**Error handling in node.js -** [Success Ibekwe](https://medium.com/@successibekwe047?source=post_page-----ef5cbfa59992--------------------------------) Jul 23, 2022

Even with the most experienced programmer you cant get errors out completely from your program. But because JavaScript is a loosely-typed language, most of the errors would be gotten during runtime since It does not give a lot of compile-time errors.

Some of the scenarios where you will get a runtime error are when trying to access an undefined variable and when trying to access an undefined method etc.

This makes Error handling a mandatory step in application development.

So in this article, we are going to look at what error handling is and the proper way of handling these errors.

Buckle up as we delve right in.

**What is error handling:**

Errors are the problems that occur in our program which cause the program to malfunction, The process of handling these errors or exceptions that takes place during runtime is what we call Error handling. Programming errors are also known as bugs.

**Why is error handling important?**

**Error handling** helps you gain resilience, it helps make your code more secure and prevents your system from breaking down due to unexpected crashes. With Proper error handling, you can make your apps more robust which helps to create a better user experience and also helps to improve productivity.

**Types of errors in JavaScript.**

Talking about error handling in javascript. it’s important to note that there are two types of error Runtime and Syntax error.

**Runtime Error:**

Runtime errors are also known as operational errors and they include the problems that occur when our application is in production, they occur from time to time due to some external factors most likely when there is a database server time out or trying to input SQL queries in an input field.

Some of the examples of Runtime errors include:

* Failed to connect to the server
* Request time out
* Invalid user input
* The System is out of memory
* Failed to resolve hostname
* The server returned a 500 response

**Syntax Error:**

This is also known as programmer error, and they occur when the code is been interpreted by a JavaScript parser.

When a syntax error occurs, it does not affect all the code, it only affects the code that is on the same thread. An example syntax error can be missing to either open or close your parentheses or square brackets.

Another example could be missing to put a comma or colon in the object. Whenever any of these errors are encountered in your code it will cause the syntax to be invalid and stop your program.

So in other to avoid these types of errors. You should constantly check your code and look them up for any missing comma or closing parenthesis.

Though this approach is not effective enough and can take a long time. So you can approach it using the IDE tools or plugins. These tools help to automatically check your code and look for syntax problems.

Some of the most popular tools that will help detect syntax errors are [eslint](https://eslint.org/" \t "_blank), [jshint](http://www.jshint.com/" \t "_blank), and [jslint](http://www.jslint.com/" \t "_blank).

**Try Catch Syntax**

*Try-catch* declaration is basically used to handle runtime errors in node.js, they are kind of similar to the *if…else*declaration but comes with more advantage than the *if…else.*

The *try-catch*is made up of two main blocks of code. The *try block* which is where the code is executed and tested for errors while the *catch block* is where you receive an object of type Error, via a parameter in the function.

If the code in the try block throws an exception, the code in the catch block will be executed.

try {  
 tryCatchTest();  
} catch (error) {  
 console.error(error);  
 // expected output: ReferenceError: tryCatchTest is not defined

We can extend the *try-catch* by introducing the *finally* clause. The*finally* clause is statements that are executed after the try statement completes. These statements are executed regardless of whether an exception was thrown or caught.

try {  
 try\_statements  
}  
catch (error) {  
 catch\_statements  
}  
finally {  
 finally\_statements  
}

**Error Object:**

Whenever there is a runtime error, it leads to objects being created and thrown. JavaScript creates this object of type **Error** and sends it as an argument to catch.

The error object is composed of two main properties by default:

1. name
2. message

**name:**The name property represents a name for the type of error. By default, they are given the name "Error"… For example, a syntax error in parsing a JSON would throw an exception of type ***SyntaxError***.

**message:**The is composed of a text message containing more details of the error.

There is another property known as **stack property**though it’s a non-standard property and may not be available in all environments.

The Stack property helps us in with tracing which of the function was called, in what order, from which line and file, and with what arguments.

In some environments where you do error handling in JavaScript, you will also have the **stack** property. The information it provides is quite useful for debugging.

Another instance is to omit the error argument in the catch function. This comes in handy if we don’t need to know the details of the error.

example:

try {   
 // your code here   
} catch {   
 // error handling here without the argument  
}

Error object is the basis of some of the native types of errors that can be thrown in JavaScript. Some of these types include:

**SyntaxError**

When a syntax error occurs when interpreting the code syntaxError is thrown.

**URIError(Uniform Resource Identifier)**  
This is thrown when an error occurs in the handling of URI, for example, sending invalid parameters in decode URI() or encode URI().

**RangeError**  
When a value out of the range of allowed values occurs. For example, a string value is a number array.

**ReferenceError**  
This is thrown when using a variable that has not been declared

**TypeError**  
This type of error is thrown when we use an operator or argument that is of a different type from what is defined in the function.

The above-listed javaScript native errors are very useful when we have no idea of ​​the error that our application will encounter. But when there is a business rule defined, we can create our own type of error which we can throw if the need arises

**Type of errors**

Below are the types of errors

**Programmer errors:**

Let’s first take a look at handling programmer errors. As we explained while defining the programmer error, they are unexpected so the best way to handle them is to crash it and gracefully restart it.

process.on('uncaughtException', (error: Error) => {  
 errorHandler.handleError(error);  
 **if** (!errorHandler.isTrustedError(error)) {  
 process.exit(1);  
 }  
});

**Operational errors**

To properly handle errors then we must say goodbye to callbacks. If you have used callbacks for dealing with errors in async node.js you would have noticed that it comes with a serious drawback with a callback.

It is very hard to follow the code flow, since it causes us to search for errors even down to the nested one which is not what you would want.

**Best Practices for proper error handling**

There are many ways to handle errors depending on your use case and code structure. Below are the general ways you can handle errors.

So we would be going through some proper ways to handle these errors.

**1. Using promises**

Using promises would be the first we are going to talk about. With promises, you can handle errors by chaining together different operations. but since you don’t normally know when this error occurs.

It is advised you handle errors in each function called (doSomethingX) and by throwing a new error inside the catch it will call the catch outside.

This is what the code would look like:

**doSomethingA ( ). then ( doSomethingA ). then ( doSomethingC ). catch ( err => console . error ( err ) )const doSomethingA = ( ) => {//...try {//...} catch ( err ) {//... handle the error locallythrow new Error ( err . message )}//...}\**To handle the errors locally you can create a function in each ***then*** and handle it there**doSomethingA ( ). then ( ( ) => {return doSomethingB ( ) . catch ( err => {// handle the errorthrow err // break the chain!} )} ). then ( ( ) => {return doSomethingA ( ) . catch ( err => {// handle the errorthrow err // break the chain!} )} ). catch ( err => console . error ( err ) )**

**2. Using the async/await approach:**

**const** yourAsyncOptions = **async** () => {  
 **try** {   
 **const** item1 = **await** option1()   
**const** item2 = **await** option2(item1)  
**const** item3 = **await** option3(item2)  
 **return** **await** item4(item3); }  
 **catch** (error) {   
 console.error(error)   
 }  
 **finally** {   
 **await** chooseAnyOption()   
 }  
 }

**1. using the error object:**

We talked about the error object but here we are going to use it to handle our error since it gives us clear information about errors.

**class** **BaseError** **extends** **Error** {  
   
 **constructor**(name, httpCode, description, isOperational) {  
 **super**(description);  
 Object.setPrototypeOf(**this**, **new**.target.prototype);  
   
 **this**.name = name;  
 **this**.httpCode = httpCode;  
 **this**.isOperational = isOperational;  
   
 Error.captureStackTrace(**this**);  
 }  
}*//free to extend the BaseError*  
**class** **APIError** **extends** **BaseError** {  
 **constructor**(name, httpCode = HttpStatusCode.INTERNAL\_SERVER, isOperational = true, description = 'internal server error') {  
 **super**(name, httpCode, isOperational, description);  
 }  
}

For the sake of simplicity, we only used a few HTTP status codes but feel free to add more later.

**export** enum HttpStatusCode {  
 OK = 200,  
 BAD\_REQUEST = 400,  
 NOT\_FOUND = 404,  
 INTERNAL\_SERVER = 500,  
}

There is no need to extend BaseError or APIError, but it is okay to extend it for common errors according to your needs and personal preferences.

**class** **HTTP400Error** **extends** **BaseError** {  
 **constructor**(description = 'bad request') {  
 **super**('NOT FOUND', HttpStatusCode.BAD\_REQUEST, true, description);  
 }  
}

So how do you use it? Just throw this in

...  
const user = **await** User.getUserById(1);  
**if** (user === null)  
 **throw** **new** APIError(  
 'NOT FOUND',  
 HttpStatusCode.NOT\_FOUND,  
 true,  
 'detailed explanation'  
 );

**Using the centralized approach:**

With the centralized approach to handling your error, you could get rid of possible code duplication.

When sending notifications to system admins, transferring events to a monitoring service like Sentry.io, and logging them. The centralized error-handling component is responsible for making the caught errors understandable.

These caught errors can be transferred to middleware. You can distinguish between error types and send them to the centralized error-handling component right in the middleware

let’s check out some codes.

...  
try {  
 userService.addNewUser(req.body).then((newUser) => {  
 res.status(200).json(newUser);  
 }).catch((error) => {  
 next(error)  
 });  
} **catch** (error) {  
 next(error);  
}  
...app.use(**async** (err, req, res, next) => {  
 **if** (!errorHandler.isTrustedError(err)) {  
 next(err);  
 }  
 **await** errorHandler.handleError(err);  
});

let's look at the centralized code

**class** **ErrorHandler** {  
 **async** handleError(err) {  
 **await** logger.error(  
 'Error message from the centralized error-handling component',  
 err,  
 );  
 **await** sendMailToAdminIfCritical();  
 **await** sendEventsToSentry();  
 }  
   
 public isTrustedError(error) {  
 **if** (error **instanceof** BaseError) {  
 **return** error.isOperational;  
 }  
 **return** false;  
 }  
}  
**export** **const** errorHandler = **new** ErrorHandler();

**Winston morgan logger:**

Using a logger like Winston morgan that is customizable is a good practice. it increases visibility thereby making it easy to track the errors and handle them. This generally helps you to save time.

Using this logger provides logging at multiple different levels and in a more formatted way, with clear colors, and according to the runtime environment, it logs into different output media.

With Winston morgan logger good you can watch and query logs by using Winston’s built-in APIs.

Let's take a look at what a customized Winston morgan logger looks like:

**const** customLevels = {  
 levels: {  
 trace: 5,  
 debug: 4,  
 info: 3,  
 warn: 2,  
 error: 1,  
 fatal: 0,  
 },  
 colors: {  
 trace: 'white',  
 debug: 'green',  
 info: 'green',  
 warn: 'yellow',  
 error: 'red',  
 fatal: 'red',  
 },  
};**const** formatter = winston.format.combine(  
 winston.format.colorize(),  
 winston.format.timestamp({ format: 'YYYY-MM-DD HH:mm:ss' }),  
 winston.format.splat(),  
 winston.format.printf((info) => {  
 **const** { timestamp, level, message, ...meta } = info;  
   
 **return** `${timestamp} [${level}]: ${message} ${  
 Object.keys(meta).length ? JSON.stringify(meta, null, 2) : ''  
 }`;  
 }),  
);**class** **Logger** {  
 private logger: winston.Logger;  
   
 **constructor**() {  
 **const** prodTransport = **new** winston.transports.File({  
 filename: 'logs/error.log',  
 level: 'error',  
 });  
 **const** transport = **new** winston.transports.Console({  
 format: formatter,  
 });  
 **this**.logger = winston.createLogger({  
 level: isDevEnvironment() ? 'trace' : 'error',  
 levels: customLevels.levels,  
 transports: [isDevEnvironment() ? transport : prodTransport],  
 });  
 winston.addColors(customLevels.colors);  
 }  
   
 trace(msg: any, meta?: any) {  
 **this**.logger.log('trace', msg, meta);  
 }debug(msg: any, meta?: any) {  
 **this**.logger.debug(msg, meta);  
 }  
   
 info(msg: any, meta?: any) {  
 **this**.logger.info(msg, meta);  
 }  
   
 warn(msg: any, meta?: any) {  
 **this**.logger.warn(msg, meta);  
 }  
   
 error(msg: any, meta?: any) {  
 **this**.logger.error(msg, meta);  
 }  
   
 fatal(msg: any, meta?: any) {  
 **this**.logger.log('fatal', msg, meta);  
 }  
}**export** **const** logger = **new** Logger();

**Summary**

If you’re a software engineer, errors are always going to be there, you should befriend them and know how to handle them properly in development to avoid disaster in production.

In the article, we explored what error handling is and the proper way of handling these errors.