Vanilla HTML / CSS / JS Tic Tac Toe Game

Concepts covered.

- MVC architecture pattern
- What is “State”
- Event delegation
- Mobile responsiveness
- CSS animations
- CSS grid
- Etc…

Prerequisites

- Beginner to intermediate-level tutorial
- Must know HTML, CSS, JS (basics)

Visual Studio Code Extensions

- Live Server
- Prettier (Optional for code formatting)

In Visual Studio code, enter ctrl (command) + p and open > User Settings (JSON) and enter the following code. Every time you save, you automatically format your file with prettier {

```
    "editor.formatOnSave": true,
    "editor.defaultFormatter": "esbenp.prettier-vscode"
}
```

Part 1 Vanilla: HTML, CSS

Building out the HTML

HTML 5 boilerplate code

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Vanilla JS T3</title>
```

```
  </head>
  <body>
    <p>Placeholder</p>
  </body>
</html>
```

Building out the css

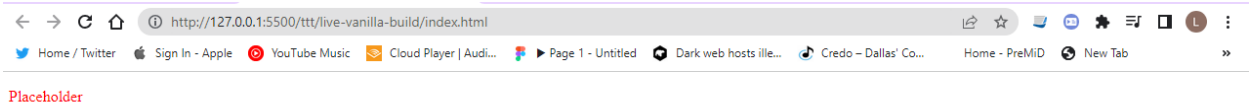
- Create a folder name css; in that folder, create a file named index.css. Type in the following code into the index.css file

```
* {
  color: red;
}
```

- Enter the following link html tag above the title tag

```
<link rel=" stylesheet" href="css/index.css" />
```

- Confirm that the Placeholder text is red: replace the placeholder paragraph tag with a main tag.




```
<head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <link rel=" stylesheet" href="css/index.css" />
    <title>Vanilla JS T3</title>
  </head>
  <body>
    <main></main>
  </body>
```

Building the Grid

- Inside of the main tag, create a div class named grid and nest N amount of div tags in the grid class div: Pictured below are 15 nested divs, creating a 3x5 grid

```
        <div class ="grid">

            <div></div> <div></div> <div></div>
            <div></div> <div></div> <div></div>
            <div></div> <div></div> <div></div>
            <div></div> <div></div> <div></div>
            <div></div> <div></div> <div></div>
```

```
    </div>
```

- In index.css, insert the following template code to initialize the fonts and colors. We are importing a Google font Montserrat and initializing css variables to the root selector

```
@import
url("https://fonts.googleapis.com/css2?family=Montserrat:wght@400;500;600&display
=swap");

:root {
  --dark-gray: #1a2a32;
  --gray: #2e4756;
  --turquoise: #3cc4bf;
  --yellow: #f2b147;
  --light-gray: #d3d3d3;
}
```

- In index.css, insert the following template code to reset any possible padding to the default.

```
*{
    padding: 0;
    margin: 0;
    box-sizing: border-box;
    list-style-type: none;
    font-family: "Montserrat", sans-serif;
    border: none;
}
```

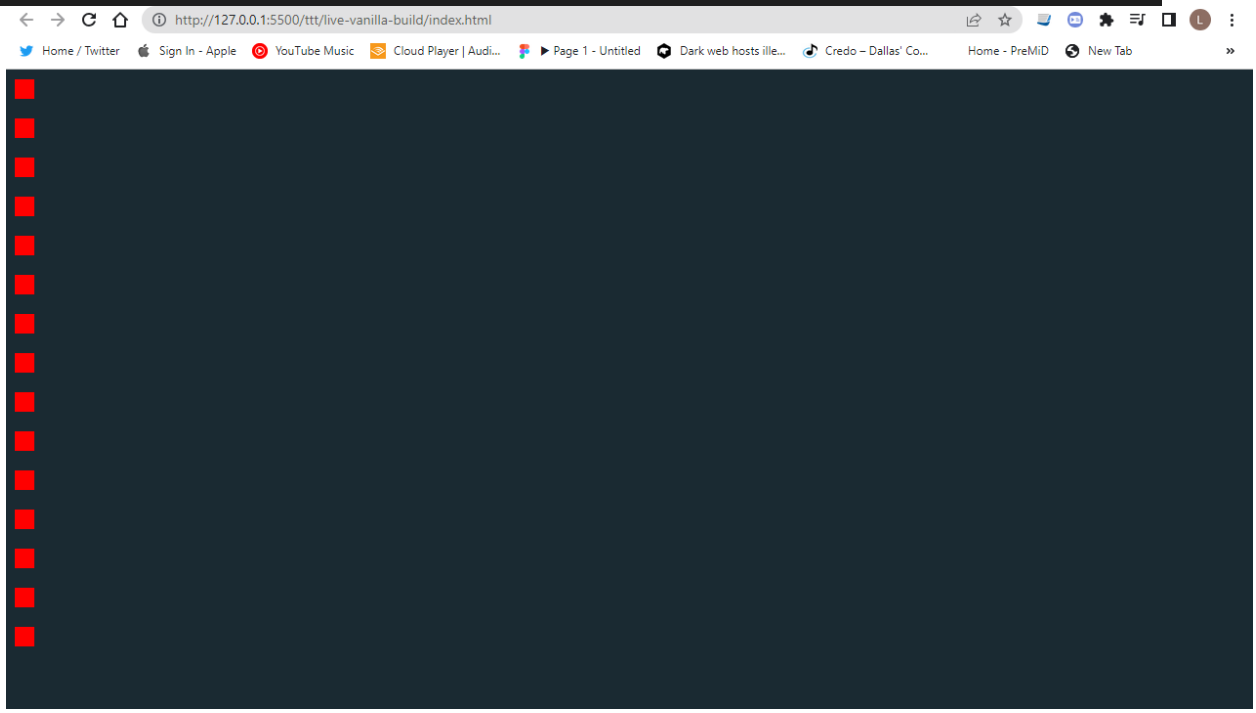
- Set our background color with the following css code.

```
html,
body {
    height: 100%;
     background-color: var(--dark-gray);

}
```

- Set the grid items' height, width, color, and margin spacing, and check the live server for the following.

```
.grid div {
  height: 20px;
  width: 20px;
```

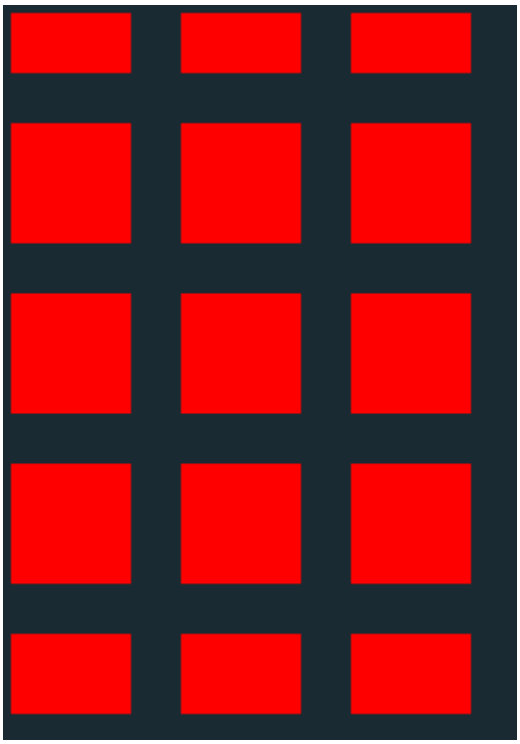
```
  background-color: red;
  margin: 10px;
}
```



- Set the grid items as rows and columns. The grid class should be as follows.

```
.grid {
  display: grid;
  grid-template-columns: repeat(3, 80px);
  grid-template-rows: 50px repeat(3, 80px) 60px;
  gap: 5px;
}

.grid div {
  background-color: red;
  margin: 10px;
}
```

- This is where CSS gets weird in my opinion: grid-template-columns dictates the width of each div as 80px, : grid-template-grid dictates the height of the first row as 50px tall, the next three rows are 80px tall, and the final row is 60px high.



- Back in the index.html, we will now distinguish the rows (TIP use ctrl+alt/option+cmd to select multiple divs at once)

```
<div class="grid">
        <div class="turn"></div>
        <div class="turn"></div>
        <div class="turn"></div>


        <div class="square"></div>
        <div class="square"></div>
        <div class="square"></div>
        <div class="square"></div>
        <div class="square"></div>
        <div class="square"></div>
        <div class="square"></div>
        <div class="square"></div>
        <div class="square"></div>

        <div class = "score"></div>
        <div class = "score"></div>
        <div class = "score"></div>
```

- Back in index.css, replace the grid div class with distinct turn square and score classes

```
.turn {
  background-color: green;

}

.square {
  background-color: red;
}

.score {
  background-color: blue;
}
```

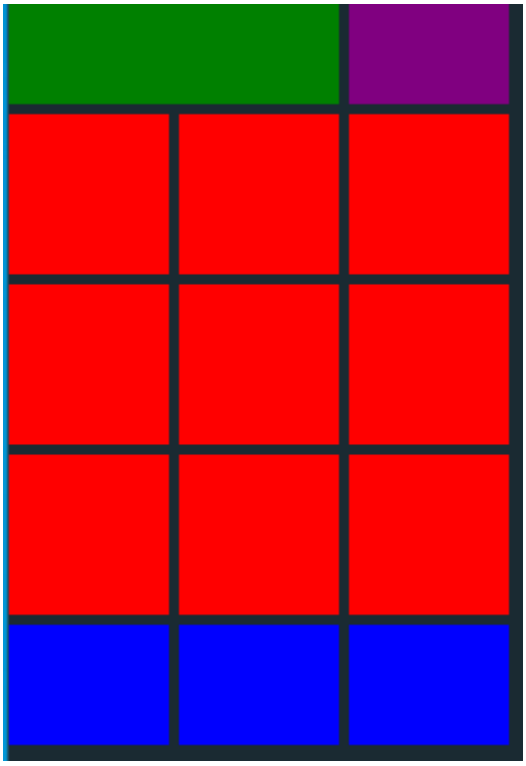
- Allow the top left grid item to span two grid spaces at once: First, edit index.html as follows: remove a turn div and create a new action div.

```
main>
      <div class="grid">
        <div class="turn"></div>
        <div class="actions"></div>

        <div class="square"></div>
        <div class="square"></div>
            …
```

- Next, edit the index.css file, add the new actions style, and add how many rows and columns the turn div spans. Save and check the live server.

```
.turn {
  background-color: green;
  grid-column-start: 1 ;
  grid-column-end: 3;

}
```

```
.actions {
    background-color: purple;
}
```



- This concludes the layout portion of this project. Next, we will return to the index.html file and add to the following style string to align the grid no matter how the wind is resized.

```
<body style="display: flex; justify-content: center; align-items: center">
```

- **In index.css, add a new body class below the html body class as follows.**

```
body {
  padding: 90px 20px;
  display: flex;
  flex-direction: column;
  justify-content: center;
  align-items: center;
}
```

- Next, we will create our footer div; this will be located in index.html below the main tag, allowing us to display the footer on every page.

```
<footer>
      <p>
        Original project by
        <a href="https://twitter.com/megfdev">@megfdev</a> and
        <a href="https://twitter.com/Ivan00sto">@Ivan00Sto</a>
      </p>

      <p>
        Refactored by
```

```
        <a href="https://twitter.com/zg_dev" style="color: var(--turquoise)"
          >@zg_dev</a and <a href="Enter either a Twitter link or another form of
contact  "> Your name/online handle </a>


        >
      </p>
    </footer>
```

- Now, we will style the footer in index.css with the following code.

```
footer {
  color: white;
  margin-top: 50px;
}

footer p {
  margin-top: 10px;
  text-align: center;
}

footer a {
  color: var(--yellow);
}
```

- Add in index.css a media center; this allows us to resize the screen differently depending on the type of device we are viewing this page on. Any screen smaller than the minimum screen width will be raised with the following.

```
/*Desktop styles */
@media (min-width: 768px) {
  .grid {
    width: 490px;
    grid-template-columns: repeat(3, 150px);
    grid-template-rows: 50px repeat(3, 150px) 60px;
    gap: 20px;
  } }
```

- Now, the majority of the html and css is now rendering functionally and can resize to display on almost any device. Next will be tackling the look and feel of each of the grid divs’; up first is the turn dive; the turn div will display which play is allowed to place ither an x or an o on the square sub-grid.
- In the index.html file, we insert some font awesome icons and a turn indicator. The turn div is as follows, also, don’t forget to import the font awesome kit to the file. To do this, add the below script div above the title dive.

```
<div class="turn" data-id="turn">
          <i class="fa-solid fa-x turquoise"></i>
          <p class="turquoise">Player 1, you're up!</p>
```

```
        </div>
```

```
<meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <link rel=" stylesheet" href="css/index.css" />
<script
      src="https://kit.fontawesome.com/3bfd3ce9e5.js"
      crossorigin="anonymous"
    ></script>

    <title>Vanilla JS T3</title>
```

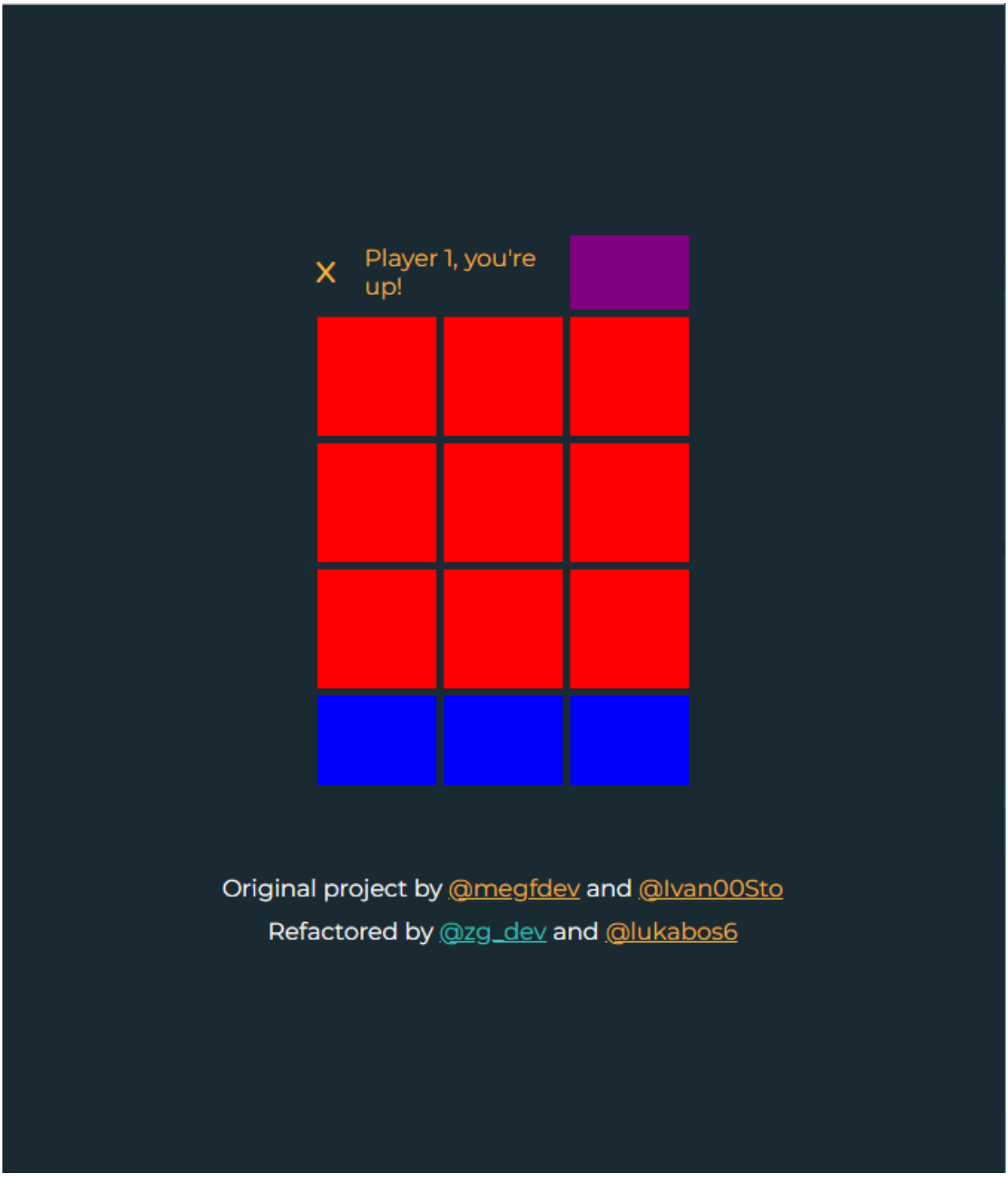
- Next, we will edit the turn style class in index.css. It will look like the following

```
.turn {
   /*debug background-color:green; */


  grid-column-start: 1;
  grid-column-end: 3;

  align-self: center;
  display: flex;
  align-items: center;

  gap: 20px;
}
```

- The last step for our turn style is to animate it! Working in index.css, we're starting with our icon selector animation, the following should be placed underneath the above code.

```
.turn p {
  font-size: 14px;
}

.turn i {
  font-size: 1.8rem;
  margin-left: 10px;

}
```

- The page should appear as the following



- We will now implement the animation itself: first, we specify the animation's name as a css property, the duration of the animation, the behavior of the animation, and the reference to the icon we are animating. The following should be added to the turn icon selector p(this animation makes the icon larger than smaller). The reference is turn-text-animation, an arbitrary value representing keyframes; we will implement the keyframes in the next step.

```
animation: 0.6s ease-in-out turn-icon-animation;
```

- The next add the turn-icon-animation as follows. What the below code does is at 0% of the time duration, it will scale to 100%, then 25% into the animation, it will scale to 140%, then scale back down to 100%.

```
@keyframes turn-icon-animation {
  0% {
    transform: scale(1);
  }
  25% {
    transform: scale(1.4);
  }
  100% {
    transform: scale(1);
  }
}
```

- Now, we will add the following text animation to the paragraph (p) turn selector.

```
animation: 0.6s ease-in-out turn-text-animation;
```

- Just like the previous reference, we will now add turn-text animation. The Below code transforms the text by -20px at 0% animation duration; then, it returns to its original position at the end of the duration.

```
@keyframes turn-text-animation {
  0% {
    opacity: 0;
    transform: translateX(-20px);
  }
  100% {
    opacity: 100%;
    transform: translateX(0);
  }
}
```

- Moving on to the next grid item, we will work on the actions (renamed to menu) div. First, we will return to the index.html file and add the following markup to the menu div. First is the button that is rendered first. Below the button div is the popover div rendered after the button is clicked, but otherwise remains hidden.

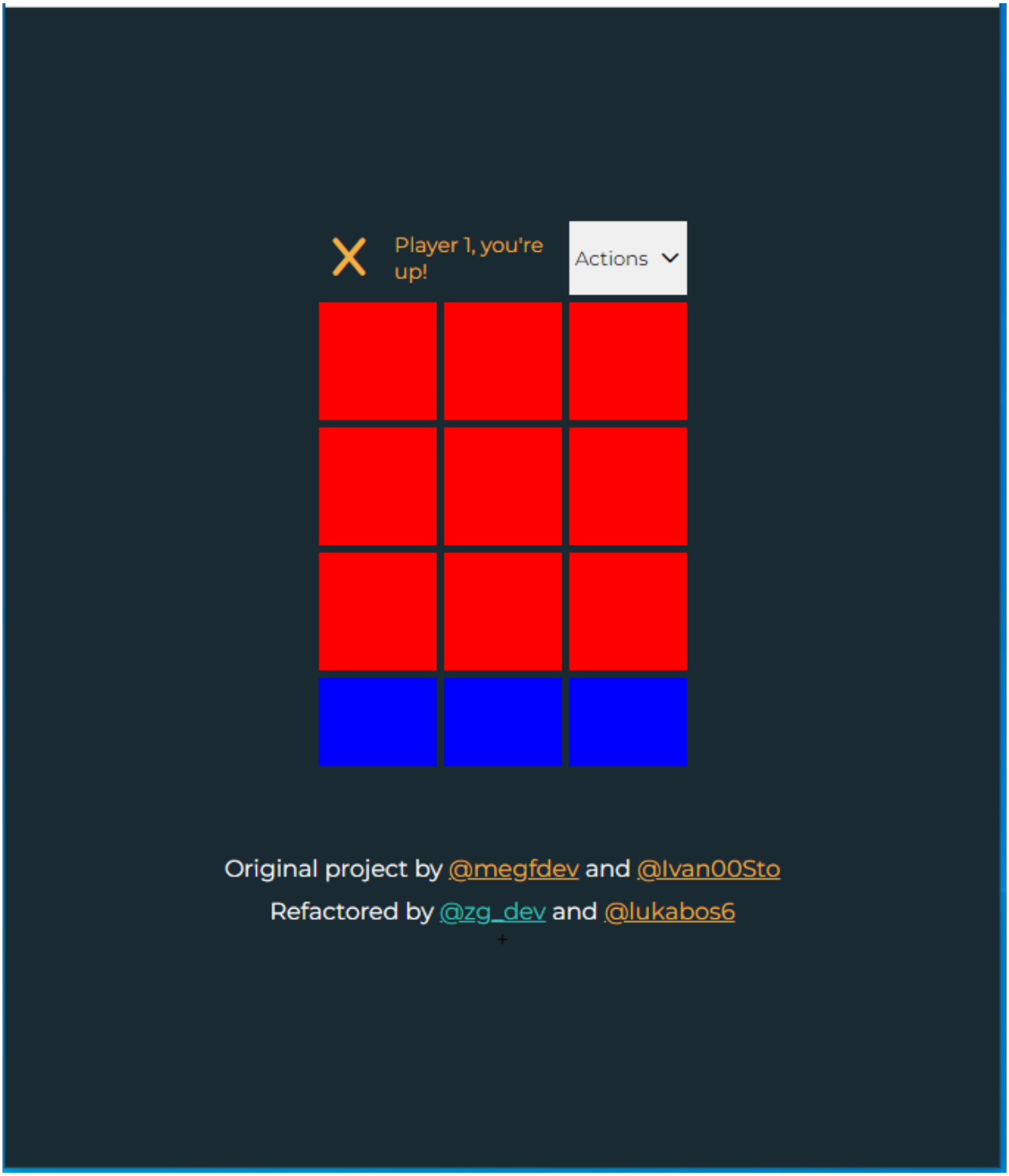
```
<div class="menu" data-id="menu">
          <button class="menu-btn" data-id="menu-btn">
            Actions
            <i class="fa-solid fa-chevron-down"></i>
          </button>

          <div class="items border hidden" data-id="menu-items">
            <button data-id="reset-btn">Reset</button>
            <button data-id="new-round-btn">New Round</button>
          </div>
        </div>
```

- Next, we will center the menu div and edit its style as follows. The position is set to relative so that the popover has the relative location in the Dom due to its absolute positioning. In the index.css file enter the following menu style class below the turn i (icon) selector.

```
.menu{

    position: relative;
}
```

- Now we will add the menu button style: The width and heigh, style the button to 100% of the pixel value of the container (150 by 50). next, we display flex, so the items within the button are flex items. We give space around and justify it to the center. It should look the following (note that I have commented out the hidden items in the drop-down menu)

```

.menu-btn {
  width: 100%;
  height: 100%;
  display: flex;
  justify-content: space-around;
  align-items: center;

}
```

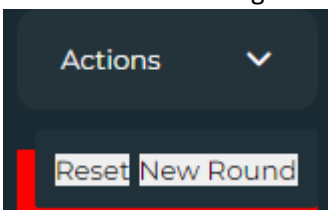


- Next, add the following to the menu style class; it should look like the following.

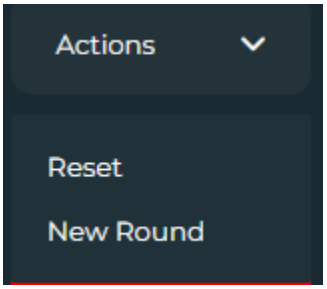
```
.menu-btn {
  width: 100%;
  height: 100%;
  display: flex;
  justify-content: space-around;
```

```
  align-items: center;
  border-radius: 10px;
  color: white;
  background-color: rgba(211, 211, 211, 0.05);
  border: 1px solid transparent;
}
```

- We define a single-use color as a background color; we set the text to white, then give it a boarder of 10px and round it off with 1px of solid transparent layer. The actions menu now appears as follows:

- Now, in the index.html file, we will uncomment the hidden drop-down menu div options and work on styling the rest of the buttons.
- Back in the index.css file, enter the following item class. In the first line, the position is absolute because the items (declared in index.html) are within the menu class. The absolute position will be relative to the nearest parent element with a relative position. For the top declaration, add a 60px wide pad relative to the absolute position of the parent element(menu-btn) or a simple 60px from the top left corner. We also set a background color, the radius of said background, and 10px of padding between the letters and the end of the border. 


```
.items {
  position: absolute;
  z-index: 10;
  top: 60px;
  right: 0;
  background-color: #203139;
  border-radius: 2px;
  padding: 10px;
}
```

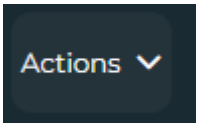
- Next for the button style is to specify that items within the menu and then any button that is within the menu is styled as so. 


```
.items button {
  background-color: transparent;
  padding: 8px;
  color: white;
}
```

- To let the user know that the buttons are clickable, we now use the following hover styles. When we hover, we underline the text and change the cursor to a pointer.

```
.items button:hover {
  text-decoration: underline;
  cursor: pointer;
```

```
}
```

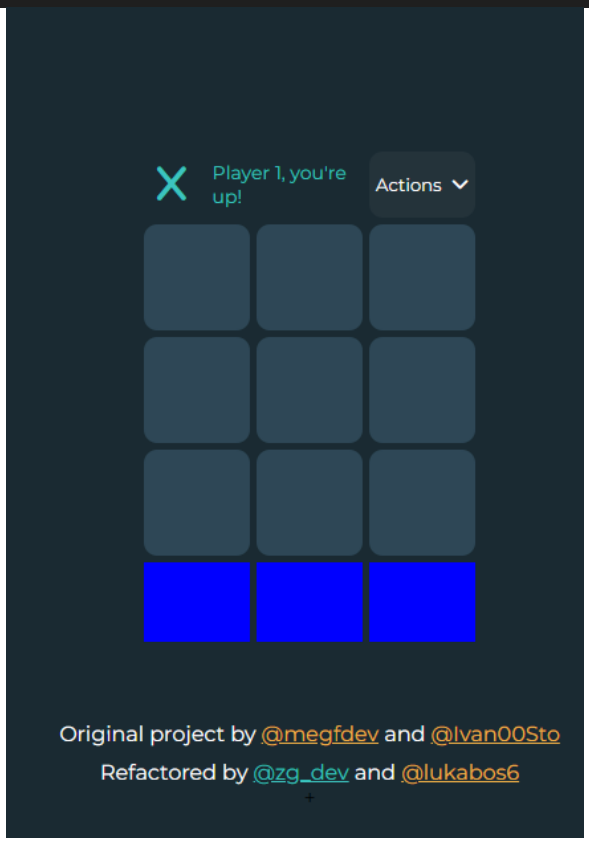
- To allow text to be hidden and to provide functionality for later, we will include the following utility classes underneath the body class. Are nested buttons are now hidden. 

- Now we move to the main game, board squares. Back in the index.html file, we first add the shadow class markup to each of the game board squares. The following ex shows a single instance

```
<div class="square shadow"></div>
```

- Next back in the index.css, replace the temporary square class with the following new class. We have a 10px boarder, make sure that the square class defines flex objects to center within. And set the font size to 3rem to increase the icon by a factor of 3. The page should look like the following

```
.square {
  background-color: var(--gray);
  border-radius: 10px;
  display: flex;
  justify-content: center;
  align-items: center;
  font-size: 3rem;
}
```



- Next, just as we did for the Actions drop-down, we will add the following a cursor hover style to the square class

```
.square:hover {
  cursor: pointer;
  opacity: 90%;
}
```

- For our final row score row, we once again return to the index.html file and add the following markup to the Scoreboard divs. For readability, we distinguish the text color in each div. Each div container contains a paragraph tag to indicate what value is displayed. The span tag displayed the value.

```
<!-- Scoreboard -->
        <div class="score shadow" style="background-color: var(--turquoise)">
          <p>Player 1</p>
          <span data-id="p1-wins">0 Wins</span>
        </div>
        <div class="score shadow" style="background-color: var(--light-gray)">
          <p>Ties</p>
          <span data-id="ties">0</span>
        </div>
        <div class="score shadow" style="background-color: var(--yellow)">
          <p>Player 2</p>
          <span data-id="p2-wins">0 Wins</span>
        </div>
      </div>
```

- Next we style the scoreboard. In the index.css file implement the following css class by replacing the placeholder blue score style .

```
.score {
    display: flex;
    flex-direction: column;
    justify-content: center;
    align-items: center;
    border-radius: 10px;
  }
```

- To style the text in the scoreboard, use the following code, we follow similar steps(sub selectors) when animating the turn text.

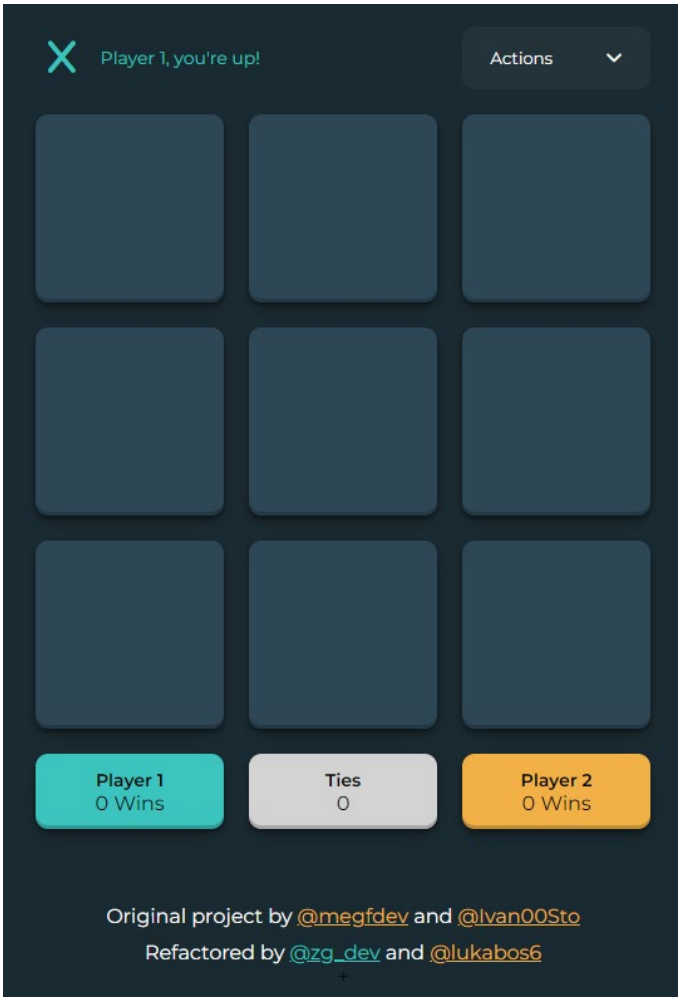
```
.score p {
  font-size: 14px;
  font-weight: 600;
}
```

- Last style is the span element. The span element displayed the amount of wins each player has.

```
.score span {
```

```
  font-size: 12px;
  margin-top: 2px;
}
```

- The html and css should appear as the following.



- The final addition that need to be made to the html and css is the modal focus element that is to be when a player wins. Open the index.html and enter the following code after the footer tag

```
<!-- Modal that opens when game ends -->
   <div class="modal hidden" data-id="modal">
     <div class="modal-contents">
       <p data-id="modal-text">Player 1 wins!</p>
       <button data-id="modal-btn">Play again</button>
     </div>
   </div>
```

- Now we return to the index.css file and style our end game modal as follows. We set the display container to flex, but the positioning is fixed. Meaning that the position is relative to the entire document. We then place the item in the center of the viewport, and set the background color of the viewport to a slightly transparent black.

```
.modal {
    display: flex;
    justify-content: center;
    align-items: center;
    position: fixed;
    width: 100%;
    height: 100%;
    background-color: rgba(0, 0, 0, 0.6);
  }
```

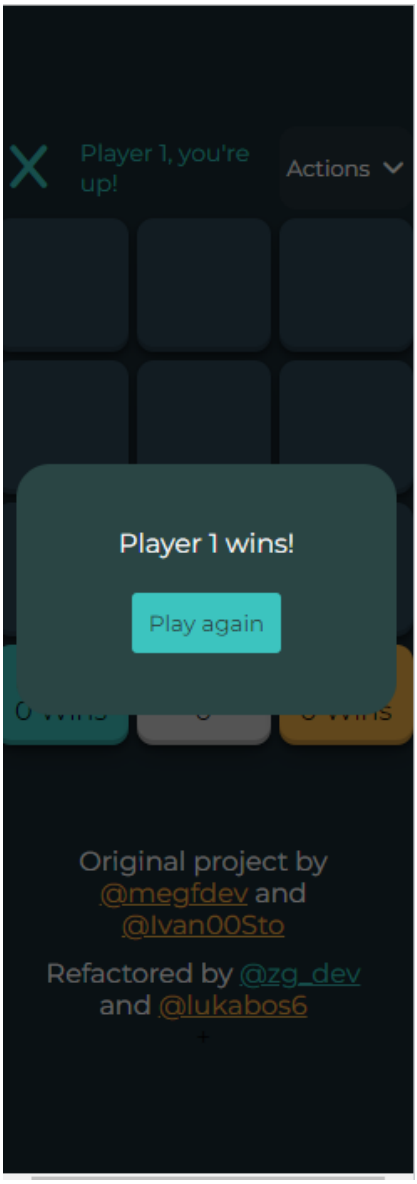
- Next, we do further styling to center the modal. By default, we want 100% width, but if the screen is big enough that 100% width is larger than 300px, default to max width. This is a trick to make mobile-first css. We align everything to the center and then give it a flex gap of 20px so the content has some spacing. The color of the text is white, and we give a 10px margin for a cleaner mobile experience

```
.modal-contents {
  /* transform: translateY(-80px); */
  height: 150px;
  width: 100%;
  max-width: 300px;
  background-color: #2a4544;
  border-radius: 20px;
  display: flex;
  flex-direction: column;
  justify-content: center;
  align-items: center;
  gap: 20px;
  color: white;
  margin: 10px;
}
```

- Lastly we style the play again button like so. Giving the button the color of turquoise and a bit of padding.

```
  .modal-contents button {
    padding: 10px;
    background-color: var(--turquoise);
    color: #2a4544;
    border-radius: 3px;
  }
```

- The final modal button when viewed on mobile



What is a “game move”?

- Who is currently up(turn indication, icon to play)
- -Did the latest move cause a tie or a game win?
- Who won? Was it a tie?

What “State” do I need to track?

-Current player

-Total wins by player

-Total ties

-Prior game history

Part 2: Adding JavaScript interactivity to the project

When writing JavaScript, where do I start?

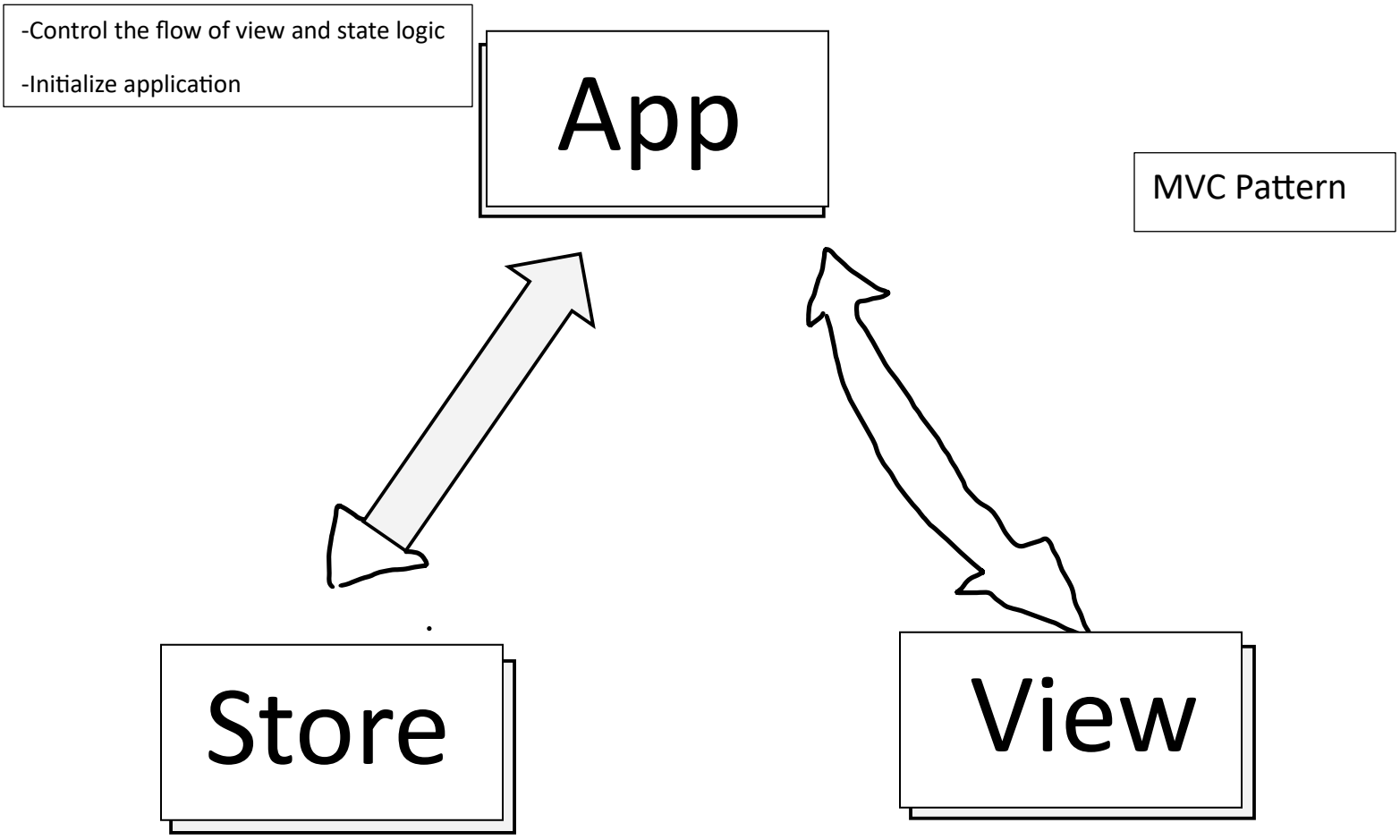
Naturally, the next question should be asked is what type of project this is.

In the context of a level one web development project, I generally categorize projects as follows: Game, Content delivery system, data science proof, or algorithm visualization. Obviously, this project is a game.

The next question that must be answered is what actions a user can take in my application?

1. Player can make a game move
2. New round
3. Rest current game
4. Toggle the menu

The systematic approach that is most conducive for front end development is the Model-View-controller. Its core design philosophy is an “separation of concerns”



What is “state”?

ntState = {
nuOpen:false

2. Server state-> const serverState = {

```
        currentPlayer:1,
        currentGame:[],
        history:[]
    }
```

This is usually a very challenging pattern to implement for beginners. Thus, we will start with a contrived example. (Instructors note: It has been challenging to me as a student instructor to remember that beginners need rudimentary explanations, and only upon the students understanding can greater theory be imparted)

Tutorial Progression

1. Naïve approach, all in one file
2. Refactor, iterate as we go
3. End up with MVC pattern that is…
    a.) Easier to debug later
    b.)Easier to extend and add features

Back to Visual Studio Code, and back to our first question

In the html.index theoretically, you can put Javascript anywhere in this document but it is a best practice to place JavaScript at the end of the body tag.

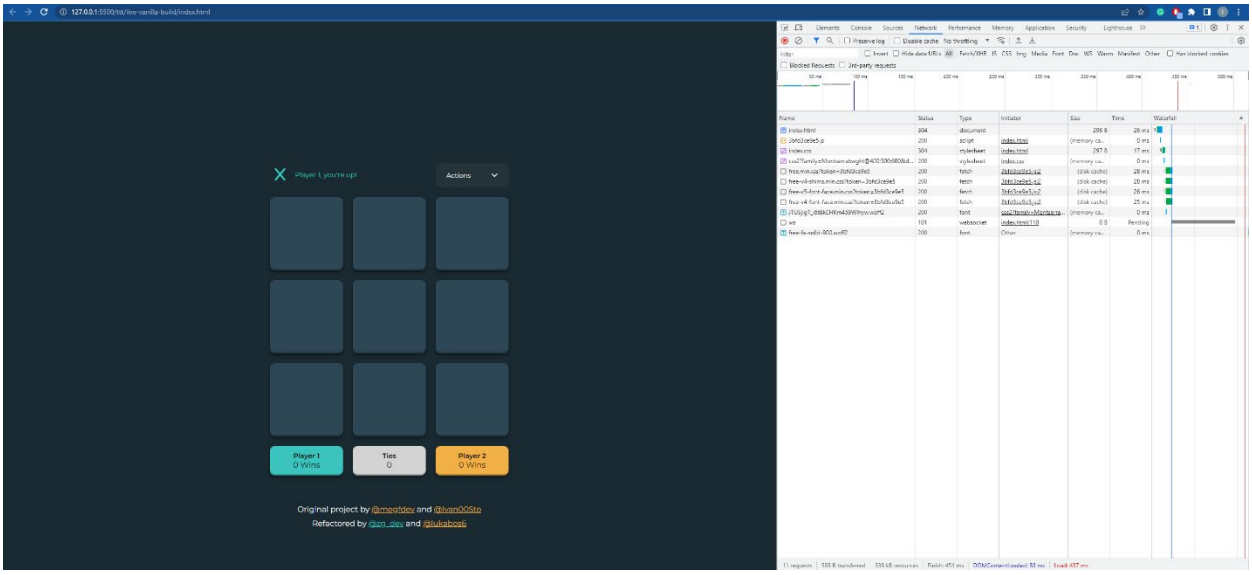
Critical Rendering Path(intermediate – Advance concept)(CRP)

The reason for that is when we are loading an HTML document in a browser, the browser has a bunch of code that is running in the background that it has to read through this HTML document, Parse all of the elements, all the styles, then each style is attached to each element, and finally it is painted to the screen

```
77      </footer>
78
79      <!-- Modal that opens when game ends -->
80      <div class="modal hidden" data-id="modal">
81        <div class="modal-contents">
82          <p data-id="modal-text">Player 1 wins!</p>
83          <button data-id="modal-btn">Play again</button>
84        </div>
85      </div>
86
87      <!-- Place JavaScript Here-->
88
89    </body>
90  </html>
91
92
```

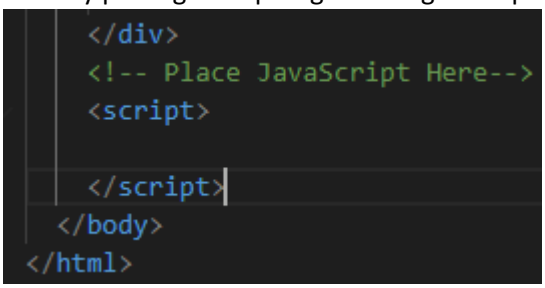
One more example for the above: To render the below, run the visual studio code live server extension, In a Chrome, Browser navigate to the Network tab after loading the document.

The network tab displays the order of the CRP, the return statis of each, as well as the performance.



Back to index.html, we will now begin incorporating JavaScript in the document. To reiterate why we need to place the JavaScript at the end of the body, from a user experience standpoint, it would be a rather crummy experience if every time CRP parses index.html, a whole heap of JavaScript had to resolve before the user sees anything. It makes much more sense to distract the user while we resolve out JavaScript.

Start by placing a script tag in our agreed upon location like so:



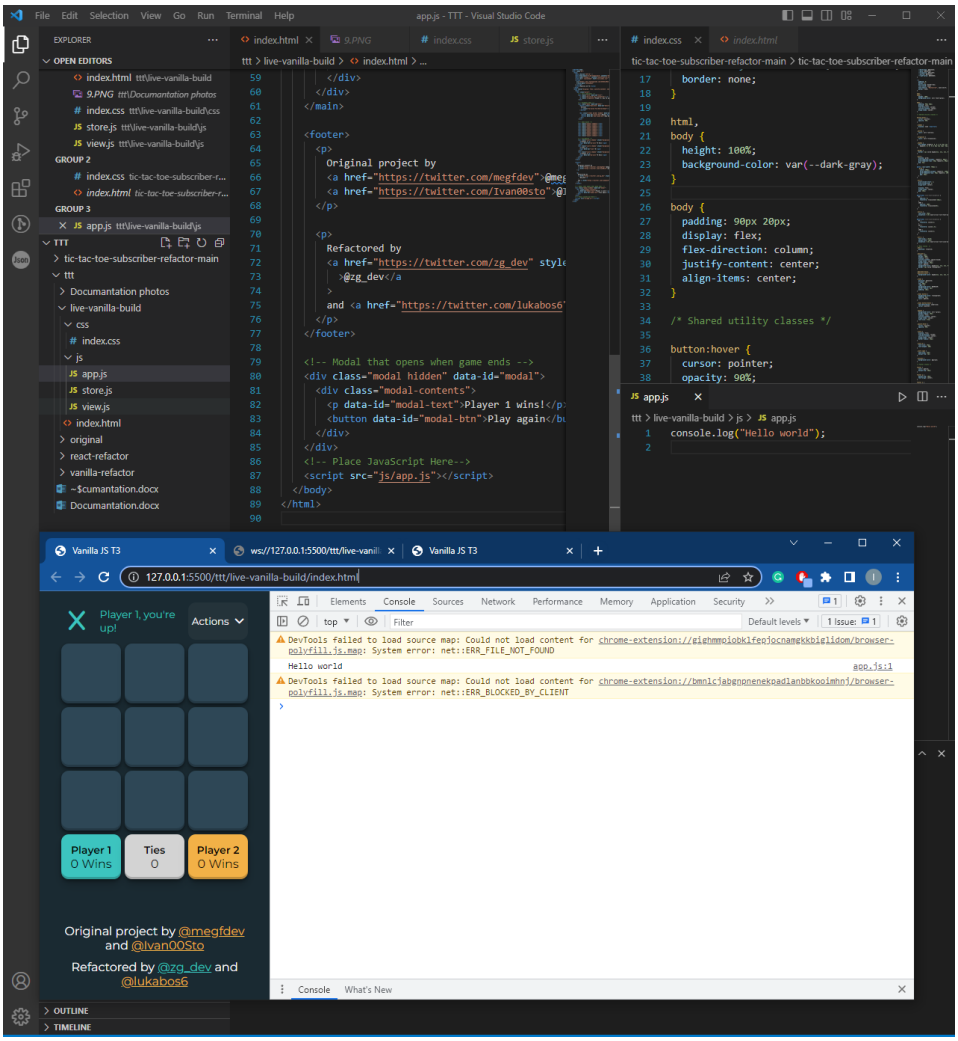
Next, add a JavaScript source tag to point at our java script file. While where at it go ahead and create a new directory named js and create a file inside called app.js. After this fill in the filepath of the file directory and file we just created("js/app.js"), finally to test if we have properly set up JavaScript go ahead and run a hello world! Program as shown below.

[Sidebar Title]

[Sidebars are great for calling out important points from your text or adding additional info for quick reference, such as a schedule.

They are typically placed on the left, right, top or bottom of the page. But you can easily drag them to any position you prefer.

When you're ready to add your content, just click here and start typing.]



In the app.js,, we will begin working backward on our features, starting with toggling the user menu. As mentioned previously, this is our rudimentary, and as we refactor hopefully the benefits become clear, let's begin!

We start by calling our globally accessible document (available to us in the browser runtime) and call the querySelector method passing in a .menu class, meaning any instance of a menu in the hypertext. Next, we add an event listener and listen for a click event; the callback, we specify that we want to receive an event object which targets our menu query variable. Now we can output when a click has occurred on all instances of menu.

```
const menu = document.querySelector(".menu")// any item that is of menu class

menu.addEventListener('click', event => {
    console.log(event.target)
});
```

The Simplest way to toggle our menu back and forth is to create a query for menuitems within the menu; with our previous logic already establishing our click logic we can now manipulate our drop down to toggle on and off every time a click is registered.

```
const menu = document.querySelector(".menu"); // any item
that is of menu class
const menuitems = menu.querySelector(".items");

menu.addEventListener("click", (event) => {
  menuitems.classList.toggle("hidden"); // Toggle the
hidden class back and forth every time the button is
clicked
});
```

1. Let's say we went ahead and added another script to our index.html and points a file called second-app.js. The contents of this file consist of a single line:

```
const menu = {};
```

```
<script src="js/second-app.js"></script>

<script src="js/app.js"></script>
 <!-- Place JavaScript Here-->
 <script src="js/app.js"></script>
 <script src="js/second-app.js"></script>
/body>
```

<- ^ This case causes a redeclaration of the menu variable, so it no longer knows what to do because we are trying to set an event listener on a constant object variable. After all, there are conflicts. Now don't get confused; programming allows you to reuse the same variable name; think of a program with multiple looping functions. You can declare a loop control (lcv)variable with the letter i, and in an adjacent method, you can redeclare an lcv with i due to the declaration not occurring within the same scope as one another. You can think of the end of the body tag where we put our js code as the Global scope. (DELETE second-app)

```
<!-- Place JavaScript Here-->
<script src="js/second-app.js"></script>
<script src="js/app.js"></script>
/body>
```

Element

(https://developer.mozilla.org/en-US/docs/Web/API/Element)

Element is the most general base class from which all element objects (i.e. objects that represent elements) in a Document inherit. It only has methods and properties common to all kinds of elements. More specific classes inherit from Element.

For example, the HTMLElement interface is the base interface for HTML elements, while the SVGElement interface is the basis for all SVG elements. Most functionality is specified further down the class hierarchy.

Best practices when developing user interfaces

1. Global scope and namespaces
2. 2 Stable selectors (data -> attribues)


```
<script src="js/app.js"></script>
<script src="js/second-app.js"></script>
```

The next logical question arises, how do we mitigate global scope variables? When developing applications is make it into a namespace. A namespace is one variable /a single name that we can Encompass or place a closure over all of our variables so that it's not going to expose every variable name in the global scope.

Back in the app.js file, we are going to create an object named app, initialized the object and some methods. We start creating a prefix of a $ sign as a bit of best practice (this creates a new sudo global variable for the app namespace). In the $ class, we initialize all selected HTML elements, we first init our dropdown menu, and next, we init menu items. We can no longer derive this based on the menu because after the menu is init-ed, we don't have a menu available to us, so we have to use the document.

```
const App = {
  // All of our selected HTML elements
  $: {
    menu: document.querySelector(".menu"),
    menuitems: document.querySelector(".items"),
  },
};
App.$.menu.addEventListener("click", (event) => {
  App.$.menuitems.classList.toggle("hidden"); // Toggle the hidden class back and
forth every time the button is clicked
});
```

You may be tempted to do this, but the below will result in an error

```
const App = { // ERROR
  // All of our selected HTML elements
  $: {
    menu: document.querySelector(".menu"),
    menuitems: App.$.menuquerySelector(".items"),
  },
};
```

OK, this code work now: but let's go ahead and make an improvement, and that would be to move our event listener out of the global scope. The init method is where we're going to add event listeners to our application (note this is es6 syntax: this is a short hand function property on an object that we're calling app)

```
init() {
    App.$.menu.addEventListener("click", (event) => {
      App.$.menuitems.classList.toggle("hidden"); // Toggle the hidden
class back and forth every time the button is clicked
    });
  },
};
window.addEventListener("load",() => App.init())// This waits for the
entired document to load
```

This can also be shorted to

```
window.addEventListener("load", App.init)// This waits for the entired
document to load
```

now rewrite the initializer to grab the item by id instead by class

```
 // All of our selected HTML elements
  $: {
    menu: document.querySelector('[data-id="menu"]'),
    menuitems: document.querySelector(".items"),
  },
```

Now complete the rest

```
$: {
    menu: document.querySelector('[data-id="menu"]'),
    menuitems: document.querySelector('[data-id="menu-items"]'),
    resetBtn: document.querySelector('[data-id="reset-btn"]'),
    newRoundBtn: document.querySelector('[data-id="new-round-btn”]'),
  },
```

Now add the rest of the event listeners as shown below

```
 init() {
```

```
    App.$.menu.addEventListener("click", (event) => {
      App.$.menuitems.classList.toggle("hidden"); // Toggle the hidden
class back and forth every time the button is clicked
    });

    App.$resetBtn.addEventListener('click', (event) =>{
      console.log("restBtn");
    });

    App.$newRoundBtn.addEventListener('click', (event) =>{
      console.log("newRoundBtn");
    });
  },
};
```

In the index.html, adds similar data id to the previous mentioned dropdown

```
 <div class="square shadow"data-id="square"></div>
```

A stable selector must be on all of the gameboard divs

Back in app.js, the square to the initializer as shown below:

```
squares:document.querySelectorAll('[data-id="square"]'),
```

And to test we will use the following output

```
init() {

    console.log(App.$squares);
…
```

Now For us to differentiate each square we need unique id's for each one, in the index.html the gameboards should now look like this.

```
        <div id="1" class="square shadow"data-id="square"></div>
        <div id="2" class="square shadow"data-id="square"></div>
        <div id="3" class="square shadow"data-id="square"></div>
        <div id="4" class="square shadow"data-id="square"></div>
        <div id="5" class="square shadow"data-id="square"></div>
        <div id="6" class="square shadow"data-id="square"></div>
        <div id="7" class="square shadow"data-id="square"></div>
        <div id="8" class="square shadow"data-id="square"></div>
        <div id="9" class="square shadow"data-id="square"></div>
```

Now we can add the event listener to seach square, but since we have a node list we will iterate through the event listeners like this. The below code prints each squares id

```
App.$.squares.forEach(square => {
      square.addEventListener("click",(event) => {
        console.log('Square with id ${event.target.id} was clicked')
      })
    })
```

Now to clean the code up a bit, we will create a new init method where we try to keep the init method clean looking. We do the following

```
init() {
    App.registerEventListener();

},

registerEventListener()
{


    App.$.menu.addEventListener("click", (event) => {
      App.$.menuitems.classList.toggle("hidden"); // Toggle the hidden
class back and forth every time the button is clicked
    });

    App.$.resetBtn.addEventListener('click', (event) =>{
      console.log("resetBtn");
    });

    App.$.newRoundBtn.addEventListener('click', (event) =>{
      console.log("newRoundBtn");
    });

    App.$.squares.forEach(square => {
      square.addEventListener("click",(event) => {
        console.log('Square with id ${event.target.id} was clicked')
      })
    })
  },
}
```

So far the only event listener that is completed is the event toggle. We will now switch back our focus to implementing the squares when they are clicked by changing the dom. The best way to approach this is to staticky render it in html the following demonstrates (switch back to index.html):

```
<div id="1" class ="square shadow" data-id="square">
        <i class="fa-solid fa-x yellow"></i>
      </div>
      <div id="2" class ="square shadow" data-id="square">
        <i class="fa-solid fa-o turquoise"></i>
      </div>
```

In order to render the x's and o's upon click, we must return to the app.js folder. The following addition allows us to click on a square and draw an x, but it come with come problematic features. For instance, when the square is clicked multiple times multiple x can occupay a single square

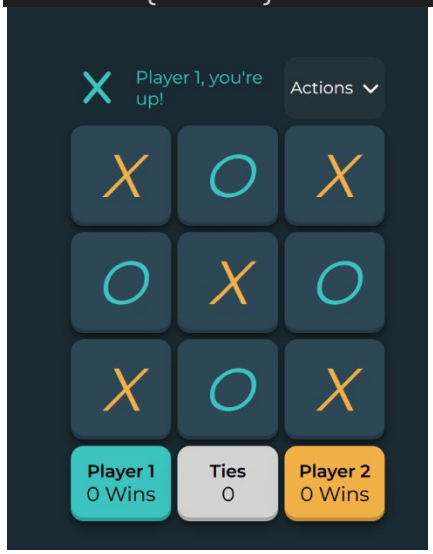
```
  square.addEventListener("click",(event) => {
        console.log("Square with id was clicked")

        const icon = document.createElement('i');
        icon.classList.add('fa-sold', 'fa-x', 'yellow');

        event.target.replaceChildren(icon);

        //<i class="fa-solid fa-x yellow"></i>
        //<i class="fa-solid fa-o turquoise"></i>
      });
```

Next, we add our logic; as we iterate over the squares, we create an icon if it is player 1's turn, then we paint a yellow on the square that's poked. If we care player two our pokes result in the paining turquois o's.

```
const currentPlayer = App.state.currentPlayer
        const icon = document.createElement('i');
        if(currentPlayer ==1)
        {
          icon.classList.add('fa-sold', 'fa-x', 'yellow');

        }else{
          icon.classList.add('fa-sold', 'fa-o', 'turquoise');
        }


        App.state.currentPlayer = App.state.currentPlayer === 1 ? 2 : 1

        event.target.replaceChildren(icon);
```

for our one icon per tile object, we add this logic statement as soon as we create the icon

```
if(event.target.hasChildNodes())
        {return}
```



Note that when we now click an occupied space nothing happens, but unforcahntly when we click on the icon its self it allows us to place another icon This is fix with the following code

```
if(square.target.hasChildNodes())
        {return}
        const currentPlayer =
App.state.currentPlayer
        const icon =
document.createElement('i');
        if(currentPlayer ==1)
        {
          icon.classList.add('fa-sold', 'fa-x',
'yellow');

        }else{
          icon.classList.add('fa-sold', 'fa-o', 'turquoise');
        }


        App.state.currentPlayer = App.state.currentPlayer === 1 ? 2 : 1

        square.target.replaceChildren(icon);
```

We continue to add to our game logic, we have opted to track the game state instead of parsing the board after every player moves. It looks like the following

```
square.addEventListener("click",(event) => {
       //check if there is already a play if so, return early
       if(square.hasChildNodes()){
        return
       }
        if(square.target.hasChildNodes())
        {return}
        // Determine witch player icon to add to the square
        const currentPlayer = App.state.currentPlayer
        const icon = document.createElement('i');
        if(currentPlayer ==1)
        {
          icon.classList.add('fa-sold', 'fa-x', 'yellow');
```

```
        }else{
          icon.classList.add('fa-sold', 'fa-o', 'turquoise');
        }
        App.state.currentPlayer = App.state.currentPlayer === 1 ? 2 : 1
        square.target.replaceChildren(icon);
        //check if there is a winner or an tie
```

Now, we need to keep track of an array of game moves where we keep track of the square player and the number of moves the player has made. The below is added after the first two if statements

```
const lastMove = App.state.moves.at(-1)

        const getOppositePlayer = (playerId) =>(playerId === 1 ? 2 : 1);

        const currentPlayer = App.state.moves.length === 0 ? 1 :
        getOppositePlayer(lastMove.playerId);
```

Next we implement the logic that determines the game state. First we declare player one by filtering and check where the player id is === to 1 and next we declare player two's moves via a similar statement. Then we declare the matric of winning patterns and check if either of the players moves have all three within their array. We assume that by default, the winner is null, then break out and iterate through the winning Pattern matrix; we then declare bool values if a winning pattern is detected. We can invoke the game state at the end of the square iteration to check the state. This is the final version of this method. Let's recap seeing how this method consists of most of the game logic.

1. Loop through each of the squares on the game boards.
2. We next add a click listener to each square
3. Check if there is a move already within that square
4. If there is we return
5. Next, check who moved last based on the state
6. We then get the opposite player and set them as the player of the game
7. Create an icon based on the player who is up
8. Push a new move to the state
9. Based on the state we'll add either an x or an o to the icon
10. Check the game's statue after that move has happened
11. if a win or tie is detected we display the modal and allow for the game to be reset

```
//1App.$.squares.forEach(square => {
//2       square.addEventListener("click", (event) => {
        // Check if there is already a play, if so, return early
//3         const hasMove = (squareId) => {
          const existingMove = App.state.moves.find((move) =>
move.squareId === squareId);
          return existingMove !== undefined;
        };
```

```
//4         if (hasMove(+square.id)) {
            return;
          }

          // Determine which player icon to add to the square
//5         const lastMove = App.state.moves.at(-1);
          const getOppositePlayer = (playerId) => (playerId === 1 ? 2 : 1);
//6         const currentPlayer = App.state.moves.length === 0 ? 1 :
getOppositePlayer(lastMove.playerId);

//7         const icon = document.createElement('i');
          if (currentPlayer === 1) {
            icon.classList.add('fa-solid', 'fa-x', 'yellow');
          } else {
            icon.classList.add('fa-solid', 'fa-o', 'turquoise');
          }

          App.state.moves.push({
            squareId: +square.id,
            playerId: currentPlayer,
          });

          square.replaceChildren(icon);

        // Check if there is a winner or a tie
          const game = App.getGameStatus(App.state.moves);

          if(game.status === 'complete')
          {
            App.$.modal.classList.remove('hidden')
            let message = ''
            if(game.winner)
            {
              message= "Player "+game.winner+' wins the game'
            }else{
             message = 'Tie game!'
            }

            App.$.modalText.textContent=message
          }
```

Now lets trigger our modal to open when someone has won the game or if there is a tie. We can do this in two steps, trigger the modal to open and then close when we are done with it. In the $ namespace create a new selector for the modal, as demonstrated below:

```
    modal: document.querySelectorAll('[data-id="modal"]'),
    modalText: document.querySelectorAll('[data-id="modal-text"]'),
    modalBtn: document.querySelectorAll('[data-id="modal-btn"]'),
```

Next place inside the game completed if statement the following to display the modal. Noe declare a variable message in the if statement and set it to blank, in the winner if statement concatenate the winner to message, else concatenate a tie, then add the message to the modals text contents

```
if(game.status === 'complete')
        {
          App.$.modal.classList.remove('hidden')
```

For the second to last step next we add an event listener to the modal button, first we set the moves state to blank, next we iterate through each square and replace it with a blank one, then we add the hidden tag back to the modal as follows:

```
 App.$.modalBtn.addEventListener("click", (event) => {

      App.state.moves = []
      App.$.squares.forEach((square) => square.replaceChildren());
      App.$.modal.classList.add('hidden');
    });
```

The last feature (counting how many wins and tie) we add a turn item to the list of selectable items in the $ name space, it should look like the following

```
turn: document.querySelector('[data-id="turn"]'),
```

finally fill in the logic like this:

```
const nextPlayer = getOppositePlayer(currentPlayer)
        const squareIcon = document.createElement("i");
        const turnIcon = document.createElement('i');
        const turnLabel = document.createElement('p');

        turnLabel.innerText = 'Player' + nextPlayer + ' you are up'
```

```
        if (currentPlayer === 1) {
          squareIcon.classList.add('fa-solid', 'fa-x',
'yellow');
          turnIcon.classList.add('fa-solid', 'fa-o',
'turquoise');
          turnLabel.classList='turquoise'

        } else {
          squareIcon.classList.add('fa-solid', 'fa-o',
'turquoise');
          turnIcon.classList.add('fa-solid', 'fa-x',
'yellow');
          turnLabel.classList='yellow'
        }
        App.$.turn.replaceChildren(turnIcon,
turnLabel)
```

Best practices when developing user interfaces.

1. Global scope and namespaces
2. Stable selectors (data -> attributes)
3. separate logic by responsibility (“separation of concerns ”)

As of now this code could be further optimized: its verbose and overall terrible to work wit, but alas we have a functional game. In the next section of this tutorial we will go about l now refactor this code

Introduction to the MVC pattern