StrongLoop™

An Introduction to Node.js

A brief introduction to what Node.js is, how it works, use cases, plus what's new in the latest release, v0.12.

Table of Contents

Introduction

In this technical paper we will cover what Node is and how it works. We'll also cover the typical use cases that make server-side JavaScript so appealing to organizations, as well as, what's new in the latest release plus some of the more popular APIs and modules that organizations are using today.

What is Node?

The official description according to the nodejs.org website is as follows:

"A platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications."

Translation

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

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

Translation

Developing distributed, multi-threaded applications using traditionally synchronous languages can be complex and daunting, Node leverages JavaScript's asynchronous programming style via the use of event loops with callbacks to make applications naturally fast, efficient, and non-blocking. If you know JavaScript, you already know quite a bit about Node!

"Perfect for data-intensive real-time applications that run across distributed devices."

Many application performance problems stem from being I/O bound. Because Node is designed to be non-blocking and event driven when manipulating data, reading files or accessing APIs, it's ideally suited to be distributed across multiple process and machines in a network. Popular uses for Node include web servers, API gateways and backends for mobile applications.

Because it's not limited to one connection per thread like most web server architectures, Node scales to many thousands of concurrent connections. This makes it perfect for writing Mobile and Internet of Things APIs which must interact with many devices in small increments, often holding open a connection while the device connects over a slow network.

Node is JavaScript on the server

Node allows 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 frontend developers, the ability to use this same skill to program server-side tasks, in theory, increases a developer's productivity. It may also reduce the need for separate languages or code bases between front-end and backend applications.

How does Node work?

Synchronous vs Asynchronous programming

C and Java traditionally use synchronous I/O, which means time is wasted waiting. You can get around this by writing multithreaded 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. Node by contrast is a single-threaded way of programming evented, non-blocking, asynchronous I/O applications.

Synchronous vs. asynchronous analogy

In order to understand non-blocking I/O, let's picture a common scenario. Suppose you are at a restaurant with friends.

A typical experience at a restaurant would be something like this:

1. You sit at a table and the server grabs your drink order.
2. The server goes back to the bar and passes your order to a bartender.
3. While the bartender is working on your drink, the server moves on to grab another table's drink order.
4. The server goes back to the bar and passes along the other table's order.
5. Before the server brings back your drinks, you order some food.
6. Server passes your food order to the kitchen.

7. Your drinks are ready now, so the server picks them up and brings them back to your table.
8. The other table's drinks are ready, so the server picks them up and takes them to the other table.
9. 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. First, you order something. Then, the server goes on to process your order and return it to you when it's ready. Once the order is handed off to the bar or kitchen, the server is free to get new orders or to deliver previous orders that are completed. Notice that at no point in time is the server doing more than one thing. They can only process one request at a time. This is how non-blocking Node.js applications work. In Node, your application code is like 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 the work off to the operating system, which takes care of processing the rest. Back to our restaurant example, if every time the server got an order request they had to wait for the bar/kitchen to finish before taking the next request, then the service for this restaurant would be very slow and customers would most likely be unsatisfied. This is how blocking I/O works.

Event loop concurrency model

Node leverages a browser-style currency model on the server. As we all know, JavaScript was originally designed for the browser where events are things like mouse movements and clicks. 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 might mean that instead of a program waiting to return a response until it queries databases, accesses files or connects to various APIs, it immediately moves on to the next unit of work until the event returns with whatever response was asked of it.

Instead of blocking entire programs waiting for I/O to complete, the event loop allows applications to move on and wait for events in order to continue the flow of the program. In this way Node achieves multitasking more efficiently than using threads.

Event loop analogy

Think of event loops as some delivering mail. They collect the letters or events from the post office (server). These letters can be equated to events or incoming requests that need to be handled i.e. delivered. The letter carrier goes to every mailbox in his area and delivers the letters/events to the destination mailboxes. These destination mailboxes can be equated to JavaScript functions or downstream.

The postman does not wait at the mailbox to receive a reply. When the user of the mailbox responds with a letter, on his routes, he picks it up. Every mailbox has a separate route and routes here can be thought of as the callback. Every incoming letter/request has a callback associated, so that a response can be sent anytime when ready (asynchronously) using the callback routes.

Event Loop code example

Let’s look at a simple example of asynchronously reading a file into a buffer. This is a two step process in which first there is a request to read the file, then a callback to handle the file buffer (or error) from the asynchronous file read.

```
var fs = require(‘fs’);
fs.readFile(‘my_file.txt’, function (err, data) {
     if (err) throw err;
     console.log(data);
});
```

The second argument to readFile is a callback function which runs after the file is read. 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 result goes to V8. It then goes through the Node Bindings in the form of a callback with the buffer. In the callback shown the data argument has the 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/');
```

Architecture

There are several building blocks that constitute Node. First, Node encapsulates libuv to handle asynchronous events and Google's V8 to provide a run-time for for JavaScript. Libuv is what abstracts away all of the underlying network and file system functionality on both Windows and POSIX-based systems like Linux, Mac OSX and Unix. The core functionality of Node, modules like Assert, HTTP, Crypto etc, reside in a core library written in JavaScript. The Node Bindings provide the glue connecting these technologies to each other and to the operating system.

What is Node good for?

Web Applications

Node is becoming popular for web application because web applications are now slowly shifting from purely server side rendered to single page application to optimize the user experience on the client.

Reasons why:

- Single page applications have the MVC paradigm self contained within the browser so that the only server side interaction that is required can be through an efficient API for RPC invocation of server side functions and data behind the firewall or in the cloud

- 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 and Node are all built on the common dynamic scripting language of JavaScript

Examples of frameworks for Node:

- Express
- Sails.js
- Compound.js
- Flatiron.js
- Derby.js
- Socketstream.js
- Meteor
- Tower.js

Mobile Backends

Node is popular for backends, especially those required by mobile applications. As an I/O library at its heart 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, the re-use of code written in JavaScript on the client side can be leveraged on 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
- Node's use of JavaScript as a scripting language makes it easy to normalize data into mobile APIs

Examples of mobile backends built in Node:

- Parse (Proprietary)
- LoopBack (Open source)
- FeedHenry (Proprietary)
- Appcelerator Cloud Services (Proprietary)

API Servers

Node is popular for backends, especially those required by mobile applications, why?

Reasons why:

- Node utilizes JSON as the content-type for data modeling and the data payload itself. This lightweight format is already evolving to become the most dominant standard for REST APIs
- Node's rich ecosystem consists of asynchronous libraries that can be easily utilized to handle massive concurrency for the API use case

Examples of open source API Servers built in Node:

- Restify
- Deployd
- LoopBack
- actionhero.js

What are Node's performance characteristics?

Everyone knows benchmarks are a specific measurement and don't account for all cases. For example, sometimes Java is faster sometimes Node is. 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 go 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 recent examples of Node performance benchmarks and related posts:

- PayPal
- Linkedin
- DZone

How do I install Node?

The good news is that installers exist for a variety of platforms including Windows, Mac OS X, Linux, SunOS and of course you can compile it yourself from source. Offical downloads are available from the nodejs.org website here:

http://nodejs.org/download/

How can I make Node useful?

What is npm?

Node Package Manager or "npm" as it is more commonly known, is the package manager for Node you leverage at the command line that manages dependencies for your application. npmjs.org is the public repository where you can obtain and publish modules.

How does npm work?

In order for your Node application to be useful it is going to need things like libraries, web and testing frameworks, data-connectivity, parsers and other functionality. You enable this functionality by installing specific modules via npm.

There's nothing to install to start using npm if you are already running Node v0.6.3 or higher. (If you want a specific version of npm or customized paths, you can get more detailed installation instructions from the npmjs.org site.)

You can install any package with this command:

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

Some popular and most depended on modules include...

express

Express is 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.

Built on connect, you can use only what you need, and nothing more. Applications can be as big or as small as you like, even a single file. Express does not force you to use any specific ORM or template engine. With support for over 14 template engines via consolidate, you can quickly craft your perfect framework.

async

Async is a utility module which provides straight-forward, 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…) as well as some common patterns for asynchronous control flow (parallel, series, waterfall…). All these functions assume you follow the Node convention of providing a single callback as the last argument of your async function.

request

Request is a simplified HTTP request client. It supports HTTPS and follows redirects by default.

underscore

underscore is JavaScript's functional programming helper library. It's a utility-belt library for JavaScript that provides support for the usual functional suspects (each, map, reduce, filter, etc.) without extending any core JavaScript objects.

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. It also enhances WebSockets by providing built-in multiplexing, horizontal scalability, automatic JSON encoding/decoding, and more.

mocha

Mocha is a feature-rich JavaScript test framework running on Node and the browser, making asynchronous testing simple and fun. Mocha tests run serially, allowing for flexible and accurate reporting, while mapping uncaught exceptions to the correct test cases.

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.

colors

Adds colors to your Node console. This is a small, but very important feature to improve the readability of your console.

What is new in Node v0.12?

Streams3

Streams2 was introduced in Node v0.10 with the idea that it would be easier to write things like parsers. But, its introduction did have one obvious downside which was that it prevented developers from tapping into other streams and pulling data into a console for example. This ultimately proved to be a less than ideal solution for many developers. With streams3 in v0.12, the idea is to make both the original streams implementation and streams2 work well together by defining what happens when the two streams modes are mixed. This means that everything streams2 does is mapped to the original streams implementation and vice versa. In a nutshell:

- A data event is fired every time `read()` is called.
- `resume()` calls read repeatedly, `pause()` stops that
- `pipe(dest)` and `on('data', fn)` resume the stream

To learn more about Streams3 check out the official docs.

Round-robin clustering

Prior to v0.12, Node's round-robin functionality didn't distribute incoming connections evenly, although that was the expectation everyone had. Node used a technique which just about all web servers have used at one time or another. The way it typically worked was, a developer would bring up a server and start a few processes that would be made ready to accept new connections. Under the hood, when a new connection was required, all the processes would race to accept the connection.

In theory, this sounded great and should have scaled well, but in practice most operating systems, specifically Linux-based systems tried to defeat this scheme.

Instead of every available process being considered for a connection, there was the tendency to pick the same process each the time. This meant that the load-balancing scheme didn't work as efficiently as it could. Now, with the new round-robin scheme implemented in v0.12, the master process accepts all the connections and *it* decides which worker gets to send a response. For example:

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

// This is the default:
cluster.schedulingPolicy = cluster.SCHED_RR;
// .. or Set this before calling other cluster functions.
cluster.schedulingPolicy = cluster.SCHED_NONE;

// Spawn as many workers as there are CPUs in the system.
for (var i = 0, n = os.cpus().length; i < n; i += 1)
 cluster.fork();
```

Note that the new round-robin scheme is on by default on all operating systems except for Windows. To learn more about the clustering feature you can read the official docs and a technical blog by Node contributor Ben Noordhuis, that dives deep into the feature.

VM improvements

For awhile now, Node has supported the idea of being able to run a sandbox or VM inside of itself. However, there were two problems that many developers encountered when they tried to use the feature:

- It was difficult to create a JavaScript sandbox in Node
- Node APIs assumed a single context

For review, a VM allows you to create a context. An example of a context might be an iframe inside of a browser. In the iframe, JavaScript can run isolated from the embedding page. This is quite useful when you think about using something like jsdom (a JavaScript implementation of the W3DC DOM) which allows you to render pages in Node without the use of actual browser.

In v0.12, Contextify was pulled into the Node core so you can now create a Contextify-based sandbox. This is basically a JavaScript sandbox where you can define what the global methods are and can communicate back to the process safely, as potentially untrusted JavaScript is now running in isolation. As a result, Node APIs have been made context-aware. In theory, you can have another version of Node running in isolation and it should just work. This work is a step towards full isolation support which would allow for things like threads in JavaScript. Not threads that access shared data, but instead of forking child processes you could run multiple Node VMs on different threads.

To learn more about the updates to VM, you can read the official docs.

execSync

Although it might be a bit odd to think about adding these synchronous features to Node, many developers have been implementing various hacks to get precisely this type synchronous behavior. execSync and spawnSync work by running a process and then blocking while it runs and then exiting when the function returns. The motivation for putting this feature into v0.12 was that an increasing amount of developers were using Node for more than just writing servers.

For example, many developers are using Node as a replacement for shell scripting. Grunt is a perfect example of this. Grunt is like make, in that it allows you to run small tasks to set things up on your operating system. Under the hood it relies heavily on shelljs. Shelljs goes to great lengths to emulate execsynch although Node doesn't actually support it. For example:

```
var child_process = require('child_process');
var fs = require('fs');

function execSync(command) {
 // Run the command in a subshell
 child_process.exec(command + ' 2>&1 1>output && echo done!
> done');

 // Block the event loop until the command has executed.
 while (!fs.existsSync('done')) {
   // Do nothing
 }

 // Read the output
 var output = fs.readFileSync('output');

 // Delete the output and done files
 fs.unlinkSync('output');
 fs.unlinkSync('done');

 return output;
}
```

The above code isn't particularly efficient. As of v0.12, the execSync and spawnSync APIs are supported. How it works under the hood is that a nested loop is spawned in libuv, which only reads the output and sleeps when nothing is happening. For example:

```
var spawnSync = require('child_process').spawnSync;
var result = spawnSync('cat',
                      ['-'],
                      { input: 'hello world!',
                        encoding: 'utf8' });

console.log('exit code: %d', result.exitCode);
console.log('output: %s', result.stdout);
console.log('error: %s', result.stderr);
```

To learn more about the spawnSync and execSync APIs, you can read this technical blog by Node contributor Bert Belder that dives deep into the feature.

Profiling APIs

Prior to v0.12, profiling could only be enabled at startup and heap dumps required a native add-on. In the latest release, APIs have been added to address these issues so that if you wanted to monitor gc behavior for example, you could use the following:

```
var v8 = require('v8');

v8.cpuProfiler.setSamplingInterval(1);
v8.cpuProfiler.start();

v8.on('gc', function() {
 console.log('Garbage collection just happened!');
});
```

To learn more about the profiling APIs, you can read more about the profiling APIs in this technical blog by Node contributor Ben Noordhuis, that dives deep into the performance optimizations in the latest release.

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, please consult the Node API documentation.

Assert

This module is used for writing unit tests for your applications, you can access it with `require('assert').`

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.

C/C++ Addons

Add-ons are dynamically linked shared objects. They can provide glue to C and C++ libraries. The API (at the moment) is rather complex, involving knowledge of several libraries including V8 JavaScript, libuv, internal Node libraries and others.

Child Process

Child processes are 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.

Console

`Console` is used for printing to `stdout` and `stderr`. Similar to the console object functions provided by most web browsers, here the output is sent to `stdout` or `stderr`. The console functions are synchronous when the destination is a terminal or a file (to avoid lost messages in case of premature exit) and asynchronous when it's a pipe (to avoid blocking for long periods of time).

Crypto

`Crypto` provides 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, and allows you to navigate source files, set breakpoints and edit variables and object properties among other things.

DNS

Contains functions for doing regular or reverse DNS lookups (e.g. of a domain name to an IP address) and fetching DNS records (e.g. A, CNAME, MX) for a domain.

Domain

Domains provide a way to handle multiple different IO operations as a single group. If any of the event emitters or callbacks registered to a domain emit an error event, or throw an error, then the domain object will be notified, rather than losing the context of the error in the process.on('uncaughtException') handler, or causing the program to exit immediately with an error code.

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 that, 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, this refers to 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!');
});
```

Which returns emitter, so calls can be chained.

File System

File system interaction functions like reading and writing files and directories, moving, copying and renaming files. Some functions have synchronous versions alongside the normal asynchronous ones, as noted in their names. These are useful in the setup phases of an application’s execution, where speed is unimportant and the simplicity of a synchronous function is desired.

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 for the creation of HTTP servers and have them listen on a given port. This 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.

HTTPS

HTTPS contains functions for creating HTTPS servers or requests. This is regular HTTP secured with SSL. See also the TLS/SSL module.

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, Net is the API you'll want to work with.

OS

Provides a few basic operating-system related utility functions like the return of the default directory for temp files, endianness of the CPU, the hostname, memory details and similar functions.

Path

Complements the File System module by providing functions to manipulate paths and filenames, resolve relative paths, etc. Almost all of `Path`'s methods perform only string transformations. The file system is not consulted to check whether paths are valid.

Process

`Process` is 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`. An example of listening for `uncaughtException` follows:

```
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.');
```

Punycode

`Punycode` is a way to represent international domain names with the limited character set ('a'-'z', '0'-'9') supported by the domain name system.

Query String

Handles parsing or composing query string parameters (including escaping and unescaping strings.)

Readline

`Readline` allows the reading of a stream (such as `process.stdin`) on a line-by-line basis. Note that once you've invoked this module, your program will not terminate until you've closed the interface.

REPL

Stands for Read-Eval-Print-Loop. 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 which 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.

StringDecoder

`StringDecoder` decodes a buffer to a string. It is a simple interface to `buffer.toString()` but provides additional support for utf8.

Timers

Timers all for the setting and clearing of timeouts and intervals just like you would in a browser.

TLS (SSL)

This API provides functionality for making or serving requests over SSL. The `tls` module uses OpenSSL to provide Transport Layer Security and/or Secure Socket Layer: encrypted stream communication. TLS/SSL is a public/private key infrastructure. Each client and each server must have a private key. All servers and some clients need to have a certificate. Certificates are public keys signed by a Certificate Authority or self-signed. See also the HTTPS module.

TTY

This API provides functions for interacting with TTYs. This will functionality will probably only be useful to you if you're writing Node programs to be run on the console (e.g. DevOps or System Administration scripts), rather than accessed over the web via HTTP requests.

UDP/Datagram

Provides functions for handling UDP servers and messages.

URL

`URL` allows you to interact with URLs and do things like parsing, formatting or resolving an absolute URL from a relative URL with a base URL.

Utilities

Functions including logging, debugging, object inspection and outputting.

VM

Allows you to compile arbitrary JavaScript code and optionally execute it in a new sandboxed environment immediately, saved or run later.

ZLIB

Functionality which provides bindings to Gzip/Gunzip, Deflate/Inflate, and DeflateRaw/InflateRaw classes. Each class takes the same options, and is a readable/writable `Stream`.

Resources

- StrongLoop website
- StrongLoop technical blog
- Node project website: nodejs.org
- Node downloads
- Node documentation
- Node on GitHub
- Official npm website: npmjs.org
- npm documentation
- Node Google Group
- Node Linkedin Group
- Node Developers Linkedin Group

About StrongLoop

StrongLoop is the leading contributor to Node.js v0.12. Launched in 2013 and based in Silicon Valley, StrongLoop was founded by engineers who have been contributing to Node.js since 2011. The company is funded by Ignition Partners and Shasta Ventures, plus advised by Marten Mickos, CEO of Eucalyptus.

Node.js is used to create “front edge” APIs that mobile applications use to connect to backend data. For developers creating these APIs, StrongLoop offers an API framework and mobile services such as push and geolocation that can be leveraged via iOS, Android and HTML5 SDKs with a variety of connectors including Oracle.

StrongLoop also offers the leading DevOps tooling for clustering, monitoring and optimizing Node applications. StrongLoop runs on all major operating systems and eight clouds including Amazon, Heroku, Red Hat's OpenShift and Rackspace. For more information, visit http://strongloop.com or drop us a line: callback@strongloop.com.