ENSF-381: **Full Stack Web Development Laboratory**

Ahmad Abdellatif and Novarun Deb
Department of Electrical & Software Engineering
University of Calgary

Lab 7

Objectives

Welcome to the ENSF381 course lab! In this lab, we will build on what we have learned in React by designing a Tic-Tac-Toe game. This lab will help reinforce key concepts like component structure, state management, and event handling while giving you hands-on experience with React's core principles.

Groups

Lab instructions must be followed in groups **of two students**.

Submission

You must submit the complete source code, ensuring it can be executed without any modifications. Also, if requested by the instructor, you may need to submit the corresponding documentation file (e.g., word and image). Only one member of the group needs to submit the assignment, but the submission must include the names and UCIDs of all group members at the top of the code.

Deadline

Lab exercises must be submitted by **11:55 PM on the same day as the lab session**. Submissions made within 24 hours after the deadline will receive a maximum of 50% of the mark. Submissions made beyond 24 hours will not be evaluated and will receive a grade of zero.

Academic Misconduct

Academic Misconduct refers to student behavior which compromises proper assessment of a student's academic activities and includes: cheating; fabrication;

falsification; plagiarism; unauthorized assistance; failure to comply with an instructor's expectations regarding conduct required of students completing academic assessments in their courses; and failure to comply with exam regulations applied by the Registrar.

For more information on the University of Calgary Student Academic Misconduct Policy and Procedure and the SSE Academic Misconduct Operating Standard, please visit: https://schulich.ucalgary.ca/current-students/undergraduate/student-resources/policies-and-procedures

This lab has been modified for this course based on the Tic-Tac-Toe tutorial.

Create a React App

- Open your terminal and create a new React app using the command:

```
npx create-react-app tic-tac-toe
```

- Navigate to the **tic-tac-toe** folder and open the project in your IDE.

Understanding the Starter Code

Now open the **App component (src/App.js)** in your project folder. This file establishes the foundation for the tic-tac-toe game and includes the necessary CSS styles, allowing us to focus on learning React and building the game's functionality.
The code is structured around three main components:

- **Square** – A React component that renders a single <button>.
- **Board** – A React component that displays a 3x3 grid of 9 squares.
- **calculateWinner** – A function that contains the game's logic for determining the winner based on the current state of the squares. This function evaluates the state of the board and returns the winner, if any, based on the updated logic. It will also helps us keep track of the game's state and render the board with the correct values.

At this stage, the components are not yet interactive, but that will change soon as we set up communication between them.
Note: All the three components need to be placed in App.js file.

Building the Board

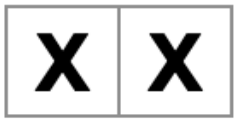
Open **App.js** and create a function called **Square**. We want to build a Tic-Tac-Toe board, but right now, there is only one square:

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

At the moment, the board contains just one square, but we need nine. If you try duplicating the square to create two, your code might look something like this:

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

Now you should see (**without the style**):



To create a full 3x3 grid, duplicate the square code a few more times. However, you will notice that the squares are arranged in a single line rather than a grid.

To fix this, we will group the squares into rows using elements and apply CSS styles. Adding numbers to each square will also help keep track of their positions.

Copy the following styles **into src/App.css**

```
.board-row {
  display: flex;
  width: 300px;
}
.square {
  width: 100px;
  height: 100px;
  font-size: 24px;
  text-align: center;
  vertical-align: middle;
  border: 1px solid black;
}
```

In App.js, update the Square component like this:

```
function Square() {
 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>
  </>
 );
}
```

Now you should see:

1	2	3
4	5	6
7	8	9

Passing Data Through Props

Next, we want each square to show an "X" when clicked. Instead of writing the same code for each of the 9 squares, we can use React's reusable component system. This enables us to create a single Square component that can be used for all squares, making our code simpler and more efficient.

Start by creating a new function named Board to manage the grid layout. This will help organize our code and make it easier to update the board as we add more functionality.

```
export default function Board() {
  //...
}
```

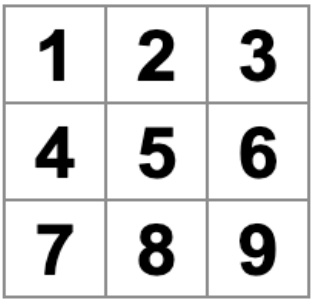
Next, we will move the square rendering logic into the Board component. After that, update the Board component to pass a value (i.e., the number) as a prop to each Square. Finally, modify the Square component to accept and handle this prop accordingly:

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

Update the Board component to pass values to Square:

```
export default function Board() {
  return (
    <>
      <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>
    </>
  );}
```

Now you should see a grid of numbers:



Making the Squares Interactive

Now, let us enable each square to display an "X" when clicked.

- **Handling Click Events**

Start by adding an event handler inside the Square component:

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

Now, when you click a square, "clicked!" appears in the console. If you click multiple times, the browser – depending on the browser - might not log duplicate lines but will instead display a counter next to the message.

- ## Storing Clicked State

Now, we want each square to "remember" when it has been clicked and update the text it displays to "X". In React, components use state to store information that changes over time.
React provides a built-in function called useState for managing state within components. We will use it to store the square's value and update it when clicked.

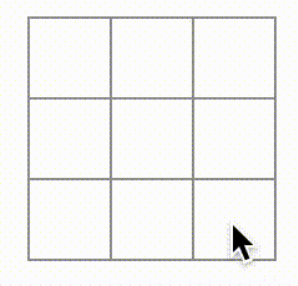
Step1: Import useState at the top of the file

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

Step2: Remove the value prop from Square and initialize state

```
function Square() {
  const [value, setValue] = useState(null);
}
```

Now, each square will handle its own state and update accordingly when clicked.



Since the Square component no longer accepts props, we need to remove the value prop from all nine of the Square components created by the Board component:

```
// ...
export default function Board() {
  return (
    <>
      <div className="board-row">
        <Square />
        <Square />
        <Square />
      </div>
      <div className="board-row">
        <Square />
        <Square />
        <Square />
      </div>
      <div className="board-row">
        <Square />
        <Square />
        <Square />
      </div>
    </>
  );
}
```

Updating State on Click

Now, let us update the Square component so that clicking it changes its value to “X”.

Replace console.log(“clicked!”) with setValue(“X”):

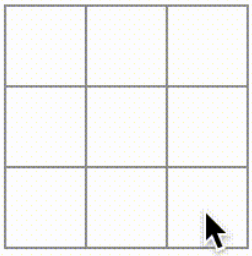
```
function Square() {
  const [value, setValue] = useState(null);

  function handleClick() {
    setValue('X');
  }
  return (
    <button
      className="square"
      onClick={handleClick}
    >
      {value}
    </button>
  );
}
```

By calling setValue("X") inside handleClick, we tell React to update the state and re-render the Square when it's clicked. After this change, clicking a square will display "X".

Independent State for Each Square

Each Square maintains its own state, so clicking one square doesn’t affect the others. When a component's (in this case, Square) state updates, React automatically re-renders that component along with any necessary child components.



Completing the Game

With the Tic-Tac-Toe board set up, it is time to add interactivity. The next steps include:

- Alternating turns between “X” and “O” when clicking squares.
- Implementing logic to determine the winner.

Managing State in the Board Component

To keep track of the game state, we need to update the Board component to store the state of all 9 squares. We will use a state variable called squares, initialized as an array of nine null values, indicating that all squares are empty at the start.

```
// ...
export default function Board() {
  const [squares, setSquares] = useState(Array(9).fill(null));
  return (
    // ...
  );
}
```

The expression Array(9).fill(null) creates an array with nine elements, all set to null. The useState hook initializes the squares state variable with this array. Each element in the array represents a square on the board.

As players take turns, the squares array will be updated to reflect their moves. For example:

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

Passing State to the Squares

Now, each Square will receive its value as a prop from the Board component:

```
<Square value={squares[i]} />
```

Each Square now gets a value prop, which can be 'X', 'O', or null (for the empty squares). To reflect this change, update the Square component to receive the value prop from Board. Since the Board component manages state, we can remove the internal state from Square **(remove handleClick function and useState from Square function):**

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

At this stage, the Tic-Tac-Toe board should be rendered, but the squares will not be interactive yet. To allow players to interact with the board, we need to update the state when a square is clicked. However, since Board owns the state, Square cannot update it directly.

To solve this, we will pass a function from Board to Square. When a square is clicked, it will call this function to notify Board about the interaction. The function we pass will call setSquares, which is necessary to make changes to the Board's state.

Handling Click Events

Start by creating a function, called onSquareClick, in Board that Square can call when clicked:

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

Next, connect onSquareClick to an actual function inside Board that updates the game state. To do this, pass a function to the onSquareClick prop of each Square component:

```
export default function Board() {
  const [squares, setSquares] = useState(Array(9).fill(null));

  return (
    <>
      <div className="board-row">
        <Square value={squares[0]} onSquareClick={handleClick} />
        //...
      </div>
    </>
  );
}
```

Finally, define the handleClick function inside Board. This function will modify the squares array and update the board's state using setSquares:

```
export default function Board() {
  const [squares, setSquares] = useState(Array(9).fill(null));

  function handleClick() {
    const nextSquares = squares.slice();
    nextSquares[0] = "X";
    setSquares(nextSquares);
  }

  return (
    // ...
  )
}
```

The handleClick function first creates a copy of the squares array using slice(). It then updates the value of the first square (index 0) to 'X'. Finally, calling setSquares notifies React to re-render the Board and its child components, including Square.

Why does this work?
JavaScript closures allow handleClick to access both the squares state and the setSquares function because they are defined within the same scope.

At this point, clicking the upper-left square will display an 'X'. However, the function is hardcoded to update only the first square. Let's generalize it so that any square can be updated.

Updating Any Square

Modify handleClick to accept an argument i, representing the square's index:

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

Now, pass the correct index when rendering each Square. You might try this:

```
<Square onClick={handleClick(0)} />
```

However, this will not work. The function will execute immediately during rendering, causing an infinite loop.

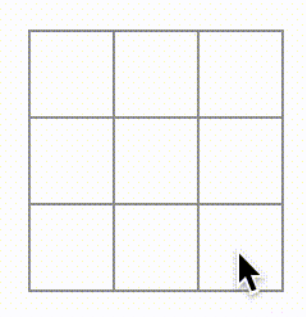
To prevent this, wrap handleClick(i) in an arrow function so it runs only when clicked:

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

Now, update all Square components to pass the correct index:

```
export default function Board() {
  // ...
  return (
    <>
      <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>
    </>
  );
};
```

Now you can again add X’s to any square on the board by clicking on them:



Now that the Board component is in charge of state, it tells each Square what to display. When you click a Square, it doesn't change itself—instead, it asks the Board to update the game state. React handles the rest, re-rendering the Board and all Squares automatically. Keeping the game state inside the Board also makes it easier to check for a winner later.

What Happens When You Click a Square?

- Clicking a Square calls the onClick function it got from the Board.
- That triggers handleClick in the Board, passing the square's index.
- The function updates the squares array, replacing null with 'X'.
- The Board's state changes, causing React to re-render the Board and all Squares.
- The clicked Square now shows 'X' instead of being empty.

Taking Turns

Right now, only 'X' appears. To alternate turns, In Board function add another state variable:

```
export default function Board() {
  const [xIsNext, setXIsNext] = useState(true);
  const [squares, setSquares] = useState(Array(9).fill(null));
}
```

Each turn, xIsNext flips between true and false, deciding whether the next move should be 'X' or 'O'. Update handleClick like this:

```
function handleClick(i) {
  const nextSquares = squares.slice();
if (xIsNext) {
    nextSquares[i] = 'X';
  } else {
    nextSquares[i] = 'O';
  }
  setSquares(nextSquares);
  setXIsNext(!xIsNext);
}
```

Now, clicks alternate between ‘X’ and ‘O’, making the game playable.

Fixing the Overwrite Issue

There is a bug: clicking the same square twice replaces ‘X’ with ‘O’. To prevent that, stop the function from updating a square if it is already filled:

```
function handleClick(i) {
  if (squares[i]) return
  const nextSquares = squares.slice();
}
```

With this fix, once a square is marked, it stays that way just like in a real game.

Declaring a Winner

Now that players can take turns, we need to implement logic to check for a winner and prevent further moves once the game is won. This can be achieved by adding a helper function called calculateWinner, which checks the state of all 9 squares, determines if there's a winner, and returns either ‘X’, ‘O’, or null (if no winner).

Here’s the function that checks for the winner:

```
function calculateWinner(squares) {
  const lines = [
    [0, 1, 2], // Horizontal rows
    [3, 4, 5],
    [6, 7, 8],
    [0, 3, 6], // Vertical columns
    [1, 4, 7],
    [2, 5, 8],
    [0, 4, 8], // Diagonal lines
    [2, 4, 6],
  ];

  // Check each line for a winner
  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 the winning symbol ('X' or 'O')
    }
  }

  return null; // No winner
}
```

Key Details:

- **Lines Array**: The lines array stores the index positions for all possible winning combinations (horizontal, vertical, and diagonal).
- **Logic**: The function checks if all three squares in any of the winning lines are filled with the same symbol ('X' or 'O'). If a winner is found, it returns that symbol; otherwise, it returns null.

Next, call calculateWinner(squares) within the handleClick function in the Board component to check if a player has won. This check should happen alongside verifying if the clicked square is already filled. If either condition is true, the function should return early:

```
function handleClick(i) {
  if (calculateWinner(squares) || squares[i]) {
  return;
 }
 const nextSquares = squares.slice();
 //...
}
```

To inform players when the game has ended, a status message should be displayed. If there is a winner, the message should indicate who won. If the game is still in progress, it should show whose turn it is:

```
export default function Board() {
 // ...
 const winner = calculateWinner(squares);
let status;
 if (winner) {
  status = 'Winner: ' + winner;
 } else {
  status = 'Next player: ' + (xIsNext ? 'X' : 'O');
 }

 return (
  <>
   <div className="status">{status}</div>
   <div className="board-row">
    // ...
  </>
 );
}
```

Final Touches

Once you integrate these updates into your Board component, you will have a fully functional Tic-Tac-Toe game that displays the winner when the game ends and tracks whose turn it is. You have also just learned how to implement basic React functionality such as state updates, event handling, and rendering dynamic content.

Submission:

1- Fill out the names and UCIDs of all group members in Answer_sheet.
2- Create a new GitHub repository and upload the code for Exercise 1 to both the new repository and D2L. **NOTE: Do not include the node_modules folder.**