Manejo de Errores en JavaScript

Escribir programas que anden bien cuando todo funciona como esperabamos es un buen comienzo. Pero esto no sucede todo el tiempo, siempre vamos a encontrar situaciones que estaban más allá de lo que podiamos esperar que suceda (usuarios). Y acá es donde se pone un poco más díficil. Veamos como podemos manejar situaciones inesperadas dentro de nuestro código, para eso vamos a aprender sobre **errores** en JavaScript.

Tipos de Errores

Mientras codeamos, o mientras ejecutamos nuestro programas pueden aparecer distintos tipos de errores en distintos momentos según quién causa el error, el tipo de error y cuando ocurre. Vamos a poder distinguir entre los siguientes tipos:

- **Errores de Sintaxis**: Se producen porque el programador no respeta las reglas sintácticas del lenguaje.
- **Errores Semánticos**: Se dan por el mal uso de algún *Statement* del lenguaje. ej: Loop infinito.
- **Errores Lógicos**: Aparecen porque el código no realiza lo que esperabamos que haga.

Desde el punto de vista de *cuando* surge el error, podemos tener:

- **Errores en tiempo de Compilación**: Aparecen cuando nuestro código es parseado por un compilador o intérprete (errores de sintaxis).
- **Errores de Runtime**: Los errores semánticos y de lógica van a aparecer cuando el código se este ejecutando.

Según quien causa el error:

- **Errores de Programación**: Es causado por un error del programador, por ejemplo: Utiliza mal una función y pasa argumentos incorrectos. Estos son los famosos `bugs`.
- **Problemas Genuinos**: Escapa a las manos del programador y ocurren en programas que están bien codeados, por ejemplo, cunado un usuario ingresa un input que la función no esperaba o el servidor al que nos queríamos conectar está caído, etc...

Podemos intentar resolver estos problemas (o alertar que ocurren) usando algunas funciones conocidas del lenguaje, como hacer un console.log() con un mensaje, o retornar un valor extraño cuando ocurra un error (por ejemplo -1), etc... Pero todo esto sólo nos servirá para controlar algunos errores en ambientes semi controlados (yo mismo invoco las funciones que estoy armando, y voy a entender cómo manejar los errores). Para los demás errores, o cuando sucede algo extraño, queremos se frene la ejecución (o cambie de rumbo) y continuar en un lugar en donde se sepa como *manejar* el error. Hacer esto, en varios lenguajes, es conocido como **manejo de excepciones**.

Manejo de excepciones

Básicamente, es posible que el código *levante (raise)* o *tire (throw)* una excepción, que es un valor (un objeto). Podríamos decir que es parecido a un `return`, pero con superpoderes, porque este `return` puede volver no sólo *salir* de la función en la que está, si no saltear varios execution contexts hasta llegar el entorno más alto donde se haya iniciado la serie de invocaciones que llegaron a generar una excepción. En inglés este proceso se conoce como *unwinding the stack*. Por suerte, cuando programamos podemos intentar *agarrar (catch)* una

excepción que va *subiendo* (o *bajando*) por el stack de ejecución, de tal forma que ejecutemos código en donde agarramos la excepción y seguir desde ahí. Para hacer esto en JavaScript vamos a usar el statement: `try` y `catch`:

```
try {
   //Código a ejecutar
   [break;]
}

catch ( e ) {
   // Código a ejecutar si ocurre una excepción (acá la agarramos)
   [break;]
}
// el finally es opcional
[ finally {
   // Siempre se ejecuta este código, haya o no una excepción
}]
```

Por ejemplo:

```
function lastElement(array) {
  if (array.length > 0)
    return array[array.length - 1];
  else
    throw "No existe el último elemento de un arreglo vacío.";
}

function lastElementPlusTen(array) {
  return lastElement(array) + 10;
}

try {
  print(lastElementPlusTen([]));
}
catch (error) {
  print("Hubo un problema ", error);
}
```

Cómo vemos en el ejemplo, `throw` es el *keyword* usado para crear una excepción. Ahora, cualquier código que se ejecute, o haya sido ejecutado desde lo que esté dentro del `try` statement, al generar una excepción, va a frenar su ejecución y devolver la excepción al `catch` statement. La variable `error`, en este caso, es el **nombre** que le damos a la *excepción* que acabamos de capturar.

> Si no hay excepciones, entonces nunca se ejecuta lo que está en `catch`.

Noten que la función `lastElementPlusTen` no tiene idea que `lastElement` puede no funcionar, simplemente la invoca. Eso es lo bueno de manejar excepciones, sólo nos tenemos que concentrar en donde se produce, y donde la atrapamos, todas las invocaciones en el medio, no tienen que enterarse.

Tal vez no lo sabíamos, pero muchos errores en realidad lo que hacen es tirar una excepción. Por ejemplo:

```
try {
  console.log(hola);
}catch (error) {
  console.log("Atrapado: " + error.message);
}
```

En casos como este, Objetos especiales son tirados como error. Estos objetos contienen una propiedad `message`, que contiene una descripción del problema. Podemos crear nosotros mismos este tipo de Objetos usando el constructor:

```
throw new Error('Hola no existe!!!');
```

Cuando una excepción es *tirada*, pero no hay nadie que la *atrape*, empieza a subir por el stack de ejecución, hasta que finalmente llega hasta el ambiente global, en donde es *atrapada* por este. Por lo tanto, cada *enviroment* va a manejar como quiera la excepción, en general dejan de ejecutar lo que estaban haciendo y te muestran la excepción con un formato particular.

Errores con el Event Emitter

En ciertos casos, por la naturaleza asincrónica de JavaScript, podemos perder el rastro de cómo suben las excepciones, o tal vez queremos saber si hay un error o no en otro contexto por el cúal no *subirá* la excepción. Para resolver esto, podemos usar el **event emitter** como un emisor de errores. Básicamente, pondríamos un *listener* a escuchar por un evento de tipo **Error**, y luego, en nuestro código simplemente emitiriamos un evento de este tipo cuando encontremos un error.

Error-Fist callback

Cuando codeamos funciones que ejecutan callbacks, si existió un error podemos crear un nuevo `Error` y pasarlo como **primer parámetro** cuando invocamos el callback:

```
  //hubo un error
  return cb(new Error('pasó tal cosa'), null);

  // no hubo problemas
  return cb(null, datos);
```

Noten, que cuando hacemos esto no ejecutamos un `throw`, ya que esperamos que alguien lo haga cuando vea el resultado del callback. De esta forma, estamos generando errores **Asincrónicos**.

> Si usamos el patrón de Event Emitter o error first callback los errores se generan asincrónicamente, si lo hacemos con `throw` lo estamos haciendo de manera sincrónica.

Ejemplos de funciones conocidas y cómo manejan los errores

Función	Tipo de función	Ejemplo de error	Tipo de error	Qué usa	Como lo manejamos
`fs.stat`	asincrónico	archivo no encontrado	genuino	callback	manejamos el error del callback
`JSON.parse`	sincrónico	input incorrecto	genuino	`throw`	`try / catch`
`fs.stat`	asincrónico	`null` como input	programación	`throw`	arreglamos el bug

> `fs.stat` devuelve datos sobre un archivo en particular, está en la librería core `fs`.

Links copados:

- Joyent