

Lab: introduction to React App Development (tic-tac-toe game)

Student's name Arvino Gega

Learning outcome: At the completion of this homework, students should possess the ability to proficiently construct a React application utilizing various React technologies and frameworks.

Approximate time to complete homework: 3 hours

Activity description: This homework is a tutorial example from [React.dev](https://react.dev). It will give you an introduction to the React concepts that you will use on a daily basis.

You will build a small tic-tac-toe game during this homework. Each tutorial will ask you to answer some questions and have screenshot along the way.

This homework does not assume any existing React knowledge. The techniques you'll learn in the tutorial are fundamental to building any React app, and fully understanding it will give you a deep understanding of React.

The tutorial is divided into several sections:

- [Setup for the tutorial](#) will give you a starting point to follow the tutorial.
- [Overview](#) will teach you the fundamentals of React: components, props, and state.
- [Completing the game](#) will teach you the most common techniques in React development.
- [Adding time travel](#) will give you a deeper insight into the unique strengths of React.

In this homework, you'll build an interactive tic-tac-toe game with React. The steps to complete the tic-tac-toe are:

1. Open the Terminal, command window, and cd to the location of the **reactjs** project folder. For example, if the project folder is located at the *Documents* folder:

```
C:\Users\Documents\reactjs
```

2. in the project folder, create a react app and name the project folder with your as *tictactoe_lastname*

windows □ C:\Users\Documents\reactjs>**create-react-app tictactoe_prof_wu**

mac □ C:\Users\Documents\reactjs>**sudo create-react-app tictactoe_prof_wu**

3. once the react project folder is created, cd to the project folder:

C:\Users\Documents\reactjs>**cd tictactoe_prof_wu**

Once pressed Enter, the current directory should show:

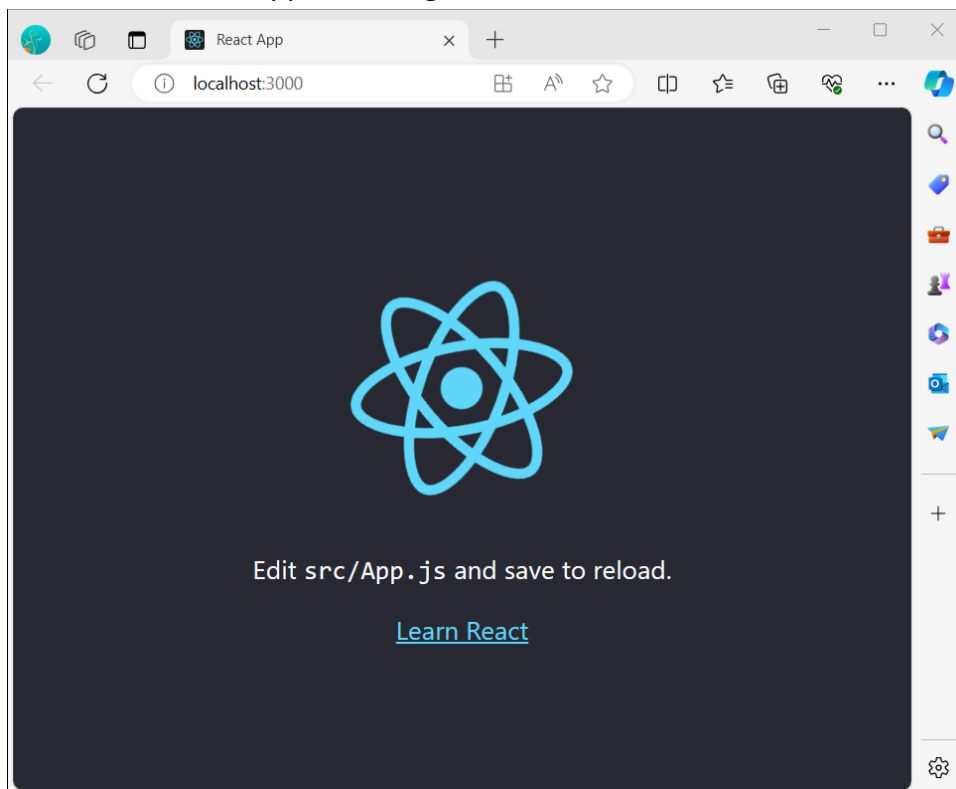
C:\Users\Documents\reactjs\tictactoe_prof_wu>

4. start the react app:

windows □ C:\Users\Documents\reactjs\tictactoe_prof_wu>**npm start**

mac □ C:\Users\Documents\reactjs\tictactoe_prof_wu>**sudo npm start**

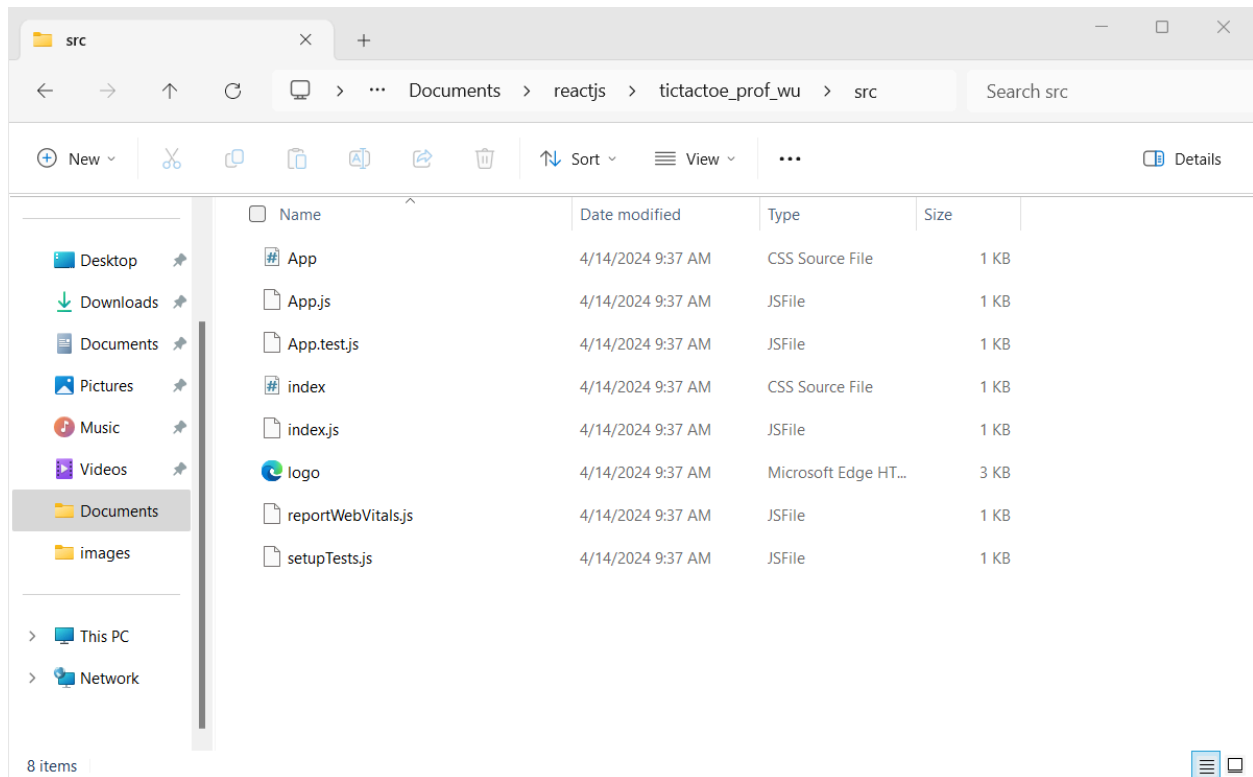
make sure that the app is running at **localhost:3000**



5. open Visual Studio Code and open the folder **tictactoe_lastname**

Inspecting the starter code

In the project folder, the **src** folder will have the following files:



The Files section with a list of files like App.js, index.js, styles.css and a folder called public:

- The code editor where you'll see the source code of your selected file.
- The browser section where you'll see how the code you've written will be displayed.
- Now let's have a look at the files in the starter code.

App.js

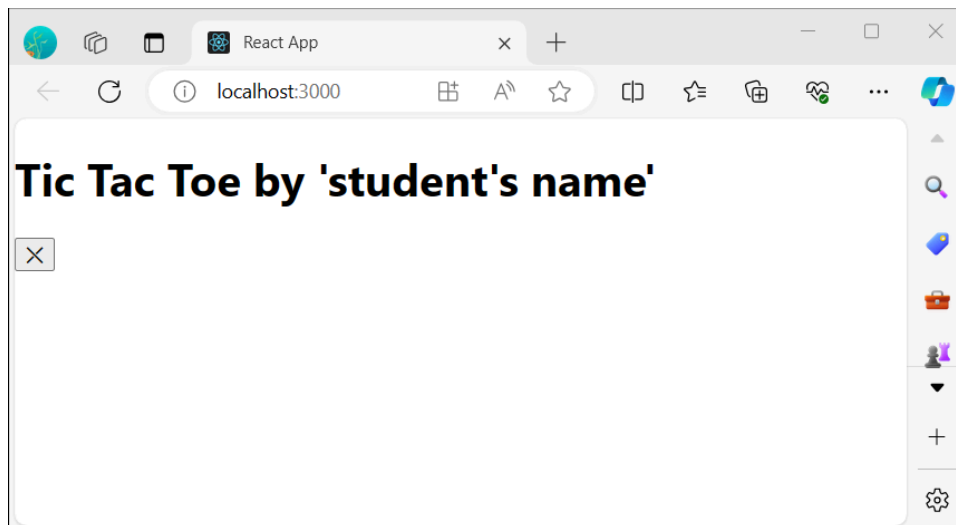
The code in App.js creates a component. In React, a component is a piece of reusable code that represents a part of a user interface. Components are used to render, manage, and update the UI elements in your application. Add the following lines in the App.js file

```
import './App.css';
```

```
function App() {
  return (
    <div>
      <h1 className='title'>Tic Tac Toe by 'student's name'</h1>
      <button className="square">X</button>
    </div>
  );
}

export default App;
```

The localhost:3000 should look as:



Click on the file labeled **styles.css** . This file defines the styles for your React app. Add the following lines in the styles.css file:

```
h1.title{
  text-align: center;
  background-color: cadetblue;
  padding: 1em;
}
button.square{
  text-align: center;
  padding: 1em;
  font-size: 2em;
}
```

Activity: The *localhost:3000* should look as (have a screenshot of your *localhost:3000* and paste the image below)



Building the board

Currently the board is only a single square, but you need nine! If you just try and copy paste your square to make two squares like this:

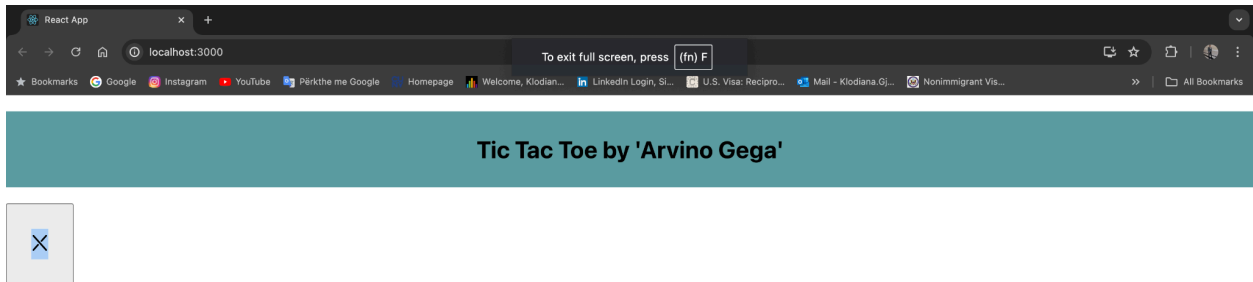
```
import './App.css';

function App() {
  return (
    <div>
      <h1 className='title'>Tic Tac Toe by 'student's name'</h1>
      <button className="square">#9932;</button>
      <button className="square">#9932;</button>
    </div>

  );
}
```

export default App;

The *localhost:3000* should look as:



Now you just need to copy-paste a few times to add nine squares and change the X with numbers from 1 to 9. **Do not forget to write your full name inside the title <h1> element:**

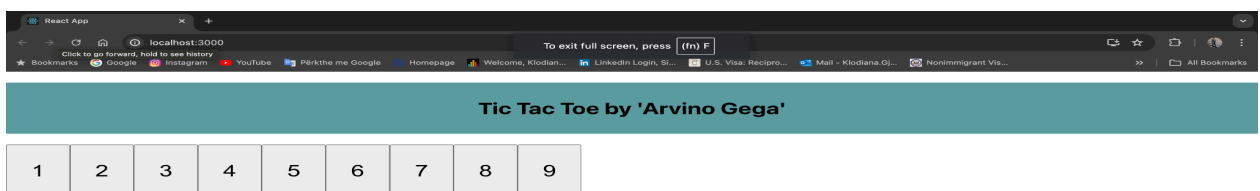
```
import './App.css';

function App() {
  return (
    <div>
      <h1 className='title'>Tic Tac Toe by 'student's name'</h1>
      <button 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>
    </div>

  );
}

export default App;
```

Activity: The *localhost:3000* should look as (have a screenshot of your *localhost:3000* and paste the image below)



Oh no! The squares are all in a single line, not in a grid like you need for our board. To fix this you'll need to group three squares into rows with divs and add some CSS classes. While you're at it, you'll give each square a number to make sure you know where each square is displayed:

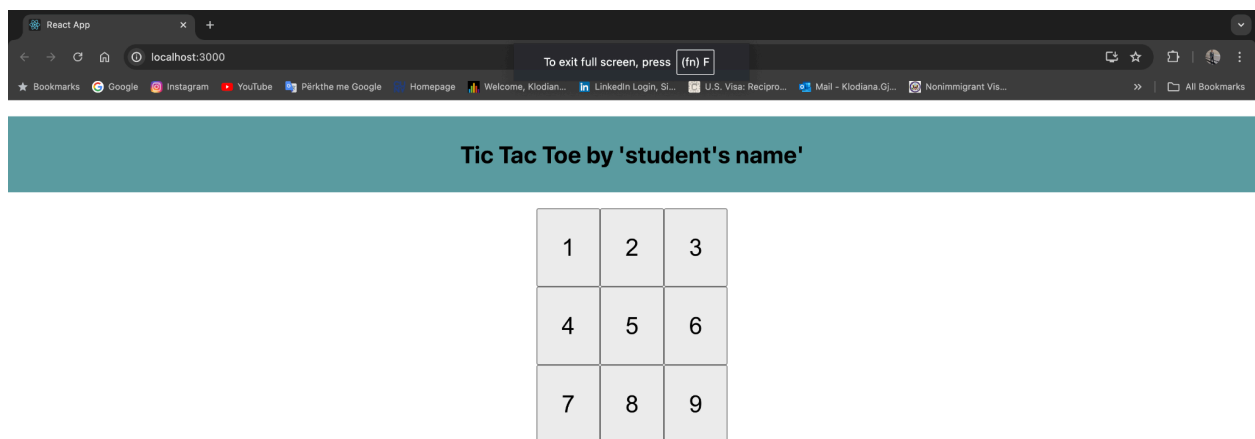
```
import './App.css';

function App() {
  return (
    <div>
      <h1 className='title'>Tic Tac Toe by 'student's name'</h1>
      <div className='boardcontainer'>
        <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>
      </div>
    </div>
  );
}
```

Click on the *style.css* file and add the following lines and save the file:

```
div.boardcontainer{
  text-align: center;
}
```


Activity: The *localhost:3000* should look as (have a screenshot of your *localhost:3000* and paste the image below)



Passing data through props

Next, you'll want to change the value of a square from empty to "X" when the user clicks on the square. With how you've built the board so far you would need to copy-paste the code that updates the square nine times (once for each square you have)! Instead of copy-pasting, React's component architecture allows you to create a reusable component to avoid messy, duplicated code.

First, you are going to copy the line defining your first square (**<button className="square">1</button>**) from your App component into a new Square component:

```
function Square() {  
  return <button className="square">1</button>;  
}
```

Then you'll update the App component to render that Square component using JSX syntax:

```
import './App.css';  
  
function Square() {  
  return <button className="square">1</button>;  
}  
  
function App() {  
  return (  
    <div>  
      <h1 className='title'>Tic Tac Toe by 'student's name'</h1>  
      <div className='boardcontainer'>  
        <div className="board-row">  
          <Square />  
          <Square />  
          <Square />  
        </div>  
        <div className="board-row">  
          <Square />  
          <Square />  
          <Square />  
        </div>  
        <div className="board-row">
```

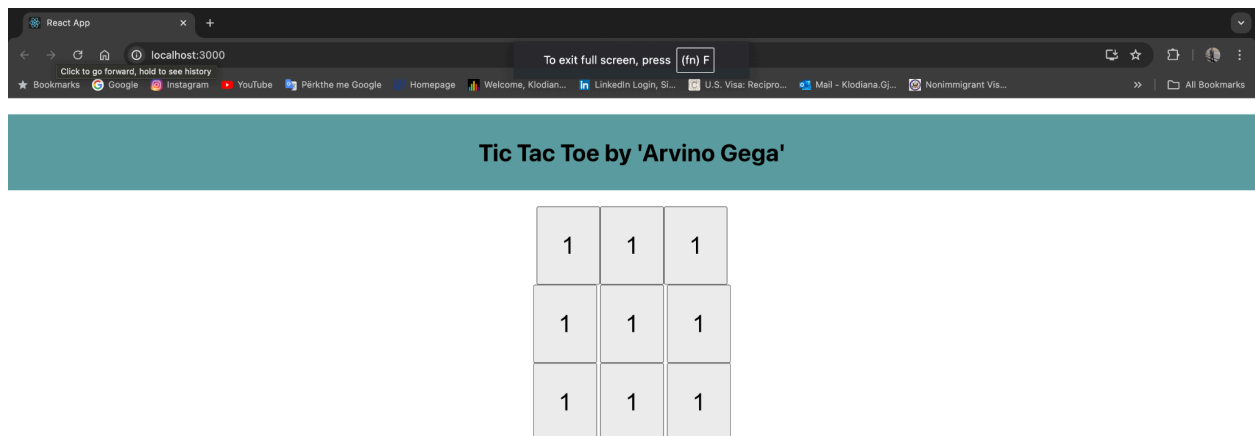
```

    <Square />
    <Square />
    <Square />
  </div>
</div>
</div>
);
}

export default App;

```

Activity: The *localhost:3000* should look as (have a screenshot of your *localhost:3000* and paste the image below)



Oh no! You lost the numbered squares you had before. Now each square says “1”. To fix this, you will use props to pass the value each square should have from the parent component (App) to its child (Square).

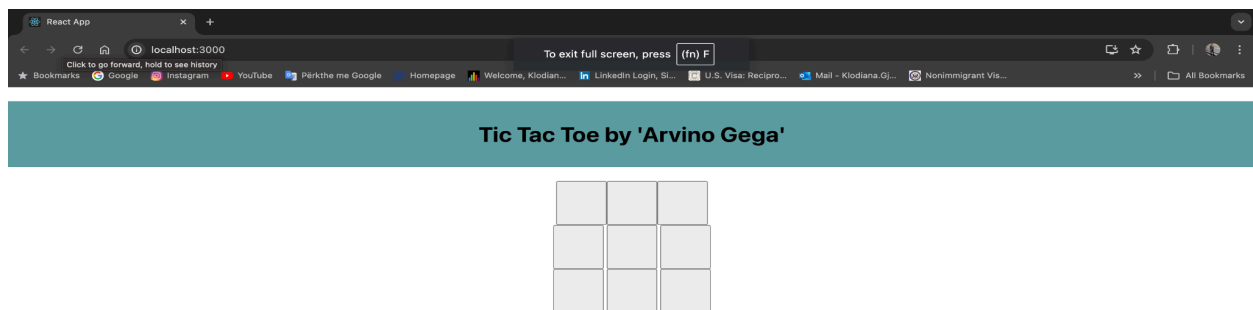
Update the Square component to read the value prop that you’ll pass from the App:

```
function Square(value) {  
  return <button className="square">1</button>;  
}
```

function Square({ value }) indicates the Square component can be passed a prop called value.

Now you want to display that value instead of 1 inside every square:

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



Activity: The *localhost:3000* should look as (have a screenshot of your *localhost:3000* and paste the image below)

This is because the App component hasn't passed the value prop to each Square component it renders yet. To fix it you'll add the value prop to each Square component rendered by the App component:

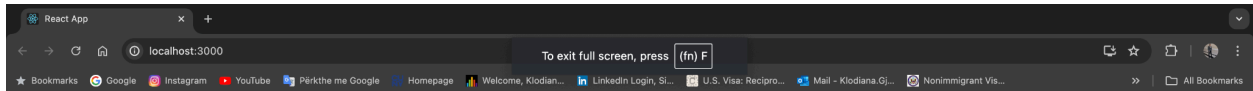
```
import './App.css';

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

function App() {
  return (
    <div>
      <h1 className='title'>Tic Tac Toe by 'student's name'</h1>
      <div className='boardcontainer'>
        <div className="board-row">
          <Square value="1" />
          <Square value="2" />
          <Square value="3" />
        </div>
        <div className="board-row">
          <Square value="4" />
          <Square value="5" />
          <Square value="6" />
        </div>
        <div className="board-row">
          <Square value="7" />
          <Square value="8" />
          <Square value="9" />
        </div>
      </div>
    </div>
  );
}

export default App;
```

Activity: The *localhost:3000* should look as (have a screenshot of your *localhost:3000* and paste the image below)



Tic Tac Toe by 'Arvino Gega'

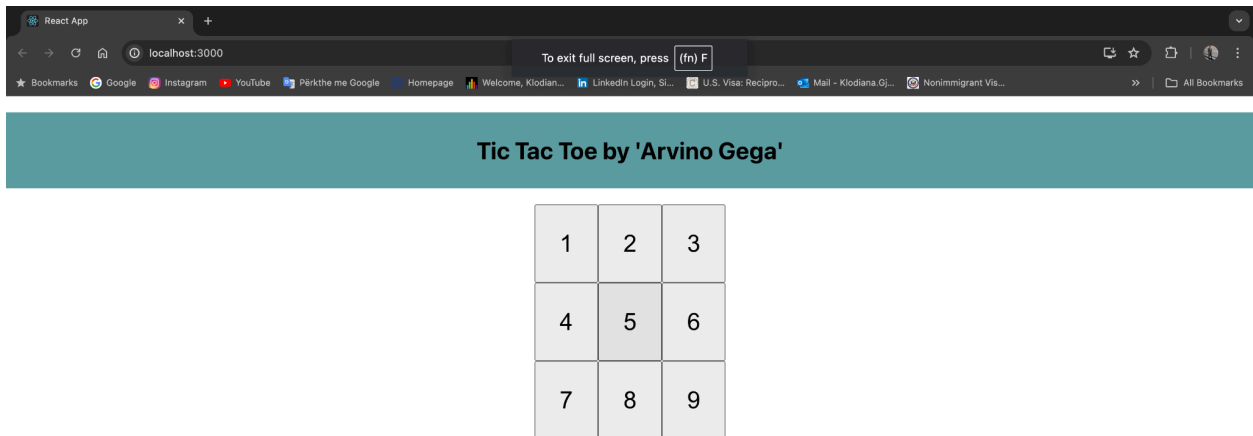
1	2	3
4	5	6
7	8	9

Making an interactive component

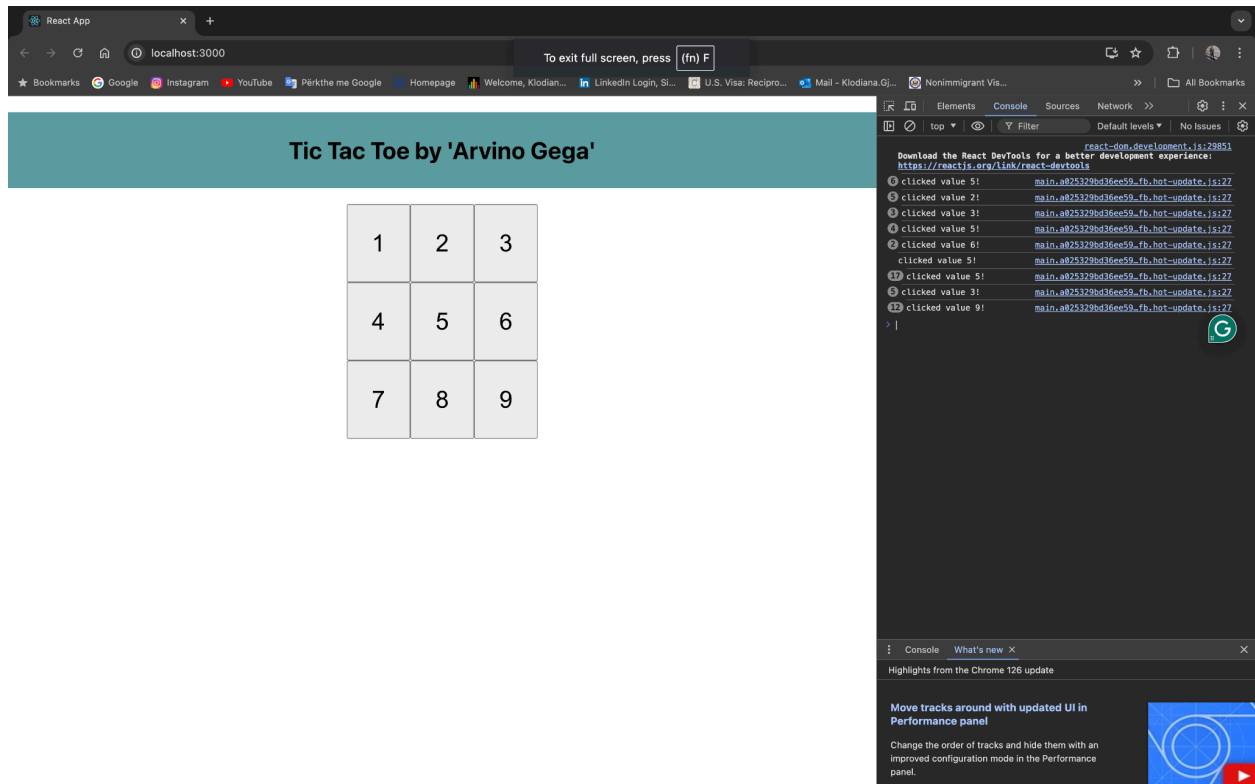
Let's fill the Square component with an X when you click it. Declare a function called **handleClick** inside of the Square. Then, add **onClick** to the props of the button JSX element returned from the Square:

```
function Square({ value }) {  
  // click function  
  function handleClick() {  
    console.log(`clicked value ${value}!`);  
  }  
  
  return (  
    <button className="square" onClick={handleClick}>  
      {value}  
    </button>  
  );  
}
```

To test the function, in the *localhost:3000* browser:



- open the developer tools, by pressing the F12 key from the keyboard.
- click the console tab.
- click on number 5 from the board.
- **Activity:** Describe what is showing in the console? Any time i press a button it will count them.



As a next step, you want the **Square** component to “remember” that it got clicked, and fill it with an “X” mark. To “remember” things, components use state.

React provides a special function called **useState** that you can call from your component to let it “remember” things. Let’s store the current value of the Square in state, and change it when the Square is clicked.

Import **useState** at the top of the file. Remove the value prop from the Square component. Instead, add a new line at the start of the Square that calls **useState**. Have it return a state variable called **value**:

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

Value stores the value and **setValue** is a function that can be used to change the value. The null passed to **useState** is used as the initial value for this state variable, so **value** here starts off equal to null.

Since the Square component no longer accepts props anymore, you’ll remove the **value** prop from all nine of the Square components created by the Board component:

```
import './App.css';
```



```

import { useState } from 'react';

function Square() {
  //use State
  const [value, setValue] = useState(null);

  // click function
  function handleClick() {
    console.log(`clicked!`);
  }

  return (
    <button className="square" onClick={handleClick}>

    </button>
  );
}

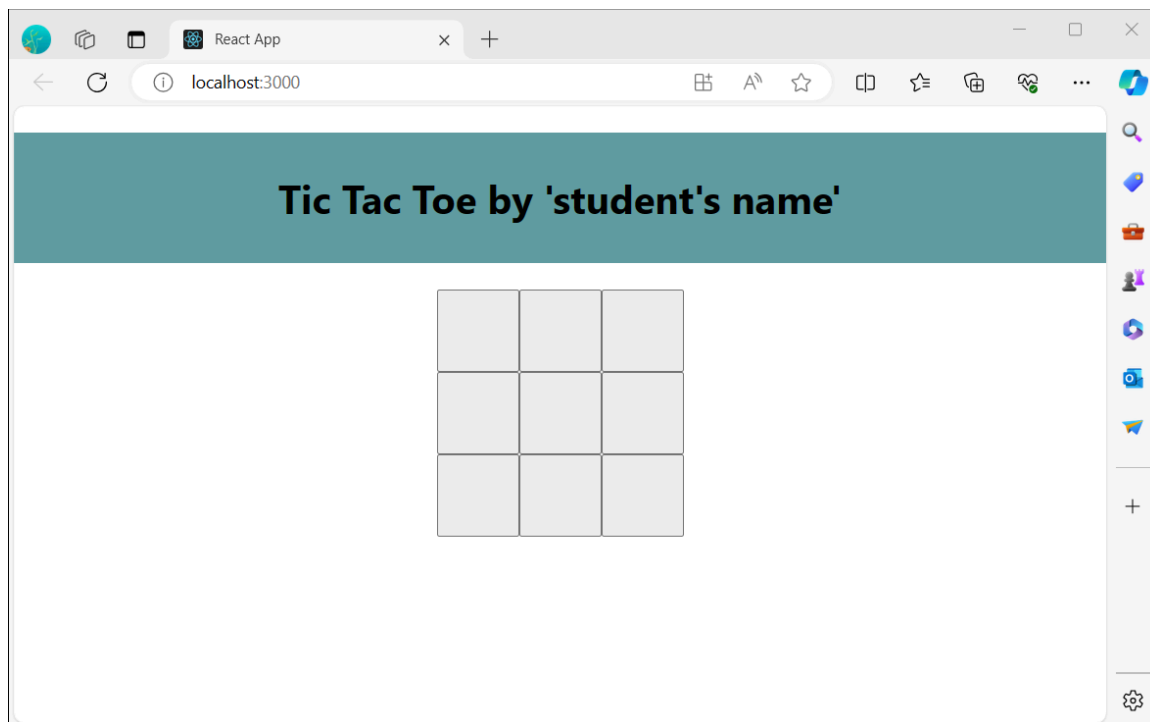
function App() {
  return (
    <div>
      <h1 className='title'>Tic Tac Toe by 'student's name'</h1>
      <div className='boardcontainer'>
        <div className="board-row">
          <Square />
          <Square />
          <Square />
        </div>
        <div className="board-row">
          <Square />
          <Square />
          <Square />
        </div>
        <div className="board-row">
          <Square />
          <Square />
          <Square />
        </div>
      </div>
    </div>
  );
}

```

```
}
```

```
export default App;
```

The `localhost:3000` should look as:



Now you'll change Square to display an "X" when clicked. Replace the `console.log("clicked!");` event handler with `setValue('X');`. Now your Square component looks like this:

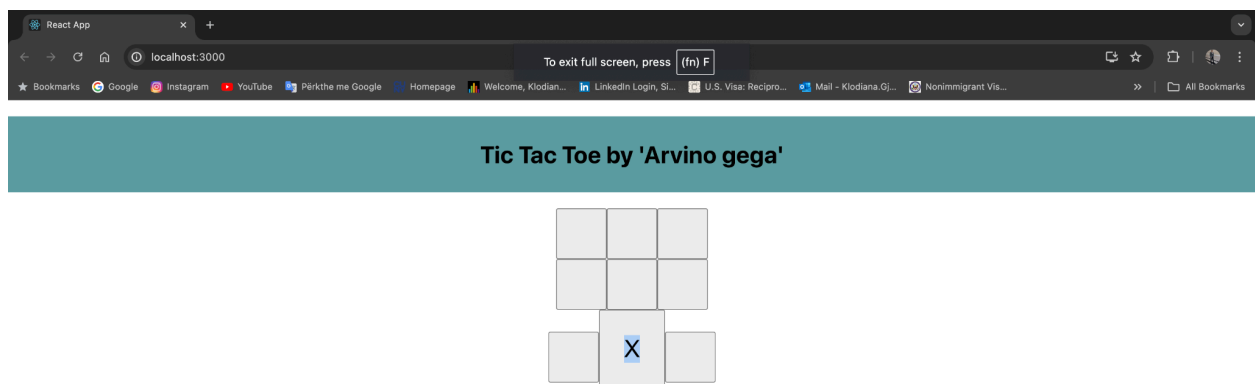
```
function Square() {  
  const [value, setValue] = useState(null);  
  
  function handleClick() {  
    setValue('X');  
  }  
  
  return (  
    <button className="square" onClick={handleClick}>  
      {value}  
    </button>  
  );  
}
```

```
}
```

By calling this set function from an **onClick** handler, you're telling React to re-render that Square whenever its `<button>` is clicked. After the update, the Square's value will be 'X', so you'll see the "X" on the game board.

Activity: to test the function, click three different cells in the board, what do you observe?

Activity: The *localhost:3000* should look as (have a screenshot of your *localhost:3000* and paste the image below)



Each Square has its own state: the value stored in each Square is completely independent of the others. When you call a set function in a component, React automatically updates the child components inside too.

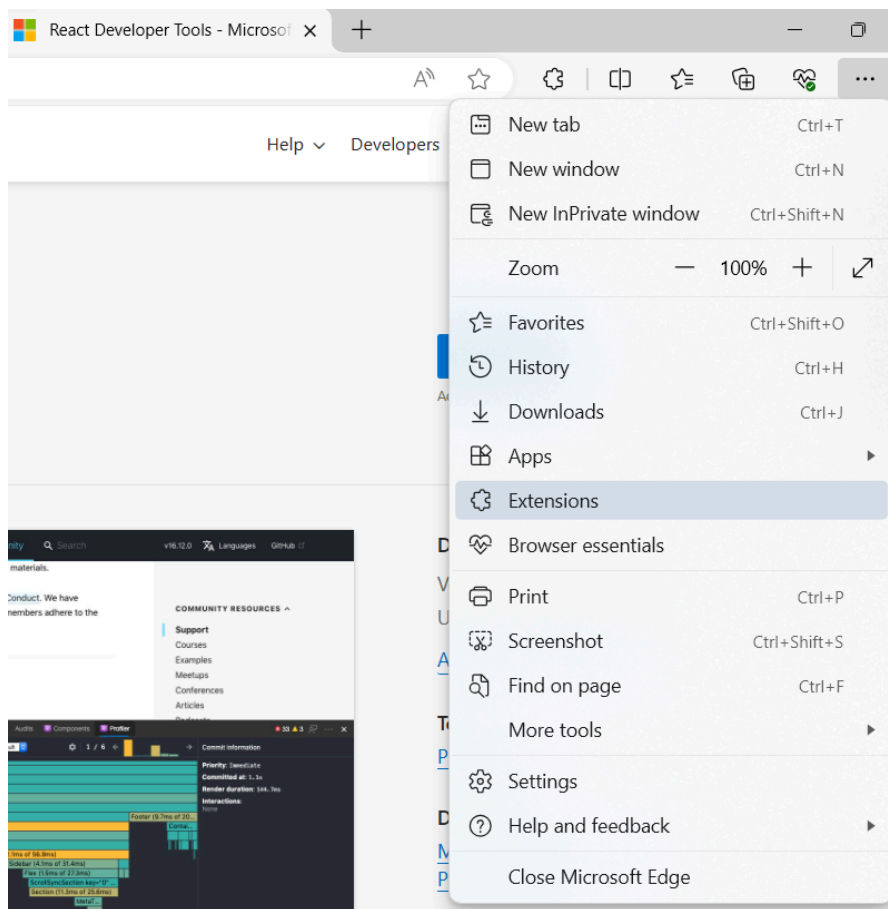
React Developer Tools

React DevTools let you check the props and the state of your React components. You can open the development tools from the internet browser by pressing the function key F12. To get access to the React developer tools, we need to install the React Tools extension.

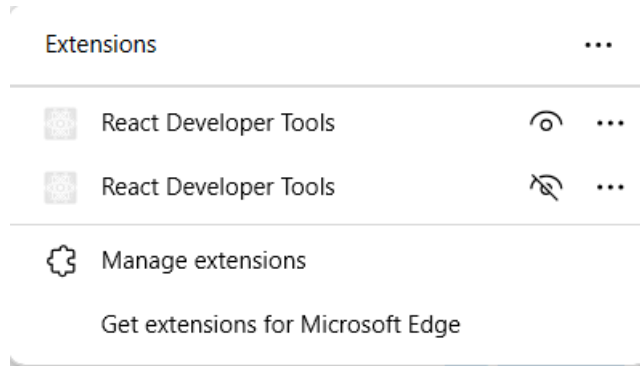
The easiest way to debug websites built with React is to install the React Developer Tools browser extension. It is available for several popular browsers:

- [Install for Chrome](#)
- [Install for Firefox](#)
- [Install for Edge](#)

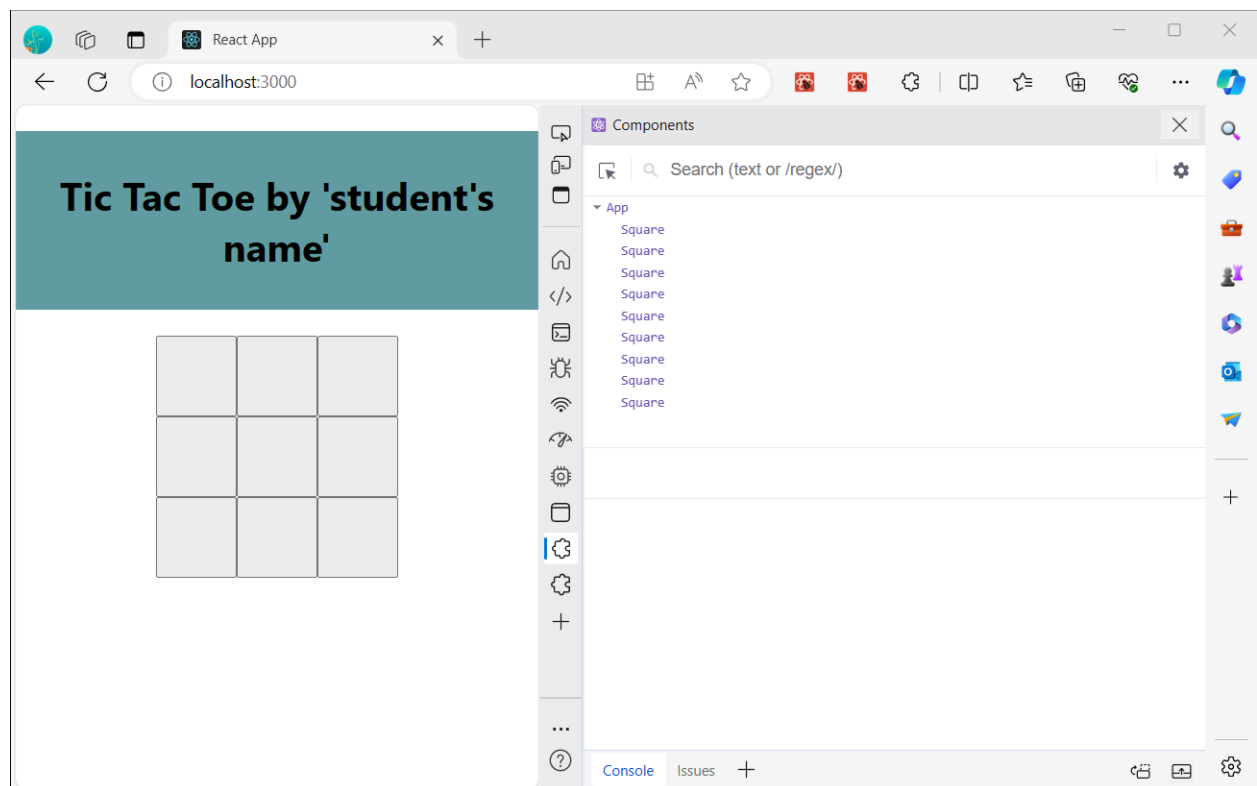
Once the extension is installed, add the extension to the internet browser:



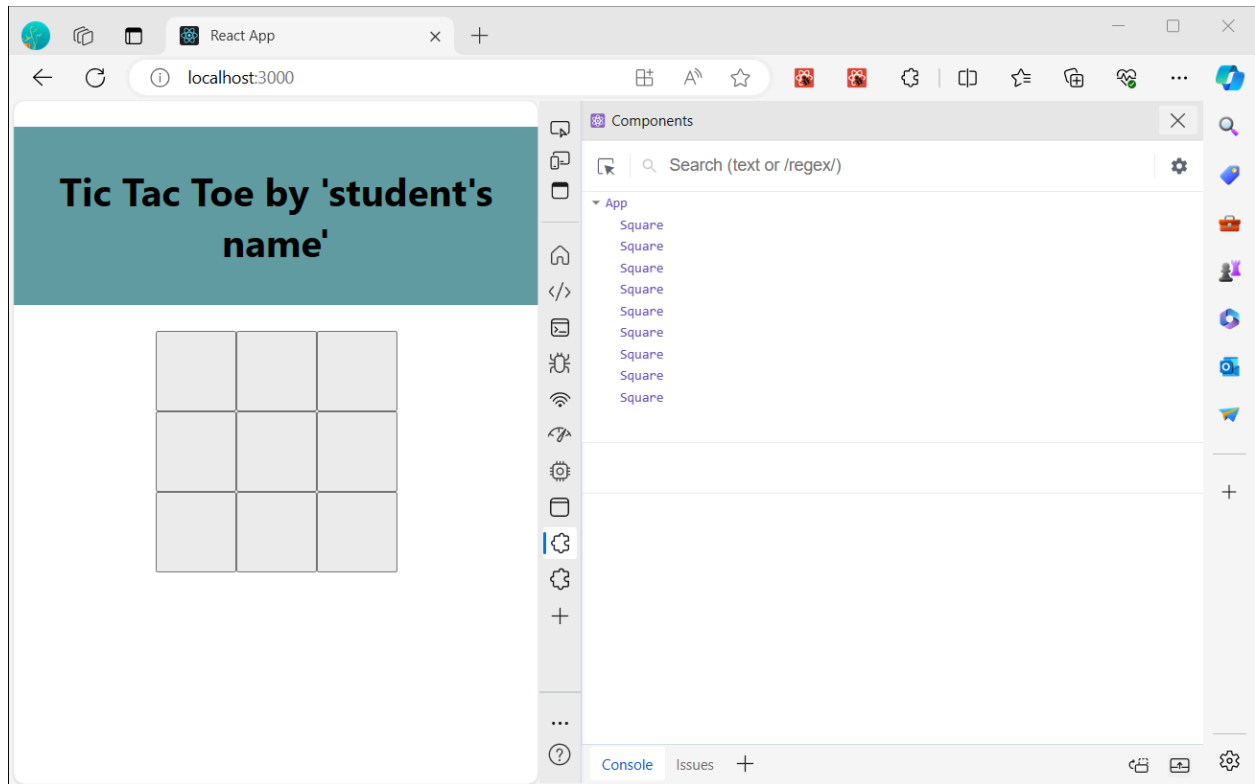
Make sure you enable the React Developer Tools



Now, if you visit a website **built with React**, you will see the *Components* and *Profiler* panels.



Open the developer tools, and "Components" and "Profiler" tabs will appear to the right.



Activity: To inspect a particular component on the tic-tac-toe board, click one of the **Square** component from the Components window. After you clicked, what do you see? _____

Completing the game

By this point, you have all the basic building blocks for your tic-tac-toe game. To have a complete game, you now need to alternate placing “X”s and “O”s on the board, and you need a way to determine a winner.

Lifting state up

Currently, each Square component maintains a part of the game’s state. To check for a winner in a tic-tac-toe game, the App would need to somehow know the state of each of the 9 Square components.

How would you approach that? At first, you might guess that the Board needs to “ask” each Square for that Square’s state. Although this approach is technically possible in React, we discourage it because the code becomes difficult to understand, susceptible to bugs, and hard

to refactor. Instead, the best approach is to store the game's state in the parent App component instead of in each Square. The App component can tell each Square what to display by passing a prop, like you did when you passed a number to each Square.

To collect data from multiple children, or to have two child components communicate with each other, declare the shared state in their parent component instead. The parent component can pass that state back down to the children via props. This keeps the child components in sync with each other and with their parent.

Lifting state into a parent component is common when React components are refactored.

Let's take this opportunity to try it out. Edit the App component so that it declares a state variable named squares that defaults to an array of 9 nulls corresponding to the 9 squares:

```
function App() {  
  //setState  
  const [squares, setSquares] = useState(Array(9).fill(null));  
  return (  
    <div>  
      <h1 className='title'>Tic Tac Toe by 'student's name'</h1>  
      <div className='boardcontainer'>  
        <div className="board-row">  
          <Square />  
          <Square />  
          <Square />  
        </div>  
        <div className="board-row">  
          <Square />  
          <Square />  
          <Square />  
        </div>  
        <div className="board-row">  
          <Square />  
          <Square />  
          <Square />  
        </div>  
      </div>  
    </div>  
  );  
}
```

Array(9).fill(null) creates an array with nine elements and sets each of them to null. The `useState()` call around it declares a `squares` state variable that's initially set to that array. Each entry in the array corresponds to the value of a square. When you fill the board in later, the `squares` array will look like this:

```
['O', null, 'X', 'X', 'X', 'O', 'O', null, null]
```

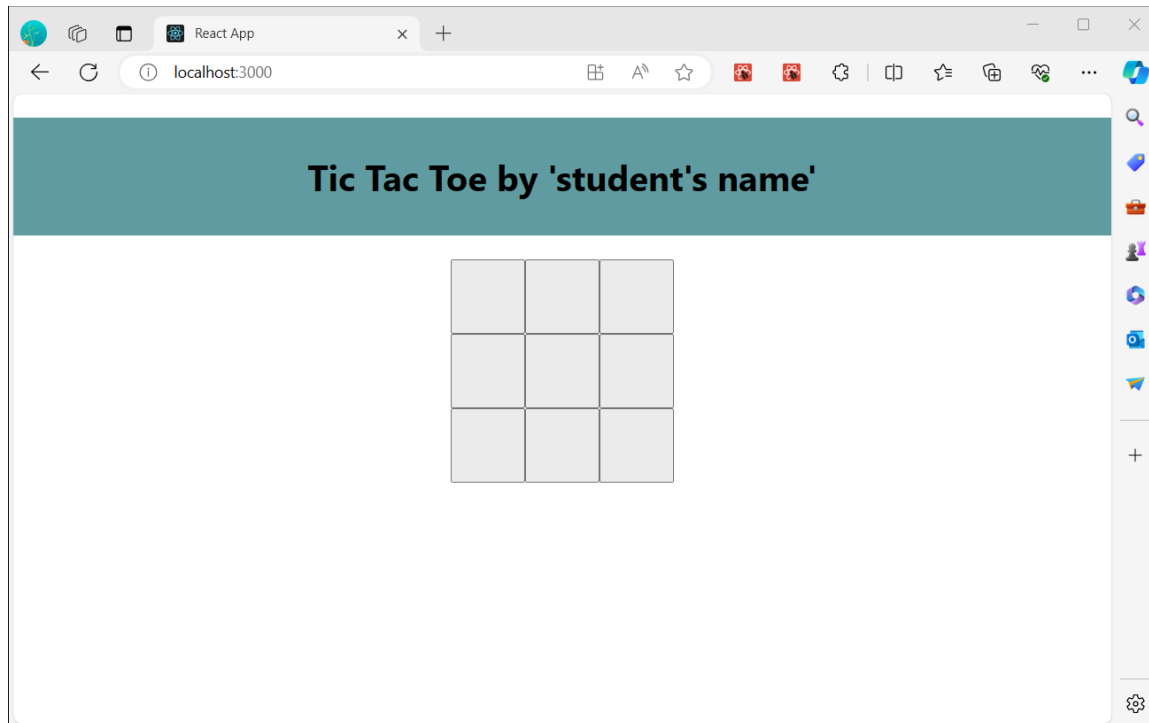
Now your `App` component needs to pass the `value` prop down to each `Square` that it renders:

```
function App() {  
  //setState  
  const [squares, setSquares] = useState(Array(9).fill(null));  
  
  return (  
    <div>  
      <h1 className='title'>Tic Tac Toe by 'student's name'</h1>  
      <div className='boardcontainer'>  
        <div className="board-row">  
          <Square value={squares[0]}/>  
          <Square value={squares[1]}/>  
          <Square value={squares[2]}/>  
        </div>  
        <div className="board-row">  
          <Square value={squares[3]}/>  
          <Square value={squares[4]}/>  
          <Square value={squares[5]}/>  
        </div>  
        <div className="board-row">  
          <Square value={squares[6]}/>  
          <Square value={squares[7]}/>  
          <Square value={squares[8]}/>  
        </div>  
      </div>  
    </div>  
  );  
}
```


Next, you'll edit the Square component to receive the value prop from the App component. This will require removing the Square component's own stateful tracking of value and the button's onClick prop:

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

At this point you should see an empty tic-tac-toe board:



Each Square will now receive a value prop that will either be 'X', 'O', or null for empty squares.

Next, you need to change what happens when a Square is clicked. The App component now maintains which squares are filled. You'll need to create a way for the Square to update the App's state. Since state is private to a component that defines it, you cannot update the App's state directly from Square.

Instead, you'll pass down a function from the App component to the Square component, and you'll have Square call that function when a square is clicked. You'll start with the function that the Square component will call when it is clicked. You'll call that function onSquareClick:

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

Next, you'll add the onSquareClick function to the Square component's props:

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

Now you'll connect the onSquareClick prop to a function in the App component that you'll name handleClick. To connect onSquareClick to handleClick you'll pass a function to the onSquareClick prop of the first Square component:

```
function App() {  
  //setState  
  const [squares, setSquares] = useState(Array(9).fill(null));  
  
  return (  
    <div>  
      <h1 className='title'>Tic Tac Toe by 'student's name'</h1>  
      <div className='boardcontainer'>  
        <div className="board-row">  
          <Square value={squares[0]} onSquareClick={handleClick}/>  
          <Square value={squares[1]}/>  
          <Square value={squares[2]}/>  
        </div>  
        <div className="board-row">  
          <Square value={squares[3]}/>  
          <Square value={squares[4]}/>  
          <Square value={squares[5]}/>  
        </div>  
      </div>  
    </div>  
  );  
}
```

```

    <div className="board-row">
      <Square value={squares[6]}/>
      <Square value={squares[7]}/>
      <Square value={squares[8]}/>
    </div>
  </div>
</div>
);}

```

Lastly, you will define the **handleClick** function inside the App component to update the squares array holding your App's state:

```

function App() {
  //setState
  const [squares, setSquares] = useState(Array(9).fill(null));

  // function to update the squares array holding your App's state:
  function handleClick() {
    const nextSquares = squares.slice();
    nextSquares[0] = "X";
    setSquares(nextSquares);
  }

  return (
    <div>
      <h1 className='title'>Tic Tac Toe by 'student's name'</h1>
      <div className='boardcontainer'>
        <div className="board-row">
          <Square value={squares[0]} onClick={handleClick}/>
          <Square value={squares[1]}/>
          <Square value={squares[2]}/>
        </div>
        <div className="board-row">
          <Square value={squares[3]}/>
          <Square value={squares[4]}/>
          <Square value={squares[5]}/>
        </div>
        <div className="board-row">
          <Square value={squares[6]}/>

```

```

    <Square value={squares[7]}/>
    <Square value={squares[8]}/>
  </div>
</div>
</div> ); }

```

The handleClick function creates a copy of the squares array (nextSquares) with the JavaScript slice() Array method. Then, handleClick updates the nextSquares array to add X to the first ([0] index) square.

Calling the setSquares function lets React know the state of the component has changed. This will trigger a re-render of the components that use the squares state (App) as well as its child components (the Square components that make up the board).

Note

JavaScript supports [closures](#) which means an inner function (e.g. handleClick) has access to variables and functions defined in a outer function (e.g. App). The handleClick function can read the squares state and call the setSquares method because they are both defined inside of the App function.

Now you can add X's to the board... but only to the upper left square. Your handleClick function is hardcoded to update the index for the upper left square (0). Let's update handleClick to be able to update any square. Add an argument i to the handleClick function that takes the index of the square to update:

// function to update the squares array holding your App's state:

```

function handleClick(i) {
  const nextSquares = squares.slice();
  nextSquares[i] = "X";
  setSquares(nextSquares);
}

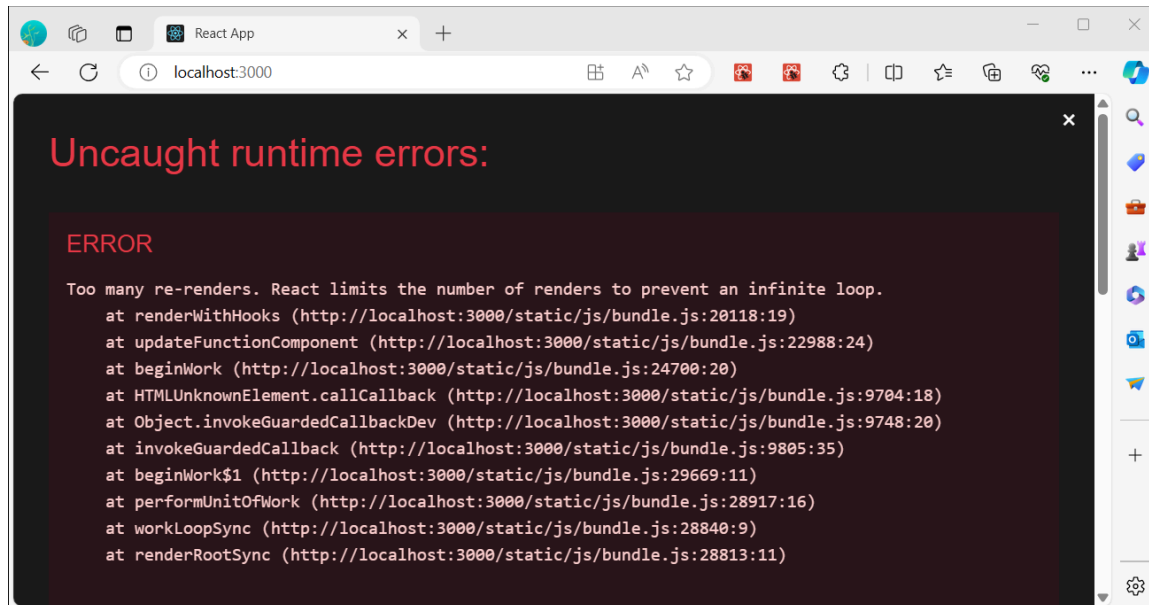
```

Next, you will need to pass that i to handleClick. You could try to set the onSquareClick prop of square to be handleClick(0) directly in the JSX like this, but it won't work:

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

Here is why this doesn't work. The handleClick(0) call will be a part of rendering the board component. Because handleClick(0) alters the state of the board component by calling

setSquares, your entire board component will be re-rendered again. But this runs handleClick(0) again, leading to an infinite loop:



Why didn't this problem happen earlier?

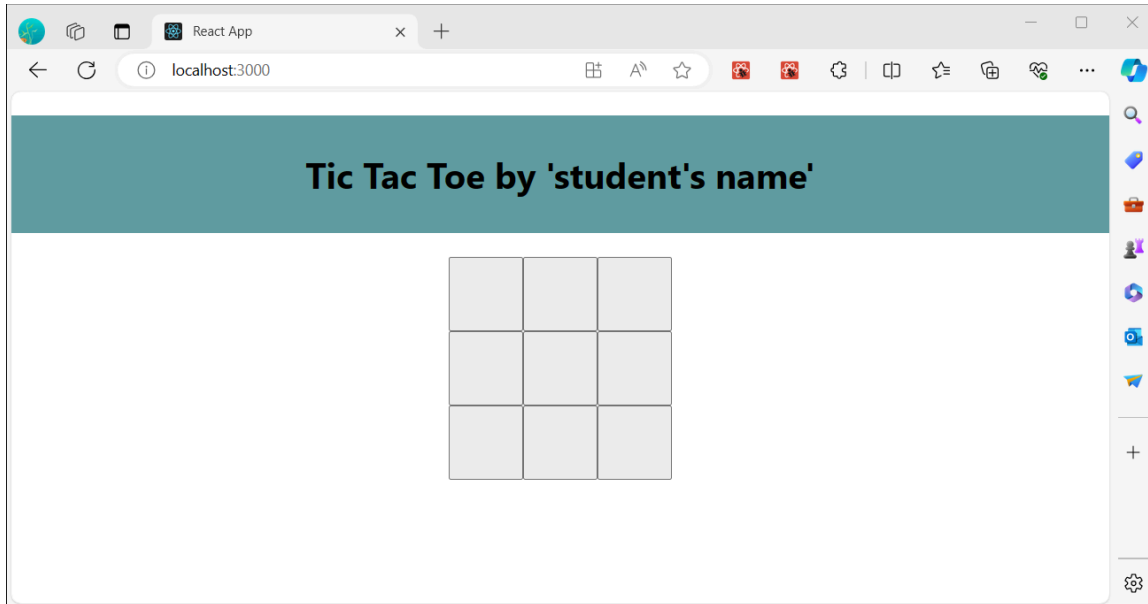
When you were passing `onSquareClick={handleClick}`, you were passing the handleClick function down as a prop. You were not calling it! But now you are calling that function right away—notice the parentheses in `handleClick(0)`—and that's why it runs too early. You don't want to call handleClick until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

After making this change, the localhost:3000 should look as:



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 you need to update the other eight squares to call `handleClick` from the arrow functions you pass. Make sure that the argument for each call of the `handleClick` corresponds to the index of the correct square:

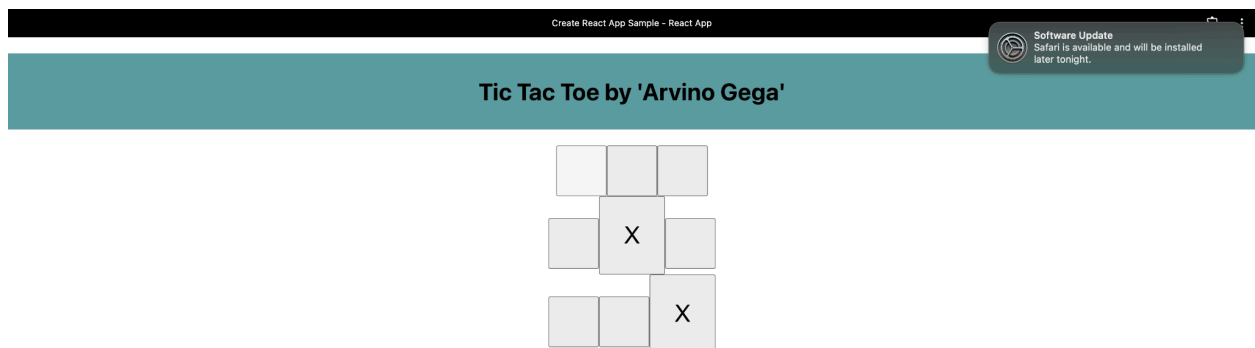
```
return (  
  <div>  
    <h1 className='title'>Tic Tac Toe by 'student's name'</h1>  
    <div className='boardcontainer'>  
      <div className="board-row">  
        <Square value={squares[0]} onClick={() => handleClick(0)} />  
        <Square value={squares[1]} onClick={() => handleClick(1)} />  
        <Square value={squares[2]} onClick={() => handleClick(2)} />  
      </div>  
      <div className="board-row">  
        <Square value={squares[3]} onClick={() => handleClick(3)} />  
        <Square value={squares[4]} onClick={() => handleClick(4)} />  
        <Square value={squares[5]} onClick={() => handleClick(5)} />  
      </div>  
      <div className="board-row">
```

```

    <Square value={squares[6]} onSquareClick={() => handleClick(6)} />
    <Square value={squares[7]} onSquareClick={() => handleClick(7)} />
    <Square value={squares[8]} onSquareClick={() => handleClick(8)} />
  </div>
</div>
</div>
);
}

```

Activity: Now you can again add X's to any square on the board by clicking on them. Click on three different cells in the board and have a screenshot of the localhost:3000. Paste the image below:



But this time all the state management is handled by the App component!

Now that your state handling is in the App component, the parent App component passes props to the child Square components so that they can be displayed correctly. When clicking on a Square, the child Square component now asks the parent App component to update the state of the App. When the App's state changes, both the App component and every child Square re-renders automatically. Keeping the state of all squares in the App component will allow it to determine the winner in the future.

Let's recap what happens when a user clicks the top left square on your board to add an X to it:

Clicking on the upper left square runs the function that the button received as its onClick prop from the Square. The Square component received that function as its onSquareClick prop from the App.

1. The App component defined that function directly in the JSX. It calls handleClick with an argument of 0.
2. handleClick uses the argument (0) to update the first element of the squares array from null to X.
3. The squares state of the App component was updated, so the App and all of its children re-render. This causes the value prop of the Square component with index 0 to change from null to X.

In the end the user sees that the upper left square has changed from empty to having a X after clicking it.

Note

The DOM <button> element's onClick attribute has a special meaning to React because it is a built-in component. For custom components like Square, the naming is up to you. You could give any name to the Square's onSquareClick prop or Board's handleClick function, and the code would work the same. In React, it's conventional to use onSomething names for props which represent events and handleSomething for the function definitions which handle those events.

Why immutability is important

Note how in handleClick, you call .slice() to create a copy of the squares array instead of modifying the existing array. To explain why, we need to discuss immutability and why immutability is important to learn.

There are generally two approaches to changing data. The first approach is to mutate the data by directly changing the data's values. The second approach is to replace the data with a new copy which has the desired changes. Here is what it would look like if you mutated the squares array:

```
const squares = [null, null, null, null, null, null, null, null, null];
squares[0] = 'X';
// Now `squares` is ["X", null, null, null, null, null, null, null, null];
```

And here is what it would look like if you changed data without mutating the squares array:

```
const squares = [null, null, null, null, null, null, null, null, null];
const nextSquares = ['X', null, null, null, null, null, null, null, null];
// Now `squares` is unchanged, but `nextSquares` first element is 'X' rather
than `null`
```

The result is the same but by not mutating (changing the underlying data) directly, you gain several benefits.

Immutability makes complex features much easier to implement. Later in this tutorial, you will implement a “time travel” feature that lets you review the game’s history and “jump back” to past moves. This functionality isn’t specific to games—an ability to undo and redo certain actions is a common requirement for apps. Avoiding direct data mutation lets you keep previous versions of the data intact, and reuse them later.

There is also another benefit of immutability. By default, all child components re-render automatically when the state of a parent component changes. This includes even the child components that weren’t affected by the change. Although re-rendering is not by itself noticeable to the user (you shouldn’t actively try to avoid it!), you might want to skip re-rendering a part of the tree that clearly wasn’t affected by it for performance reasons. Immutability makes it very cheap for components to compare whether their data has changed or not. You can learn more about how React chooses when to re-render a component in [the memo API reference](#).

Taking turns

It's now time to fix a major defect in this tic-tac-toe game: the "O"s cannot be marked on the board.

You'll set the first move to be "X" by default. Let's keep track of this by adding another piece of state to the Board component:

```
// change state, from X to O
const [xlsNext, setXlsNext] = useState(true);
//setState
const [squares, setSquares] = useState(Array(9).fill(null));
```

Each time a player moves, xlsNext (a boolean) will be flipped to determine which player goes next and the game's state will be saved. You'll update the Board's handleClick function to flip the value of xlsNext:

```
// function to update the squares array holding your App's state:
function handleClick(i) {
  const nextSquares = squares.slice();
```

```
  //condition to change state
  if (xlsNext) {
    nextSquares[i] = "X";
  } else {
    nextSquares[i] = "O";
  }
```

```
  setSquares(nextSquares);
  setXlsNext(!xlsNext);
}
```

Now, as you click on different squares, they will alternate between X and O, as they should!

Activity: But wait, there's a problem. Try clicking on the same square multiple times.

What symbol, X or O, is showing after several clicks? Describe the behavior: By clicking multiple time in the same square we are able to change the square from X to O.

The X is overwritten by an O! While this would add a very interesting twist to the game, we're going to stick to the original rules for now.

When you mark a square with a X or an O you aren't first checking to see if the square already has a X or O value. You can fix this by returning early. You'll check to see if the square already has a X or an O. If the square is already filled, you will return in the handleClick function early—before it tries to update the App state.

```
// function to update the squares array holding your App's state:
```

```
function handleClick(i) {
```

```
  //condition to lock the each cell once it's clicked
```

```
  if (squares[i]) {
```

```
    return;
```

```
  }
```

```
  const nextSquares = squares.slice();
```

```
  //condition to change state
```

```
  if (xIsNext) {
```

```
    nextSquares[i] = "X";
```

```
  } else {
```

```
    nextSquares[i] = "O";
```

```
  }
```

```

setSquares(nextSquares);
setXIsNext(!xIsNext);
}

```

Activity: Now you can only add X's or O's to empty squares! Try the following in the App:

- Pick a cell and click on it multiple times. Describe what happen to the cell? _____

Wherever we click on the squares the first value is X

- Click on two more different cells. Describe what happen to three cells? _____ By

clicking on multiple squares we will se that the value it will be (X to O) —(X to O) and so

on

Activity: The *localhost:3000* should look as (have a screenshot of your *localhost:3000* and paste the image below)



Declaring a winner

Now that the players can take turns, you'll want to show when the game is won and there are no more turns to make. To do this you'll add a helper function called **calculateWinner** that takes an array of 9 squares, checks for a winner and returns 'X', 'O', or null as appropriate. Don't worry too much about the calculateWinner function; it's not specific to React:

```
// function to calculate the points for each X and O
function calculateWinner(squares) {
  const lines = [
    [0, 1, 2],
    [3, 4, 5],
    [6, 7, 8],
    [0, 3, 6],
    [1, 4, 7],
    [2, 5, 8],
    [0, 4, 8],
    [2, 4, 6]
  ];
  for (let i = 0; i < lines.length; i++) {
    const [a, b, c] = lines[i];
    if (squares[a] && squares[a] === squares[b] && squares[a] === squares[c]) {
      return squares[a];
    }
  }
  return null;
}
```

Note

It does not matter whether you define calculateWinner before or after the App. Let's put it at the end so that you don't have to scroll past it every time you edit your components.

You will call calculateWinner(squares) in the App component's handleClick function to check if a player has won. You can perform this check at the same time you check if a user has clicked a square that already has a X or and O. We'd like to return early in both cases:

```
// function to update the squares array holding your App's state:
function handleClick(i) {
```

```

//condition to lock the each cell once it's clicked
if (squares[i] || calculateWinner(squares)) {
  return;
}

const nextSquares = squares.slice();

//condition to change state
if (xIsNext) {
  nextSquares[i] = "X";
} else {
  nextSquares[i] = "O";
}

setSquares(nextSquares);
setXIsNext(!xIsNext);
}

```

To let the players know when the game is over, you can display text such as “Winner: X” or “Winner: O”. To do that you’ll add a status section to the App component. The status will display the winner if the game is over and if the game is ongoing you’ll display which player’s turn is next. Add the following lines after the handleClick(i) function

```

const winner = calculateWinner(squares);
let status;
if (winner) {
  status = "Winner: " + winner;
} else {
  status = "Next player: " + (xIsNext ? "X" : "O");
}

```

After it, add `<div className="status">{status}</div>` inside the **boardcontainer**

```

return (
  <div>
    <h1 className='title'>Tic Tac Toe by 'student's name'</h1>
    <div className='boardcontainer'>

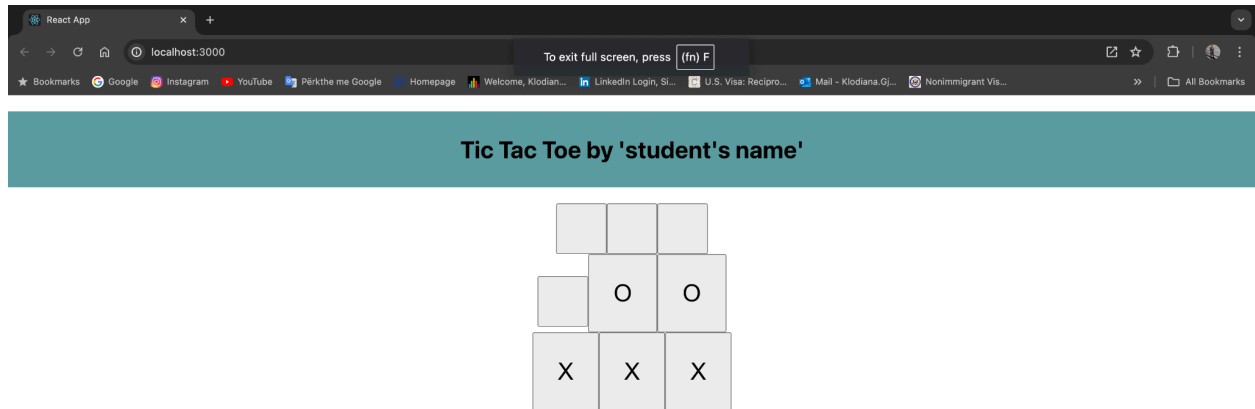
```

```
<div className="status">{status}</div>
  <div className="board-row">
    <Square value={squares[0]} onSquareClick={() => handleClick(0)} />
    <Square value={squares[1]} onSquareClick={() => handleClick(1)} />
    <Square value={squares[2]} onSquareClick={() => handleClick(2)} />
  </div>
  <div className="board-row">
    <Square value={squares[3]} onSquareClick={() => handleClick(3)} />
    <Square value={squares[4]} onSquareClick={() => handleClick(4)} />
    <Square value={squares[5]} onSquareClick={() => handleClick(5)} />
  </div>
  <div className="board-row">
    <Square value={squares[6]} onSquareClick={() => handleClick(6)} />
    <Square value={squares[7]} onSquareClick={() => handleClick(7)} />
    <Square value={squares[8]} onSquareClick={() => handleClick(8)} />
  </div>
</div>
);
```

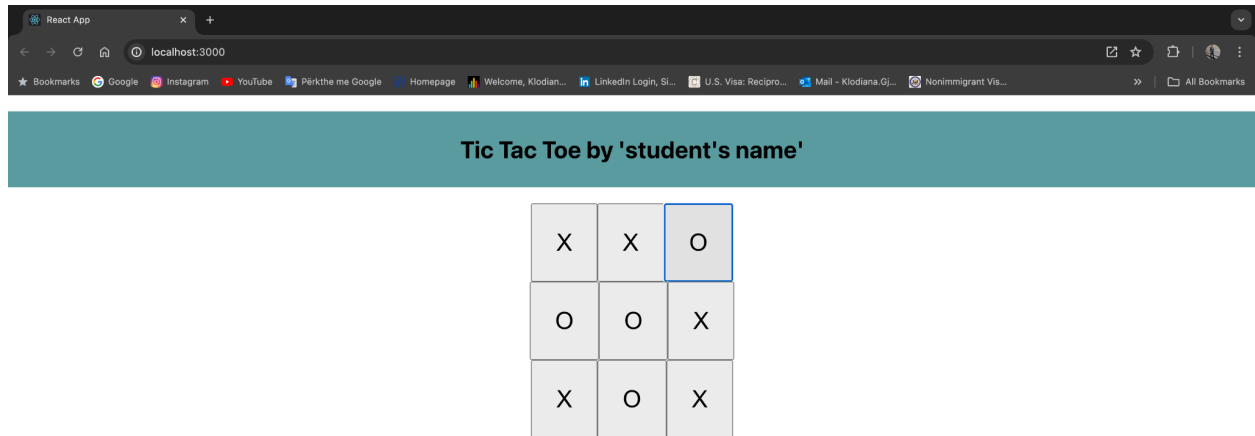
Congratulations! You now have a working tic-tac-toe game. And you've just learned the basics of React too. So you are the real winner here.


Activity: The final test that the App works is to run to set of game:

- Make different moves and complete three X in a row, column, or diagonal. After it, take a screenshot of the localhost:3000, and paste the image below:



- Make different moves until it completes the board without any winners. After it, take a screenshot of the localhost:3000, and paste the image below:



-----  Congratulations! You have completed the homework -----