

Xseed

Curso de React _ Technisys

Clase 02 -

Tipos de Componentes, más sobre Props, manejo de State, keyword this

by Diego Cáceres

Repaso

Un componente en React

En React generalmente se utiliza una sintaxis llamada **JSX** (de la que se hablará posteriormente). Los componentes en React deben tener sí o sí una función `render`, que es donde se define cómo se dibuja el componente en la UI.

```
class HelloMessage extends React.Component {  
  render() {  
    return <div>Hello {this.props.name}</div>;  
  }  
}  
  
ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

JSX le llamamos a poder usar tags de HTML en el código Javascript.

JSX

```
var element = <h1>Hello, world!</h1>;
```

Esto no es HTML ni un string. Es JSX, que es una **extensión a la sintaxis de JavaScript**. Se utilizará en React para definir cómo se debe ver la UI de un componente.

JSX produce elementos de React, que en realidad son simplemente objetos en JavaScript!

```
var element = <h1>Hello, world!</h1>;
```

Se traduce en

```
var element = React.createElement(  
  "h1",  
  null,  
  "Hello, world!"  
);
```

Pueden probar traducir cualquier código JSX a JavaScript normal online en [Babel](#).

Atributos en JSX

A los tags de JSX se les puede especificar atributos, al igual que sus tags equivalentes en HTML. La diferencia es que en JSX se escriben utilizando nomenclatura camelCase:

- `tabindex` se convierte en `tabIndex`
- `onclick` se convierte en `onClick`

```
const element = <div tabIndex="0"></div>;
```

Un caso a tener en cuenta: En Javascript `class` es una palabra reservada, por lo que para proporcionar una clase, en JSX usamos `className`.

```
const element = (  
  <h1 className="greeting">  
    Hello, world!  
  </h1>  
>;
```

La clase `greeting` está definida en otro archivo CSS. Por ejemplo:

```
.greeting {  
  color: "red"  
}
```

Componentes y Props (1)

Los componentes permiten separar la UI en pedazos chicos, independientes y reutilizables.

Conceptualmente, los componentes son como funciones JavaScript; reciben parámetros (props) y retornan lo que se debe mostrar en pantalla (en JSX).

```
function WelcomeMessage(props) {  
  return <h1>Hello, {props.name}</h1>;  
}
```

Esta función es un componente válido en React, y es llamado “functional component”

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

Esto es llamado “class component”, y está utilizando las clases de Javascript introducidas en ES6.

Estos dos componentes son equivalentes desde el punto de vista de React, aunque el class component tiene algunas características diferentes que veremos más adelante.

Componentes y Props (2)

Hasta ahora se vieron elementos de React que utilizan sólo tags de HTML, pero se pueden crear elementos a partir de los Componentes que nosotros mismos definimos.

```
// Definimos el Componente
function WelcomeMessage(props) {
  return <h1>Hello, {props.name}</h1>;
}

// Utilizamos el Componente
var element = <WelcomeMessage name="Anne" />;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

Cuando React encuentra un elemento definido por nosotros, le pasa los atributos definidos en JSX al componente como **"props"**

Tipos de componentes en React

Tipos de componentes

```
import React, { Component } from 'react';
import ReactDOM from 'react-dom';

class HelloWorld extends React.Component {
  render () {
    return (
      <div>
        Hello {this.props.name}
      </div>
    )
  }
}

ReactDOM.render(<HelloWorld name='Diego' />,
  document.getElementById('app'))
```

Class Component

```
import React from 'react';
import ReactDOM from 'react-dom';

function HelloWorld (props) {
  return (
    <div>Hello {props.name}</div>
  );
}

ReactDOM.render(<HelloWorld name='Diego' />,
  document.getElementById('app'))
```

Functional Component (Stateless Component)

Class Component

Al declarar un **Class Component** se está utilizando la `class` incorporada en ES6 a JavaScript. Siempre es necesario que tenga al menos un método `render`, pero puede tener otros.

Una de las características que tienen estos componentes es que permiten utilizar los **Lifecycle Methods** de React, que son métodos de ciclo de vida de los componentes (se verá en detalle más adelante).

Además, también se puede manejar el **State** (estado) en estos componentes.

Class Component – Ejemplo

```
class HelloWorld extends React.Component {  
  componentDidMount() {  
    callSomeApiToGetInfo();  
  }  
  someHelperMethod(name, lastName) {  
    return `${name} ${lastName}`;  
  }  
  render () {  
    return (  
      <div>  
        Hello {this.someHelperMethod(this.props.name, this.props.lastName)}  
      </div>  
    )  
  }  
}
```

Functional Component

Como el nombre lo dice, estos son simples funciones que pueden recibir parámetros o no (comúnmente llamados *props*), y retornan “UI”. Como sólo retornan UI también suelen ser llamados **componentes de presentación**.

Estos componentes son más simples, pero no tienen la posibilidad de tener **state**, ni métodos. Es una buena práctica utilizar este tipo de componentes cuando sea posible.

Todos los componentes propios debemos nombrarlos con mayúscula, al igual que con los Class Components, de forma que React no piense que son tags html.

Functional Component – Ejemplo

```
function HelloWorld (props) {  
  return (  
    <div>Hello {props.name}</div>  
  );  
}  
  
// Tambien podemos usar arrow functions (ES6)  
const HelloWorld = (props) => {  
  return (  
    <div>Hello {props.name}</div>  
  );  
}
```

Más sobre las *props*

Props son Read-Only

No importa si se trabaja con **Class Component** o **Functional Component**, en cualquiera de los casos los valores que se reciben en *props* no se pueden modificar, son **sólo de lectura**.

“All React components must act like pure functions with respect to their props.”

Las **funciones puras** son un concepto relacionado con la programación funcional. Una función es pura cuando no modifica sus *inputs*, y devuelve siempre el mismo resultado para el mismo *input*.

Funciones puras e impuras

```
// Función Pura
```

```
function sum(a, b) {  
    return a + b;  
}
```

```
// Función Impura
```

```
function withdraw(account, amount) {  
    account.total -= amount;  
}
```

¿Cómo se modifica un componente?

Como las *props* son *Read-Only*, hasta ahora es perfecto para construir sitios estáticos, pero claramente no siempre es el caso deseado.

La forma para que los componentes y sitios desarrollados con React sean más interesantes, y que se conviertan en sitios dinámicos, que cambian su información a lo largo del tiempo, es utilizando un nuevo concepto de React llamado ***state*** (estado).

State – Estado

El **state** permite que los componentes de React modifiquen su *output* a lo largo del tiempo en respuesta a acciones del usuario, a respuestas de la red, y cualquier otro evento, sin violar la regla de que tienen que actuar como funciones puras con respecto a sus *props*.

Es un simple **objeto JavaScript** que contiene las variables necesarias para el componente, por lo tanto a partir del **state** y las *props*, el componente determina qué UI (User Interface) renderizar.

Manejo del State

Manejo del State (1)

El *state* es similar a las *props*, pero es **privado** para el componente y **controlado** completamente por el componente.

No se puede acceder al *state* de un componente desde otro; y si esto se logra mediante algún truco de JavaScript, igualmente va en contra de los conceptos de React.

Como se mencionó anteriormente, el estado local es una característica que sólo tienen los componentes de tipo **Class**. Es una práctica común crear un *Functional Component* y que más adelante haya que convertirlo en un *Class Component* por necesidad de manejar estado.

Manejo del State (2)

La forma de declarar el *state* inicial en un componente es mediante el **constructor**, que no es propio de React sino de la `class` de JavaScript introducida en ES6. Es importante notar que este es en el único lugar donde se va a asignar el *state* con un simple **=**

```
class MyComponent extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {  
      description: 'Soy un ejemplo'  
    };  
  }  
  render() { <div> Hello World! </div> }  
}
```

El state es un Objeto que contendrá todas las propiedades que se quieran manejar localmente dentro del componente.

Manejo del State (3)

Para acceder al valor de *state* se puede hacer a través de la keyword `this`

```
class MyComponent extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {  
      description: 'Soy un ejemplo'  
    };  
  }  
  render() {  
    <div>  
      <div> Hello World! </div>  
      <p> Mi descripcion: {this.state.description} </p>  
    </div>  
  }  
}
```

Manejo del State (4)

Como se mencionó anteriormente, no es posible actualizar el *state* simplemente utilizando la asignación con `=`.

```
// Incorrecto!
```

```
this.state.comment = 'Hello';
```

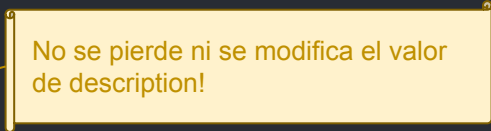


El motivo de esto, es que si se hace de esta forma React no va a volver a renderizar el Componente, por lo que no se verá el cambio en la UI. Una de las ventajas de React es que es eficiente a la hora de actualizar el DOM (lo que el usuario ve en la pantalla). Por eso es importante que si el estado interno de un componente cambia, esto se refleje correctamente en pantalla al usuario.

Manejo del State (5)

La forma correcta es utilizar la función **setState** que recibe un objeto, con el cual actualiza el estado del componente. Sólo actualiza las propiedades que contiene el nuevo objeto, por lo que no es necesario que contenga las propiedades que no se modificaron.

```
constructor(props) {  
  super(props);  
  this.state = {  
    description: "Soy un ejemplo",  
    comment: "Comentario inicial"  
  };  
}  
  
// En algún lugar del componente:  
this.setState({  
  comment: "Comentario actualizado"  
})
```



No se pierde ni se modifica el valor de description!

setState es asíncrono (1)

React puede juntar varias llamadas al `setState` de forma de ejecutarlas juntas para optimizar, lo que quiere decir que al llamar a `setState` no es garantizado que se ejecute de inmediato.

Por eso es que no hay que basarse en `this.props` o `this.state` para calcular el siguiente estado.

```
// Esto puede fallar, hay que evitarlo.  
this.setState({  
  counter: this.state.counter + this.props.increment,  
});
```



setState es asíncrono (2)

Para este caso, se puede usar una segunda forma de invocar a `setState` que recibe una función en vez de un objeto. Esta función recibirá el estado anterior como primer parámetro, y las *props* al momento de ejecutarse la actualización como segundo parámetro. La función será invocada automáticamente por React quien proveerá los parámetros correctos.

```
// Esto garantiza que tendrá los valores correctos
this.setState(function(prevState, props) {
  return {
    counter: prevState.counter + props.increment
  }
});
```

Inmutabilidad en React (1)

Hay normalmente dos formas de modificar la información. La primera, consiste en **mutarla**, modificando directamente el valor de una variable. La segunda consiste en **reemplazar** los datos con una nueva copia del objeto que también incluye los cambios deseados.

```
var player = {score: 1, name: 'Jeff'};  
player.score = 2;  
// Ahora player es {score: 2, name: 'Jeff'}
```

Mutando los datos

```
var player = {score: 1, name: 'Jeff'};  
var newPlayer = Object.assign({}, player, {score: 2});  
// Ahora player no cambió, pero newPlayer es {score: 2, name: 'Jeff'}
```

Sin mutar los datos

Inmutabilidad en React (2)

Seguimiento de cambios

Determinar si un objeto mutado ha cambiado es complejo porque los cambios se realizan directamente en el objeto. Esto requiere comparar el objeto actual con una copia anterior, recorrer todo el árbol de propiedades y comparar cada propiedad y valor. Este proceso puede llegar a ser cada vez más complejo.

Determinar cómo ha cambiado un objeto inmutable es considerablemente más fácil. Si el objeto al que se hace referencia es diferente al de antes, entonces el objeto ha cambiado.

Inmutabilidad en React (3)

Determinar cuándo volver a renderizar en React

El mayor beneficio de la inmutabilidad en React viene cuando se construyen componentes puros simples. Esto se debe a que los datos inmutables pueden determinar con más facilidad si se han realizado cambios, también ayuda a determinar cuándo un componente necesita ser vuelto a renderizar.

Keyword `this` en JavaScript

Keyword `this`

Es importante entender este concepto ya que en JavaScript el valor de `this` depende del contexto, lo cual puede generar problemas al utilizarla.

Si se tiene la siguiente función, ¿cuál es el valor de `name`?

```
var sayName = function(name) {  
    console.log('Hello, ' + name);  
}
```

No se sabe hasta que
alguien la invoca.

```
sayName('Diego');
```

Algo similar ocurre con `this`, pero hay ciertas reglas que ayudan:

- Implicit Binding
- Explicit Binding
- new Binding
- window Binding

Keyword `this` – Implicit Binding

Esta regla consiste simplemente en mirar a la izquierda del punto al momento de la invocación. Este es el caso más común.

```
var me = {  
  name: 'Diego',  
  age: 26,  
  sayName: function() {  
    console.log(this.name);  
  }  
};  
  
me.sayName()
```

Se mira a la izquierda del "." cuando se invoca y se ve 'me', entonces es me.name => Diego

[JSBin - Ejemplo 1](#)

[JSBin - Ejemplo 2](#)

Keyword `this` – Explicit Binding

Con explicit binding, lo que hacemos es especificar con que contexto queremos que una funcion o metodo se ejecute. `bind` devuelve una nueva función con el contexto aplicado.

```
var sayName = function(food1, food2) {  
    console.log("My name is " + this.name + " and I like " + food1 + " and " + food2);  
};  
  
var tom = {  
    name: "Tom",  
    age: 33  
};  
  
var foods = ["pizza", "chocolate"];  
var newFn = sayName.bind(tom, foods[0], foods[1]);  
console.log("Antes de invocar");  
newFn();
```

State y uso del bind

En React es común que se utilice el `bind` para que las funciones definidas dentro de un Class Component, siempre estén 'bindeadas' al mismo contexto, es decir, al contexto del componente.

```
class InputExample extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = { text: "" };  
    this.change = this.change.bind(this);  
  }  
  change(ev) {  
    this.setState({ text: ev.target.value });  
  }  
  render() {  
    return <input type="text" value={this.state.text} onChange={this.change} />;  
  }  
}
```

Si no se "bindea" `change`, al utilizar `this.setState` no estaría definido

Ejercicios

Herramienta extra

En React existen las Developer Tools que se pueden instalar como una extensión de [Chrome](#) o [Firefox](#) (Buscar React Developer Tools) y son muy útiles ya que permiten inspeccionar la jerarquía de componentes:

```
▼<Game>
  ▼<div className="game">
    ▼<div className="game-board">
      ▼<Board>
        ▼<div>
          <div className="status">Next player: X</div>
          ▼<div className="board-row">
            ▶<Square index=0>...</Square>
            ▶<Square index=1>...</Square>
            ▶<Square index=2>...</Square>
          </div>
          ▼<div className="board-row">
            ▶<Square index=3>...</Square>
            ▶<Square index=4>...</Square>
            ▶<Square index=5>...</Square>
          </div>
          ▼<div className="board-row">
            ▶<Square index=6>...</Square> == $r
            ▶<Square index=7>...</Square>
            ▶<Square index=8>...</Square>
          </div>
        </div>
      </Board>
    </div>
    ▼<div className="game-info">
      <div />
      <ol />
    </div>
  </div>
</Game>
```

Ejercicios

Descargar el **zip** EjerciciosClase02. Cada carpeta dentro corresponde a un ejercicio, y es un proyecto creado con Create React App, por lo que para ejecutarlo es necesario:

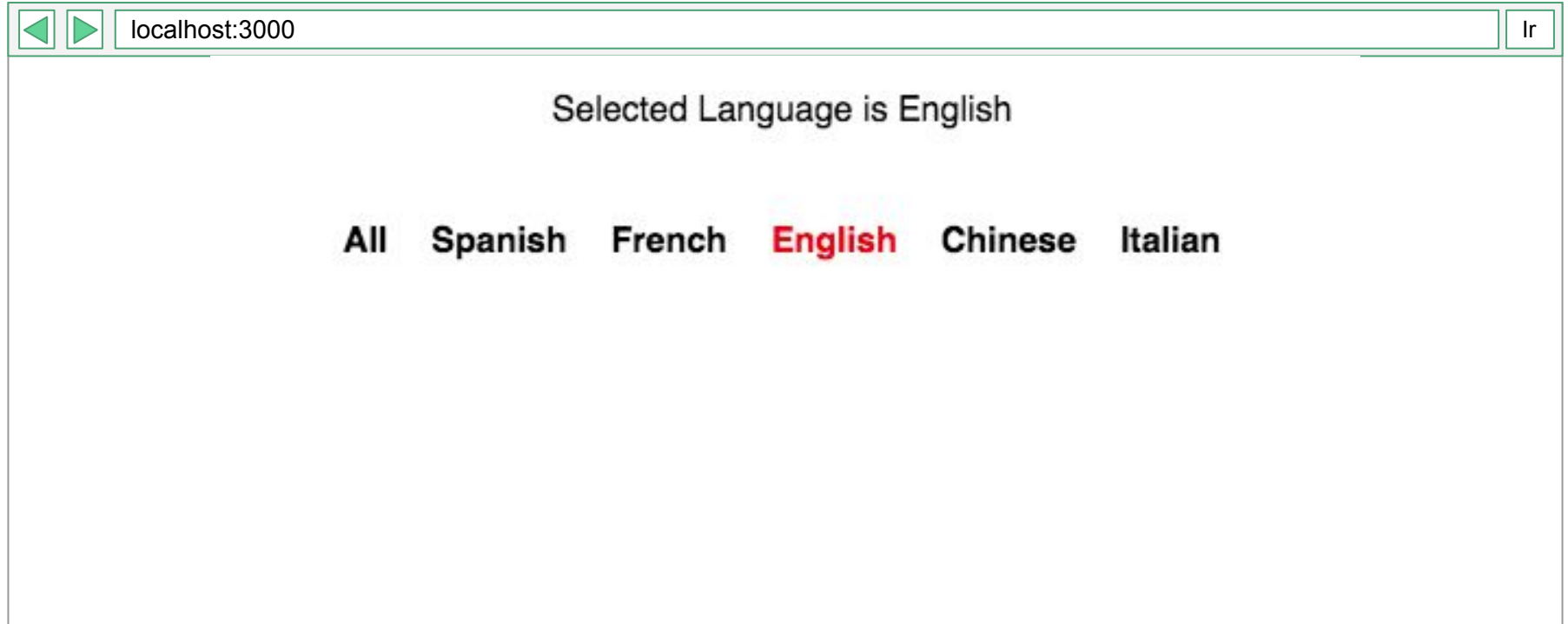
- Si usan yarn:
 - `$ yarn install`
 - `$ yarn start`
- Si usan npm:
 - `$ npm install`
 - `$ npm start`

Se debería abrir automáticamente el explorador con el ejercicio corriendo en <http://localhost:3000/>.

Ejercicio 1

- Editar el archivo `src/App.js` para implementar un componente header, que permita cambiar el lenguaje seleccionado.
 - a. Para esto van a tener que definir un estado inicial.
 - b. Bindear el método `updateLanguage` para que tenga acceso a `this.setState`.
 - c. Implementar `updateLanguage` para actualizar el estado.
 - d. Modificar en el `render` para que muestre el lenguaje seleccionado.

Ejercicio 1 (cont)



Ejercicio 2

1. Editar el archivo `src/App.js`. Se necesita que `Counter` deje de ser un functional component, para que tenga estado y sea quien maneje el valor a mostrar, para que funcione como un contador cuando se presione el botón y aumente su propio valor.
 - a. Primero se debe definir `Counter` como una subclase de `Component`, con su método `render`, y cambiar `props` por `this.props`, y verificar que sigue funcionando.
 - b. Definir un constructor con el estado inicial, determinando qué estado necesita manejar `Counter`.
 - c. Dejar de mostrar el valor recibido por `props`, y cambiarlo por el del estado local. También se puede cambiar en `App` cuando utilizamos `Counter` a: `<Counter />`.
 - d. Implementar un método para actualizar el valor del contador con el `onClick` del botón. No olvidarse del `bind` y recordar que el `setState` es asíncrono...

Ejercicio 3 – TicTacToe (Realizado en conjunto)

- Revisar el archivo `src/Game.js` para entender que contiene, e iniciar el proyecto, deberían ver simplemente unos recuadros blancos.
- Pasar por props el valor de cada cuadrado desde Board a Square, de forma que quede lo siguiente:

Next player: X

0	1	2
3	4	5
6	7	8

- En el componente Square, agregar al botón un event handler `onClick={ }` para que ejecute una función con un simple `alert("click")`. (Solución hasta Break1)
- (Continúa)

Ejercicio 3 – TicTacToe (2)

- Vamos a agregar state al componente `Square` para guardar un `value: null` y ahora dentro del botón se muestre ese valor en vez del índice. También modifiquemos el botón para que actualice el state con un `value: "X"`.
- Al presionar sobre los cuadrados se deberán ir llenando con X. (Notar que si utilizan una función inline, `this` no va a estar definido, por lo que deben declarar un método de `Square` y realizarle un `bind`). (Solución hasta Break2)
- En este momento cada `Square` tiene su estado y su valor. Para completar el juego necesitamos poder colocar alternativamente X y O en el tablero. ¿Cómo solucionarían esto, ya que cada `Square` debe decidir qué letra poner? ¿Cómo entra `Board` en esto?
- (Continua)

Ejercicio 3 – TicTacToe (3)

- **Lifting State Up** -> Práctica común a la hora de refactorizar código en React. Subimos el state al componente padre común para poder compartirlo entre componentes hermanos.
- En Board vamos a definir un constructor y almacenar en el state un array con el valor de los 9 Square, en principio null. Volvemos a pasarle el valor a mostrar desde el Board a cada Square, pero en vez del índice, el valor que hay en el array correspondiente al índice del square.
- En Square borramos el constructor, no tendremos state, y en el botón volvemos a mostrar el valor recibido por props. ¿Y para el click del botón? ¿Como le comunicamos al Board que valor cambiar y en qué posición del array, desde Square?
- Para esto, el patrón común es pasarle una función desde Board a Square (Continua)

Ejercicio 3 – TicTacToe (4)

- En `renderSquare` de `Board` debería quedar así:

```
renderSquare(i) {  
  var self = this;  
  return (  
    <Square value={this.state.squares[i]} onPress={function() {  
      self.handleClick(i);  
    }}  
    />  
  );  
}
```

- El `handleClick` lo definimos como método de `Board` y desde ahí actualizamos el state de `Board` (en siguiente slide).

Ejercicio 3 – TicTacToe (5)

- Es importante al actualizar el state de Board en el handleClick, respetar la inmutabilidad de React, por eso realizamos una copia del array, la modificamos, y ese nuevo array es el que asignamos al state:

```
handleClick(i) {  
  var squares = this.state.squares.slice();  
  squares[i] = "X";  
  this.setState({ squares: squares });  
}
```

- En el componente Square, dentro del onClick simplemente tenemos que invocar a la prop onPress que nos llega:
- Solución hasta Break3.

```
<button className="square" onClick={this.props.onPress}>  
  {this.props.value}  
</button>
```

Ejercicio 3 – TicTacToe (6)

- `Square` ya no necesita ser un Class Component, así que hay que convertirlo en un Functional Component.
- Ahora necesitamos poder tomar turnos en el juego. Agregar en el state de `Board` quién es el próximo en jugar, por defecto empieza X. `xIsNext: true`
- Modificar el método `handleClick` en `Board` para que cambie el valor en el array según a quién le toca jugar, y luego actualice el valor de `xIsNext` en el state. Ya deberían poder jugar, simplemente no indica si alguien gana.
- También modificar el `render` de `Board` para que muestre correctamente a quien le toca jugar. (Solución hasta Break4)

Ejercicio 3 – TicTacToe (7)

- Hay una función auxiliar que determina si alguien ganó llamada `calculateWinner`, que recibe un array de 9 valores. Invocarla en el render de Board para mostrar un mensaje si alguien gana.
- También agregar en el `handleClick` de Board que no se pueda volver a clickear sobre una celda ya completada, y un chequeo para que si alguien ya ganó retorne y no siga actualizando el state. (Solución final)