Estado & Ciclo de vida

THE BRIDGE
DIGITAL TALENT ACCELERATOR

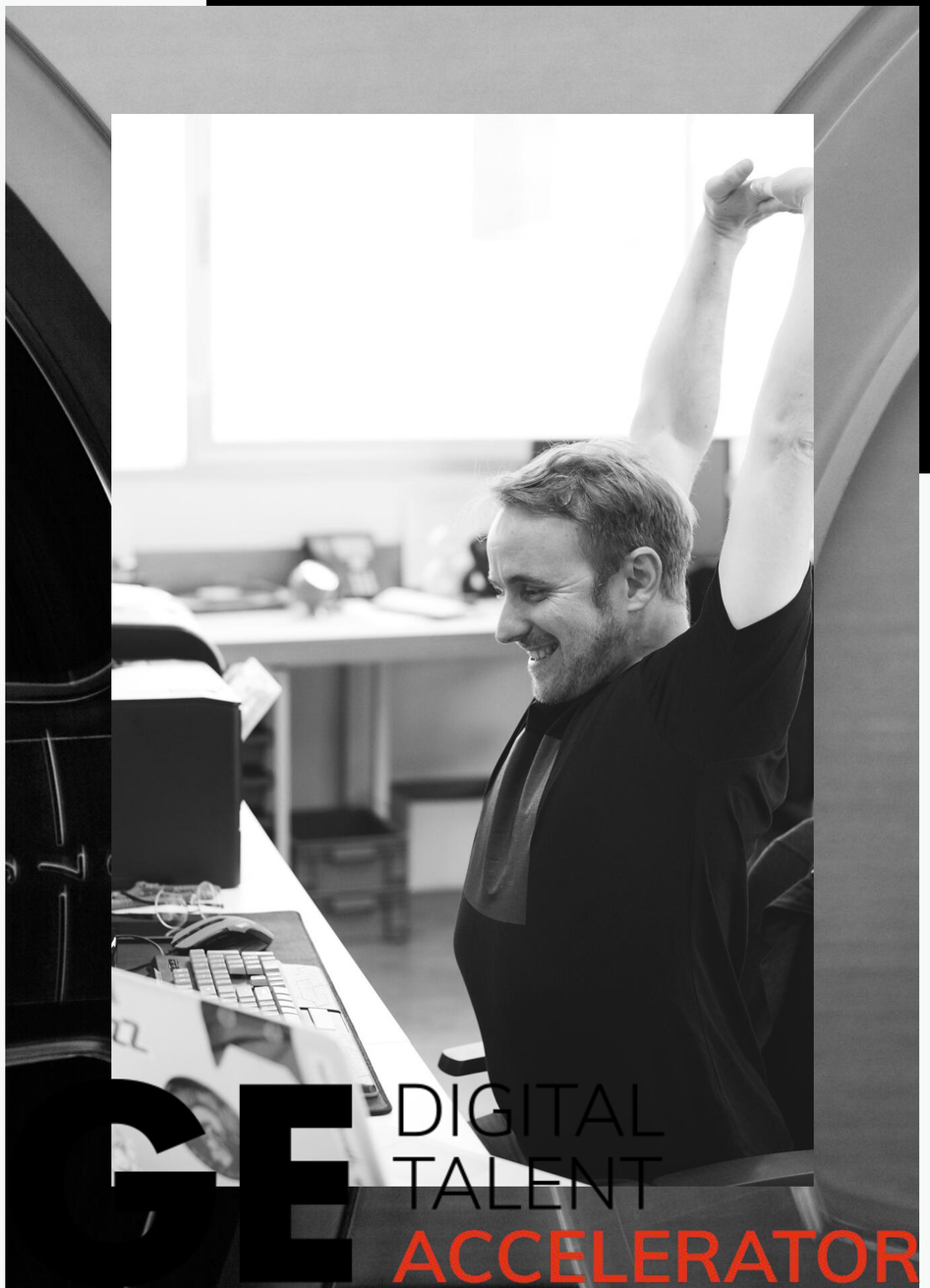
Índice

State con clases

State en componentes funcionales

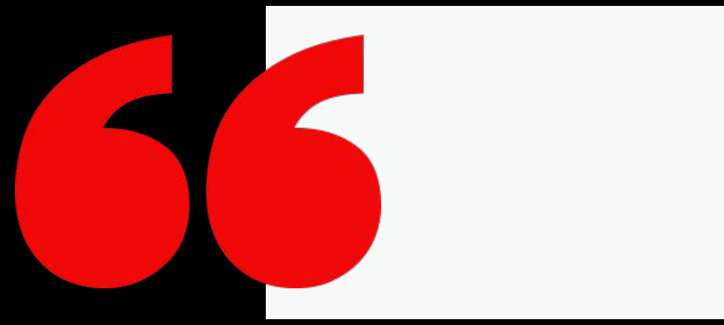
Ciclo de vida con clases

Ciclo de vida con Hooks



THE BRIDGE DIGITAL TALENT ACCELERATOR

State



Al igual que las **props**, el **estado** contiene información sobre el componente. Sin embargo, el tipo de información y **cómo se maneja es diferente**.

Son diferentes debido a una importante razón: **props** se pasa al componente (similar a los parámetros de una función) mientras que **state se administra dentro del componente** (similar a las variables declaradas dentro de una función).

Componente sin estado

Por defecto, un componente **no tiene** estado como en el ejemplo del componente Welcome:

```
import React from 'react';

class Welcome extends React.Component {
  render() {
      return <h1>Hello {this.props.name}</h1>;
  }
}
```

Componente sin estado

Creamos un componente **Counter** que por defecto **no tiene** un estado:

```
import React, { Component } from 'react'

class Counter extends Component {
  render() {
    return (
      <div>
        <p>Counter: {this.props.counter}</p>
      </div>
    )
  }
}

export default Counter
```

Componente Counter

Para cambiar el contador nos veremos **tentad@s a hacer lo siguiente:**

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

class Counter extends Component {

  increment =()=> {
    this.props.counter + 1;
  }

  render() {
    return (
      <div>
        <p>Counter: {this.props.counter}</p>
        <button onClick={this.increment}>+</button>
      </div>
    );
  }
}

export default Counter;
```

Entonces, ¿cuándo usar el estado?

Cuando un componente **necesita** realizar un seguimiento de la información para que pueda **crear, actualizar y usar el estado**.

El corazón de cada componente de React es su **"state"**, un objeto que determina cómo se renderiza y se comporta ese componente. En otras palabras, el **"state"** es lo que **permite crear componentes** que son **dinámicos e interactivos**.

Componente Counter

Creamos el componente counter y definimos el **estado** del contador:

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

export default class Counter extends Component {
  constructor() {
    super();
    this.state = {
      counter: 0,
    };
  }
  render() {
    return <button>{this.state.counter}</button>;
  }
}
```

Definimos el estado

Usamos el estado

Cambiando el estado

Estaremos **tentados** a hacer lo siguiente:

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

export default class Counter extends Component {
  constructor() {
    super();
    this.state = {
      counter: 0,
    };
  }
  increment = () => {
    this.state.counter + 1;
    console.log(this.state.counter);
  };
  render() {
    return <button onClick={this.increment}>{this.state.counter}</button>
  }
}
```

Definimos nuestra función increment

La llamamos aqui

Cambiando el estado

La forma correcta es setear el estado usando **setState**:

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

export default class Counter extends Component {
  constructor() {
    super();
    this.state = {
      counter: 0,
    };
  }
  increment = () => {
    this.setState({ counter: this.state.counter + 1 });
  };
  render() {
    return <button onClick={this.increment}>{this.state.counter}</button>
  }
}
```

Definiendo el estado por props

Podemos pasarle por **props** el **valor inicial** del contador:

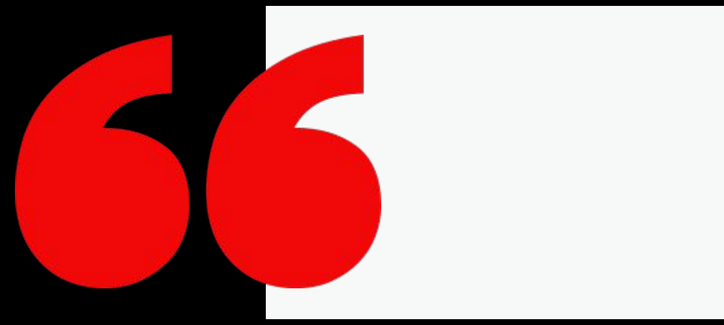
```
function App() {
  return (
    <div className="App">
      <Counter initialValue={0}/>
      <Counter initialValue={2}/>
    </div>
  );
}
```

Definiendo el estado por props

Le decimos que el **estado del contador** es el que nos viene por **props**:

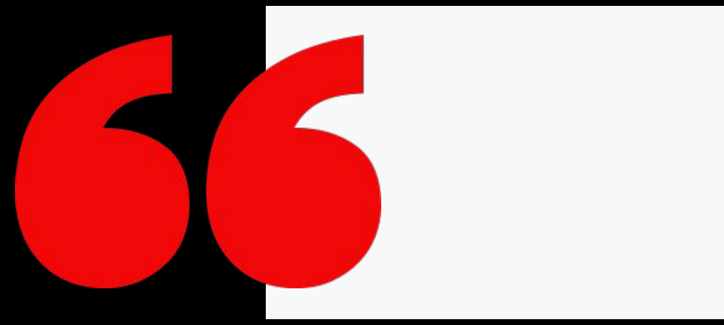
```
import React, { Component } from "react";

export default class Counter extends Component {
  constructor(props) {
    super(props);
    this.state = {
      counter: this.props.initialValue,
    };
  }
  increment = () => {
    this.setState({ counter: this.state.counter + 1 });
  };
  render() {
    return <span onClick={this.increment}>{this.state.counter}</span>;
  }
}
```

Utilizamos las props



¿Qué son los Hooks?

Son una nueva incorporación en React 16.8. Permiten usar el estado y otras características de React sin escribir una clase.



useState

El hook de estado (**State**) permite añadir estados en el **componente funcional**. En lugar de establecer un **estado inicial** con la declaración de estado en el **constructor**, podemos importar { useState } de React. Esto nos permitirá establecer el estado inicial como un **argumento**.

React Hooks

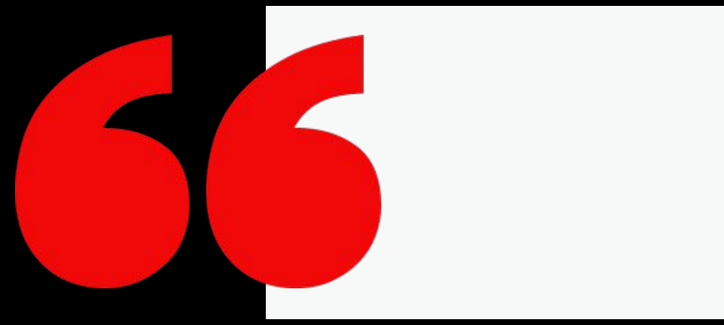
Aquí podemos ver un componente **Counter** usando hooks.

importamos useState

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

const Counter = (props) => {
  const [counter, setCounter] = useState(props.initialValue);
  const increment = () => {
    setCounter(counter + 1);
  };
  return <button onClick={increment}>{counter}</button>;
};

export default Counter;
```

Inicializamos el estado

Ciclos de vida



Cada componente en React tiene un **ciclo de vida** que podemos monitorear y manipular durante sus **tres fases principales**.

Las tres fases son: **Montaje** del componente, **Actualización** del componente y **Desmontaje** del mismo.

Montando el componente de clase

Montar significa poner elementos en el DOM. React tiene varios métodos integrados que se llaman al montar un componente (en este orden). Estos son los más comunes:

- **constructor()**
- **render()**
- **componentDidMount()**

El método **render()** es obligatorio y siempre se llamará, los demás son opcionales y se llamarán si se definen.

componentDidMount

Se llama al método **componentDidMount()** después de renderizar el componente. Aquí es donde ejecuta declaraciones que requieren que el componente ya esté colocado en el DOM.

componentDidMount

Al principio, mi color favorito es el rojo. Después de un segundo, será verde:

```
import React from "react";

export default class Example extends React.Component {
  constructor() {
    super();
    this.state = { favoriteColor: "red" };
  }

  componentDidMount() {
    setTimeout(() => {
      this.setState({ favoriteColor: "green" });
    }, 1000);
  }

  render() {
    return <h1>My Favorite Color is {this.state.favoriteColor}</h1>;
  }
}
```

Actualizando el componente

La **siguiente fase** del ciclo de vida es cuando se **actualiza un componente**. Un componente **se actualiza** cada vez que hay un **cambio en el estado** o las **propiedades** del componente.

React tiene métodos que se llaman, **en este orden**, cuando se actualiza un componente:

- **render()**
- **componentDidUpdate()**

componentDidUpdate()

Se llama al método **componentDidUpdate** después de actualizar el componente en el DOM. El siguiente ejemplo puede parecer complicado, pero todo lo que hace es esto:

- Cuando el componente **se está montando**, se **renderiza** con el color favorito "rojo".
- Cuando **se ha montado** el componente, un temporizador **cambia el estado** y el color se vuelve "verde".
- Esta acción **desencadena la fase de actualización** y, dado que este componente tiene un método **componentDidUpdate**, este método **se ejecuta** y escribe por consola un mensaje.

componentDidUpdate

Lo podemos ver en el siguiente ejemplo:

```
import React from "react";

export default class Example extends React.Component {
  constructor() {
    super();
    this.state = { favoriteColor: "red" };
  }
  componentDidMount() {
    setTimeout(() => {
      this.setState({ favoriteColor: "green" });
    }, 1000);
  }
  componentDidUpdate() {
    console.log("el componente se ha actualizado");
  }
  render() {
    return (
      <div>
        <h1>My Favorite Color is {this.state.favoriteColor}</h1>
      </div>
    );
  }
}
```

componentWillUnmount()

La siguiente fase en el ciclo de vida es cuando **se elimina un componente del DOM**, o se **desmonta**.

React tiene solo un método incorporado que se llama cuando se desmonta un componente:

- **componentWillUnmount()**

componentWillUnmount()

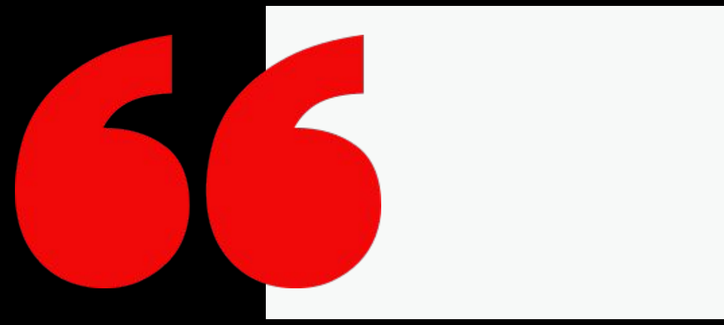
Tenemos el siguiente componente:

```
export default class example extends Component {
 constructor() {
   super()
   this.state = { favoriteColor: 'red', show: true   }
 }
 componentDidMount() {
   setTimeout(() => {
     this.setState({ favoriteColor: 'green' })
   }, 1000)
 }
 componentDidUpdate() {
   console.log('el componente se ha actualizado')
 }
 handleRemove = () => {
   console.log(this.state.show)
   this.setState({ show: false })
 }
 render() {
   return (
     <div>
       <h1>My Favorite Color is {this.state.favoriteColor}</h1>
       {this.state.show ? <ExampleChild /> : null}
       <button onClick={this.handleRemove}>
         Eliminar el componete ExampleChild
       </button>
     </div>
   )
 }
}
```

componentWillUnmount()

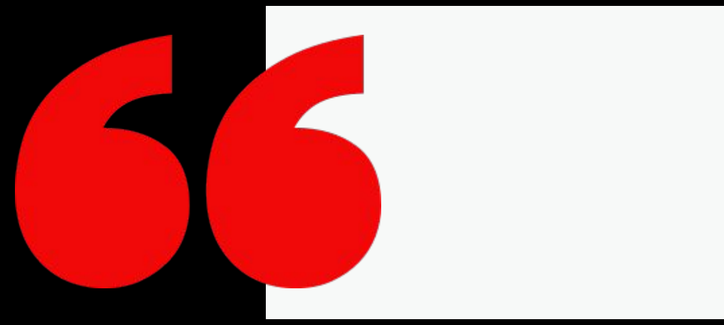
Este método se llama exactamente momentos antes de que el componente se elimine del renderizado.

```
class ExampleChild extends React.Component {
    componentWillUnmount() {
        alert("El componente ExampleChild está a punto de ser eliminado (desmontado).");
    }
    render() {
        return <h1>Hola The Bridge!</h1>;
    }
}
```



Ciclo de vida

en Componentes funcionales

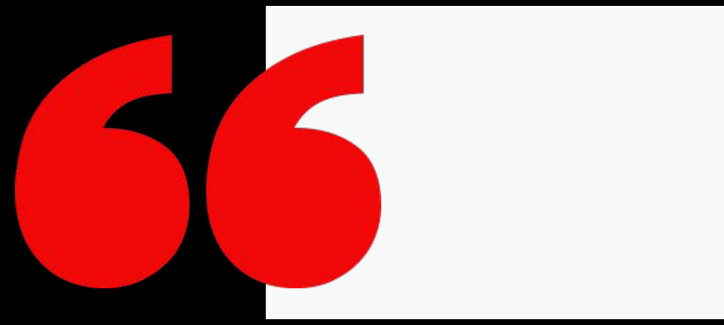


componentDidMount a hooks

useEffect

Si el **segundo argumento** es una **array vacía**, se comportará exactamente como el **componenteDidMount**, solo ejecutándose en la primera representación.

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

const Example = () => {
  const [favoriteColor, setFavoriteColor] = useState('red');
  useEffect(() => {
    setTimeout(() => {
      setFavoriteColor("green");
    }, 1000);
  }, []);
  return <h1>My Favorite Color is {favoriteColor}</h1>;
};

export default Example;
```

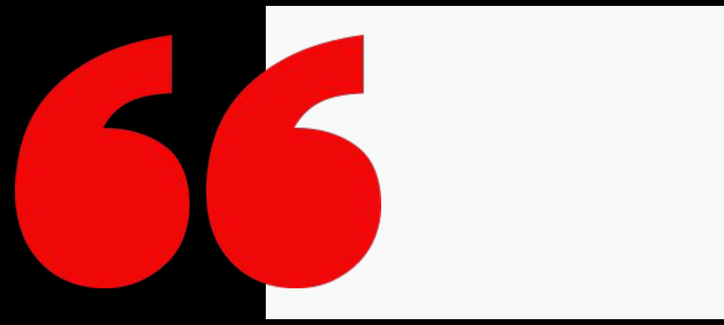


componentDidUpdate a hooks

useEffect

- El equivalente de **componentDidUpdate** en hooks es el hook useEffect **sin** el segundo argumento.
- Las funciones pasadas al enlace useEffect se ejecutan en cada representación del componente, a menos que le pasemos un **segundo argumento** como una **array de dependencias**.

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

const Example = () => {
  const [favoriteColor, setFavoriteColor] = useState('red');
  useEffect(() => {
    setTimeout(() => {
      setFavoriteColor("yellow");
    }, 1000);
  });
  useEffect(()=>{
    console.log("el componente se ha actualizado");
  })
  return <h1>My Favorite Color is {favoriteColor}</h1>;
};
export default Example;
```



componentWillUnmount a

hooks

useEffect

Tenemos el siguiente componente:

```
const Example = () => {
  const [show, setShow] = useState(true);

  const handleRemove = () => setShow(false);

  return (
    <div>
      {show ? <ExampleChild /> : null}
      <button onClick={handleRemove}>Eliminar el componete ExampleChild</button>
    </div>
  );
};
export default Example;
```

useEffect

Es posible que tengas algún código que deba ejecutarse cuando el componente se elimina del árbol DOM. Con el hook **useEffect**, puedes devolver una función de la función pasada al hook que se ejecutará cuando se desmonte el componente.

```
const ExampleChild = () => {
  useEffect(() => {
    return () => {
      alert("El componente ExampleChild ha sido eliminado(desmontado).");
    };
  });
  return <h1>Hola The Bridge!</h1>;
};
```