1.1 What's making Node more popular than Rails and other alternatives?

Node markets itself as an asynchronous, event-driven framework built on top of Chrome's JavaScript engine and designed for creating scalable network applications. It's basically JavaScript plus a bunch of C/C++ under the hood for things like interacting with the filesystem, starting up HTTP or TCP servers and so on.

Node is single-threaded and uses a concurrency model based on an event loop. It is non-blocking, so it doesn't make the program wait, but instead it registers a callback and lets the program continue. This means it can handle concurrent operations without multiple threads of execution, so it can scale pretty well.

In a sequential language such as PHP, in order to get the HTML content of a page you would do the following:

$response = file\_get\_contents("http://example.com");

print\_r($response);

In Node, you register some callbacks instead:

var http = require('http');

http.request({ hostname: 'example.com' }, function(res) {

res.setEncoding('utf8');

res.on('data', function(chunk) {

console.log(chunk);

});

}).end();

There are two big differences between the two implementations:

* Node allows you to perform other tasks while waiting to be notified when the response is available.
* The Node application is not buffering data into memory, but instead it's outputing it chunk-by-chunk.

While other event loop systems exist (such as the EventMachine library in Ruby or Twisted in Python), there is a significant difference between them and Node.

In Node, all of the libraries have been designed from the ground up to be non-blocking, but the same cannot be said for the others.

1.2 Use cases

Node is ideal for I/O bound applications (or those that wait on user events), but not so great for CPU-heavy applications. Good examples include data-intensive realtime applications (DIRT), single page applications, JSON APIs, and data-streaming applications.

1.3 npm, the official Node package manager

Node owes a big part of its success to npm, the package manager that comes bundled with it. There are a lot of great things about npm:

* It installs application dependencies locally, not globally.
* It handles multiple versions of the same module at the same time.
* You can specify tarballs or git repositories as dependencies.
* It's really easy to publish your own module to the npm registry.
* It's useful for creating CLI utilities that others can install (with npm) and use right away

## 3 Node Fundamentals

We are going to look at the main Node.js concepts next:

* How to include external libraries by requiring modules
* The role of callbacks
* The EventEmitter pattern
* Streams data one chunk at a time

### 3.1 Modules

Java or Python use the import function to load other libraries, while PHP and Ruby use require. Node implements the CommonJS interface for modules. In Node you can also load other depencies using the require keyword.

For example, we can require some native modules:

var http = require('http');

var dns = require('dns');

We can also require relative files:

var myFile = require('./myFile'); // loads myFile.js

To install modules from npm, either search for them on the [website](https://npmjs.org/) or on Github. The syntax for installing an npm module locally is pretty straightforward:

# where express === module name

$ npm install express

You can require module install from npm as you would do with the native ones, no need to specify the absolute or relative path:

var express = require('express');

The nice thing about requiring Node modules is that they aren't automatically injected into the global scope, but instead you just assigned them to a variable of your choice. That means that you don't have to care about two or more modules that have functions with the same name.

When creating your own modules, all you have to do is take care when exporting something (wheather it's a function, an object, a number or so on). The first approach would be to export a single object:

var person = { name: 'John', age: 20 };

module.exports = person;

The second approach requires adding properties to the exports object:

exports.name = 'John';

exports.age = 20;

A thing to note about modules is that they don't share scope, so if you want to share a variable between different modules, you must include it into a separate module that is then required by the other modules. Another interesting thing you should remember is that modules are only loaded once, and after that they are cached by Node.

Unlike the browser, Node does not have a window global object, but instead has two others: globals and process. However, you should seriously avoid adding properties on the two.

### 3.2 Callbacks

In asynchronous programming we do not return values when our functions are done, but instead we use the continuation-passing style (CPS). You can read more on CPS [here](http://en.wikipedia.org/wiki/Continuation-passing_style).

With this style, an asynchronous function invokes a callback (a function usually passed as the last argument) to continue the program once the it has finished.

Below is an example that looks up IPv4 addresses for a domain:

var dns = require('dns');

dns.resolve4('www.google.com', function (err, addresses) {

if (err) throw err;

console.log('addresses: ' + JSON.stringify(addresses));

});

We have passed a callback (the inline anonymous function) as the second argument to the dns.resolve4 asynchronous function. Once the async function has the response ready for us it will invoke the callback, thus continuing the program execution. This is how we make use of CPS.

### 3.3 Events

The standard callback pattern works well for the use cases where we want to be notified when the async function finishes. However, there are situations that require being notified of different events that do not occur at the same time.

Let us look at an example involving an IRC client:

var irc = require('irc');

var client = new irc.Client('irc.freenode.net', 'myIrcBot', {

channels: ['#sample-channel']

});

client.on('error', function(message) {

console.error('error: ', message);

});

client.on('connect', function() {

console.log('connected to the irc server');

});

client.on('message', function (from, to, message) {

console.log(from + ' => ' + to + ': ' + message);

});

client.on('pm', function (from, message) {

console.log(from + ' => ME: ' + message);

});

We are listening to different types of events in the above example:

* The connect event is emitted when the client has successfully connected to the IRC server.
* The error event is triggered in case an error occurs.
* The message and pm events are emitted for incoming messages.

The events mentioned above make this situation ideal for using the EventEmitterpattern.

The EventEmitter pattern allows implementors to emit an event to which the consumers can subscribe if they are interested. This pattern may be familiar to you from the browser, where it is used for attaching DOM event handlers.

Node has an [EventEmitter](http://nodejs.org/api/events.html" \l "events_class_events_eventemitter) class in core which we can use to make our own EventEmitter objects. Let's create a MemoryWatcher class that inherits from EventEmitter and emits two types of events:

* A data event at a regular interval, representing the memory usage in bytes
* An error event, in case the memory exceeds a certain limit imposed

The MemoryWatcher class will look like the following:

var EventEmitter = require('events').EventEmitter;

var util = require('util');

function MemoryWatcher(opts) {

if (!(this instanceof MemoryWatcher)) {

return new MemoryWatcher();

}

opts = opts || {

frequency: 30000 // 30 seconds

};

EventEmitter.call(this);

var that = this;

setInterval(function() {

var bytes = process.memoryUsage().rss;

if (opts.maxBytes &amp;&amp; bytes &gt; opts.maxBytes) {

that.emit('error', new Error('Memory exceeded ' + opts.maxBytes + ' bytes'));

} else {

that.emit('data', bytes);

}

}, opts.frequency);

}

util.inherits(MemoryWatcher, EventEmitter);

Using it is very simple:

<!-- code lang=javascript linenums=true -->

var mem = new MemoryWatcher({

maxBytes: 12455936,

frequency: 5000

});

mem.on('data', function(bytes) {

console.log(bytes);

})

mem.on('error', function(err) {

throw err;

});

An easier way to create EventEmitter objects is to make new instances from the raw EventEmitter class:

var EventEmitter = require('events').EventEmitter;

var emitter = new EventEmitter();

setInterval(function() {

console.log(process.memoryUsage().rss);

}, 30000);

### 3.4 Streams

[Streams](http://nodejs.org/api/stream.html) represent an abstract interface for asynchronously manipulating a continuous flow of data. They are similar to Unix pipes and can be classified into five types: readable, writable, transform, duplex and "classic".

As with Unix pipes, Node streams implement a composition operator called .pipe(). The main benefits of using streams are that you don't have to buffer the whole data into memory and they're easily composable.

To have a better understanding of how streams work we will create an application that reads a file, encrypts it using the AES-256 algorithm and then compresses it using gzip. All of this using streams, which means that for each chunk read it will encrypt and compress it.

var crypto = require('crypto');

var fs = require('fs');

var zlib = require('zlib');

var password = new Buffer(process.env.PASS || 'password');

var encryptStream = crypto.createCipher('aes-256-cbc', password);

var gzip = zlib.createGzip();

var readStream = fs.createReadStream(\*\*filename); // current file

var writeStream = fs.createWriteStream(\*\*dirname + '/out.gz');

readStream // reads current file

.pipe(encryptStream) // encrypts

.pipe(gzip) // compresses

.pipe(writeStream) // writes to out file

.on('finish', function () { // all done

console.log('done');

});

Here we take a readable stream, pipe it into an encryption stream, then pipe that into a gzip compression stream and finally pipe it into a write stream (writing the content to disk). The encryption and compression streams are transform streams, which represent duplex streams where the output is in some way computed from the input.

After running that example we should see a file called out.gz. Now it's time to implement the reverse, which is decrypting the file and outputting the content to the terminal:

var crypto = require('crypto');

var fs = require('fs');

var zlib = require('zlib');

var password = new Buffer(process.env.PASS || 'password');

var decryptStream = crypto.createDecipher('aes-256-cbc', password);

var gzip = zlib.createGunzip();

var readStream = fs.createReadStream(\_\_dirname + '/out.gz');

readStream // reads current file

.pipe(gzip) // uncompresses

.pipe(decryptStream) // decrypts

.pipe(process.stdout) // writes to terminal

.on('finish', function () { // finished

console.log('done');

});

## 4 Error Handling

Error handling is one of the most important topics in Node. If you ignore errors or deal with them improperly, your entire application might crash or be left in an inconsistent state.

### 4.1 Error-first callbacks

The "error-first" callback is a standard protocol for Node callbacks. It originated in Node core, but it has spread into userland as well to become today's standard. This is a very simple convention, with basically one rule: the first argument for the callback function should be the error object.

That means that there are two possible scenarios:

* If the error argument is null, then the operation was successful.
* If the error argument is set, then an error occured and you need to handle it.

Let's take a look at how we read a file's content with Node:

fs.readFile('/foo.txt', function(err, data) {

// ...

});

The callback for `fs.readFile has two arguments: the error and the file content.

Now let's implement a similar function that reads the content of multiple files, passed as an array argument. The signature for the function should look similar, but instead of passing a single file path we will pass in an array this time:

readFiles(filesArray, callback);

We will respect the error-first pattern and won't handle the error in the readFiles function, but will delegate that responsibility to the callback. The readFiles function will loop over the file paths and read the content for each. If it encounters an error, it will invoke the callback only once. After it's finished reading the content for the last file in the array it will invoke the array with null as the first argument.

var fs = require('fs');

function readFiles(files, callback) {

var filesLeft = files.length;

var contents = {};

var error = null;

var processContent = function(filePath) {

return function(err, data) {

// an error was previously encountered and the callback was invoked

if (error !== null) { return; }

// an error happen while trying to read the file, so invoke the callback

if (err) {

error = err;

return callback(err);

}

contents[filePath] = data;

// after the last file read was executed, invoke the callback

if (!--filesLeft) {

callback(null, contents);

}

};

};

files.forEach(function(filePath) {

fs.readFile(filePath, processContent(filePath));

});

}

### 4.2 EventEmitter errors

We have to be careful when dealing with event emitters (that means streams too), because if there's an unhandled error event it will crash our application. Here is the most simple example of such an event, triggered by ourselves:

var EventEmitter = require('events').EventEmitter;

var emitter = new EventEmitter();

emitter.emit('error', new Error('something bad happened'));

Depending on your application this might be a fatal error (unrecoverable) or an error that should not crash your application (like failed sending an email for example). Either way you should attach an error event handler:

emitter.on('error', function(err) {

console.error('something went wrong with the ee:' + err.message);

});

### 4.3 Propagating more descriptive errors with the verror module

There are a lot of situations where we'll want to delegate the error to the callback and don't deal with the error ourselves. In fact, that's exactly what we did with the readFiles function we created earlier. In case there's an error reading the file we will just delegate that to the callback.

Let's try to call the function with a non-existent file and see what happens:

readFiles(['non-existing-file'], function(err, contents) {

if (err) { throw err; }

console.log(contents);

});

The output should be something like the following:

$ node readFiles.js

/Users/alexandruvladutu/www/airpair-article/examples/readFiles.js:34

if (err) { throw err; }

^

Error: ENOENT, open '/Users/alexandruvladutu/www/airpair-article/examples/non-existing-file'

That's not super helpful, especially because in real-world situations there will probably be a function that calls another function that calls the original function. For example, you might have another function called readMarkdownFiles that will only read markdown files using the `readFiles function.

Also, the output above doesn't even provide a useful stack trace, so you would have to dig deeper to find out where exactly the error came from. Luckily we can do something about that by integrating the verror [module](http://npm.im/verror) into our application.

With verror, we can wrap our errors to provide more descriptive messages.

We will have to require the module at the beginning of the file and then wrap the error when invoking the callback:

var verror = require('verror');

function readFiles(files, callback) {

...

return callback(new VError(err, 'failed to read file %s', filePath));

...

}

Now let's try to run the example again:

$ node readFiles-verror.js

/Users/alexandruvladutu/www/airpair-article/examples/readFiles-verror.js:35

if (err) { throw err; }

^

VError: failed to read file /Users/alexandruvladutu/www/airpair-article/examples/non-existing-file: ENOENT, open '/Users/alexandruvladutu/www/airpair-article/examples/non-existing-file'

at /Users/alexandruvladutu/www/airpair-article/examples/readFiles-verror.js:17:25

at fs.js:207:20

at Object.oncomplete (fs.js:107:15)

And there it is! Instead of having to search Google for 'ENOENT' and digging through the code, now we know that there was a problem reading the file and that came from the `readFiles function.

This is a simple example but it shows the power of verror. In production this module will be a lot more useful because the codebase will probably be large and the error will be propagated through more functions than in our basic example.

### 4.4 Resources

For more details check out [Joyent's guide on Node error handling](https://www.joyent.com/developers/node/design/errors).

## 5 Debugging Node applications with node-inspector

For small bugs you can always use console.log to track down one thing or another, but for more complex situations there's [node-inspector](https://github.com/node-inspector/node-inspector). It has a lot of goodies baked in, but the most important are:

* It's based on the Blink Developer Tools, so it should look and feel familiar to frontend developers.
* It has the ability to setup breakpoints.
* We can step over, step in, step out, resume (continue).
* We can inspect scopes, variables, object properties.
* Besides inspecting, we can also edit variables and object properties.

node-inspector is installable via npm:

$ npm install -g node-inspector

Let's say we have the following basic Node example:

var http = require('http');

var port = process.env.PORT || 1337;

http.createServer(function(req, res) {

res.writeHead(200, { 'Content-Type': 'text/html' });

res.end(new Date() + '\n');

}).listen(port);

console.log('Server running on port %s', port);

To run our example with node-inspector we just need to type in the following command:

# basically `node-debug` instead of `node`

$ node-debug example.js

That should start our application and open the node-inspector interface in Chrome. Let's setup a breakpoint in the request handlers (by clicking on the line number for the one containing res.writeHead). Now open another tab and visit [http://localhost:1337](http://localhost:1337/). The browser should be in a loading stage, but switch to the node-inspector interface.

If you open the console you can inspect the request ansd response objects, modify them and so on. This is just a basic example to get you started with node-inspector, but in real-world applications you will stil benefit from these debugging techniques to track down more complicated issues.

## 6 Creating a realtime application with Express and Socket.io

Express is the most popular web framework for Node, while Socket.IO is a realtime framework that enables bi-directional communication between web clients and the server. We are going to create a basic tracking pixel application using the two that has a dashboard which reports realtime visits.

Besides Express and Socket.IO we will need to install the emptygif module. When the user visits <http://localhost:1337/tpx.gif>, a message will be sent to all the users that are viewing the [homepage](http://localhost:1337/). The message will contain information related to the clients, mainly their IP address and user agent.

Below is the code for the server.js file:

var emptygif = require('emptygif');

var express = require('express');

var app = express();

var server = require('http').createServer(app);

var io = require('socket.io')(server);

app.get('/tpx.gif', function(req, res, next) {

io.emit('visit', {

ip: req.ip,

ua: req.headers['user-agent']

});

emptygif.sendEmptyGif(req, res, {

'Content-Type': 'image/gif',

'Content-Length': emptygif.emptyGifBufferLength,

'Cache-Control': 'public, max-age=0' // or specify expiry to make sure it will call everytime

});

});

app.use(express.static(\_\_dirname + '/public'));

server.listen(1337);

Now on the frontend all we have to do is listen for that 'visit' event emitted by the server and modify the UI accordingly, like so:

<!DOCTYPE HTML>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Realtime pixel tracking dashboard</title>

<style type="text/css">

.visit {

margin: 5px 0;

border-bottom: 1px dotted #CCC;

padding: 5px 0;

}

.ip {

margin: 0 10px;

border-left: 1px dotted #CCC;

border-right: 1px dotted #CCC;

padding: 0 5px;

}

</style>

</head>

<body>

<h1>Realtime pixel tracking dashboard</h1>

<div class="visits"></div>

<script src="https://code.jquery.com/jquery-1.10.2.min.js"></script>

<script src="//cdnjs.cloudflare.com/ajax/libs/moment.js/2.8.1/moment.min.js"></script>

<script src="/socket.io/socket.io.js"></script>

<script>

$(function() {

var socket = io();

var containerEl = $('.visits');

socket.on('visit', function(visit) {

var newItem = '<div class="visit">';

newItem += '<span class="date">' + moment().format('MMMM Do YYYY, HH:mm:ss') + '</span>';

newItem += '<span class="ip">' + visit.ip + '</span>';

newItem += '<span class="ua">' + visit.ua + '</span></div>';

containerEl.append(newItem);

});

});

</script>

</body>

</html>

Now start up the application, open the dashboard in a tab and the tracking pixel URL in different browsers. The dashboard should be similar to this one: