¡Les damos la bienvenida!

¿Comenzamos?

Esta clase va a ser

- grabada

Clase 06. REACT JS

Promises, asincronía y MAP

CODERHOUSE

Temario

05

Componentes II

- ✓ Componentes II: Introducción
- ✓ Efectos

06

Promises, asincronía y MAP

- ✓ Promise
- ✓ MAP

07

Consumiendo API´s

- ✓ Paradigmas de intercambio de información
- ✓ REQUESTS VIA HTTP/S
- ✓ REQUESTS EN EL BROWSER

Objetivos de la clase

- **Conocer l**a API de promise, profundizando conceptos de asincronismo.
- **Aplicar** el método MAP para el rendering de listas.

CLASE N°5

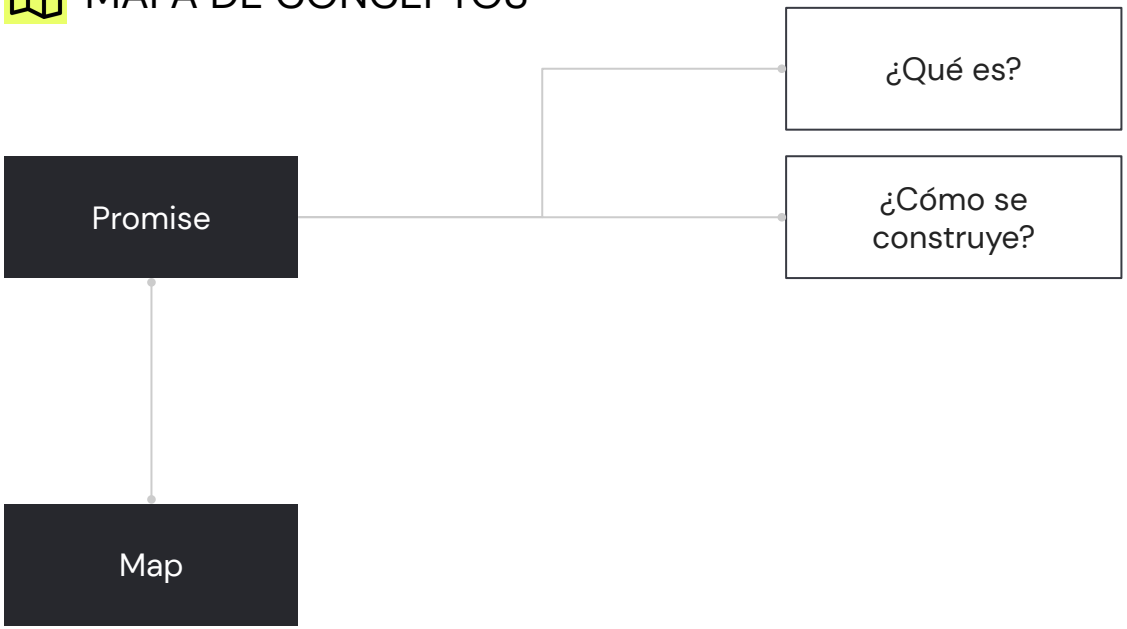
Glosario

Ciclo de vida: no es más que una serie de estados por los cuales pasa todo componente a lo largo de su existencia. Esos estados tienen correspondencia en diversos métodos, que podemos implementar para realizar acciones cuando se van produciendo.

Métodos de ciclos de vida:

- **useEffect(callback, []):** este método del ciclo de vida es de tipo montaje. Se ejecuta justo despues del primer renderizado del componente.
- **useEffect(callback, [props]):** Este método del ciclo de vida es de tipo cambio. Se ejecuta justo despues del primer renderizado del componente. Recuerda que el efecto se ejecutará tambien en el montaje.

MAPA DE CONCEPTOS



Promise

Promise

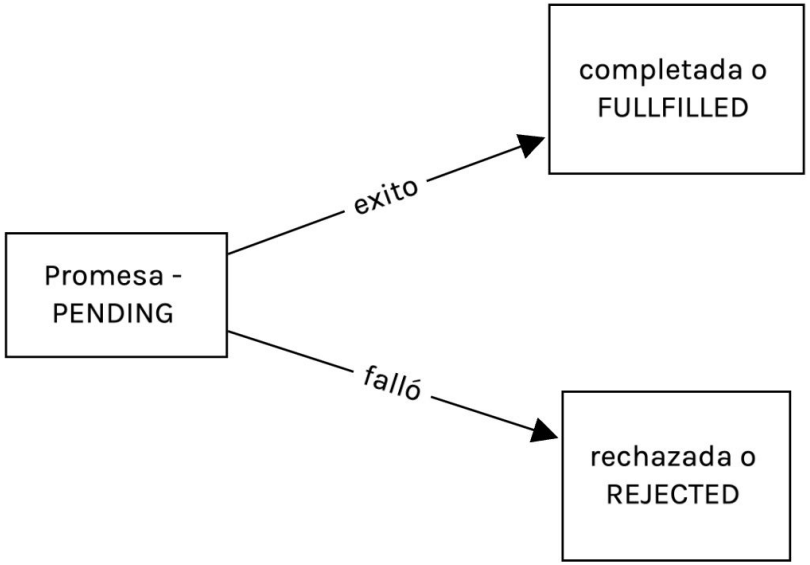
JavaScript tiene una API que nos permite crear y ejecutar distintas operaciones o conjuntos de operaciones.

Una promise (promesa en castellano) es un **objeto que permite representar y seguir el ciclo de vida de una tarea/operación (función).**

Estados posibles:
PENDING => (FULLFILLED || REJECTED)
PENDIENTE => (COMPLETADA || RECHAZADA)

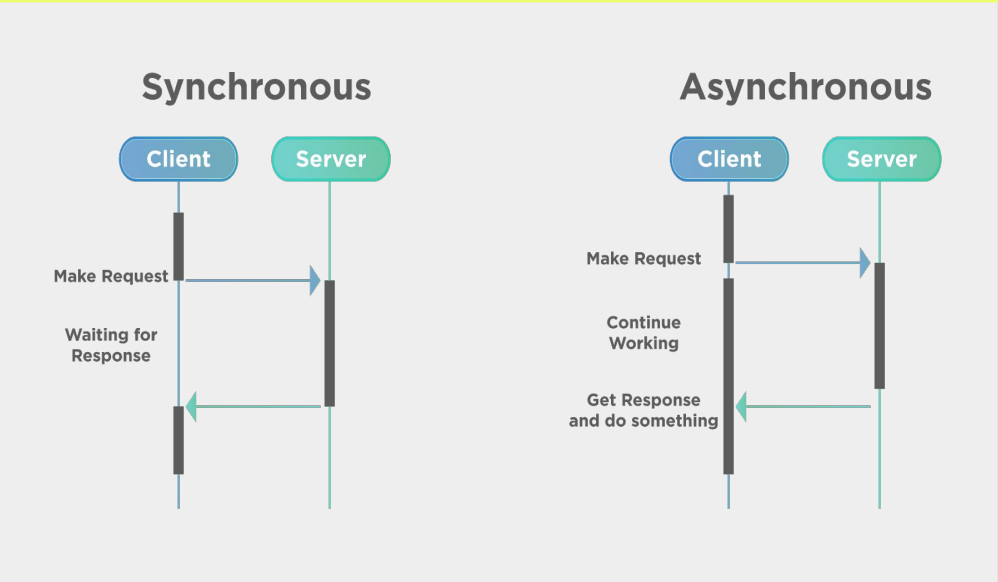
CODERHOUSE

Promise



CODERHOUSE

Promise



En contra de lo que se suele pensar, la sincronicidad o asincronicidad de una promise depende de qué tarea le demos.

Por defecto y diseño, **lo único que ocurre de manera asincrónica es la entrega del resultado.**

Promise

JavaScript

```
const task = new Promise((resolve, reject) => {
    // Tarea sincrónica
    resolve(true);
});

task.then( result => {
  console.log(result);
});
```

Console

```
true
```

Se construye de la siguiente manera.

Ejemplo de una promise que es siempre completada.

Ejemplo en vivo

Vamos al código

CODERHOUSE

Promise

JavaScript

```
const task = new Promise((resolve, reject) => {
    // Tarea sincrónica
    reject('Mensaje de error');
});

task.then( result => {
  console.log('No es error: ' + result);
  // No pasa por aquí
}, err => {
  console.log('Error: ' + err);
})
```

Console

```
"Error: Mensaje de error"
```

Si hay un rechazo, se captura de esta manera.

Ejemplo de una promise que es siempre rechazada.

Promise

JavaScript

```
const task = new Promise((resolve, reject) => {
    // Tarea sincrónica
    resolve(true);
});

task.then( result => {
  throw new Error("Cometimos error aquí");
  console.log('No es error: ' + result);
  // No pasa por aquí
}, err => {
  console.log('Error: ' + err);
}).catch(err => {
  console.log('Captura cualquier error en el proceso');
})
```

Console

```
"Captura cualquier error en el proceso"
```

Si fallamos en el callback del resultado.

Ejemplo de una promise donde fallamos al procesar el resultado.

Casos raros

```
const task = new Promise((resolve, reject) => {
    // Tarea sincrónica
    resolve(true);
});

task.then(res => {
  throw new Error('Oops!')
  console.log('Resolved: '+ res);
}, err => {
  console.log('Rejected: ' + err)
}).catch(err => {
  // Si recibo error puedo retornar
  // un valor por defecto
  console.log('Problema en lectura de resultado');
  return 'default_value'
}).then(fallback => {
   console.log(fallback);
});
```

```
"Problema en lectura de resultado"
"default_value"
>
```

CODERHOUSE

Promise

Usaremos **.then** para ver el resultado del cómputo de la tarea.

Algo interesante:

Todos los operadores then y catch son encadenables
.then().catch().then().then()

CODERHOUSE

Pro tip

En algunos navegadores ya tendremos disponible el **.finally()**, que lo podemos llamar al final de la cadena para saber cuando han terminado tanto los **completados** como los **rechazos.**

CODERHOUSE

Garantías de una -Promise-

- ✓ Las funciones callback nunca serán llamadas previo a la terminación de la ejecución actual del bucle de eventos en JavaScript.
- ✓ Las funciones callback añadidas con .then serán llamadas después del éxito o fracaso de la operación

Mock Async Service

Crea en JSBIN una promesa que resuelva en tres segundos un **array** de objetos de tipo producto.

Duración: **15 minutos**

ACTIVIDAD EN CLASE

Mock Async Service

Descripción de la actividad.

Crea en JSBIN una **promesa** que resuelva en tres segundos un **array** de objetos de tipo producto.

Al resolver, imprímelos en consola
{ id: string, name: string, description: string, stock: number }

Cuentas con 15 minutos para realizar esta actividad.

CODERHOUSE

Break

¡10 minutos y volvemos!

MAP

MAP

El **método map()** nos permite generar un **nuevo array** tomando de base otro **array** y utilizando una función transformadora.

Es particularmente útil para varios casos de uso.

CODERHOUSE

Ejemplo en vivo

Vamos al código

CODERHOUSE

Método MAP

JavaScript

```
const users = [
  { nombre: 'coder' },
  { nombre: 'house'}
]

console.log(users.map(user => user.nombre))

console.log(users.map(user => user.nombre).join(','))
```

Console

```
["coder", "house"]
"coder,house"
>
```

CODERHOUSE

Método MAP

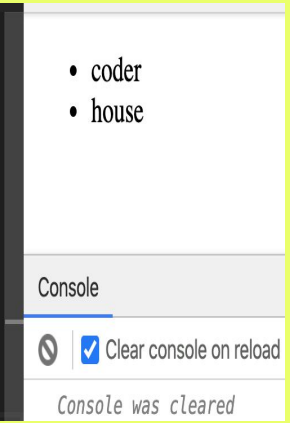
En react, con el método map, podremos hacer **render de una colección de objetos.**

Por ejemplo:

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

function App() {
  return <ul>
    {["coder", "house"].map(u => <li>{u}</li> )}
  </ul>
}

render(<App text="hello" />, document.getElementById('root'));
```



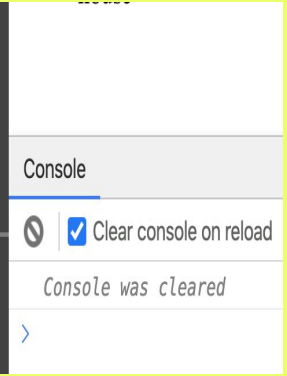
CODERHOUSE

Método MAP

Idealmente debemos **incluir** en c**ada elemento** la **propiedad key,** que marque la **identidad del elemento**. Esto ayudará a react a **optimizar el rendering a**nte cambios en el array.

De no tenerla podemos auto-generarla con el **index** provisto por el segundo parámetro de **map,** pero sólo optimizará si hay adiciones al final del array.

```
function App() {
  const [users, setUsers] = useState([
    { id: 1, name: 'coder' },
    { id: 2, name: 'house' },
  ])
  return <ul>
    {users.map(u => <li key={u.id}>{u.name}</li> )}
  </ul>
}
```



CODERHOUSE

#Coderalert

Ingresa al manual de prácticas y realiza la segunda actividad “Catálogo con MAPS y Promise”. Ten en cuenta que el desarrollo de la misma será importante para la resolución del Proyecto Final.

Actividad Nº 2

Segunda pre-entrega PF

Catálogo con MAPS y promises

Descripción de la actividad.

- ✓ Crea los componentes Item.js e ItemList.js para mostrar algunos productos en tu ItemListContainer.js. Los ítems deben provenir de un llamado a una promise que los resuelva en tiempo diferido (setTimeout) de 2 segundos, para emular retrasos de red

Actividad Nº 2

Catálogo con MAPS y promises

Recomendaciones.

- ✓ Item.js: Es un componente destinado a mostrar información breve del producto que el user clickeará luego para acceder a los detalles (los desarrollaremos más adelante)
- ✓ ItemList.js Es un agrupador de un set de componentes Item.js (Deberías incluirlo dentro de ItemListContainer de la primera pre-entrega del Proyecto Final)
- ✓ Implementa un async mock (promise): Usa un efecto de montaje para poder emitir un llamado asincrónico a un mock (objeto) estático de datos que devuelva un conjunto de item { id, title, description, price, pictureUrl } en dos segundos (setTimeout), para emular retrasos de red.

¿Preguntas?

Resumen
de la clase hoy

- ✓ Promises
- ✓ MAP

Opina y valora esta clase

Muchas gracias.

#DemocratizandoLaEducación