


WORKSHOP #2

ROCK SCISSORS PAPER

WHERE TO START

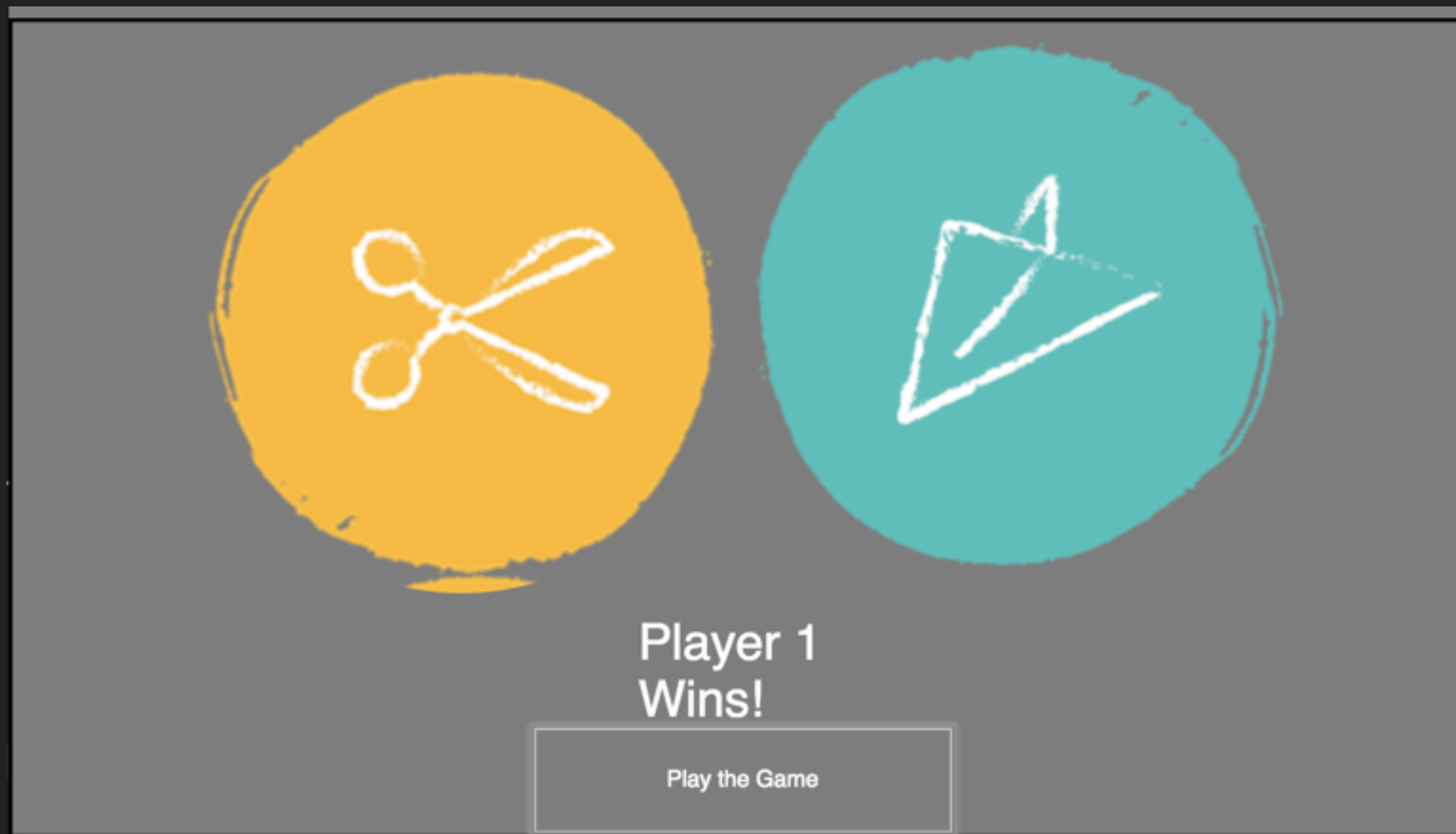
STACKBLITZ.COM

- ▶ Lets get started by navigating to stackblitz.com
- ▶ Click on the  button to start a new project.
You should find this as you scroll down the main page.

ROCK, SCISSORS, PAPER

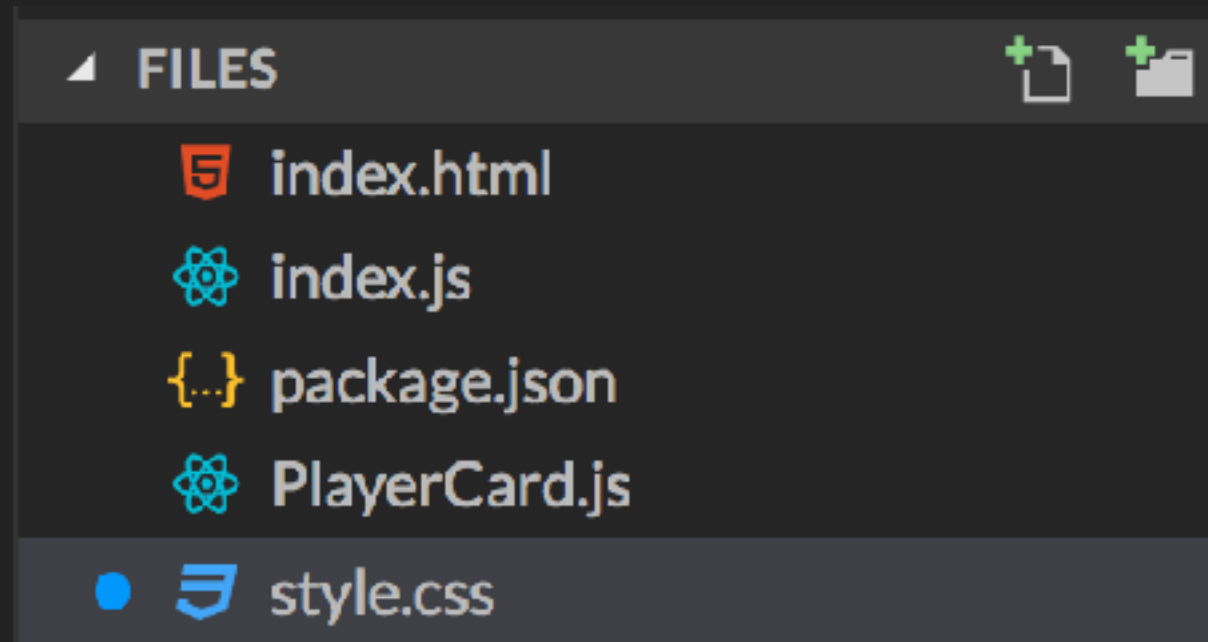
OUR GOAL

Our goal is to build an rock, scissors, paper game that will randomly generate both players picks and decide the winner.



ROCK, SCISSORS, PAPER

- ▶ Delete all the code in your index.js file and let's start from scratch again. Delete the Hello.js file from our file list on the left side, and create a file named PlayerCard.js.



This is how your Files tree should look.

- ▶ So now that we have an idea on what we want to create. Take a moment and think about where should we start. See if you can answer these questions.
- ▶ What is the major functionality of the application?
- ▶ What type of components should we implement based on the functionality of the application? I.E. Stateful class component, or a pure functional component?
- ▶ How will you structure the JSX and styles to accomplish this view or UI?

- ▶ We know that this application will randomly generate the picks.
- ▶ Our RPS application displays the players picks and the winner of the hand. Usually, and almost always, when you have values that are changing in your views you are going to need State.
- ▶ I'll provide the CSS and guide you along the way on how to add in styling.

- ▶ Lets start by navigating to the github and pulling the CSS that is prebuilt for this workshop.
- ▶ github.com/voodoobrew/workshop-1

ROCK, SCISSORS, PAPER

- ▶ Lets start by importing the proper libraries into your **index.js** file.
- ▶ Import React and the Component so we can create our stateful class component and use JSX. Then we want to import ReactDOM from the react-dom library so we can attach our component to the DOM. Finally import the PlayerCard component so we can use this in our main index.js file.

```
1  import React, { Component } from 'react';
2  import ReactDOM from 'react-dom'
3  import PlayerCard from './PlayerCard'
4  import './style.css';
5
```

Importing style.css file allows us to use that file to add styling to our JSX

- ▶ Now that we have our imports complete we need to create our Game class component.

```
6  class Game extends Component {  
7    render () {  
8      return (  
9        <div></div>  
10      )  
11    }  
12  }
```

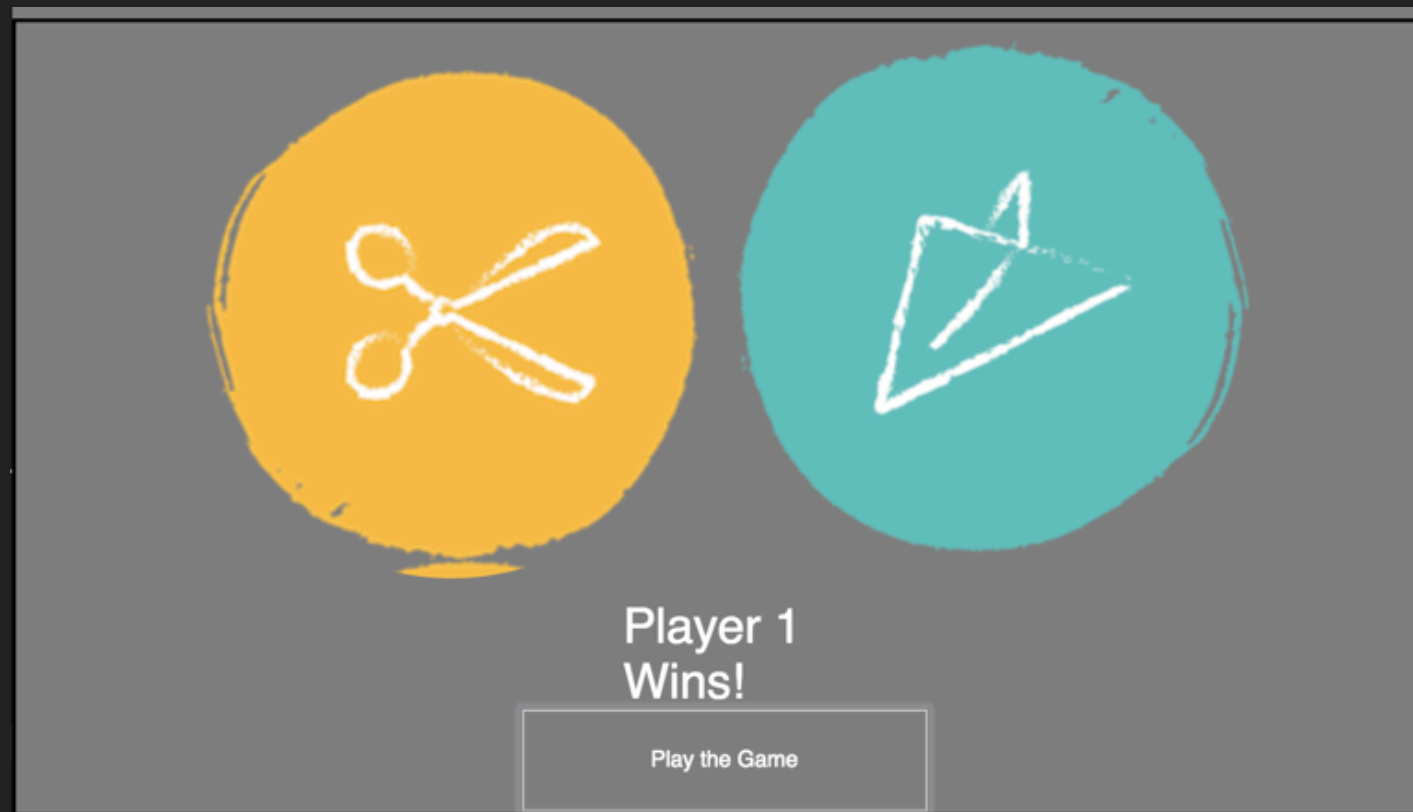
- ▶ Render you Game component to the DOM by declaring it as the first argument in your ReactDOM.render method. Note that we write the component as a self-closing JSX tag. This will be the very last line of code in our main class component.

```
ReactDOM.render(<Game />, document.getElementById('root'));
```

The first argument for the render method is the component you wish to attach to the DOM, and the second argument is how we attach it to a specific HTML element. In this case its the element with the id 'root'.

ROCK, SCISSORS, PAPER

- ▶ Now that we have our component built, we should think about what we want our app to look like so we can start building the JSX. Lets take a look at the end result again.



Here we will the two PlayerCard components

Here we display who won the round and the button to Play the Game.

ROCK, SCISSORS, PAPER

- ▶ To help get you started lets go ahead and build out what we know so far.

```
render() {  
  return (  
    <div className="style">  
      <div>  
        Put the PlayerCard Components here  
      </div>  
      <div className="winner">Place the winner here</div>  
      <button type="button">Play the Game</button>  
    </div>  
  )  
}
```

In your return statements you can only have one parent JSX element. A common pattern is to wrap everything in a div.

- ▶ In this workshop we will be using a functional component we defined as `PlayerCard`. Lets navigate to the **PlayerCard.js** file and start coding our functional component.
- ▶ This component is going to display each players randomly generated pick.

```
7  const PlayerCard = (props) => {  
8    return (  
9      <div className="playerCard"></div>  
10   )  
11 }
```

ROCK, SCISSORS, PAPER

- ▶ We only want two of our PlayerCard components to be rendered in our main index.js file between a div tag.

```
return (  
  <div className="style">  
    <div>  
      <PlayerCard />  
      <PlayerCard />  
    </div>  
    <div className="winner">Put the winner here</div>  
    <button type="button" >Play the Game</button>  
  </div>  
)
```

You should see two circles
with a button

- ▶ Remember, that we need a stateful component. Think for a moment: what should change about the view of our RPS app as user interact with it, and how could we represent that data?
- ▶ The data that changes in our UI is the players generated pick that is being represented by a string at first and then an image. We can represent this as a string. Let's add state to our component, and some behavior that will cause that state to change.
- ▶ To do this we need to add a constructor to your Game class component and define state. Note that state is always defined as a Javascript object that have key value pairs.

- ▶ In this workshop we are going to be adding the signs as well. We will use this array to randomly generate picks from, but now we can set them in our constructor so we can use them through our application.

```
6  class Game extends Component {  
7    constructor() {  
8      super();  
9      this.signs = ["rock", "scissors", "paper"]  
10     this.state = {  
11       playerOne: "rock",  
12       playerTwo: "scissors",  
13     }  
14   }
```

We initialize our state by setting this.state to a Javascript object and the key and value we want our component to be initialized to.

- ▶ Now that we have state that represents the playerOne and playerTwo picks. We want to pass those values into our PlayerCard component that will allow us to represent each players generated choice.

```
<div>
  <PlayerCard sign={this.state.playerOne} />
  <PlayerCard sign={this.state.playerTwo} />
</div>
```

The sign property we are passing to our component is going to be passed in as a key on the props object.

ROCK, SCISSORS, PAPER

- ▶ Now that we're passing the sign property into our PlayerCard component. Lets grab that and represent that data between our div tags.
- ▶ Remember that props are an object with key value pairs so we can use the dot notation to access the values within the object.

```
7  const PlayerCard = (props) => {  
8    return (  
9      <div className="playerCard">{props.sign}</div>  
10    )  
11  }
```

ROCK, SCISSORS, PAPER

- ▶ Now that we have our blueprint all set up. Lets make this game randomly generate the signs.
- ▶ Navigate to our *index.js* file. We want to create a method in our Game component that will allow us to set the state to a randomly generated sign from our signs array. Think for a moment how you would do this in Javascript.

```
playGame = () => {  
  
}
```

- ▶ When we need to set the state to a new value. We always call the `setState` method. `setState` is actually copying a new value into the key `playerOne` and `playerTwo` like it would if these were a brand new object.

```
16  ▢ playGame = () => {  
17  ▢   this.setState({  
18     playerOne: this.signs[Math.floor(Math.random() * 3)],  
19     playerTwo: this.signs[Math.floor(Math.random() * 3)],  
20   })  
21 }
```

We use the `signs` array and then call `Math.random() * 3` because `Math.random` generates values between 0-1. Then we use `Math.floor()` to remove any decimal places and round to the nearest number.

ROCK, SCISSORS, PAPER

- ▶ Now that we have our method we should be able to attach a `onClick` event listener to our button to make sure this method fires when we click Play the Game.

```
return (  
  <div className="style">  
    <div>  
      <PlayerCard sign={this.state.playerOne} />  
      <PlayerCard sign={this.state.playerTwo} />  
    </div>  
    <div className="winner">Put the Winner Here</div>  
    <button type="button" onClick={this.playGame}>Play the Game</button>  
  </div>  
) ;
```

- ▶ Our game should now randomly generate signs for each player.
- ▶ Now we need to make sure our Game decides who has won the round. To do this we need to create another method. Lets call this method decideWinner.

```
16      decideWinner = () => {  
17  
18      }
```

Think for a moment on how this is actually going to decide the winner.

- ▶ We know that rock beats scissors, paper beats rock, and scissors beats paper. Lets represent these conditionals in our method to decide who has won the round. If we coded this properly we should only have to do it for one player.

```
17  decideWinner = () => {
18    const playerOne = this.state.playerOne
19    const playerTwo = this.state.playerTwo
20
21    if (playerOne === playerTwo) {
22      return "It's a Tie!"
23    }
24    else if ((playerOne === "rock" && playerTwo === "scissors") ||
25             (playerOne === "paper" && playerTwo === "rock") ||
26             (playerOne === "scissors" && playerTwo === "paper")) {
27
28      return "Player 1 Wins!"
29    } else {
30      return "Player 2 wins!"
31    }
32  }
```

- ▶ Now we have the ability to decide who wins the round. We know we wanted to represent who won above the "Play the Game button" so lets place in the correct spot.

```
41  render() {  
42    return (  
43      <div className="style">  
44        <div>  
45          <PlayerCard sign={this.state.playerOne} />  
46          <PlayerCard sign={this.state.playerTwo} />  
47        </div>  
48        <div className="winner">{this.decideWinner()}</div>  
49        <button type="button" onClick={this.playGame}>Play the Game</button>  
50      </div>  
51    );  
52  }
```

We know that every time the button is clicked that state changes, causing our game to re-render. By invoking our decideWinner method in this manner, this will run every time the button is clicked. Giving us the correct winner to be represented.

ROCK, SCISSORS, PAPER

- ▶ Now our end goal was to have images instead of the plain text we see rendering to our application. Lets set up where our PlayerCard component displays an image.
- ▶ First we have to pull these images in from a external website. This is because stackblitz cannot handle static assets.
- ▶ In our PlayerCard.js file lets add these lines of code.

```
3  const scissors = "https://i.imgur.com/pgjyhIZ.png";  
4  const rock = "https://i.imgur.com/LghSkIw.png";  
5  const paper = "https://i.imgur.com/2gsdqvR.png";
```

- ▶ Now that we have the url of our images. We need to set these images in our PlayerCard.js component. We can accomplish this by adding a `` tag between the main `<div>` in our JSX.

```
19     return (  
20         <div className="player-card">  
21             <img src={rock}/>  
22         </div>  
23     )
```

Right now we have it hard coded to our rock image. Lets think about how we're going to represent this data if the `props.sign` being passed in is something other than rock.

ROCK, SCISSORS, PAPER

- ▶ To accomplish the task of rendering the correct image corresponding to our props.sign value. We need to first create a variable that will hold our image string that we will set to depending on what value is being passed in.

```
7  const PlayerCard = (props) => {
8      const sign = props.sign
9      const image= ""
10
11  if (sign == "rock") {
12      image = rock;
13  } else if (sign == "paper") {
14      image = paper;
15  } else {
16      image = scissors
17  }
18
19  return (
20      <div className="player-card">
21          <img src={image}/>
22      </div>
23  )
24  }
```

We create our image variable on line 9

Then, depending on what value the sign being pass in is. We then set the image to the correct url to be displayed.

Then we set the src of our `` tag to represent this url

ROCK, SCISSORS, PAPER EDGE CASES

- ▶ Now your game should be correctly displaying the images, generating random signs, and picking the correct winner.
- ▶ EDGE CASE TIME!
- ▶ Set some names for the players.
- ▶ Make one player controlled and able to select what sign to play.