Node.js

Server-Side JavaScript for Backends, API Servers, and Web Apps

UPDATED BY DAVE WHITELEY

WHAT IS NODE?

First, a quick note: The terms "Node.js" and "Node" are used interchangeably. The official description according to the nodejs.org website is:

"Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine"

Translation: Node runs on Google's open source JavaScript engine called V8, which is written in C++ and is used in Google's Chrome browser. It's fast!

"Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient."

Translation: Node.js pairs JavaScript's naturally **event-driven, asynchronous coding style** with **non-blocking I/O libraries** for performing server tasks such as working with file systems and databases. This pairing makes it easy to write fast, efficient, and non-blocking applications that would be difficult and complex to author in traditionally synchronous languages.

"Node.js' package ecosystem, npm, is the largest ecosystem of open source libraries in the world."

Translation: npm (name always lower-case) is the tool used to install and manage dependencies in the Node world as well as the main repository where public Node.js packages are registered. In addition to Node.js libraries, npm also lists more and more front-end packages, but that's a different topic!

NODE IS JAVASCRIPT ON THE SERVER

Node enables developers to write server-side applications in JavaScript. Server-side applications perform tasks that aren't suitably performed on the client, like processing and persisting data or files, plus tasks like connecting to other networked servers, serving web pages, and pushing notifications. Seeing that JavaScript is an incredibly popular language with web and mobile front-end developers, the ability to use this same skill to program server-side tasks can increase a developer's productivity. In some cases, client and server can even share **the same code**.

HOW DOES NODE WORK?

SYNCHRONOUS VS. ASYNCHRONOUS PROGRAMMING

C and Java traditionally use synchronous I/O, which means that when a program starts an I/O operation, **the rest of the program stops until that operation is completed**. You can get around this by writing multi-threaded programs, but for some developers, writing these types of applications in a distributed networking environment can be daunting. Of course there is also the issue of the number of threads a system can actually spawn, and writing "thread-safe" code adds significant complexity to any codebase. Node, by contrast, is single-threaded, but provides for asynchronous and non-blocking code **by default**.

SYNCHRONOUS VS ASYNCHRONOUS: BY ANALOGY

To understand non-blocking I/O, picture a common scenario: ordering food at a restaurant. A typical experience might be:

- You sit at a table and the server takes your drink order.
- The server goes back to the bar and passes your order to a bartender.
- While the bartender is working on your drink, the server moves on to take another table's drink order.
- The server goes back to the bar and passes along the other table's order.
- Before the server brings back your drinks, you order some food.
- The server passes your food order to the kitchen.
- Your drinks are ready now, so the server picks them up and brings them back to your table.
- The other table's drinks are ready, so the server picks them up and takes them to the other table.
- Finally your food is ready, so server picks it up and brings it back to your table.

Basically every interaction with the server follows the same pattern.

1. You order something from the server.
2. The server hands your order off to the bar or kitchen, freeing him up to take new orders or to deliver orders that are completed.
3. When your order is completed, the kitchen or bar alerts the server, and he delivers it to you.

Notice that at no point in time is the server doing more than one thing. He can only process one request at a time, **but he does not wait around for the orders to be filled**. This is how non-blocking Node.js applications work. In Node, your application code is like

StrongLoop
An IBM Company

Node.js Software and Expertise from the Largest Corporate Contributors

Connect devices to data with APIs developed in Node

LoopBack is an open source API Server powered by Node

- Model-driven development with auto-generate REST APIs
- Run LoopBack APIs on-premises or on cloud services like Bluemix
- Easily model business data and behavior along side auto-generated CRUD actions
- Built-in API security and management with token authentications
- iOS, Android, AngularJS and Xamarin SDKs to develop hybrid or native apps
- Prebuilt mobile services like push, geolocation and offline sync
- Open source and extensible by design

Manage Node and APIs in production

Use StrongLoop Process Manager to monitor, optimize and scale Node apps

- Create, manage and scale Node clusters across multiple machines
- CPU, memory and heap profiling
- Memory leak detection
- Root cause analysis with deep transaction tracing
- Event loop and response time monitoring
- Resource utilization
- Bottleneck identification
- Debugging
- Build and remote deploy

For more information visit: strongloop.com

a restaurant server processing orders, and the bar/kitchen is the operating system handling your I/O calls.

Your single-threaded JavaScript application is responsible for all the processing up to the moment it requires I/O. Then, it **hands off the work** to the operating system, which processes the I/O and calls your application back when it's finished. For contrast, imagine if our restaurant were **synchronous**: every time the server took an order, he would wait for the bar/kitchen to finish the order before taking the next request. This would be very slow! This is how blocking I/O works.

EVENT LOOP CONCURRENCY MODEL

Node leverages a browser-style concurrency model on the server. As we all know, JavaScript was originally designed for the browser where code execution is triggered by *events* such as mouse clicks or the completion of an Ajax request. Moved to the server, this same model allows for the idea of an event loop for server events such as network requests. In a nutshell, JavaScript waits for an event and whenever that event happens, a *callback* function occurs.

For example, your browser is constantly looping waiting for events like clicks or mouse-overs to occur, but this listening for events **doesn't block the browser** from performing other tasks. On the server this means that instead of a program waiting for a response from a database query, file access, or connection to an external API, it immediately moves on to the next unit of work until the event returns with a response. Instead of blocking the server waiting for I/O to complete, the event loop enables applications to process other requests while waiting for I/O results. In this way Node achieves multitasking more efficiently than using threads.

EVENT LOOP ANALOGY: MAIL CARRIER

A mail carrier "loops" through the mailboxes on her route, checking each for new letters (*events*). In your postbox she finds a letter addressed to IBM; you are requesting a quote for their cloud backup service. She takes the letter, and when her loop takes her past IBM headquarters she drops it off. **She does not wait for IBM's response**, she continues on her loop, picking up & delivering other letters while IBM processes your request (*does an I/O operation*).

The next three times she passes IBM there is no response ready, so she continues on her loop delivering other letters. The fourth time she passes IBM, they have a response ready. She picks it up & delivers the quote to you. The response, **addressed to the original sender**, is like the callback, routed to the original caller with the response data (the quote).

EVENT LOOP CODE EXAMPLE

Let's look at a simple example of asynchronously reading a file. This is a two step process:

1. The filename is passed to the OS (via Node and libuv).
2. The readHandler callback function is executed with the response (a Buffer).

```
var fs = require('fs');
fs.readFile('my_file.txt', function readHandler(
  err, data) {
    if (err) throw err;
    // convert the buffer to string and output it
    console.log(data.toString());
});
```

The request to read the file goes through Node bindings to **libuv**. Then **libuv** gives the task of reading the file to a thread. When the thread completes reading the file into the buffer, the results goes to V8 and then through the Node bindings in the form of a callback function. In the callback shown the data argument is a Buffer with the file data.

Example of an HTTP server using Node:

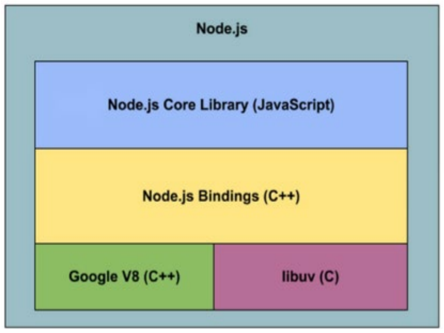
```
var http = require('http');

http.createServer(function (request, response) {
    response.writeHead(200, {
        'Content-Type': 'text/plain'});
    response.end('Hello World\n');
}).listen(8080);

console.log('Server running at http://localhost:8080/');
```

The "event >> callback" mechanism here is the same as in our read file example, but in one case we initiated the operation (read file) that results in an event being triggered, whereas in this example, the events are triggered in response to external input (an incoming HTTP request).

ARCHITECTURE



There are four building blocks that constitute Node:

- Google's V8 provides a run-time for JavaScript, as we've seen.
- libuv is a C library that handles asynchronous I/O and abstracts underlying network and file system functionality so Node can run on different operating system platforms.
- Node core is a JavaScript library that includes modules like Assert, HTTP, Crypto, and so on, that provides Node's base JavaScript API.
- Node bindings provide the glue connecting these technologies to each other and to the operating system.

HOW DO I INSTALL NODE?

Installers exist for a variety of platforms including Windows, Mac OS X, Linux, SunOS—and of course you can compile it yourself from source. Official downloads are available from the nodejs.org website: nodejs.org/en/download

WHAT ARE THE PERFORMANCE CHARACTERISTICS OF NODE?

Everyone knows benchmarks are a specific measurement and don't account for all cases. Certainly, what and how you measure matters a lot. But there's one thing we can all agree on: at high levels of concurrency (thousands of connections) your server needs to become asynchronous and non-blocking. We could have finished that sentence with IO, but the issue is that if any part of your server code blocks, you're going to need a thread. At these levels of concurrency, you can't go about creating threads for every connection. So, the whole code path needs to be non-blocking and async, not just the IO layer. This is where Node excels.

Some examples of Node performance benchmarks and related posts:

- Strongloop
- LinkedIn
- "Node.js Benchmarks"
- "Comparing Node.js vs PHP Performance"
- "The Computer Language Benchmarks Game"

WHAT IS NODE GOOD FOR?

WEB APPLICATIONS

Many modern web applications are SPAs (single page applications) that put most rendering and UI concerns in the client code, calling the server only to request or update data.

REASONS WHY:

- Single page applications most frequently request and send **data** (e.g. JSON) rather than rendered pages (HTML), thus payloads are smaller but more frequent. Node's asynchronous model excels at handling these high-frequency, small-payload requests.
- Node's rich ecosystem of npm modules allows you to build web applications front to back with the relative ease of a scripting language that is already ubiquitously understood on the front end.
- Single Page Applications **must** be written in (or compiled to) JavaScript. A Node backend means the whole stack is in one language.

EXAMPLES OF FRAMEWORKS FOR NODE:

- LoopBack
- Express
- Sails.js
- Hapi
- Meteor
- MEAN.js

API SERVERS AND MOBILE BACKENDS

An I/O library at its heart, Node.js is a popular choice for APIs and mobile backends. Node's ease of use has been applied toward the classic enterprise application use case to be able to gather and normalize existing data and services.

REASONS WHY:

- As the shift toward hybrid mobile applications becomes more dominant in the enterprise, client JavaScript code can be leveraged on the server.
- Node's rich ecosystem has almost every underlying driver or connector to enterprise data sources such as RDBMS, Files, NoSQL, etc. that would be of interest to mobile clients.
- JSON, the de-facto standard format for API data interchange, is a representation of native JavaScript objects, so of course Node handles it easily. If you need to support another format (for example, XML), there's probably a module for it.

EXAMPLES OF MOBILE BACKENDS BUILT WITH NODE:

- LoopBack (Open-source framework)
- FeedHenry (Proprietary)
- Appcelerator Cloud Services (Proprietary)
- Restify (Open-source framework)

IoT SERVERS

As more objects in the workplace, the home, and beyond get connected into the "Internet of Things," Node.js is emerging as the server technology of choice for many IoT platforms.

REASONS WHY:

- Sensors reporting temperature, vehicle speed, etc. can generate lots of data points, each one as small as a single number. Node.js is built to efficiently handle this sort of "many requests, small payloads" use-case.
- Node.js' popularity as a platform for building APIs means that its strengths and weaknesses here are well explored, and there are many mature solutions that fit the IoT problem.

EXAMPLES OF OPEN SOURCE IOT SERVERS BUILT IN NODE:

- Zetta
- mqtt-connection
- adafruit-io
- phant

HOW CAN I MAKE NODE USEFUL?

WHAT IS NPM?

Node Package Manager ("npm") is the command-line package manager for Node that manages dependencies for your application. npmjs.com is the public repository where you can obtain and publish modules.

HOW DOES NPM WORK?

For your Node application to be useful, it needs things like libraries, web and testing frameworks, data-connectivity, parsers and other functionality. You enable this functionality by installing specific modules via npm. npm comes bundled with Node.js (since v0.6.3) so you can start using it right away!

You can install any package with this command:

```
$ npm install <name of module>
```

Some popular and most used modules include:

express
A fast, unopinionated, minimalist web framework for Node. Express aims to provide small, robust tooling for HTTP servers, making it a great solution for single page applications, web sites, hybrids, or public HTTP APIs.

lodash
A "toolbelt" utility library with methods for performing lots of common JavaScript tasks. It can be used stand-alone, in conjunction with other small libraries, or in the context of a larger framework.

async
A utility module that provides straightforward, powerful functions for working with asynchronous JavaScript. Although originally designed for use with Node, it can also be used directly in the browser. Async provides around 20 functions that include the usual 'functional' suspects (map, reduce, filter, each…) in addition to your async function.

request
A simplified HTTP request client. It supports HTTPS and follows redirects by default.

grunt
A JavaScript task runner that helps automate tasks. Grunt can perform repetitive tasks like minification, compilation, unit testing, linting, etc. The Grunt ecosystem is also quite large with hundreds of plugins to choose from. You can find the listing of plugins here.

socket.io
Socket.io makes WebSockets and real-time possible in all browsers and provides built-in multiplexing, horizontal scalability, automatic JSON encoding/decoding, and more.

mongoose
A MongoDB object modeling tool designed to work in an asynchronous environment. It includes built-in type casting, validation, query building, business logic hooks and more, out of the box.

HOW IS NODE.JS MAINTAINED AND IMPROVED ON A DAY-TO-DAY BASIS?

Some people may not know the various high-level Node.js groups and how day to day work on Node.js happens. Let's take a look at the various groups and how they all work on Node.

The Node.js Foundation Board, which is made up of corporate member representatives, a Technical Steering Committee representative, and elected individual membership class representatives, **does not deal with the day to day work**. Instead, responsibilities of the board are primarily:

- set business/technical direction
- oversee intellectual property (IP) management
- marketing
- event planning

The Technical Steering Committee (TSC), which is the technical governing body of the Node.js Foundation, **does not deal with the day to day work**. It admits and retains oversight of all top-level projects under the Node.js Foundation's purview.

The Core Technical Committee, which is in charge of the ongoing maintenance and evolution of Node.js as well as driving the project and community forward, *does handle day to day technical decisions, however only when they need to be made*. (See below for more information.)

Primarily, the development work done on Node.js core is governed by a distributed consensus model, managed by a group of collaborators.

The process goes roughly as follows:

- a pull request is made against the repository
- if more than a single collaborator agrees it should be merged, it will move forward
- the PR should land within 48 hours (72 if during the weekend)

The only time a decision goes to the CTC is when a consensus cannot be reached.

The release schedule of Node.js loosely follows these guidelines:

- major semver increments happen bi-yearly
- current releases can happen weekly or bi-weekly
- LTS releases fluctuate based on needs; typically, early in the LTS term, releases happen every two weeks but it slows to a monthly pace. (Only even numbered releases go to LTS.)

More details can be found in the Collaborator Guide.

NODE API GUIDE

Below is a list of the most commonly-used Node APIs. For a complete list and for an APIs current state of stability or development, see the Node API documentation.

Buffer
Functions for manipulating, creating and consuming octet streams, which you may encounter when dealing with TCP streams or the file system. Raw data is stored in instances of the Buffer class. A Buffer is similar to an array of integers but corresponds to a raw memory allocation outside the V8 heap. A Buffer cannot be resized.

Child Process
Functions for spawning new processes and handling their input and output. Node provides a tri-directional popen(3) facility through the child_process module.

Cluster
A single instance of Node runs in a single thread. To take advantage of multi-core systems, the user will sometimes want to launch a cluster of Node processes to handle the load. The cluster module allows you to easily create child processes that all share server ports.

Crypto
Functions for dealing with secure credentials that you might use in an HTTPS connection. The crypto module offers a way of encapsulating secure credentials to be used as part of a secure HTTPS net or http connection. It also offers a set of wrappers for OpenSSL's hash, hmac, cipher, decipher, sign and verify methods.

Debugger
You can access the V8 engine's debugger with Node's built-in client and use it to debug your own scripts. Just launch Node with the debug argument (**node debug server.js**). A more feature-filled alternative debugger is node-inspector. It leverages Google's Blink DevTools, allows you to navigate source files, set breakpoints and edit variables and object properties, among other things.

Events
Contains the `EventEmitter` class used by many other Node objects. Events defines the API for attaching and removing event listeners and interacting with them. Typically, event names are represented by a camel-cased string; however, there aren't any strict restrictions on case, as any string will be accepted. Functions can then be attached to objects, to be executed when an event is emitted. These functions are called **listeners**. Inside a listener function, the object is the `EventEmitter` that the listener was attached to. All EventEmitters emit the event `newListener` (when new listeners are added) and `removeListener` (when a listener is removed).

To access the `EventEmitter` class use:

```
require('events').EventEmitter.

emitter.on(event, listener) adds a listener to the end of the
listeners array for the specified event. For example:

server.on('connection', function (stream) {
console.log('someone connected!');
});
```

This calls returns emitter, which means that calls can be chained.

Filesystem

The filesystem API contains methods to manipulate files on disk. In addition to read and write methods, there are methods to create symlinks, watch files, check permissions and other file stats, and so on. The major point of note here is that as I/O operations, the core filesystem methods are **asynchronous**. There are synchronous versions of most methods as well (e.g. readFileSync in addition to readFile), but these **synchronous methods should be avoided** when performance is a consideration! Node.js' asynchronous nature depends on the event loop continuing to loop; if it's blocked for a synchronous I/O operation, much of Node's speed is lost.

Globals

Globals allow for objects to be available in all modules. (Except where noted in the documentation.)

HTTP

This is the most important and most used module for a web developer. It allows you to create HTTP servers and make them listen on a given port. It also contains the **request** and **response** objects that hold information about incoming requests and outgoing responses. You also use this to make HTTP requests from your application and do things with their responses. HTTP message headers are represented by an object like this:

```
{ 'content-length': '123',
  'content-type': 'text/plain',
  'connection': 'keep-alive',
  'accept': '*/*' }
```

In order to support the full spectrum of possible HTTP applications, Node's HTTP API is very low-level. It deals with stream handling and message parsing only. It parses a message into headers and body but it does not parse the actual headers or the body.

Modules

Node has a simple module loading system. In Node, files and modules are in one-to-one correspondence. As an example, foo.js loads the module circle.js in the same directory.

The contents of foo.js:

```
var circle = require('./circle.js');
console.log( 'The area of a circle of radius 4 is '
  + circle.area(4));
```

The contents of circle.js:

```
var PI = Math.PI;

exports.area = function (r) {
    return PI * r * r;
};

exports.circumference = function (r) {
    return 2 * PI * r;
};
```

The module `circle.js` has exported the functions `area()` and `circumference()`. To add functions and objects to the root of your module, you can add them to the special exports object. Variables local to the module will be private, as though the module was wrapped in a function. In this example the variable PI is private to circle.js.

Net

Net is one of the most important pieces of functionality in Node core. It allows for the creation of network server objects to listen for connections and act on them. It allows for the reading and writing to sockets. Most of the time, if you're working on web applications, you won't interact with Net directly. Instead you'll use the HTTP module to create HTTP-specific servers. If you want to create TCP servers or sockets and interact with them directly, you'll want to work with the Net API.

Process

Used for accessing stdin, stdout, command line arguments, the process ID, environment variables, and other elements of the system related to the currently-executing Node processes. It is an instance of `EventEmitter`. Here's example of listening for `uncaughtException`:

```
process.on('uncaughtException', function(err) {
    console.log('Caught exception: ' + err);
});

setTimeout(function() {
    console.log('This will still run.');
}, 500);

// Intentionally cause an exception, but don't catch it.
    nonexistentFunc();
console.log('This will not run.');
```

REPL

Stands for **R**ead-**E**val-**P**rint-**L**oop. You can add a REPL to your own programs just like Node's standalone REPL, which you get when you run node with no arguments. REPL can be used for debugging or testing.

Stream

An abstract interface for streaming data that is implemented by other Node objects, like HTTP server requests, and even stdio. Most of the time you'll want to consult the documentation for the actual object you're working with rather than looking at the interface definition. Streams are readable, writable, or both. All streams are instances of `EventEmitter`.

Util

This API contains various utility methods, mostly for meta-tasks like logging and debugging. It's included here to highlight one method: **inherits**. Constructor inheritance can be a bit unwieldy in JavaScript, util.inherits makes such code a bit easier to write (and read).

DEVELOPER TOOLS FOR NODE

Below are some key tools widely adopted in the enterprise and in the community for developing Node applications:

DEVELOPMENT ENVIRONMENTS

PRODUCT/PROJECT	FEATURES/HIGHLIGHTS
WebStorm	• Code analysis • Cross-platform • IntelliJ based
Sublime Text	• Goto anything • Customizable • Cross-platform
Atom	• Built on JavaScript • Extensible • Maintained by GitHub
Nodeclipse	• Open source • Built on Eclipse
Cloud9 IDE	• Cloud-based • Collaborative • Debug and deploy
Visual Studio	• Open source • Debug and Profile • TypeScript integration

APPLICATION PERFORMANCE MONITORING

PRODUCT/PROJECT	FEATURES/HIGHLIGHTS
StrongLoop Arc	• Error tracing • Event loop response times • Slowest endpoints
New Relic	• Error rates • Transaction response times • Throughput monitoring
AppDynamics	• Error tracing • Endpoint response time • Historical metrics
Keymetrics	• CPU monitoring • Load balancing • Exception reporting

DEBUGGING

PRODUCT/PROJECT	FEATURES/HIGHLIGHTS
V8 Debugger	• Manual code injection • Breakpoints • Event exception handling
Node Inspector	• Google Blink Dev-Tools based • Breakpoints • CPU & memory Profiling

PRODUCT/PROJECT	FEATURES/HIGHLIGHTS
Cloud9 IDE	• Cloud-based • Code completion • Debug and deploy
WebStorm	• Code analysis • Cross-platform • VCS integration
Nodeclipse	• Code completion • Built-on Eclipse • Tracing and breakpoints

RESOURCES

- StrongLoop website
- StrongLoop technical blog
- Official Node website: nodejs.org
- Node downloads
- Node documentation
- devdocs (searchable documentation)
- Node on GitHub
- Official npm website: npmjs.com
- npm documentation
- Nodeschool
- Stack Overflow

ABOUT THE REFCARD UPDATER

DAVE WHITELEY focused in on Node.js in 2012 as part of Vancouver-based start-up NodeFly. After StrongLoop acquired NodeFly in 2013, he became part of the StrongLoop team and helped grow its presence in the Node.js community. Then he joined IBM after it acquired StrongLoop in 2015. Today, his mission is to help grow the Node.js and open-source communities and spread the word about LoopBack and the OpenAPI Specification.

DZONE, INC.
150 PRESTON EXECUTIVE DR.
CARY, NC 27513
888.678.0399
919.678.0300

REFCARDZ FEEDBACK WELCOME
refcardz@dzone.com

SPONSORSHIP OPPORTUNITIES
sales@dzone.com

ISBN-13: 978-1-936502-77-6
ISBN-10: 1-936502-77-1
50795
9 781936 502776

BROUGHT TO YOU IN PARTNERSHIP WITH
StrongLoop
An IBM Company