**Q: What is npm?**

*npm is a package manager for the JavaScript programming language. It is the default package manager for the JavaScript runtime environment Node.js. It consists of a command line client, also called npm, and an online database of public and paid-for private packages, called the npm registry. Wikipedia*

**Q: What is Node.js?**

*Node.js is an open-source, cross-platform JavaScript run-time environment that executes JavaScript code outside of a browser.*

**Q: What are the two types of API functions in Node.js?**

*Asynchronous, Non-blocking functions.*

*Synchronous, Blocking functions.*

**Q: List out the differences between AngularJS and NodeJS?**

*AngularJS is a JavaScript framework created by Google to make life easier for creating reactive client side applications with JavaScript.*

*NodeJS, on the other hand, is JavaScript + a set of libraries that make use of Google's V8 engine to run on the back end of a web application.*

**Q: What is global installation of dependencies?**

*Installing a package globally allows you to use the code in the package as a set of tools on your local computer.*

*To download and install packages globally, on the command line, run the following command:*

*npm install -g <package\_name>*

**Q: What are the core modules of Node,js?**

*Node.js Core Modules. Node.js is a light weight framework. The core modules include bare minimum functionalities of Node.js. These core modules are compiled into its binary distribution and load automatically when Node.js process starts.*

**Q: What is an error-first callback?**

*Error-first callbacks# Most asynchronous methods exposed by the Node.js core API follow an idiomatic pattern referred to as an error-first callback. With this pattern, a callback function is passed to the method as an argument.*

Q: How you can monitor a file for modifications in Node.js ?

*Using fs.watchfile*

const fs = require('fs');

require('log-timestamp');

const buttonPressesLogFile = './button-presses.log';

console.log(`Watching for file changes on ${buttonPressesLogFile}`);

fs.watchFile(buttonPressesLogFile, (curr, prev) => {

console.log(`${buttonPressesLogFile} file Changed`);

});

Q: Could we run an external process with Node.js?

var exec = require('child\_process').exec;

exec('pwd', function callback(error, stdout, stderr){

// result

});

**Q: What's the difference between operational and programmer errors?**

*Error handling*

*Errors can be divided into two main parts: operational errors and programmer errors.*

***Operational errors***

*Operational errors can happen in well-written applications as well, because they are not bugs, but problems with the system / a remote service, like:*

*request timeout*

*system is out of memory*

*failed to connect to a remote service*

*Handling operational errors*

*Depending on the type of the operational error, you can do the followings:*

*Try to solve the error - if a file is missing, you may need to create one first*

*Retry the operation, when dealing with network communication*

*Tell the client, that something is not ok - can be used, when handling user inputs*

*Crash the process, when the error condition is unlikely to change on its own, like the application cannot read its configuration file*

*Also, it is true for all the above: log everything.*

***Programmer errors***

***Programmer errors are bugs. This is the thing you can avoid, like:***

*called an async function without a callback*

*cannot read property of undefined*

*Handling programmer errors*

*Crash immediately - as these errors are bugs, you won't know in which state your application is. A process control system should restart the application when it happens, like: supervisord or monit.*

**Q: What are the benefits of using Node.js?**

*Node.js is an open source, cross-platform built on Chrome's JavaScript runtime for fast and scalable server-side and networking applications. Being an interface to the V8 JavaScript runtime, it enables event-driven programming to the web servers through super-fast JavaScript interpreter that runs in the Chrome browser.*

**Q: What is the difference between Nodejs, AJAX, and jQuery?**

*The only similarity between AJAX and Node.js is that they are both JavaScript; yet they both serve completely different purposes. AJAX (short for Asynchronous Javascript and XML) is a client-side technology, often used for updating the contents of a page without refreshing it.*

**Q: How to make Post request in Node.js?**

var post\_data = querystring.stringify({

'compilation\_level' : 'ADVANCED\_OPTIMIZATIONS',

'output\_format': 'json',

'output\_info': 'compiled\_code',

'warning\_level' : 'QUIET',

'js\_code' : codestring

});

// An object of options to indicate where to post to

var post\_options = {

host: 'closure-compiler.appspot.com',

port: '80',

path: '/compile',

method: 'POST',

headers: {

'Content-Type': 'application/x-www-form-urlencoded',

'Content-Length': Buffer.byteLength(post\_data)

}

};

// Set up the request

var post\_req = http.request(post\_options, function(res) {

res.setEncoding('utf8');

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

console.log('Response: ' + chunk);

});

});

// post the data

post\_req.write(post\_data);

post\_req.end();

}

// This is an async file read

fs.readFile('LinkedList.js', 'utf-8', function (err, data) {

if (err) {

// If this were just a small part of the application, you would

// want to handle this differently, maybe throwing an exception

// for the caller to handle. Since the file is absolutely essential

// to the program's functionality, we're going to exit with a fatal

// error instead.

console.log("FATAL An error occurred trying to read in the file: " + err);

process.exit(-2);

}

// Make sure there's data before we post it

if(data) {

PostCode(data);

}

else {

console.log("No data to post");

process.exit(-1);

}

});

**Q: If Node.js is single threaded then how it handles concurrency?**

<https://blog.jayway.com/2015/04/13/600k-concurrent-websocket-connections-on-aws-using-node-js/>

<https://60devs.com/synchronization-of-concurrent-http-requests-in-node.html>

**Q: Is Node a single threaded application?**

const cluster = require('cluster');

const http = require('http');

const numCPUs = require('os').cpus().length;

if (cluster.isMaster) {

console.log(`Master ${process.pid} is running`);

// Fork workers.

for (let i = 0; i < numCPUs; i++) {

cluster.fork();

}

cluster.on('exit', (worker, code, signal) => {

console.log(`worker ${worker.process.pid} died`);

});

} else {

// Workers can share any TCP connection

// In this case it is an HTTP server

http.createServer((req, res) => {

res.writeHead(200);

res.end('hello world\n');

}).listen(8000);

console.log(`Worker ${process.pid} started`);

}

**Q: What are the key features of Node.js?**

*Asynchronous and Event Driven – it means that API call doesn't wait for results and doesn't block other calls. After it finish executing it will run an callback or notify about certain execution parts using events. Even though Node.js is running on single thread with event looping, it can handle more requests than, for example Apache HTTP server, because of using asynchronous non-blocking way of handling code execution.*

*It's (very) fast – it's based on Google Chrome's V8 JavaScript Engine, which is very fast in executing JavaScript code..*

*High scalability - Thanks to the event mechanism Node.js can be easily scalable.*

*NPM (The Node Package Manager) - it's a tool that handles installing and updating of reusable modules from online collection. It also takes care of version and dependency management of reusable modules from online collection. NPM can be compared to Ruby Gems.*

*Community - there are a lots of community tutorials, resources or shared code. Node,js is very popular now and it became one of most of used technolgies used nowadays.*

**Q: What is control flow function?**

*A control flow function is a lightweight, generic piece of code which runs in between several asynchronous function calls and which take care of the necessary housekeeping to: control the order of execution, collect data, limit concurrency and*

*You should be using promises. Bluebird is a great promise library. Faster than native and comes with great features. With promises you can chain together functions, and know that one will not be called until the previous function resolves. No need to set timeouts or delays. Although you can if you'd like. Below is example of a delay. Function B wont run until 6 seconds after A finishes. If you remove .delay(ms) B will run immediately after A finishe*s.

var Promise = require("bluebird");

console.time('tracked');

console.time('first');

function a (){

console.log('hello');

console.timeEnd('first');

return Promise.resolve();

}

function b (){

console.log('world');

console.timeEnd('tracked');

}

a().delay(6000)

.then(b)

.catch(Promise.TimeoutError, function(e) {

console.log('Something messed up yo', e);

});

node test.js

hello

first: 1.278ms

world

tracked: 6009.422ms

**Q: What do you mean by Asynchronous API?**

*Asynchronous requests are useful in maintaining functionality in an application rather than tie up application resources waiting on a request. An API may be synchronous where data or service availability, resources and connectivity are high and low latency is a requirement.*

**Q: What is the purpose of setTimeout function?**

*setTimeout is a native JavaScript function (although it can be used with a library such as jQuery, as we'll see later on), which calls a function or executes a code snippet after a specified delay (in milliseconds).*

**Q: What is REPL in context of Node?**

*The repl module provides a Read-Eval-Print-Loop (REPL) implementation that is available both as a standalone program or includible in other applications. It can be accessed using:*

**Q: How can you avoid callback hells?**

*Don't nest functions. Give them names and place them at the top level of your program*

*Use function hoisting to your advantage to move functions 'below the fold'*

*Handle every single error in every one of your callbacks. Use a linter like standard to help you with this.*

*Create reusable functions and place them in a module to reduce the cognitive load required to understand your code. Splitting your code into small pieces like this also helps you handle errors, write tests, forces you to create a stable and documented public API for your code, and helps with refactoring.*

**Q: What is Callback?**

*Callback is an asynchronous equivalent for a function. A callback function is called at the completion of a given task. Node makes heavy use of callbacks. All the APIs of Node are written in such a way that they support callbacks.*

**Q: What is a blocking code?**

*In coding theory, block codes are a large and important family of error-correcting codes that encode data in blocks. ... The term block code may also refer to any error-correcting code that acts on a block of k bits of input data to produce n bits of output data (n,k). Consequently, the block coder is a memoryless device.*

**Q: How Node prevents blocking code?**

https://nodejs.org/en/docs/guides/dont-block-the-event-loop/

**Q: What is Event Loop?**

*Node.js - Event Loop. Node.js is a single-threaded application, but it can support concurrency via the concept of event and callbacks. ... Node uses observer pattern. Node thread keeps an event loop and whenever a task gets completed, it fires the corresponding event which signals the event-listener function to execute.*

**Q: What is stream and what are types of streams available in Node.js?**

*There are four fundamental stream types within Node.js: Writable - streams to which data can be written (for example, fs.createWriteStream() ). Readable - streams from which data can be read (for example, fs.createReadStream() ). Duplex - streams that are both Readable and Writable (for example, net.Socket ).*

**Q: What is Event Emmitter?**

*Much of the Node.js core API is built around an idiomatic asynchronous event-driven architecture in which certain kinds of objects (called "emitters") emit named events that cause Function objects ("listeners") to be called. ... All objects that emit events are instances of the EventEmitter class.*

**Q: How to avoid callback hell in Node.js?**

*This is about as simple as it gets, and assumes:*

*You are using Express or similar as your route handler, and you can return a promise instead of using the callback*

*You are using Mongoose or similar as the DB, and can return a promise instead of using the callback.*

*Check your versions for Promise support.*

*You appear to be missing some query parameters for Banner at least, and you need to work out if any of these are dependent or, as illustrated, they can be run in parallel.*

router.get('/', (req, res) => {

return Promise.all([

Account.find(),

Banner.findOne(),

Account.findOne({ accType: 'Awaiting' }),

]).then(([ col, banner, doc ]) => {

res.render('admin/', {

title: 'Admin pannel',

user: req.user,

employers: col,

areAwaiting: awaiting,

banner: banner,

);

}).catch(e => { console.error(e) });

});

**Q: What is purpose of Buffer class in Node?**

*Pure javascript, while great with unicode-encoded strings, does not handle straight binary data very well. This is fine on the browser, where most data is in the form of strings. However, node.js servers have to also deal with TCP streams and reading and writing to the filesystem, both which make it necessary to deal with purely binary streams of data.*

*One way to handle this problem is to just use strings anyway, which is exactly what Node.js did at first. However, this approach is extremely problematic to work with; It's slow, makes you work with an API designed for strings and not binary data, and has a tendency to break in strange and mysterious ways.*

> var frosty = new Buffer(24)

> var snowman = new Buffer("☃", "utf-8")

> frosty.write("Happy birthday! ", "utf-8")

16

> snowman.copy(frosty, 16)

3

> frosty.toString("utf-8", 0, 19)

'Happy birthday! ☃'

*Don't use binary strings. Use buffers instead!*

**Q: When should I use EventEmitter?**

*All objects that emit events are instances of the EventEmitter class. These objects expose an eventEmitter.on() function that allows one or more functions to be attached to named events emitted by the object. Typically, event names are camel-cased strings but any valid JavaScript property key can be used.*

*Whenever it makes sense for code to SUBSCRIBE to something rather than get a callback from something. The typical use case would be that there's multiple blocks of code in your application that may need to do something when an event happens.*

[*https://nodejs.org/api/events.html*](https://nodejs.org/api/events.html)

*For example, let's say you are creating a ticketing system. The common way to handle things might be like this:*

[*https://stackoevrflow.com/questions/38881170/when-should-i-use-eventemitter*](https://stackoverflow.com/questions/38881170/when-should-i-use-eventemitter)

**Q: What is difference between synchronous and asynchronous method of fs module?**

*The difference is that in the first example, the program will block in the first line. The next line (console.log) will have to wait.*

*In the second example, the console.log will be executed WHILE the query is being processed. That is, the query will be processed in the background, while your program is doing other things, and once the query data is ready, you will do whatever you want with it.*

*So, in a nutshell: The first example will block, while the second won't*

<https://stackoverflow.com/questions/16336367/what-is-the-difference-between-synchronous-and-asynchronous-programming-in-node>

**Q: What are streams?**

*Streams are collections of data — just like arrays or strings. The difference is that streams might not be available all at once, and they don’t have to fit in memory. This makes streams really powerful when working with large amounts of data, or data that’s coming from an external source one chunk at a time.*

*https://medium.freecodecamp.org/node-js-streams-everything-you-need-to-know-c9141306be93*

**Q: What is the preferred method of resolving unhandled exceptions in Node.js?**

*As you know Node js is a part of Javascript. To handle exception we can have try and catch block.*

*ut good practice says, you should use Process.*

*Process is a global object that provides information about the current Node.js process.*

*Process is a listener function that is always listening to events.*

*Few events are :*

*Exit*

*disconnect*

*unhandledException*

*rejectionHandled*

*If any uncaught exception occurs in your code flow, that exception will be caught in code shown below:*

process.on('uncaughtException', function(err) {

// handle the error safely

console.log(err)

})

**Q: When should we use Node.js?**

*You did a great job of summarizing what's awesome about Node.js. My feeling is that Node.js is especially suited for applications where you'd like to maintain a persistent connection from the browser back to the server. Using a technique known as "long-polling", you can write an application that sends updates to the user in real time. Doing long polling on many of the web's giants, like Ruby on Rails or Django, would create immense load on the server, because each active client eats up one server process. This situation amounts to a tarpit attack. When you use something like Node.js, the server has no need of maintaining separate threads for each open connection.*

*This means you can create a browser-based chat application in Node.js that takes almost no system resources to serve a great many clients. Any time you want to do this sort of long-polling, Node.js is a great option.*

*It's worth mentioning that Ruby and Python both have tools to do this sort of thing (eventmachine and twisted, respectively), but that Node.js does it exceptionally well, and from the ground up. JavaScript is exceptionally well situated to a callback-based concurrency model, and it excels here. Also, being able to serialize and deserialize with JSON native to both the client and the server is pretty nifty.*

*I look forward to reading other answers here, this is a fantastic question.*

*It's worth pointing out that Node.js is also great for situations in which you'll be reusing a lot of code across the client/server gap. The Meteor framework makes this really easy, and a lot of folks are suggesting this might be the future of web development. I can say from experience that it's a whole lot of fun to write code in Meteor, and a big part of this is spending less time thinking about how you're going to restructure your data, so the code that runs in the browser can easily manipulate it and pass it back.*

*Here's an article on Pyramid and long-polling, which turns out to be very easy to set up with a little help from gevent: TicTacToe and Long Polling with Pyramid*

**Q: How to use Buffer in Node.js?**

*31*

*down vote*

*accepted*

*A Buffer is a chunk of memory, just like you would have it in C/C++. You can interpret this memory as an array of integer or floating point numbers of various lengths, or as a binary string. Unlike higher-level data structures like arrays, a buffer is not resizable.*

onst buf = Buffer.from('hello world', 'ascii');

console.log(buf.toString('hex'));

// Prints: 68656c6c6f20776f726c64

console.log(buf.toString('base64'));

// Prints: aGVsbG8gd29ybGQ=

console.log(Buffer.from('fhqwhgads', 'ascii'));

// Prints: <Buffer 66 68 71 77 68 67 61 64 73>

console.log(Buffer.from('fhqwhgads', 'utf16le'));

// Prints: <Buffer 66 00 68 00 71 00 77 00 68 00 67 00 61 00 64 00 73 00>

*It corresponds roughly to:*

*char\* or char[] in C/C++*

*byte[] in Java*

*A mutable bytes or a non-resizable bytearray in Python*

*Strings in php if they were mutable*

**Q: What is Chaining in Node?**

*Node.js Streams, Pipe and chaining. Posted on December 7, 2015 January 1, 2016 by Naeem. A stream is an abstract interface implemented by various objects in Node.js. Due to asynchronous and event driven nature Node.js is very good at handling I/O bound tasks/streams, streams are actually unix pipes*

**Q: What are the global objects of Node.js?**

*Node.js - Global Objects. Node.js global objects are global in nature and they are available in all modules. We do not need to include these objects in our application, rather we can use them directly. These objects are modules, functions, strings and object itself as explained below.*

**Q: Explain how does Node.js work?**

*Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, npm, is the* *largest ecosystem of open source libraries in the world.*

**Q: How does Node.js handle child threads?**

https://nodejs.org/dist/latest-v8.x/docs/api/cluster.html#cluster\_cluster

**Q: How can you listen on port 80 with Node?**

sudo iptables -t nat -A PREROUTING -i eth0 -p tcp --dport 80 -j REDIRECT --to-port 3000

<https://stackoverflow.com/questions/16573668/best-practices-when-running-node-js-with-port-80-ubuntu-linode>

Q: When to not use Node.js?

Q: Is Node.js entirely based on a single-thread?

No

Q: Does Node.js support multi-core platforms? And is it capable of utilizing all the cores?

Yes

**Q:** **Why to use Buffers instead of binary strings to handle binary data ?**

*You can have PHP store all output into a buffer and output all of it at once improving network performance.*

*You can access the buffer content without sending it back to browser in certain situations.*

https://nodejs.org/api/buffer.html

**Q: What's a stub? Name a use case.**

*This isn’t a node.js, or a js-specific thing. Stubbing is the process of creating fake endpoints in code so that you can delay writing complex code, or to isolate what you are testing (see below re: mocks). Say you have a method that you know is going to be complex to write, but the results that are returned are relatively simple, you could write a stub that returns a reasonable value, while keeping track of that for implementation later. You’ve gone to the trouble of figuring out exactly what method you need, so the interface (or contract) is complete (or likely complete, subject to changing requirements).*

[*https://www.quora.com/What-is-the-stub-in-Node-js*](https://www.quora.com/What-is-the-stub-in-Node-js)

**Q: What tools can be used to assure consistent code style?**

*You have plenty of options to do so:*

*JSLint by Douglas Crockford*

*JSHint*

*ESLint*

*JSCS*

Q: How to gracefully Shutdown Node.js Server?

<https://hackernoon.com/graceful-shutdown-in-nodejs-2f8f59d1c357>

const server = app.listen(3000, () => console.log('Example app listening on port 3000!'));

process.on('SIGTERM', () => {

console.info('SIGTERM signal received.');

console.log('Closing http server.');

server.close(() => {

console.log('Http server closed.');

});

})

Q: Is Node.js entirely based on a single-thread?

No

Q: **What is the purpose of \_\_dirname variable?**

*The directory name of the current module. This is the same as the path.dirname() of the \_\_filename.*

**Q: What is the purpose of \_\_filename variable?**

*The file name of the current module. This is the resolved absolute path of the current module file.*

*For a main program this is not necessarily the same as the file name used in the command line.*

*See \_\_dirname for the directory name of the current module.*

**Q: What are the timing features of Node.js?**

**https://www.quora.com/What-are-the-timing-features-of-Node-js**

*Extremely quick: Node.js is based on Google Chrome's V8 JavaScript Engine, so its library is quick in code execution.*

*I/O is Asynchronous and Event Driven: All APIs of Node.js library are offbeat i.e. non-blocking. So a Node.js based server never sits tight for an API to return information. The server moves to the following API subsequent to calling it and a warning instrument of Events of Node.js encourages the server to get a reaction from the past API call. It is likewise a reason that it is quick.*

*Single Threaded: Node.js pursues a solitary strung model with occasion circling.*

*Very Scalable: Node.js is profoundly versatile in light of the fact that occasion system causes the server to react in a non-blocking way.*

*No buffering: Node.js chops down the general preparing time while transferring sound and video records. Node.js applications never support any information. These applications essentially yield the information in lumps.*

*Open source: Node.js has an open source network which has delivered numerous amazing modules to add extra capacities to Node.js applications. We are Providing The Online Training To Cover The All Topics. It is One Of Topic Read More Info On*

**Q: Name some of the events fired by streams.**

*Event: 'close'*

*Event: 'data'*

*Event: 'end'*

*Event: 'error'*

*Event: 'readable'*

https://nodejs.org/api/stream.html#stream\_class\_stream\_readable

**Q: What is Piping in Node?**

*Streams are unix pipes that let you easily read data from a source and pipe it to a destination. ... For example, in a Node.js based HTTP server, request is a readable stream and response is a writable stream. You might have used fs module which lets you work with both readable and writable file streams*

**Q: Explain usage of NODE\_ENV**

*NODE\_ENV is an environment variable made popular by the express webserver framework. When a node application is run, it can check the value of the environment variable and do different things based on the value. NODE\_ENV specifically is used (by convention) to state whether a particular environment is a production or a development environment. A common use-case is running additional debugging or logging code if running in a development environment.*

*var environment = process.env.NODE\_ENV1*

{

"development": {

"config\_id": "development",

"app\_name": "my app",

"app\_desc": "my app desc",

"node\_port": 3000,

"json\_indentation": 4,

"database": "my-app-db-dev"

},

"testing": {

"config\_id": "testing",

"database": "my-app-db-test"

},

"staging": {

"config\_id": "staging",

"node\_port": 8080,

"database": "my-app-db-stag"

},

"production": {

"config\_id": "production",

"node\_port": 8080,

"database": "my-app-db-prod"

}

}

Q: Provide some example of config file separation for dev and prod environments

[`](https://stackoverflow.com/questions/5869216/how-to-store-node-js-deployment-settings-configuration-files)https://codeburst.io/node-js-best-practices-smarter-ways-to-manage-config-files-and-variables-893eef56cbef

**And save all your npm using –save flast to json with it's version and using in production npm install**

**Q. Explain some javascript keyworlds**

[**http://www.javascripter.net/faq/reserved.htm**](http://www.javascripter.net/faq/reserved.htm)

**Q: How would you handle errors for async code in Node.js?**

<https://stackoverflow.com/questions/39655038/node-js-best-practice-exception-handling-after-async-await>

*You can use process 'uncaughtException' and 'unhandledRejection' events.*

*Also remember that it is not safe to resume normal operation after 'uncaughtException', because the system becomes corrupted:*

*The correct use of 'uncaughtException' is to perform synchronous cleanup of allocated resources (e.g. file descriptors, handles, etc) before shutting down the process.*

*process*

*.on('unhandledRejection', (reason, p) => {*

*console.error(reason, 'Unhandled Rejection at Promise', p);*

*})*

*.on('uncaughtException', err => {*

*console.error(err, 'Uncaught Exception thrown');*

*process.exit(1);*

*});*

**Q: Explain what is Reactor Pattern in Node.js?**

*Enter Reactor Pattern- Heart of Node.js. Reactor Pattern is an idea of non-blocking I/O operations in Node.js. ... This demultiplexer is a notification interface that is used to handle concurrency in non-blocking I/O mode and collects every request in form of an event and queues each event in a queue*

**Q: Rewrite the code sample without try/catch block**

var validateObject = function (obj, callback) {

if (typeof obj !== 'object') {

return callback(new Error('Invalid object'));

}

return callback();

};

validateObject('123', function (err) {

console.log('Callback: ' + err.message);

});

Q: Consider following code snippet

Q: Explain some Error Handling approaches in Node.js you know about. Which one will you use?

function myApiFunc(callback) {

/\*

\* This pattern does NOT work!

\*/

try {

doSomeAsynchronousOperation((err) => {

if (err) {

throw (err);

}

/\* continue as normal \*/

});

} catch (ex) {

callback(ex);

}

}

**Reference: https://www.joyent.com/node-js/production/design/errors**

**Q: How would you scale Node application?**

*Horizontally scaling a Node.js application*

*Multiple processes on same machine*

*Native cluster mode*

*PM2 Cluster mode*

*Multiple machines with network load balancing*

*https://medium.com/iquii/good-practices-for-high-performance-and-scalable-node-js-applications-part-1-3-bb06b6204197*

**Q: What is LTS releases of Node.js why should you care?**

*LTS: LTS is an acronym for Long-Term Support, and is applied to release lines (yes, that's plural) that will be supported and maintained by the Node.js project for an extended period of time.May 16, 2018*

**Q: Why should you separate Express 'app' and 'server'?**

In production it is standard to expose your client app with an http server which is better in serving static files, e.g. nginx.

1

All you're doing is serving files with express as though its a standard HTTP server. Its popular to serve these files with something like Nginx instead of relying on express to serve the files. Nginx scales better for this kind of thing. Its also possible to use a CDN to distribute your content to get it closer to your end user.

Either way, using express isn't horrible, but if you plan to scale its probably easier to scale the backend independent from the frontend because the backend is going to be a lot more resource hungry than a process serving static files.

*You can instantiate a connect (the guts of express) server instance when starting both the server and the client from the same script and have it route the requests to node-static when the url starts with public and to connect otherwise.*

*Having all of that in app.js will get out of hand very quickly, even more so once you have more models than just user.*

*Also note that this isn't the only way to organize your routes/models/views nor is it the "best" way for all cases. You'll have you decide for yourself what will work best in each application.*

*Note, the code in the sample is of course all just sample code that probably won't work. My model methods were based off of mongoose model methods.*

*var AllRoutes = require('./app/routes/index.js');*

*app.use('/', AllRoutes);*

*Something like*

*var connect = require('connect');*

*server = connect.createServer();*

*server.use('/public', client.handler\_function);*

*server.use(server.express\_handler\_function);*

**Q: Can Node.js use other engines than V8?**

MEAN stack, reorganized from back to front:

*MongoDB: data persistence, stores data for later retrieval*

*Node.js: web application server, responds to requests from clients*

*Express: web application framework, reduces Node boilerplate*

*Angular.js: browser framework*

*So Node.js does not "work on Firefox" (it doesn't work on Google Chrome either): its a server-side technology. Think of it as a replacement for Python/Ruby/Java in that role. So it can/does respond to requests from all sorts of clients (like Google Chrome and Firefox).*

*What the "built on V8" means is that it uses the same JavaScript interpreter/just-in-time compiler as Google Chrome. But the similarities with chrome pretty much stop there: Node has no rendering engine/css parser/DOM but does have things you need in a server like an HTTP library and a filesystem API.*

*Also, and I mean no offence: we all started where you are, the fact that you are even asking the question (which again is not a bad thing!) means that building on a stack like MEAN is over your head. The documentation is going to assume that you know things you seem to not know. I strongly recommend furthering your understanding of JavaScript and Node through some tutorials and barebones test apps before trying to throw databases and frameworks into the mix.*

**W*hat is the relationship between Node.js and V8?***

*V8 is the Javascript engine inside of node.js that parses and runs your Javascript. The same V8 engine is used inside of Chrome to run javascript in the Chrome browser. Google open-sourced the V8 engine and the builders of node.js used it to run Javascript in node.js.*

***Can Node.js work without V8?***

*No. The current node.js binary cannot work without V8. It would have no Javascript engine and thus no ability to run code which would obviously render it non-functional. Node.js was not designed to run with any other Javascript engine and, in fact, all the native code bindings that come with node.js (such as the fs module or the net module) all rely on the specific V8 interface between C++ and Javascript.*

*There is an effort by Microsoft to allow the Chakra Javascript engine (that's the engine in Edge) to be used with node.js. They build a V8 shim on top of Chakra so that the node.js binary code that expects to be talking to V8 can continue to do what it was doing, but actually end up talking to the Chakra engine underneath. From what I've read this is particularly targeted at Microsoft platforms that already have the Chakra engine and do not have the V8 engine running on them, though presumably you could use it on Windows too.*

**Q: What is the difference between process.nextTick() and setImmediate() ?**

*Use setImmediate if you want to queue the function behind whatever I/O event callbacks that are already in the event queue. Use process.nextTick to effectively queue the function at the head of the event queue so that it executes immediately after the current function completes.*

*So in a case where you're trying to break up a long running, CPU-bound job using recursion, you would now want to use setImmediate rather than process.nextTick to queue the next iteration as otherwise any I/O event callbacks wouldn't get the chance to run between iterations.*

**Q: How to solve "Process out of Memory Exception" in Node.js ?**

[https://github.com/mysqljs/mysql#pooling-connections](https://github.com/mysqljs/mysql" \l "pooling-connections)

*'ll summarize my comments since they sent you on the correct path to address your issue.*

*First, you have to establish whether the issue is caused by your database or not. The simplest way to do that is to comment out the database portion and see how high you can scale. If you get into the thousands without a memory or CPU issue, then your focus can shift to figuring out why adding the database code into the mix causes the problem.*

*Assuming the issues is caused by your database, then you need to start understanding how it is handling things when there are lots of active database requests. Oftentimes, the first thing to use with a busy database is connection pooling. This gives you three main things that can help with scale.*

*It gives you fast reuse of previously opened connections so you don't have every single operation creating its own connection and then closing it.*

*It lets you specify the max number of simultaneous database connections in the pool you want at the same time (controlling the max load you throw at the database and also probably limiting the max amount of memory it will use). Connections beyond that limit will be queued (which is usually what you want in high load situations so you don't overwhelm the resources you have).*

*It makes it easier to see if you have a connection leak problem as rather than just leak connections until you run out of some resource, the pool will quickly be empty in testing and your server will not be able to process any more transactions (so you are much more likely to see the problem in testing).*

*Then, you probably also want to look at the transaction times for your database connections to see how fast they can handle any given transaction. You know how many transactions/sec you are trying to process so you need to see if your database and the way it's configured and resourced (memory, CPU, speed of disk, etc...) is capable of keeping up with the load you want to throw at it.*

**How does Node.js support multi-processor platforms, and does it fully utilize all processor resources?**

Since Node.js is by default a single thread application, it will run on a single processor core and will not take full advantage of multiple core resources. However, Node.js provides support for deployment on multiple-core systems, to take greater advantage of the hardware. The Cluster module is one of the core Node.js modules and it allows running multiple Node.js worker processes that will share the same port.

**What is typically the first argument passed to a Node.js callback handler?**

Node.js core modules, as well as most of the community-published ones, follow a pattern whereby the first argument to any callback handler is an optional error object. If there is no error, the argument will be null or undefined.

A typical callback handler could therefore perform error handling as follows:

function callback(err, results) {

// usually we'll check for the error before handling results

if(err) {

// handle error somehow and return

}

// no error, perform standard callback handling

}

**Consider following code snippet: { console.time("loop"); for (var i = 0; i < 1000000; i += 1){ // Do nothing } console.timeEnd("loop"); } The time required to run this code in Google Chrome is considerably more than the time required to run it in Node.js. Explain why this is so, even though both use the v8 JavaScript Engine.**

Within a web browser such as Chrome, declaring the variable i outside of any function’s scope makes it global and therefore binds it as a property of the window object. As a result, running this code in a web browser requires repeatedly resolving the property i within the heavily populated window namespace in each iteration of the for loop.

In Node.js, however, declaring any variable outside of any function’s scope binds it only to the module’s own scope (not the window object) which therefore makes it much easier and faster to resolve.

**What's a test pyramid? How can you implement it when talking about HTTP APIs?**

The "Test Pyramid" is a metaphor that tells us to group software tests into buckets of different granularity. It also gives an idea of how many tests we should have in each of these groups. Although the concept of the Test Pyramid has been around for a while, teams still struggle to put it into practice properly. This article revisits the original concept of the Test Pyramid and shows how you can put this into practice. It shows which kinds of tests you should be looking for in the different levels of the pyramid and gives practical examples on how these can be implemented.

<https://martinfowler.com/articles/practical-test-pyramid.html>

**Pro and Cons of Nodejs**

**Node.JS Pros:**

Powerful underlying libraries. libuv, the library behind Node that handles non-blocking I/O, is super awesome. If your application involves interacting at a low level with other processes running on the machine, this by itself is a reason to choose Node.

Package management. In my opinion, npm is better than gem. For example, dependencies are installed locally rather than globally, regardless of how your Node.JS installation is set up. There are other reasons why npm is better than gem, but that's a topic for another thread.

Ecosystem. Because it is a newer platform, most of the community-driven libraries are also newer. It's really easy to fall into the trap of using a library in Rails that is no longer actively maintained. However, Node's ecosystem is also mature enough that I've never not been able to find a library to do something I wanted.

WebSockets. JavaScript was written to handle asynchronous requests, like those seen in WebSockets. Support for WebSockets has arrived in Rails 4, but it's just not as natural as it is in Node.

Full stack. From HTTP server to templating engine, Node.JS is the whole package. Rails relies on something like Nginx or Apache with mod\_rails (aka Passenger) in order to actually work.

**Node.JS Cons:**

Error handling. If your application throws an exception, the whole application crashes, and you have to rely on a daemon like Forever to restart the Node process. For a variety of reasons, Rails doesn't have this problem.

Memory management. Node.JS is garbage collected by V8. When the application performs a memory-intensive operation and triggers the garbage collector to run, though, it blocks the thread and causes poor performance for all users. This can be a pretty big problem on resource-limited machines, like the kind commonly found on shared hosts. I am not familiar with how Ruby's GC works and whether or not it has the same problem.

Maintainability. It's really easy for me to maintain my own code, but if someone else came onto my project, they would probably have no idea what's going on. Rail's opinionatedness makes this kind of thing not as much of an issue.

**Q-15. What Are Globals In Node.Js?**

Each module is supposed to be independent. The require doesn't cost anything anyways after the first one for each module.

What if you wanted to test one module alone? You'd be having a lot of issues because it wouldn't recognize some "global" requires that you have in your app.

Yes, globals are bad, even in this case. Globals almost always ruin: testability, encapsulation and ease of maintenance.

Updated Dec. 2012

The global object now has the global scope within the application and can be used to store any data/functions that need to be accessed from all modules.

Updated Dec. 2012

The global object now has the global scope within the application and can be used to store any data/functions that need to be accessed from all modules.