Glosario

Declaración:

El proceso de definir una variable o función usando *let, function o const*. La diferencia entre let y const es que const **no puede ser modificado porque es constante.**

Las variables declaradas con const *siempre deben tener un valor inicial.*

```
let numero = 1 // Valor inicial
numero = numero + 1
console.log(numero) // En la consola va a salir el numero 2


const numeroConstante = 1 // Valor inicial
numeroConstante = 3 // Error
```

Si declaras adentro de una función, la variable no va a existir fuera de la función. **Se borra**.

```
function declarar() {

        const x = 5;
        console.log(x)

}

declarar() // Se ejecuta y cuando llega al console.log sale un 5 porque está adentro de la función
console.log(x) // En la consola, va a salir undefined. X aquí fuera no existe.
```

Lo que **sí se puede hacer**, es declarar una variable afuera de una función (con let) y cambiar el valor dentro de una función.

```
let numero = 1

function sumarUno() {

        numero = numero + 1
}

console.log(numero) // En la consola sale un 2
```

Asignación:

El proceso de asignarle un valor a una variable utilizando el operador de asignación "=". *Lo de la derecha se mete adentro de la izquierda.* No confundir el operador asignación con el operador flecha "=>".

```
edad = 12

function duplicar(numero) {

        return numero * 2;

}

const x = duplicar(edad)

¿Qué va a salir si escribo console.log(x)?
Aquí, quien está devolviendo o retornando algo, es duplicar(edad). Ninguna otra instrucción
está devolviendo nada. Ver valor de retorno más adelante.
```

Función:

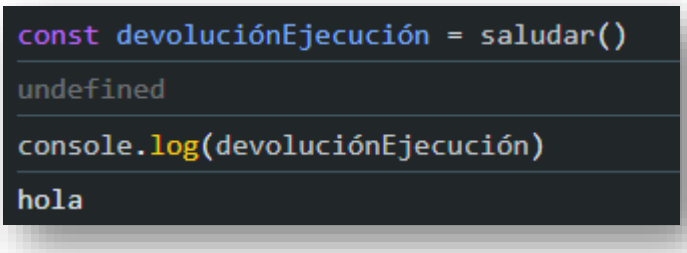
Una función es un bloque de código que realiza una tarea específica. Se puede llamar varias veces para ejecutar esa tarea. Puede recibir cero o más ***parámetros*** (leer más adelante) entre paréntesis y opcionalmente puede devolver un ***valor de retorno o devolución*** (leer más adelante).

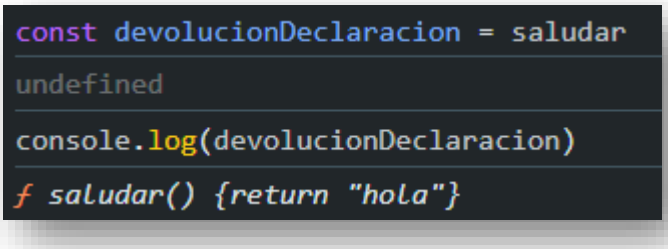
```
function saludar() {
        return "Hola"
}

const mensaje = saludar()
console.log(mensaje) // En consola sale Hola
```

Si nosotros queremos **ejecutar** una función, va con paréntesis (y parámetros) después de escribir el nombre de esta. Si nosotros queremos referenciar la **declaración** de la función (queremos ver su código, no que se ejecute), entonces escribimos la función sin paréntesis.

Para el ejemplo de saludar, se ve de la siguiente forma;





Ambas formas de escribir una función **devuelven** algo, y esa cosa que **devuelven** la estamos **asignando a la variable** constante. *La ejecución devuelve lo que va en el return,* mientras que *la declaración devuelve el código (definición)* de la función.

Parámetro:

Un parámetro es una variable que se utiliza para recibir información en una función. Esta información siempre son variables de cualquier tipo, las cuales pueden ser manipuladas u de otro modo utilizadas por las diferentes funciones. Los parámetros van entre paréntesis **separados por comas** después de el nombre de la función.

¿Te diste cuenta de que cuando usamos console.log, siempre escribimos algo entre paréntesis?

Eso que escribimos es el **parámetro que le estamos pasando**.

```
function logearParametro(param) {
        console.log(param)
}
logearParametro("Hola") // En la consola va a salir "Hola"

function sumar(x, y) {
        return x + y
}
console.log( sumar(2,3) ) // En la consola va a salir 5
const x = sumar(1,1)       // x va a ser igual a 2
```

Siempre que enviamos una variable por parámetro, lo que estamos enviando realmente es una "copia". Si mandamos un parámetro y ese parámetro se modifica, eso **no cambia a la variable original**.

```
let e = 2
function duplicar(numero) {
        numero = numero * 2
}
duplicar(e)
console.log(e) // En la consola va a seguir siendo 2
```

Si quisiéramos hacer el ejemplo de duplicar, se podría hacer así:

```
let e = 2
function duplicar(numero) {
        return numero * 2
}
e = duplicar(e)
console.log(e) // En la consola ahora sí sale 4
```

Valor de Retorno / Devolución:

El valor de retorno es el resultado que una función envía de vuelta cuando termina su ejecución. Se usa para enviar datos desde la función a donde fue llamada.

Típicamente se indica con *return,* pero no es necesario especificarlo en las ***funciones de retorno implícito*** (visto más adelante).

Una invocación de función siempre será igual a su valor de retorno. Por ejemplo, si una función devuelve (retorna) un 1, entonces (función() == 1) es true o ***truthy***.

```
// El operador igualdad que se escribe "==" (no confundir con asignación) devuelve true o false

function devolverCuadrado(numero) { return numero * numero }


console.log( devolverCuadrado(2) == 4 )                          // En consola sale un True
console.log( devolverCuadrado(2) == 2 * 2 )                      // En consola sale un True
console.log( devolverCuadrado(-2) == devolverCuadrado(2) )       // En consola sale un True
console.log( devolverCuadrado(1) == 2 )                          // En consola sale un False
```

Lean **el ejemplo de asignación en el glosario para ver un valor** de retorno en acción.

También se usan mucho en los métodos de Arrays (visto más adelante). Los **métodos de Arrays que devuelven otro array se deben asignar a una variable.**

```
const arrayDevuelto = arrayOriginal.map( … ) let
otroArrayDevuelto = CualquierCosa
otroArrayDevuelto = arrayOriginal.map( … )

¿arrayDevuelto va a ser el mismo array que arrayOriginal?
¿arrayOriginal vale lo mismo antes y después de usar el .map? La
respuesta está más abajo en la parte de métodos de array.
```

Truthy:

En JavaScript, un valor se considera truthy si se evalúa como verdadero en un contexto booleano. Esto quiere decir que cualquier *truthy* es igual a true si se usan en un entorno donde es esperan truthys. Los contextos booleanos son generalmente lo que va dentro del paréntesis de una declaración *if (truthy) { … }.*

Los valores booleanos se pueden combinar entre sí usando AND u OR explicados más adelante.

```
( True && False ) == False
( True || False ) == True
```

Los valores truthy incluyen:
- Cualquier número distinto de 0
- Cualquier string no vacío (“hola”)
- Objetos y arrays

```
If ( <truthy> ) { … } // El if siempre se va a ejecutar

Truthy puede ser cualquier cosa de las listadas arriba utilizando opcionalmente AND u OR. Por
ejemplo Truthy puede ser ( -1 && 3 > 4 && “Hola”)
```

Falsy:

Lo opuesto de truthy. Todos los falsys son igual a falso.

And:

Se usa para unir un componente truthy o falsy con otro componente usando “&&”. Esto sólo devuelve true si ambos componentes son truthy.

Or:

Se usa para unir un truthy o un falsy con “||”. Esto sólo devuelve true si al menos un componente es truthy.

Comparadores:

Son operadores que comparan dos valores y retornan un true o un false, por lo que se pueden usar para **crear truthys o falsys**. Por ejemplo (1 > 2) es un Falsy. Estos comparadores se incluyen a continuación. Nota: *NO* se debe confundir el mayor igual con el operador flecha (=>).

Nombre	Símbolo	Ejemplo	Valor de Ejemplo
Igualdad	**==**	1 == “1”	**True**
Igualdad Estricta	**===**	1 === “1”	**False**

Desigualdad	**!=**	1 != 2	**True**
Menor Que	**<**	3 < 5	**True**
Menor Igual	**<=**	5 <= 5	**True**
Mayor Que	**>**	3 > 5	**False**
Mayor Igual	**>=**	3 >= -8	**True**

```
const obj = { nombre: “Javier”, edad: 45}
( obj.nombre == “Javier” && obj.edad > 40 ) == True
```

Una comparación como esta se puede usar, por ejemplo en un array de objetos usando array.filter()

If (condicional):

Un if es una declaración condicional que se utiliza para ejecutar un bloque de código solo si una condición se evalúa como **true** (verdadero). La condición es una variable que siempre debe estar dentro de paréntesis y puede ser cualquier tipo de dato. Generalmente algo que indique si la variable es truthy o falsy.

```
if (condición) {
        // Código que se ejecuta si la variable condición es truthy
        // Si es falsy, simplemente se saltea
}


let numero = 5;
if (numero > 0) {
        console.log("El número es positivo");
 }
```

If con Else:

A veces, queremos hacer algo si la condición es **false** o **falsy**. Para esto usamos un else, que se ejecutará solo cuando la condición del if no se cumpla.

```
if (condición) {
        // Código que se ejecuta si la condición es true o truthy
} else {
        // Código que se ejecuta si la condición es false o falsy
}

let numero = -3;

if (numero > 0) {
        console.log("El número es positivo");
} else {
        console.log("El número es negativo");
}
```

Array []:

También llamado vector, es una colección *ordenada* de elementos que cada uno cuenta con un índice. Se debe declarar con **[corchetes]** y sus elementos van separados por comas en un orden específico. Se accede a sus componentes con **array[numÍndice]**.

```
const variableArray = [1,2,3,4,5]
let otraVariableArray
otraVariableArray = [ 1, 2,  [ "a",
"b"], "c", { letra: "d"} ]

otraVariableArray[0] == 1

otraVariableArray[1] == 2

otraVariableArray[3]  ==  [  "a",  "b"]
otraVariableArray[3][0]    ==       "a"
otraVariableArray[3][1]    ==       "b"
otraVariableArray[5] ==  { letra: "d"}
otraVariableArray[5].letra == "d"
```

Objeto {}:

Un **objeto** es una colección de **duplas de clave:valor** separadas entre comas donde cada valor puede ser de cualquier tipo. Los objetos permiten almacenar datos relacionados de una manera estructurada. Los objetos se definen entre **{llaves}** que contienen duplas clave:valor separadas con comas. Para *acceder* a un objeto y tomar una de sus variables o incluso agregar nuevas, se debe hacer **objeto.clave** u **objeto.método.**

Los objetos declarados por const, no pueden cambiar su valor, **pero sí se puede cambiar su colección de duplas clave:valor añadiendo nuevas o modificando existentes**.

```
const variableConstanteObjeto = { nombre: "Pepe", edad: 24}

let variableLetObjeto = { nombre: "Mateo", juguetes: ["Dinosaurio", "Batman"]}  variableLetObjeto.edad
= 8
variableLetObjeto.nombre = "Matías"

variableConstanteObjeto.hijo = variableLetObjeto

¿Qué sale si hago variableConstanteObjeto.hijo.nombre?
¿Cómo puedo hacer un console.log de el primer juguete de Matías?
¿Cómo puedo hacer un console.log del segundo juguete del hijo de Pepe?
```

Método:

Un método es una función guardada dentro de un objeto *(recordar que la mayoría de los componentes complejos en JavaScript son secretamente objetos).*

```
const objetoMetodos = {
        saludar: () => { console.log("Hola") }
        decir: ( parámetro ) => { console.log( parámetro) }
}

const objetoContenedor = {
```

contenido: objetoMetodos
}

¿Cómo puedo acceder al método saludar? objetoMetodos.saludar()
objetoContenedor.contenido.saludar()

¿Cómo puedo acceder al método decir? objetoMetodos.decir("Esto es lo que digo") objetoContenedor.contenido.decir("Acá digo otra cosa")

React:
React es una biblioteca de JavaScript que se utiliza para construir aplicaciones web usando componentes especiales creados por el programador.

Hook:
Un **hook** es una **función especial de React** que permite a los componentes funcionales acceder a características de React, como el estado y los efectos.

Componente de React:
Un componente de React es una parte reutilizable de la interfaz que puede contener su propia lógica y devuelve un **componente renderizable con etiquetas HTML**. Esto se desarrolla más adelante.

Variable de Estado:
Es una variable creada con useState en react. **Se necesitan cuando se quiere almacenar un dato en un componente de react**, ya que las variables normales se "olvidan" de sus valores.

useState:
useState es un hook de React que permite a los componentes funcionales manejar el estado interno. Te **permite crear variables de estado** que se pueden actualizar y utilizar en el componente sin que se pierda su valor.

useEffect: useEffect es un hook de React que permite realizar efectos secundarios (ejecutar código) en componentes. Se ejecuta después de que el componente se renderiza y se puede configurar para que se ejecute en ciertas condiciones. **Recibe siempre una función flecha y un array de dependencias**.

Fetch:
Fetch es una función nativa de JavaScript que se utiliza para realizar solicitudes HTTP y obtener datos de servidores.

Props:
Parámetros que reciben los componentes de react.

Tabla de Métodos de Arrays

Hay que recordar que los elementos de las funciones parámetro de algunos de estos métodos pueden tener cualquier nombre.

Array.map() Array.forEach() y Array.filter() todas reciben una **función flecha de tipo implícito** la cual *itera* sobre todos los elementos del array.

Método	Parámetro recibido	Valor de Retorno	Ejemplo

Array.map()	Función que toma un elemento y devuelve otro elemento modificado.	Devuelve un nuevo Array con sus elementos reemplazados. **No modifica el array original** por lo que, si no se está asignando a una variable el nuevo array devuelto, no se está guardando en ningún lado.	[0,1,2,3].map(num => num = num + 1) personas.map(p => ({ age: p.age, *etc* })) personas.map(p => <Component age={p.age}/>)
Array.forEach()	Recibe una función que toma un elemento y puede devolver o no alguna otra cosa. **forEach es un map sin retorno**	Nada. **No modifica el array original.**	[“pedro”, ”pepe”].forEach(x => console.log(x))
Array.filter()	Recibe una función que toma un elemento y lo transforma en un truthy o en un falsy.	Devuelve un nuevo array con la misma cantidad de elementos o menos. **No modifica el array original.**	personas.filter(per => per.edad >= 18)
Array.push()	Un elemento cualquiera para añadirlo al final del array	La nueva longitud del array. **Modifica el array original.**	personas.push({nombre: “Matías”, edad: 21})
Array.pop()	Nada. Saca el último elemento.	El elemento sacado. **Modifica el array original.**	[0,1].pop() == 1

Ejemplo para combinar dos o más métodos comenzando desde un solo array original

```
const arrayOriginal = [ 15, 16, 17, 18, 19, 20, 21, 22 ]

const arrayObjetos = arrayOriginal.map(
        numero => (  {edad: numero}  )
 )

const arrayMayorDeEdad = arrayObjetos.filter(
        elemento => elemento.edad >= 18

)
```

Componentes de React

Los componentes en react **son básicamente funciones** con algunas particularidades a tener en cuenta:

1. Pueden o no recibir parámetros (Props). Si se reciben parámetros, se debe hacer de la siguiente forma:

   *export function ComponenteEjemplo**({prop1, prop2, propN})** { … }*

   Children es un tipo especial de prop que se puede recibir como parámetro. “Children” será todo lo que esté entre la etiqueta de abertura del prop y la etiqueta de cerrado. Hay un ejemplo en el número 7.

2. Siempre deben retornar por lo menos un tag HTML vacío entre paréntesis

   return (<> … </>)

3. Siempre comienzan con mayúscula.

4. Siempre se deben exportar usando “export” o si no **“export default” en el caso que el componente sea uno que se deba comportar como página.** Se recomienda sólo exportar una cosa por archivo JavaScript.

5. Si se quiere que se ejecute lógica dentro de estos componentes, debe estar definida dentro del componente, pero antes del return.

6. Si se quiere utilizar una **variable de estado** (de useState) o un prop dentro del return, se deben escribir entre llaves.

```
// Falta importar useState aquí

export function Componente({ nameProp }) {
const [ ageState, setAgeState ] = useState(0)

return (<>
      <p> ¡Hola, tengo { ageState } años y mi nombre es { nameProp }! </p>  </>) }
```

7. Si se quiere pasarle un prop al componente, se debe hacer de la siguiente manera:

   <Componente nameProp={ “Ramiro” } />

¿Cómo uso un componente en react?

Una vez que tenemos nuestro componente definido (y exportado), debemos importarlo en la página en donde deseemos utilizarlo.

NOTA: Este es un componente que se llama “Componente”, el cual lo fui definiendo en la sección anterior con los 7 pasos.

```
import { Componente } from "./componentes/componenteRepaso.js";
```

Una vez esté importado, se puede utilizar en el return o antes del return si se quiere meter los componentes en una variable o en un .map o algo por el estilo.

Si se mete dentro del return, se hace de las siguientes formas:

```
return(<>
        <Componente nameProp={"Ramiro"}></Componente>
        <Componente nameProp={ variableDeEstadoParaNombre }/>
        { arrayDeComponentes }
        { arrayMapeadoAComponentes }
</>)
```

El arrayDeComponentes y el arrayMapeadoAComponentes **son variables javascript**, mientras que <Componente/> es un **componente HTML**. Noten la forma diferente que tienen de renderizarse.

Hay que recordar que **todas las variables y arrays que se quieran renderizar en el return son de estado** creados con useState y cambiaron su valor con un setter.

Antes de leer el resto de esta sección, ***deberías leer la sección de useState en el glosario.*** Para hacer un array de componentes, se puede hacer lo siguiente:

```
import { useState } from "react" import {
Componente } from "./componentes.js"
 export default function Página()
{
    const [ listaComponentes, setListaComponentes ] = useState([])
const nombres = ["Juan", "Juana", "Pedro", "Pepe"]

    const listaMapeada = nombres.map(         elemento =>
elemento = <Componente name={elemento} key={elemento}/>     )
    // Lo que hace este map es reemplazar los nombres por un componente
    // "key" es un prop que deben tener las listas de componentes
    // Key no puede repetirse por lista, por lo que elegí los nombres

setListaComponentes(listaMapeada)
     return(<> { listaComponentes }
</>)
}
```

Nota: **Si usan una base de datos, el mapeo debe estar dentro de useEffect** de la misma forma que hicimos juntos en clase.

Nota: **KEY NO ES NECESARIO** si no quieren usarlo, pero va a llenar la consola de errores o warnings. Aún así, el programa va a funcionar igual. Agreguen key sólo si sienten que lo entienden, y recuerden que cada elemento dentro de un array debe tener un “key” único. Generalmente, con personas o personajes, una buena key que se puede meter es su nombre (siempre y cuando el nombre sea único).

En bases de datos como las de Rick y Morty, los personajes tienen un **Id Propio que se puede usar como key**. Si el objeto tiene Id, usen eso en lugar de nombre.

useState

Asegurarse de haber leído **useState**, **hook y variable de estado** en el glosario antes de leer esto.

React es un framework que hace que un componente se recargue en la página muchísimas veces sin que el usuario se entere.

Una consecuencia de estos recargos es que **se pierden los valores almacenados en variables**, ya que al recargar, todos vuelven a tener su valor inicial.

La solución a este problema es useState.

Este es un hook específico de React, por lo que para que sea utilizado en cualquier lado, se debe importar con

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

useState se define usando un **array de 2 elementos** (no olvidarse del const)

El primero, es la **variable de estado** y el segundo es una función para **modificar su estado**. Los dos elementos del array pueden tener cualquier nombre, pero se recomienda que el segundo se llame "set" seguido del nombre del primero.

Entonces, un useState **define una variable de estado y su función setter**, además de también definir **su valor inicial**. Se recomienda que estos valores iniciales sean algo vacío, ya sea un cero, un objeto vacío {} o un array vacío []. Un ejemplo puede ser el siguiente;

```
const [varNumber, setNumber] = useState(0)
```

La variable entonces se guarda en varNumber, con un valor inicial de cero (el **valor inicial es lo que hay entre paréntesis de useState**).

Si se quiere cambiar, entonces se debe hacer setNumber(nuevoValor **)**. Nuevo valor puede ser cualquier tipo de dato válido de JavaScript, ya sean números, Arrays u objetos.

NO SE PUEDE CAMBIAR EL VALOR DE UNA VARIABLE DE ESTADO USANDO EL OPERADOR ASIGNACIÓN =. SÓLO SE PUEDE CAMBIAR CON SU FUNCIÓN SETTER.

useEffect

useEffect es un hook en React que te permite ejecutar código después de que un componente se renderiza. Es útil para manejar efectos secundarios, como llamadas a una base de datos.

El useEffect recibe 2 parámetros:

1) Una función flecha
2) Un array, llamado array de dependencia.

La función flecha contendrá todo el código que queremos que se ejecute **según los contenidos** del array de dependencia. ¿Qué quiere decir esto?

Un **array de dependencia vacío** hará que el código sólo se ejecute cuando el componente **es renderizado por primera vez**.

Si nosotros le añadimos variables dentro del array de dependencia, entonces el componente estará **atento a su valor**.

Cada vez que cambie el valor de cualquier elemento del array de dependencia, se ejecuta la función flecha.

Un array vacío también nos sirve para que la función flecha sólo se ejecute una única vez, independientemente de si se vuelve a renderizar como hace react tan seguido. Por lo general, un Fetch va dentro de un useEffect.

Al igual que useState, useEffect se importa directamente desde react. Se puede usar cualquiera de los dos métodos a continuación dependiendo de si queremos uno o ambos.

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

A continuación se demuestra un useEffect básico con una función que no hace nada porque todavía está vacía.

```
useEffect(
        () => {},   // La función que depende del array de dependencias
        []          // El array de dependencias
)
```

Más ejemplos de useEffect

```
useEffect(
() => {
            console.log("Alguna variable de estado cambió")
        },
        [varEstado1, varEstado2]
    )

useEffect(         () => {             const listaMapeada =
nombres.map(                elemento =>                 elemento =
<Componente name={elemento} key={elemento}/>
            )
setListaComponentes(listaMapeada)
        }, // Si la variable nombres cambia, esto se mapea de nuevo
        [nombres]
    )
```

Nota: La función flecha de un useEffect **JAMÁS DEBE PODER MODIFICAR UNA VARIABLE DENTRO DE SU PROPIO ARRAY DE DEPENDENCIA**.

¿Por qué es esto?

Porque si cada vez que cambia el valor del array de dependencia, se ejecuta la función flecha entonces el valor del array de dependencia cambia una vez más. Entonces, como cambió su valor, la función flecha se vuelve a ejecutar y vuelve a cambiar su valor y sigue así constantemente. En cortas palabras **se genera un loop infinito**.

Sin embargo se puede usar la variable dentro de la función.

Se pueden usar métodos como el .map() o el .filter() ya que no modifican el array original. También la variable se puede asignar a una variable auxiliar usando asignación. Esto se debe a que estas dos cosas sólo "leen" el valor de la variable de dependencia y no cambian nada.

Conceptos para Tener en la Cabeza

Todo esto lo fui explicando a lo largo del documento, pero aquí hay un recordatorio muy superficial. Si quieren detalles, van a tener que leer el documento de nuevo.

1) Un array.map() sirve para *transformar* todos los elementos de un array en algo nuevo. *Lo de la izquierda del operador flecha se transforma en lo de la derecha*.

2) Siempre que se quieran guardar datos en react **deben usar useState y una variable de estado**. Se va a necesitar 1 useState por dato que deseen guardar.

3) Siempre que se quiera **cambiar el valor de una variable de estado** hay que usar su setter. NO hay que usar asignación.

4) Si tienen una *variable de estado que es un array* y quieren mapearla, el método más sencillo es usar una variable auxiliar, y luego meterla dentro de otra variable de estado con un setter. Esto podría estar dentro de un useEffect.

```
const arrayAux = arrayPersonajes.map(
        elemento  => <TarjetaPersonaje nombre={elemento.name} edad={elemento.age}/>
)

setArrayTarjetas(arrayAux)

Ahora la variable de estado arrayTarjetas contendrá un mapeo de arrayPersonajes
```

5) Los métodos de los arrays se pueden combinar entre sí. Hay un ejemplo de esto en la página de la tabla de los métodos de arrays.

6) Un useEffect se usa para ejecutar el código dentro de la función flecha *de manera controlada y sólo cuando nosotros queramos.* Según los contenidos del array de dependencias, se ejecutan:

a. Sólo una vez cuando el componente se renderiza por primera vez *(El array de dependencias está vacío)*.

b. Cuando se renderiza por primera vez y además se ejecuta *cada vez que el valor de cualquier elemento del array de dependencias cambia*. Puede pensarse como que el código "depende" de sus valores y el useEffect los está "vigilando".

7) Después de hacer un Fetch, si nosotros ***seteamos la data usando un setter***, entonces la variable de estado contendrá información que será una promesa. **Las promesas son variables que se están descargando de internet** y pueden tener 2 estados:

   a. Descargando *(generalmente representado como un tipo de dato vacío como por ejemplo un array)*
   b. Descargado *(Ya tenemos la información disponible para utilizar)*

   Estos cambios entre descargando y descargado **se pueden detectar con el array de dependencias de un useEffect**. Esto es útil cuando queremos, por ejemplo, mapear los datos descargados a un array de componentes.

8) Para crear un nuevo proyecto, se debe usar el comando npx create-next-app@latest

   Y para correr un proyecto, se debe usar
   npm run dev

   Si para el examen les piden correr el backend además de correr el frontend que sería su proyecto, no se asusten. Es simplemente abrir la carpeta del backend en otra ventana de vscode y correrlo con el run dev.