Published in Softup Technologies

Gerald Haxhi Follow
Mar 2, 2020 · 6 min read · Listen

Node.js Internals: An introduction to Node's runtime and architecture

Node.js is an open-source and cross-platform JavaScript runtime environment for executing Javascript outside the browser. It is backed by Google's V8 engine, which makes it extremely performant.

An asynchronous event-driven runtime

One of the most common statements that we encounter when we are introduced to Node is that it runs on a single thread. That said, everyone might wonder how is it possible that Node is one of the most popular tools for building fast and scalable APIs?

Technically, the fact that Node.js uses a single thread is not 100% correct. Node actually uses many threads, but the event loop (which we will mention later) and the user code run on one single thread. If we go through the documentation, we see that Node uses an event-driven, non-blocking I/O model that makes it lightweight and efficient.

What is an event-driven, non-blocking I/O model?

According to Node's guidelines, blocking methods execute synchronously and non-blocking methods execute asynchronously. Suppose that we have to write some code to read the content of a file and print it in the console. There are two ways to do it in Node: synchronously and asynchronously. Let's see the synchronous version first:

```
const fs = require('fs');

const data = fs.readFileSync('test.txt');
console.log(data);

console.log('Done');
```

Read file: Synchronous version

The above code does the following: Firstly, it requires the FS module. In the second line, the *readFileSync* method is invoked and the result is stored in the *data* variable. Node's main thread blocks on this line until all the content of the file is read. Then the content is logged on the console and finally, the "Done" statement will be printed. Now let's see the same code executed asynchronously:

```
const fs = require('fs');

fs.readFile('test.txt', (error, data) => {
  if (error) {
    throw error;
  }
  console.log(data);
});

console.log('Done');
```

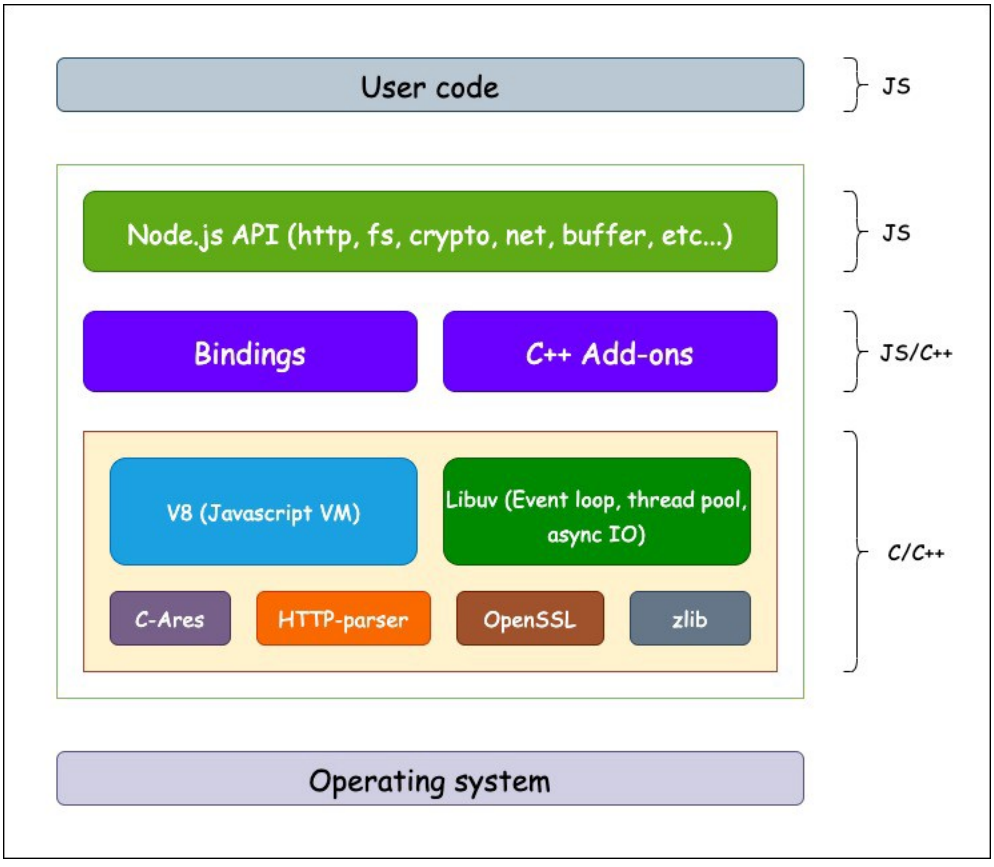
threads by default). As soon as the reading is done, the corresponding callback is pushed in a queue that is used by the event loop. On the next iteration of the event loop, during the callback execution phase, this callback will be pushed onto V8's call stack and will eventually be executed. All this work is done on the background and Node's main thread is responsible for only executing the callbacks. So, getting back to the example above, the "Done" statement will be printed first and the result from the file read will be logged after that. This is what is meant by "non-blocking I/O" and this is why in every Node.js guide you read, people suggest using async methods instead of their sync versions.

How does Node behave differently from other web-servers?

Compared to multithreaded servers, Node's event-driven runtime behaves much differently. In a multithreaded server, each connection spawns a new thread to handle the request and all the work that is done within that thread can be blocking, without affecting the other connections (i.e you can query the database and wait for the result, then do some other work). Since nowadays every CPU has many cores, this approach utilizes the processor power very well. However, there are also many challenges. In a multithreaded environment, each thread adds some overhead since it needs memory, which means that there is a limited number of threads that can be used. What happens if this limit is reached? The new connection will eventually time out. In addition to that, if an application is mostly I/O-bound, each thread would waste a considerable amount of time waiting for results from the network or disk. Node on the other hand handles everything in a single thread. Similar to what we explained above with the file operations, the event loop acts as a dispatcher that continuously listens for new events and delegates work to the kernel or other worker threads. It never blocks (unless told to do so). So, a server can accept a new client connection, then do some other work, then continue accepting new client connections all over again. A client connection doesn't need a new thread allocated, it just needs a socket handler that is managed by the kernel. This approach is fast, lightweight, and very scalable and it's the main reason why Node can handle so many simultaneous connections.

Node's runtime architecture

Node's runtime is designed as a set of layers where the user code sits on top and each layer uses the API provided by the layer below.



Node.js runtime architecture

2. **Node.js API**: A list of methods that Node offers, which can be used in the user code (e.g. *HTTP* modules for using HTTP methods, *crypto* module, *fs* for file system operations, *net* for network requests, etc…). For a full list of methods offered by Node, you can check the documentation here. Also, here you can find the source code implementation. Node's API is written in Javascript.
3. **Bindings and C++ add-ons**: When reading about Node, you see that V8 is written in C++, Libuv is written in C, and so on. Basically, all the modules are either written in C or C++, since these languages are so good and fast in dealing with low-level tasks and using OS API. But how is it possible that the Javascript code on the upper layers can use code that is written in other languages? This is what *bindings* do. They act as the glue between the two layers, so Node can smoothly use low-level code written in C or C++. So, what should we do if we want to add a C++ module on our own? We implement the module in C++ first and then write the binding code for that. This code that we write is called an add-on. More information can be found here.
4. **Node's dependencies**: This layer represents the low-level libraries that Node uses. The biggest dependencies are Google's V8 engine and Libuv. Other libraries include OpenSSL (for SSL, TLS, and other basic cryptographic functions), HTTP-parser (for parsing HTTP requests and responses), C-Ares (for asynchronous DNS requests), and Zlib (for fast compression and decompression).
5. **Operating system**: This is the lowest level that represents the OS API (system calls), which are used by the libraries mentioned above. Since OS-es are different, the libraries include implementations for both Windows and Unix variations, which make Node platform-independent.

A few words about V8 and Libuv

Libuv is a library written in C that is used to abstract non-blocking I/O operations. It offers the following features:

- The event loop
- Asynchronous TCP and UDP sockets
- Asynchronous DNS resolution
- Asynchronous file and file system operations
- Thread pool
- Child processes
- High-resolution clock
- Threading and synchronization primitives
- Polling
- Streaming
- Pipes

V8 is the library that provides Node.js with a Javascript engine. It is a JIT (Just-in-time) compiler, which means that it switches between compiling and running blocks of code continuously. The advantage of alternating between compiling and running is that it can collect information while running the code and based on the data received, it can speculate what will happen in the future. Those speculations are useful for compiling better code.

If you want to learn more about Node.js, I recommend the following links:

- Introduction to Node.js: https://nodejs.dev/introduction-to-nodejs

- Node.js dependencies: https://nodejs.org/en/docs/meta/topics/dependencies/
- Node.js guides: https://nodejs.org/en/docs/guides/
- Official Node.js repo on GitHub: https://github.com/nodejs/node