Tipos de errores: Operacionales vs. Errores de programador

Errores operacionales: Éstos errores representan problemas en el runtime. Estos errores son esperados en el runtime de node y deben ser tratados de manera correcta.
Errores operacionales más comunes:

- failed to connect to server
- failed to resolve hostname
- invalid user input
- request timeout
- server returned a 500 response
- socket hang-up
- system is out of memory

Errores de programador: Éstos errores son los que llamamos bugs. Representan problemas en el código.
Errores de programador más comunes:

- called an asynchronous function without a callback
- did not resolve a promise
- did not catch a rejected promise
- passed a string where an object was expected
- passed an object where a string was expected
- passed incorrect parameters in a function

Error Object

El objeto error es un objeto del runtime de node. Da una serie de información sobre un error cuando ocurre.
Un error básico se ve así:

```
const error = new Error("An error message")
console.log(error.stack)
```

Tiene un campo error.stack que brinda un trazo mostrando de donde viene el error. También lista todas las funciones que fueron llamadas antes de la ocurrencia del error. El error.stack también muestra el error.message.

Técnicas de manejo de errores

1. **Bloque try…catch**

El bloque try rodea el código donde puede ocurrir un error.
El bloque catch maneja las excepciones.

Ejemplo:

```
var fs = require('fs')

try {
const data = fs.readFileSync('/Users/Kedar/node.txt')
} catch (err) {
  console.log(err)
}

console.log("an important piece of code that should be run at the end")
```

Y se obtiene la salida:

```
$node main.js
{ Error: ENOENT: no such file or directory, open '/Users/Kedar/node.txt'
    at Error (native)
    at Object.fs.openSync (fs.js:641:18)
    at Object.fs.readFileSync (fs.js:509:33)
    at Object.<anonymous> (/home/cg/root/7717036/main.js:4:17)
    at Module._compile (module.js:570:32)
    at Object.Module._extensions..js (module.js:579:10)
    at Module.load (module.js:487:32)
    at tryModuleLoad (module.js:446:12)
    at Function.Module._load (module.js:438:3)
    at Module.runMain (module.js:604:10)
  errno: -2,
  code: 'ENOENT',
  syscall: 'open',
  path: '/Users/Kedar/node.txt' }
an important piece of code that should be run at the end
```

2. Callbacks

Una callback function (generalmente usado para código asíncrono) es un argumento a la función en la cual implementamos el manejo de errores.
El propósito de una callback function es chequear los errores antes que el resultado de la función primaria sea usado.
Sintaxis de una callback function:

```
function (err, result) {}
```

El primer argumento es para un error, y el segundo para el resultado. En caso de error, el primer atributo va a contener el error y el segundo atributo va a ser undefined, y viceversa.

Ejemplo de intentar leer un archivo aplicando esta técnica:

```
const fs = require('fs');

fs.readFile('/home/Kedar/node.txt', (err, result) => {
  if (err) {
    console.error(err);
    return;
  }

  console.log(result);
});
```

Ejemplo de un callback implementado en una función definida por el usuario:

```
const udf_double = (num, callback) => {
  if (typeof callback !== 'function') {
    throw new TypeError(`Expected the function. Got: ${typeof callback}`);
  }

  // simulate the async operation
  setTimeout(() => {
```

```
    if (typeof num !== 'number') {
      callback(new TypeError(`Expected number, got: ${typeof num}`));
      return;
    }

    const result = num * 2;
    // callback invoked after the operation completes.
    callback(null, result);
  }, 100);
}

// function call
udf_double('2', (err, result) => {
  if (err) {
    console.error(err)
    return
  }
  console.log(result);
});
```

3. Promises:

Promise es una forma contemporánea de manejar errores y son preferibles respecto a los callbacks.

Ejemplo:

```
const udf_double = num => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (typeof num !== 'number') {
        reject(new TypeError(`Expected number, got: ${typeof num}`));
      }

      const result = num * 2;
      resolve(result);
    }, 100);
  });
}
```

En la función, vamos a retornar una “promise”. Pasamos 2 argumentos mientras definimos el objeto Promise:

- Resolve: Usado para resolver promises y entregar resultados
- Reject: Usado para reportar/throw errores.

Así ejecutamos la función enviando el input:

```
udf_double('8')
  .then((result) => console.log(result))
  .catch((err) => console.error(err));
```

Podemos también, usar una utilidad como util.promisify() para convertir código basado en callback a una Promise.
Ejemplo:

```
const fs = require('fs');
const util = require('util');

const readFile = util.promisify(fs.readFile);

readFile('/home/Kedar/node.txt')
  .then((result) => console.log(result))
  .catch((err) => console.error(err));
```

3.5. Async/await:

Es un agregado sintáctico para aumentar promises. Provee una estructura sincrónica a un código asíncrono.
El retorno de una función async es una Promise. El await espera a que la promesa sea resolved o rejected.
Ejemplo:

```
const fs = require('fs');
const util = require('util');

const readFile = util.promisify(fs.readFile);

const read = async () => {
  try {
    const result = await readFile('/home/Kedar/node.txt');
    console.log(result);
  } catch (err) {
    console.error(err);
  }
};

read()
```

4. Event emitters

Podemos usar la clase EventEmitter del módulo ‘events’ para reportar errores en escenarios complejos. Vamos a usarlo en un ejemplo donde recibimos información y chequeamos si es correcta. Queremos chequear si los primeros 6 índices son integers, excluyendo el índice 0. Si alguno de los 6 índices no es un integer, emitimos un error.
Ejemplo:

```
const { EventEmitter } = require('events'); //importing module

const getLetter = (index) =>{
    let cypher = "*12345K%^*^&*" //will be a fetch function in a real scenario which will fetch
a new cypher every time
    let cipher_split = cipher.split('')
    return cipher_split[index]
}

const emitterFn = () => {
  const emitter = new EventEmitter(); //initializing new emitter
  let counter = 0;
  const interval = setInterval(() => {
    counter++;

    if (counter === 7) {
      clearInterval(interval);
      emitter.emit('end');
    }

    let letter = getLetter(counter)

    if (isNaN(letter)) { //Check if the received value is a number
      (counter<7) && emitter.emit(
        'error',
        new Error(`The index ${counter} needs to be a digit`)
      );
      return;
    }
    (counter<7) && emitter.emit('success', counter);

  }, 1000);

  return emitter;
}

const listner = emitterFn();
```

```
listner.on('end', () => {
  console.info('All six indexes have been checked');
});

listner.on('success', (counter) => {
  console.log(`${counter} index is an integer`);
});

listner.on('error', (err) => {
  console.error(err.message);
});
```

Output:

```
1 index is an integer
2 index is an integer
3 index is an integer
4 index is an integer
5 index is an integer
The index 6 needs to be a digit
All six indexes have been checked
```

Manejar errores (handling errors)

1. **Reintentar la operación**

A veces, los errores pueden ser causados por un sistema externo para requests válidas. Por ejemplo, recibir un error 503 al hacer un fetch de una API, el cual es causado por sobrecarga o un error de red. Dada esa situación, el servicio debería volver a funcionar en unos pocos segundos, y reportar un error no es lo mejor, así que se reintenta la operación.

2. **Reportar el fallo al cliente**

Al recibir un input erróneo del cliente, la mejor opción es finalizar el resto del proceso y reportarlo al cliente.

3. **Crashear inmediatamente**

En caso de errores irrecuperables, crashear el programa es la mejor opción