Updated - Comprehensive Review of The Error Class

In this document we undertake a comprehensive review of the Error class of JavaScript. This class is also used in NodeJS. [1]

Any application running in NodeJS can experience 4 categories of errors:

- Standard JS errors such as **EvalError**, **SyntaxError**, **RangeError**, **ReferenceError**, **TypeError** and **URIError**. [1][3][4][5][6][7][8]
- System errors triggered by underlying operating system constraints such as attempting to open a file that does not exist or attempting to send data over a closed socket. [1]
- User-specified errors (CustomError) triggered by application code. [1]
- **AssertionError**(s) are a special type of error that can be triggered when NodeJS detects an exceptional logic violation that should never occur. [1]

All JavaScript and system errors raised by Node.js inherit from, or are instances of, the standard JavaScript **Error** class and are **guaranteed to provide at least** the properties available on that class. [1]

NodeJS supports several methods for handling and intercepting errors that occur during an application's execution. How these errors are reported and intercepted depends on the type of **Error** and the style of API that is called. [1]

All JavaScript errors are handled as exceptions that **immediately** generate and throw an error using the standard JavaScript throw mechanism. If an error is **thrown** using the **throw** mechanism, it must be handled immediately or the NodeJS process will exit. [1]

Synchronous functions usually use **throw** to report errors. [1]

Errors occurring in **Asynchronous** functions can be reported as follows:

- Some async functions return a promise. Any returned promise can be rejected.
- Most async functions that accept a callback function as an argument will also accept an error as the first argument to it. If the argument passed as an error is not null or if it is an instance of **Error**, then an error has indeed occurred and it should be handled. [1]
- When an async function is called on an object that is an EventEmitter, then errors can be routed to that object's error event. [1]
- A few async methods in NodeJS still use **throw** to raise exceptions that must be handled using **try…catch.** [1]
- The **error** event mechanism is common for **stream-based** and **event-emitter** based APIs, both of which are a series of async operations over time and not a single operation that can pass or fail. [1]
- For all EventEmitter objects, if a handler for the error event isn't provided, then the error will be thrown and be reported as an exception and make the NodeJS application crash, unless a handler is registered for the UncaughtException event. Errors generated in this way cannot be intercepted using **try…catch** as they are thrown after the calling code has already exited. [1]

The Error Class

An Error object doesn't represent the specific reason as to why an error occurred. They capture a **stack trace** telling us where the Error was **instantiated** (occurred). **Any error** that occurs in NodeJS and this includes system errors & JS errors will either be instances of or inherit from the Error class. [1]

- The constructor of the Error class. It commonly accepts the **message** and has **options**. One common option is the **cause**. If the **cause** option is provided, then it is assigned to the **error.cause** property. [1]

- The **Error.captureStackTrace** creates a **.stack** which contains the location in the code where the captureStackTrace was called. [1]

- The **Error.stackTraceLimit** property specifies the no. of stack frames. The default value is 10. [1]

- The **error.cause** property, if present, records the underlying cause of the error. It is used when catching an error and throwing a new one with a different message or code in order to still have access to the original error. The error.cause property is typically set by calling new **Error(message, { cause })**. It is not set by the constructor if the cause option is not provided. [1]

- The **error.code** property is a string label that identifies the kind of error. It is the most stable way to identify an error. Error codes never change even between major versions of NodeJS. [1]

- The **error.message** property is the string description of the error. The message passed to Error's constructor will also appear in the first line of the **stack trace**. However, any change to the message will not be reflected in the first line of the stack trace if the change is made after creating the Error object. [1]

- The **Error.stack** is a property that describes the point in the code where the error was instantiated. Each line beginning with **at** is a stack frame. Each frame describes a call site within the code that led to the error being generated. **V8** tries to describe the name for every function (through function name, file name or variable name). However, sometimes it won't be able to find a name, and **in that case, only location** information will be displayed there. [1]

- The number of frames captured by the stack trace is bounded by the smaller of **Error.stackTraceLimit.** [1]

Table of Errors that Inherit from the Error Class

Error Name	What It Does	Operational	Identification
AssertionError	Shows assertion failure.	No	error.code [2]
RangeError	Shows value out of range for a function. Used by Node as a form of validation.	No	RangeError.prototype.name [5]
ReferenceError	Shows that we are trying to access a variable that is not defined. Shows erroneous program.	No	ReferenceError.prototype.name [6]
SyntaxError	Shows that a program isn't valid JS. Only occurs due to a result of code evaluation. Shows erroneous program.	No	SyntaxError.prototype.name [4]
SystemError	System errors are generated when exceptions occur. An exception occurs when an **OS constraint is violated**. Eg - Trying to read a nonexistent file. It covers FileSystem & Network Errors.	No	error.code [1]
TypeError	Indicates that a variable is not an allowable type. Eg - Passing argument of wrong type. NodeJS uses it for validation.	Can be if the said function is an exposed API endpoint.	TypeError.prototype.name [7]
EvalError	Caused when we try to execute invalid JS code in the **eval()** function	No	EvalError.prototype.name [3]
URIError	Wrong use of global URI function.	No	URIError.prototype.name [8]

Error vs. Exception:
A JS **exception** is a value thrown as a result of an invalid operation or as the target of a **throw** statement. These values need not always be instances of the **Error** class or classes that inherit from **Error**. However, all exceptions thrown by NodeJS or the JS runtime will be instances of **Error**. Some exceptions are unrecoverable and the NodeJS process will always crash due to that.

OpenSSL Errors:
Errors originating in **crypto** or **tls** are of class **Error**. They have some additional properties other than **standard** and **code**. They are listed below:

- error.opensslErrorStack -> An array of errors that can give context to where in the OpenSSL library an error originates from.
- error.function -> The OpenSSL function the error originates in.
- error.library -> The OpenSSL library the error originates in.
- error.reason -> A human-readable string describing the reason for the error.

REFERENCES
[1] https://nodejs.org/api/errors.html

[2] https://nodejs.org/api/assert.html#class-assertassertionerror

[3]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/EvalError

[4]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SyntaxError

[5]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RangeError

[6]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/ReferenceError

[7]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/TypeError

[8]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/URIError