



Clase 06. React JS

# ***Promises, asincronía y MAP***



## ***OBJETIVOS DE LA CLASE***

- Conocer la API de **promise** profundizando conceptos de asincronismo
  - Aprender a aplicar el método MAP para el rendering de listas

***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:

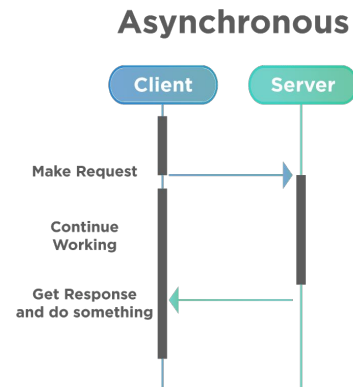
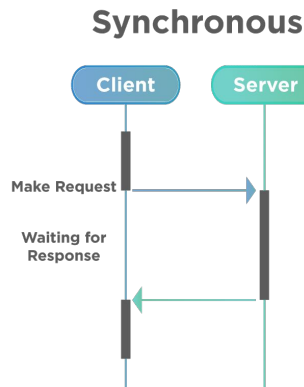
PENDING => (FULLFILLED || REJECTED)

PENDIENTE => (COMPLETADA || RECHAZADA)

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.**



***VAMOS AL CÓDIGO***



***CODER HOUSE***

Se **construye** de la siguiente manera:

JavaScript ▼

```
const task = new Promise((resolve, reject) => {  
  // Tarea sincrónica  
  resolve(true);  
});  
  
task.then( result => {  
  console.log(result);  
});
```

Console

true



*Ejemplo de una promise que es siempre completada*

Si hay un **rechazo** se captura de esta manera

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"

>

*Ejemplo de una promise que es siempre rechazada*



## De ocurrir un **error**

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"

>

*Ejemplo de una promise que es siempre rechazada*

## Si fallamos en el callback del resultado

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"

>

*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"
```

>

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()
```

The logo for JavaScript Promises, featuring the letters 'JS' in yellow inside a dark brown square, followed by the word 'Promise' in a bold, dark brown sans-serif font, all set against a bright yellow rectangular background.

## 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**

JavaScript ▾

```
const task = new Promise((resolve, reject) => {  
  // Tarea sincrónica  
  resolve(true);  
});  
  
task.then(res => {  
  console.log('Resolved: ' + res);  
}, err => {  
  console.log('Rejected: ' + err)  
}).finally(() => {  
  console.log('Finalizado');  
})
```

Console

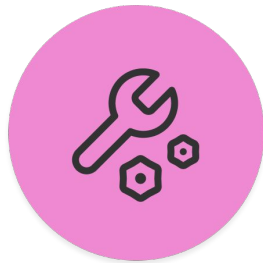
"Resolved: true"

"Finalizado"



# ***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***

Crear en [JSBIN](#) una promesa que resuelva en tres segundos un **array** de objetos de tipo producto. Al resolver, imprimirlos en consola

```
{ id: string, name: string, description: string, stock: number }
```

*Tiempo: 15 minutos*

***BREAK***



***CODER HOUSE***



***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.



***VAMOS AL CÓDIGO***



***CODER HOUSE***

## 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"
```



# 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'));
```

- coder
- house

Console



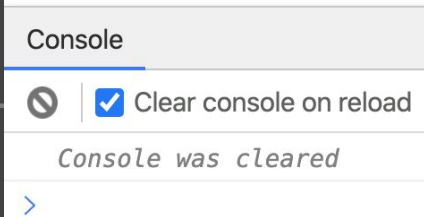
☒ Clear console on reload

Console was cleared

# Método MAP: Keys

Idealmente debemos **incluir** en **cada elemento** la **propiedad key**, que marque la **identidad del elemento**. Esto ayudará a react a **optimizar el rendering** ante cambios en el 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>  
}
```



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



# ***Catálogo con maps y promises***

Crea los componentes Item.js e ItemList.js para mostrar algunos productos en tu Home.js

Los ítems deben provenir de un llamado a una promise que los resuelva en tiempo diferido (setTimeout) de 2 segs para emular retrasos de red

*Formato de entrega: carpeta comprimida con archivos del proyecto*

*Tiempo: 40 minutos*

***¿PREGUNTAS?***

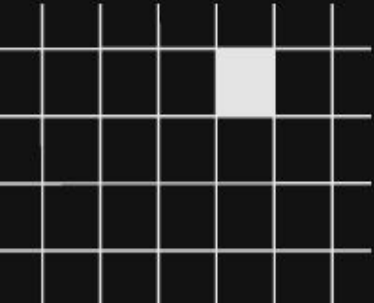






# ***¡MUCHAS GRACIAS!***

Resumen de lo visto en clase hoy:

- Asincronía, promises y maps en el enriquecimiento y optimización de UI's
- 



***OPINÁ Y VALORÁ ESTA CLASE***