

***RECUERDA PONER A GRABAR LA
CLASE***





¿DUDAS DEL ON-BOARDING?

MIRALO AQUI



Clase 06. REACT JS

PROMISES, ASINCRONÍA Y MAP



OBJETIVOS DE LA CLASE

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

GLOSARIO:

Clase 5

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 (class based):

- **componentWillMount()*:** este método del ciclo de vida es de tipo montaje. Se ejecuta justo antes del primer renderizado del componente.
- **componentDidMount():** método de montaje, que solo se ejecuta en el lado del cliente. Se produce inmediatamente después del primer renderizado. Una vez se invoca este método ya están disponibles los elementos asociados al componente en el DOM.
- **componentWillReceiveProps():** método de actualización que se invoca cuando las propiedades se van a actualizar, aunque no en el primer renderizado del componente, por lo tanto no se invocará antes de inicializar las propiedades por primera vez.
- **shouldComponentUpdate (nextProps, nextState):** es un método de actualización y tiene una particularidad especial con respecto a otros métodos del ciclo de vida, que consiste en que debe devolver un valor booleano.
- **componentWillUpdate (nextProps, nextState):** este método de actualización se invocará justo antes de que el componente vaya a actualizar su vista.

GLOSARIO:

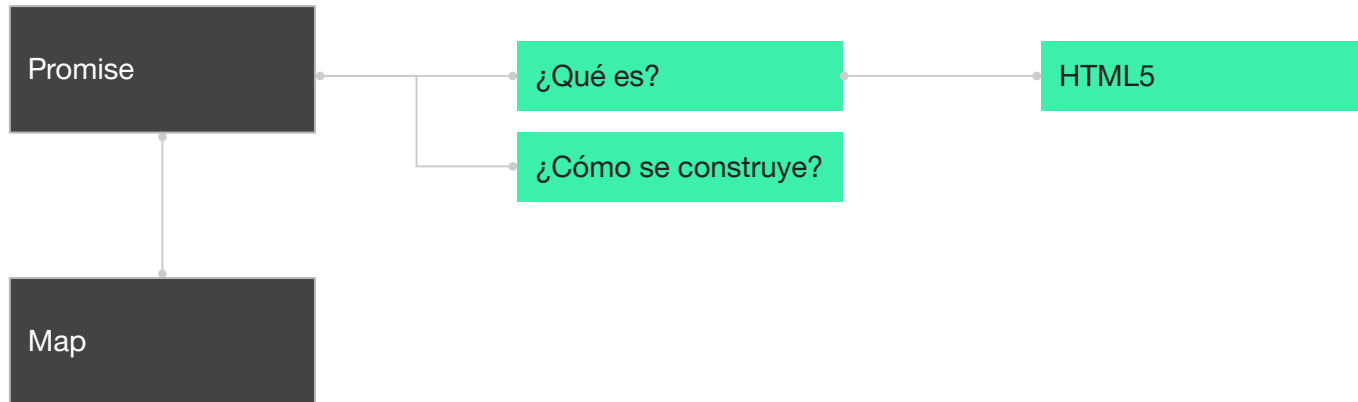
Clase 5

- **componentDidUpdate (prevProps, prevState):** método de actualización que se ejecuta justamente después de haberse producido la actualización del componente.
- **componentWillUnmount():** este es el único método de desmontado y se ejecuta en el momento que el componente se va a retirar del DOM.

MAPA DE CONCEPTOS

MAPA DE CONCEPTOS CLASE 6

¡Para
recordar!



CRONOGRAMA DEL CURSO

Clase 5



Componentes II



EJEMPLOS EN VIVO



CLICK TRACKER



CONTADOR CON BOTÓN

Clase 6



Promises, asincronía y MAP



EJEMPLOS EN VIVO



MOCK ASYNC SERVICE



CATÁLOGO CON MAPS Y PROMISES

Clase 7



Consumiendo API's



EJEMPLOS EN VIVO



FETCH API-CALL



DETALLE DE PRODUCTO A Y B

PROMISE



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

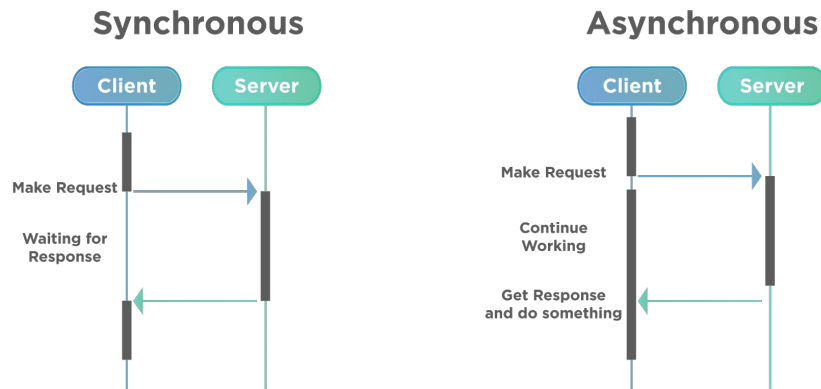
Una promise (promesa en castellano) es un **objeto que permite representar y seguir el ciclo de vida de una tarea/operación (función).**

Estados posibles:

PENDING => (FULLFILLED || REJECTED)

PENDIENTE => (COMPLETADA || RECHAZADA)

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.

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 dark brown sans-serif font, all on a yellow 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

Crea en [JSBIN](#) una promesa que resuelva en tres segundos un **array** de objetos de tipo producto.



¡A PRACTICAR!

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.



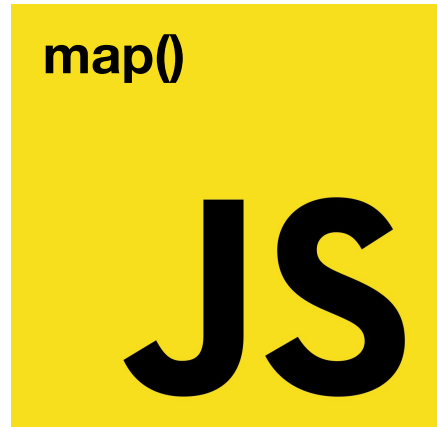
BREAK

¡5/10 MINUTOS Y VOLVEMOS!

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

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"
```

>

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>
}
```

Console

 ☒ Clear console on reload

Console was cleared

>

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



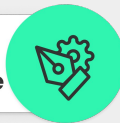
CATÁLOGO CON MAPS Y PROMISES

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

CATÁLOGO CON MAPS

Formato: link a último commit con el feature. Debe tener el nombre "Idea+Apellido".

Desafío
entregable



>> Consigna: 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

>>Aspectos a incluir en el entregable:

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 del desafío 3)

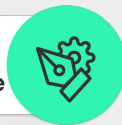
Implementa un **async mock** (promise): Usa un efecto de montaje para poder emitir un llamado asincrónico a un mock (objeto) estático de datos que devuelva un conjunto de item { id, title, description, price, pictureUrl } en dos segundos (setTimeout), para emular retrasos de

CODER HOUSE

CATÁLOGO CON MAPS

Formato: link a último commit con el feature. Debe tener el nombre “Idea+Apellido”.

Desafío
entregable



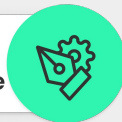
>> Ejemplo inicial:

```
function Item({ item }) {  
  // Desarrolla la vista de un ítem donde item es de tipo  
  // { id, title, price, pictureUrl }  
}  
  
function ItemList({ items }) {  
  // El componente va a recibir una prop `items` y va a mapear estos `items` al componente  
  `<Item ... />`  
}
```

CATÁLOGO CON MAPS

Formato: link a último commit con el feature. Debe tener el nombre "Idea+Apellido".

Desafío
entregable



>> Ejemplo Item:

Songs & Ballads



Compendio de textos

[Ver detalle del producto](#)

Stock disponible: 17

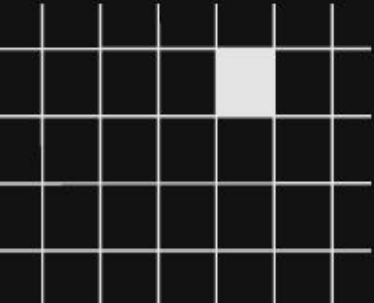
¿PREGUNTAS?





¡MUCHAS GRACIAS!

Resumen de lo visto en clase hoy:

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



OPINA Y VALORA ESTA CLASE