

## **React**

React is a JS library, but built as if it's a framework. Builds on core Vanilla JS and gives you extra features. Clue is in the name – it REACTS to changes!

### **The Journey of the web**

- Static HTML – originally just pages that never changed and had to be completely reloaded to handle a change
- Server-rendered HTML – Later started using DOM manipulation – interactivity with JS, use things like logins, authentication, serve different pages to different users
- Front-end frameworks – Now taking things up another notch! As computers got more complex, processing power increased and demands from consumers increased so performance issues arose from regular vanilla JS. So people built libraries and frameworks to handle increased performance requirements. Gives us a way to structure our code and gives functionality under the hood to help optimise everything.

React is not the only option (trade-offs!) – also Angular and Vue are 2 other popular frameworks.

React was made by Facebook in 2011 and released as open source in 2013. It's also the most used and wanted web framework according to official stats (StackOverflow Dev survey – check it out!). That's why we love it and SOC use it.

React is used by many of the top fortune 500 companies and massive sites like Facebook, PayPal, Netflix, Twitter, Reddit, BBC, AirBnB etc.

### **Why React?**

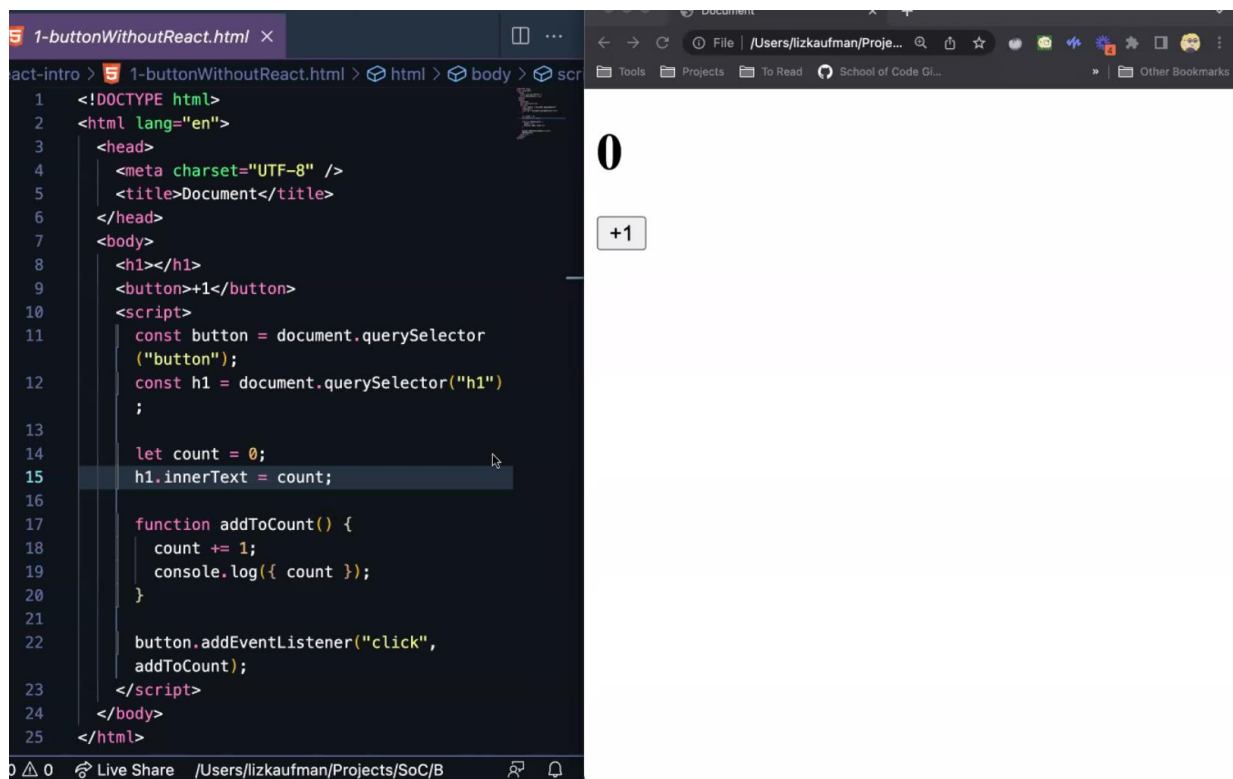
- Keeps your app in sync – it Reacts to changes and easily updates the values of components
- Simplicity – not as in “easy”, but very mature and supported and well documented etc.
- Modular, reusable components – Not just in one app, across multiple apps and just change values!
- Scalable and maintainable – if you have an e-shop with 5 items for sale, you can immediately replicate the data for those 5 items and make it in to 500 items
- Performance – Very fast due to virtual DOM
- Open-source – Always a good thing!
- Widely used – StackOverflow Dev Survey
- Influential – Means there is a drive to keep it maintained etc. (don't always go with the trend though!)
- Just like JavaScript (well, almost!)

Some code with vanilla JS:

- Selecting the button and storing it in a const
- Query selecting the H1 element
- Setting a count variable
- Setting H1 to the count
- Writing a function addToCount
- Adding an event listener to the click to call addToCount

Clicking the button doesn't update the value of H1 – it's only console.logging each time  
The point is JS doesn't automatically/dynamically keep the browser up to date (you would need to put line 15 inside the function addToCount as well)

This is fine on this small scale but if you have to update loads of things, that's lots of code

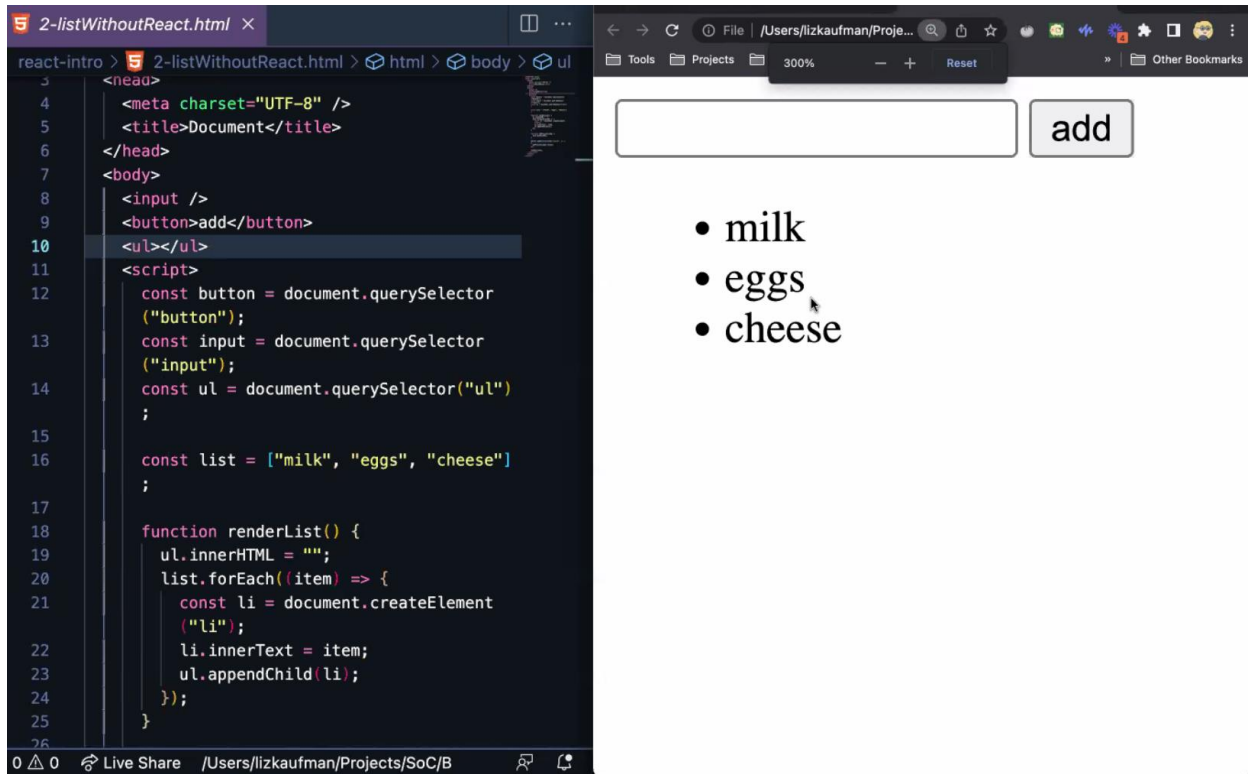


Some more vanilla JS code for a more complex example:

- Selecting button, input and list (ul)
- Create an array containing shopping
- Create a function called renderList that is the equivalent of a for loop to clear the list then recreate the the list using the above shopping list
- Create addToList function to add an item to the array above
- Add event listener to click button to call addToList

You can add an item to the array but it will never be displayed dynamically as it is not updating the innerHTML of the ul after it's done originally

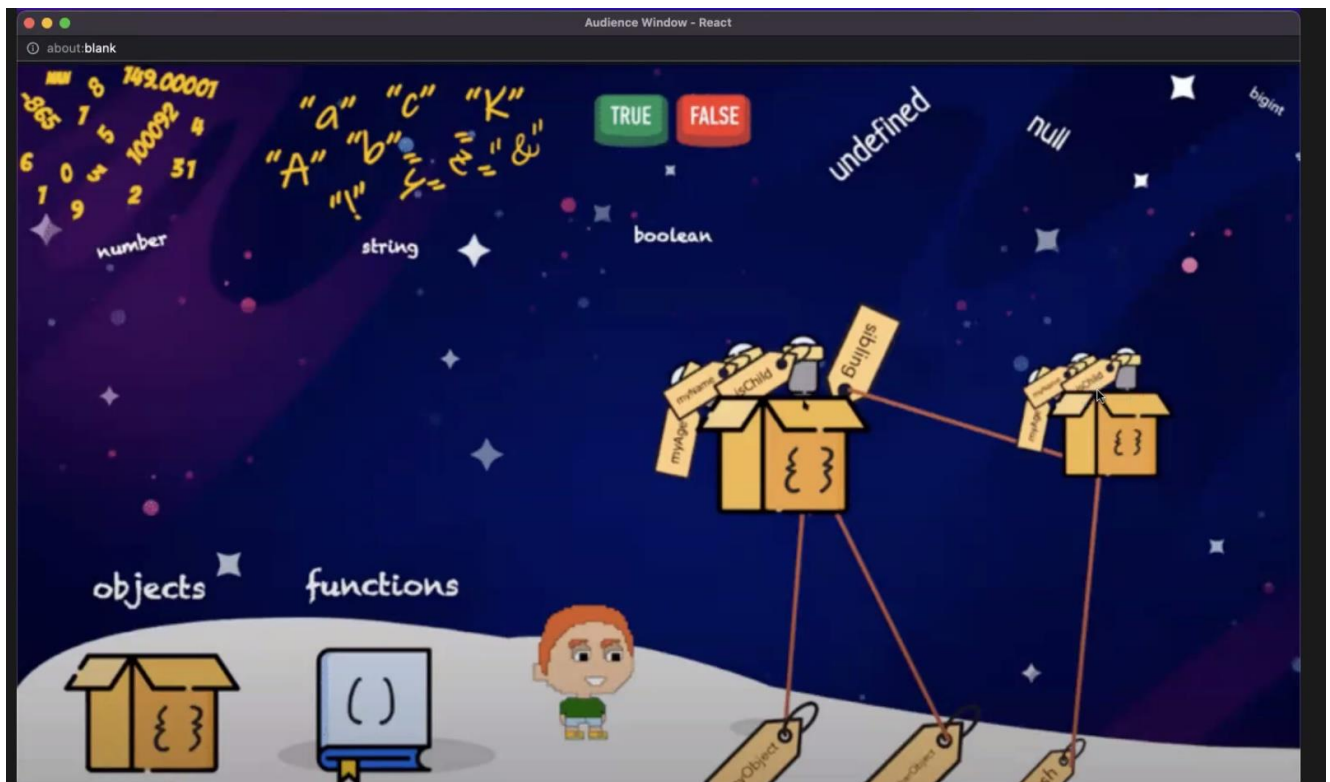
So need to call renderList on the button click as well (can call more than one function for an event)



React is like a really good spot the difference app:



Think back to telescopes pointing at fixed primitives like letters and numbers, but for objects their labels are changeable as they are different places in memory. They would not pass an equality test, as they only reference the same thing, they aren't the same thing. You can have 2 different pointers at the same box (like the my object label). This is similar to how React works:

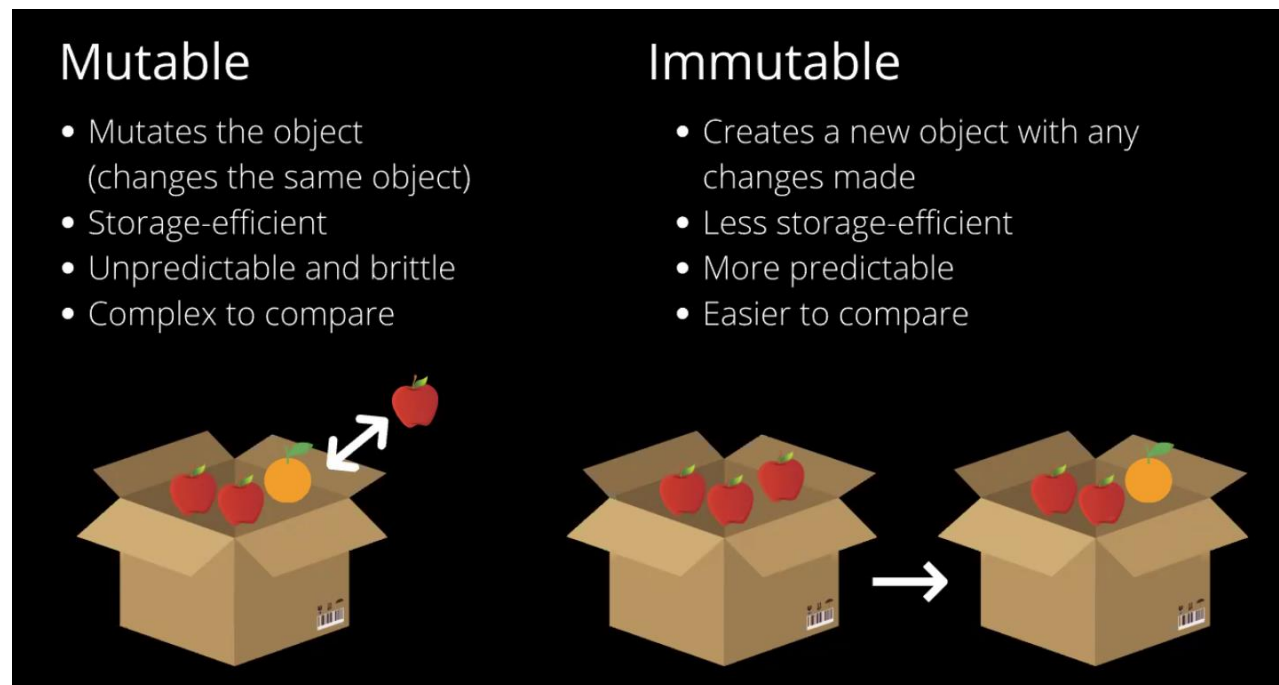


## Concept of Mutable vs. Immutable

Mutable = changeable to the original

Anything immutable can't be changed, you can only create a new object with changes.

Trade-offs as listed



So the shopping list array from the above example is mutable as you can push things to the end of the array without needing to create a new one. To make it immutable, you would need to make a new copy of the array and add something to it like this: [...list, newItem] ...is the spread operator, which makes a copy of a list

Seaside pictures above are like this picture to React, there are 2 separate copies of things and it compares one to another.

```
/path/to/image1.png === /path/to/image2.png
```

```
myOldObject === myNewObject
```

**So in React, make changes immutably so React can reconcile/compare changes and update accordingly!**

So how does React do this? **Using the Virtual DOM.** Compares and runs a reconciliation algorithm to update only if it finds a change.

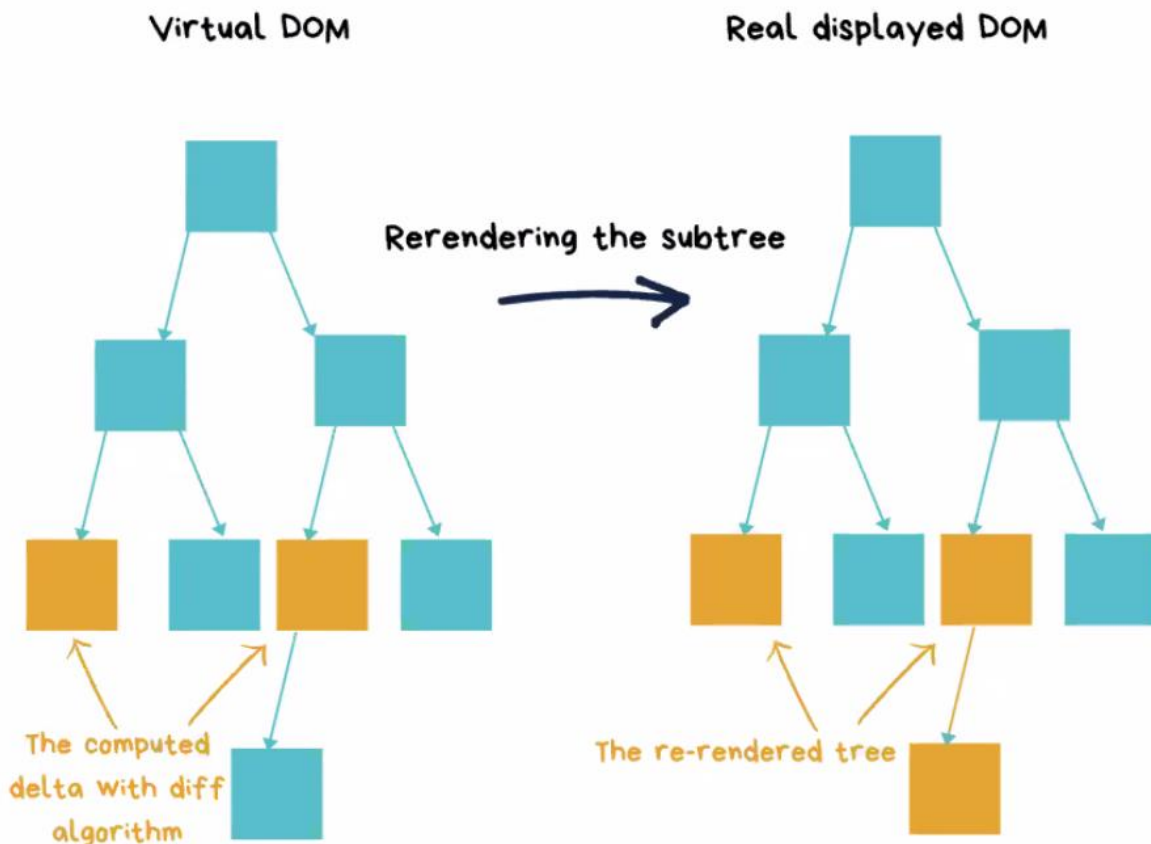
# Virtual DOM

- Virtual representation of UI kept in memory
- Quicker to update
- Reconciled with actual DOM (compares changes and updates accordingly)



Delta just means change – so these are the changes it has worked out

**Still does the child element** on the right as a safety feature to ensure nothing has changed that it hasn't picked up.

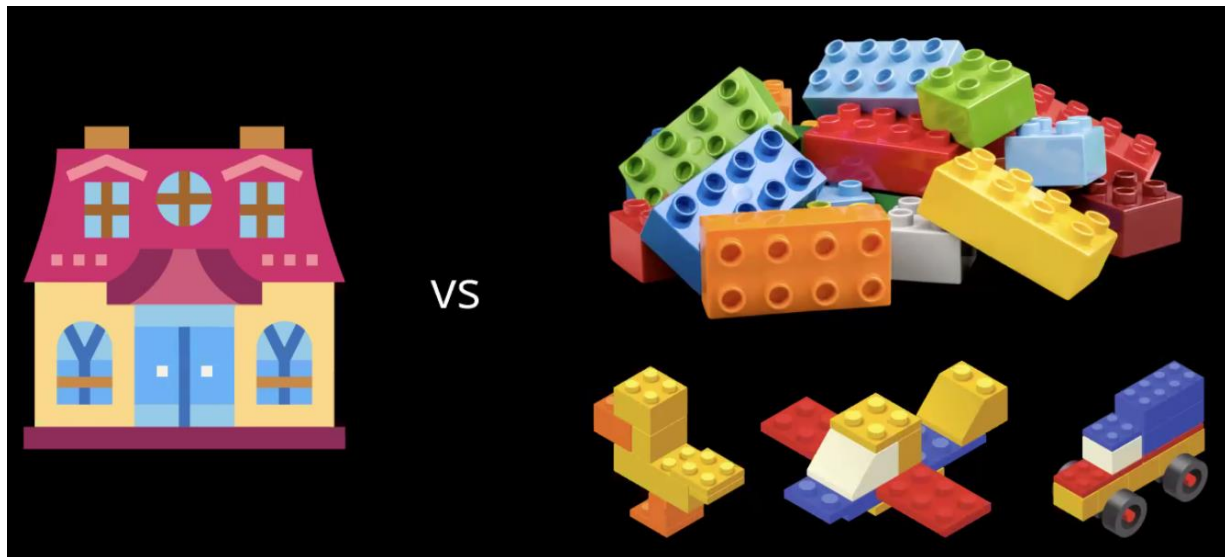




React uses component-based approach

Doll house is all in one so is equivalent of the above examples of vanilla JS code. Can't reuse parts of it because a lot of the code depends on other stuff in the code (e.g. the array).

React is like lego blocks – you can reuse them anywhere at any time!



### **Practical React**

We are learning React from the bottom up – the nitty gritty of building the simplest React app first, manually rendering it then build layers of abstraction. This is important to do it this way so you can understand what is going on under the hood.

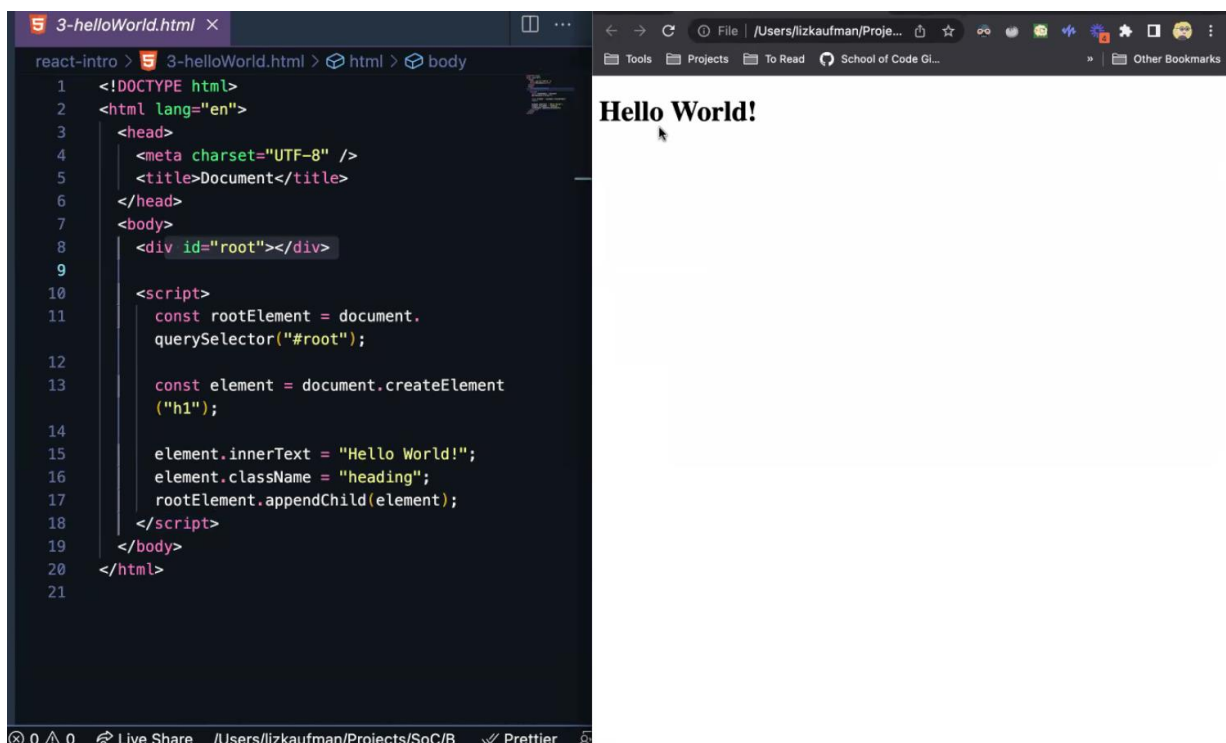
The below code is creating an element that's a `h1`, giving it a class of `.greeting` then setting its text to Hello world! Very similar to using vanilla JS to `document.createElement`, making a `h1`, `element.classList` to give it class of greeting, then setting its `innerHTML` to Hello world!. React abstracts this away.

Then the bottom bit is actually rendering the element and where to put it – like `root.appendChild`.

**This isn't how React is actually used, this is the “under the hood” version.**

```
1 const element = React.createElement(  
2   'h1',  
3   {className: 'greeting'},  
4   'Hello world!'  
5 )  
  
1 ReactDOM.render(element, document.querySelector("#root"));
```

Starting with vanilla JS for a simple Hello World:



The screenshot shows a web browser window with the title "3-helloWorld.html". The browser's developer tools are open, displaying the HTML and JavaScript code. The HTML code is as follows:

```
1 <!DOCTYPE html>  
2 <html lang="en">  
3   <head>  
4     <meta charset="UTF-8" />  
5     <title>Document</title>  
6   </head>  
7   <body>  
8     <div id="root"></div>  
9   </body>  
10  </html>
```

The JavaScript code is as follows:

```
11 const rootElement = document.  
12   querySelector("#root");  
13  
14 const element = document.createElement  
15   ("h1");  
16  
17 element.innerText = "Hello World!";  
18 element.className = "heading";  
19 rootElement.appendChild(element);  
20  
21
```

The browser window displays the text "Hello World!" in a large, bold, black font.

Refactoring it in to React:



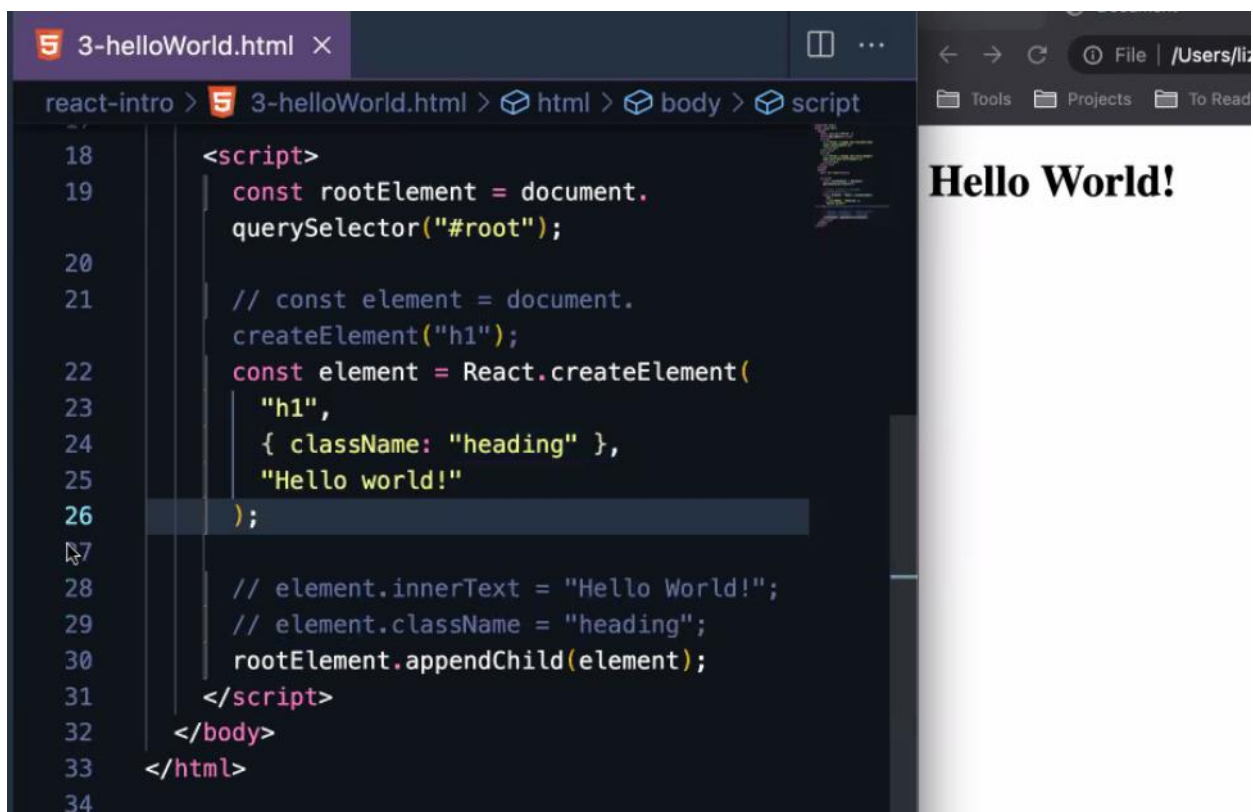
For our baby steps, the simplest way to get React working is calling upon React files hosted in a server elsewhere, rather than installing in node\_modules. React is hosted on a Content Delivery Network (a server with content on it that is delivered when asked). This is only suitable for development, not production. Google React CDN, gives you script tags to load JS inside HTML – put them inside the HEAD.

Instead of

```
const element = document.createElement("h1");
element.innerText = "Hello World!";
element.className = "heading";
```

Use:

```
const element = React.createElement(Takes 3 arguments
  "h1", element type
  {className: "heading"}, an object with any properties
  "Hello World!" innertext
)
```



Instead of `rootElement.appendChild(element);`

`ReactDOM.render(element, rootElement);` 2 arguments - Which element you are appending and which element to append it to

## **Creating a functional component**

Something we can reuse, like a function!

Convention is to capitalise first letter of a functional component

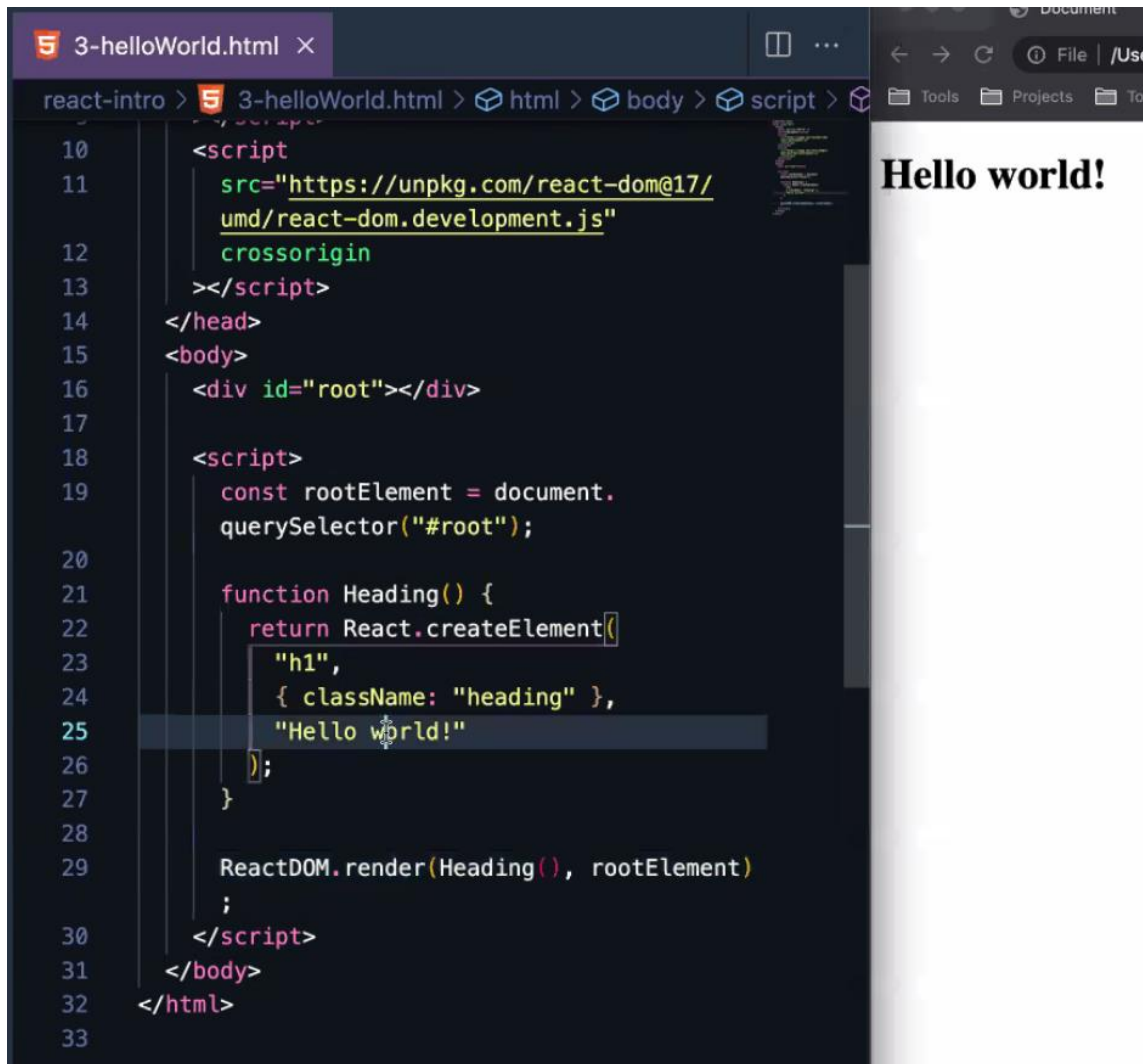
In our example, we want to make a functional component that returns the created h1 instead of making it outside

```
function Heading() {  
  const element = React.createElement(  
    "h1",  
    {className: "container"},  
    "Hello World"  
  )  
}
```

Then need to change the thing that renders it:

```
ReactDOM.render(Heading(), rootElement);
```

Screenshot of proper code below:



The screenshot shows a web browser window with a dark-themed code editor on the left and the rendered output on the right. The code editor displays the following HTML and JavaScript code:

```
10 <script
11   src="https://unpkg.com/react-dom@17/
12   umd/react-dom.development.js"
13   crossorigin
14 ></script>
15 </head>
16 <body>
17   <div id="root"></div>
18
19   <script>
20     const rootElement = document.
21     querySelector("#root");
22
23     function Heading() {
24       return React.createElement(
25         "h1",
26         { className: "heading" },
27         "Hello world!"
28       );
29     }
30
31     ReactDOM.render(Heading(), rootElement)
32   </script>
33 </body>
</html>
```

On the right side of the browser, the text "Hello world!" is displayed in a large, bold, black font.

Next piece of the puzzle is to make it customisable, so instead of saying Hello world! everytime, we want to be able to customise it

**In React we do this with PROPS (short for properties).** By passing it in to the Heading function. We've called our props "text" in this example, but is often passed in as "props":

```
function Heading(text) {
  const element = React.createElement(
    "h1",
    {className: "container"},
    text
  )
}
```

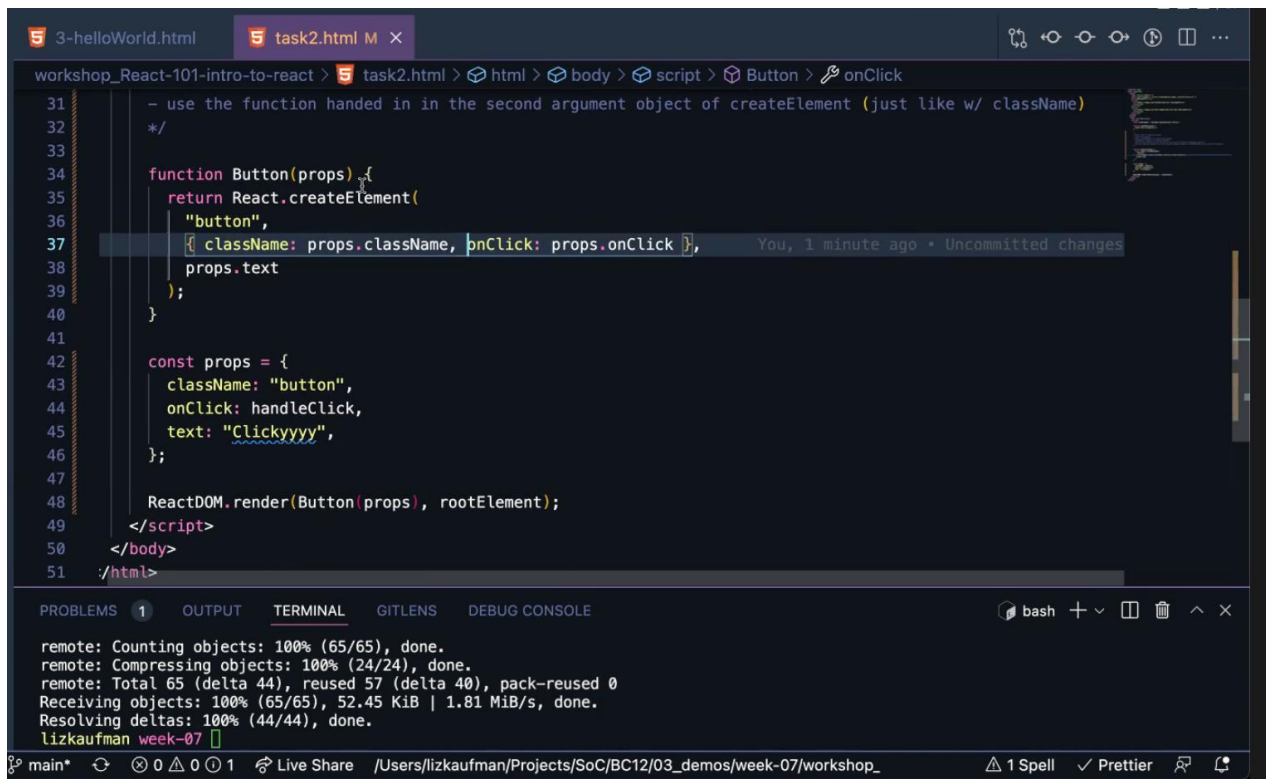
*Calling what has been passed in*

```
ReactDOM.render(Heading("Hello everyone!"), rootElement);
```

So this function now is reusable as when the render method is called, we can hand in any instance of the heading. But only has the values we have hard-coded, so next step:

### **Below code example shows passing in props**

Defining props outside of creating the element so the code that makes the element is completely reusable. Define props on line 42-46, pass it in on line 34 and 48, then access its values on line 37-38.



```
31  - use the function handed in in the second argument object of createElement (just like w/ className)
32  */
33
34  function Button(props) {
35    return React.createElement(
36      "button",
37      { className: props.className, onClick: props.onClick },
38      props.text
39    );
40  }
41
42  const props = {
43    className: "button",
44    onClick: handleClick,
45    text: "Clickyyyy",
46  };
47
48  ReactDOM.render(Button(props), rootElement);
49 </script>
50 </body>
51 </html>
```

remote: Counting objects: 100% (65/65), done.  
remote: Compressing objects: 100% (24/24), done.  
remote: Total 65 (delta 44), reused 57 (delta 40), pack-reused 0  
Receiving objects: 100% (65/65), 52.45 KiB | 1.81 MiB/s, done.  
Resolving deltas: 100% (44/44), done.  
lizkaufman week-07