# Node.js The Big Picture

The following is a list of sections in this document. Using Microsoft Word, you can use these as hyperlinks to navigate to any particular section. But using Apache Open Office, these hyperlinks do not work; instead, they merely serve as a table of contents. You can navigate to the start of any section via bookmarks; type F5 to bring up the Navigator; then double-click Bookmark1 for 1st section header, Bookmark 2, for 2nd section header, etc.

|  |  |
| --- | --- |
| [Considering Node.js](#_Considering_Node.js) | [Where is Node.js Commonly Found?](#_Where_is_Node.js) |
| [What Makes up Node.js?](#_What_Makes_up) | [A Brief History / When Node May Not Be the Best Fit](#_A_Brief_History) |
| [Node’s Event Loop](#_Node’s_Event_Loop) | [Asynchronous Development](#_Asynchronous_Development) |
| [Node API’s and Event Emitters](#_Node_API’s_and) | [Streams](#_Streams) |
| [Modularizing Your Application](#_Modularizing_Your_Application) | [npm and Application Dependencies](#_npm_and_Application) |
| [Installing Node / Testing Applications](#_Installing_Node_/) | [Debugging / Conclusion](#_Debugging_/_Conclusion) |
|  |  |

## Considering Node.js

Node.js is used to build software that is

* driven by asynchronous events
* scalable
* in network applications
* commonly server-side javascript (but not always)

## Where is Node.js Commonly Found?

Node.js is commonly found in microservices and APIs. Unlike .NET (in which there is a server to deploy your code to), your Node.js code is the server. A simple example . . .

const http = require(‘http’);

const hostname = ‘127. 0.0.1’;

const port = 3000;

const server = http.createServer((req, res) => {

res.statusCode = 200;

res.setHeader(‘Content-Type’, ‘text/plain’);

res.end(‘Hello World\n);

});

Server.listen(port, hostname, () => {

Console.log(‘Server running at http://${hostname}:${port}/’);

});

A 2nd place where Node.js is found is in the Cloud – particularly with serverless functions. For example, it is used by AWS and Azure.

Other places where Node.js is found are in command-line applications and in desktop applications.

## What Makes up Node.js?

Part of the answer to the question “What Makes up Node.js?” is answered by examining the Chrome web browser. Among the many components of Chrome are the Blink Rendering Engine and the V8 Java Engine. Node.js gets its Java Script support from the V8 Java Engine, as well. Another component (an open-source product), libuv, provides Node’s asynchronous I/O and its event loop. Finally, Node.js includes some custom code. The implication of using the V8 Java Engine is that the Node.js user – as a front-end developer - will not be concerned with Java-Script incompatibilities or the choice of the web browser.

## A Brief History / When Node May Not Be the Best Fit

Ryan Dahl created Node.js in 2009. Joyent became Node.js’s sponsor in 2010. Also in 2010 the npm package manager was released. Joyent and Microsoft collaborated in 2011 in bringing Node.js to Windows. This included the introduction of libuv (mentioned in a previous section). In 2014 there emerged some frustration with Joyent’s governance, this resulted in a fork from Node called io.js. The parallel versions of Node.js and Io.js lasted only briefly, and in 2015 they merged. Later in 2015 Node Foundation was created, and it released their 1st version Node 4.0. From 2015 to 2020 new releases have occurred regularly every 6 months.

The following are some reasons why Node may not be a good fit for a developer.

* Node does not perform CPU-intensive tasks well. It is much better suited to I/O-intensive tasks.
* Node is not suited to the software developer who does not like JavaScript.

## Node’s Event Loop

On the server side Node.js handles requests coming from several clients. Node does not do this using process-per-client (multi-process) software, nor does Nod do this using thread-per-client (multi-threaded) software. Instead, Node uses a software model called Event Loop (single-threaded). This does not mean that multiple threads are missing from the behind-the-scene modules; instead, it means that the Node programmer is not concerned with multiple threads. One implication is that there is no shared memory. Another implication is that problems (if any) arising from concurrency – e.g. in a database shared by the clients – must be managed somehow outside of Node.js. Events recognized by the Node.js server include notification from the operating system of file-open completion, network activity such as an HTTP request, a timer which had been set previously. The client will typically register a call-back function with the Node server for some of these events, and the Node server will comply by waking up the client, and providing pertinent information.

## Asynchronous Development

The code in the Node.js client should execute quickly. If it spends too much time executing, the Event Loop in the Node server becomes blocked, and performance is degraded.

Consider an analogy where a waitress serves several customers in a restaurant; the waitress also interacts with the kitchen. The synchronous model – as follows – is inappropriate for Node.js.

function serveCustomer ( customer ) {

let order = customer . placeOrder ( menu )

let food = cook . prepareFood ( order )

let tip = customer . eatAndPay ( food )

return tip

}

If this synchronous-model code were running in a multi-threaded engine, the concurrency of multiple threads would use the wait time for each step efficiently. But for Node.js the appropriate code could be

function serveCustomer ( customer, done ) { // “done” is the name of our call-back function

customer . placeOrder ( menu, ( error, order ) => {

// (error, order) is an inline anonymous call-back function which is called when the

// customer is ready to place his order. The body of the function is as follows.

cook . prepareFood ( order, ( error, food ) => { // 2nd inline anonymous call-back fnc

customer . eatAndPay ( food, done )

}

}

}

The multiple level nesting can make Node.js hard to read. A node.js syntax improvement – as follows – provides an appropriate vehicle for writing the code.

const serveCustomer = async ( customer ) => {

let order = await customer . placeOrder ( menu )

let food = await cook . prepareFood ( order )

let tip = await customer . eatAndPay ( food )

return tip

}

## Node API’s and Event Emitters

There is a new concept called an EventEmitter. EventEmitter is a java script class; a couple of its methods are emitter.emit() and emitter.on(). In the following code emitter is an instance of EventEmitter, and ‘data’ is the name of an event.

emitter . emit ( ‘data’, ‘Hello World’ )

emitter . on ( ‘data’, ( msg ) => {

console . log ( msg )

} )

Imagine that the emitter.on statement is executed 1st. It tells Node to invoke console logging (as a callback function) whenever a ‘data’ event is occurs in the context of emitter.emit(). Subsequently when the emitter.emit statement executes, the text ‘Hello World’ is logged to the console.

The Pluralsight instructor shows how the code in the previous section could have been written using the EventEmitter class. customer and order are instances of EventEmitter. ‘decided’, ‘prepared’, and ‘leaving’ are names of events.

const serveCustomer = ( customer, done ) => {

customer . on ( ‘decided’, order => {

order . on ( ‘prepared’, food => customer . eatAndPay ( food ) )

cook . prepareFood ( order )

} )

customer . on ( ‘leaving’, tip => done ( null, tip ) )

customer . placeOrder ( menu )

}

The code preceding placeOrder() simply sets up callback functions. The placeOrder() statement gets the ball rolling.

This implementation is somewhat more obscure than the use of async/await, and the instructor does not recommend that it be used routinely. His purpose in introducing it is that it occurs in higher-order Node.js constructs, which the programmer will probably encounter, and it’s important to know that they exist.

## Streams

Streams refer to interactions with external data in a sequential-access manner. (Imagine reading a file of characters - starting from the beginning and continuing to the end using the file’s natural order.) This processing of data sequentially must be done in compliance with Nod.js’s asynchronous programming model.

There are 3 flavors of streams – readable streams, writable streams, and read/write streams (duplex, transform).

Events of readable streams include **readable**, **data**, **end**, and **error**.

Methods of readable streams include **read**, **pause**, **resume**, and **destroy**.

Properties of readable streams include **readable** and **readableLength**.

Events of writable streams include **drain**, **close**, **finish**, and **error**.

Methods of writable streams include **write**, **destroy**, and **end**.

Properties of writable streams include **writable** and **writableLength**.

Both readable and writable streams are equipped with buffers, which accommodate differences in processing speed between the “supply” end and the “consume” end. When the fixed number of buffers become filled, the faster end - “supply” or “consume” must pause to let the other end catch up. This is handled automatically by the stream.

A facility that is a companion to streams is “Piping”. The method .pipe() can be used to move the data from a readable stream to a writable stream. For example, one might read a file from disk and pipe that data to a web request.

With regard to read/write streams, one example is a transform stream that the API can write to using writable-stream methods; the transform stream, in turn, will Gzip the contents into compressed data, which would then be available as a readable stream.

The following is a simple example, that uses all 3 stream types. This example builds a simple HTTP server that – for any request – will read a file from disk, gzip compress it, and send that compressed data back as the HTTP response; the name of the file is lorem.txt.

const server = http.createServer ( ( req, res ) => {

res . setHeader ( ‘Content-Type’, ‘text/plain’ );

res . setHeader ( ‘Content-Encoding ‘, ‘gzip’ )

fs . creatReadStream ( path . join ( \_\_dirname, ‘lorem.txt’ ) )

. pipe ( zlib . createGzip() )

. pipe ( res )

} )

The Pluralsight instructor warns us that event emitters and streams are built on the callback approach found in Node.js APIs today. But using them with promises or async/await is not straightforward; it would be advisable to avoid combining streams with promises or async/await until Node.js makes improvements.

## Modularizing Your Application

Looking back at an earlier example of Node.js code (in [Asynchronous Development](#_Asynchronous_Development) ) it is good practice to subdivide this application into 3 separate files – waitress.js, cook.js, and customer.js.

cook.js

const ingredients = ‘stuff’

const prepareFood = ( order, done ) => {

// prepare the food

}

module . exports = { prepare Food }

Everything in cook.js is private, except the prepareFood function.

customer.js

const moment = require ( ‘moment’ )

class Customer {

// methods and properties

}

module . exports = Customer

Instead of exporting a function (as done by cook.js), customer.js exports the class Customer.

waitress.js refers to exports from cook.js and customer.js. These foreign references are expressed via the require() function.

waitress.js

const cook = require ( ‘./cook’ )

const Customer = require ( ‘./customer’ )

// cook . prepareFood()

// new Customer ()

These foreign reference allows us to define the **cook** variable and the **Customer** class (which we use in the last two (commented-out statements). Incidentally **cook** is not capitalized because it is a variable, and **Customer** is capitalized, because it is a class. These are not rules in Node.js; they are merely conventions. Notice, also, that the file names have a prefix that specifies their paths; “./” means “look for the file in the current folder”. r**equire(‘moment’)** is a foreign reference to a third-party module. The **moment** library becomes available with the help of **npm** (Node Package Manager), which will be discussed in the next section.

## npm and Application Dependencies

To begin with npm is a repository where o developer can download third-party modules to a Node project. Secondly npm is a command-line application. A few commands one might use in a Node project are as follows.

**$ npm init**

**npm init** bootstraps a project and creates a basic package.json file. package.json is found in almost every Node project and is the central location for the project’s metadata.

**$ npm install**

**npm install** is used to add dependencies to a project. Example: “$ npm install –save moment” adds the “moment” library; **save** is a qualifying parameter, which says “update the project’s package.json file.” 2nd Example: “$ npm install –save-dev mocha” adds the “mocha” package – for writing tests; the **save-dev** parameter causes this dependency to be recorded in the **devDevelopment** section of the project’s package.json file.

Recording dependencies into the project’s package.json file makes it unnecessary to check these settings into source control with our own code. Instead, when our project is checked out of source control (e.g. by a new developer), the recipient will also get package.json; simply invoking **npm install** will download all of the third-party modules. These modules will have dependencies of their own, which implies a large number of additional third-party modules.

## Installing Node / Testing Applications

The Node facility can be downloaded from the Node.js website; use the URL https://nodejs.org/en/. I tried using that URL on my Windows-7 computer – selecting the **Recommended For Most Users** option, and it reported an incompatibility with the operating system. It does install successfully, however, on my Windows 10 computer.

* Clicking the **Recommended For Most Users** option (version 14.17.0) caused the download executable to be downloaded to my **Downloads** folder.
* Double-click the download executable to download Node.js and npm.
* Open the Windows start menu to confirm that Node.js has been installed.
* To get additional confirmation, start the Visual Studio 2019 Developer Command Prompt, and type the following commands.

> node –v

The response was “v14.17.0”.

>npm –v

The response was “6.14.13”.

***Refer, also, to the Windows 7 computer, Documents\Current Pixel\Node\_js\Node\_js.docx.***

The Pluralsight instructor recommended ”Express”, which is a package (to be installed via **npm**). You can read about **Express** via https://expressjs.com/.

With regard to testing Node application, the Pluralsight instructor recommended **Mocha** (mochajs.org), **Chai** (www.chaijs.org), **Sinon** (sinonjs.org), and **Instanbul** (instanbul.js.org)

The Pluralsight instructor displays sample code that contains references to Mocha, Chai, and Sinon. He displays sample printouts of what Instanbul could provide.

## Debugging / Conclusion

To debug a Node application, the instructor displays Visual Studio Code – as a tool. An alternative that the Pluralsight instructor mentioned is “Console.log”.