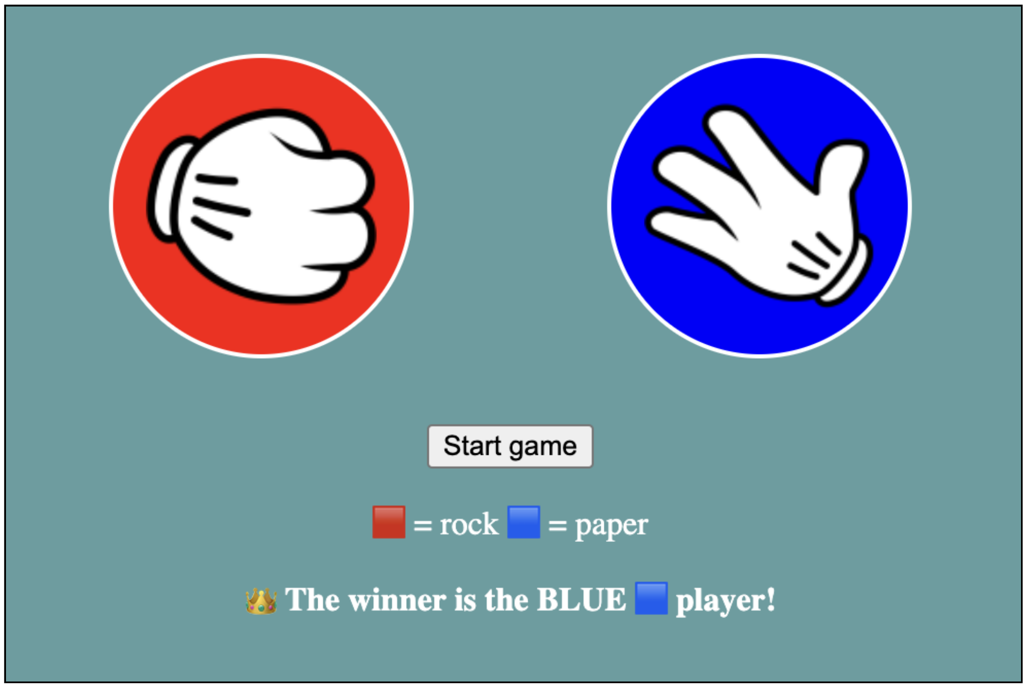
Example: let's build a small game with React

We have learned so many cool things about how to use various features of React. Now let the fun start! Let's put all of this to work by making a few small functional apps.

And what could be more fun, as our first example, than building a real small working game !?

It will be a Rock Paper Scissor game. This is how our app will look like:



When the user presses the `Start game` button then our game will do the following:

- start a shuffling animation. The animation is made by randomly change the rock, paper, or scissor signs for both players 20 times
- pick a final random rock, paper, or scissor sign for the Red player
- pick also a random sign for the Blue player
- and will decide the winner of the round. The rules are that rock wins against scissors, paper wins against rock, and scissors wins against paper.

You can see a short video of the application here.

▶ SEE VIDEO FOR DETAILS

What are our learning objectives for his example:

- splitting the app into smaller React components
- working with basic concepts of components such as props, state or inline styles
- parent components triggering events in child components
- the useEffect() hook, and its cleanup function
- using the Context API and the useContext() hook for global state management
- using refs to store values and prevent bugs

... and much more.

So, let's get to work!

The general arhitecure of our app

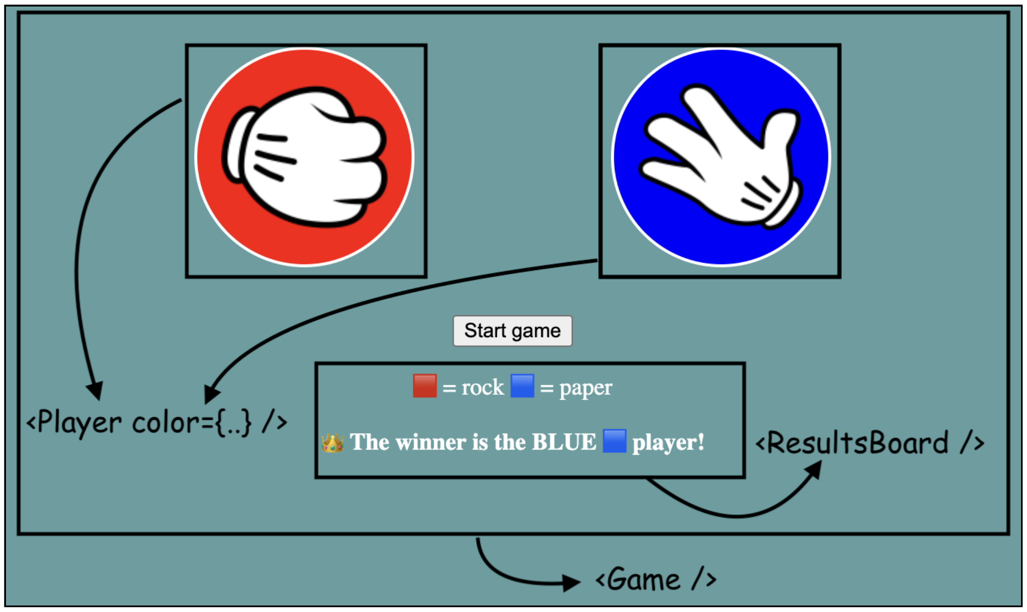
Before we can get our hands dirty we will need to stop and think a bit about the architecture of our app. There are two aspects we need to decide upon:

- how the UI will be split into React components
- and how these components will interact, how the data will flow between them

For this particular example, I've thought we could have the following React components:

```
├── App
| ├── Game
| ├──── Player
| ├──── ResultBoard
```

Let's see what this will look like on the screen:



.

The main scope of the `App` component will be to render a `Game` component plus define an app game context (more about this in the next sections).

The `Game` component will serve as the main container of our app. It will contain the button to start the game, it will initiate both the components of type `Player` and the `ResultBoard` component.

Next, we have the `Player` component. Our game will have two instances of this component. It will use the `color` property to customize the look of each instance.

Finally, we have the `ResultBoard` component. The role of this one is to check what were the signs picked by the Red and Blue players and decide the winner.

By the way, we can have various other ways of splitting this UI into components. I encourage you to experiment with different component configurations and test their corresponding advantages and challenges.

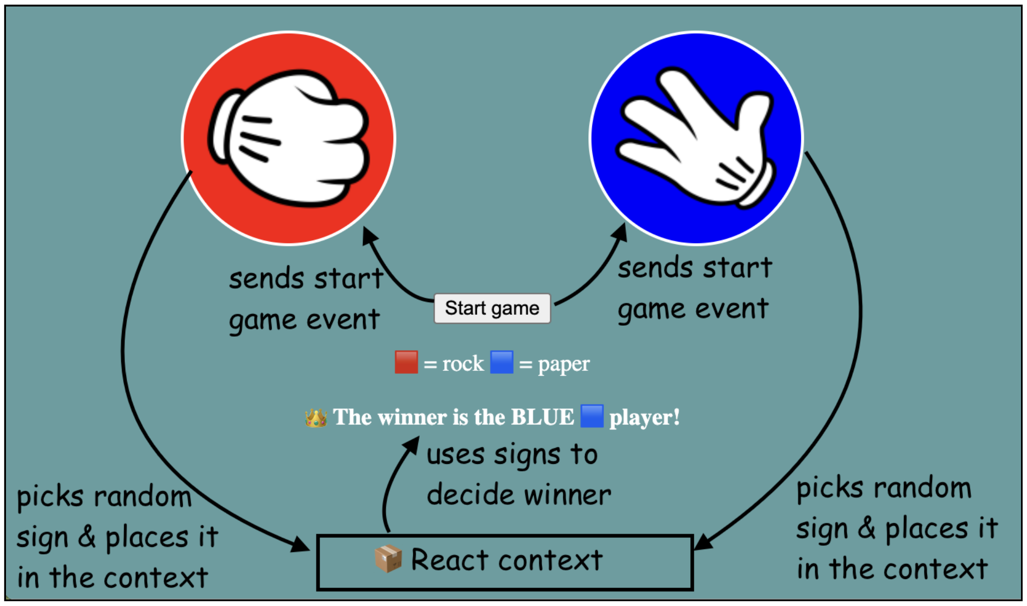
How the data will flow between components

One other important thing to decide upon is how all of these components will interact with each other.

Also in this case there is no single answer. We can have something like a useReducer(), or even a full Redux store in place. Or just pass all via simple props (even though this will be a bit clumsy).

After the exercise is ready, just try to play with different approaches and see how it goes. I guarantee that you will learn a lot from all of these experiments.

Coming back to our example I've decided to use the following data flow:


sends start game event
Start game
sends start game event
🟥 = rock 🟦 = paper
👑 The winner is the BLUE 🟦 player!
uses signs to decide winner
picks random sign & places it in the context
📦 React context
picks random sign & places it in the context

Let's see how all of this works:

- the button from the `Game` component will send an event to both `Player` components
- when a start game event is received, a `Player` component will do the following:
  1. it will run the shuffling signs animation
  2. it will pick a random sign from the `['rock', 'paper', 'scissors']` array
  3. it will place the selected sign in a React Context store. In our case, it will be named `GameContext`
- after each player has picked its random sign then the `ResultBoard` component comes into play. It will retrieve from the `GameContext` the random signs, decide the winner of the game, and show the corresponding message.

Now the fun part starts! Coding!

The initial setup

We will start with the below code:

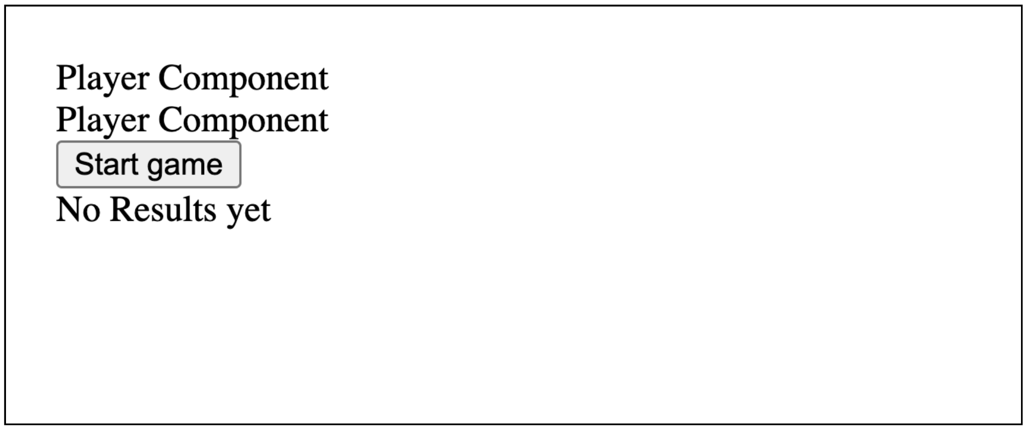
```
const Player = ()=> {
    return (<div className='player'>
        Player Component
    </div>)
}
```

```
const ResultBoard = () => {
    return (<div>
        No Results yet
    </div>)
}

const Game = () => {
    return (<div>
        <div className='container'>
            <Player />
            <Player />
        </div>
        <button>Start game</button>
        <ResultBoard />
    </div>)
}

const App = ()=> (<Game />)

const root = ReactDOM.createRoot(document.getElementById("root"))
root.render(<App />)
```

Nothing fancy here. Just the skeleton for our `App`, `Game`, `Player` and `ResultBoard` components.

The `Game` component will render two instances of `Player` components and one `ResultBoard`.

Thefore, we will start from this layout:


```
Player Component
Player Component
```

```
No Results yet
```

Adding props and picking a random sign in the Player component

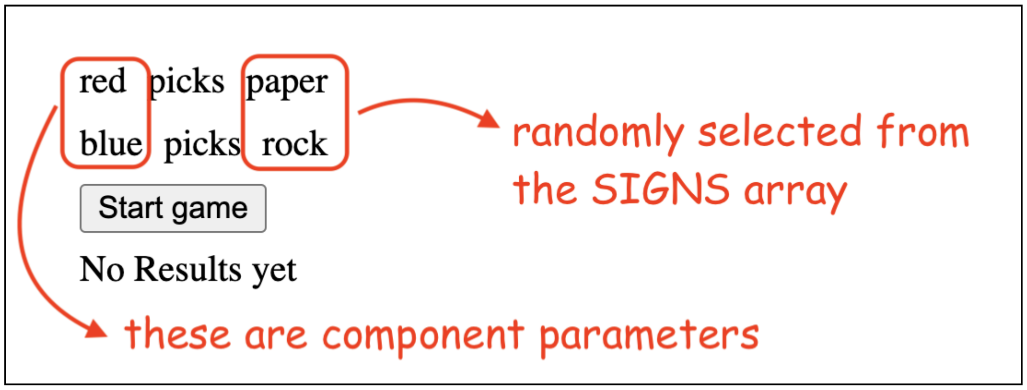
At this point both instances of the Player component are identical.

We will need to customize them by passing a `color` parameter for each player. Also, we want them to select a random sign from an array like this one: `const SIGNS = ['rock', 'paper', 'scissors']`.

This is the code that will achieve this:

```
// Define an array of possible signs for the game
const SIGNS = ['rock', 'paper', 'scissors']

const Player = ({color})=> {
    // Generate a random index to pick a random sign from the array
    const randIndex = Math.floor(Math.random() * SIGNS.length)
    const randSign = SIGNS[randIndex]

    return (<div className='player'>
        {color} picks { randSign }
    </div>)
}

// the ResultBoard component remains unchanged

const Game = () => {
    return (<div>
        <div className='container'>
            <Player color='red' />
            <Player color='blue' />
        </div>
        <button>Start game</button>
        <ResultBoard />
    </div>)
}

const App = ()=> (<Game />)
```

This is how our app stands at the moment:


red picks paper
blue picks rock
Start game
No Results yet
randomly selected from the SIGNS array
these are component parameters

You can also see it live in this code sandbox:

▶ RUN CODE

CSS styles

Each app needs some CSS styles. Given that this book is not about CSS we will not go too much into these details. But overall the CSS property names should be quite self-explanatory.

You may have noticed that some elements have the `className` property.

```
<div className='player'>

<div className='container'>
```

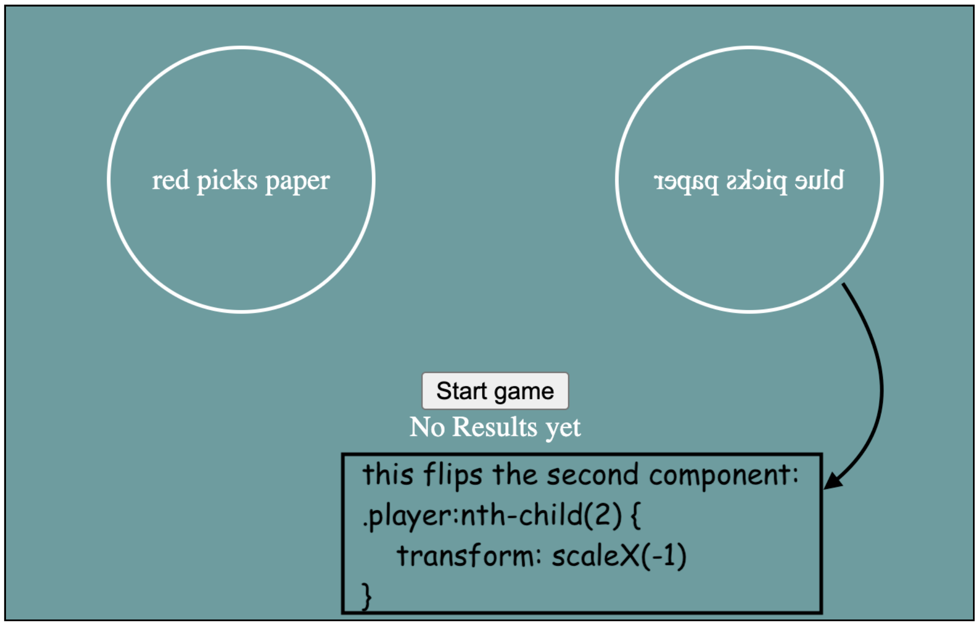
This property will link the given element with the corresponding CSS class:

This is all the CSS code that we will use in our app:

```
body {
  color: white;
  background-color: cadetblue;
  text-align: center;
}

.container {
  display: flex;
  margin: 2rem;
  justify-content: space-around;
  max-width: 35rem;
  margin: 2rem auto;
}

.player {
  --card-size: 7rem;

  width: var(--card-size);
  height: var(--card-size);
  line-height: var(--card-size);
  border: 2px solid;
  border-radius: 50%;
  background-repeat: no-repeat;
  background-size: contain;
  padding: 1rem;
  background-origin: content-box;
  &:nth-child(2) {
      transform: scaleX(-1)
  }
}
```

A few things to notice about the `.player` class :

- the class uses native CSS selector nesting. If you are interested I've written a separate article about it here.
- we use the `--card-size` CSS variable to sync the values in different properties
- the `transform: scaleX(-1)` property applied to the second Player component allows us to flip its view so that the signs face each other. You will see in the next section why we need this. For the moment it only flips the text of the component:

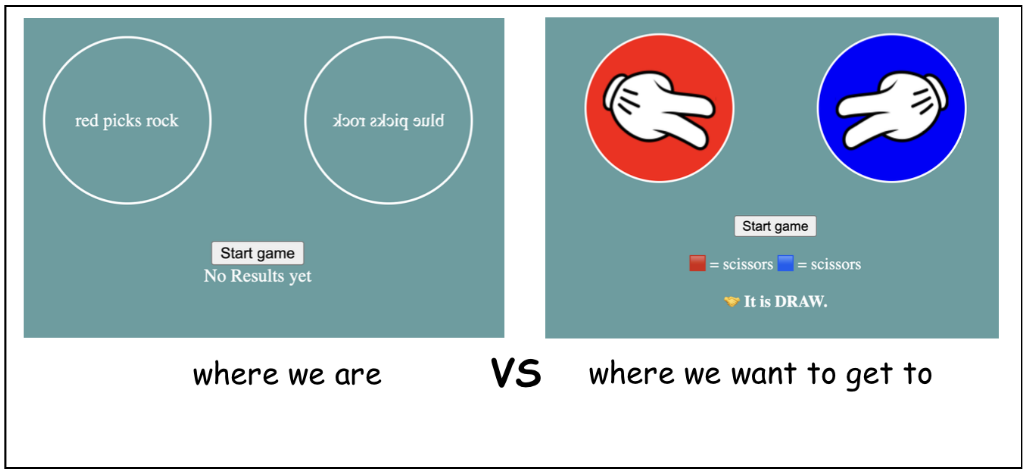


```
this flips the second component:
.player:nth-child(2) {
    transform: scaleX(-1)
}
```

▶ RUN CODE

Using inline styles

Our UI is quite dull right now. Take a look at where we are and where we want to get to:



Would be nice to use pictures instead of text for the rock, paper, scissors signs and have the player components colored in red and blue.

Well, this is what we will do in this chapter!

For the "use pictures instead of text for the signs" I've already made some pictures we can use and hosted them on the site. You can see each of them in this list:

- rock: https://www.js-craft.io/_assets/rpc/rock.png
- paper: https://www.js-craft.io/_assets/rpc/paper.png
- scissors: https://www.js-craft.io/_assets/rpc/scissors.png

Now how can we get these pictures, and the red-blue colors into our components? We can use inline styles for this.

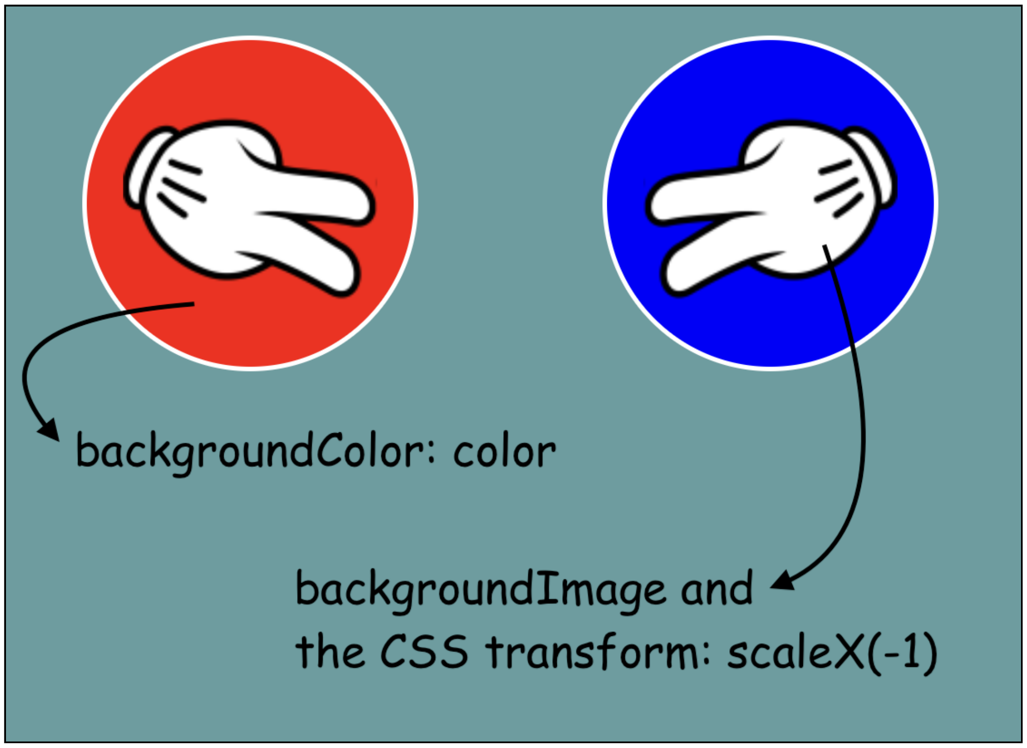
For each React component we can specify CSS inline style like so:

```
<Element style={{color: 'white', backgroundColor: 'red'}} />
```

Knowing this we can now use the received `color` parameter and the randomly generated `randSign` to set the needed inline style so that our components are colored and have pictures instead of text:

```
const SIGNS = ['rock', 'paper', 'scissors']

const Player = ({color})=> {
    const randIndex = Math.floor(Math.random() * SIGNS.length)
    const randSign = SIGNS[randIndex]
    const backgroundImage = `url(https://www.js-craft.io/_assets/rpc/${rand

    const inlineStyles = {
        backgroundColor: color,
        backgroundImage
    }

    return (<div className='player' style={inlineStyles} />)
}
```

And this is the current look of our app:



Note that for the `backgroundImage` style we are using template literals to interpolate values into a string:

```
`url(https://www.js-craft.io/_assets/rpc/${randSign}.png)`
```

By the way, if you want to see a cool trick with template literals check out this article I wrote some while ago.

Even if we are lacking parts of functionality such as shuffling animations or deciding the winner, we have made real progress here!

► RUN CODE

Next, let's see how to make that start button do something.

Invoking a function in a child component from a parent component

Before we make the button to actually start the game let's discuss a bit about the initial state of our app.

At this moment the random signs are selected for the players when the game first shows up on the page. But if we take a look at the behavior of the final example we see that initially, the players don't have a sign and random selection should take place only when the user presses the start game button.

First of all, this means that the state of our component will be changed: from a "show no sign" to "show a random sign".

If so, we will need to add a state to the `Player` components:

```
const Player = ({color})=> {
    const [sign, setSign] = useState(null)

    const pickRandomSign = ()=> {
        const randIndex = Math.floor(Math.random() * SIGNS.length)
        const randSign = SIGNS[randIndex]
        // will need to put the randSign in sign
    }

    const backgroundImage = sign ? `url(https://www.js-craft.io/_assets/rpc

    const inlineStyles = {
        backgroundColor: color,
        backgroundImage
    }

    return (<div className='player' style={inlineStyles} />)
}
```

I've made a couple of changes here:

- first the `sign` is now managed by a state variable. This will allow us to change the signs when the start game button is pressed
- given that the `sign` starts now with a null value, we will need to check if we can set a background image: `sign ?` url(https://www.js-craft.io/_assets/rpc/${randSign}.png)` : ''`

- and we moved the random selection of the signs to a separate method `pickRandomSign()`. For the time being this method is not used at all.

In some cases, a developer may need to call a child's function from a parent component. In our case, we want the `Game` component to be able to call the `pickRandomSign()` method from within its two children of type `Player`.

We can achieve this in various ways. For example, one way is to use a mix of `useImperativeHandle()` and `forwardRef()`.

For this example, I will do it via the `useEffect()` hook and a trigger state. And the reason for this is to get a better grip on the fundaments of working with useEffect.

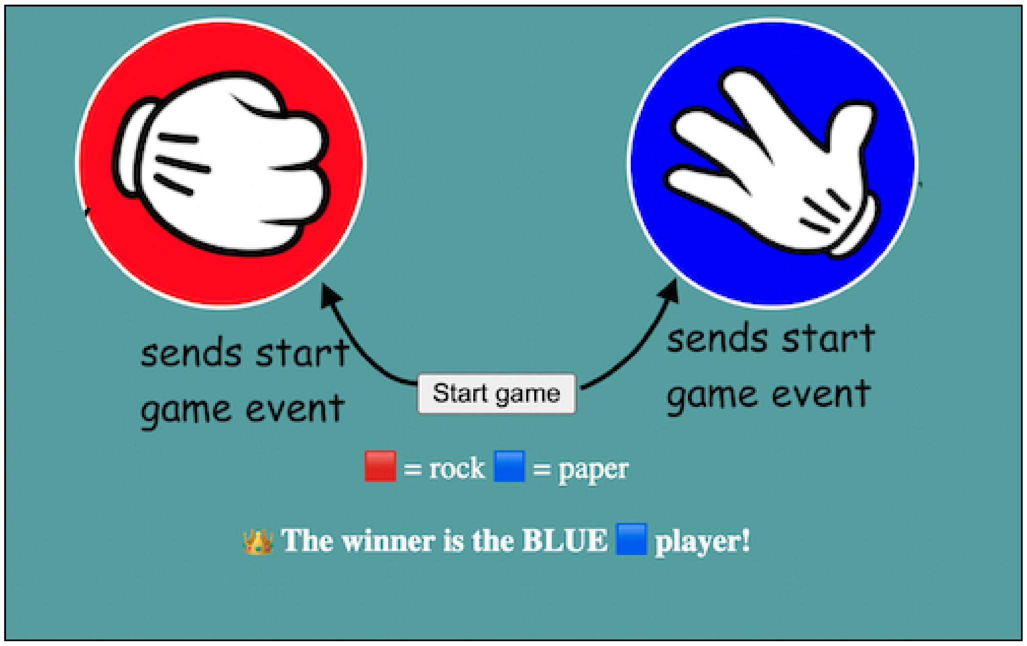
Here is what we are going to do:

1. Define a trigger state within the parent component and pass it down to the child component. We will name it `startTime`
2. In the child monitor changes in this state with useEffect, and when it undergoes a change, invoke the needed function.
3. With this in place, if we wish to call the `pickRandomSign()` function from the `Game` component, ensure to modify the trigger state `startTime`

Ok, I know this may sound confusing, but it will get simpler when we see it in code:

```
const Player = ({color, startTime})=> {
    const [sign, setSign] = useState(null)

    useEffect(()=> {
        if (startTime) {
            pickRandomSign()
        }
    }, [startTime])

    const pickRandomSign = ()=> {
        const randIndex = Math.floor(Math.random() * SIGNS.length)
        const randSign = SIGNS[randIndex]
        setSign(randSign)
    }

    const backgroundImage = sign ? `url(https://www.js-craft.io/_assets/rpc

    const inlineStyles = {
        backgroundColor: color,
        backgroundImage
    }
    return (<div className='player' style={inlineStyles} />)
}

const Game = () => {
    const [startTime, setStartTime] = useState(null)

    const startGame = ()=> setStartTime(Date.now())

    return (<div>
        <div className='container'>
            <Player color='red' startTime={startTime}  />
            <Player color='blue' startTime={startTime} />
        </div>
        <button onClick={startGame}>Start game</button>
        <ResultBoard />
    </div>)
}
```

What we have done here is solve this part from the data flow diagram:



Each time we press the button a new date is generated by `Date.now()` and placed into `startTime`. This update will be propagated to the child `startTime` parameter and from there `useEffect()` calls `pickRandomSign()`.

▶ RUN CODE

Next on the list? Adding animations.

Adding animations

How can we add that shuffle animations? Currently, we just pick a random sign and that's it.

In this case, also there are multiple ways to achieve the same goal. I think we can do it even with pure CSS.

But to go more into the weeds with how React works, will make this using a timer and the `setInterval()` function of Javascript.

In order to simulate the animation what we do is to pick a new random sign at a 100ms interval for each Player component.

We will track how many random picks we have done in a state variable named `animCounter`.

After 20 random picks, we will stop by calling `clearInterval()` and keep the last sign.

Let's translate all of this into code:

```
const Player = ({color, startTime})=> {
    const [sign, setSign] = useState(null)
    // keeps track of the number of steps that compose the shuffling animat
    const [animCounter, setAnimCounter] = useState(0)

    useEffect(()=> {
        if (startTime) {
            pickRandomSign()
        }
    }, [startTime])

    const pickRandomSign = ()=> {
        // reset the counter at each animation start
        setAnimCounter(0)
        let animInterval = setInterval(() => {
            // updating a state based on a previous value; see earlier chap
            setAnimCounter(prev => prev + 1)
            const randIndex = Math.floor(Math.random() * SIGNS.length)
            const randSign = SIGNS[randIndex]
            setSign(randSign)
            // stop the random picks after 20 iterations
            if(animCounter > 20) {
                clearInterval(animInterval)
            }
        }, 100)
    }

    const backgroundImage = sign ? `url(https://www.js-craft.io/_assets/rpc
    const inlineStyles = {
        backgroundColor: color,
        backgroundImage
    }

    return (<div className='player' style={inlineStyles} />)
}
```

You can take a look at the current state of the app in this sandbox.

▶ RUN CODE

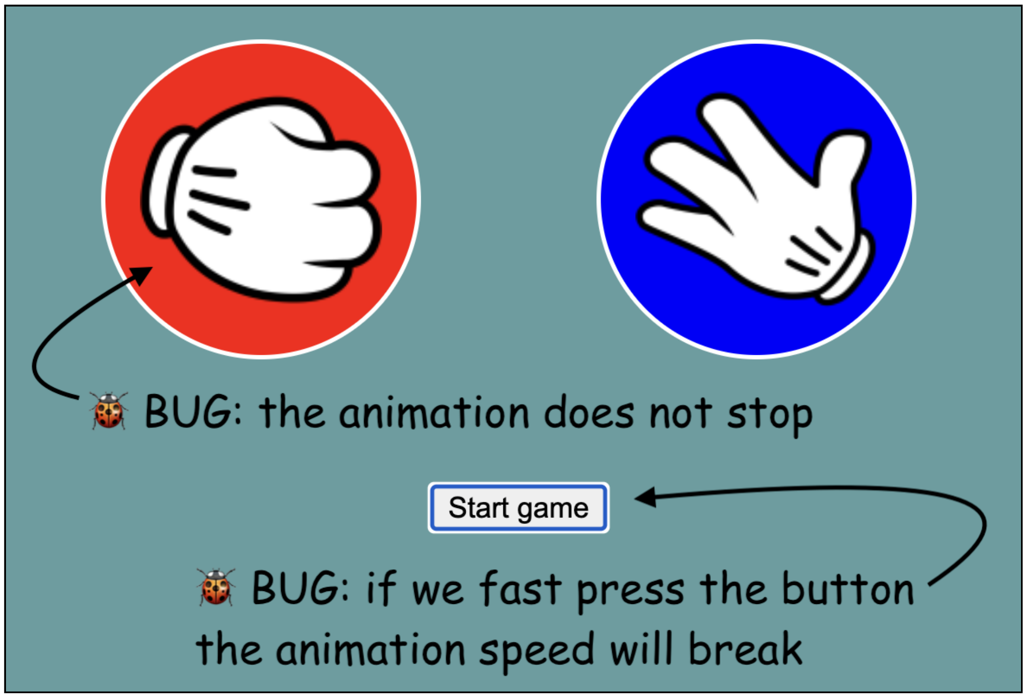
Unfortunately, while the animation starts nicely we have some bugs in our application.

Can you spot them? Take a deeper look. I will wait.

....

....

Ok, here they are:


🐞 BUG: the animation does not stop
Start game
🐞 BUG: if we fast press the button the animation speed will break

Do you have any idea why they happen? Let's see in the next chapter how to fix them.