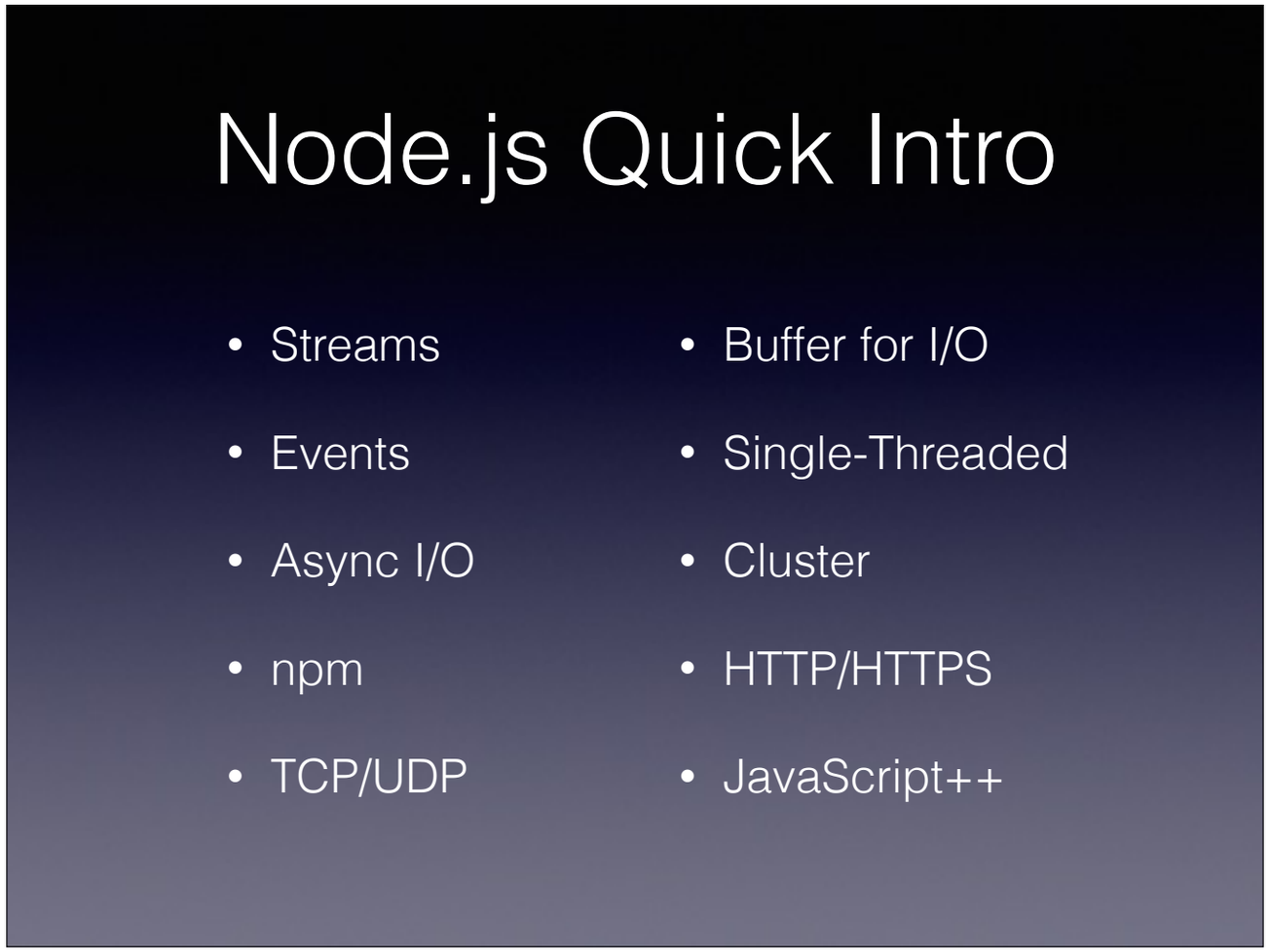
Node.js in 20 Minutes

Edmond Meinfelder
https://github.com/stdarg/Node20Min

Hello! This is Node.js in 20 minutes. Why 20 minutes, it's all the time I have. There may be more slides than time, but they're online, with the code, at the Github URL.

This is a code-focused introduction to Node.js. Though the code works and does what I intend, it's illustrative, designed to teach, and not meant to represent good coding practice. Please only use this code to understand and teach Node.js.



Node.js has some nifty features. Think of it as JavaScript with additions enabling you to build useful applications.

The killer feature of Node happens to be something not mentioned on the nodejs.org website, npm. Every moment you spend learning npm will pay off.

Node’s Hello World

```
var http = require(‘http');

http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(1337, ‘127.0.0.1');

console.log('Server running at http://127.0.0.1:1337/');
```

```
$ node hello.js
Server running at http://127.0.0.1:1337/
```

```
$ curl http://127.0.0.1:1337/
Hello World
```

The classic hello world from the nodejs.org website. It shows how to get an HTTP server listening and responding to requests. Normally, we don’t see this as web frameworks like hapi.js do this for us. The req and res objects are both stream objects and event emitters. Node.js uses events to communicate information about the streams. We’ll touch on events and streams later in the presentation.

Convention: Error First

```
var fs = require('fs');
fs.readFile(‘/etc/no.such.file’, function (err, data) {
  if (err) throw err;
  console.log(data.toString());
});
```

```
var fs = require('fs');
fs.rename('/tmp/hello', '/tmp/world', function (err) {
  if (err) throw err;
  console.log(‘file renamed');
});
```

In Node.js there is a convention where, if a callback has an error as a parameter, the error is the first parameter. I like it, because if there is an error, it’s the first thing you should handle. If there is no error, the parameter is usually null or undefined. I prefer undefined; null is an object with a false value (weird).

Convention: CB Last

```
var fs = require('fs');
fs.readFile(‘no.such.file’, function (err, data) {
  if (err) throw err;
  console.log(data.toString());
});
```

```
var net = require('net');
var server = net.createServer(function(c) {
  c.write('hello world\r\n');
  c.pipe(c);
});
server.listen(8000, function() {
  console.log('server bound');
});
```

The other convention is when there is a function that takes a callback as a parameter, the callback is the last parameter. I like this, too. Often, you just have a few lines needed in a callback to do and it’s nice to “cuddle” it up to the function responsible (as shown in the code samples).

Problem: Nested Callbacks

```
var fs = require('fs');
var filename = './nested_cbs.js';

fs.readFile(filename, function(err, data) {
  if (err)
    return console.error('Error reading the file:',filename+'\n' +
        err.stack);
  var str = data.toString();
  str = str.replace(/var/g, 'banana');
  fs.writeFile(filename+'.new', str, function(err) {
    if (err)  return console.error(err.stack);
    console.log(filename+'.new written.');
  });
});
```

A big question, when coding in an asynchronous language is, “How do I order a sequence actions?” For example, what if I wanted to do a set of steps sequentially? That’s tricky with a set of asynchronous functions. One solution is to nest the callback to enforce a sequential order.

Here, we read a file and then, after error checking, we want to replace all instances of “var” with “banana,” don’t ask why. After that, we asynchronously write the new file. It’s not terrible, but imagine if we had a additional steps that had to happen sequentially - doing sequential things in async code can be awkward, leading to callback hell.

Solution: async

```
var fs = require('fs');
var filename = ‘./async_sol.js';
var async = require('async');
async.waterfall([
  function(cb) {
    fs.readFile(filename, function(err, data) {
      if (err) return cb(err);
      var str = data.toString();
      str = str.replace(/var/g, 'banana');
      cb(null, str);
    });
  },
  function(fContents, cb) {
    fs.writeFile(filename+'.new', fContents, function(err) {
      if (err)  return cb(err);
      console.log(filename+'.new written.');
      cb();
    });
  }],
  function(err) {
    if (err) return console.error(err.stack);
    console.log('Done');
  }
);
```

One solution to avoiding nested callback is the async module. Async is a control flow library. There are many others, e.g. slide, seq, noflo and others. Async is popular, I happen to know it, and here it is. Waterfall is a way to do sequential steps where the output of one step is the input to the next. If you can imagine a control flow scenario, you can likely do it in async.

When you first look at async, there’s a strong “WTF?” reaction, but after a while Stockholm syndrome sets in and you’re happy.

Solution: Promises

```
var fs = require('fs');
var filename = './cb_hell_bbird.js';
var promise = require('bluebird');
promise.promisifyAll(fs);

fs.readFileAsync(filename)
.then(function(data) {
  var str = data.toString();
  str = str.replace(/var/g, 'banana');
  return str;
})
.then(function(str) {
  return fs.writeFileAsync(filename+'.new', str);
})
.catch(function(err) {
  return console.error('File Error:',filename+'\n' + err.stack);
})
.done(function() { console.log('done!'); });
```

Promises are another form of control flow for asynchronous code. Though q is the most popular promise-based module, I'm personally biased towards bluebird. Promises take the async code and present it in a comprehensible, declarative form. The cultural rift between promises and callbacks is as divisive as tabs or spaces, Backbone or Angular, Java or C++.
A weakness is, bluebird has to "promisify" the Node modules, but it does a great job and the resulting code is clean and easy to read.

Yes, "Callback Hell" exists, but there reasonable solutions exist.

Streams (in 2 minutes)

- Uses events to pass data, communicate errors, and the end of input.
- Data is passed using Buffer objects or strings.
- Streams can be readable, writable, both or transformational.
- Readable streams can be piped to writable streams.
- Streams have an API, i.e. methods to implement, if you need to create your own.

Streams are as core to Node.js as asynchronous I/O and events. The HTTP request and response objects are stream objects that emit events for you to read and write data. You can also create your own stream objects. You could create a transform stream to transcode one media format to another or to change a character encoding.

Reading From Stream

```
function getBody(req, cb) {
  var body = '';

  // collect the body
  req.on('data', function (data) {
    body += data.toString();
  });

  // we have the body & header, now process.
  req.on('end', function () {
    req.body = body;
    cb(undefined, body);
  });
}
```

Reading from streams is easy. Node.js uses events to signal to the asynchronous events, e.g. data ready for reading and end of stream reached. This code example could be used with the nodejs.org example to acquire the body from the HTTP request.

One thing to note, the data object on the data event is a buffer, not a string.

Cluster (in 2 minutes)

- If you've used the Unix fork/exec system calls, it's like that. Otherwise, it's magic.
- The master fork()s child processes.
- Each child process calls listen(), but the master opens the socket and sends the requests to the children (round-robin), who are listening on that port
- Events coordinate the master & children: fork, online, listening, disconnect, exit and setup.

JavaScript code is single-threaded. v8 and Node have threads, but the JavaScript code you write will run in a single thread. Context switches occur when you call asynchronous functions. Thus, if you had a CPU-intensive algorithm, you might want to move the CPU-intensive work to a worker process, using cluster. If you have a multi-core box, cluster is a simple way to utilize all the CPUs. And if you must to do some CPU heavy work, e.g. transcoding or cryptography, doing it in a worker is a good idea.

Simple Cluster Example

```
var cluster = require('cluster');
var http = require('http');

if (cluster.isMaster) {
  for (var i=0; i<3; i++)  cluster.fork();

  cluster.on('listening', function(worker, address) {
    console.log('Worker '+worker.id+
        ' is now listening on port', address.port);
  });
} else {
  http.createServer(function(req, res) {
    res.writeHead(200);
    res.end('Hello world from: '+cluster.worker.id+'\n');
  }).listen(8000);
}
```

In the code sample, I modified the nodejs.org “Hello World” example to use cluster, except I identify which worker is saying hello. If you’ve ever done Unix systems programming with fork/exec, the logic for the master and worker code paths should seem familiar.

What’s happening is that same code executes for the master and the worker, so we need to distinguish what the worker and the master does. Here, the master creates three workers and the workers each respond to HTTP requests with “Hello world from: <id>”.

Cluster Hello World

```
$ node cluster.js
Worker 2 is now listening on port 8000
Worker 1 is now listening on port 8000
Worker 3 is now listening on port 8000
```

```
$ curl http://localhost:8000/
Hello world from: 3
$ curl http://localhost:8000/
Hello world from: 3
$ curl http://localhost:8000/
Hello world from: 3
$ curl http://localhost:8000/
Hello world from: 3
```

Here, for clarity, I show how to run the cluster the cluster example. You might spot 2 interesting aspects in the output when I use cluster.

Sockets - TCP

```
var net = require('net');
var conn = {};
var server = net.createServer(function(c) {
  conn[c._handle.fd] = c;
  c.on('end', function() {
    delete conn[c._handle.fd];
  });
  c.on('data', function(buf) {
    for (var i in conn) {
      if (Number(i) !== c._handle.fd)
          conn[i].write(buf);
    }
  });
});
server.listen(3000);
```

Now that we are used to asynchronous callbacks and events, let’s look at a simple TCP server. Here we create a server with a callback that’s called a “connection listener,” called on every new connection with a socket object.

I’m only handling the end and the data events. If you were going to make a production TCP server, there are other events you’d want to handle.

Sockets - UDP

```
var dgram = require('dgram');
var socket = dgram.createSocket('udp4');
var port = 4102;

socket.on('message', function (msg, rinfo) {
  console.log('received: ' + msg + ' from ' +
    rinfo.address + ':' + rinfo.port);
});

socket.bind(port, function() {
  socket.setBroadcast(true);
});

setInterval(function() {
  var msg = new Buffer('Hello world!');
  socket.send(msg, 0, msg.length, port, 'localhost', function(err) {
    if (err) return console.error(err.stack);
  });
}, 1000);
```

Here, we create a UDP socket. UDP is an under-appreciated protocol, in my opinion. On local area networks, it's easy to create a zero-config discovery protocol and now you can set the TTL for network hops.

In this example, we create a UDP socket, set up the message event to display what is received, bind the socket to listen to port 4201 and then, every second send "Hello world!" to the socket.

Async Errors - Try/Catch

```
try {
  setTimeout(function() {
    console.log('Timeout!');
    throw new Error('Timeout err!');
  }, 1);
} catch(err) {
  console.log(‘We will never see this.');
}
```

Try/Catch won’t work w/ async

Try/catch won’t catch async errors, e.g. exceptions thrown in callbacks. That’s just how JavaScript rolls. Fortunately, Node.js has a iffy solution.

Domains (in 2 minutes)

- You cannot catch async errors using try/catch.
- Global uncaught exception handlers are … clumsy.
- It would be awesome to have error handling cover a part of your code.

Domains are a great and under-appreciated feature of Node.js. In JavaScript, the only way to handle errors in synchronous callbacks is to place a try/catch in the callback. However, in Node.js we can wrap the exception of code into a domain that is able to respond to asynchronous events, e.g. errors.

If you are writing Node.js code, take a look at this great feature.

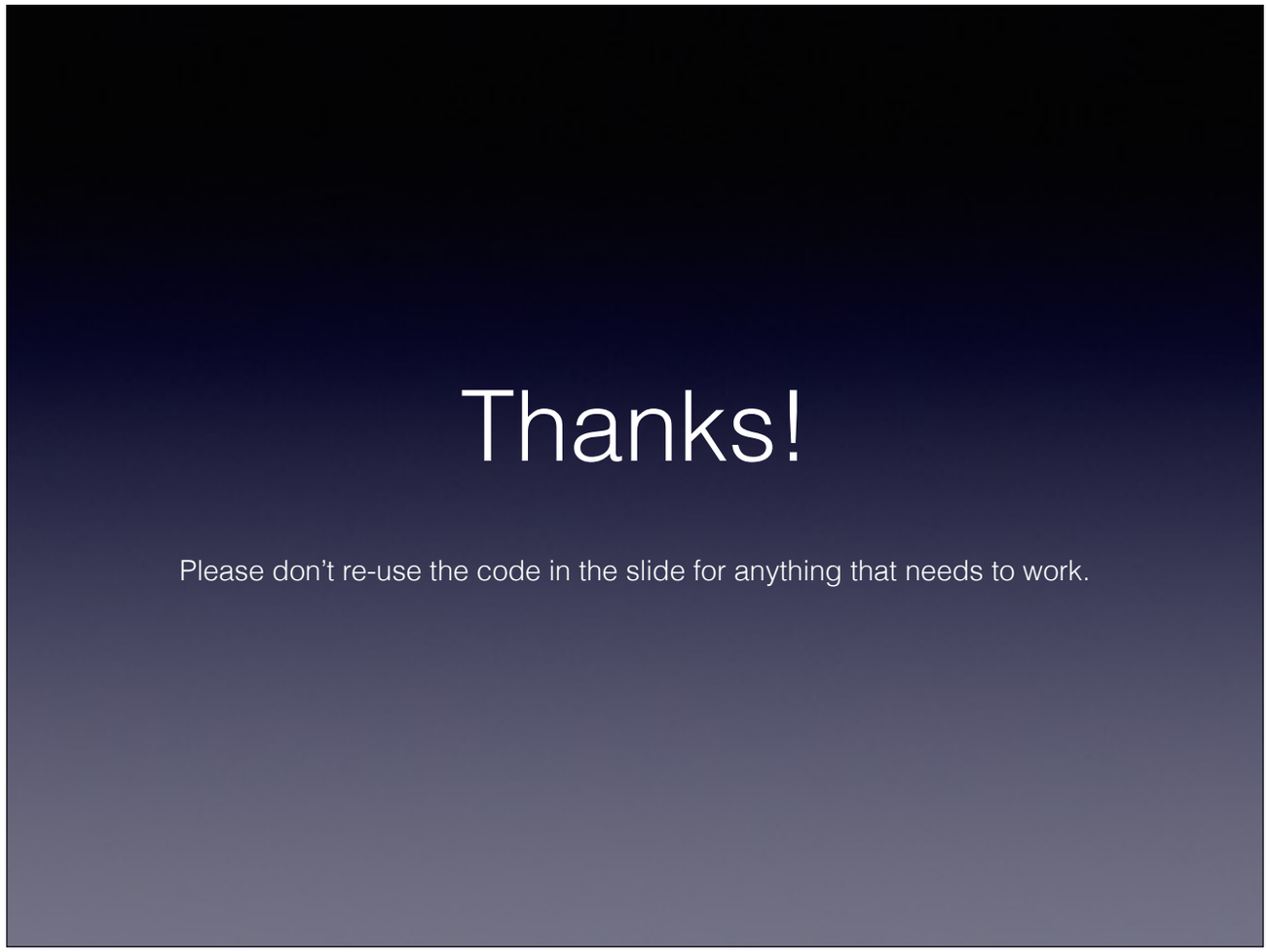
Async Errors - Domains

```
var domain = require('domain');
var d = domain.create();

d.on('error', function(err) {
  console.error('error caught by domain:',
'\n', err.stack);
});

d.run(function() {
  setTimeout(function() {
    console.log('Timeout!');
    throw new Error('Timeout err!');
  }, 1);
});
```

With Node.js domains you can handle async errors.



Thanks!

And again, the code in the presentation is intended to illustrate features of Node.js and not serve as a good representation of production code.