27 March 2020

Tic Tac Toe

*Users: is a student - at a college - majoring in arts and craft, specialising in acting. They like simple, formal things, and they are interactive.

*User story: Create a simple, responsive Tic Tac Toe game with time abilities.

∴ Yeah, a redo functionality, up to 9, complete moves.

*Acceptance criteria: 1. Responsive interface
2. Styled accordingly - to what, ?
3. Game system application
4. Two players, X and O
5. 9 possible moves in 9 grid layout
6. Winner, loser, draw result.
7. Restart if winner, give options to quit if draw, restart if loser as well.

∴ 1 = One hr, 2 = One hr, 3 = Five hr, ~~4+5+6~~ +7 = One hr

∴ = + 8 hr coding effort + 2 hr ~~2 day sprint~~

* Sub tasks

- Create a Game class:component that will be the parent app, for ~~state~~ information management and, well redux or information magement.

  So, this~~~~ class will have
  - constructor, with in it,
    → a state object
  - handleButtonClick function for processing the button plays
  - setHistory function for adding plays to an array.
  - getHistory function for geting the history of plays so far.
  - and of cause a render function for rendering components of the Game componentent.

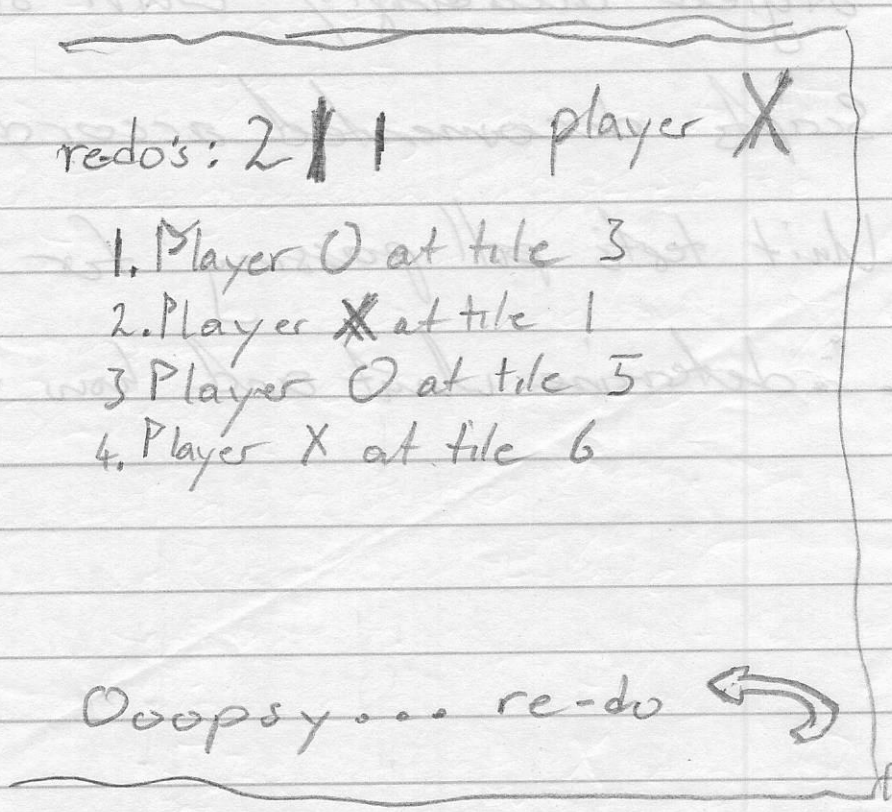
- Create a Board function:component that will be the playing board. A table, bootstraped, of a 3×3 grid system for action buttons.

- Create a Button function:component that will ~~repte~~ represent access to action events. Something happens when pressed. The Button component will be a sub-Board component.

(3)

- Create a Dashboard function component. There's going to be a bit of ~~for~~ information-flow, so a class would makes sense, but it's not complex to ~~~~ require wrapping in functions, so lightweight function.

∴



∴ The building blocks of the Dashboard can simply be ~~be~~ jsx elements. with dynamic content.

∴ Can we maybe use formik for creating this Dashboard layout?

* Definition of 'Done'

- Is component ~~constructed~~ layout build following html best practise designs.
  - i.e. ~~following~~ using descriptive element propoly, structurely.
- styled accordingly with scss/css.
- Scripts documented accordingly
- Unit tests all passing for functionality.
  - determine what and how to test.

(1)

2 April 2020

Scripts 1

* Game.js

```
import React, {Component} from 'react';
import './main.css';
import {Board} from './Board.js';
import {Dashboard} from './dashboard.js';


export class Game extends Component {

  constructor() {
    super();

    this.state = {isXTurn: false, history: [],
                  movesStack: []

                };
  }

  getHistory = (a) => {  ——————> unit test
    switch (a ? true: false) {
      case true: {
        return this.state.history[a]; }
      default: {
        return this.state.history;
      }
```

(2)

```
setHistory = e => {                    → definitely unit test
  switch (e.target.id) {
    case 'rewind': {                   → test rewind functionality
      let lastMove = this.state.movesStack.pop();
      let stackCopy = this.state.movesStack;
      let lastMove = stackCopy.pop();
      // update state with new records
      this.setState({
        history[lastMove]: "",
        movesStack: stackCopy,
        isXTurn: !isXTurn
      });
```

definitely something up there

still the same user playing →

flag {change button state} → `this.disableEnable(lastMove);`

```
      break;
    }
    case default: {                    → test game play
      let move = e.target.id;
      let historyCopy = this.state.history.slice();
      // update state with new data
      this.setState({
        history: Object.assign(historyCopy, historyCopy[move] =
        this.state.isXTurn ? 'X' : 'O'),
        movesStack:
      let stackCopy = this.state.movesStack;
      this.setState({
        history: Object.assign(historyCopy[move] =
        this.state.isXTurn ? 'X' : 'O',
        movesStack: stackCopy.push(move),
        isXTurn: !this.state.isXTurn
      });
      this.disableEnable(move);
```

flag

(2.1)

```
~~setHistory = move => {~~
setHistory = moves, stack => {
  this.setState(() => {
    return {history: moves, movesStack, stack,
      ~~isXTurn~~ : !this.state.isXTurn}});
  };
~~handleClick~~ handleClick = e => {
  ~~te~~
  let historyCopy = this.state.history.slice();
  let stackCopy = this.state.moves.stack.slice();

  switch(e.target.id){
    case 'rewind': {
      let move = stackCopy.pop();
      historyCopy[move] = null;
      // update history
      this.setHistory(historyCopy, stackCopy);
      // double break;
      }
    default: {
      ~~let moveBy =~~
      historyCopy[e.target.id] =
        this.state.isXTurn ? 'X' : 'O';
      stackCopy.push(e.target.id);
      // update history
      this.setHistory(historyCopy, stackCopy);
      break;
      }

  }
  // turn buttons on/off(
```

(3)

```
        break;
      }
    } // close switch
    this.disableEnable(move);
  }
```

→ needs to be within default at the bottom of snippet code before break.

```
disableEnable = move => {
  let button = document.getElementById(move);
  ~~let clickableState = button.disable;~~
  ~~if (button.disable) {~~
    ~~button.disabl~~
  ~~}~~
  button.disable = !button.disable;
}
```

Unit test button state after action

← don't need.

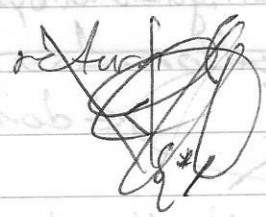
```
determineWinner = history => {  → unit test
  let rows = [
      [history[0], history[1], history[2]],
      // ... group all rows into array,
      [history[2], history[4], history[8]]
    ];
  for (let i=0; i<rows.length; i++) {
    const [a, b, c] = rows[i]
    if (a && b === c && a === c) {
      return a;
    }
  }

  return null;
}
```

(4)

```
render(){
  if(this.determineWinner()){   // for the time being: this.state.history
    alert("Winner: ${this.state.isXTurn ?
    alert(`Winner: ${this.determineWinner()}`);
    return;
  }else{
```

flow



```
  return (
    <section className="game-row">
      <article className="col-8">
        <Board
          getHistory={this.getHistory}
          setHistory={this.setHistory}
        />
      </article>
      <article className="col">
        setHistory=
        <Dashboard
          setHistory={this.setHistory}
          moves={this.state.movesStack}
          getHistory={this.getHistory}
        />
      </article>
    </section>
  );
}}
```

Give Dashboard property `isXTurn={this.state.isXTurn}`

①

2 April 2020

Script 2

*Dashboard.js

```
~~const~~ expect const Dashboard = props => { -> unit test

  return (
    <form title="Progress Dashboard">
      <span className="sub-head">
        <button
          ~~onClick={props.s~~ id="rewind"
          onClick={event => props.setHistory(event)}
        >
          <i className="fa fa-arrow-left"></i>
        </button>
        <label name="turns">
          {(props.isXTurn)? 'X' : 'O'}
        </label>
      </span>
      <ul>
        {
          props.moves.map(move => {
            return (
              ~~<li>~~ <li key={move}>
                Player {props.getHistory(move)} at...
                tile {move + 1}
              </li>
            );
          });
```

(margin note, with arrow to the label expression:) conput this component through a shallow mount and give it props

(margin note, beside the props.moves.map block:) Can test this block by giving this component an array/list expecting a list size same as array

```
        }
      </ul>
    </form>
  );
}
```

2 April 2020

Script 3

*Board.js

```
import React from 'react';
//import buttons component
import { Buttons } from './buttons.js';

export const ~~Button~~ Board = props => { - unit test

  return (
    <table className="table grid">
      ~~<thead>~~<tbody>
        <tr>
          <td>
            <Buttons
              boardProps={props} index={0}
            />
          </td>
          <td>
            <Buttons
              boardProps={props} index={1}
            />
          </td>
          <td>
            <Buttons
              boardProps={props} index={2}
            />
          </td>
        </tr>
```

* Theres static content rendered on this component - few things to test - but will look at testing if all grid items are mounted.

* Buttons.js

~~test~~

~~const~~ export const Buttons = props => { -> unit test

```
return (
    <button
        id={props.index}
        onClick={event => props.setHistory(event)}
        title={`button ${props.index + 1}`}
    >{props.
        {getHistory(props.index) ? 'block' : ''}}
    </button>
);
}
```

block

... maybe through a shallow load of this ~~component~~ component - give it made up props

(1)

2 April 2020

Tests - unit - analysis

* Script 1 -> Game.js

⊙ As the system, I can access 'plays over time' by calling a function.

∴ - Test getHistory function
1. With an argument, i.e 0, 1, 2 etc
3. Null value argument

∴ - To verify test pass, check returned values.

⊙ As a user, when I click on a play button, the system is updated with my action

⊙ As a user, when I click on a rewind button, the system deletes last play record.

∴ - Test the setHistory function
1. With a simulated button with id rewind
2. And simulated button with id as an integer index.

∴ - Verify test pass by checking updates in state of component
∴ or state change of fired

- As a user, when I click on a button to play, it is disabled after turn.

  ∴ - Test the disableEnable function
  Whats function is to turn buttons on/off.

  ∴ - Verify by checking state of button

- As a user, when I have joined 3 adjacent blocks with my symbol, I win game.

  ∴ - Test the functionality of determining a winner, by testing determineWinner() function
  ∴ - give it a made up history
  - Verify tests by checking returned values by function.

(3)

* Script 2 -> Dashboard.js

- As a user, when it is my turn to play, my user symbol is shown.
  - Test whether interface renders label with 'X' or 'O'.
  - Verify if targeted label has X or O as innerHTML.
- As a system, when n moves have been played, they are rendered on interface
  - Test if the progress is being mapped on the <ol> element
  - Verify size of <ol> is greater than 1 after play.

* Script 3 -> Board.js

- As the system, when I am loaded, I should have a 3x3 grid (or 9 buttons in total for playing.

  ∴ - Test if this component has a total of 9 rendered buttons

  ∴ - Verify by getting the list of button elements and checking length.

* Script-sub -> Buttons.js

- As a user, when I click on a button to play, my symbol is saved in history. ~~and appears on button~~

- As a user, when I click on button, ~~my~~ symbol appears on button after.

  ∴ - Test with artificial content on a shallow mount. Give component a simple setHistory function and getHistory for data.

  ∴ - Verify data changes in arbitrary sets and button state changes.

3 April 2020

Test Algorithms

* Script 1 → Game.js

This script mostly has component that are functionality-focused, instead of presentation or layout.

1. Connect to the Game.js file ~~with input~~
2. Initialise an arbitrary history array with content
3. ~~If true~~

1. test for getHistory
   1.1 declare a step variable
      1.1.1 initialise step with getHistory()
   1.2 check if step is [] for pass

2. test for getHistory given argument
   1.1 declare index variable
      1.1.1 initialise index with value 0-8
   1.2 ~~check~~ if declare step variable
      1.2.1 initialise index with getHistory(index)
   1.3 check if step is null for pass

// the above algorithm assumes history empty ([])

3. test setHistory with argument
   1.1 declare moves array
      1.1.1 populate moves with dummy data
   1.2 declare history array
      1.2.1 populate history at moves ~~with~~
   1.3 connect to the Game class
   1.4 call setHistory with argument with connection.object
   1.5 check the length of state.history is the same local history

4. test handClick 'rewind'
   1.1 declare an empty object
      1.1.1 initialise with property target.id = "rewind"
   1.2 connect to Game class
   1.3 call handleClick with object as argument
   1.4 check if state.history is empty

5. test handleClick with 'playing button'
   1.1 declare an empty object
      1.1.1 initialise it with property target.id = "4"
   1.2 connect to Game class
   1.3 call handleClick with object as argument
   1.4 check for added value in state.history at index '4'

* We'd have to globalise our connection to Games as

(3)

6. test if button turns on/off → disableEnable function
   1.1 initialise move with button id (0-8)
   1.2 call disableEnable with move as argument
   1.3 get button of id 'move' object
   1.4 check if property disable is true or false

7. test determineWinner function
   1.1 declare empty history array
   1.2 populate history with arbitrary moves
   1.3 call disableEnable from Game with history array and handle returned value
   1.3 check if return is null (for no winner - or expected "winner")

(4)

* Script 2 → Dashboard.js

1. test if interface renders 'X' or 'O'
   1.1 initialise empty object with {isXTurn: true}
   1.2 mount the component on a shallow state and give object as property
   1.3 get element ~~with~~ with ~~name~~ name = turns
   1.3 check if element has content 'X'

Note

- testing <button> element with setHistory function:
  - Not certain if I need to test this part of the Dashboard component.
    Might consider ~~testing~~ this functionality ~~during~~ during integration.

∴ Same as testing the mapped list elements when application is running.

(5)

* Script 3 → Board.js

There's really no functionality for testing on this component, ~~so~~ so we could simply include a snapshot test as it forms a very important UI of the application. - 3×3 grid. I could maybe check 3×3 <tr>, <td> structure but ~~what's~~ its hard-coded elements.

* Sub-script → Button.js

~~Set~~ This component has functionality associated with another component, so can't test that on unit testing. Plus the component will be in a shallow state.