

NODE.JS

Technical Seminar



APRIL 8, 2018

LORDS INSTITUTE OF ENGINEERING AND TECHNOLOGY
Himayat Sagar, Hyderabad.

NODE JS

Content

1. Introduction

1.1.Introduction to Node.js. 3

1.2.Fundamental of Node.js.

1.2.1. Google V8 Engine. 5

1.2.2. Synchronous and Asynchronous 6

1.2.3. Event Driven Programming 8

1.2.4. Node Package Manager (NPM) 9

2. Downloading and Installing Node.js

2.1. Installing on Windows 11

2.2. Installing on Linux 11

3. Using Node.js

3.1. Node.js REPL Shell 15

3.2. Launching a Node.js script 16

3.3. Exporting and Importing modules 17

4. File Handling utilities in Node.js

4.1. FS module 19

4.2. FS Read and FS Write 20

5. Creating a simple server 22

6. Frameworks of Node.js	
6.1. Express.js	25
6.2. Scket.js	25
6.3. Hapi.js	26
7. When to Use Node.js	
7.1. Advantages	27
7.2. Disadvantages	28
8. Conclusion	29

1. Introduction

1.1. Introduction to Node.js

Node.js is an open source, cross-platform runtime environment for developing server-side and networking applications. Node.js applications are written in JavaScript and can be run within the Node.js runtime on OS X, Microsoft Windows, and Linux.

Node.js is a platform built on Chrome's JavaScript runtime for easily building fast and scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.

Node.js also provides a rich library of various JavaScript modules which simplifies the development of web applications using Node.js to a great extent.

Node.js = Runtime Environment + JavaScript Library

Features of Node.js

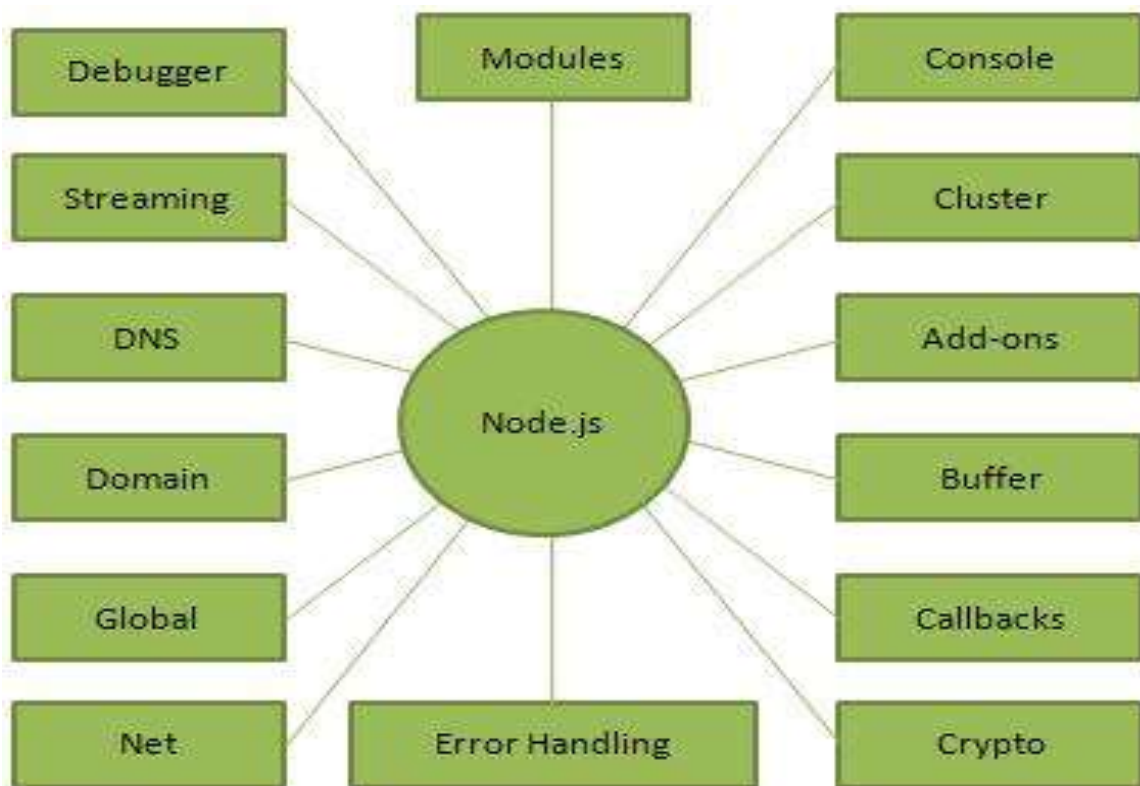
Following are some of the important features that make Node.js the first choice of software architects.

- **Asynchronous and Event Driven** – All APIs of Node.js library are asynchronous, that is, non-blocking. It essentially means a Node.js based server never waits for an API to return data. The server moves to the next API after calling it and a notification mechanism of Events of Node.js helps the server to get a response from the previous API call.
- **Very Fast** – Being built on Google Chrome's V8 JavaScript Engine, Node.js library is very fast in code execution.
- **Single Threaded but Highly Scalable** – Node.js uses a single threaded model with event looping. Event mechanism helps the server to respond in a non-blocking way and makes the server highly scalable as opposed to traditional servers which create limited threads to handle requests. Node.js uses a single threaded program and the same program can

provide service to a much larger number of requests than traditional servers like Apache HTTP Server.

- **No Buffering** – Node.js applications never buffer any data. These applications simply output the data in chunks.
- **License** – Node.js is released under the [MIT license](#)
- The following diagram depicts some important parts of Node.js which we will discuss in detail in the subsequent chapters.

Node.js is a highly efficient and scalable non-blocking I/O platform that was build on top of Google Chrome V8 engine and its ECMAScript. This means that



most front-end JavaScript (another implementation of ECMAScript) objects, functions and methods are available in Node.js.

1.2. Fundamentals of Node.js

1.2.1. Google V8 Engine

Chrome V8, or simply **V8**, is an open-source JavaScript engine developed by The Chromium Project for Google Chrome and Chromium web browsers. The project's creator is Lars Bak. The first version of the V8 engine was released at the same time as the first version of Chrome: September 2, 2008. It has also been used in Couchbase, MongoDB and Node.js that are used server-side. V8 compiles JavaScript directly to native machine code before executing it, instead of more traditional techniques such as interpreting bytecode or compiling the whole program to machine code and executing it from a filesystem. The compiled code is additionally optimized (and re-optimized) dynamically at runtime, based on heuristics of the code's execution profile. Optimization techniques used include inlining, elision of expensive runtime properties, and inline caching. The garbage collector is a generational incremental collector. V8 can compile to x86, ARM or MIPS instruction set architectures in both their 32- and 64-bit editions; as well, it has been ported to PowerPC and IBM s390 for use in servers.

V8 is intended to be used both in a browser and as a standalone high-performance engine that can be integrated into independent projects. V8 is used in the following software:

- Google Chrome, Chromium, Opera and Vivaldi web browsers
- Couchbase database
- Node.js runtime environment.
- Electron software framework, the underlying component for Atom and Visual Studio Code text editors

1.2.2. Synchronous and Asynchronous

The difference between synchronous and asynchronous execution may seem a bit confusing at first. Program execution in most high-level languages is usually very straightforward. Your program starts at the first line of source code and each line of code executed sequentially thereafter. Easy enough.

Synchronous program execution is somewhat similar to the above. Your program is executed line by line, one line at a time. Each time a function is called, program execution waits until that function returns before continuing to the next line of code.

This method of execution can have undesirable ramifications. Suppose a function is called to start a time-consuming process. What if you want to stop the lengthy process? With synchronous execution, your program is “stuck,” waiting for the process to end, with no way out.

Asynchronous execution avoids this bottleneck. You are essentially saying, “I know this function call is going to take a great deal of time, but my program doesn’t want to wait around while it executes.”

```
// Exposure time in seconds
TakePicture (long lExposureTime)
Begin

Do
stuff to take a picture...

While (notCancelledByUser)

if (notCancelledByUser == TRUE)
return NoError;
else
return Error;
End

Main
Begin
```

```
TakePicture (120);  
Print ("TakePicture () function returns!")  
End
```

In the above pseudo-code, using synchronous execution, you'll have to wait two minutes for the call to `TakePicture ()` return and display the "TakePicture () function returns!" message. There is no way to cancel the picture.

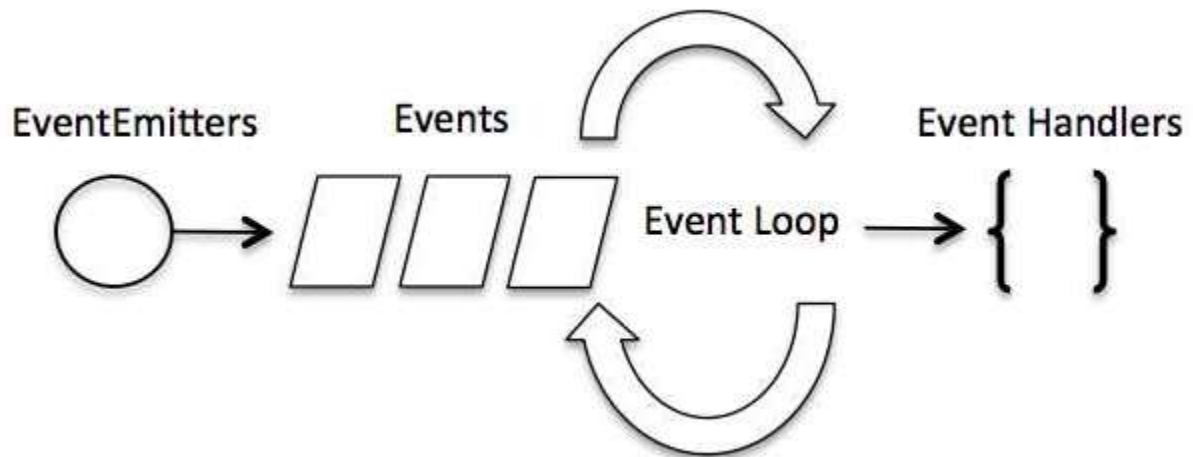
Using asynchronous execution, the `TakePicture ()` function returns immediately and shows the message. Although the two-minute process is not complete, your program can continue to execute. In this manner, your program could set the *notCancelledByUser* variable to `FALSE` to cancel the picture. It can also poll or ask the `TakePicture ()` function when the exposure is completed, or if an error occurred during the process.

Node.js achieves its "concurrency" by relying upon its runtime construct called the event loop. The event loop is at the core of every Node.js application because it is through it that Node.js can achieve its high input and output (IO) volumes. The way it works at a high level is that it heavily relies on the asynchronous programming concept to enable the application to be non-blocking. That is, request/statement 2 does not wait for request 1 to finish and request/statement 3 does not wait for request/statement 2 to finish before it can proceed. And the same applies to every other task that needs to be performed by the runtime.

1.2.3. Event Driven Programming

Node.js uses events heavily and it is also one of the reasons why Node.js is pretty fast compared to other similar technologies. As soon as Node starts its server, it simply initiates its variables, declares functions and then simply waits for the event to occur.

In an event-driven application, there is generally a main loop that listens for events, and then triggers a callback function when one of those events is detected.



Although events look quite similar to callbacks, the difference lies in the fact that callback functions are called when an asynchronous function returns its result, whereas event handling works on the observer pattern. The functions that listen to events act as **Observers**. Whenever an event gets fired, its listener function starts executing. Node.js has multiple in-built events available through events module and Event Emitter class which are used to bind events and event-listeners.

1.2.4. Node Package Manager(NPM)

npm, short for Node Package Manager, is two things: first and foremost, it is an online repository for the publishing of open-source Node.js projects; second, it is a command-line utility for interacting with said repository that aids in package installation, version management, and dependency management. A plethora of node.js libraries and applications are published on npm, and many more are added every day. These applications can be searched for on <http://search.npmjs.org/>. Once you have a package you want to install, it can be installed with a single command-line command.

Let's say you're hard at work one day, developing the Next Great Application. You come across a problem, and you decide that it's time to use that cool library you keep hearing about - let's use Caolan McMahon's `async` as an example. Thankfully, npm is very simple to use: you only have to run `npm install async`, and the specified module will be installed in the current directory under `./node_modules/`. Once installed to your `node_modules` folder, you'll be able to use `require()` on them just like they were built-ins.

Let's look at an example of a global install - let's say `coffee-script`. The npm command is simple: `npm install coffee-script -g`. This will typically install the program and put a symlink to it in `/usr/local/bin/`. This will then allow you to run the program from the console just like any other CLI tool. In this case, running `coffee` will now allow you to use the `coffee-script` REPL.

Another important use for npm is dependency management. When you have a node project with a `package.json` file, you can run `npm install` from the project root and npm will install all the dependencies listed in the `package.json`. This makes installing a Node project from a git repo much easier! For example, `vows`, one of Node's testing frameworks, can be installed from git, and its single dependency, `eyes`, can be automatically handled:

npm install (in package directory, no arguments):

Install the dependencies in the local `node_modules` folder.

In global mode (i.e., with **-g** or **--global** appended to the command), it installs the current package context (i.e., the current working directory) as a global package.

By default, **npm install** will install all modules listed as dependencies in **package.json**.

With the **--production** flag (or when the **NODE_ENV** environment variable is set to **production**), npm will not install modules listed in **devDependencies**.

npm install <folder>:

Install the package in the directory as a symlink in the current project. Its dependencies will be installed before it's linked. If **<folder>** sits inside the root of your project, its dependencies may be hoisted to the top-level **node_modules** as they would for other types of dependencies.

2. Downloading and Installing Node.js

2.1. Installing on windows

Here's the abbreviated guide, highlighting the major steps:

1. Open the official page for [Node.js downloads](#) and download Node.js for Windows by clicking the "Windows Installer" option
2. Run the downloaded Node.js `.msi` Installer - including accepting the license, selecting the destination, and authenticating for the install.
 - This requires Administrator privileges, and you may need to authenticate
3. To ensure Node.js has been installed, run `node -v` in your terminal - you should get something like `v6.9.5`
4. Update your version of npm with `npm install npm --global`
 - This requires Administrator privileges, and you may need to authenticate
5. Congratulations - you've now got Node.js installed and are ready to start building!

2.2. Installing on Linux

1. Install Node.js using your distribution's package manager (Stable, but outdated versions)

Node.js is available in the default repositories of most Linux distributions. It might not be latest version, but stable. If you want to have a stable Node.js on your Linux, you better install it using your distribution's package manager as shown below.

On Arch Linux and its derivatives like Antergos, Manajaro Linux, run the following command to install it:

```
sudo pacman -S node.js npm
```

On Debian, Ubuntu, Linux Mint:

```
sudo apt-get install node.js npm
```

On RHEL, CentOS, you need to enable EPEL repository first.

```
sudo yum install epel-release
```

And, then install Node.js using command:

```
sudo yum install node.js npm
```

On Fedora:

```
sudo dnf install node.js npm
```

Note: Since the packages from the default repositories are outdated, you will get the following error when you try to install any Node.js modules using npm.

```
/usr/bin/env: 'node': No such file or directory
```

To solve this error, you need to create symlink as shown below.

```
sudo ln -s /usr/bin/node.js /usr/bin/node
```

2. Install Node.js from NodeSource

Like I already said, node.js is available in the default repositories, but it might be bit outdated. To install the most recent version, install the latest version from NodeSource as shown below.

On Debian, Ubuntu distributions:

Add Node.js 6.x repository:

```
curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -
```

For Node.js 7.x:

```
curl -sL https://deb.nodesource.com/setup_7.x | sudo -E bash -
```

Then, install Node.js using command:

```
sudo apt-get install node.js npm
```

On RHEL, CentOS:

Run the following command as root user to add Node.js v6 LTS repository:

```
curl --silent --location https://rpm.nodesource.com/setup_6.x | bash -
```

To add Node.js v7 repository:

```
curl --silent --location https://rpm.nodesource.com/setup_7.x | bash -
```

To add Node.js 0.10 repository:

```
curl --silent --location https://rpm.nodesource.com/setup | bash -
```

Then install, as root:

```
yum install node.js npm
```

For Fedora, follow the same instructions above.

Install build tools (Optional)

To compile and install native addons from npm repository, you may also need to install build tools.

To install build tools on Debian, Ubuntu distributions, run the following command:

```
sudo apt-get install -y build-essential
```

On RHEL based systems:

```
sudo yum groupinstall 'Development Tools'
```

On Fedora:

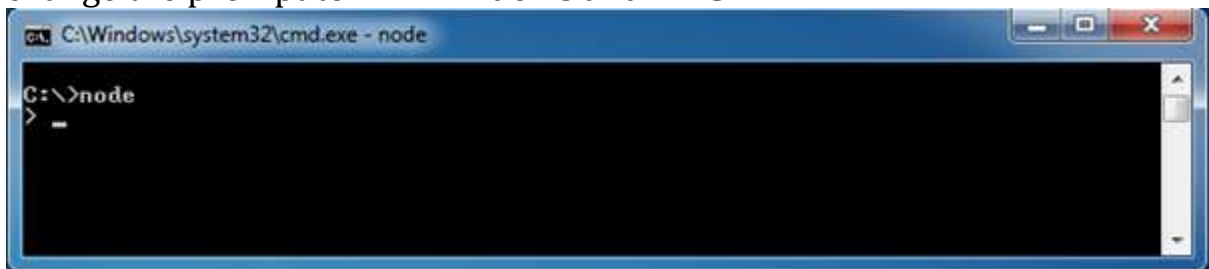
```
sudo dnf groupinstall 'Development Tools'
```

3. Using Node.js

3.1. Node.js REPL Shell

Node.js comes with virtual environment called REPL (aka Node shell). REPL stands for Read-Eval-Print-Loop. It is a quick and easy way to test simple Node.js/JavaScript code.

To launch the REPL (Node shell), open command prompt (in Windows) or terminal (in Mac or UNIX/Linux) and type *node* as shown below. It will change the prompt to `>` in Windows and MAC.



Launch Node.js REPL

You can now test pretty much any Node.js/JavaScript expression in REPL. For example, if you write `"10 + 20"` then it will display result 30 immediately in new line.

```
> 10 + 20
```

```
30
```

The `+` operator also concatenates strings as in browser's JavaScript.

```
> "Hello" + "World"
```

```
Hello World
```

You can also define variables and perform some operation on them.

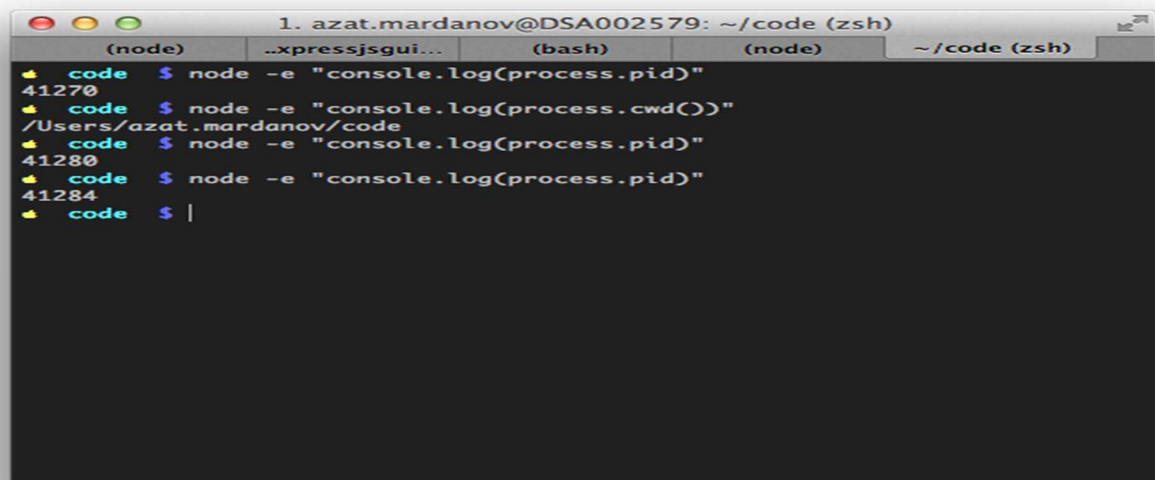

```
> var x = 10, y = 20;  
> x + y  
30
```

If you need to write multi line JavaScript expression or function then just press **Enter** whenever you want to write something in the next line as a continuation of your code. The REPL terminal will display three dots (...), it means you can continue on next line. Write. break to get out of continuity mode.

3.2. Launching a Node.js Script

To start a Node.js script from a file, simply run `$ node filename`, e.g., `$ node program.js`. If all we need is a quick set of statements, there's a `-e` option that allow to run inline JavaScript/Node.js, e.g., `$ node -e "console.log (new Date ());"`.

Each Node.js script that is running is a process in its essence. For example, `ps aux | grep 'node'` will output all Node.js programs running on a machine. Conveniently, developers can access useful process information in code with process object, e.g., `node -e "console.log(process.pid)"`:



```
1. azat.mardanov@DSA002579: ~/code (zsh)  
(node) ..xpressjsgui... (bash) (node) ~/code (zsh)  
★ code $ node -e "console.log(process.pid)"  
41270  
★ code $ node -e "console.log(process.cwd())"  
/Users/azat.mardanov/code  
★ code $ node -e "console.log(process.pid)"  
41280  
★ code $ node -e "console.log(process.pid)"  
41284  
★ code $ |
```

3.3. Exporting and Importing modules

Another *bad part* in browser JavaScript is that there's no way to include modules. Scripts are supposed to be linked together using a different language (HTML) with a lacking dependency management. CommonJS and RequireJS solve this problem with AJAX-y approach. Node.js borrowed many things from the CommonJS concept.

To export an object in Node.js, use `exports.name = object;`, e.g.,

```
var messages = {  
  
  find: function(req, res, next) {  
  
    ...  
  
  },  
  
  add: function(req, res, next) {  
  
    ...  
  
  },  
  
  format: 'title | date | author'  
  
}  
  
exports.messages = messages;
```

While in the file where we import aforementioned script (assuming the path and the file name is `route/messages.js`):

```
var messages = require('./routes/messages.js');
```

However, sometime it's more fitting to invoke a constructor, e.g., when we attach properties to Express.js app (more on Express.js in [Express.js Fundamentals: An Essential Overview of Express.js](#)). In this case module.exports is needed:

```
module.exports = function(app) {
```

```
app.set('port', process.env.PORT || 3000);
```

```
app.set('views', __dirname + '/views');
```

```
app.set('view engine', 'jade');
```

```
return app;
```

```
}
```

In the file that includes the example module above:

```
...
```

```
var app = express();
```

```
var config = require('./config/index.js');
```

```
app = config(app);
```

```
...
```

The more succinct code: `var = express(); require('./config/index.js')(app);`. The most common mistake when including modules is a wrong path to the file. For core Node.js modules, just use the name without any path, e.g., `require('name')`. Same goes for modules in `node_modules` folder. More on that later in the NPM section.

For all other files, use `.` with or without a file extension, e.g.,

```
var keys = require('./keys.js');
```

```
messages = require('./routes/messages.js');
```

In addition, for the latter category it's possible to use a longer looking statements with `__dirname` and `path.join()`, e.g., `require(path.join(__dirname, 'routes', 'messages'));` If `require()` points to a folder, Node.js will attempt to read `index.js` file in that folder.

4. File Handling Utilities in Node.js

4.1. FS module

The `fs` module provides an API for interacting with the file system in a manner closely modeled around standard POSIX functions.

To use this module:

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

All file system operations have synchronous and asynchronous forms. The asynchronous form always takes a completion callback as its last argument. The arguments passed to the completion callback depend on the method, but the first argument is always reserved for an exception. If the operation was completed successfully, then the first argument will be null or undefined.

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

fs.unlink('/tmp/hello', (err) => {
  if (err) throw err;
  console.log('successfully deleted /tmp/hello');
});
```

Exceptions that occur using synchronous operations are thrown immediately and may be handled using `try/catch`, or may be allowed to bubble up.

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

try {
  fs.unlinkSync('/tmp/hello');
  console.log('successfully deleted /tmp/hello');
} catch (err) {
  // handle the error
}
```

Note that there is no guaranteed ordering when using asynchronous methods. So the following is prone to error because the `fs.stat()` operation may complete before the `fs.rename()` operation.

```
fs.rename('/tmp/hello', '/tmp/world', (err) => {
  if (err) throw err;
  console.log('renamed complete');
});
fs.stat('/tmp/world', (err, stats) => {
  if (err) throw err;
  console.log(`stats: ${JSON.stringify(stats)}`);
});
```

To correctly order the operations, move the `fs.stat()` call into the callback of the `fs.rename()` operation:

```
fs.rename('/tmp/hello', '/tmp/world', (err) => {
  if (err) throw err;
  fs.stat('/tmp/world', (err, stats) => {
    if (err) throw err;
    console.log(`stats: ${JSON.stringify(stats)}`);
  });
});
```

In busy processes, the programmer is *strongly encouraged* to use the asynchronous versions of these calls. The synchronous versions will block the entire process until they complete — halting all connections. While it is not recommended, most fs functions allow the callback argument to be omitted, in which case a default callback is used that rethrows errors. To get a trace to the original call site, set the `NODE_DEBUG` environment variable:

4.2. FS Read and FS Write

fs.readFile()

Asynchronously reads the entire contents of a file. Example:

```
fs.readFile('/etc/passwd', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

The callback is passed two arguments (`err, data`), where `data` is the contents of the file.

If no encoding is specified, then the raw buffer is returned.

If `options` is a string, then it specifies the encoding. Example:

```
fs.readFile('/etc/passwd', 'utf8', callback);
```

When the path is a directory, the behavior of `fs.readFile()` and `fs.readFileSync()` is platform-specific. On macOS, Linux, and Windows, an error will be returned. On FreeBSD, a representation of the directory's contents will be returned.

```
// macOS, Linux, and Windows
fs.readFile('<directory>', (err, data) => {
  // => [Error: EISDIR: illegal operation on a directory, read <directory>]
});

// FreeBSD
fs.readFile('<directory>', (err, data) => {
  // => null, <data>
});
```

Any specified file descriptor has to support reading.

If a file descriptor is specified as the `path`, it will not be closed automatically.

The `fs.readFile()` function reads the entire file in a single threadpool request. To minimize threadpool task length variation, prefer the partitioned APIs `fs.read()` and `fs.createReadStream()` when reading files as part of fulfilling a client request.

fs.writeFile()

Asynchronously writes data to a file, replacing the file if it already exists. `data` can be a string or a buffer.

The `encoding` option is ignored if `data` is a buffer. It defaults to 'utf8'.

Example:

```
fs.writeFile('message.txt', 'Hello Node.js', (err) => {
  if (err) throw err;
});
```

```
console.log('The file has been saved!');  
});
```

If `options` is a string, then it specifies the encoding. Example:

```
fs.writeFile('message.txt', 'Hello Node.js', 'utf8', callback);
```

Any specified file descriptor has to support writing.

Note that it is unsafe to use `fs.writeFile` multiple times on the same file without waiting for the callback. For this scenario, `fs.createWriteStream` is strongly recommended.

If a file descriptor is specified as the `file`, it will not be closed automatically.

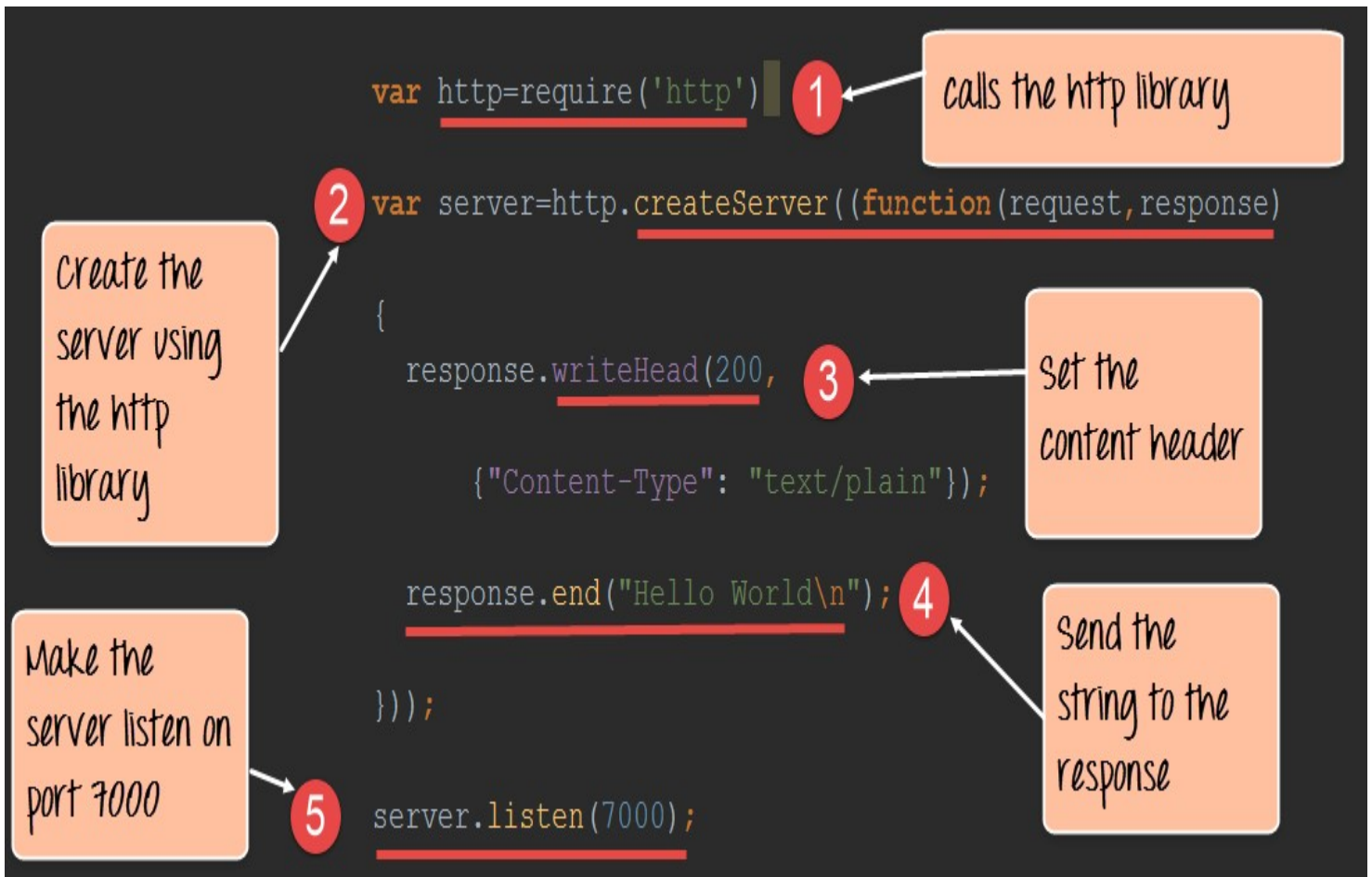
5. Creating a simple server

The Node.js framework is mostly used to create server based applications. The framework can easily be used to create web servers which can serve content to users.

There are a variety of modules such as the "http" and "request" module, which helps in processing server related requests in the web server space. We will have a look at how we can create a basic web server application using Node js.

Let's look at an example of how to create and run our first Node js application.

Our application is going to create a simple server module which will listen on port no 7000. If a request is made through the browser on this port no, then server application will send a 'Hello' World' response to the client.



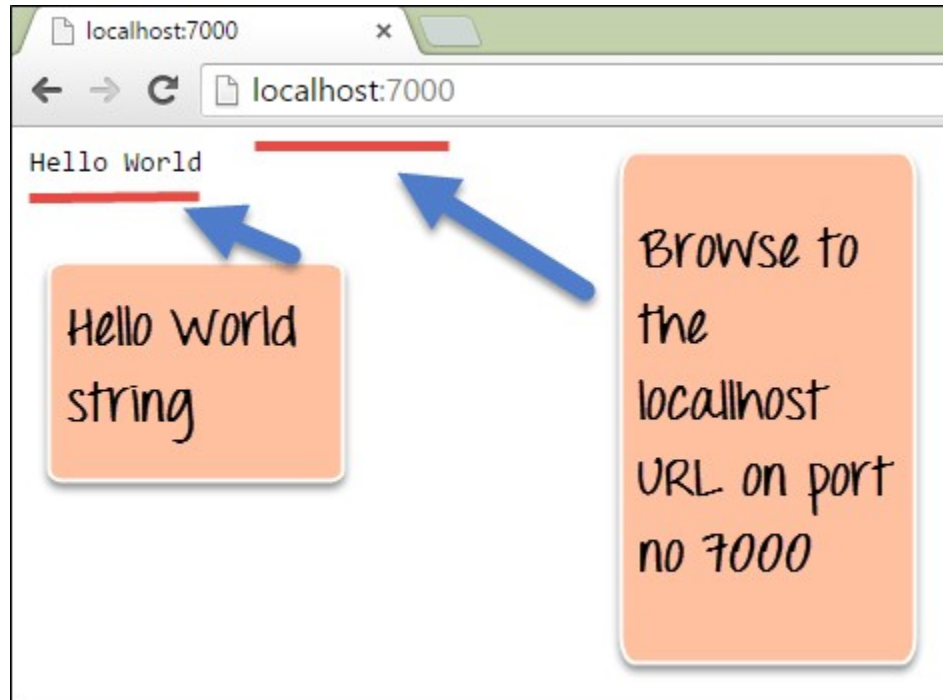
Code Explanation:

1. The basic functionality of the `require` function is that it reads a JavaScript file, executes the file, and then proceeds to return the exports object. So in our case, since we want to use the functionality of the `http` module, we use the `require` function to get the required functions from the `http` module so that it can be used in our application.
2. In this line of code, we are creating a server application which is based on a simple function. This function is called whenever a request is made to our server application.
3. When a request is received, we are saying to send a response with a header type of '200.' This number is the normal response which is sent in an http header when a successful response is sent to the client.
4. In the response itself, we are sending the string 'Hello World.'

5. We are then using the `server.listen` function to make our server application listen to client requests on port no 7000. You can specify any available port over here.

If the command is executed successfully, the following Output will be shown when you run your code in the browser.

Output:



From the output,

- You can clearly see that if we browse to the URL of localhost on port 7000, you will see the string 'Hello World' displayed in the page.
- Because in our code we have mentioned specifically for the server to listen on port no 7000, we are able to view the output when browsing to this url.

Frameworks of Node.js

6.1. Express.js

Express.js, or simply **Express**, is a web application framework for Node.js, released as free and open-source software under the MIT License. It is designed for building web applications and APIs. It has been called the de facto standard server framework for Node.js.

The original author, TJ Holowaychuk, described it as a Sinatra-inspired server, meaning that it is relatively minimal with many features available as plugins. Express is the backend part of the MEAN stack, together with MongoDB database and AngularJS frontend framework.

Express is a framework of Node.js that allows you to use several very useful and powerful features without having to reinvent the wheel, helps you organize your application's routing and use any templating solution with minimal effort. It also allows for much better organization of your code. In my personal opinion, Express.js is the best Node.js framework but Meteor.js, Derby.js, Sails.js and Flatiron.js are some of the best alternatives. Especially Meteor which is more of a full stack framework is superb and can do quite impressive things with minimal effort.

6.2. Socket.io

Socket.IO is a JavaScript library for Realtime web applications. It enables Realtime, bi-directional communication between web clients and servers. It has two parts: a client-side library that runs in the browser, and a side library for Node.js. Both components have a nearly identical API. Like Node.js, it is event-driven.

Socket.IO primarily uses the WebSocket protocol with polling as a fallback option,^[3] while providing the same interface. Although it can be used as simply a wrapper for WebSocket, it provides many more features, including broadcasting to multiple sockets, storing data associated with each client, and asynchronous I/O.

It can be installed with the npm tool

Libraries like Socket.IO are needed when we need real time in our app. Let me explain this in little more detail. Let's assume that you are developing a game, which is multiplayer and 2 or more users can play that simultaneously. Then in that case you won't be making HTTP or HTTPS calls because of many

reasons and one of them is that their packet size is large and other is that these calls are very slow. In such scenarios we use libraries like sockets to send and receive data to and from server. sockets are really fast and are capable of sending only those data packets which are needed. using HTTP programming you can never create any multiplayer game or any app which will be interacting with server on Realtime basis.

6.3. Hapi.js

hapi.js (also known as hapi) is an open-source framework for web applications. The most common use of hapi is to build web services such as JSON API. You can build application programming interface (API) servers, websites, and HTTP proxy applications with hapi.js.

hapi was created by the mobile team at Walmart Labs—led by Eran Hammer, who created OAuth—to handle their traffic for events like Black Friday, one of the busiest days for online shopping in the U.S. calendar.

The original versions of hapi used the Express framework. Walmart found that there were limitations with Express that made the framework unsuitable for their particular requirements. Express was missing some key features, so Walmart eventually had hapi evolve to its own stand-alone framework.

I started using Express to build my web apps. I was working on small projects, and Express is well-suited for small projects. But as I began to work on larger, more complex projects, I kept running into problems with the framework. It is tough to do scaling in Express. There's no single recommended way of organizing things in Express, which can be a trap for beginners and experienced developers alike, and can result in unmaintainable projects.

Walmart has a lot of users, and their systems are complicated. You could do enterprise-scale apps successfully in hapi. The transition to using hapi for me was quick and painless. I liked the simple configuration-driven APIs and the powerful plugin system of hapi.js.

hapi is important for developers to understand because it's designed to let them focus their time on critical project tasks instead of building infrastructure. We're going to take a deep dive into why hapi is a good thing for you to know about. Then I'm going to walk you through exactly how to use hapi.

7. When to use Node.js

7.1. Advantages

Good use cases

Node.js is a good choice for applications that have to process a high volume of short messages requiring low latency. Such systems are called real-time applications (RTAs), and they can be easily developed with Node.js. Thanks to its specifications, Node.js will be a good choice for the real-time collaborative drawing/editing-type apps, where you can watch the document being modified live by someone else (such as Trello, Dropbox Paper or Google Docs).

Fast and scalable environment.

Ruby on Rails might not be sufficient in terms of speed, if you have a ton of requests. Node.js will prove useful in situations when something faster and more scalable than Rails is needed. Node's ability to process many requests with low response times, as well as sharing things such as validation code between the client and server, make it a great fit for modern web applications that carry out lots of processing on the client's side. For these reasons, Node.js is a popular choice among "single-page application" sites, where all the rendering is done on the client's side, and the backend only provides a JSON API.

Node.js also comes in handy, when you want to process high volumes of IO-bound requests. It won't really be all that efficient if a lot of CPU processing is required to serve the request. Yet, if it's primarily just shuffling data around, then it will be quite efficient, because a single instance can serve a lot more requests with the same hardware, compared to the usual "thick" application server (like with rails).

7.2. Disadvantages

CPU-heavy jobs

The first thing that comes to my mind are heavy-computing apps. Node.js is based on an event-driven, non-blocking I/O model, and uses only a single CPU core. CPU-heavy operations will just block incoming requests, rendering the biggest advantage of Node.js useless. If you are considering building CPU-heavy software, try a different, more suitable, technology that will give you better results.

CRUD

Finally, using Node.js would be superfluous for simple HTML or CRUD applications in which you don't need a separate API, and all data comes directly from the server. Your application might be marginally more scalable, but don't expect more traffic coming to your app just because you used Node.js. In such cases, you should stick to such proven frameworks as Ruby.

Package quality

There are more things to consider before kicking off with Node.js. Many packages for Node.js applications are available in npm. The community is vibrant, the technology is becoming more mature, and npm is the largest available repository at the moment. Yet, packages vary in their quality. Sometimes, you can still encounter issues with packages supported only by individual users and not maintained properly; for instance, when connecting your Node app to an obscure or old database system.

8. Conclusion

Where Node.js really shines is in building fast, scalable network applications, as it's capable of handling a huge number of simultaneous connections with high throughput, which equates to high scalability. How it works under-the-hood is pretty interesting. Compared to traditional web-serving techniques where each connection (request) spawns a new thread, taking up system RAM and eventually maxing-out at the amount of RAM available, Node.js operates on a single-thread, using non-blocking I/O calls, allowing it to support tens of thousands of concurrent connections (held in the event loop).

Node.js is an amazing scalable framework for developing fast web-apps. There are many more interesting things you can do with Node.js, like enabling authentication using passport.js, or using the node-mailer to configure sending mails automatically.

This was a just a scratch of the amazing things that can be done with node.js if you want to learn more there are some easy to follow tutorials given below

<https://www.w3schools.com/nodejs/default.asp>

<http://www.tutorialsteacher.com/nodejs/nodejs-tutorials>