Callbacks y APIs

Callbacks y promesas

{desafío} latam_

Utilizar elementos de programación asíncrona para resolver un problema simple distinguiendo los diversos mecanismos para su implementación acorde al lenguaje Javascript.

Utilizar el objeto XHR y la API Fetch para el consumo de una API externa y su procesamiento acorde al lenguaje Javascript.

{desafío} latam_

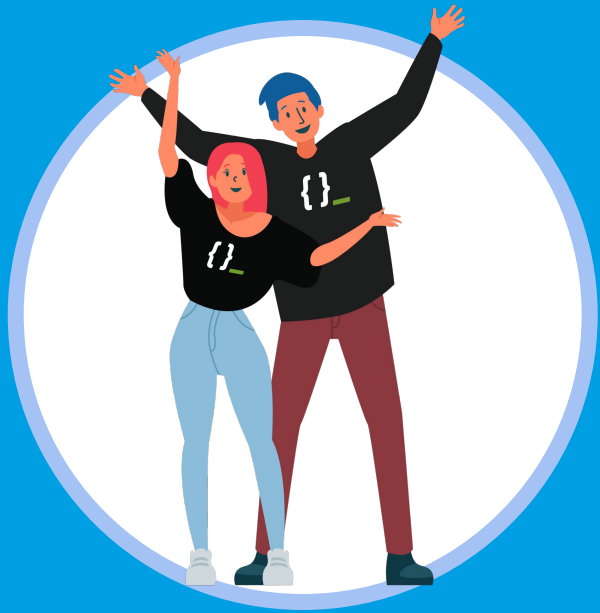
- Unidad 1:
  ES6+ y POO
- Unidad 2:
  Herencia
- Unidad 3:
  Callbacks y APIs

Te encuentras aquí

¿Qué aprenderás en esta sesión?

- *Utiliza instrucciones de ejecución asíncrona (callbacks, promises, async/await) para resolver un problema simple de asincronía acorde al lenguaje Javascript.*
- *Utiliza generación y captura de errores personalizados para resolver un problema simple de programación asíncrona acorde al lenguaje Javascript.*

Comentemos: ¿Qué conceptos de la clase te parecen más complejos?



{desafío} latam_

/* setTimeout */

setTimeout

El método setTimeout() llama a una función o evalúa una expresión después de un número específico de milisegundos. Así mismo, pertenece a una de las funciones Web APIs que tienen los navegadores y que se agrega directamente en el Call Stack.

Este método tiene doble funcionalidad:

1. El callback pasado como primer argumento se ejecutará después del tiempo establecido en el segundo argumento.

2. Su ejecución no bloquea el stack, por lo que es una función que se procesa de forma asíncrona.

setTimeout

Sintaxis

El único argumento requerido es “function” y es lo que se va a ejecutar o será la expresión que se evaluará al cumplirse el tiempo, que es lo que establece el argumento milliseconds para indicar cuánto deberá esperar function para ejecutarse, este valor es opcional. Los argumentos siguientes son parámetros adicionales que se pueden pasar a function y también son opcionales.

```
setTimeout(function(){}, milliseconds, param1, param2, ...)
```

setTimeout

Implementación de setTimeout

El primer ejemplo consiste en esperar una cantidad de tiempo para evaluar la expresión que contiene la función que se le pasó por argumento, que en este caso es mostrar en la consola un mensaje.

```
setTimeout(() => {
     console.log('hola mundo!');
}, 1000)
```

1. Establecemos qué será lo que ejecutará la función que le pasamos a setTimeout.
2. Le indicamos el tiempo que demorará en realizar la operación que le indicamos.
3. Luego de 1000ms, se ejecuta el código y aparece el log en la consola del navegador.

/* Promesas (Promise) */

Promesas (Promise)

Una promesa es un objeto que representa el estado de una operación asíncrona. Estos estados son:

- pendiente - cuando todavía no empieza o no ha terminado de ejecutarse.
- resuelto - en caso de éxito.
- rechazado - en caso de fallo.

El resultado de una promesa podría estar disponible ahora o en el futuro. Por otra parte, las promesas permiten ejecutar código asíncrono y devolver valores como si fueran secuenciales, ya que permiten continuar con nuevas sentencias una vez resuelta la promesa. Lo que retorna una promesa es otra promesa por lo que permite que se pueda continuar creando sentencias o expresiones utilizando el método then para cuando la promesa anterior sea resuelta y el método catch cuando la promesa sea rechazada.

```
new Promise((resolve, reject) => {/*.....*/})
```

Promesas (Promise)

Se requiere pasar como argumento una función callback que contenga dos (2) parámetros, puedes nombrarlos como quieras, en este caso se usarán resolve y reject que son los valores por defecto. Como se muestra a continuación:

```
const promise = new Promise((resolve, reject) => {
     const value = true;
     value ? resolve('Exito') : reject('Rechazado')
})
promise.then(resp => console.log(resp)) // output: Exito
```

```
Exito                                                    VM21729:6
<· ▶ Promise {<resolved>: undefined}
```

Promesas (Promise)

Vemos que el resultado de la promesa se resolvió y devolvió “Exito”. Ahora veamos qué pasa en caso contrario. Es decir, si modificamos la variable “value” por false.

Aquí la promesa fue rechazada por lo que devolvió “Rechazado”, pero adicionalmente a esto, se generó un error no controlado debido a que en nuestro código no está manejado este tipo de situaciones, por lo que veremos cómo hacerlo en el capítulo Manejo de errores de la Lectura - Callbacks y APIs (Parte II). Para mayor información también puedes revisar la siguiente documentación.

Lo visto hasta el momento, es solo la sintaxis de Promise y cómo funciona de manera general, pero en el punto a continuación, si se trabajan con las promesas de forma completa, explicada paso a paso y con ejemplos de implementación reales.

/* Promise.all */

Promise.all

- Permite ejecutar un objeto iterable o arreglo de múltiples promesas, y esperar que se resuelvan para entregar todos los resultados de una vez.

- “Retorna” es un arreglo con tantos elementos que se le hayan pasado al método all en el objeto iterable.

```
Promise.all(objetoIterable)
.then((result) => {/*...*/})
.catch((error) => {/*...*/})
```

Ejercicio guiado: Promise.all

Ejercicio guiado

Promise.all

Aplicando la sintaxis anterior, se deben cumplir todas las promesas iniciadas en variables separadas, la primera se debe igualar a un número, la segunda se debe igualar a Promise.resolve(2), y la tercera a una nueva promesa que retorne el número tres cuando se resuelva.
Implementar la sentencia del Promisa.all y ver cómo se deben cumplir todas las promesas para que se retorne un arreglo con los resultados. Por consiguiente, todas las promesas se deben enviar al método “all”, y cuando estas se cumplan, se debe mostrar por terminal el resultado.

Paso 1: Crear una carpeta en tu lugar de trabajo favorito y dentro de ella crea un script.js.
Seguidamente, en el archivo script.js vamos a crear primeramente las tres constantes que tendrán las promesas por individual.

```
const promise1 = 1;
const promise2 = Promise.resolve(2);
const promise3 = new Promise((resolve, reject) => {
   setTimeout(() => {
       resolve(3)
   }, 1000);
});
```

Ejercicio guiado

Promise.all

Paso 2: Luego, utilizando el Promise.all, pasamos como arreglo las tres promesas por individual, seguidamente cuando ya se cumplan todas las promesas (then) se muestra la respuesta recibida mediante un console.log.

```
Promise.all([promise1, promise2, promise3]).then(response => {
   console.log(response); // [1, 2, 3]
});
```

Paso 3: Ejecutamos ahora el archivo script.js desde la terminal con ayuda de Node, para ver el resultado:

```
[ 1, 2, 3 ]
```

Pero, qué fue lo que hizo el código cuando se ejecutó en la terminal: la primera promesa sólo devuelve un uno (1). Mientras que la segunda promesa retorna una promesa resuelta con valor de respuesta dos (2). Posteriormente, la tercera promesa retorna una promesa resuelta, pero después de 1 segundo. Luego de finalizar su ejecución, todas las promesas se resuelven y retornan a Promise.all con la respuesta de éstas, devolviendo un arreglo con las tres respuestas.

Promise.race

Ejercicio guiado: Promise.race

Promise.race es bien parecido a Promise.all, solo que para esta función basta con que una de las promesas esté resuelta para retornar el resultado de la promesa ganadora.

Realizar un ejemplo donde existan tres promesas, cada una regresará un valor distinto (1, 2 y 3 respectivamente), de acuerdo a la ejecución del tiempo que tenga cada una.

Ese tiempo debe originarse en una función externa, la cual va a retornar un número aleatorio que servirá como tiempo para los setTimeout de cada promesa, dando un valor aleatorio a cada una de ellas.

Para realizar este ejemplo, se deben seguir los siguientes pasos:

Ejercicio guiado: Promise.race

Paso 1: Crear una carpeta en tu lugar de trabajo favorito y dentro de ella crea un archivo script.js. Luego, en el archivo script.js, creamos primeramente la función llamada randomNumber que retorna el número aleatorio, recibiendo dos valores (uno mínimo y uno máximo), para ayudar al método Math.random() a generar el número entre la diferencia de los dos números recibidos por la función.

```
const randomNumber = (min, max) => {
   return parseInt(Math.random() * (max - min) + min);
}
```

Paso 2: Crear la primera promesa que posteriormente replicamos cambiando solo algunos valores. Esta primera promesa se construirá sobre una constante y dentro de ella se utilizará el método setTimeout para resolver la promesa, enviando el número 1 en este caso, es importante indicar que el valor del tiempo del setTimeout se solicita a la función creada anteriormente, denominada randomNumber, pasando el valor para el mínimo y el máximo.

```
const promise1 = new Promise((resolve, reject) => {
   setTimeout(() => { resolve(1) }, randomNumber(500, 2000));
});
```

Ejercicio guiado: Promise.race

Paso 3: Copiar la primera promesa creada y le cambiamos los valores de las constantes, es decir de la variable, y el valor que resuelve para cumplir la promesa. En este caso, como el primer valor que regresa la promesa es uno (1), entonces en estas dos promesas se regresará el dos (2) y el tres (3) respectivamente.

```
const promise2 = new Promise((resolve, reject) => {
   setTimeout(() => { resolve(2) }, randomNumber(500, 2000));
});
const promise3 = new Promise((resolve, reject) => {
   setTimeout(() => { resolve(3) }, randomNumber(500, 2000));
});
```

Ejercicio guiado: Promise.race

Paso 4: Ya solo queda hacer el llamado implementando el Promise.race y pasarle como argumento las tres promesas creadas para que retorne la promesa ganadora y entonces (then) se muestre por la terminal el resultado.

```
Promise.race([promise1, promise2,
promise3]).then(response => {
   console.log(response);
});
```

Paso 5: Ejecutamos ahora el archivo script.js desde la terminal con ayuda de Node con el siguiente comando node script.js, para ver el resultado:

```
2
```

/* Fetch API */

¿Qué es Fetch?

- Fetch API es una funcionalidad nativa de JavaScript.
- Permite realizar peticiones HTTP para obtener información de forma asíncrona desde un servidor.
- Fetch trabaja con promesas.
- Lo podemos utilizar para extraer recursos como texto, imágenes, objetos, entre otros.

Fetch API

Ejercicio guiado Fetch API

A continuación, realizaremos un ejercicio en el cual vamos a extraer información de un arreglo de objetos y mostrar los datos que se encuentren almacenados.
El objetivo es conocer el flujo de utilización de fetch en JavaScript para realizar peticiones y mostrar la información obtenida.

Ejercicio guiado Fetch API

- **Paso 1:** Creamos un archivo HTML y uno para JS, ambos quedarán enlazados.
- **Paso 2:** Creamos un archivo en el mismo nivel del HTML y el JS que llamaremos datos.txt.
- **Paso 3:** En el archivo datos.txt escribiremos el siguiente texto “Estoy aprendiendo a utilizar fetch en JavaScript”

Ejercicio guiado Fetch API

- **Paso 4:** En el archivo de JavaScript escribimos el código para leer el archivo txt y mostrarlo en la consola del navegador.

```
const consultarDatos = () => {
  const url = "datos.txt";
  fetch(url)
    .then((respuesta) => {
      console.log(respuesta);
      return respuesta.text();
    })
    .then((datos) => console.log(datos));
};

consultarDatos();
```

Ejercicio guiado Fetch API

Análisis del código anterior.

1. Creamos una función consultarDatos.
2. Almacenamos en una constante url la ruta del archivo datos.txt.
3. Hacemos **fetch** a la url.
4. Una vez realizado el fetch, **entonces (primer then)** muéstrame en la consola la respuesta a esta petición y retorna la respuesta en formato texto.
5. Entonces, **(segundo then)** una vez realizado lo anterior, quiero que me muestres los datos que están dentro del archivo txt.

```
Estoy aprendiendo a utilizar fetch en JavaScript                fetchApi.js:8
```

Ejercicio guiado Fetch API

Fetch async await

- **Paso 5:** Transformemos la función a un proceso asíncrono, esto permitirá utilizar async y await. De este modo, tendremos acceso a los datos de manera más directa con menos código.

```
const consultarDatos = async () => {
  const url = "datos.txt";
  const response = await fetch(url);
  const datos = await response.text();
  console.log(datos);
};

consultarDatos();
```

Ejercicio guiado Fetch API

Fetch async await

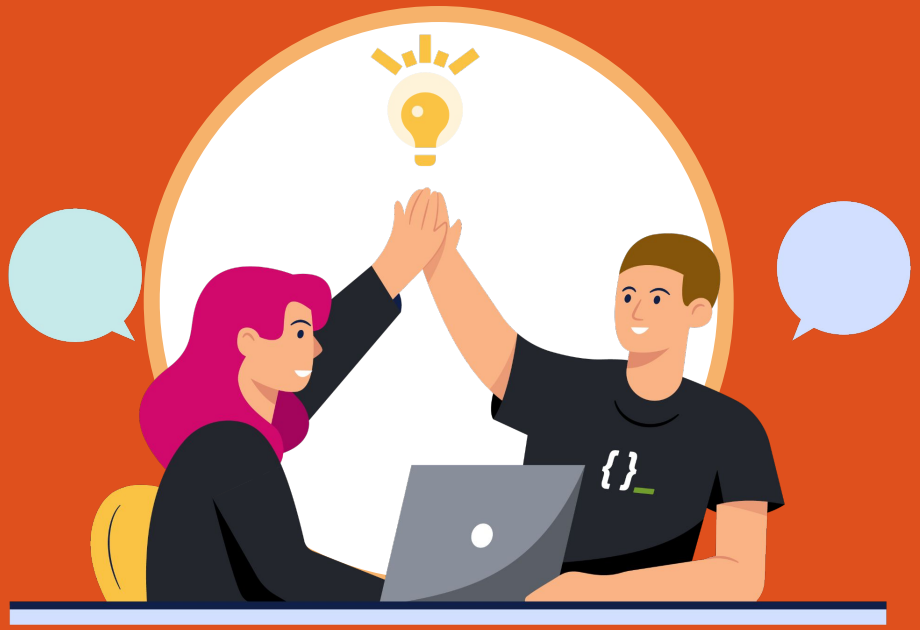
- **Paso 6:** También podríamos consultar información proveniente de un servidor externo conocido también como API.

```
const consultarDatos = async () => {
  const url = "https://jsonplaceholder.typicode.com/todos/1";
  const response = await fetch(url);
  const datos = await response.text();
  console.log(datos);
};

consultarDatos();
```

Datos provenientes de un servidor externo.

En siguientes sesiones profundizaremos en el concepto de API y obtención de datos con Fetch.



Resume con tus palabras: ¿Qué has aprendido hoy?

{desafío} latam_

Resumiendo

- El método setTimeout() llama a una función o evalúa una expresión después de un número específico de milisegundos.
- Los estados de una promesa son
  - pendiente - cuando todavía no empieza o no ha terminado de ejecutarse.
  - resuelto - en caso de éxito.
  - rechazado - en caso de fallo.
- El resultado de una promesa podría estar disponible ahora o en el futuro. Por otra parte, las promesas permiten ejecutar código asíncrono y devolver valores como si fueran secuenciales, ya que permiten continuar con nuevas sentencias una vez resuelta la promesa.

Próxima sesión...

- *Desafío opcional - Promesas*

{desafío}
latam_

{desafío}
latam_

Academia de
talentos digitales