sitepoint

Build a Rock Paper Scissors Game from Scratch with React

Build a Rock Paper Scissors Game from Scratch with React

- **Author:** Madars Biss
- **Cover Design:** Alex Walker

Notice of Rights

Notice of Liability

Trademark Notice

Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood
VIC Australia 3066
Web: www.sitepoint.com
Email: books@sitepoint.com

About SitePoint

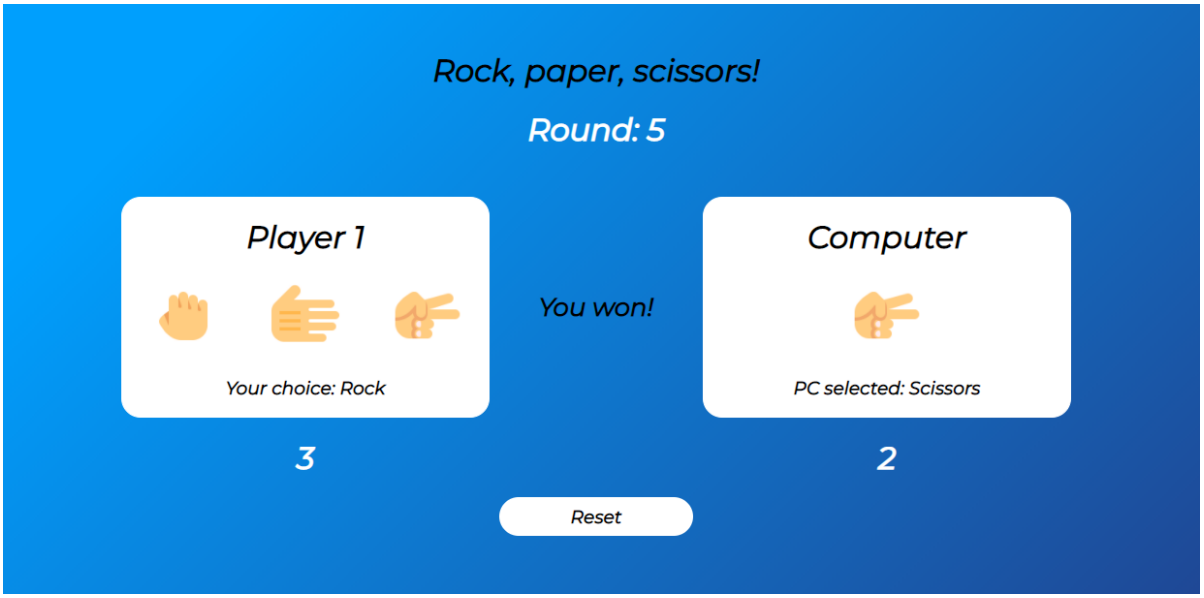
SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit sitepoint.com to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, Ruby, mobile development, design, and more.

Build a Rock Paper Scissors Game from Scratch with React

In this tutorial, we'll build a Rock Paper Scissors game that allows users to play against the computer. We'll use React to build the app. React is a feature-rich JavaScript library for building interactive user interfaces for websites, web applications and games.

We'll learn how to create the wireframe, style it, set up the project, keep track of states, create the components, and implement the game logic.

For reference, you can check the source code and the deployed demo.



Planning the Features

Traditionally, Rock Paper Scissors is a hand game where each player simultaneously chooses one of three possible shapes for their hand: rock, paper or scissors. Rock beats scissors, paper beats rock, and scissors beats paper. Of course, in our game we'll be playing against the computer, so we'll use icons instead of our hand.

The user will be able to choose between three options by clicking on them. To improve the UX, we'll use the icons that represent each choice.

The opponent to beat will be the computer, whose choice will be calculated randomly. We'll create a function for that in the later phases of the tutorial.

The choices of each individual round will be compared. The first player to reach a specific number of wins (ten by default) will be victorious in the whole game, and the user will have an option to start another game.

Creating the Wireframe

To better understand the layout we'll need to create, we'll first make a wireframe with all the necessary components. Our main focus will be on the position of the elements in the game wrapper.

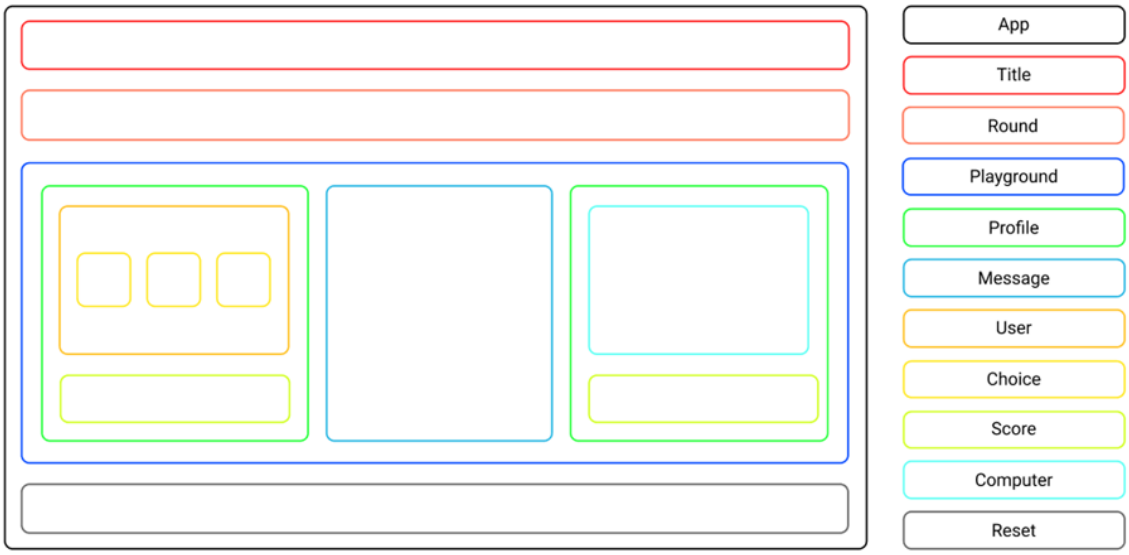
We'll use the top section of the game to display the game name and show the number of rounds that have been played.

The main play area will be divided into two main blocks. The first one will be for the user, and the second one for the computer. In these blocks, we'll include information such as the player name, choice options, and score.

Between the two main blocks, we'll include the message component, which will display the result of each round and display the end result of the game.

Under the main play area, we'll dedicate a space for the reset button, so users will be able to reset and start fresh at any point of the game.

If we put everything on the wireframe, we get the schema pictured below.



Styling the Game

We'll use a dark blue gradient as a background. We'll also set the dark blue fallback background value for the browsers that don't support the CSS properties for gradients.

Both player cards will have a white background so they're highlighted on the dark blue background. Inside the player cards, the information like player name and selection message will have black text, which will give a great contrast to the white background of the card.

The score and the round count will be highlighted, since that's the most important information of the game's progress. Both will be displayed on a dark blue background directly. The use of white text will make sure they're easy to read.

Other information like the game name and round messages will be secondary, so they'll use the black font color that will be displayed on the dark blue.

Our color palette is shown below.

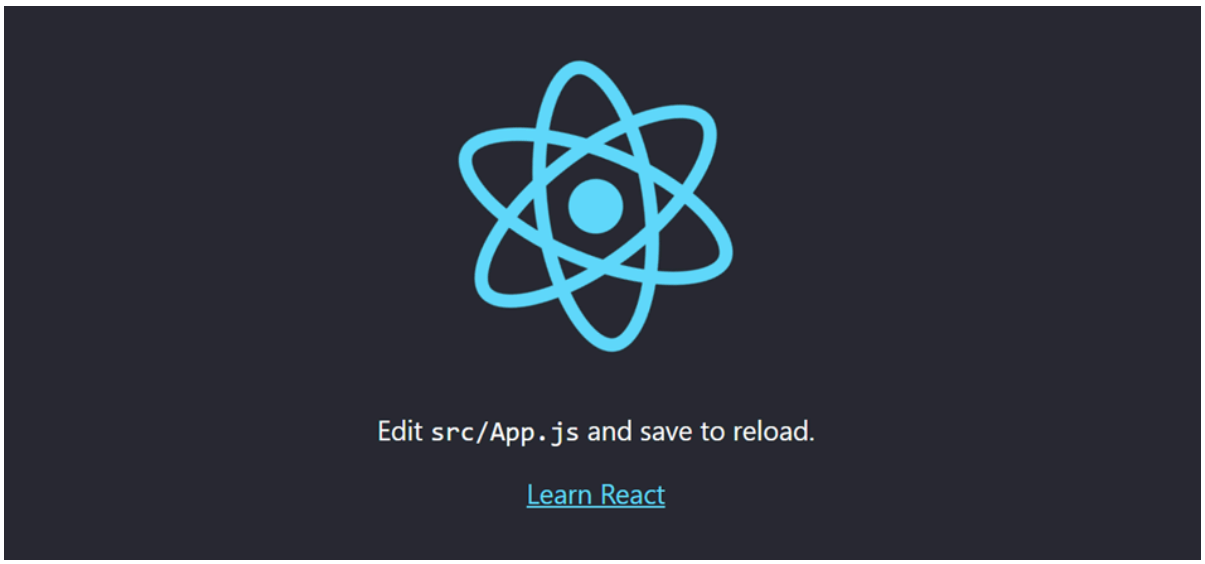
Setting up the Project

In order to set up the boilerplate for the project, we'll use Create React App, which will initialize a fully configured React project in a minute or less.

To do that, open your terminal and run `npx create-react-app rock-paper-scissors`.

After the setup is complete, switch to the newly created folder by running `cd rock-paper-scissors` and then run `npm start`, which will start the development server.

This should open your browser automatically, and you should be presented with the default React app. If not, enter `http://localhost:3000/` manually in the browser's URL bar and execute it.



We'll need to do some cleaning. Open the `src` folder and remove everything except the files `index.js` and `App.js`. Make sure to remove the content from these files, as we'll write

everything from scratch. Also create a new file `styles.css` in the same directory.

Getting the Icons

To improve the UX, we'll use icons for user choices and the result of the game.

You can pick whatever icons you want from sites like FlatIcon or Icons8, but for simplicity here, I've already collected all the necessary icons for you to use.

You can download them from here.

Unzip the downloaded file and extract the `assets` folder into the `src` folder.

Configuring the Settings

First, we'll define the initial values for the settings of the game.

Open the `src` folder and create a new folder called `configs` inside it. Then create a new file called `game.js` inside it and add the following code:

```
export const settings = {
  gameName: "Rock, paper, scissors!",
  userName: "Player 1",
  pcName: "Computer",
  winMessage: "You won!",
  tieMessage: "It's a tie!",
  lostMessage: "You lost!",
  waitingMessage: "Waiting for your selection!",
  winTarget: 10,
};
```

We've created the `settings` object with all the settings, so all of them are in one place and you have easy access and full control to change them later if you want.

Creating the Base

Next, we need to implement some core functionality to build our game upon.

We'll configure the render file, so the app we'll be developing gets rendered properly into the DOM. Open the `index.js` file in the `src` folder and include the following code:

```
import ReactDOM from "react-dom";

import App from "./App";

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

Also, we'll create a base template for the `App.js` file, which will hold the main logic of our app. Open it and add the following code:

```
import { settings } from "./configs/game";

import rock from "./assets/rock.png";
import paper from "./assets/paper.png";
import scissors from "./assets/scissors.png";
import trophy from "./assets/trophy.png";

import "./styles.css";

export default function App() {
  return (
    <div className="App">
      <p>Rock Paper Scissors Game</p>
    </div>
  );
}
```

We've imported the settings and the assets that we set up in the previous steps. We can now use syntax like `settings.gameName` or `<img src={rock} alt="icon">` to access the settings values and images, respectively.

Then we've set up an `App` function that will be the main wrapper of our app. For now, we've also included a basic placeholder, which we'll remove later.

Finally, let's add some global styling rules that we'll use throughout the whole game. For that, open the `styles.css` file and include the following style rules:

```
@import url("https://fonts.googleapis.com/css2?family=Montserrat:ital@1&display=swap");

* {
  margin: 0;
  padding: 0;
  box-sizing: border-box;
}
```

```
body {
  width: 100vw;
  min-height: 100vh;
  display: grid;
  place-items: center;
  background-color: #2a2a72;
  background-image: linear-gradient(315deg, #2a2a72 0%, #009ffd 74%);
  font-family: "Montserrat", sans-serif;
}

.App {
  width: 1000px;
  min-height: 600px;
  padding: 20px;
  text-align: center;
}

img {
  width: 80px;
  margin: 10px 0;
}
```

We've first imported the Montserrat font from Google Fonts, and applied it in the body so it's used in all the child elements.

Then we've set up some reset rules for the styling, like removing default margin and padding, so we don't have layout inconsistencies across browsers. It's recommended to do that for every app you ever create.

We've set `body` to always stretch across the entire viewport, used grid layout, and centered the main container (which will be the `App` wrapper we created earlier). We've also set the background to be a dark blue gradient.

We've set the `App` wrapper to be a specific width and minimal height for responsive screens. We've also added inside padding and centered all the text inside it.

Finally, we've set all the icons used in our app to use specific width and added a small margin on the top and bottom.

Setting the States

The game will include the user interaction and progress, so there will be multiple state variables to keep track of. We'll use the React `useState` hook, which is a standard way of handing states in React applications.

Open the `App.js` file and add the following code:

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

// other imports from the previous step

export default function App() {
  let [game, setGame] = useState({
    userSelection: "",
    pcSelection: "",
    round: 0,
    userScore: 0,
    pcScore: 0,
    message: "",
  });

  return (
    <div className="App">
      <p>Rock Paper Scissors Game</p>
    </div>
  );
}
```

First, we've imported `useState` at the very top of the file.

After that, we've created a single state object `game` with all the states, so we can easily update and access any of them later.

Let's now look at what each state is for:

- `userSelection` will be updated every time user clicks on the selection (rock, paper, or scissors) by setting the name of the respective choice as a string.
- `pcSelection` will be updated every time the computer has randomly calculated the selection (rock, paper, or scissors) by setting the name of the choice as a string.
- `round` will be incremented by 1 every time both choices have been compared and the winner/tie of the round has been calculated.
- `userScore` will be incremented by 1 each time the user's choice in the individual rounds has been superior to the computer's choice.
- `pcScore` will be incremented by 1 each time the computer's choice in the individual rounds has been superior to the user's choice.

- `message` will hold the information that is being displayed on the screen about the status of each individual round (the user won, the user lost, or it was a tie).

Creating the Components

In this phase, we'll create the individual blocks that we designed earlier in the wireframe.

To keep everything neat and organized, create a new `components` folder in the `src` folder and create separate JS and CSS files for each component we designed in the wireframing phase—except `Profile`, as it will be a pure wrapper component.

You can do that manually, or you can use this time-saving terminal command:

```
mkdir components && cd components && touch Choice.js Choice.css Computer.js Computer.css Message.js Message.css P
```

Next, we'll include the code for each component and set the necessary style rules so the components look great. We'll set the props for each component once imported in `App.js`.

Open `Choice.js` and include the following code:

```
import "./Choice.css";

export const Choice = ({ value, choiceIcon, onClick }) => {
  return (
    <div value={value} onClick={onClick}>
      <img className="choice-icon" src={choiceIcon} alt="icon" />
    </div>
  );
};
```

The `Choice` component will include the `img` that represents the choice as an icon, its `value` as a string, and have an `onClick` prop that will trigger the function when any of the icons are pressed (user has made a selection).

Then open the `Choice.css` file and include the following style rules:

```
.choice-icon {
  border-radius: 50%;
  transition: transform 0.1s;
}

.choice-icon:hover {
```

```
  cursor: pointer;
  background-color: rgb(224, 224, 224);
  transform: scale(1.1);
}
```

This will make sure that there's a rounded border around the icon. Also on hover, the icon will have a grey background, the icon will zoom in a little, and the cursor will change to the pointer.

Open `Computer.js` and include the following code:

```
import { settings } from "../configs/game";
import "./Computer.css";

export const Computer = ({
  pcScore,
  userSelection,
  pcSelection,
  rockIcon,
  paperIcon,
  scissorsIcon,
  trophyIcon,
}) => {
  return (
    <div className="computer-card">
      <h1>Computer</h1>
      {pcScore < settings.winTarget ? (
        userSelection === "" ? (
          <h3 className="waiting-message">{settings.waitingMessage}</h3>
        ) : (
          <>
            <img
              src={
                pcSelection === "Rock"
                  ? rockIcon
                  : pcSelection === "Paper"
                  ? paperIcon
                  : scissorsIcon
              }
              alt="icon"
            />
            <h3>PC selected: {pcSelection}</h3>
          </>
        )
      ) : (
        <>
          <img src={trophyIcon} alt="trophy" />
          <h3>Victory!</h3>
```

```
        </>
      )}
    </div>
  );
};
```

We've first imported the `settings` object we created earlier, so we can access its values.

The component logic is based on the two conditions. If the user hasn't made a choice, the waiting message will be displayed; otherwise the calculated computer choice will be presented as an icon. If the computer has reached the winning threshold, the victory icon will be displayed.

Now open the `Computer.css` file and include the following style rules:

```
.computer-card {
  height: 220px;
  padding: 25px 0;
  background-color: white;
  margin-bottom: 20px;
  border-radius: 20px;
}

.waiting-message {
  margin-top: 20px;
}
```

For the computer-card, we've made sure that it has a specific height, added padding to the top and bottom, set the white background, added some margin to the bottom, and set a rounded border, so it fits well for the overall styling of the game.

For the waiting message, we've added some margin to the top so that it's separated from the player name (computer) at the top of the card.

Open `Message.js` and include the following code:

```
import "./Message.css";

export const Message = ({ userSelection, message }) => {
  return (
    <div className="message-box">
      <h2>{userSelection === "" ? "VS" : message}</h2>
    </div>
  );
};
```

The `Message` component will display the “VS” message on the initial launch of the game, to indicate that the game has not been started yet.

Once the game has been started, it will show the results of each individual round and the final status of the game once one of the players has reached the winning threshold.

Then open the `Message.css` file and include the following style rules:

```
.message-box {
  display: grid;
  place-items: center;
  padding: 20px;
  height: 220px;
}
```

For the message box, we’ve set a grid layout, centering the message in it both horizontally and vertically. We’ve also added some padding to it and set a specific height.

Open `Playground.js` and include the following code:

```
import "./Playground.css";

export const Playground = ({ children }) => {
  return <div className="play-ground">{children}</div>;
};
```

The `Playground` component will be the wrapper component, which will hold both `Profile` components for the user and computer and the `Message` component between them.

Now open the `Playground.css` file and include the following style rules:

```
.play-ground {
  display: grid;
  grid-template-columns: 2fr 1fr 2fr;
  margin-bottom: 30px;
}
```

For the `Playground`, we’ve used grid and set a three-column layout, where the first and third columns (user and computer) are twice as wide as the middle column (message). We’ve also added some margin to the bottom so there’s a space between the reset area directly below it.

Open `Profile.js` and include the following code:

```
export const Profile = ({ children }) => {
  return <div>{children}</div>;
};
```

The `Profile` will be the wrapper component, which will include the `User` and `Computer` components and a separate `Score` component for both of them.

Open `Reset.js` and include the following code:

```
import { settings } from "../configs/game";
import "./Reset.css";

export const Reset = ({ onClick, userSelection, userScore, pcScore }) => {
  return (
    userSelection !== "" && (
      <div onClick={onClick} className="reset-btn">
        <h3>
          {userScore === settings.winTarget || pcScore === settings.winTarget
            ? "Play again"
            : "Reset"}
        </h3>
      </div>
    )
  );
};
```

The `Reset` component will make sure the user can restart the game. It will be displayed only in two states: when the game is in progress, or when the game is finished. If the game is finished, the text of the button will change from **Reset** to **Play Again**.

Now open the `Reset.css` file and include the following style rules:

```
.reset-btn {
  display: grid;
  place-items: center;
  width: 200px;
  height: 40px;
  margin: 0px auto 20px auto;
  background-color: white;
  border-radius: 20px;
  transition: transform 0.1s;
}

.reset-btn:hover {
  cursor: pointer;
```

```
  transform: scale(1.1);
}
```

To add some styling to the `Reset`, we've set it to use the grid layout, centered the included text, set the specific width and height, and added a margin to the bottom. To improve the UX, the button will zoom in on hover as well as change the cursor to a pointer.

Open `Round.js` and include the following code:

```
import "./Round.css";

export const Round = ({ userSelection, round }) => {
  return (
    <h1 className="round">
      {userSelection === "" ? "No rounds yet!" : `Round: ${round}`}
    </h1>
  );
};
```

The `Round` component will display the "No rounds yet" message if the game hasn't been started or the round count if the game is in progress or finished.

Now open the `Round.css` file and include the following style rules:

```
.round {
  color: white;
  margin-bottom: 50px;
}
```

We've set the round text color to be white and added some border to the bottom so there's some space between it and the `PLayground` component directly below it.

Open `Score.js` and include the following code:

```
import "./Score.css";

export const Score = ({ score }) => {
  return <h1 className="score">{score}</h1>;
};
```

The `Score` component will show the number of wins for the user and the computer during the active game in the progress.

Now open the `Score.css` file and include the following style rules:

```
.score {
  color: white;
}
```

We've set the text color of the score to be white.

Open `Title.js` and include the following code:

```
import { settings } from "../configs/game";
import "./Title.css";

export const Title = () => {
  return <h1 className="title">{settings.gameName}</h1>;
};
```

The `Title` component will display the name of the game, which we've imported from the `settings` object from the `game.js` file in the `configs` folder.

Now open the `Title.css` file and include the following style rules:

```
.title {
  margin-bottom: 20px;
}
```

We've added some margin to the bottom so that there's some space between the `Title` and the `Round` component below it.

Open `User.js` and include the following code:

```
import { settings } from "../configs/game";
import "./User.css";

export const User = ({ userScore, userSelection, trophyIcon, children }) => {
  return (
    <div className="user-card">
      <h1>{settings.userName}</h1>
      {userScore < settings.winTarget ? (
        <>
          <div className="choice-grid">{children}</div>
          <h3>
            {userSelection === ""
```

```
              ? "Pick one!"
              : `Your choice: ${userSelection}`}
          </h3>
        </>
      ) : (
        <>
          <img src={trophyIcon} alt="trophy" />
          <h3>Victory!</h3>
        </>
      )}
    </div>
  );
};
```

We've first imported the `settings` object, so we can access its values.

The `User` component logic is based on two conditions. If the user hasn't made a selection, the "Pick one!" message will be displayed, asking for the user to make a choice. If the game is active and the user has made a selection, the selected choice will be displayed as an icon.

Finally, if the user has reached the winning threshold, the victory trophy will be displayed.

Now open the `User.css` file and include the following style rules:

```
.user-card {
  height: 220px;
  padding: 25px 0;
  background-color: white;
  margin-bottom: 20px;
  border-radius: 20px;
}

.choice-grid {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
  place-items: center;
}
```

We've first set the specific height for the card and added some padding to the top and the bottom. We've also set the background to white, and added a border radius and a margin to the bottom.

For the choice icon wrapper, we've set the grid layout and three-column layout so each of the choices is displayed side by side. Finally, we've centered the included icons.

Implementing the Structure

To create the structure of the game, we'll first need to import all the components we created in the previous phase of this tutorial and lay them down in the same order as we designed in the wireframe.

Open the `App.js` file and add the following code:

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

import { Title } from "./components/Title";
import { Round } from "./components/Round";
import { Playground } from "./components/Playground";
import { Profile } from "./components/Profile";
import { User } from "./components/User";
import { Choice } from "./components/Choice";
import { Computer } from "./components/Computer";
import { Score } from "./components/Score";
import { Message } from "./components/Message";
import { Reset } from "./components/Reset";

//import settings, assets, styles...

export default function App() {
  //state object

  return (
    <div className="App">
      <Title />
      <Round {...game} />
      <Playground>
        <Profile>
          <User {...game} trophyIcon={trophy}>
            <Choice {...game} value="Rock" onClick={play} choiceIcon={rock} />
            <Choice {...game} value="Paper" onClick={play} choiceIcon={paper} />
            <Choice
              {...game}
              value="Scissors"
              onClick={play}
              choiceIcon={scissors}
            />
          </User>
          <Score score={userScore} />
        </Profile>
        <Message {...game} />
        <Profile>
          <Computer
```

```
            {...game}
            rockIcon={rock}
            paperIcon={paper}
            scissorsIcon={scissors}
            trophyIcon={trophy}
          />
          <Score score={pcScore} />
        </Profile>
      </Playground>
      <Reset {...game} onClick={reset} />
    </div>
  );
}
```

We've imported all the components at the top and provided all the necessary props we included in the individual components.

Where possible, we've used the object spread syntax (`{...game}`). This means that, if the individual component has some props of the same name as in the `game` object, they'll be picked from the `game` the same way as we would use `componentpropname={game.componentpropname}` . It's especially handy if there are multiple props with the same name.

As you will have noticed, for now, we've also used the non-existent function names (`play` and `reset`) for the `onClick` props. We'll create them in the next phase of the tutorial.

Adding the Functionality

To add the functionality for our game, we need to detect the results of the rounds and update the `game` object accordingly for each case. We'll create separate functions that get executed when the user makes a choice selection or clicks on a reset button.

Add the following code to the `App.js` :

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

//import components, settings, assets, styles

export default function App() {
  //state object

  const reset = () => {
    setGame({
      ...game,
      userSelection: "",
```

```
      pcSelection: "",
      round: 0,
      userScore: 0,
      pcScore: 0,
      message: "",
    });
  };

  const { winMessage, tieMessage, lostMessage, winTarget } = settings;
  const { pcScore, userScore } = game;

  const play = (e) => {
    if (pcScore < winTarget) {
      const userSelection = e.target.parentNode.getAttribute("value");
      const pcSelection = ["Rock", "Paper", "Scissors"][
        Math.floor(Math.random() * 3)
      ];

      userSelection === pcSelection
        ? setGame({
            ...(game.message = tieMessage),
          })
        : (userSelection === "Rock" && pcSelection === "Scissors") ||
          (userSelection === "Paper" && pcSelection === "Rock") ||
          (userSelection === "Scissors" && pcSelection === "Paper")
        ? setGame({
            ...(game.userScore += 1),
            ...(game.message = winMessage),
          })
        : setGame({
            ...(game.pcScore += 1),
            ...(game.message = lostMessage),
          });

      setGame({
        ...game,
        round: (game.round += 1),
        userSelection,
        pcSelection,
      });
    }
  };

  return <div className="App">//components...</div>;
}
```

We've first created the *reset* function, which sets all the state variables of the *game* object to their default values as they were on the initial launch.

Next, we've used some object destructuring to get the necessary values from the `settings` and `game` objects. For example, in the object `const person = {name: "John"}`, you can access the `name` using `const {name} = person`.

Finally, we've created the `play` function, which is executed where the user has selected their choice and but the computer hasn't reached the winning threshold.

The `userSelection` is defined from the value of the `Choice` component. The computer selection is being calculated using JavaScript's built-in `Math.random()` function. First, a random value from 0 to less than 3 is being generated, then floored to the largest integer less than or equal to a given number (for example, 2.66 to 2) and then it gets used as the position index to pick a selection from the given array of the possible choices.

Next, we compare both selections and create a logic based on that.

If the user and computer select the same choice, the individual round is tied and the `message` is changed to display the appropriate text in the `Message` component.

If the user is victorious, the `userScore` is incremented and the `message` is set to display that to the user.

If the user loses the round (the opposite combinations mentioned earlier), the `pcScore` is incremented and the `message` is set to inform the user about that.

Finally, there are some `game` state updates that get executed every time the user makes a choice (whatever the result of the round). Each time the `play` function is run, the `round` number is incremented and `userSelection` and `pcSelection` are set to their respective values, based on both choices in the individual round.

Adding Responsiveness

At this stage, our game looks great and is fully functional, but it lacks responsiveness.

We'll add a couple of media queries to the elements so they adapt to the screens that they're viewed on. Keep in mind that the media rules are normally added at the bottom of each file.

First, open the global `styles.css` file and add the following media rule:

```
@media only screen and (max-width: 1000px) {
```

```
  .App {
    max-width: 100vw;
  }
}
```

This will make sure the `App` wrapper uses all the available viewport width for screens up to the `1000px` width.

Next, switch to the `components` folder, open the `Playground.css` file, and add the following media rule:

```
@media only screen and (max-width: 700px) {
  .play-area {
    grid-template-columns: 1fr;
  }
}
```

This will make sure the `Playground` component will use the one-column layout for the screens up to `700px`, meaning that all the included children components, both `Profile` components for user and computer, and the `Message` component will be shown directly below other.

Finally, open the `Message.css` file and add the following media rule:

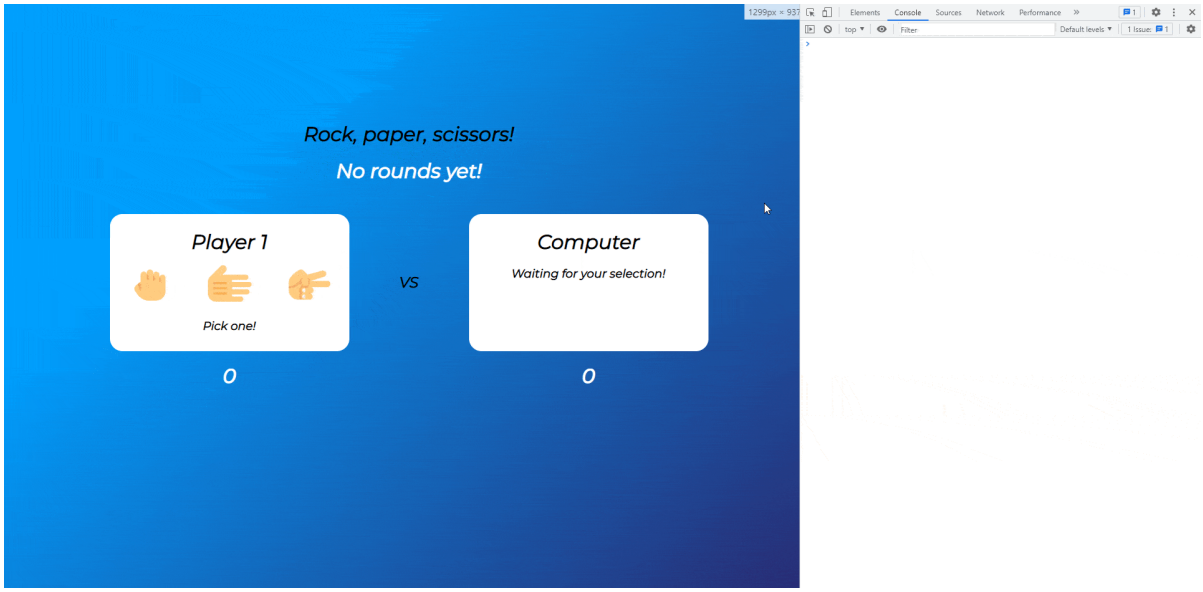
```
@media only screen and (max-width: 700px) {
  .message-box {
    height: auto;
  }
}
```

This will make sure the height of the `Message` component will auto-adjust the content for the screens up to `700px`.

That's all there is to it! If you've followed along, your game should now look great on various screen sizes. The last thing left to do is to test it.

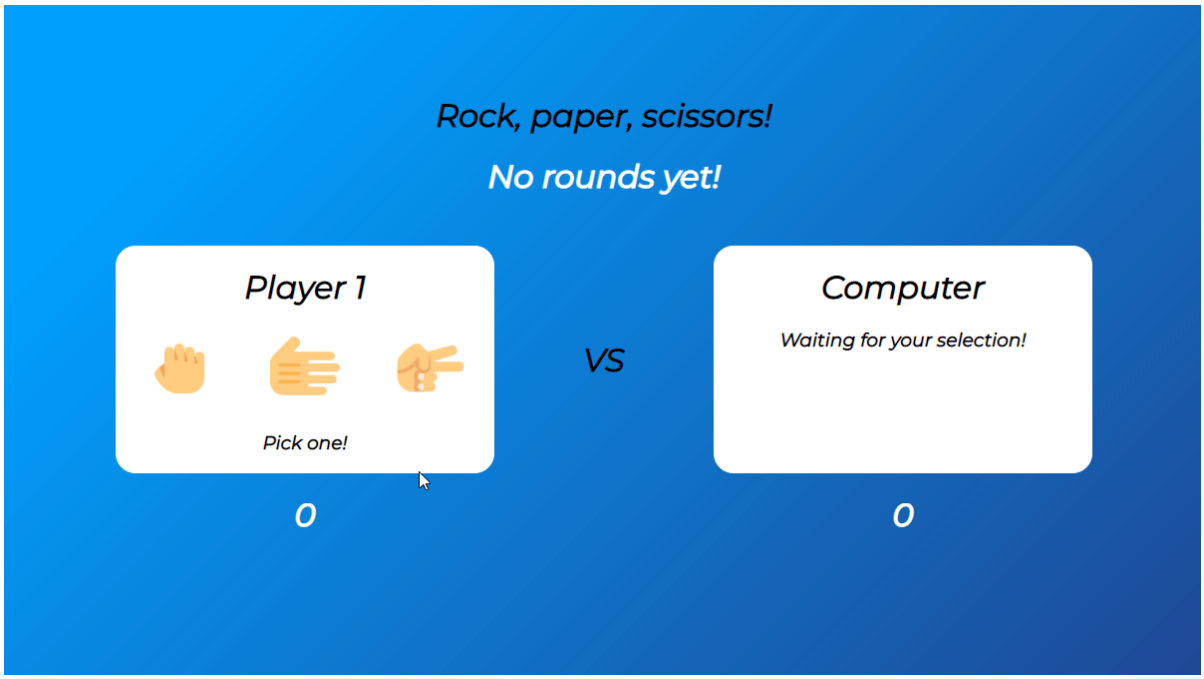
Make sure the development server is still running (if not, run `npm start` in your terminal), open the browser, enter `http://localhost:3000/` in the address bar and launch the developer console by pressing the `F12` key on your keyboard.

Now drag the sidebar to see how the game adjusts the different widths of the screen.



Conclusion

Congratulations! You've now created a fully functional and responsive game! Feel free to play on the desktop or while you're on the go! Share it with your friends and family!



During the building process, we learned the building principles of the game, like how to import the graphics, separate the configuration settings from the main codebase, update the states, and

use conditional rendering accordingly.

There are various ways you could improve the game by adding your own custom features to it. For example, you could create a dialog window where the user can enter the username and configure the settings of the game before playing.

Also, if you don't want to play against the computer, you could make use of WebSockets or socket.io to allow multiple people to connect and play against real people online.

I hope you've learned some practical knowledge that you'll be able to use in your future projects! Thanks for reading, and make sure to give a star on the GitHub repo if you found this useful.