

Centro Universitario de Ciencias Exactas e Ingenierías

Departamento de Ciencias Computacionales



“Otras herramientas para el manejo de errores”

Alumno: Verduzco González Carlos Ernesto

Código de alumno: 218744953

Carrera: Ingeniería en Computación

Materia: Computación Tolerante a Fallas

Profesor: López Franco Michel Emanuel

Sección: D06

NRC: 179961

Ciclo Escolar: 2022-A

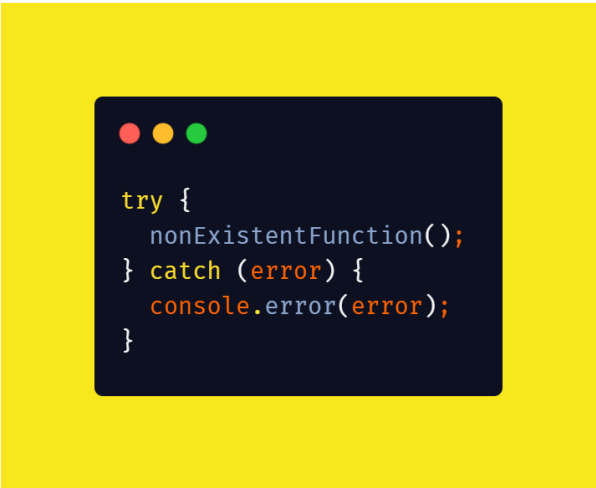
Fecha: 08/02/2022

Formas de tratar con errores en JavaScript

Herramientas dadas por el mismo lenguaje para el tratamiento de errores y el fortalecimiento del código:

try ... catch

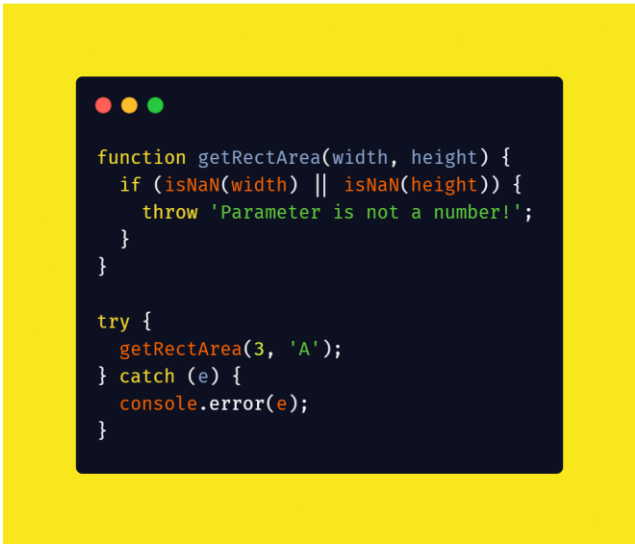
Con este tipo de tratamiento de error todo el código que está dentro del bloque de código que cubre el try este cubierto ante una posible excepción/error y ante cualquier de estos va a ejecutar lo que este dentro del catch, evitando posibles fallas durante la ejecución.



```
try {
  nonExistentFunction();
} catch (error) {
  console.error(error);
}
```

throw

El throw lo que hace es disparar una excepción, debe ser utilizado dentro de un try ... catch sino va a disparar el error de que no hay nadie que haga catch al error; mas que para tratar con errores es una forma de hacer errores personalizados ante un código ya prediseñado para ser tolerante a fallas.



```
function getRectArea(width, height) {
  if (isNaN(width) || isNaN(height)) {
    throw 'Parameter is not a number!';
  }
}

try {
  getRectArea(3, 'A');
} catch (e) {
  console.error(e);
}
```

Ejemplo

El código abajo es un ejemplo rápido para utilizar el try ... catch; en el código de abajo le pido al usuario que introduzca 2 números para posteriormente dividirlos, por la forma como funciona JavaScript la división no va a dar error sino que los si las variables a y b no son números entonces c va a ser igual a NaN (significa Not A Number) entonces para evitar un comportamiento no esperado entonces verifico si es NaN, si así es hago uso de throw para lanzar un error controlado que posteriormente despliego en consola.

Si todo va bien solamente imprimo el resultado en consola de la división.

```
a = prompt("Enter a number: ");
b = prompt("Enter another number: ");

try {
  c = a / b;
  if (isNaN(c))
    throw "a or b is not a number";
  console.log(c)
} catch (error) {
  console.log("Error: " + error);
}
```

```
> a = prompt("Enter a number: ");
b = prompt("Enter another number: ");

try {
  c = a / b;
  if (isNaN(c))
    throw "a or b is not a number";
  console.log(c)
} catch (error) {
  console.log("Error: " + error);
}
```

Error: a or b is not a number VM416:10

< undefined

> a

< "5"

> b

< "hola"

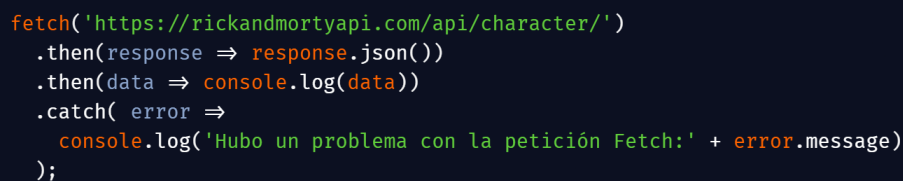
> c

< NaN

>

Manejo de errores en promesas.

Esta es la forma como se manejan posibles errores en la comunicación entre sistemas, esto es utilizado en JavaScript para manejar posibles errores en las llamadas a la API, algún error directamente en el servidor de la API o algún fallo de credenciales. Por ejemplo: El código siguiente está haciendo una llamada a la RickAndMorty API cuando se resuelve la petición la respuesta la parsea a JSON después la imprime en consola, si durante alguna parte del proceso hay un error se ejecuta el catch que imprime el error que hubo.




```
fetch('https://rickandmortyapi.com/api/character/')  
  .then(response => response.json())  
  .then(data => console.log(data))  
  .catch( error =>  
    console.log('Hubo un problema con la petición Fetch:' + error.message)  
  );
```

Las promesas han reemplazado a los callbacks como la forma nueva y mejorada de escribir código asíncronico en JavaScript.

Manejo de errores en Callbacks.

Los Callbacks son la forma más básica de entregar un error de forma asíncronica. El usuario le pasa una función, la devolución de llamada, y la invoca en algún momento más tarde, cuando se completa la operación asíncronica. El patrón habitual es que la devolución de llamada se invoca como devolución de llamada (err, result), donde solo uno de err y result no es nulo, dependiendo de si la operación se realizó correctamente o no.

Los Callbacks han existido durante años. Es la forma más antigua de escribir código JavaScript asíncronico. También es la forma más antigua de entregar errores de forma asíncronica. Pasas una función Callback como parámetro a la función de llamada, que luego invocas cuando la función asíncronica termina de ejecutarse; el patrón habitual se ve así:

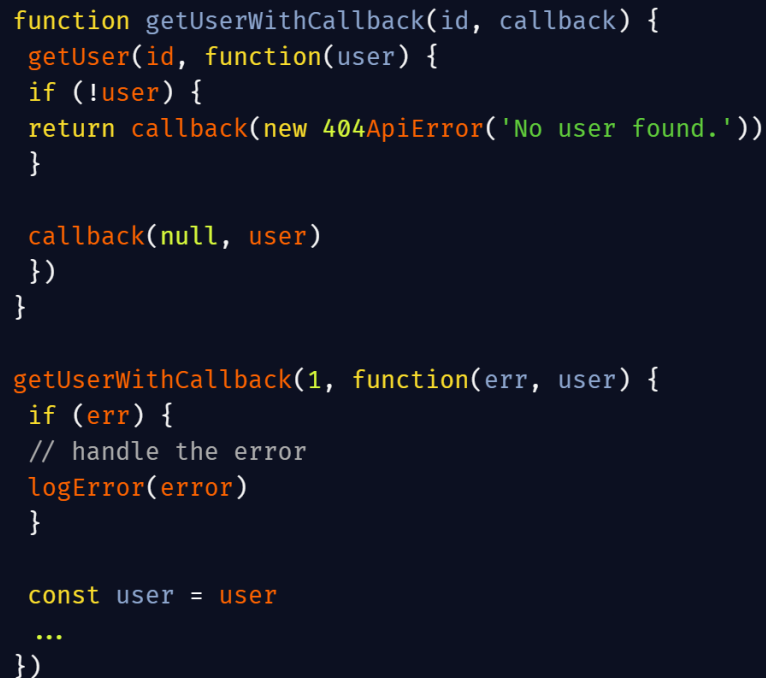


```
callback(err, result)
```

Otras herramientas para el manejo de errores

El primer parámetro del Callback es siempre el error.

Dentro de la función del Callback, primero verificará si el error existe y solo si es un valor no nulo, continuará ejecutando la función de devolución de llamada.



```
function getUserWithCallback(id, callback) {
  getUser(id, function(user) {
    if (!user) {
      return callback(new 404ApiError('No user found.'))
    }

    callback(null, user)
  })
}

getUserWithCallback(1, function(err, user) {
  if (err) {
    // handle the error
    logError(error)
  }

  const user = user
  ...
})
```

Manejo de errores en NodeJS con EventEmitter.

En algunos casos, no puede confiar en el rechazo de promesas o los callbacks. ¿Qué pasa si estás leyendo archivos de una transmisión? O buscar filas de una base de datos y leerlas a medida que llegan. Un caso de uso que veo a diario es la transmisión de líneas de registro y su manejo a medida que llegan.

No puede confiar en un error porque necesita escuchar eventos de error en el objeto EventEmitter.

En este caso, en lugar de devolver una Promesa, su función devolvería un EventEmitter y emitiría eventos de fila para cada resultado, un evento final cuando se informaron todos los resultados y un evento de error si se encuentra algún error.



```
net.createServer(socket => {  
  ...  
  
  socket  
    .on('data', data => {  
      ...  
    })  
    .on('end', result => {  
      ...  
    })  
    .on('error', console.error) // handle multiple errors  
})
```

Throw(manejo convencional con try...catch), Callback, Promesas o EventEmitter: ¿Qué patrón de tratar errores es el mejor?

Para errores operativos, debe usar Rechazos de promesas o un bloque try-catch con async / await. Desea manejar estos errores de forma asincrónica. Funciona bien y se usa ampliamente.

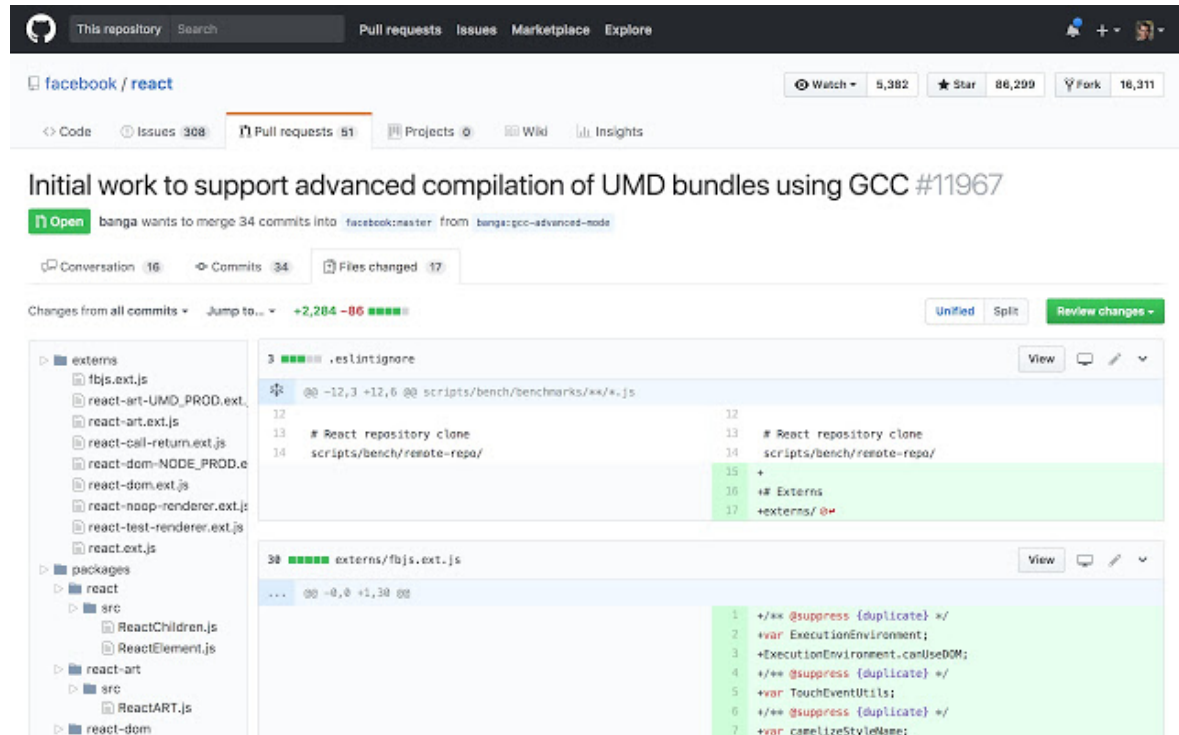
Si tiene un caso más complicado como el que se vio en la parte del EventEmitter, debería usar este mismo.

Desea lanzar errores explícitamente si es necesario desenrollar toda la pila de llamadas. Esto puede significar cuando se manejan errores del programador y desea que la aplicación se reinicie.

Otras herramientas para el manejo de errores

Pull Request / Code Reviews

Estas son utilizadas en cualquier compañía para tu encargado o compañeros supervisen tus aportes al código para así verificar que se siguieron las pautas del proyecto y evitar posibles errores de inexperiencia.



Pruebas Unitarias

Los test unitarios sirven para asegurarnos de que un bloque de nuestro código, función o clase, funcionan correctamente y abarca la mayoría de los casos de uso que se puedan dar. Además, proporciona robustez y calidad a nuestro código y confirma que funciona correctamente.

Escribir un test unitario no es algo complejo, pero sí que lleva su tiempo. Normalmente, si en realizar una funcionalidad se tarda 1 hora, para realizar los test de dicha funcionalidad se suele calcular multiplicando por tres el tiempo de desarrollo, ya que tenemos que abarcar, a ser posible, un mínimo del 90% de los posibles casos que se puedan dar: por ejemplo, cuando el código se ejecuta correctamente o cuando queremos controlar una excepción.

Todo ello conlleva unas acciones que, para tener un test correcto, hay que cubrir en la mayoría de las posibilidades o casos que se puedan dar en nuestro código.

Ejemplo.

En el siguiente ejemplo veremos como realizar una prueba unitaria a la siguiente función, que tiene el objetivo de retornar la descripción(atributo) de cierto objeto, cuando los posibles estados están previamente definidos en el map objectMapping:

```
const objectMapping = {
  ACTIVE: 'Activo',
  INACTIVE: 'Inactivo',
  OBSOLETE: 'Obsoleto',
};

function getObjectDescription(type) {
  if (!type) {
    return "El argumento 'type' no existe";
  }
  return objectMapping[type];
}

export default getObjectDescription;
```

Para la prueba unitaria tome la estructura básica de la mayoría de las librerías en JavaScript para realizar testing, agrupamos los test con el describe y después cada una de las pruebas con it; al final es probar cada parte de la funcionalidad con el objetivo de si llega a escalar el código el tener un forma rápida y confiable para asegurarnos que no alteramos el flujo de la aplicación.

```
import getObjectDescription from './ejemplo.js';

describe('Test unitarios de nuestro módulo "getObjectDescription"', () => {
  it('getObjectDescription to be truthy', () => {
    // Con toBeTruthy estamos diciendole a jest que esperamos que exista nuestro método
    expect(getObjectDescription()).toBeTruthy();
  });
  it("getObjectDescription('ACTIVE') to be 'Activo'", () => {
    const result = getObjectDescription('ACTIVE');
    expect(result).toBe('Activo'); // Ponemos 'Activo' porque es el valor de nuestro objeto
  });
  it("getObjectDescription('DRAFT') to be false", () => {
    const result = getObjectDescription('DRAFT');
    expect(result).toBeFalsy(); // toBeFalsy comprueba si el valor es nulo/undefined/false
  });
  it("getObjectDescription('') to be false", () => {
    const result = getObjectDescription('');
    expect(result).toBeFalsy(); // toBeFalsy comprueba si el valor es nulo/undefined/false
  });
  it("getObjectDescription() to be 'El argumento 'type' no existe'", () => {
    const result = getObjectDescription('');
    expect(result).toBe('El argumento 'type' no existe');
  });
});
```


Enlace al repositorio con el código

Enlace: <https://github.com/Catoras/Otras-herramientas-para-el-manejo-de-errores>

Conclusiones

Con lo que mas me quedo de esta pequeña investigación es con el hecho de haber aprendido más de los test unitarios, ya que a mi consideración las formas implícitas en el código no las suelo ver mucho en ambientes reales de producción ya que se supone que refuerzas el código que realizas para evitar los fallos porque al utilizar cosas como el try catch si se llega a producir un error al final es lo mismo ya que estas alterando el flujo del programa, entonces a pesar de no estar presentando a error catastrófico es lo equivalente a hiciste mal la chamba; a lo que si le encuentro mucho mas valor es el delimitar y separar el código de forma coherente y tener los respectivos test de cada funcionalidad, esto es mas tardado pero al final es lo mejor para la escalabilidad del proyecto.

Con lo anterior dicho yo considero que esta mal abusar de los mecanismos de código para tolerar problemas ya que estos al final solo son contenciones al bug; pero de igual forma el que se presentara el bug denota un fallo en la línea de producción y en el departamento de lo que si de verdad importa que es el testing.

Referencias Bibliográficas

- <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/try...catch>
- https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch
- <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/throw>
- https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Control_flow_and_error_handling
- <https://sematext.com/blog/node-js-error-handling/>
- <https://desarrolloweb.com/articulos/fetch-ajax-javascript.html#:~:text=Fetch%20es%20un%20nuevo%20API,del%20c%C3%B3digo%20en%20nuestras%20aplicaciones.>
- <https://www.paradigmadigital.com/dev/test-unitarios-javascript-introduccion/#:~:text=Un%20framework%20de%20pruebas%20unitarias%20es%20una%20herramienta%20que%20nos,ni%20en%20la%20propia%20aplicaci%C3%B3n.>