**Node.js**

**Node.js Philosophy**

-> In Node.js, some of these principles come directly from its creator, Ryan Dahl.

-> Some of the principle are inherited from the JavaScript culture.

-> Are influenced by the Unix Philosophy

**History of Node.js**

Node.js is an open-source, cross-platform JavaScript run-time environment for executing JavaScript code server side. Historically JavaScript was used primarily for client-side scripting, in which script written in JavaScript are embedded in a web page's HTML, to be run client side by a JavaScript engine in the user's web browser. Node.js enables JavaScript to be used for server- side scripting and runs scripting server - side to produce dynamic web page content before the page is sent to the user's web browser. Consequently, Node.js has become one of the foundational elements of the "JavaScript Everywhere" Paradigm allowing web application development to unify around a single programming language, rather than rely on a different language for writing server side scripts.

Though .js is the conventional filename extension for JavaScript code, the name "Node.js" is not referring to a particular file in this content. It is the Just name for the product.

Node.js has an event-driven architecture capable of asynchronous IO operation. These design choices aim to optimize throughput and scalability in Web application with many input/output operation, as well as for real time web application (real time communication programs and browser games).

The Node.js distributed development project, governed by the Node.js foundation is facilitated by the Linux foundation.

Node.js was originally written by Ryan Dahl in 2009, about three years after the introduction of the first server side JavaScript environment. The initial release supported by only Linux and Mac OS. Its development and maintenance was led by Dahl and later sponsored by Joyent.

Unix Philosophy in 9 points

1. Small is beautiful

2. Make each program do one thing well

3. Build a prototype as soon as possible

4. Choose portability over efficiency

5. Store data in a flat text file

6. Use software leverage to your advantage

7. Use shell script to increase leverage and portability

8. Avoid captive user interfaces.

9. Make every program a filter.

So, in term of Node.js, here's I do express the Unix philosophy

1. Write modules that do one thing well. Write new modules rather than complicate an old one.

2. Write modules that encourage composition rather than extension

3. Write module that handle data Stream, because that is universal interface.

4. Write module that are agnostic about the source of their input or the destination of their output.

5. Write module that solve a problem you know, so you can learn about the ones you don't

6. Write modules that are small, Iterate quickly, Refactor ruthlessly, Rewrite bravely

7. Write modules quickly, to meet your needs, with just a few tests for compliance, Avoid extensive specifications. Add a test for each bug you fix.

8. Write modules for publication, even if you only use them privately. You will appreciate documentation in the future.

https://en.wikipedia.org/wiki/List\_of\_software\_development\_philosophies

It is based on Small Core/ Small Module / small Surface.

Other projects like Node.js

1. Vert.x---> Polygot programming

2. Akka ---> Scala and Java

3. Tornado ---> Phython

4. Libevent ---> C

5. EventMachine --->Ruby

Why Node.js

1. Non Blocking IO

2. V8 JavaScript Engine

3. Single Thread with Event loop

4. 40,025 Modules

5. Window, Linux, Mac

6. 1 Language for backend and frontend

7. Active community

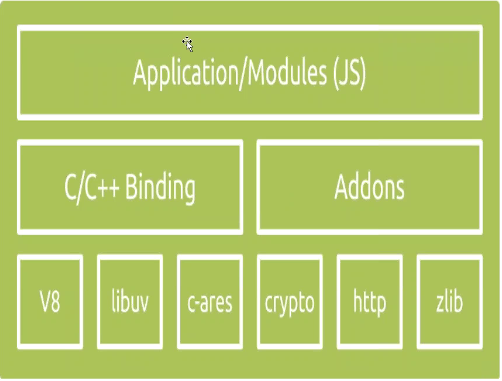
**ECMAScript**

ECMA stands for European Computer Manufacture's Association. ECMAScript is a standard for a scripting language.

JavaScript is basically ECMAScript as its core but builds upon it. Languages such as ActionSCript, JavaScript and JScript all use ECMAScript as its core. As a comparison, AS/JS/JScript are 3 different cars, but they all use the same engine. ECMAScript is standard. JavaScript and ActionScript are well - known implementation of ECMAScript Standard.

JavaScript was submitted to ECMA for standardization but due to trademark issues with name JavaScript the standard became called ECMAScript.

Node.js Architecture

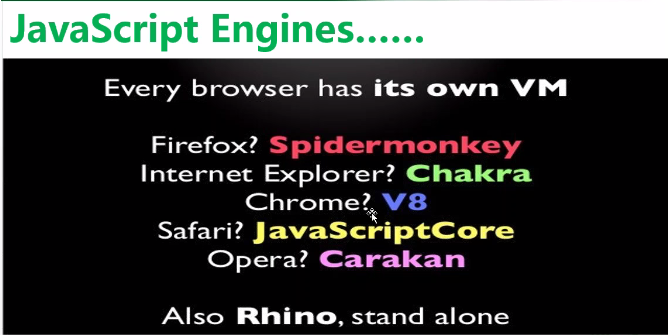


**Node.js Components and Dependencies**

V8: The high - performance JavaScript engine open sourced by Google and implemented in C++. This is the same engine that resides in you Chrome browser. V8 takes you write in JavaScript, compiles in into machine code (therefore blazing fast), and executes it. It has been use in many other projects, such as Couchbase, MongoDB and NodeJS that are used server side.

V8 compiles JavaScript directly to native machine code before executing it, instead of more traditional techniques such as interpreting byte codes compiling the whole program to machine code and executing it from a file system.

Different browser contains different types of engine to execute the JavaScript code like



**Libuv :** In Simple word it is a C library that provides asynchronous features or It is a library that allow you JavaScript code (via V8) to perform I/O, whether is it network, file etc. So from TCP level connectivity all the way to file/system Operations are actually performed by the Libuv Library. It maintain event loops, a thread pool, file system I/O, DNS functionality and network IO, among other critical functionalities.

Though it was built for NodeJS, libuv is used by other framework / platform as well as the Luvit framework which implements libuv in lua, the Julia language, PyUV interface for Python.

**Addons**

Node.js Addons are dynamically linked shared library, written in C or C++, that can be loaded into Node.js using the require() function and used just as if they were an ordinary Node.js module. They are primarily to provide an interface between JavaScript running in NOde.js and C/C++ libraries.

such as c-ares, crypto (OpenSSL), http-parser and zlib. These component or dependencies provide low-level interactions with servers to establish important functionalities such as networking, compressing, encrypting etc.

**c-ares:** C-ares is a library for asynchronous DNS request (including names resolves) C89 compatibility, MIT licensed, builds for and runs on POSIX, Window, Netware, Android and many more operating system.

Zlib : It is a module is used to provide compression and decompression (zip and unzip) functionalities. It is implemented using Gzip and deflate / inflate.

Binding :You probably have noticed by this time that Node.js is written in both JavaScript and C/C++. The reason that there are so many C/C++ code/libraries is simple. they are fast. however, how is it possible that the code you write in JavaScript end up communicating smoothly with code written in C/C++? Are not they three difference programming language. yes they are. And normally code written in different languages cannot be communicate with each other. No without bindings. Bindings, as the name implies, are glue codes that "bind" one language with another so that they can talk with each other. In this case (Node.js), bindings simply expose core Node.js internal libraries written in C/C++ (C-area, zlib, openSSL, http-parser etc) to JavaScript. One motivation behind writing bindings is code reuse. If a desired functionality is already implemented, why write the entire thing again, just because they are in different languages. why not just bridge them. Another motivation is performance, system programming language such as C/C++ are generally much faster than other high level programming language (ex Python, JavaScript, Ruby, Java). Therefore it might be wise to designate CPU-intensive operations to codes written in C/C++.

Another way to explain the Binding

Bindings basically are libraries that "bind" two different programming languages so that code written in one language can be used in code written in another library. With the presence of bindings, you don't have to write all the code again just because they are in different languages. another motivation for bindings in that you can benefit from the advantages of different programming languages. For example, C/C++ are much faster than JavaScript. It might be beneficial to write some code in C/C++ for performance purposes.

Now let's take a look at the picture you attached. V8 engine, according to Google official website, is written in C++. Libuv adds a layer of abstraction that provides asynchronous I/O operations, written in C. However, the core functionalities of Node.js such as networking, Database queries, file system I/O are provided in libraries (ore modules if you prefer) that are written in JavaScript. Plus, you code is written in JavaScript as well. Now in order for these piece of technologies written in different programming languages to communicate with each other, you have to bind them together, using bindings, these bindings are node.js binding.

**Node.JS Terminologies**

I/O: Shorthand for Input/Output. It basically denotes any computer operations handled primarily by the system's IO subsystem. IO bound operation usually involves interactions with disk/drives. Example include database access and file system operations. Some other related concepts include CPU-bound, memory-bound etc. A good way to determine whether an operation belongs to IO-bound, CPU-bound or other is to check by increasing which resource would that specific operations have a better performance. For example, if an operation would go noticeably faster if CPU power is increased, then it is CPU-bound.

Node.JS IO operation based upon on the Streams.

Stream are Unix pipes that let you easily read data from a source and pipe it to a destination. Simply put, a stream is nothing but an EventEmitter and implements some special methods, Depending on the methods implemented, a stream becomes Readable, Writable or Duplex (both readable and writable). Readable stream let you read data from a source while writable stream let you write data to a destination.

**Non-Blocking/Asynchronous :**Normally, when a request come in, the application would handle the request and halt all other operations until the request is processed. This immediately presents a problem. When a large number of requests come in at the same time, each request would have to wait until the previous requests are processed. In other words, the previous operation will block the one following it. To make it worse, if the previous requests have long response time (eg. Calculating the first 1000 prime numbers, or reading 3GB of data from database), all other requests would be halted/ blocked for a long time. To address this issue one can resort to multiprocessing and / or multithreading as every new request, all the request are handled on one single main thread, and that's pretty much all that it does. Handle requests. All IO operations contained in the request, such as file system access, database read/write operations, are sent to the worker threads maintained by libuv in the background. In other words IO operations in the requests are handled asynchronously, not on the main thread. This way the main thread is never blocked, as heavy lifting are shipped elsewhere. You and thus your application code ever only get to work with the one and only main thread. All the worker threads in libuv's thread pool are shielded from you, You never get to work directly with (or need to worry about) them. Node.js takes care of them for you. This architecture makes IO operation especially efficient. However, it is not without disadvantage. Operations include not only IO bounds ones, but CPU-bounds, memory-bounds, etc as well. Out of box, Node.js only provides asynchronous functions for IO tasks. there are ways to work around CPU-intensive operations.

**Event Driven :** Typically, in a most all modern systems, after the main application kicks off, processes are initiated by incoming requests. However, how things go from there differ, sometimes drastically, among different technologies. Typical implementation handle a request procedurally. A thread is spawn for a request. operations are done one after another. If an operation is slow, all following operation halts on that thread, when all operations complete successfully, a response to returned. However, in Node.js all operation are registered to Node.js as events, waiting to be triggered, either by the main application or requests.

Runtime (System) : Node.JS runtime is the entire codebase, both low-level and high-level that together supports the execution of a Node.js application.

At last

Node.JS it issued for IO than CPU intensive events which can be handled asynchronously with a single thread.

**Event Loop**

The Node.js event loop runs under a single thread, this means the application code you write is evaluated on a single thread. Node.js itself uses many threads undemeath through Libuv, but you never have to deal with those when writing Node.js code.

Every call that involves IO call require you to register a callback. This call also returns immediately, this allows you to do multiple IO operations in parallel without using threads in your application code. As soon as an IO operation is completes it is callback will be pushed on the even loop. It will be executed as soon as all the other callback that where pushed on the even loop before it are executed.

There are a few methods to do basic manipulation how call backs are added to the event loop. Usually you should not need these, but every now and then they can be useful.

