Robust Error Handling in Node.js

Who are you?

- I'm Lewis, I like JavaScript (and other things)
- Made Bee-Queue, Redis-backed job queue for Node.js
  - Like Celery, Resque, Kue, Bull
- Worked on raven-node, Sentry's Node error reporting SDK
  - It captures and reports about 100 million errors per week

Why should you want robust error handling in Node?

This talk is for you if…

You've ever written code like this:

```
fs.readFile('myFile.txt', { encoding: 'utf8' }, function (err, data) {
  console.log(data);
});
```

Or this:

```
http.get(myUrl, function (res) {
  doSomethingWith(res);
});
```

Or even this:

```
try {
  data = JSON.parse(userInput);
} catch (e) {
  // this should never happen
}
doSomethingWith(data);
```

"this should never happen"

Famous last words

We can do better!

This is a little better:

```
fs.readFile('myFile.txt', { encoding: 'utf8' }, function (err, data) {
  if (err) return console.error(err);
  doSomethingWith(data);
});
```

But we can do a lot better

Overview of topics

- Background and how Node is special/different
- Handle what you can, avoid what you can't
- The robust game plan to follow
- Exceptions, Callbacks, Promises, EventEmitters, and more
- Catching, reporting, shutting down, restarting gracefully

Error Mechanisms in Other Languages

- Python: `try/except` and `raise`
- Ruby: `begin/rescue` and `raise`
- PHP: `try/catch` and `throw`
- Lua: `pcall()` and `error()`

So Node has some similar thing, right?

Well, sure, this works:

```
try {
  throw new Error('boom!');
} catch (e) {
  console.log('Aha! I caught the error!');
}
```

Well, sure, this works:

```
try {
  throw new Error('boom!');
} catch (e) {
  console.log('Aha! I caught the error!');
}
```

```
$ node try-catch.js
```

Well, sure, this works:

```
try {
  throw new Error('boom!');
} catch (e) {
  console.log('Aha! I caught the error!');
}
```

```
$ node try-catch.js
> Aha! I caught the error!
```

But where try-catch comes up short:

```
try {
  setTimeout(function () {
    throw new Error('boom!');
  }, 0);
} catch (e) {
  console.log('Aha! I caught the error!');
}
```

But where try-catch comes up short:

```
try {
  setTimeout(function () {
    throw new Error('boom!');
  }, 0);
} catch (e) {
  console.log('Aha! I caught the error!');
}
```

```
$ node try-settimeout.js
```

But where try-catch comes up short:

```
try {
  setTimeout(function () {
    throw new Error('boom!');
  }, 0);
} catch (e) {
  console.log('Aha! I caught the error!');
}
```

```
$ node try-settimeout.js
/Users/lewis/dev/node-error-talk/try-settimeout.js:3
    throw new Error('boom!');
    ^
Error: boom!
    at Timeout._onTimeout (/Users/lewis/dev/node-error-talk/try-settimeout.js:3:11)
    at ontimeout (timers.js:365:14)
    at tryOnTimeout (timers.js:237:5)
    at Timer.listOnTimeout (timers.js:207:5)
```

Why?

- Try-catch is synchronous, setTimeout is asynchronous
- Callback is queued and will throw the error later
- Catch block is no longer waiting to catch the exception
- Run-to-completion semantics & event loop are behind this

How Node is Different

- Other languages:
  - Each process handles one request at a time
  - Everything is synchronous, try/catch works fine
  - Easy to keep one request from blowing everything up
- Node: single process, cooperative concurrency, asynchronous I/O
  - Handles multiple requests at the same time in the same thread
  - Try/catch doesn't work well in asynchronous world
  - Any one request blowing up could mess up the others

What this means for us

- We need other mechanisms to handle asynchronous errors
- We can't `try/catch` and throw exceptions for everything
- An error in one request can take down the entire server
- We have to be extra careful to keep things online
- Our program can end up in unknown states
    - Only correct thing to do might be shut down!

Errors vs Exceptions

What's the difference, anyway?

Errors

Error is just a special class in JavaScript
- You can pass an `Error` object around like any other value
- Runtime errors throw an `Error` object
- Has a `message` and a `stack` property
- Also `RangeError`, `ReferenceError`, `SyntaxError`, others

```
var myError = new Error('my error message');
```

Exceptions

Exception: what happens when you throw something

- Usually you throw an `Error` object
- Call stack unwinds looking for a `catch` block
- You can throw anything, not just `Error` objects...but don't
- If an exception is unhandled, Node will shut down

```
throw new Error('something bad happened!');
throw 'something bad happened' // avoid this
```

Stack trace example

```
function a() {
  // call stack here is [a]
  b();
}

function b(x) {
  // call stack here is [b, a]
  try {
    c()
  } catch (e) {
    console.log(e.stack);
  }
}

function c() {
  // call stack here is [c, b, a]
  throw new Error('boom');
}

a();
```

```
Error: boom
    at c (/Users/lewis/dev/node-error-talk/try-catch.js:14:9)
    at b (/Users/lewis/dev/node-error-talk/try-catch.js:7:5)
    at a (/Users/lewis/dev/node-error-talk/try-catch.js:2:3)
    at Object.<anonymous> (/Users/lewis/dev/node-error-talk/try-catch.js:17:1)
    ...
    <more node core module frames>
```

This is useful! We want to see it!

Errors vs Exceptions in Async Node Land

- We're generally not going to throw Exceptions
- We're going to pass `Error` objects around a lot

Three Guiding Principles

Always know when your errors happen

Avoid patterns like:

```
function (err, result) {
  if (err) { /* drat, ignore */ }
}

try { ... } catch (e) {
  // this should never happen
}

Promise.catch(function (reason) {
  // surely this won't happen
});

req.on('error', function (err) {
  // oh well, not gonna do anything
});
```

Handle what you can, avoid what you can't

Operational Errors vs Programming Errors

- Operational error: recoverable - expect and handle these
  - Typically an `Error` object being passed around
- Programming error: nonrecoverable - try to avoid these
  - Typically an Exception thrown

Operational Errors to Expect

- Network timeouts
- Database is down
- Disk got full
- 3rd party API returning errors: S3 goes down
- Unexpected or missing user inputs (`JSON.parse`)

Programming Errors to Avoid

- Silently ignoring/swallowing errors instead of handling them
- Classic JavaScript errors
  - `TypeError: undefined is not a function`
  - `TypeError: Cannot read property 'x' of undefined`
  - `ReferenceError: x is not defined`
- Invoking a callback twice
- Using the wrong error mechanism in the wrong place

What to do with each:

- Operational errors
  - Known; handle manually wherever they may occur
  - Recoverable if handled correctly: S3 being down doesn't kill us
  - Avoid assuming anything is reliable outside your own process
- Programming errors
  - Unknown; catch with global error handler
  - Nonrecoverable: we're gonna have to abandon ship
    - No amount of additional code can fix a typo
  - Use a linter to help avoid many common problems

Don't keep running in an unknown state

Don't keep running in an unknown state

- State shared across multiple requests: less isolation
- Unexpected error in one request can pollute state of others
- Polluted state can lead to undefined behavior
    - Memory leaks, infinite loops, security issues
- Only way to get back to a known good state: bail out, restart

This leads us to the game plan

The game plan

1. Follow the guiding principles
2. Know and use different mechanisms for effective handling
3. Have a global catch-all for the errors you couldn't handle
4. Use a process manager so shutting down is no big deal
5. Accept when it's time to pack up shop, clean up, shut down

1. Know and use different mechanisms for effective handling

Error Mechanisms in Node

- Try/Catch - `throw` and `try/catch`
- Callbacks - `err` first argument and `if (err)`
- Promises - `reject(err)` and `.catch()`
- Async/Await - Sugar for promises + `try/catch`
- EventEmitters - `error` events and `.on('error')`
- Express - `next(err)` and error-handling middleware

Callbacks & Try-catch

```
function readAndParse(file, callback) {
  fs.readFile(file, { encoding: 'utf8' }, function (err, data) {
    if (err) return callback(err);
    var parsed;
    try {
      parsed = JSON.parse(data);
    } catch (e) {
      return callback(e);
    }
    callback(null, parsed);
  });
}
```

Promises

```
var p = new Promise(function (resolve, reject) {
  fs.readFile('data.txt', function (err, data) {
    if (err) return reject(err);
    resolve(data);
  });
});
p.then(parseJson)
.then(doSomethingElse)
.catch(function (reason) {
  // if readFile or JSON parsing or something else failed,
  // we can handle it here
});
```

Async/Await

```
try {
  await somePromiseThatRejects()
} catch (e) {
  // e is the rejection reason!
}
```

EventEmitters

- Servers, sockets, requests, streams
    - Long-lived objects with asynchronous stuff going on
- They can emit `error` events: listen for them!
- If an `error` event is emitted without an `error` listener…
    - The `Error` object will be thrown instead!
    - How operational errors become programming errors

EventEmitter Request Example

```
var req = http.get(url, function (res) {
  doSomething(res);
});

req.on('error', function (err) {
  // we caught the request error, let's recover
}
```

Express Error Middleware

```
app.post('/login', function (req, res, next) {
  db.query('SELECT ...', function (err, user) {
    if (err) return next(err);
  });
});

app.use(function (req, res, next, err) {
  // spit out your own error page, log the error, etc
  next();
});
```

2. Have a global catch-all for the errors you couldn't handle

Basic Global Error Handler

```
process.on('uncaughtException', function (err) {
  console.log('Uncaught exception! Oh no!');
  console.error(err);
  process.exit(1);
});
```

3. Use a process manager so shutting down is no big deal

Process managers and Node

- Run multiple server processes
  - One of them dying won't take us offline
  - Node cluster module
- Process managers: systemd, pm2, forever, naught
  - Will automatically restart processes when they die
  - Some provide further Node-specific functionality

Example with naught

```
var server = http.createServer(...);

process.on('uncaughtException', function (err) {
  console.log('Uncaught exception! Oh no!');
  console.error(err);
  // tell naught to stop sending us connections & start up a replacement
  process.send('offline');
  process.exit(1);
});

server.listen(80, function () {
  // tell naught we're ready for traffic
  if (process.send) process.send('online');
});
```

4. Accept when it's time to pack up shop, clean up, shut down

When we catch a "fatal" error, we want to:

- Quit accepting new connections
- Start reporting whatever we're gonna report
- Tell proc manager we're gonna die so it starts replacement
- Wait for any existing requests, sockets, etc to be dealt with
- Close any open resources, connections, etc
- Shut down

We want to report that stack trace!

```
function reportError(err, cb) {
  // send the stack trace somewhere, then call cb()
}

process.on('uncaughtException', function (err) {
  process.send('offline');
  reportError(err, function (sendErr) {
    // once error has been reported, let's shut down
    process.exit(1);
  });
});
```

Maybe get a text message:

```
var myPhone = "..."
function reportError(err, cb) {
  console.error(err);
  twilio.sendTextMessage(myPhone, err.message, cb);
}
```

Also useful to report operational errors

```
db.query('SELECT ...', function (err, results) {
  if (err) {
    return reportError(err);
  }
  doSomething(results);
});
```

But then the database goes down…

Alternatively, use Sentry

The `raven` npm package is Sentry's Node error reporting SDK:

```
var Raven = require('raven');
Raven.config('<my-sentry-key>');

function reportError(err, cb) {
  console.error(err);
  Raven.captureException(err, cb);
}
```

Graceful shutdown

```
server = http.createServer(...);
function shutDownGracefully(err, cb) {
  // quit accepting connections, clean up any other resources
  server.close(function () {
    // could also wait for all connections: server._connections
    reportError(err, cb)
  });
}

process.on('uncaughtException', function (err) {
  process.send('offline');
  shutDownGracefully(function () {
    process.exit(1);
  });
});
```

- Wait for any existing requests, sockets, etc to be dealt with
- Close any open resources, connections, etc

Big Combined Example

```
server = http.createServer(...);

function reportError(err, cb) {
  console.error(err);
  Raven.captureException(err, cb);
}

function shutDownGracefully(err, cb) {
  // quit accepting connections, clean up any other resources
  server.close(function () {
    // could also wait for all connections: server._connections
    reportError(err, cb)
  });
}

process.on('uncaughtException', function (err) {
  process.send('offline');
  shutDownGracefully(function () {
    process.exit(1);
  });
});

server.listen(80, function () {
  if (process.send) process.send('online');
});
```

What NOT to do with a global catch-all:

- Just log the error and carry on
- Keep the process running indefinitely
- Try to recover in any way: it's too late!
- Try to centralize handling of operational errors into one place

Other global error mechanisms

- `process.on('uncaughtException')`
- `process.on('unhandledRejection')`
  - Currently non-fatal, warning starting in Node 7
  - Future: fatal, will cause process exit
- Domains: application code shouldn't need them

Recap: overall

1. Follow the guiding principles
2. Know and use different mechanisms for effective handling
3. Have a global catch-all for the errors you couldn't handle
4. Use a process manager so shutting down is no big deal
5. Accept when it's time to pack up shop, clean up, shut down

Related things I didn't go into

- Run-to-completion semantics & the event loop
- V8 stacktrace API
- The "callback contract"
- Asynchronous stacktraces
- Cluster module & process managers
- Domains
- async_hooks (!!!!!)

Some related links

- https://sentry.io/for/node/
- https://www.joyent.com/node-js/production/design/errors
- https://nodejs.org/api/errors.html

Thank you!

- Slides available at GitHub.com/LewisJEllis/node-error-talk
- I'm Lewis J Ellis: @lewisjellis on Twitter and GitHub