Error Handling Best Practices

Terminology

Error

An instance of the Error class.

Errors can be constructed and passed to another function or thrown.

Exception

An event that occurs that disrupts the normal flow a program and that requires special processing to handle.

Three Main Error Delivery Methods

Throw the Error

Turns the error into an exception.

Pass the Error to a Callback

The callback function would be provided specifically for handling errors and results of asynchronous operations.

Emit an Error Event

Use the event emitter to broadcast that an error has occurred.

Types of Errors

Operational Errors

Runtime problems that occur in correctly written code and not the result of bugs.

These sorts of errors must be handled properly to avoid a bad user experience.

When these sorts of errors are handled correctly, it doesn't necessarily indicate a bug or even a serious problem.

Operational Error Examples

- Things wrong with the system itself: out of memory, too many files open, file not found
- System configuration problems: no route to a remote host
- Network errors: socket hangup, no network connection
- Remote service errors: 500 error, invalid user input

Programmer Errors

Bugs in the code that we write.

These sorts of errors can't be properly handled because the correct solution is to fix the bug in the code.

Programmer Error Examples

- Trying to read a property from an undefined variable.
- Called an asynchronous function without a callback.
- Passing the wrong data type for what is expected.
- A typo in a variable name.

Shared Causes

It's not always clear cut, sometimes there can be both operational and programmer errors as part of the same problem.

Handling Operational Errors

Proper Error Handling

Proper handling of errors is a necessary part of building robust applications and can't be tacked on as an afterthought on a program that doesn't have error handling implemented.

Proper error handling cannot be centralized. It must be fine-grained, thus you need to know what caused the error.

Any code that does anything that might fail need to have defined behaviors for when failure occurs.

Potential Approaches

- Deal with the failure directly.
- Propagate the failure to the client.
- Retry the operation.
- Let the program crash.
- Log the error and move on.

Deal with the Failure

If it's clear what you can do to move forward after an error occurs, you can simply address the error by doing whatever is necessary to fix it before continuing on with operations as normal.

Propagate the Error

If you don't know how to deal with the error that occurred, another option is to abort the operation and deliver the error back to the client.

This is a valid approach for errors for which there's a reasonable expectation that the cause of the error won't be resolved soon.

Depending on what the operation is, some cleanup might be necessary.

Retry the Operation

Retrying the operation after a short time is a valid error handling response for network and remote service errors if you have reason to believe that the error is a result of platform failure rather than programming failure.

Use caution however, as retrying the operation might not always be a valid procedure, such as if you're several remote calls in.

When taking this option, be sure to thoroughly document that you may retry the operation, how many times you'll retry before taking another error handling approach, and the time waited between retries.

Let the Program Crash

When you have an error that are programmer errors that appear as operational errors, logging an error message then crashing is a valid option.

Another situation where crashing the program is a valid option is for operational errors from which there is no recovery.

In production environments, alerts will be in place to notify regarding crashes so that they can be dealt with.

Log the Error and Go On

There are cases where there is nothing to be done about an error, no operation to retry or abort, but also no reason to crash the program. In these situations, simply logging the error and allowing for normal operation to continue is a valid response.

That the error occurred should at the very least be logged.

Handling Programmer Errors

Don't

Don't Expanded

Programmer errors happen in code that is supposed to do something but is broken (for example, a mistyped variable name). That problem can't and shouldn't be caught using normal error handling techniques.

The program should crash, the erroneous code should be corrected, and the application redeployed.

Attempting to handle the error and recover could lead to unexpected behavior from everything interacting with the erroneous code. These sorts of errors can be difficult to track down and debug.

What Could Go Wrong?

- A variable shared across requests could be left null, undefined or otherwise storing an invalid value, so subsequent requests are referencing incorrect data.
- A database connection may be leaked, reducing the number of requests able to be handled concurrently.
- A connection may be left in an authenticated state for a subsequent connection, providing users with access to other people's data.
- A socket may be left open when it should be closed.
- Memory references to variables that are no longer needed might still be kept, which could cause the system to run out of memory or spend too much time in garbage collection, causing poor performance.

The best way to recover from programmer errors is to crash immediately.

Crash Recovery

Programs can be run bundled with a restarter that will automatically restart the program upon a crash. When this is in place, crashing is actually the fastest way of restoring service in the face of programmer error.

There will be temporary disruption, but that disruption is better than difficult to debug data errors being propagated.

Fixing these errors is then given top priority, and code is then redeployed.

Crash Recovery Cont.

If you're properly testing, unit and otherwise, bugs in production should be relatively rare.

Client programs should be designed and written to deal with server failure by reconnecting and retrying requests (i.e. treating the issue like an operational error).

If the program is crashing and restarting so often that these disruptions are a frequent problem, the server-side code is too buggy to be properly in production.

Error Delivery In-Depth

Three Basic Approaches

- Throw an Error Synchronously
- Callbacks to Handle the Error
- Emit an Error Event

Throw an Error Syncronously

When using try/catch blocks, an error can be thrown in the same context where the function was called.

Callbacks to Handle the Error

The most basic way of delivering an error asynchronously. The user passes a function to be run later when the asynchronous operation completed.

The usual pattern for Node for supplying data parameters to callbacks is error first.

Emit an Error Event

Used in more complicated cases, such as performing a series of operations where multiple errors may occur.

What Type to Use?

If the function is synchronous, you will likely want to use throw. Otherwise, for asynchronous functions, you'll want to use one of the asynchronous error handling methods (callbacks or event emitter).

Most operational errors occur in asynchronous functions, and for a majority of these cases, passing the error to a callback is best approach.

For synchronous errors, throwing an error is a lot more common than simply returning it.

Synchronous vs Asynchronous

For a given function, if any operational error can be delivered asynchronously, then all operational errors that could occur in that function should be delivered asynchronously.

A function may deliver operational errors synchronously (by throwing) or asynchronously (by passing them to a callback or emitting an error event), but it should not do both.

Looking At Errors

Delivering Errors