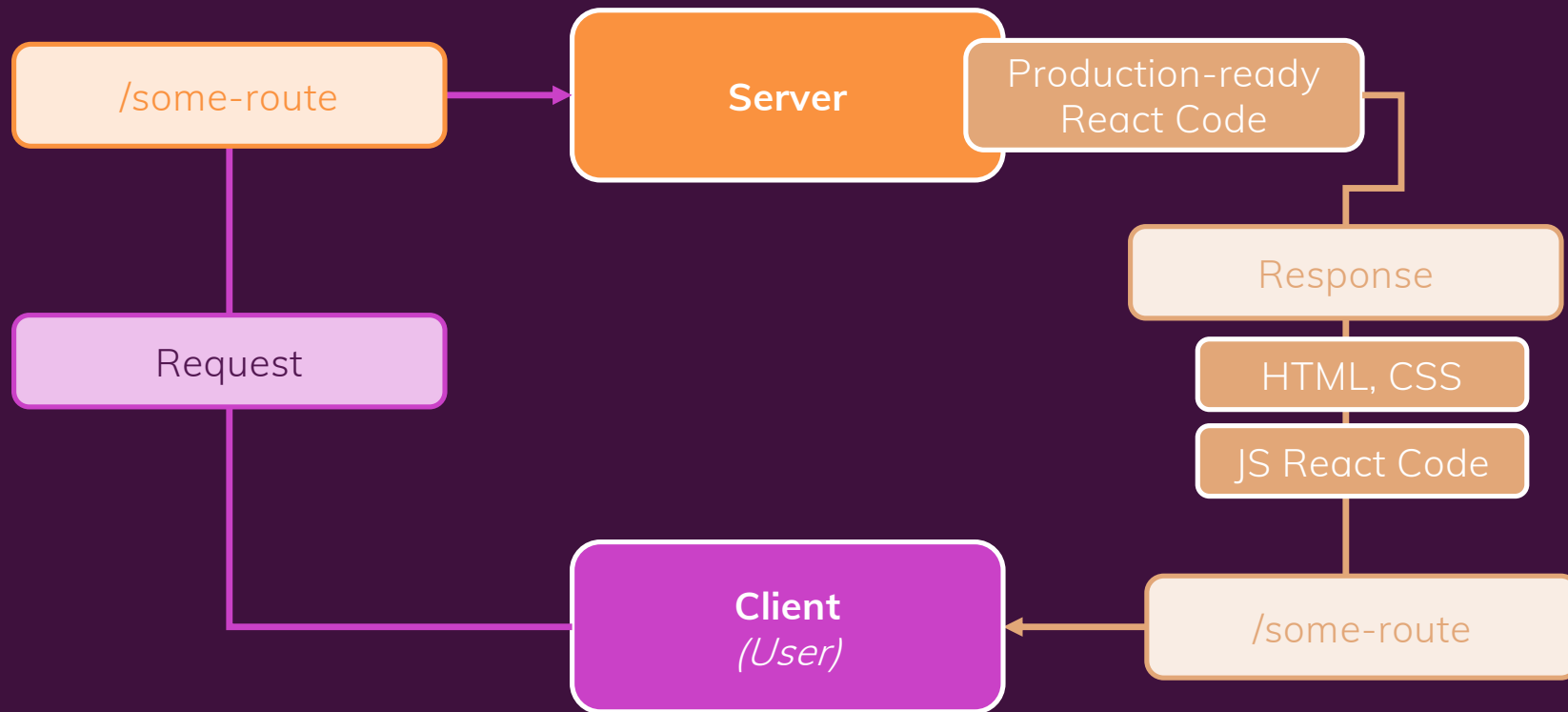
Only HTML, CSS & JavaScript

A React SPA is a “Static Website”

Only HTML, CSS & JavaScript

A Static Site Host Is Needed

Server-side Routing vs Client-side Routing



Server-side Routing with SPAs

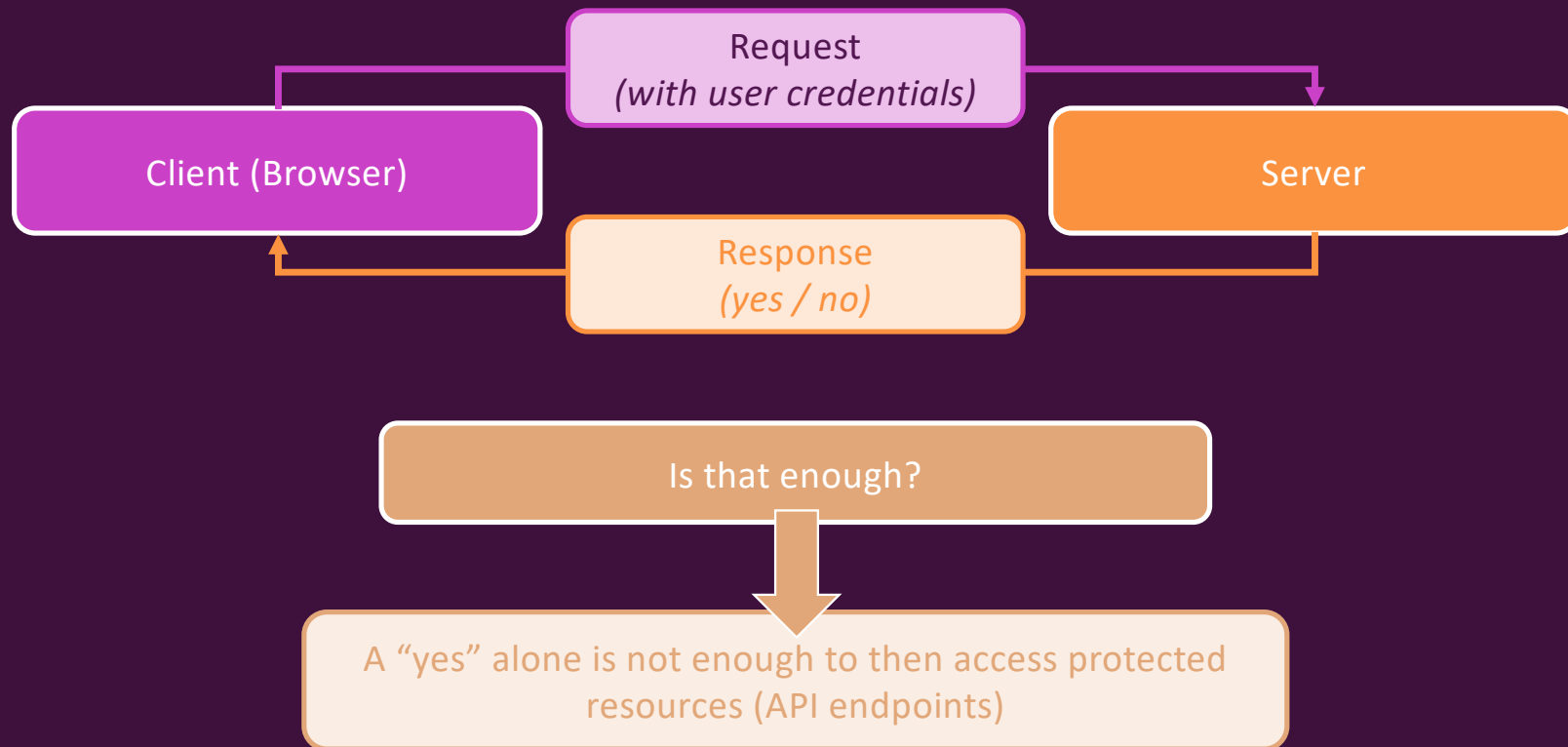
The server must be configured to **always return the index.html** file and ignore the route



Then, React Router (client-side) can take over and render the correct page content

Authentication is needed if content
should be **protected**
(not accessible by everyone)

Getting Permission



How Does Authentication Work?

We can't just save and use the "yes"

We could send a fake "yes" to the server to request protected data

Server-side Sessions

Store unique identifier on server, send same identifier to client

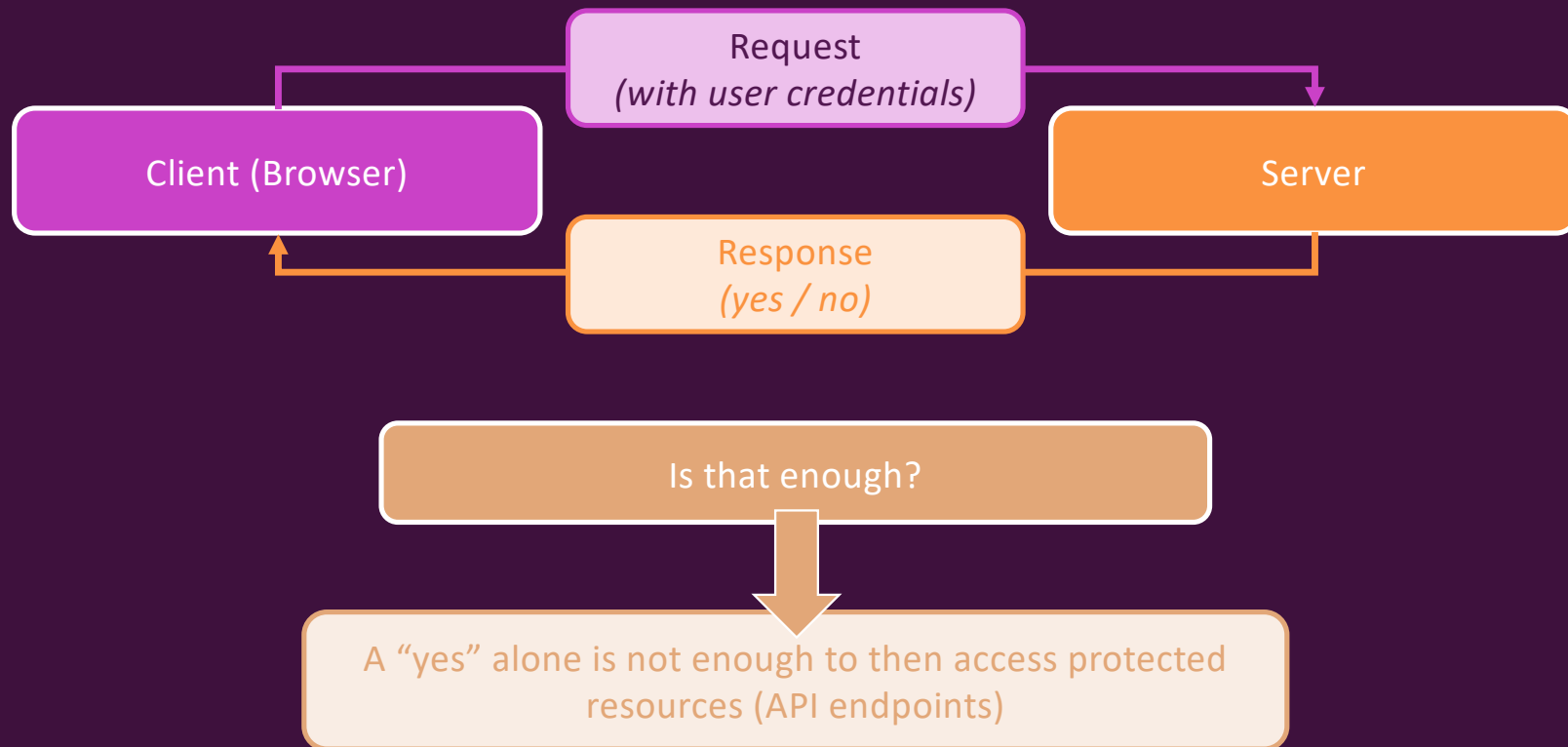
Client sends identifier along with requests to protected resources

Authentication Tokens

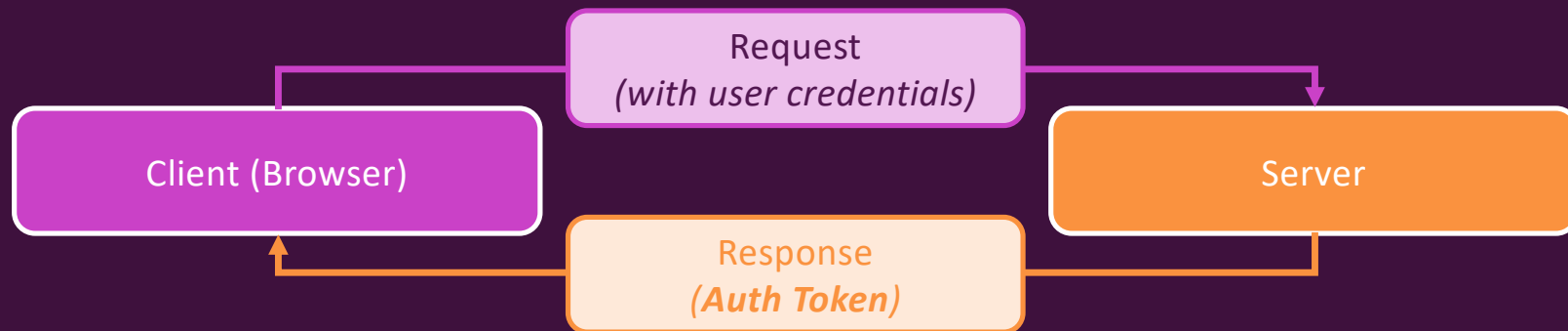
Create (but not store) "permission" token on server, send token to client

Client sends token along with requests to protected resources

Getting Permission



Getting Permission



Deployment Steps



Test Code



Optimize Code



Build App for Production



Upload Production Code to Server



Configure Server

Lazy Loading

Load code only when it's needed

A React SPA is a “**Static Website**”

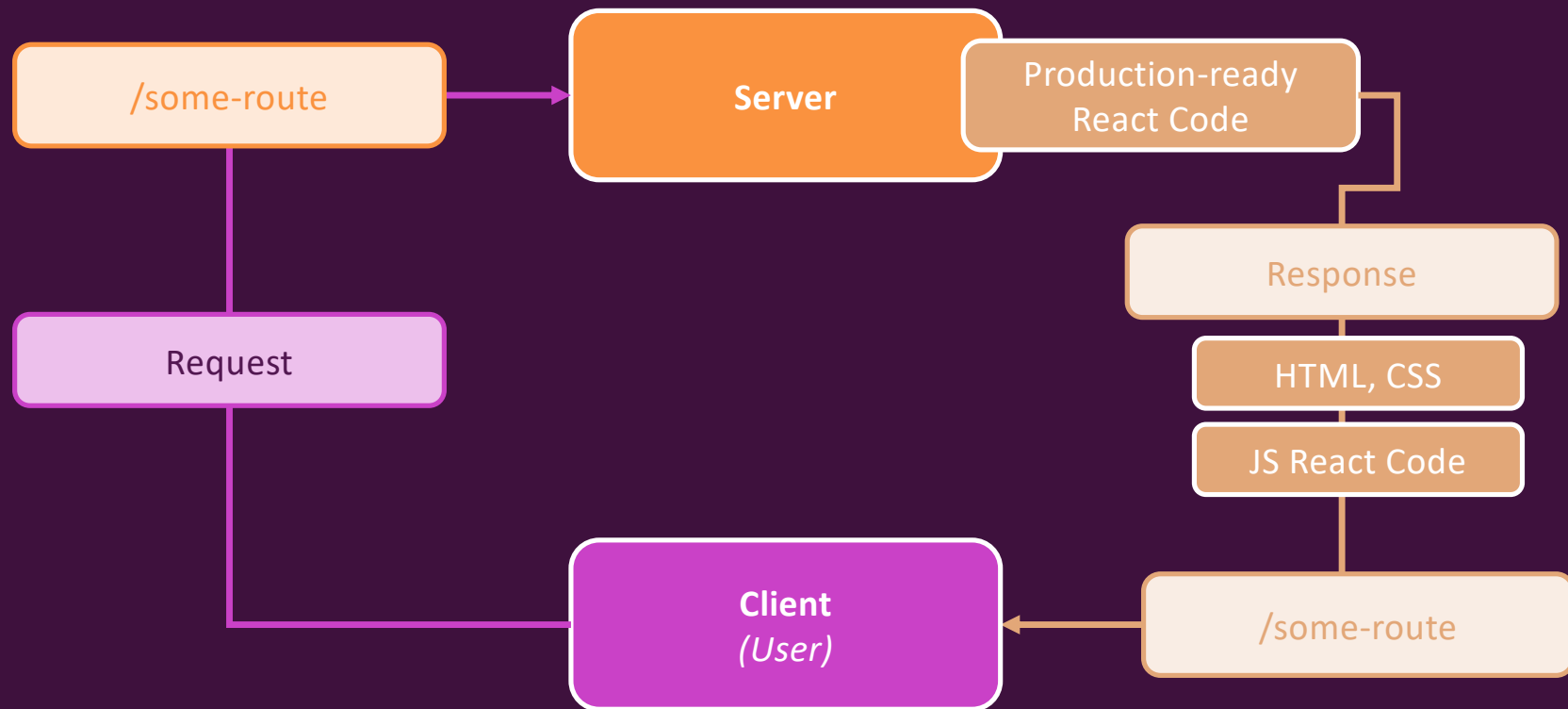
Only HTML, CSS & JavaScript

A React SPA is a “**Static Website**”

Only HTML, CSS & JavaScript

A **Static Site Host** Is Needed

Server-side Routing vs Client-side Routing



Server-side Routing with SPAs

The server must be configured to **always return the `index.html`** file and ignore the route



Then, React Router (client-side) can take over and render the correct page content

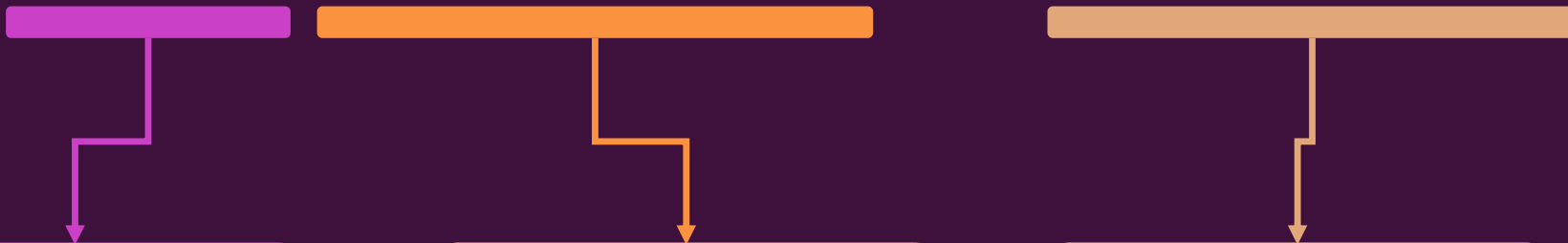
What is NextJS?

The React Framework for Production

A **fullstack** framework for
ReactJS

NextJS solves common
problems and makes
building React apps easier!

The React Framework for Production



You still write React code,
you still build React
components and use
React features (props,
state, context, ...)

NextJS just enhances your
React apps and adds
more features

Lots of built-in features
(e.g. routing) that help you
solve common problems &
clear guidance on how to
use those features

There are certain problems
which you will need to
solve for almost all
production-ready React
apps: NextJS solves those
for you

NextJS – Key Features & Benefits



File-based Routing

Define pages and routes with files and folders instead of code

Less code, less work, highly understandable



Server-side Rendering

Automatic page pre-rendering: Great for SEO and initial load

Blending client-side and server-side: Fetch data on the server and render finished pages



Fullstack Capabilities

Easily add backend (server-side) code to your Next / React apps

Storing data, getting data, authentication etc. can be added to your React projects

What is "Testing"?

Manual Testing

Write Code <> Preview & Test
in Browser

Very important: You see what
your users will see



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

Automated Testing

Code that tests your code

You test the individual building
blocks of your app



Very technical but allows you to
test ALL building blocks at once

Different Kinds Of Automated Tests

Unit 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

Integration Tests

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

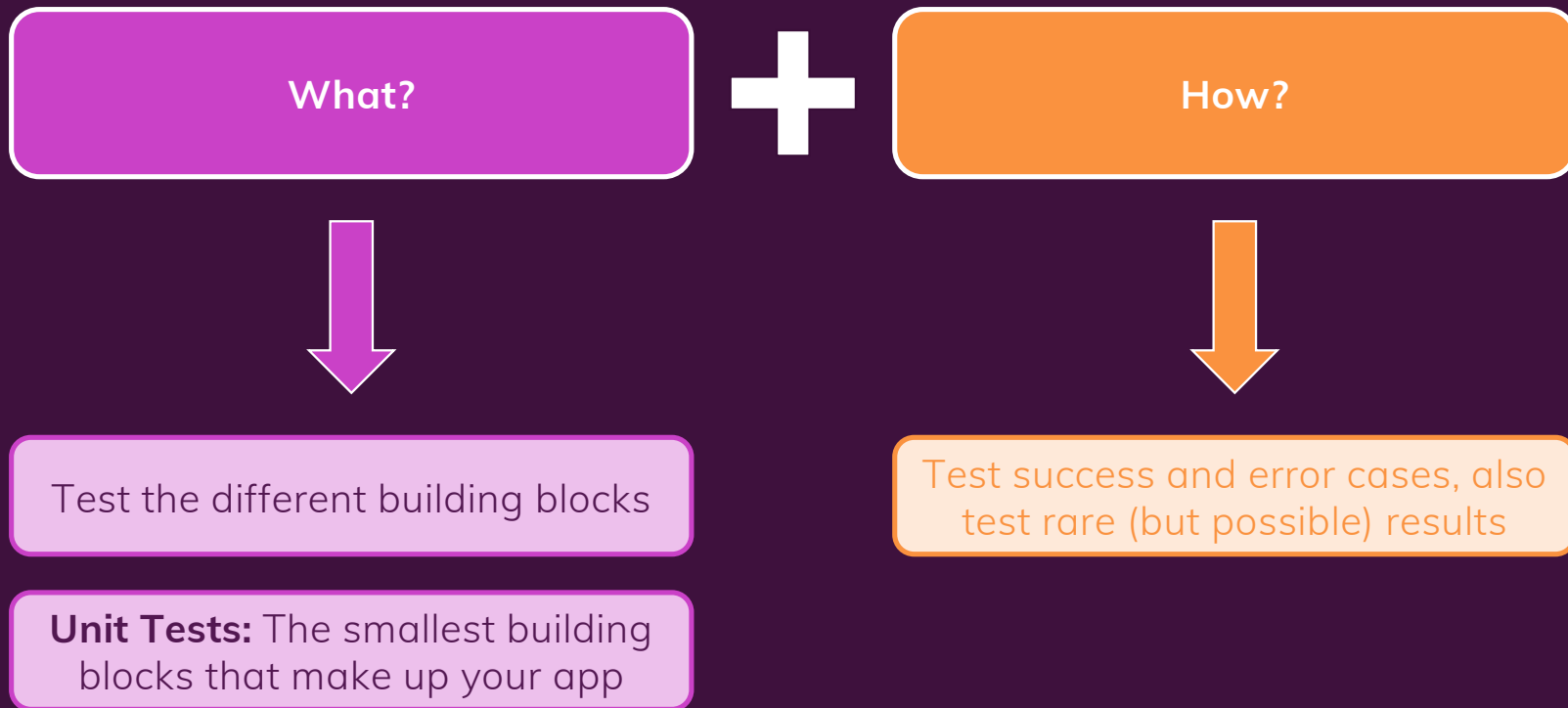
End-to-End (e2e) Tests

Test complete scenarios 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 To Test



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



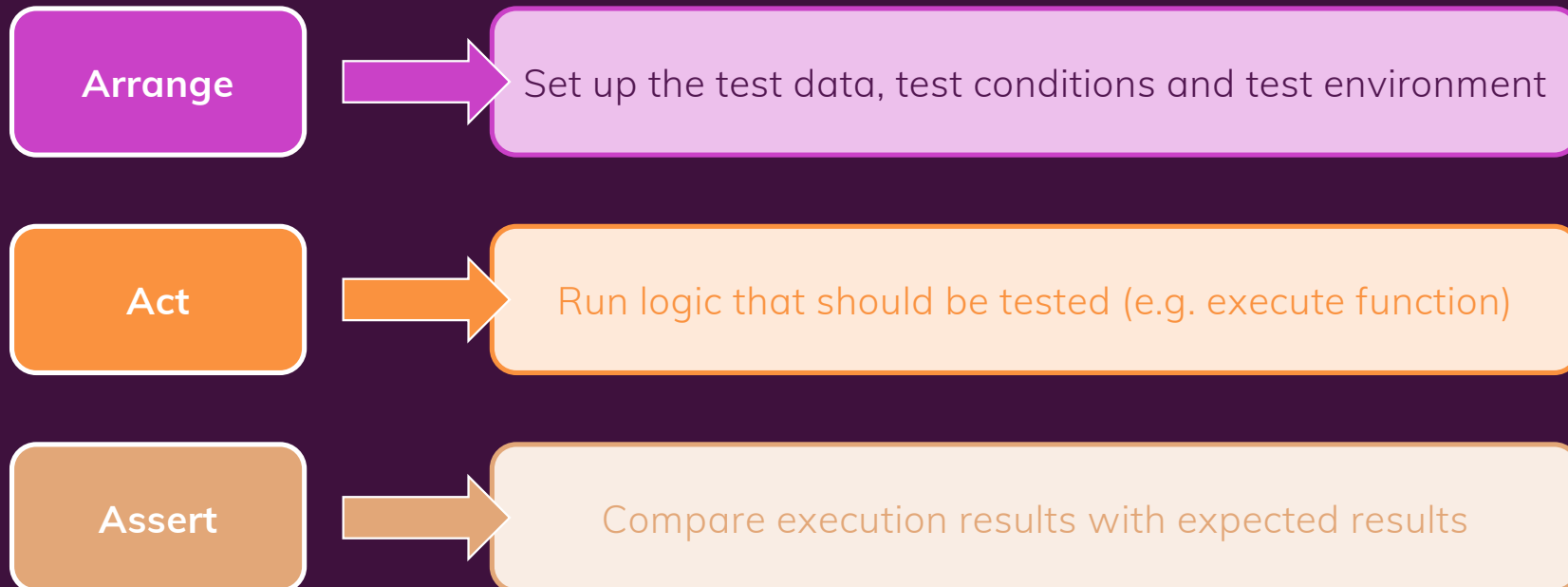
React Testing Library



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



Writing Tests – The Three “A”s



What & Why?

TypeScript is a “**superset**” to
JavaScript

TypeScript adds **static typing** to JavaScript

JavaScript on its own is
dynamically typed