Node.js

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

UPDATED BY JUSTIN ALBANO, SOFTWARE ENGINEER, CATALOGIC SOFTWARE
WRITTEN BY DAVE WHITELEY, PERFORMANCE MARKETING MANAGER, IBM

CONTENTS

WHY NODE.JS?

Since its inception in 2009, Node.js (pronounced "Node JS" or "Node" for short) has irrevocably changed the landscape of server-side programming. Prior to its release, server-side programming was commonly accomplished with languages such as Java, Ruby, or Python, combined with frameworks such as Spring or Rails, which differed from those used on the client.

With the creation of Node.js, developers can now use the same language, namely JavaScript, on both the client and server. Additionally, due to the single-threaded event loop architecture of Node.js, developers can also perform IO-intensive operations (such as responding to Hypertext Transfer Protocol, or HTTP, requests) in a much swifter and simpler manner.

While Node.js is not the first technology suite to introduce JavaScript on the server side of the equation or the first to use a single-threaded event loop to handle work, it has been the most effective technology thus far to aggregate these concepts. Combined with the ubiquity of the Node Package Manager (NPM), Node.js has cemented its place as the de facto standard for server-side JavaScript.

ARCHITECTURE

The Node.js architecture can be divided into two main objectives: (1) handling events of interest using the event loop and (2) executing JavaScript on a server rather than in a browser. Prior to understanding these ideas, though, it is essential to understand the differences between synchronous and asynchronous programming.

SYNCHRONOUS VS. ASYNCHRONOUS PROGRAMMING

The main difference between synchronous and asynchronous programming is how each deals with a **blocking** operation, where a task takes a non-trivial amount of time to complete or waits on external input to complete (which may never come and, therefore, the task may never complete). In **synchronous programming**, the thread that handles the task blocks and waits for the task to complete before doing any more work. In **asynchronous programming**, the thread delegates the work to another thread (or the operating system) and continues executing the next instruction. When the background thread completes the task, it notifies the original thread, allowing the original thread to handle the now-completed task.

These differences can be illustrated using an analogy. Suppose a restaurant has a kitchen, a bar, and one waitress. Since the waitress has numerous tables to serve, she cannot take drink orders from a table and wait at the bar until the drinks are done before taking food orders for the table. Likewise, the waitress cannot wait at the kitchen until a submitted food order is completed before serving another table. Instead, the waitress takes drink orders, and if ready, food orders for a table, gives the bartender the drink order, and gives the kitchen the food order. She then proceeds to serve the next table, repeating this process for each table she must serve.

habitat

Get the most of your Node.JS apps with Habitat

TARGET ANY CLOUD, ANY CONTAINER, WITH A SINGLE BUILD.

Habitat from Chef automates the build and packaging of Node.JS apps so you can target any cloud, or any container runtime from a single build output. Connect your GitHub repo to Habitat, check in code, and automatically create a build artifact that can run on AWS, Azure, GCP, using Docker, Kubernetes and more. And the artifact is automatically published with management code for DevOps bliss.

Get started in 10 minutes at http://habitat.sh

CHEF

Between providing the bartender with new drink orders, the waitress checks to see if any drink orders have been fulfilled. If there are fulfilled orders, she takes the drinks to the table that ordered them. Likewise, if the kitchen has completed a food order, the waitress takes the food to the table that ordered it. Although there is only one waitress, she is able to simultaneously serve multiple tables because she offloads time-consuming work to the bartender and the kitchen and does not block, waiting for an order to be fulfilled. When either the bartender or the kitchen completes an order, she handles the completed order by bringing it to the table that made the order.

This is asynchronous programming in action. Had the waitress instead taken drink orders and waited at the bar for the order to be completed before returning to the table to take food orders, she would have been operating synchronously. In the case of Node.js, an event loop (akin to the waitress) is used to cyclically check worker threads or the operating system (akin to the bar or the kitchen) to see if offloaded work has been completed. This helps ensure that performing a blocking operation in Node.js does not bring the system to a halt until the operation completes.

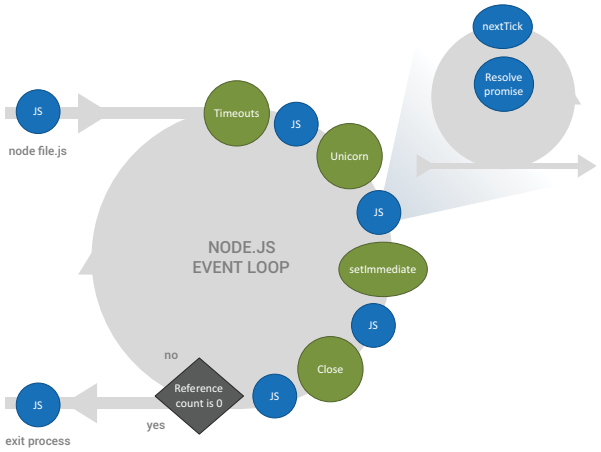
THE EVENT LOOP

Unlike many large-scale frameworks, Node.js application code executes using a single thread. Instead of creating a series of threads and assigning each a task, Node.js uses an event loop, allowing a developer to register **callback functions** to execute when an event of interest occurs. For example, instead of creating a new thread to block and handle each HTTP request received by the server, a developer registers a function with the Node.js framework that Node.js uses to handle the request. This allows Node.js to asynchronously execute the callback function whenever it deems most effective.

This scheme has many important advantages, including:

1. The developer is not responsible for handling the nuances of multithreading, such as deadlock avoidance and race conditions.
2. It is possible to make non-blocking IO calls, or if desired, block with a handful of timer-based Application Programming Interfaces (APIs)..
3. Callbacks encapsulate the complete logic for handling an event, allowing Node.js to queue the handling logic to be executed when most effective.

The core of the Node.js event-driven architecture is the event loop, illustrated on the next column.



The Node.js event loop starts by processing a single JavaScript file, which in turn registers callbacks for its events of interest. For each callback function registered, a reference counter is incremented, allowing Node.js to track how many callbacks are still left to be executed.

As the event loop proceeds, it executes each step, or **phase**, in the loop (the green and blue circles), handling each of the events associated with each phase. Once the callback reference count reaches 0, the event loop exits and the Node.js process terminates. If the reference count is greater than 0 at the end of a loop iteration, the loop executes again. Commonly, the reference count will not decrement to 0, and thus, Node.js will continue to handle events (such as HTTP requests) indefinitely, unless the process is manually terminated.

EVENT LOOP PHASES

The green circles in the event loop represent Node.js code that is executed, while the blue circles represent application code that is executed in response to events that occur within the Node.js code. For example, the Timeout phase represents the Node.js code that is executed to find timer events that have expired. Once these events have been collected, the blue application code is executed to handle these timeout events.

The four phases of the event loop are:

1. **Timeout**: Checks to see if any timeouts have occurred (i.e. registered using the `setTimeout` or `setInterval` functions). For example, if a timer is set in application code for 200 ms and at least 200 ms has expired since the initiation of the timer, the timer is declared expired; once the expired timers have been queued, the callback associated with each timer is executed. This process continues until all expired timers have been handled.
2. **Unicorn**: IO events, such as completed file system and network events, are handled. This process continues until all IO events for which callbacks have been registered have been handled.

3. `setImmediate`: Events declared via the `setImmediate` function in the previous loop iteration are executed. This process continues until all `setImmediate` callback functions have been executed.

4. **Close**: Internal clean-up is executed (i.e. closing sockets) and any callbacks associated with these events are executed. This process continues until all associated callbacks have been executed.

Once all of the callbacks associated with the Close phase have been executed, the callback reference count is checked. If the count is greater than 0, the loop is repeated; if the reference count equals 0, the loop is exited and the Node.js process is terminated.

RESOLVED PROMISES AND `nextTicks`

Within each of the application code steps (blues circles), all callbacks — called **promises** in this context — are executed and all callback functions registered using the `nextTick` function are executed. Once all registered promises and `nextTick` callbacks have been completed, the user code circle is exited and the next phase of the event loop is entered.

Note that promises and `nextTick` callbacks are executed in a sub-loop, as promises and `nextTick` callbacks can introduce more callbacks to be executed in order to handle the current promise or `nextTick` callback. For example, recursively calling `nextTick` within a `nextTick` callback will queue up a callback that must be completed before execution of the event loop can continue. Note that developers should be wary of making recursive `nextTick` calls, as this may starve the event loop, not allowing the next callback to be completed or next phase of the loop to be entered. See the documentation on understanding `nextTick` for more information.

`nextTick` VS. `setImmediate`

One of the unfortunate blunders in Node.js is the naming of the `nextTick` and `setImmediate` functions (see the morning keynote at Node.js Interactive Europe 2016 for more information). The former is used to execute the supplied callback at the completion of the current callback, while the latter is used to defer the execution of the supplied callback until the next iteration of the event loop.

Many developers rightfully confuse these functions, reasoning that *tick* represents an iteration of the event loop and therefore, *next tick* is the next iteration. Likewise, *immediate* is often inferred to mean the current callback, and hence, the callback supplied to `setImmediate` is assumed to be completed immediately following the current callback. While understandable, this is *not* how Node.js works. Although this misnomer has been a known problem since the early days of Node.js, the frequent use of both functions in application code has precluded renaming these functions.

Developers should, therefore, be careful not to confuse the use of these two functions and accidently apply them in the reverse manner.

`async` AND `await`

Although promises allow for deferred, non-blocking execution, they also have a major disadvantage: They are syntactically scrupulous when chained. For example, creating a chain of promises in Node.js (or JavaScript in general) results in code that resembles the following:

```
someObject.call()
    .then(response1 => {
        anotherObject.call(response1)
            .then(response2 => {
                thirdObject.call(response2)
                    .then(response3 => {
                        // Do something with 3rd response
                    })
                    .catch(console.error);
            })
            .catch(console.error);
    })
    .catch(console.error);
```

Asynchronous chains such as these (disparagingly referred to as the pyramid of doom) can be lead to unreadable code and difficult to debug applications. With the advent of version 5.5 of the V8 JavaScript engine (see below) in October 2016, Node.js applications can now utilize the `async` and `await` keywords introduced in JavaScript ES2017. This allows the above code to be reduced to the following:

```
const runCalls() = async() => {
    try {
        const response1 = await someObject.call();
        const response2 = await anotherObject.
        call(response1);
        const response3 = await thirdObject.
        call(response2);
    catch (err) {
        console.error(err);
    }
};
runCalls();
```

The use of the `async` keyword creates an asynchronous function that removes much of the boilerplate code and repetition required for the resolution of promises. Within asynchronous functions, the `await` operator can be used to wait for a promise to be resolved and return its resolved value. In the snippet above, response1 is not assigned until the promise associated with `someObject.call()` resolves, after which the response1 is assigned the value of the resolved `someObject.call()` promise. Likewise, `anotherObject.call(response1)` is not executed until `response1` has been assigned after the resolution of `someObject.call()`. If any of the `awaited` functions throws an error (which was previously caught using the `.catch(...)` syntax), the catch clause of the try-catch block will handle the error (since the `await` operator throws any errors received when resolving the promise it is awaiting on).

It is important to note that the above code, although seemingly

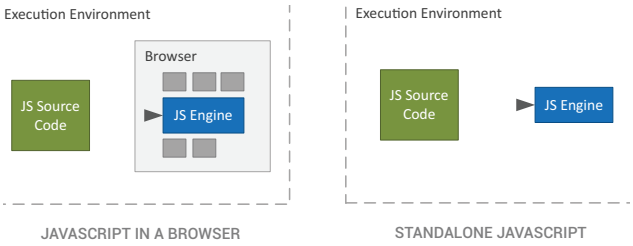
asynchronous, is actually asynchronous. The `await` operator simply abstracts an underlying promise (e.g. one returned from `someObject.call()`) and allows code to be written to handle asynchronous code in a *seemingly* synchronous manner.

SERVER-SIDE JAVASCRIPT

The second key to the success of Node.js is that developers write Node.js server applications in JavaScript. This allows developers to write server-side applications in the same language in which they are accustomed to writing client-side applications. Apart reducing the learning curve, this allows developers to reuse portions of client code in server applications and vice-versa.

THE V8 JAVASCRIPT ENGINE

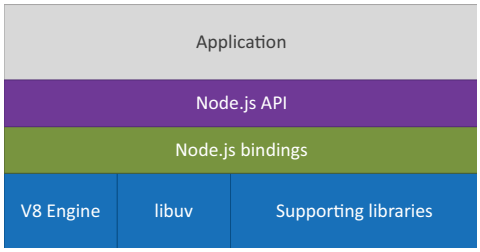
The ability of Node.js to execute JavaScript on the server is achieved by deploying the **Chrome V8 JavaScript engine** in a standalone environment rather than within the confines of a browser. While JavaScript engines are often relegated to browsers in order to interpret and execute JavaScript code retrieved from a server, Node.js incorporates the V8 JavaScript engine to execute JavaScript in any environment that supports Node.js, as illustrated in the figure below.



THE NODE.JS STACK

Since the V8 JavaScript engine is written in C++, Node.js includes a **binding layer** that allows application code to be written in JavaScript but execute natively through the JavaScript engine. Above the binding layer, Node.js includes an **API layer** — written in JavaScript — providing core functionality, such as managing sockets and interfacing with the file system through Node.js (see the list of Node.js APIs below for more information). At the top of the stack, developers create applications using the Node.js API layer in conjunction with third-party libraries and APIs.

Along with the V8 JavaScript engine, Node.js also includes the **Unicorn Velociraptor Library (libuv)**, which houses the Node.js event loop and the internal mechanisms used to process registered callback functions. Node.js also includes a set of **supporting libraries** that allow Node.js to perform common operations, such as opening a socket, interfacing with the file system, or starting an HTTP server. While libuv and the Node.js supporting libraries are written in C++, their functionality can be accessed by Node.js applications using the API and bindings layer, as illustrated in the next column.



ISOMORPHIC JAVASCRIPT

The ability of Node.js to render JavaScript on the server has created a natural extension in web development: **Isomorphic JavaScript**. Sometimes termed **universal JavaScript**, isomorphic JavaScript is JavaScript that can run on both the client and the server. In the early days of webpages, static Hypertext Markup Language (HTML) pages were generated by the server and rendered by the browser, providing a simplified User Experience (UX). While these webpages were slow to render, since a single click required the loading of an entirely new page, Search Engine Optimization (SEO) was straightforward, requiring that SEO crawlers parse a static HTML page and follow anchored links to obtain a mapping of a website.

As JavaScript was introduced to the client and rendered by the browser, Single-Page Applications (SPAs) became the defacto standard. Compared to static webpages, SPAs allow users to quickly and seamlessly transition from one page to another, requiring that only JavaScript Object Notation (JSON) or eXtensible Markup Language (XML) data be transferred from the server to the client, rather than an entire HTML page. While SPAs do have many speed and UX advantages, they commonly require an upfront load time (which is usually accompanied by a loading message) and are difficult to tune for SEO, since crawlers can no longer obtain a static view of a page without rendering a large portion of the SPA themselves.

Isomorphic JavaScript is a happy-medium between these two extremes. Instead of performing all of the page generation on the server or all on the client, applications can be run on both the client and the server, providing for numerous performance optimizations and SEO tunings. Companies such as Twitter have even gone so far as to **reduce initial load times to 1/5 of those experienced with client-only rendering** by offloading the generation of webpages to the server, removing the need for the client to load the entire SPA before rendering the first page displayed to users. Apart from performance and SEO improvements, isomorphic JavaScript allows developers to share code between the client and server, reducing the logical duplication experienced when implementing many large-scale web applications (such as domain and business logic being written in one language on the server and repeated in another language in the SPA). For more information, see **Isomorphic JavaScript: The Future of Web Apps by Spike Brehm** (of Airbnb).

QUICKSTART

The first step to using Node.js is downloading the Node.js installer, which can be found on the official **Node.js website** (the source code for Node.js can also be obtained this Downloads page). Note that the installer provides both the Node.js command line tool and the Node Package Manager (NPM) tool.

Once the installer is ready, the installation can be verified by checking the version of Node.js using the following command:

```
node -v
```

To run a JavaScript file using Node.js, simply execute the following command, where file.js is the name of the JavaScript file to be executed:

```
node file.js
```

For example, a file named index.js can be created with the following content:

```
console.log("Hello, world!");
```

…and executed using the following command:

```
node index.js
```

Executing this file results in the following output:

```
Hello, world!
```

Although this is a pedagogical example, the Node.js APIs combined with NPM provide a rich set of supporting code and infrastructure that facilitates the creation of powerful applications. For example, a simple HTTP server can be created using the following code:

```
const http = require('http');
http.createServer((request, response) => {
    const { method, url } = request;
    response.writeHead(200, {'Content-Type': 'application/
    json'});
    response.write(`{"message": "Received a ${method}
    request with context path '${url}'"}`);
    response.end();
})
.listen(8080);

console.log("Started server");
```

The pertinent naming of the functions in the Node.js APIs makes the above code nearly readable in English, where a server is created (listening on port `8080`) that handles incoming requests by setting the status of the response header to `200` (OK) and the content type of the response to `application/json`. The handler then writes a JSON message containing the request method and URL to the response body and closes the response body with the end method.

Although Node.js is single-threaded, listening on the specified port *will not* block execution. Instead, a callback is registered that executes the above handler logic when an HTTP request is received on port `8080`. Once the callback is registered, the phrase `Started server` is outputted.

Note that this application requires that the http package be installed using npm `install --save http` (see the NPM section for more information).

COMMON APIS

Node.js ships with a vast array of powerful APIs that can significantly reduce application development time. A complete list of APIs included with Node.js can be found in the **Node.js API documentation**.

API Name	Description
Buffer	An optimized abstraction API that allows developers to interact with octet streams. Since the release of ECMAScript 2015 (ES6), this API has changed significantly (primarily with respect to the instantiation of Buffer objects); developers should consult the official documentation closely to ensure that best practices (such as instantiation through factory methods) are followed. Note that the Buffer API is a global API and therefore does not necessitate the require function to import this API.
Child Process	A collection of synchronous and asynchronous functions for spawning new processes and executing native commands (i.e. `ls`) and command line scripts (i.e. `.bat` and `.sh`).
Crypto	A cryptographic library that includes wrapper functions for OpenSSL hashes, keyed-Hash Message Authentication Codes (HMACs), ciphers, digital signatures, and the Diffie-Hellman Key Exchange (DHKE). Note that it is possible to build Node.js without cryptographic support and, therefore, an error can be thrown when importing the cryptographic API through `require('crypto')`.
Debugger	An inspection framework that allows developers to debug Node.js application code and interface with the V8 Engine inspector APIs.
Events	An asynchronous library designed for emitting and handling named events. This library includes abstractions for executing handlers in an asynchronous or synchronous (executing listeners in the order in which they were registered) and handling repeated events.
HTTP	A simple yet powerful API for handling incoming HTTP requests. This API includes straightforward methods to establish an HTTP handler and configure desired responses while also allowing developers to adjust more fine-grained options, such as timeouts and permissible numbers of concurrent sockets that can be open to handle requests.

Modules	An API that allows developers to interact with Node.js modules (such as imported Node.js files), similar to classpath management in Java or package management in Python. This API allows developers to control common aspects of a module, such as exports, as well as more complicated aspects of modules, such as caching. The module API is a global API and does not require an explicit import of the form `require('module')`.
Net	A network API that provides for the creation clients and servers for Transmission Control Protocol (TCP) connections, as well as Inter-process Communication (IPC) sockets in both Windows and Unix-based systems. In the specific case of HTTP servers, the HTTP module should be favored; when more general TCP sockets are needed, developers should use the Net module.
Process	An API that allows developers to interface with the current Node.js execution process context. This API includes functionality for managing lifecycle events (i.e. exiting the current Node.js process), extracting information about the current execution environment (i.e. supplied command line arguments), and interaction with related processed (i.e. sending IPC messages to parent processes).
Stream	An abstraction API for stream data. This API is used throughout the core APIs of Node.js (i.e. HTTP requests and standard output — `stdout` — are both streams) and is very useful for understanding the fundamental implementation of other Node.js APIs.

NODE PACKAGE MANAGER (NPM)

Although Node.js by itself is a powerful technology for server application development, the addition of NPM facilitates the development of truly large-scale applications. Akin to Maven or Gradle for JavaScript, NPM is a dependency management tool that allows a developer to specify the dependencies associated with an application, install these dependencies in any supported environment, and import these dependencies within the application code.

Following the standard dependency management process, NPM provides three principal components:

1. **Dependency specification**: A standard file format providing developers with a means of specifying the dependencies associated with an application, as well as specifying metadata that allows the application to be used as a dependency (called a **package**) for another application; this specification includes information, such as versioning, that allows a developer to precisely specify the package he or she wishes to include as a dependency.

2. **Dependency tool**: A program that parses the dependency specification and downloads the dependencies to a known location from which the dependencies can be imported and used within the application.

3. **Repository**: A central location containing packages provided by developers that can be used as dependencies in an application; this repository is in a well-known location, allowing the developer to specify the name of dependencies without having to obfuscate the specification with the remote location of the package.

QUICKSTART

NPM is included with the official Node.js installation and does not require any further configuration to use. To create a new NPM package, change directory (using the cd command) to the directory containing the JavaScript source files to be packaged and execute the following command:

```
npm init
```

Using the defaults, NPM generates a **package.json** file (that constitutes the dependency specification as a JSON file) that contains the following, where `<package-name>` is the name of the directory in which the initialization command above was executed (see the `package.json` documentation for the constraints on NPM package names):

```
{
    “name”: “<package-name>”,
    “version”: “1.0.0”,
    “description”: “”,
    “main”: “index.js”,
    “scripts”: {
       “test”: “echo \”Error: no test specified\” && exit 1”
    },
    “author”: “”,
    “license”: “ISC”
}
```

A new dependency can be added using the NPM tool. For example, to add the colors package to the project, execute the following command:

```
npm install --save colors
```

The `install` command simply installs the supplied package name to the `node_modules/` directory (in the current directory in which the command was executed; see the How to Install Local Packages documentation), while the `--save` flag instructs NPM to update the `package.json` file to include the new dependency (version reflects most recent package at the time of writing):

```
{
...
    “dependencies”: {
        “colors”: “^1.1.2
    }
}
```

This dependency can now be utilized through the Node.js require function in a JavaScript source file:

```
var colors = require('colors');
console.log("Hello, world!".red);
```

NPM can also be used to execute sets of commands called **scripts**. For example, we can create a start script in the `package.json` file that can be used to execute the JavaScript source code above (assuming it is contained in `index.js`):

```
"scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node index.js"
}
...
```

This script can then be executed using the following command:

```
npm start
```

This in turn executes the script, which prints `Hello, world!` in red to the console.

THIRD-PARTY MODULES AND FRAMEWORKS

The growth of Node.js has facilitated the creation of numerous supporting NPM packages and frameworks built on top of the Node.js environment, including the following:

Package Name	Description
sails	A Model-View-Controller (MVC) framework, akin to Ruby on Rails for Node.js, for production or enterprise-level applications. See the Sails.js documentation for more information.
express	A minimalist web application framework that supports the creation HTTP-based APIs. See the Express.js website for more information.
colors	Stylizer for console text, including the foreground color, background color, weight (i.e. bold), transforms (i.e. underline or strikethrough), etc.
moment	Date abstractions including functionality for comparing, parsing, displaying, and manipulating dates. Also includes support for date locales.
async	Simple but rich functions for handling asynchronous flow control and functional programming in JavaScript. See the Async documentation for more information.
rxjs	Support for observable behavior and functional programming. See the ReactiveX JavaScript documentation for more information.
browserify	Bundler for Node.js modules and its associated dependencies for ease of access through browsers. See the Browserify website for more information.

A list of the most depended upon packages can be found at the official NPM repository. For more Node.js frameworks, see the Node Frameworks website.

GOVERNANCE AND CONTRIBUTION

The Node.js Foundation is primarily responsible for the governance of the Node.js project, although it does not play an active role in the day-to-day development of Node.js. Rather, the Foundation is responsible for setting the business and technical direction of the project, managing Intellectual Property (IP), and planning and coordinating events. Subsidiary to the Foundation, the Node.js Technical Steering Committee (TSC) is responsible for technical oversight of the top-level projects within Node.js.

Unlike the Foundation and the TSC, the Node.js Core Technical Committee (CTC) does handle technical nuances within Node.js, but only when required (such as making decisions when a community or collaborator consensus cannot be reached).

CONTRIBUTION WORKFLOW

Any developer abiding by the Node.js Code of Conduct may contribute to Node.js by complying with the following process:

1. Fork the Node.js Github project.
2. Create a new branch.
3. Make changes.
4. Push the branch.
5. Open a Pull Request (PR).
6. Receive feedback and make corresponding adjustments.

Once approval has been given by at least one Node.js Collaborator and a Continuous Integration (CI) test run has been completed on the change (and barring any objections from another contributor), the PR is merged (landed in the Node.js vernacular) into the Node.js codebase. A PR must remain open for a minimum of 48 hours (72 hours on weekends) before it can be landed, but may take longer depending on the nature of the change. More information can be found at the Node.js Contribution documentation.

RELEASE SCHEDULE

The release of new versions of Node.js follow a standard cadence, subject to the following guidelines:

- Major releases occur every six months.
- Even releases (e.g. v4, v6, v8, etc.) occur in April.
- Odd releases (e.g. v3, v5, v7, etc.) occur in October.
- Each odd release transitions the previous even release to a **Long Term Support (LTS)** plan (e.g. release of v11 transitions v10 to LTS).
- LTS versions will be support for an 18-month period after the date of release and will transition to **Maintenance Mode** for a 12-month period at the expiration of the LTS plan.

This schedule ensures that only two major releases are concurrently supported by an LTS plan for a maximum of six months. More information can be found on the **Node.js Release Working Group page** and the **Node.js Release documentation**.

RESOURCES

- **Node.js Official Website**: The official Node.js website containing download links for the latest version of Node.js, as well as the official API documentation for Node.js.
- **Node.js Developer Guides**: The official Node.js developer guides, which include detailed and pragmatic examples of how to use Node.js in a practical environment, including samples on creating a Hello World application, debugging existing applications, and understanding the minutia of the Node.js APIs.
- **NodeSchool**: An open source project dedicated to teaching developers using Node.js as a programming basis; NodeSchool hosts numerous international events and there are numerous international chapters throughout Asia, Europe, North America, Latin America, Africa, and the South Pacific.
- **Node.js Github**: The official Node.js Foundation GitHub, containing the source code for Node.js and its associated projects; also used to manage the open and resolved issues associated with Node.js and its companion projects.
- **Node.js LinkedIn Group**: The official Node.js LinkedIn Group (invitation only).
- **NPM Official Website**: The official NPM website containing download links for the latest version of NPM and documentation for getting started with NPM, installing existing packages, creating new packages, and using the NPM command line tool; also contains a complete list of the packages uploaded to the NPM repository.

Written by Justin Albano, Software Engineer, Catalogic Software

Justin is responsible for building distributed catalog, backup, and recovery solutions for Fortune 50 clients, focusing on Spring-based REST API and MongoDB development. Justin has worked on a wide array of professional tasks, including Angular-based UI and cloud-based backend projects for the Federal Aviation Administration, safety-critical and real-time systems for the United States military, and commercial backup management software for Catalogic Software. Justin has earned a BS and Master's in Software Engineering at Embry-Riddle Aeronautical University and when not working or writing, he can be found at the ice rink, watching hockey, drawing, or reading.

DZone, Inc.
150 Preston Executive Dr. Cary, NC 27513
888.678.0399 919.678.0300