Lab 3 - React

Graphical User Interfaces[*]

In this lab, we introduce the key ideas of the React framework. On completion of this tutorial you will have a greater understanding of:

- Navigating the react project template
- Adding to the DOM using react
- Managing state within a react app

In our tutorial, we will be making a "find the needle" in the haystack game. We will have a grid of hay, with a needle hidden behind one of the haystacks. The user must click to find it.

1 Introduction & Installation

React is a Javascript library that we can use to build user interfaces. It is a great library to learn because, if you learn how to use react, then it is just as easy to pick up other Javascript libraries (e.g. Angular.js).

React manages a *virtual* Document Object Model (DOM), which is used to modify our actual DOM within Javascript. It means that instead of updating our entire DOM every time we make a change, only the specific parts of the DOM that need modification are altered, making our websites faster.

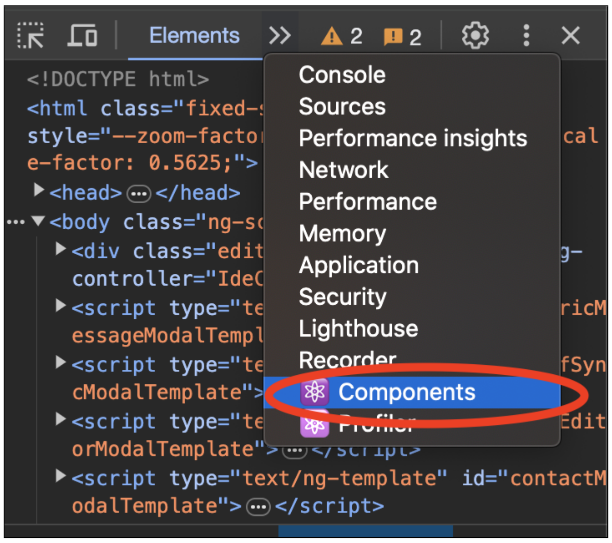
1.1 Installing NodeJS

To run React code and install our dependencies, we have to install NodeJS. The simplest method is to use the installer downloadable from this link `https://nodejs.org/en`, following the steps on their webpage. If you are familiar with package managers such as brew then feel free to use this also. This is required for this tutorial.

[*]Written by Corey Ford 2023/24 for Queen Mary University of London (c.j.ford@qmul.ac.uk)

We will be writing code in Visual Studio, which makes running terminal commands super easy, amongst other benefits. Make sure this is installed.

If you are using Google Chrome, you can add the react developer tools as an extension – `https://chromewebstore.google.com/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi`. We do not use this in the tutorials, but it is recommended. There are also tools for Firefox and other browsers, with instructions available online. Once installed, on opening the Javascript console (View >> Developer >> Javascript console), you will be able to see brand new options when clicking the two arrows, as shown in the figure below.

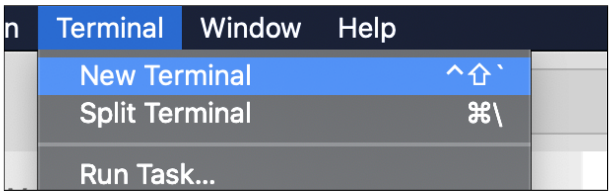


```
<!DOCTYPE html>
<html class="fixed-
style="--zoom-facto                                cal
e-factor: 0.5625;">
 <head>⋯</head>
 <body class="ng-sc
  <div class="edit                                 g-
  controller="IdeC
  <script type="te                                 ricM
  essageModalTempl
  <script type="te                                 fSyn
  cModalTemplate">
  <script type="te                                 Edit
  orModalTemplate">⋯</script>
  <script type="text/ng-template" id="contactM
  odalTemplate">⋯</script>
```

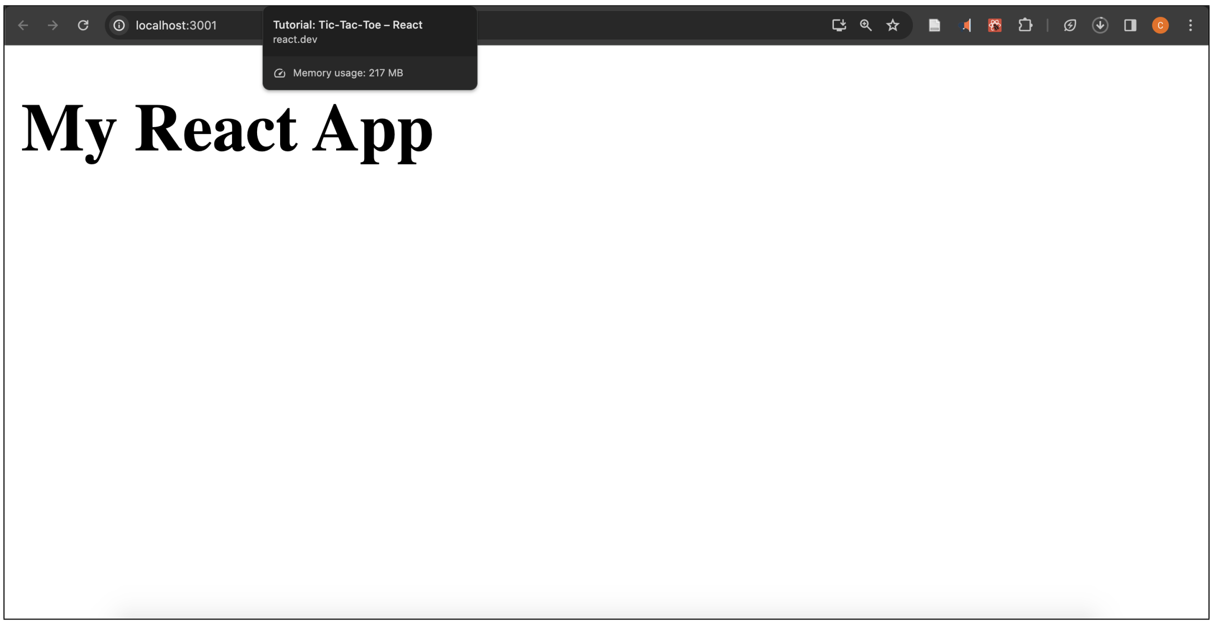
1.2 Our Template

The boilerplate code for React can be found on QMPLUS. Open this code in Visual Studio.

Then, in Visual studio, go to terminal >> new terminal.



In the terminal window at the bottom of the screen, then run **npm install** to create our **node_modules** folder, followed by **npm start**. Make sure that you are inside the correct folder. On **npm start** a web-page should open that looks like this.



My React App

1.3 The project files

The files inside our project are organised as follows:

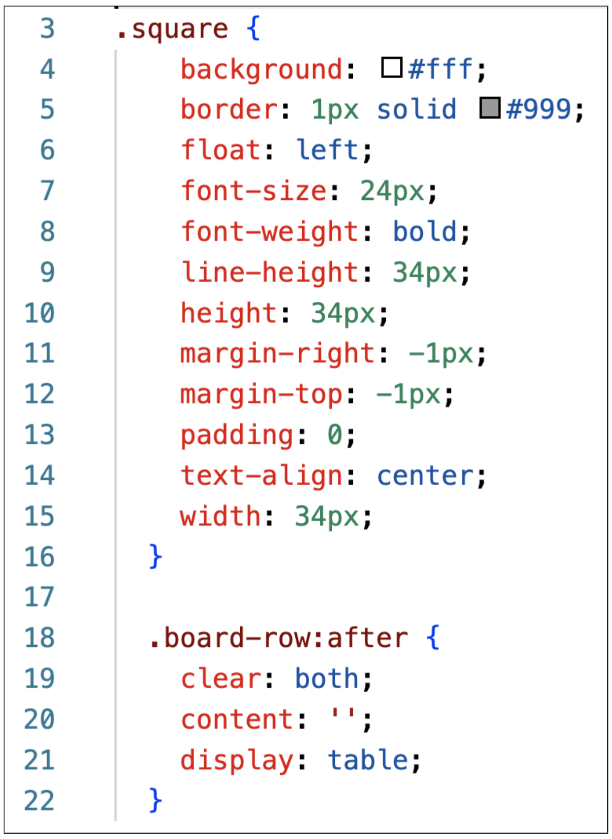
In the public folder, you will find out index.html file. On line 31 there is a div with the ID root. Everything we write in react will be added to this DIV.

In the src folder are the files we will be altering. First is index.js which renders a react component called App. On line 10 is a call to the App react component – this is attached to the root DIV from our *index.html* on line 7. Lines 1-5 show links to the react and react DOM library, our CSS stylesheet, and the App.js file.

If you open the App.js file we see a function which returns our heading tag, we see in the browser. All react components which we write must be placed here, in between the tags on lines 3 and 5. Although this looks like HTML, this is actually JSX (Javascript as XML) which is compiled to Javascript – meaning we can write more HTML styled code to create UX elements. On line 17, the keyword export means that we can access the function in files outside of this one, and default, means that the App function is the default entry point.

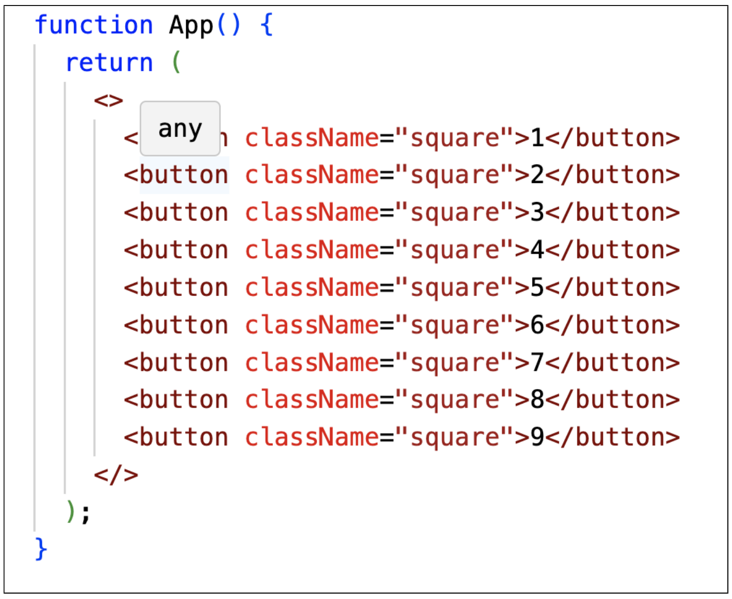
2 Game Grid

Lets create our game grid. First, add the code below to the style.css file.



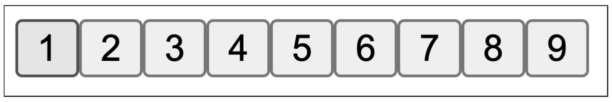
```css
.square {
    background: □#fff;
    border: 1px solid ■#999;
    float: left;
    font-size: 24px;
    font-weight: bold;
    line-height: 34px;
    height: 34px;
    margin-right: -1px;
    margin-top: -1px;
    padding: 0;
    text-align: center;
    width: 34px;
  }

  .board-row:after {
    clear: both;
    content: '';
    display: table;
  }
```

Lets now turn our attention to the App.js file. Instead of headers, we want to add 9 buttons. These include the attribute className="square" to use our CSS style, and must be added inbetween lines 3 and 4. See below.

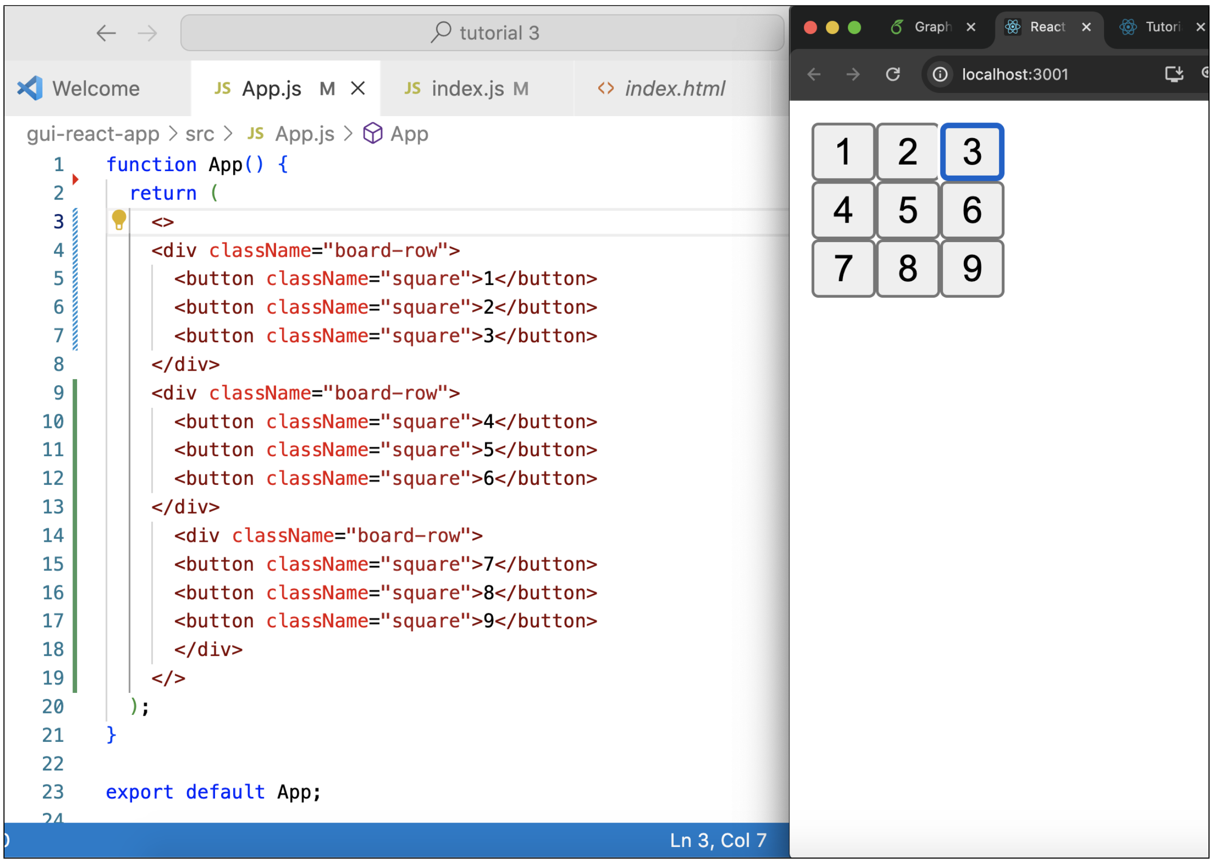


```jsx
function App() {
  return (
    <>
      < any n className="square">1</button>
      <button className="square">2</button>
      <button className="square">3</button>
      <button className="square">4</button>
      <button className="square">5</button>
      <button className="square">6</button>
      <button className="square">7</button>
      <button className="square">8</button>
      <button className="square">9</button>
    </>
  );
}
```

At this stage our webpage will look like this (be sure to save the code file for the browser to update).

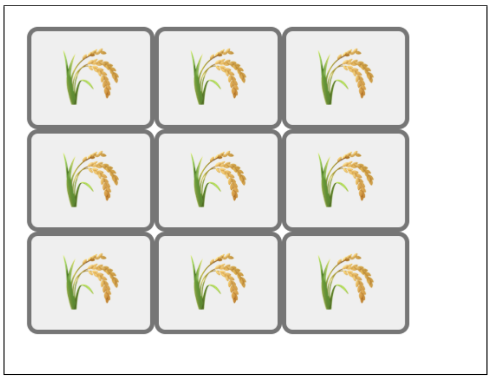


To make the grid lets wrap the buttons in DIVs, and apply our CSS. The code below should give as a grid.



```
function App() {
  return (
    <>
    <div className="board-row">
      <button className="square">1</button>
      <button className="square">2</button>
      <button className="square">3</button>
    </div>
    <div className="board-row">
      <button className="square">4</button>
      <button className="square">5</button>
      <button className="square">6</button>
    </div>
      <div className="board-row">
      <button className="square">7</button>
      <button className="square">8</button>
      <button className="square">9</button>
      </div>
    </>
  );
}

export default App;
```

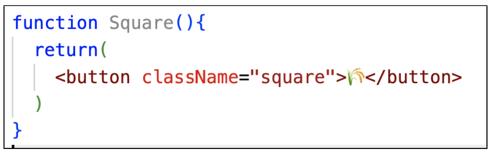
We want all our grid buttons to show a haystack, so replace numbers 1-9 with the haystack emoji. This can be copied and pasted from here `https://emojidb.org/haystack-emojis?user_typed_query=1&utm_source=user_search`.



3 Passing data through props

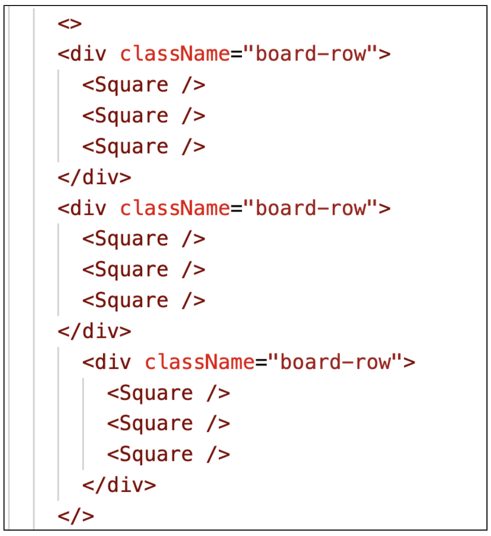
We want to change our haystack emoji to either a cross or a needle, depending of whether the needle is hidden behind this particular grid element or not.

To do this we will move our button components into a react component. Remember, a react component is a function which returns a JSX element (e.g. button). above the current app function. Add the function below to App.js.



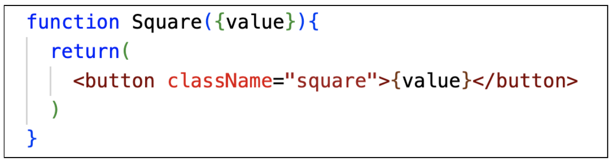
```
function Square(){
  return(
    <button className="square">🌾</button>
  )
}
```

We can then call our Square component using the JSX syntax as shown below:



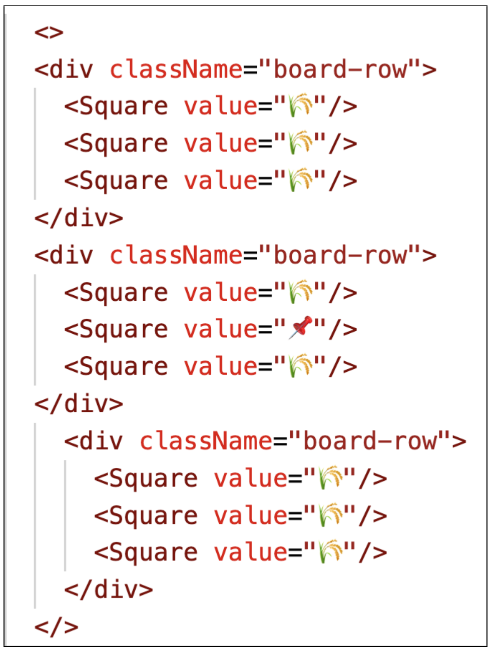
```
<>
<div className="board-row">
  <Square />
  <Square />
  <Square />
</div>
<div className="board-row">
  <Square />
  <Square />
  <Square />
</div>
  <div className="board-row">
    <Square />
    <Square />
    <Square />
  </div>
</>
```

To pass values to our square function, we can then use {}. You can think of using {} as "escaping into JavaScript" from JSX. Over time, this starts to become natural, but is weird at first. So, we pass a variable "value" to our function and replace our emoji.



```
function Square({value}){
  return(
    <button className="square">{value}</button>
  )
}
```

We can then pass various values like we would CSS attributes.



```
<>
<div className="board-row">
  <Square value="🌾"/>
  <Square value="🌾"/>
  <Square value="🌾"/>
</div>
<div className="board-row">
  <Square value="🌾"/>
  <Square value="📌"/>
  <Square value="🌾"/>
</div>
  <div className="board-row">
    <Square value="🌾"/>
    <Square value="🌾"/>
    <Square value="🌾"/>
  </div>
</>
```

4 Making it interactive - one button

We could add an on click function to our Square component, so that each button will either then show a pin or a cross, depending on if the pin is found in the haystack. We are actually going to implement this functionality in a way that allows us to track the games state, as this is more efficient in some cases, and means that everything is tracked for the DOM by React.

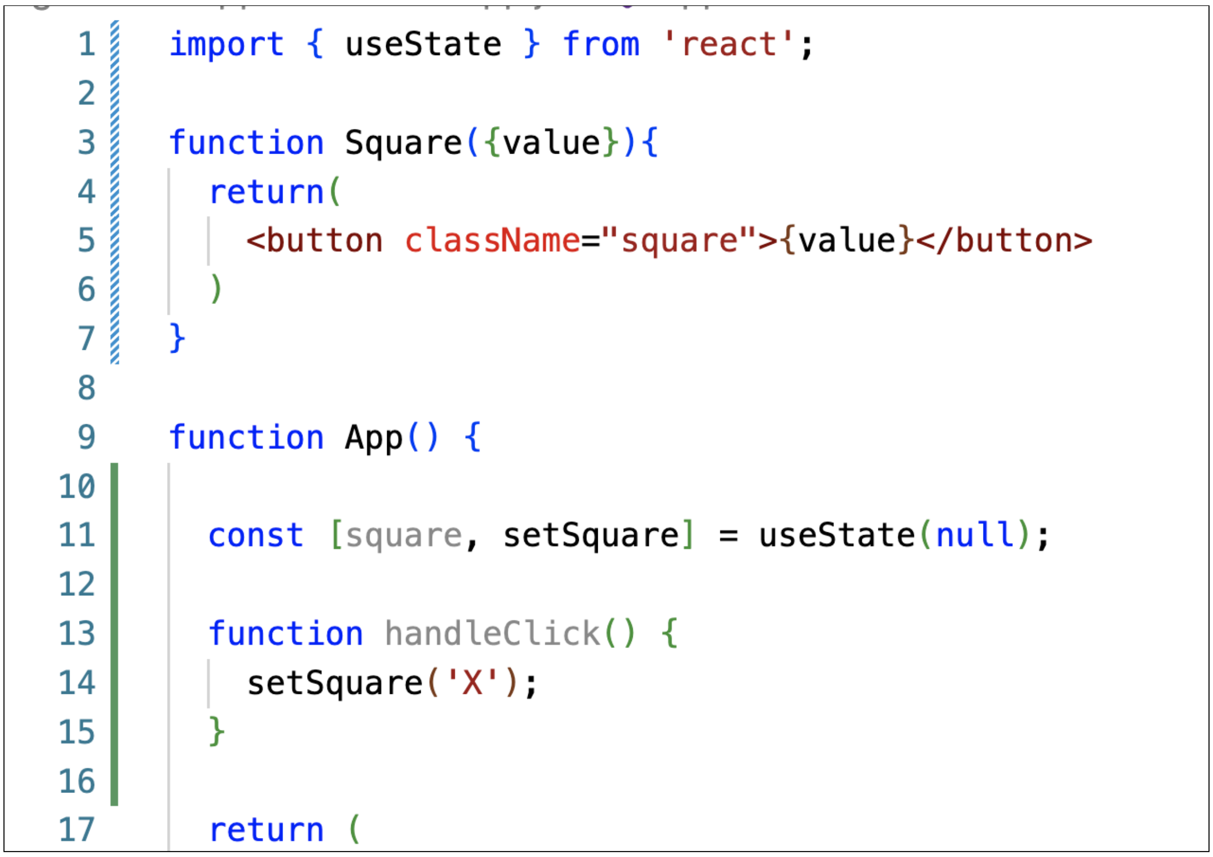
First we will do this for one button, then for multiple. So, for one button:

a. Add **import { useState } from 'react';** to the top of the App.js file. This is necessary to use the useState hook, which enables state management in functional components within React by providing access to state variables and their update functions. This facilitates the management of state within functional components, allowing for dynamic updates and rendering in the DOM based on changes in state.

b. Add **const [square, setSquare] = useState(null);** within our App function, but above our return statement. The two values in the square brackets are as follows – **square** is the current state, and **setSquare** is a function that we can call to change this state. **useState(null)** sets the initial state of **square** to null.

c. Write a function below the line above, but still before the return statement (all within App()), named handleClick(). Within this function, call **setSquare('X');**.

So far your code should look like this:



```
import { useState } from 'react';

function Square({value}){
  return(
    <button className="square">{value}</button>
  )
}

function App() {

  const [square, setSquare] = useState(null);

  function handleClick() {
    setSquare('X');
  }

  return (
```

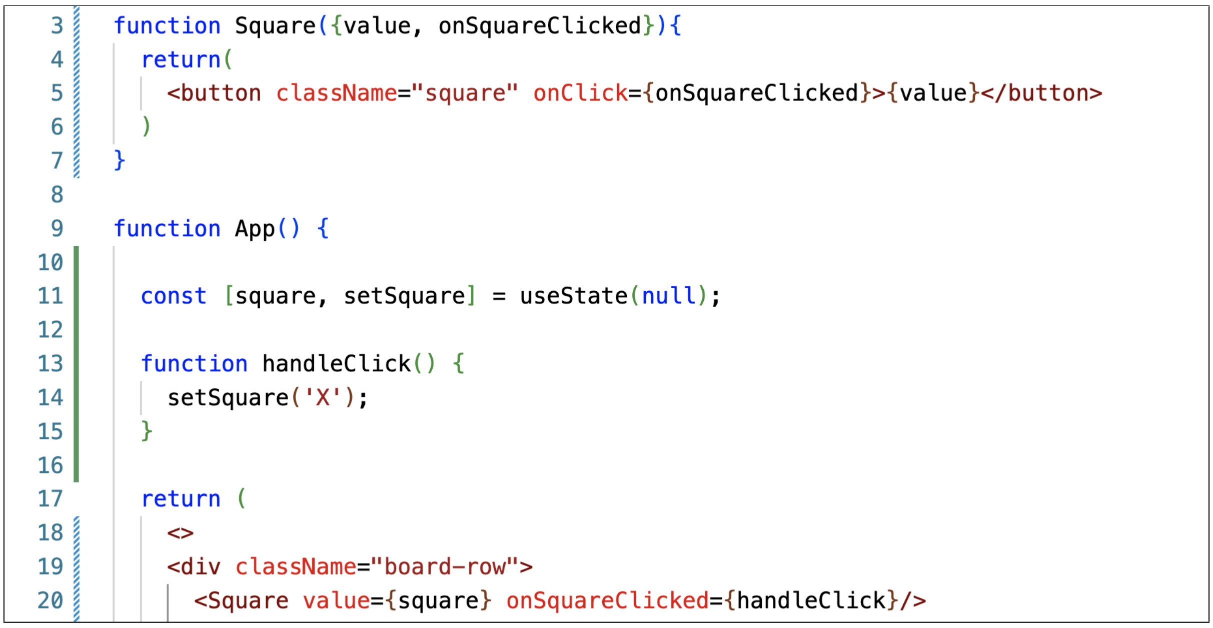
Next,

a. We will also pass a variable **onSquareClicked** to our square function, beside **value**.

b. In our square function, we will apply the **onSquareClicked** variable to the **onClick** attribute of button.

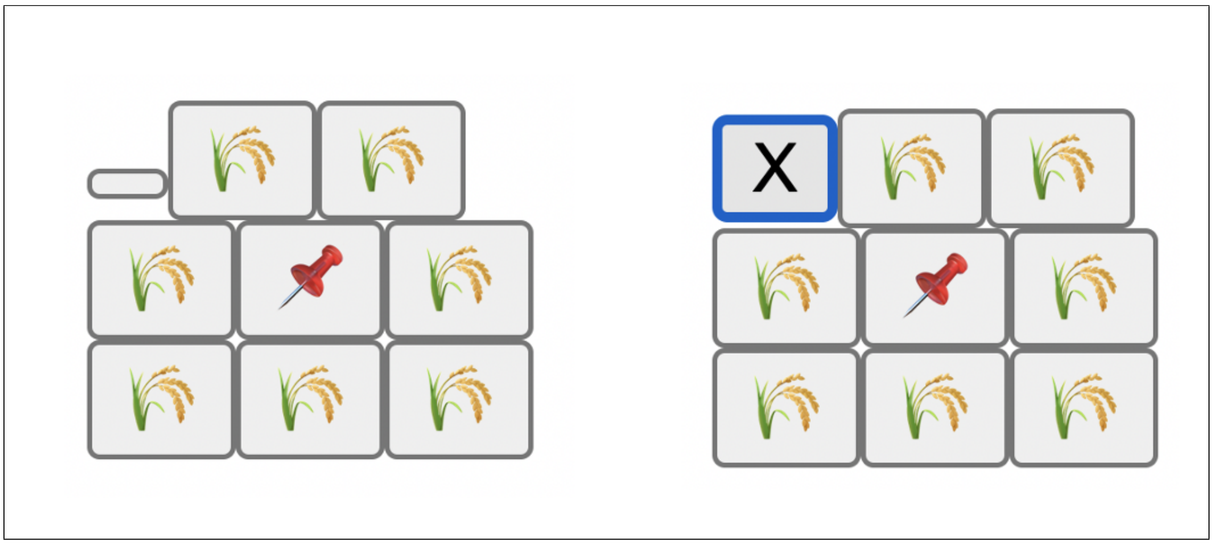
c. We will parse both the **square** value and **handleClick** value to our leftmost button.

Our code should now look like this:



```
function Square({value, onSquareClicked}){
  return(
    <button className="square" onClick={onSquareClicked}>{value}</button>
  )
}

function App() {

  const [square, setSquare] = useState(null);

  function handleClick() {
    setSquare('X');
  }

  return (
    <>
    <div className="board-row">
      <Square value={square} onSquareClicked={handleClick}/>
```

When we run our app we will see a squished square (will a null value), and an X value when we click on it, shown below.



5 Making it interactive - all buttons

Let's now track the state of the app for all the buttons.

a. Modify our squares code to be an array of Nine values.

```
const [squares, setSquares] = useState(Array(9).fill(null));
```

b. Lets also modify our instances of Square to take the values in the array.

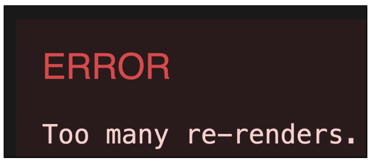
```
<div className="board-row">
  <Square value={squares[0]} onSquareClicked={handleClick}/>
  <Square value={squares[1]} onSquareClicked={handleClick}/>
  <Square value={squares[2]} onSquareClicked={handleClick}/>
</div>
```

Now let's turn our attention to the handle click function. The challenge here is that we need to know which square to change. Lets try parsing an index, like in the code below.

```
13  function handleClick(i) {
14    const nextSquares = squares.slice();
15    nextSquares[i] = "X";
16    setSquares(nextSquares);
17  }
18
19  return (
20    <>
21    <div className="board-row">
22      <Square value={squares[0]} onSquareClicked={handleClick(0)}/>
23      <Square value={squares[1]} onSquareClicked={handleClick(1)}/>
24      <Square value={squares[2]} onSquareClicked={handleClick(2)}/>
25    </div>
```

Note on line 14 we call .slice() to create a copy of the squares array instead of modifying the existing array – this gives us benefits later on. Child components in React typically re-render automatically when a parent component's state changes, impacting even unaffected child components, yet for optimal performance, there may be a need to selectively avoid re-rendering parts of the component tree that remain unaffected by the state change – hence, creating a copy here.

On running the code, however, we will see the following error:



```
ERROR

Too many re-renders.
```

When you were passing onSquareClick=handleClick, you were passing the handleClick function down as a prop. You were not calling it! But now you are calling that function right away—notice the parentheses in handleClick(0)—and that's why it runs too early. You don't want to call handleClick until the user clicks!

You could fix this by creating a function like handleFirstSquareClick that calls handleClick(0), a function like handleSecondSquareClick that calls handleClick(1), and so on. You would pass (rather than call) these functions down as props like onSquareClick=handleFirstSquareClick. This would solve the infinite loop.

However, defining nine different functions and giving each of them a name is too verbose. Instead, let's do this:

- **<Square value={squares[0]} onSquareClick={() => handleClick(0)} />**

Notice the new () => syntax. Here, () => handleClick(0) is an arrow function, which is a shorter way to define functions. When the square is clicked, the code after the => "arrow" will run, calling handleClick(0).

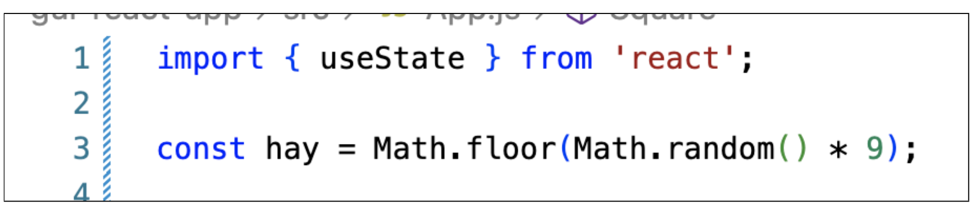
Now update the other eight squares to call handleClick from the arrow functions you pass.

At this stage you might have a non-visible game grid – don't panic! We add the final mechanics in the next step!

6 Final game mechanics

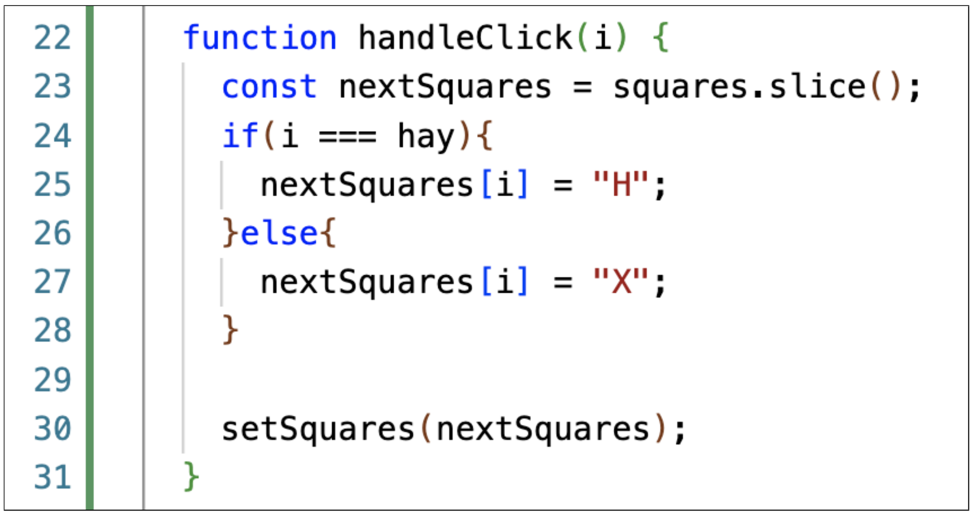
Lets add the final game mechanics.

First, create a random number between zero and 9. Add this const variable outside of the other functions.



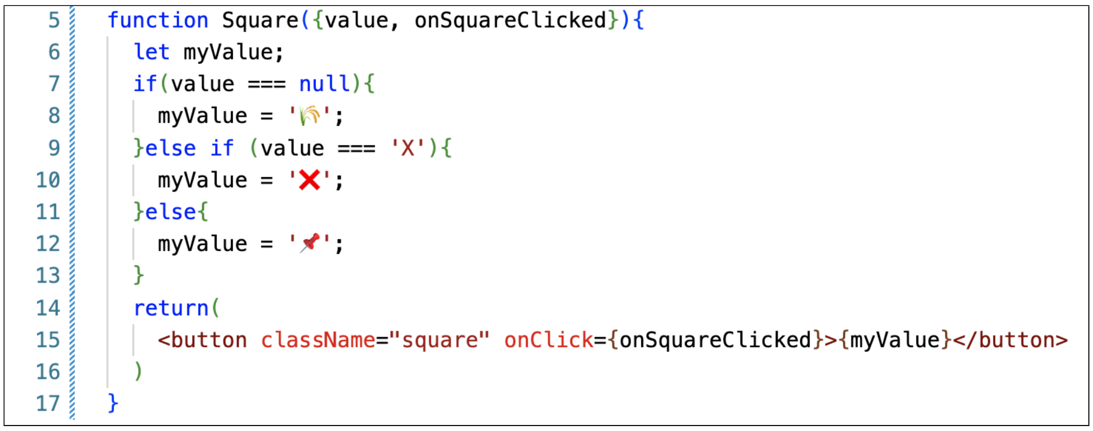
```
import { useState } from 'react';

const hay = Math.floor(Math.random() * 9);
```

Next, add an if-statement to check for our random number within the handleClick function.



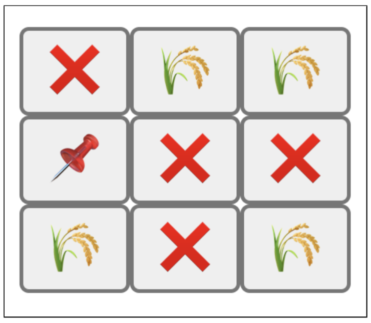
```
function handleClick(i) {
  const nextSquares = squares.slice();
  if(i === hay){
    nextSquares[i] = "H";
  }else{
    nextSquares[i] = "X";
  }

  setSquares(nextSquares);
}
```

Finally, lets update our Square react component to show the correct emojis, adding a if-statement to the function, as follows.



```
function Square({value, onSquareClicked}){
  let myValue;
  if(value === null){
    myValue = '🌾';
  }else if (value === 'X'){
    myValue = '❌';
  }else{
    myValue = '📌';
  }
  return(
    <button className="square" onClick={onSquareClicked}>{myValue}</button>
  )
}
```

Enjoy playing your game.



7 Summary

In this tutorial, we introduced the key concepts of the React framework. The tutorial covered essential aspects such as navigating the React project template, adding elements to the DOM using React, and managing state within a React app. The practical example involved creating a "find the needle" game with a grid of haystacks, where users click to find the hidden needle. The tutorial progressed through sections on introduction and installation, setting up the game grid, passing data through props, and making the game interactive with React state management. The final implementation included randomizing the needle's position and dynamically updating the grid based on user interactions, resulting in an enjoyable and functional game.

8 Reading

This tutorial was simplified and adapted from: `https://react.dev/learn/tutorial-tic-tac-toe`. This is a great tutorial to test out if you want to recap the skills in this tutorial, and go into more detail.

Linked-In learning (given to you as part of Queen Mary) also has some great courses on React. In particular, I recommend **ReactJS Essential Training**.