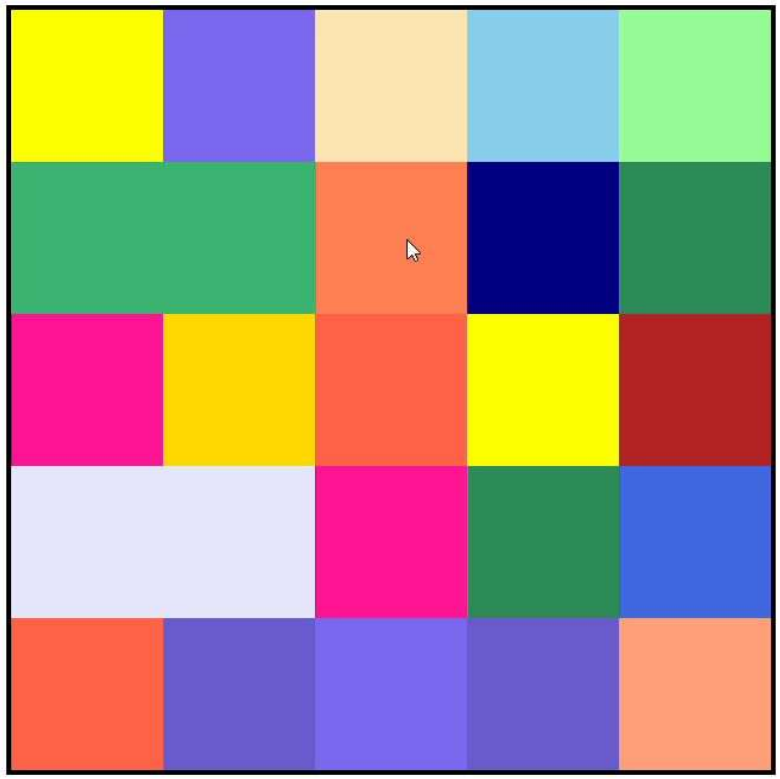
# Exercício ColorBox

# Exercício: Color Box

- Criar um *grid* de cores que são alteradas ao clicar em cada quadrado.



- Quais as formas de resolver esse problema?

# Exercício: Color Box

- O React é baseado em componentes, por isso, iremos resolver utilizando componentes.
- O menor componente que podemos perceber é o quadrado colorido.
- A partir dele, podemos construir o quadrado maior composto por vários quadrados menores.
- Iremos chamar estes componentes de *ColorBox* e *ColorBoxGrid*.

# Exercício: Color Box

```
import { useState } from "react"
import "./ColorBox.css"

export default function ColorBox({colors}){

    function getRandomColor(){
        const randomIndex = Math.floor(Math.random() * colors.length);
        return colors[randomIndex];
    }

    function changeColor(){
        let randomColor = getRandomColor();
        console.log(randomColor)
        setColor(randomColor);
    }

    const [color, setColor] = useState(getRandomColor());

    return <div className="colorbox" style={{backgroundColor: color}}
onClick={changeColor}>

    </div>
}
```

# Exercício: Color Box

- Para não escrever os 25 componentes a mão, podemos fazer um loop.

```
import ColorBox from "./ColorBox"
import "./ColorBoxGrid.css"

export default function ColorBoxGrid({colors}){

    const colorBoxElements = [];

    for(let i=0; i<25; i++){
        colorBoxElements.push(<ColorBox colors={colors} />);
    }

    return <div className="colorboxgrid">
        {colorBoxElements}
    </div>
}
```

# Exercício: Color Box

- CSS

```
.colorboxgrid{
    width: 800px;
    height: 800px;
    border: 5px solid black;
    display: flex;
    flex-wrap: wrap;
}

.colorbox{
    width: 20%;
    height: 20%;
}
```

# Exercício: Color Box

- Main

```
<ColorBoxGrid colors={colors} />
```

```
const colors = [
    "#87CEEB",
    "#FFC0CB",
    "#98FB98",
    "#FFA07A",
    "#FFD700",
    "#E6E6FA",
    "#00FF00",
    "#FF7F50",
    "#000080",
    "#FFE5B4",
    "#FF69B4",
    "#00CED1",
    "#FF6347",
    "#7B68EE",
    "#3CB371",
    "#FF4500",
    "#6A5ACD",
    "#7FFFD4",
    "#B22222",
    "#20B2AA",
    "#FF1493",
    "#4169E1",
    "#2E8B57",
    "#8A2BE2",
    "#FFFF00"
];
```

# AULA 5/6

# States Parte II

# States com Object

- Até agora, estávamos passando variáveis simples para o State do React (números, Strings, etc), não passamos referências.

- Como vimos, o React tenta otimizar o desempenho do App não renderizando a página quando é passado o mesmo valor de parâmetro para o setState().

# States com Object

- Isso pode se tornar um problema quando trabalhamos com objetos, pois a referência ao objeto é sempre a mesma, alteramos somente as propriedades internas deste objeto.

- Neste sentido, é preciso utilizar outra maneira para manusear os estados de um objeto com React.

# States com Object

```jsx
import React, { useState } from 'react';

function UserExample() {
  // Define a state variable 'user' as an object
  let myUser = {
    name: 'John Doe',
    age: 30,
    email: 'john@example.com'
  };

  const [user, setUser] = useState(myUser);

  // Function to update the user object
  const updateUser = () => {
    // Use spread operator (...) to create a copy of the current state object
    // Then update the specific property you want to change

    setUser({
      ...user,
      age: user.age + 1 // Update age to 31
    });


    /*
    //wrong way
    user.age = user.age + 1;
    setUser(user)
    console.log(user.age)
    */
  };

  return (
    <div>
      <h2>User Information</h2>
      <p>Name: {user.name}</p>
      <p>Age: {user.age}</p>
      <p>Email: {user.email}</p>
      <button onClick={updateUser}>Increment Age</button>
    </div>
  );
}

export default UserExample;
```

# States com Object

- { ...user,}
  - O spread operator (...) é usado para criar uma cópia do objeto user. Isso garante que não estamos modificando o objeto original diretamente, mas sim criando uma nova versão dele.
  - O operador ...user expande o objeto user e todas as suas propriedades são copiadas para o novo objeto.

- O resultado final é um novo objeto user com todas as propriedades do objeto original, exceto a propriedade age, que é atualizada para o valor incrementado.

- Este padrão é muito comum em React para atualizar o estado de objetos mantendo a imutabilidade.

# States com Object

```
/* Outra forma de fazer */
setUser((previousUser) => {
    return {...previousUser, age: previousUser.age + 1}
  });
```

# States Parte II
# Arrays

# Emojies

# Emojies

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

export default function Emojies(){

    const [emojies, setEmojies] = useState(["😊"]);

    function addEmoji(){

        const randomIndex = Math.floor(Math.random()*randomEmojies.length);
        const newEmoji = randomEmojies[randomIndex];

        setEmojies(
            (oldEmojies) => [...oldEmojies, newEmoji]
        );
    }

    return (
        <div>
            <span style={{fontSize: "5rem"}}>{emojies}</span>
            <br />
            <button onClick={addEmoji}>Add Emoji</button>
        </div>
    );
}
```

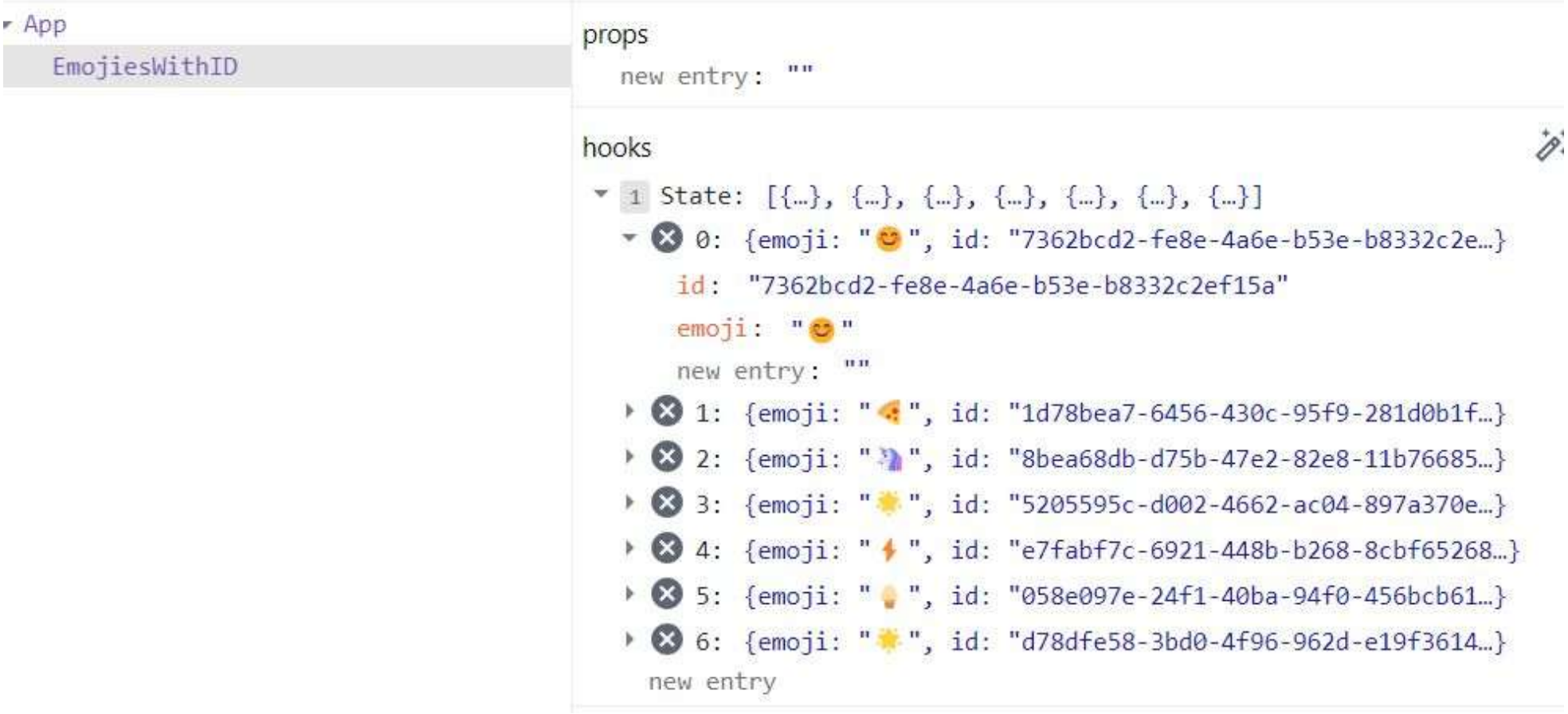
# Emojies

- Agora, vamos alterar nosso exemplo dos Emojies para trabalhar com um Dicionário ao invés de um Array.
- Essa mudança irá nos permitir associar um ID a cada emoji na tela e, com isso, poderemos remover um Emoji ao clicarmos nele.
- Para gerar os IDs, vamos usar a biblioteca UUID.
  - https://www.npmjs.com/package/uuidv4
  - npm install uuidv4

# Emojies com UUID

```
import { useState } from "react";
import {v4 as uuid} from "uuid";

export default function EmojiesWithID(){

    const [emojies, setEmojies] = useState([{"id": uuid(), "emoji": "😊"}]);

    function addEmoji(){

        const randomIndex = Math.floor(Math.random()*randomEmojies.length);
        const newEmoji = randomEmojies[randomIndex];

        setEmojies(
            (oldEmojies) => [...oldEmojies, {"id": uuid(), "emoji": newEmoji}]
        );
    }

    return (
        <div>
            {emojies.map(
                (emoji) => <span key={emoji.id} style={{fontSize: 5rem"}}>{emoji.emoji}</span>
            )}

            <br />
            <button onClick={addEmoji}>Add Emoji</button>
        </div>
    );
}
```

# Emojies com UUID



# Emojies: removendo ao clicar na imagem

- Agora vamos adicionar uma nova funcionalidade, a de remover o Emoji ao clicar em cima da imagem ().

# Emojies: removendo ao clicar na imagem

```
export default function EmojiesWithID(){

    const [emojies, setEmojies] = useState([{"id": uuid(), "emoji": "😊"}]);

    function addEmoji(){
       //CODIGO AQUI
    }

    function removeEmoji(id){
        //Vai alterar o array sem o elemento que foi filtrado
        const newArray = emojies.filter( (e) => e.id !== id )
        setEmojies(newArray);

    }

    return (
        <div>
            {emojies.map(
                (emoji) => <span key={emoji.id} style={{fontSize: "5rem"}} onClick={()
=> removeEmoji(emoji.id) }>{emoji.emoji}</span>
            )}
            <br />
            <button onClick={addEmoji}>Add Emoji</button>
        </div>
    );
}
```

# Emojies: removendo ao clicar na imagem

- Uma outra forma mais resumida de remover o elemento é usando o filter() dentro da arrow function.

```
 function removeEmoji(id){
        /* outra forma  */
 setEmojies(
            (previousEmojies) => {
return previousEmojies.filter( (e) => e.id !== id )}
  );

    }
```

# Emojies: removendo ao clicar na imagem

- Vamos adicionar uma mensagem de confirmação antes de deletar o item...

```
const confirmDelete = window.confirm('Deseja realmente
excluir este emoji?’);

if(confirmDelete){
//Codigo da deleção aqui...
}
....
....
....
```