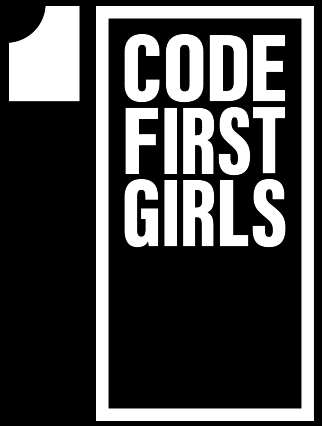
WELCOME TO CFG
YOUR INTRODUCTION TO JAVASCRIPT



TECH SHOULDN'T JUST BE A BOYS CLUB.

COURSE JOURNEY

MODULE 01	MODULE 02	MODULE 03	MODULE 04	MODULE 05	MODULE 06	MODULE 07	MODULE 08
INTRO JAVASCRIPT	CONDITIONS & LOGIC	THE DOM	INTRO REACT	REACT COMPONENTS	STYLING COMPONENTS	STATES & EVENTS	PROJECT PRESENTATION

05 JAVASCRIPT

Functional & Class Components

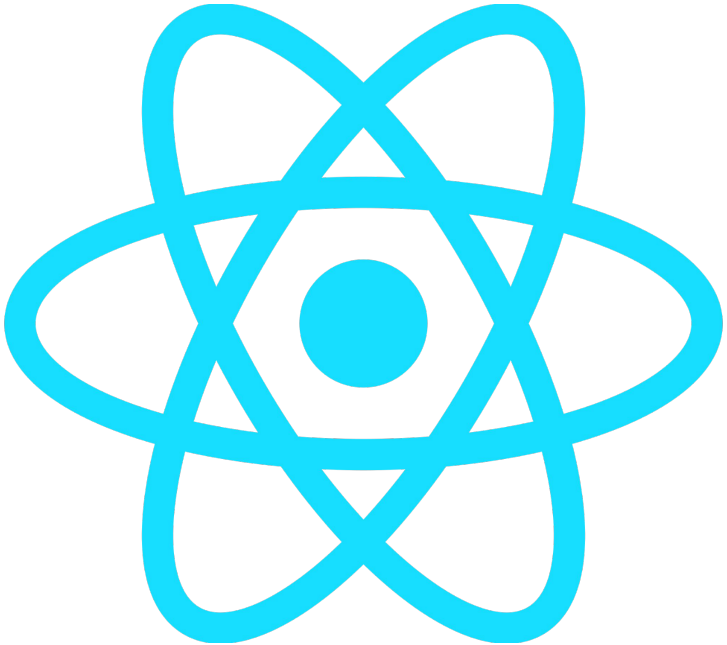
Props

Sub-Components

Project Debrief

LETS RECAP REACT?

- Popular JavaScript framework
- Maintained by Meta
- Uses JSX file formats - JS with HTML
- CSS can be used to add styles
- Package managers (like npm) can add code from external sources
- Some external packages can give you boilerplate code!
- Node.js provides a runtime environment - but it's not essential
- It utilises a virtual DOM for speed
- Enables pages to be broken down into reusable components
- It doesn't provide the "full stack" but a frontend app
- Alternatives are out there!



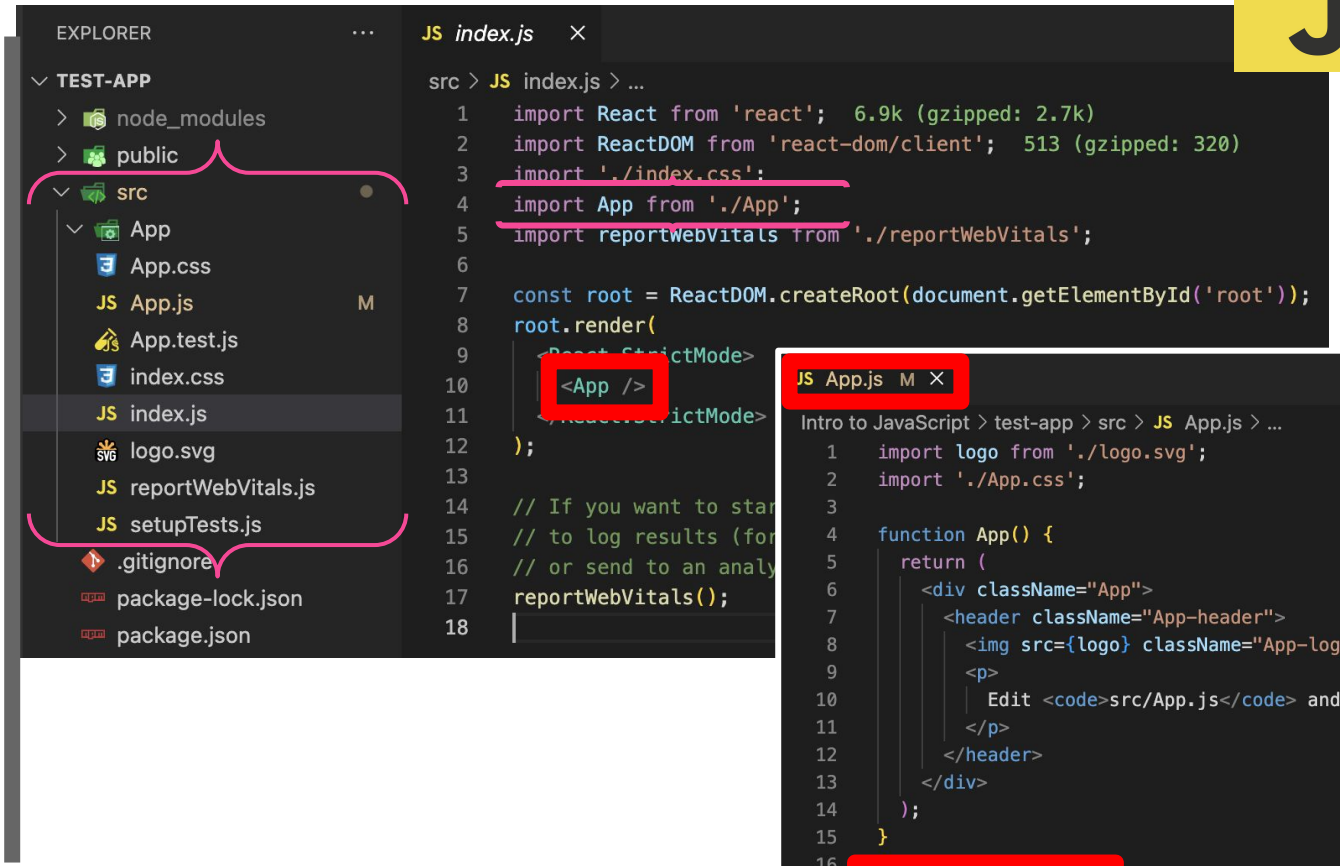
File Structure, Imports & Exports

Easiest Imports:

Import relative files (use `from ./`)

Use the name of the file without the extension (`App` instead of `App.js`)

Use `export default` in the file being imported, this keeps the import nice and simple (`import App`)



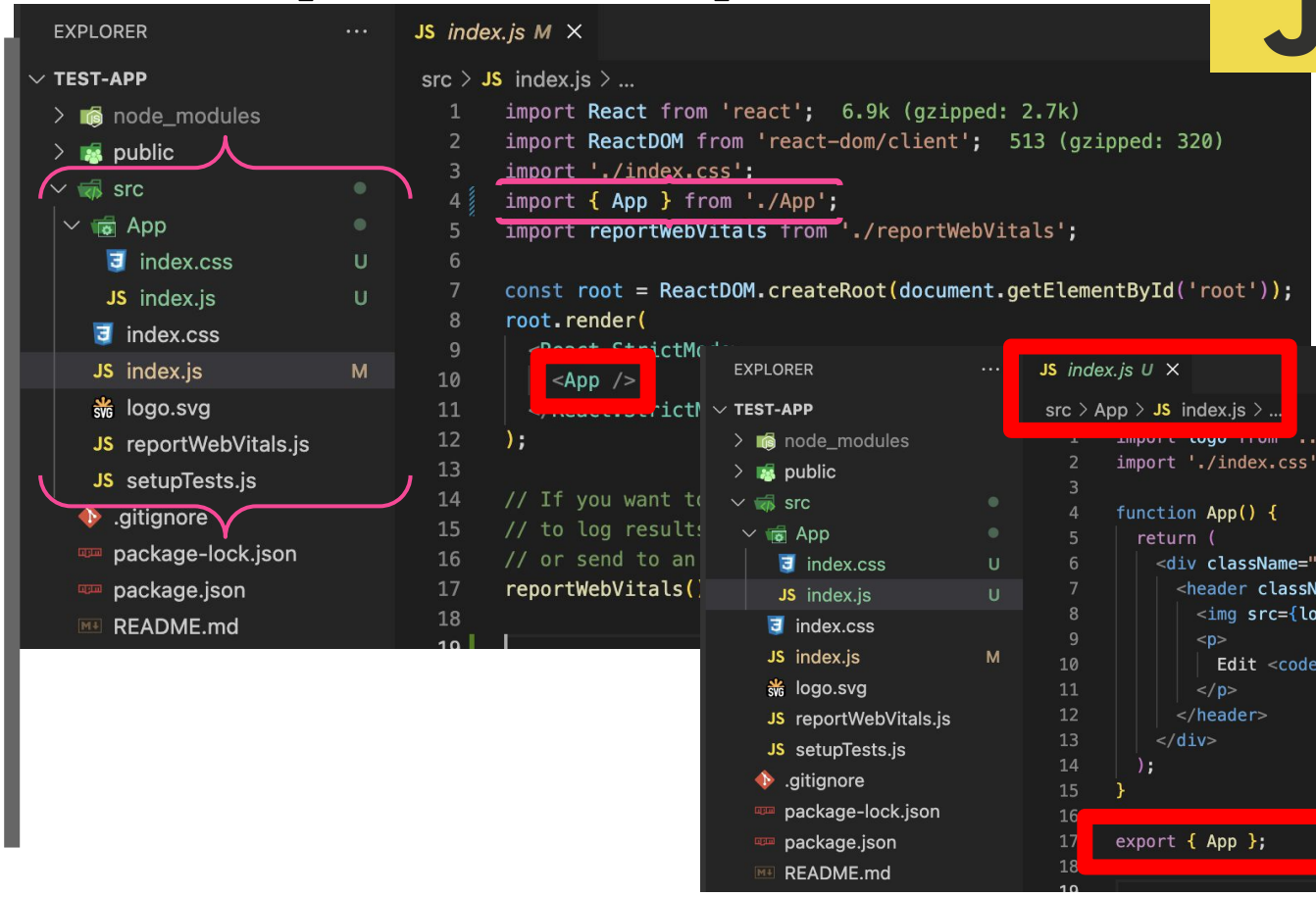
File Structure, Imports & Exports

Harder Imports:

Import relative files (use `from ./`)

Use the name of the folder that contains an index.js (`App` will import `App/index.js`)

Use `export` object in the file being imported, this allows us to import/export multiple things from that one file (`export { App }` and `import { App }`)



NOW LET'S PRACTICE TOGETHER

IMPORTING & EXPORTING

5 MINS

Exercise 1.0 - Make a simple function

Make a new directory and in that folder make a `HelloWorld.js` file. Make a simple function `hello` making sure you call the function at the end of the file. This command should run your code:

```
node HelloWorld.js
```

5 MINS

Exercise 1.1

Export your function using `export { hello }`

Exercise 1.2

Create another file in that same directory and import your function from the previous task.

```
import { hello } from './HelloWorld'
```

Exercise 1.3

Call your function after the import (making sure to remove your function call in `HelloWorld.js`) so you get the same output when you run this file using node.

LETS GET FUNCTIONAL - Expression

React **components can be defined using classes or functions**. Originally it was purely class based but since support for functions was introduced **functions are the recommended method for defining components**.

Ok so functional is the way forward, but there's **TWO** ways to define a function in JavaScript 😟

On the right is an example of a component `Button` defined using a **function expression**. Key parts of this expression include:

- `function` keyword
- Name of the function (`Button`)
- Round brackets - function arguments go here
- Curly brackets - inside goes the actual function code
- `return` keyword - what's being sent back

```
import React from 'react';


function Button() {
 return (
   <button type="button">Click Me</button>
 );
};


export default Button;
```

LETS GET FUNCTIONAL - Arrow

The alternative to using the function expression for defining functions is the **arrow syntax**. If you flick quickly between the two approaches there's few differences, including:

- Assignment to a variable `const Button =` (not essential but allows it to be used later in the file)
- Round brackets - function arguments go here
- The arrow `=>` is what states it's a function
- Curly brackets - inside goes the actual function code
- `return` keyword - what's being sent back

```
import React from 'react';


const Button = () => {
 return (
   <button type="button">Click Me</button>
 );
}


export default Button;
```

Pros:

Simple and single liner functions where the function doesn't need to be stored e.g. `() => 'Hello World'`. For a single line `{}` isn't needed and the code after the arrow is returned by default.

LET'S GET CLASSY

Worth taking note and understanding because functional components weren't originally supported. For **legacy codebase support** you'll need to understand class based components.

Key parts of a class component include:

- Importing `Component` from react - not technically necessary and accessible via `React.Component` but is much more readable.
- `class` keyword
- Name of the class (`Button`)
- `extends` keyword
- Name of the class to build on top of (`Component`) - React provides a parent class with common methods (e.g. `render`)
- `render` method - an internal function that defines what is rendered in the DOM

```
import React, { Component } from 'react';


class Button extends Component {
 render() {
   return <button type="button">Click Me</button>;
 }
}


export default Button;
```

Walkthrough: Creating our first component

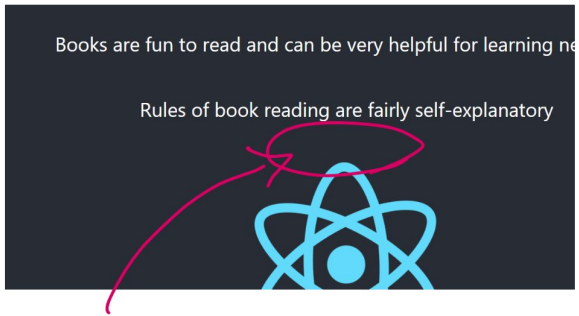
We'll create our own custom button this time, together with your instructor

Create a new file called **Button.js** - we'll write our component code in here

Write this boilerplate in it below! Your instructor will explain what each part does (e.g. A, B, C and D on the right)

You'll need to import your new component into App.js afterwards as a heads up in order for it to be visible!

SKETCH OF WHERE BUTTON SHOULD GO



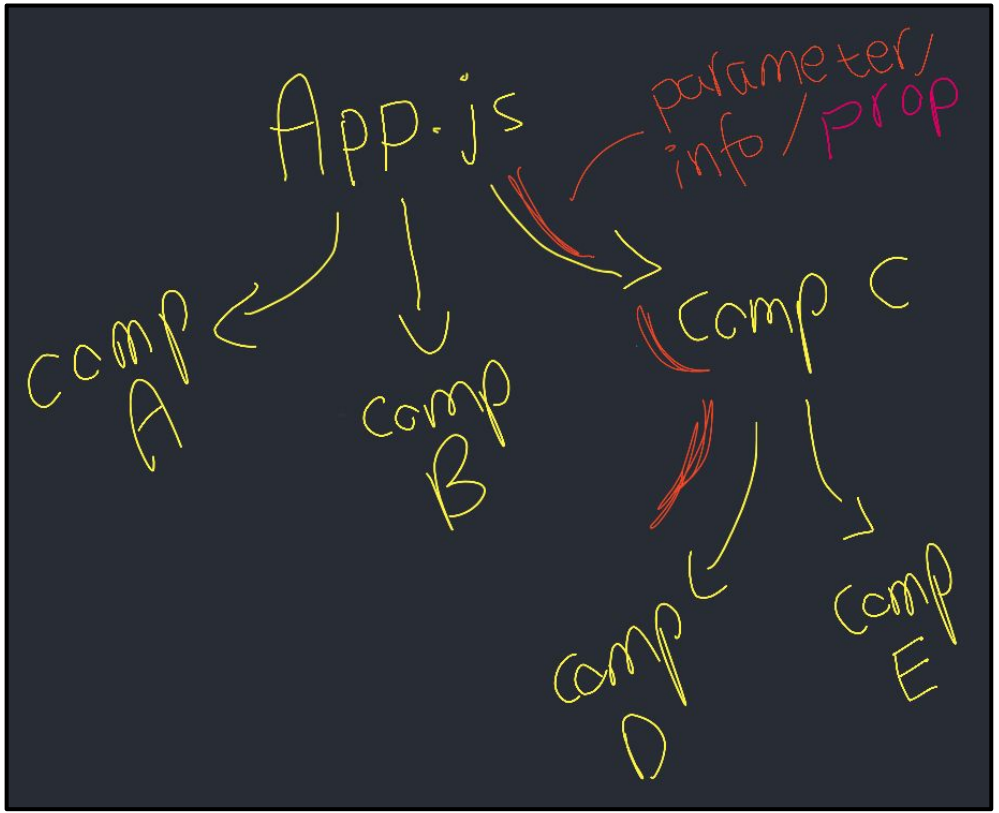
New button component to ideally go here

EXAMPLE OUTPUT


```
1   import './Button.css';
2
3   {/*
4     My imports go here!
5     These are React version
6     of <link> or <script>
7   */}
8
9   function Button() {
10    return (
11      {/* ??? */}
12    );
13  }
14
15  export default Button;
```

IT'S ALL ABOUT THE PROPS

- Components can pass pieces of information to each other - called **props**
- It's extremely similar to how we call functions in Python and pass values to them for use
- We can pass anything, and assign any ID / property name to the lower component - for example:
  nameDoesNotMatter = "Prop *'nameDoesNotMatter'* value**"**
- Note that all components receive a prop object anyway! Adding values to it in the parent component above just means that the object actually has a **propNameID : propValue** this time



Walkthrough: Passing information via props

Make the button customisable instead of hard-coding stuff!

DOT SYNTAX

- All properties add to `props` object
- Access any given property by `props.<prop name>`

```
const Button = (props) => {
 return (
   <button type="button">{ props.message }</button>
 );
}
```

UNPACKING PROPS OBJECT

- All properties add to `props` object
- Unpack required properties to variables in the function scope `const { <prop name> } = props`

```
const Button = (props) => {
 const { message } = props;
 return (
   <button type="button">{ message }</button>
 );
}
```

UNPACKING IN FUNCTION DEFINITION

- Only defined properties are unpacked and set to variables in the function scope

```
const Button = ({ message }) => {
 return (
   <button type="button">{ message }</button>
 );
}
```

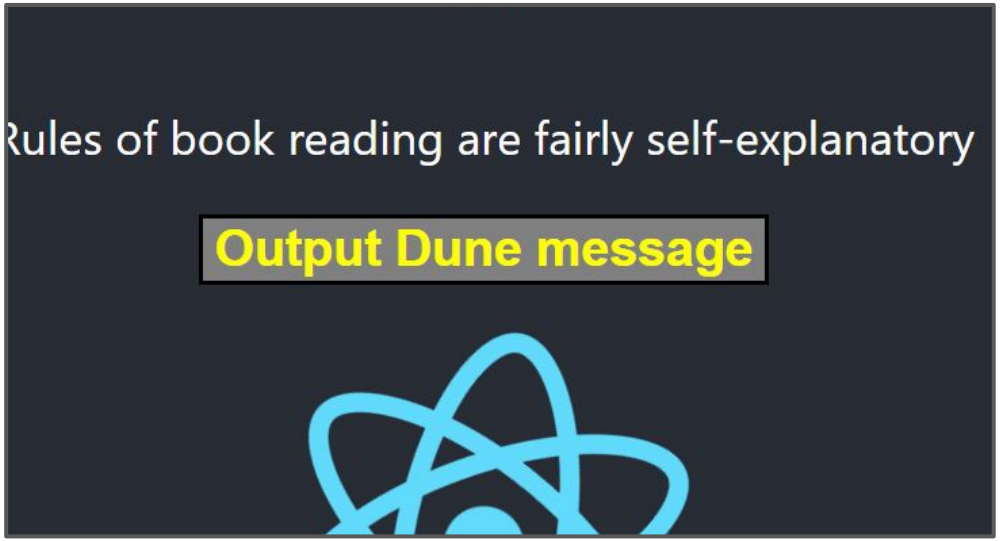
Exercise: Adding props to your button

🕐 ***For the next 10 minutes, add a prop to your button that changes its button text***

- What if in the future, I wanted to change the button text? It'd be easy to modify everything in one central place (e.g. App.js) with the information being passed down appropriately (prop all the way down to button)

- For this exercise, ensure that you have the ability to change your button text **from App.js**.

EXAMPLE OUTPUT



Prop Validation

The whole point on components is reusability **but** how can you ensure the props being passed into that same component are what's need???

You can do this with vanilla JavaScript but it's much easier (and common) using the package `prop-types`.

- Go to your app directory in a terminal
- Run `npm install prop-types`
- Now you can import this in your component
- It comes pre-packaged with all sorts of useful types you can check against and required props (essential for your component) can be marked with `isRequired`

Always read the doc :)

```
import React from 'react';
import PropTypes from 'prop-types'
import "./Button.css";

const Button = ({ message = "Click Me" }) => {
 return (
   <button className="button" type="button">
     <h2 className="button_text">
       {message}
     </h2>
   </button>
 );
}

Button.propTypes = {
 message: PropTypes.string
}

export default Button;
```

Exercise: Play around & cause errors

Your code likely just returns 1 button - what if we try to return another element with it too?

- Your code may look like this at the moment - just a component file that returns a button for App.js (or whoever is above it in terms of hierarchy) to render

```
function Button() {
  return (
    <button className="duneButton">Click me!</bu
  );
}
```

- What happens if you try say, returning a button *and* another element e.g. a p tag like this?

```
function Button() {
  return (
    <button className="duneButton">Click me!</bu
    <h1>Buenos Dias</h1>
  );
```

- *What happens? If its an error, how can you fix it? Your instructor can discuss these solutions after a few minutes*

(Potential?) solutions can be revealed after a click

PONDER & THINK!

*You have approx. **7 minutes** for this (depending on your instructor's discretion + current time!). Google when you can!*

Componentception

React components can be used in multiple places, this includes in other components! This is an example of a subcomponent.

Q. Why use sub-components?

See the right for an example.

SIMPLER TESTS!

React is designed to be interactive and dynamic, this makes testing more complicated! To combat the try to breakdown complex components into testable parts.

Key test examples include:

- Unit testing - typically simple input/output tests for functions that could (or should) live outside components.
- Snapshot testing - with dummy inputs does your component render and do the DOM elements match up to last time?
- Component testing - with the component rendered does it change to user inputs as expected?

```
const sum = (a,b) => a + b;

describe("sum tests", () => {
 test('adding 1 + 2 should return 3', () => {
   expect(sum(1, 2)).toBe(3);
 });
})
```

```
it("renders correctly", () => {
 const tree = shallow(<App />);
 expect(toJson(tree)).toMatchSnapshot();
});
```

Component Test Example

COURSE PROJECTS CRITERIA

‘Must have’

- A minimum of 2 HTML web pages and one external JS file
- All links working
- JavaScript used to enable user interaction
- Basic React implementations

‘Nice to have’

- All of the ‘Must Have criteria’
- Effective use of classes and IDs
- Adding States & Events

PROJECT EXPECTATIONS 

- We'd like you to **work within a team to build a website from scratch** while implementing what you've learned through this course!
- Be sure to meet the **"Must have" criterias** mentioned in the previous slide
- While we want you to work within a team to simulate the industry practice, if you can not due to personal circumstances and need to work on your own, please let your instructor know
- You will be **presenting your website live in Session 8**. You may prepare a slide deck discussing:
  - Introduce your idea and what challenges you faced and how you overcame them
  - Show your code
  - Display your website
- Your presentation should not be longer than **7 minutes**

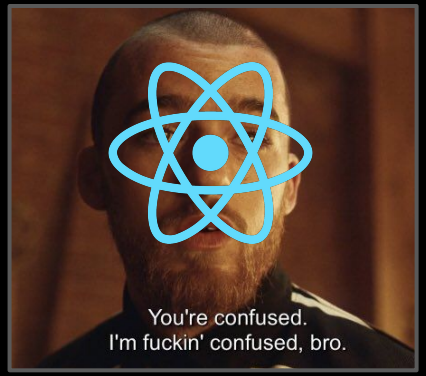
PROJECT TIPS & TRICKS

- Share your code with other team members frequently
- Try to break down big tasks into smaller chunks
- **This is NOT a requirement** but you may use Github Desktop to manage your work and Github Pages to host your website.
- If you choose to try Github, you may use the resources below:
  - Download Github Desktop
  - Follow this tutorial for Github Desktop
  - Follow this tutorial for Github Pages

SUMMARY

CODE FIRST GIRLS

- React can be confusing for **everyone** - remember that it is effectively JSX (combo of HTML + JS) and all it allows you to do is create replicable 'components' (more web dev customizability than standard HTML + CSS).



- These components can be imported and repeated again and again - much easier to customise and repeat **1 component** (a Button.js) than writing its CSS rules and re-checking its consistency across **x repeats.**

- Essentially, all our concerns are encapsulated and abstracted behind one file **only** enabling it to be tested once and used lots.

HOMEWORK

+ **Homework Task**

Continute working on your projects! Sketch out a rough idea of a component you can use in your project, break it down into sub-components and easily testable parts. Start building from the smallest to biggest.

THANK YOU
HAVE A GREAT WEEK!