|  |  |
| --- | --- |
| Node JS | Node JS Document |

Table of Contents

[Introduction 4](#_Toc529211030)

[Features of Node.js 4](#_Toc529211031)

[1. Asynchronous and Event Driven 4](#_Toc529211032)

[2. Very Fast 4](#_Toc529211033)

[3. Single Threaded but Highly Scalable 4](#_Toc529211034)

[4. No Buffering 4](#_Toc529211035)

[5. License 4](#_Toc529211036)

[WHY to use Node JS 5](#_Toc529211037)

[Reasons to use NodeJS 5](#_Toc529211038)

[Reasons not to use NodeJS 6](#_Toc529211039)

[Setup Node.js Development Environment 6](#_Toc529211040)

[Install Node.js on Windows 7](#_Toc529211041)

[Download Node.JS Installer for Windows 7](#_Toc529211042)

[Node.js Installation 7](#_Toc529211043)

[Verify Installation 7](#_Toc529211044)

[Install Node.js on Mac/Linux 8](#_Toc529211045)

[Node Environment Setup 8](#_Toc529211046)

[Verify Node.js Installation on OS X 8](#_Toc529211047)

[IDE 8](#_Toc529211048)

[Demo “My First Node JS” Node app 8](#_Toc529211049)

[Steps to create Node Js program to print “ My First Node JS” 8](#_Toc529211050)

[What makes Node.js so great? 9](#_Toc529211051)

[What is Event Driven Programming? 10](#_Toc529211052)

[Callback Style Programming 11](#_Toc529211053)

[Promises 11](#_Toc529211054)

[What is Promises style programming? 11](#_Toc529211055)

[Event Emitters 11](#_Toc529211056)

[What are Event Emitters? 11](#_Toc529211057)

[Node Package Manager (NPM) 13](#_Toc529211058)

[What is NPM? 13](#_Toc529211059)

[How to use Modules in Node.js? 14](#_Toc529211060)

[Node.js Console - REPL 14](#_Toc529211061)

[Node.js Example in REPL 15](#_Toc529211062)

[REPL Commands 16](#_Toc529211063)

[Node.js Basics 17](#_Toc529211064)

[Primitive Types 17](#_Toc529211065)

[Loose Typing 17](#_Toc529211066)

[Object Literal 17](#_Toc529211067)

[Functions 18](#_Toc529211068)

[Buffer 18](#_Toc529211069)

[Process Object 18](#_Toc529211070)

[Defaults to local 19](#_Toc529211071)

[Access Global Scope 19](#_Toc529211072)

[Node.js Module 20](#_Toc529211073)

[Node.js Module Types 20](#_Toc529211074)

[Loading Core Modules 21](#_Toc529211075)

[Example: Load and Use Core http Module 21](#_Toc529211076)

[Node.js Web Server 22](#_Toc529211077)

[Create Node.js Web Server 22](#_Toc529211078)

[Handle HTTP Request 23](#_Toc529211079)

[Sending JSON Response 25](#_Toc529211080)

[Node.js File System 26](#_Toc529211081)

[Reading File 26](#_Toc529211082)

[Example: Reading File Synchronously 27](#_Toc529211083)

[Writing File 28](#_Toc529211084)

[Parameter Description: 28](#_Toc529211085)

[Open File 28](#_Toc529211086)

[Flags 29](#_Toc529211087)

[Delete File 30](#_Toc529211088)

[Example:File Open and Read 30](#_Toc529211089)

[Important method of fs module 30](#_Toc529211090)

[Express 31](#_Toc529211091)

[Install ExpressJS 31](#_Toc529211092)

[Database integration 34](#_Toc529211093)

[MongoDB 34](#_Toc529211094)

[Structuring Your API 36](#_Toc529211095)

[Routing 41](#_Toc529211096)

[Controllers 42](#_Toc529211097)

[Mongoose Models (MongoDB) 46](#_Toc529211098)

[Utilities 52](#_Toc529211099)

[How to Create a Node.js Cluster for Speeding Up Your Apps 55](#_Toc529211100)

[Node.js Cluster Module: what it is and how it works 56](#_Toc529211101)

[How a Cluster Module is Used in a Node.js App 57](#_Toc529211102)

[How to Develop a Highly Scalable Express Server 58](#_Toc529211103)

[Advanced Operations 60](#_Toc529211104)

[Communication Between Master and Workers 60](#_Toc529211105)

[Zero Down-time 61](#_Toc529211106)

[Conclusions 63](#_Toc529211107)

# Introduction

Node.js (Node) is an open source development platform for executing JavaScript code server-side. Node is useful for developing applications that require a persistent connection from the browser to the server and is often used for real-time applications such as chat, news feeds and web push notifications.

Node.js is intended to run on a dedicated HTTP server and to employ a single thread with one process at a time. Node.js applications are event-based and run asynchronously. Code built on the Node platform does not follow the traditional model of receive, process, send, wait, receive. Instead, Node processes incoming requests in a constant event stack and sends small requests one after the other without waiting for responses.

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

Node.js was written and introduced by Ryan Dahl in 2009. Visit Wikipedia to know the history of Node.js.

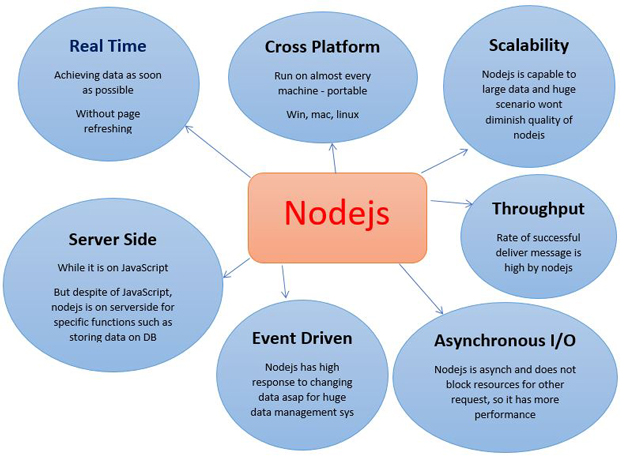
Node.js official web site: <https://nodejs.org>

Node.js on GitHub: <https://github.com/nodejs/node>

Node.js community conference <http://nodeconf.com>

# Features of Node.js

1. 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.
2. Very Fast − Being built on Google Chrome's V8 JavaScript Engine, Node.js library is very fast in code execution.
3. 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.
4. No Buffering − Node.js applications never buffer any data. These applications simply output the data in chunks.
5. License − Node.js is released under the MIT license.



# WHY to use Node JS

## Reasons to use NodeJS

1. It runs JavaScript, so you can use the **same language** on server and client, and even share some code between them (e.g. for form validation, or to render views at either end.)
2. The [single-threaded](http://www.haneycodes.net/to-node-js-or-not-to-node-js/) event-driven system is [**fast**](https://stackoverflow.com/questions/9290160/node-js-vs-net-performance) even when handling lots of requests at once, and also simple, compared to traditional multi-threaded [Java](http://www.infoworld.com/article/2883328/java/java-vs-nodejs-an-epic-battle-for-developer-mindshare.html) or ROR frameworks.
3. The ever-growing pool of [packages](http://npmjs.org/)**accessible through NPM**, including client and server-side libraries/modules, as well as command-line tools for web development. Most of these are conveniently hosted on GitHub, where sometimes you can report an issue and find it fixed within hours! It's nice to have everything under one roof, with standardized issue reporting and easy forking.
4. It has become the defector standard environment in which to run **JavaScript-related tools** and other **web-related tools**, including task runners, minifies, beautifiers, linters, preprocessors, bundlers, and analytics processors.
5. It seems quite suitable for prototyping, agile development, and **rapid product iteration**.

## Reasons **not** to use NodeJS

1. It runs JavaScript, which has no compile-time type checking. For large, complex **safety-critical** systems, or projects including collaboration between different organizations, a language which encourages **contractual interfaces** and provides **static type checking** may save you some debugging time (and *explosions*) overall. (Although the JVM is stuck with null, so please use Haskell for your nuclear reactors.)
2. Added to that, many of the packages in NPM are a little **raw**, and still under rapid development. Some libraries for older frameworks have undergone a decade of testing and bug fixing, and are very **stable** by now. [Npmjs.org has no mechanism to rate packages](https://pinboard.in/u:dandv/t:npm/t:against), which has led to a proliferation of packages doing more or less the same thing, out of which a large percentage are no longer maintained.
3. Nested callback hell. (Of course, there are [20 different solutions](https://github.com/nodejs/node/wiki/Modules#async-flow) to this...)
4. The ever-growing pool of packages can make one NodeJS project appear **radically different** from the next. There is a large diversity in implementations due to the huge number of options available (e.g. Express/[Sails.js](http://sailsjs.org/)/[Meteor](https://www.meteor.com/)/[Derby](http://derbyjs.com/)). This can sometimes make it harder for a new developer to jump in on a Node project. Contrast that with a **Rails** developer joining an existing project: he should be able to get familiar with the app quickly, because all Rails apps are encouraged to use a **similar structure**.

# Setup Node.js Development Environment

In this section, you will learn about the tools required and steps to setup development environment to develop a Node.js application.

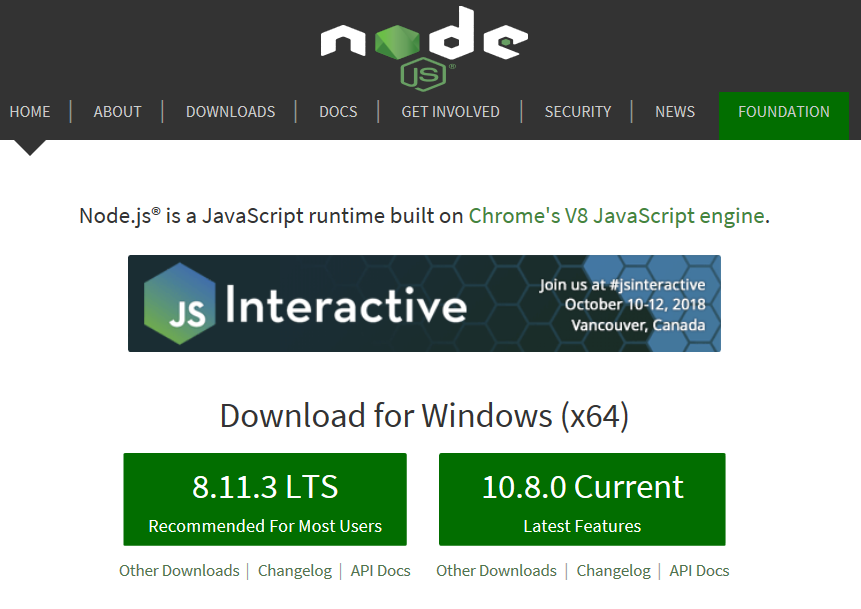
Node.js development environment can be setup in Windows, Mac, Linux and Solaris. The following tools/SDK are required for developing a Node.js application on any platform.

* Node.js
* Node Package Manager (NPM)
* IDE (Integrated Development Environment) or Text Editor

NPM (Node Package Manager) is included in Node.js installation since Node version 0.6.0., so there is no need to install it separately.

## Install Node.js on Windows

Visit Node.js official web site <https://nodejs.org>. It will automatically detect OS and display download link as per your Operating System.



Here, it has automatically detected the operating system.

### Download Node.JS Installer for Windows

Download node MSI for windows by clicking on 8.11.3 LTS or 10.8.0 Current button.

After you download the MSI, double-click on it to start the installation.

### Node.js Installation

Click Next to read and accept the License Agreement and then click Install. It will install Node.js quickly on your computer. Finally, click finish to complete the installation.

### Verify Installation

Once you install Node.js on your computer, you can verify it by opening the command prompt and typing node -v. If Node.js is installed successfully then it will display the version of the Node.js installed on your machine.

## Install Node.js on Mac/Linux

Visit Node.js official web site [https://nodejs.org/en/download](https://nodejs.org/en/download%20) page. Click on the appropriate installer for Mac (. pkg or .tar.gz) or Linux to download the Node.js installer.

### Node Environment Setup

Once downloaded, click on the installer to start the Node.js installation wizard. Click on Continue and follow the steps. After successful installation, it will display summary of installation about the location where it installed Node.js and NPM.

### Verify Node.js Installation on OS X

After installation, verify the Node.js installation using terminal window and enter the following command. It will display the version number of Node.js installed on your Mac.

$ node -v

Optionally, for Mac or Linux users, you can directly install Node.js from the command line using Homebrew package manager for Mac OS or Linux brew package manager for Linux Operating System. For Linux, you will need to install additional dependencies, viz. Ruby version 1.8.6 or higher and GCC version 4.2 or higher before installing node.

$ brew install node

## IDE

Node.js application uses JavaScript to develop an application. So, you can use any IDE or text editor tool that supports JavaScript syntax. However, an IDE that supports auto complete features for Node.js API is recommended e.g. Visual Studio, Sublime text, Eclipse etc.

# Demo “My First Node JS” Node app

One of the most common uses for node is running your server.

## Steps to create Node Js program to print “ My First Node JS”

**Step 1**: Go to terminal & create a folder called “My First Node JS”

*mkdir My First Node JS*

**Step 2**: Go inside the project & create a file in the root called app.js

*cd My-First-Node-JS  
touch app.js*

**Step 3:** Paste this in your app.js file:

const http = require('http');

var server = http.createServer(function (request, response) {  
 response.writeHead(200, {"Content-Type": "text/plain"});  
 response.end("My First Node JS");  
});

server.listen(4000);

This http variable contains a function called create Server. This is all you need to do to create an http server.

*Line 2, 3, 4:* This function is a callback → We use the response variable that’s passed in to the callback to write the head and pass in the content type and we end that response with hello world. → This function returns an object that we are going to put in to our server variable and this object is going to have another function called “listen” →

*Line 5:* the minute that we call listen that’s when our server is going to start running as we listen to this port. You can choose any port.

By default, HTTP uses port80 but ideally you should specify a port.

**Step 4:** You can then run your server by going to your terminal and typing:

node app.js

**Step 5:** Now go to your port…

http://localhost:4000/

You should see “My First Node JS”. :)

Congratulations, you now have a server running!

# What makes Node.js so great?

It’s a JavaScript runtime built on **Chrome’s V8 JavaScript engine**.  
- Both Node and the JS that is executed inside of your browser are running on the same engine → It’s an open source engine that takes JS code and compiles it too much faster machine code → this is what makes Node.js so fast!

Uses an event-driven, non-blocking I/O model that makes it light weight & efficient  
- *What is event-driven programming?*Basically, it’s a different way of thinking about your program flow. The flow of your program is defined by the events that are taking place. (see below for more details)  
- *What is I/O?*Input/output  
- *Difference between blocking & non-blocking software development* Blocking methods execute synchronously and non-blocking methods execute asynchronously. (see below for more details)

Node.js’ package ecosystem, [npm](https://www.npmjs.com/), is the largest ecosystem if open source libraries in the world (see below for more details)

Let’s dive in to what all this means in more detail. The following are a few concepts that we need to understand:

# What is Event Driven Programming?

*Event Driven Programming*is a [computer programming](https://www.computerhope.com/jargon/p/progming.htm) paradigm in which [control flow](https://www.computerhope.com/jargon/c/contflow.htm) of the [program](https://www.computerhope.com/jargon/p/program.htm) is determined by the occurrence of [events](https://www.computerhope.com/jargon/e/event.htm). These events are monitored by [code](https://www.computerhope.com/jargon/c/code.htm) known as an [event listener](https://www.computerhope.com/jargon/e/event-listener.htm) that, if it detects that its assigned event has occurred, runs an event “handler”, typically a callback [function](https://www.computerhope.com/jargon/f/function.htm) or [method](https://www.computerhope.com/jargon/m/method.htm). This handler deals with the event by responding to it with program code.

If you look back at the server function, the response only runs once we get a request — **that’s**event driven programming:

....  
var server = http.createServer(function (**request**, **response**) {   
....

It doesn’t just happen automatically. There is a process that happens and you have to wait for that.

const fs = require(‘fs’);

var contents = fs.readFileSync('package.json').toString();  
console.log(contents);

\*\*\*When running Node you want to have your application as asynchronous as possible since it is constantly running an event loop. Even things like reading a file can be run asynchronously. The above snippet of code is synchronous. Let’s see how we can improve it to avoid ‘blocking’…

Asynchronous version:

const fs = require(‘fs’);

fs.readFile('package.json', function (err, buf){  
 console.log(buf.toString());  
});

In the above snippet, I am passing a callback as opposed to having it freeze while it reads the file. The function is a bit different than the synchronous version we had initially. By passing a callback, the minute it reads a chunk of that file it will execute the callback and move on, letting other events run. This solves the ‘blocking’ issue.

GOAL: \*\*\*In event driven programming your goal is to get the events timing tied as closely to the actual data flow. You want it to pretty much deal with the chunk of data you need.

# Callback Style Programming

We have mentioned callbacks a lot — It is important to understand this concept: The event loops result in callback type programming. In simple terms, it’s where you end up splitting your program in to smaller & smaller chunks until each chunk is mapped to operation with data.

This kind of program caused ‘*callback insanity*’ → many nested callbacks. It makes it not as readable and manageable. Most importantly, it makes it harder to debug. So how should we handle it??

Promises style programming!!!

# Promises

## What is Promises style programming?

When a function will return a promise object that will return something in the future. It promises that it will do something for you. You can also chain promises together and it really helps simplify your code.

promise.then(function(result) {  
 console.log(result); // "Stuff worked!"  
}, function(err) {  
 console.log(err); // Error: "It broke"  
});

then() takes two arguments, a callback for a success case, and another for the failure case

Chaining: then() isn't the end of the story, you can chain then together to transform values or run additional async actions one after another.

promise.then(function(val) {  
 console.log(val); // 1  
 return val + 4;  
}).then(function(val) {  
 console.log(val); // 5  
})

# Event Emitters

Ok so we reviewed some important concepts but now what happens when we want to pull data from the server that we built? Well, we would have to change it up a bit and use ‘event emitters’.

## What are Event Emitters?

An Event emitter, as it sounds, is just something that triggers an event to which anyone can listen. Different libraries offer different implementations and for different purposes, but the basic idea is to provide a framework for issuing events and subscribing to them.

In node.js an event can be described simply as a string with a corresponding callback. An event can be “emitted” (or in other words, the corresponding callback be called) multiple times or you can choose to only listen for the first time it is emitted.

Let’s say your node.js app is connected to an office automation system

var office = require('./office');

office.door.on('knock', function() {   
 /// do something   
});

The ‘on’ function pretty much says whenever I get a data event, run this function. As a client of this object you start listening to “knock” events by using the. on method & passing in a function that gets called whenever an event with that specific name happens

The on or add Listener method (basically the subscription method) allows you to choose the event to watch for and the callback to be called.

The emit method (the publish method), on the other hand, allows you to "emit" an event, which causes all callbacks registered to the event to 'fire', (get called).

// get the reference of EventEmitter class of events module  
const events = require('events');

//create an object of EventEmitter class by using above reference  
const em = new events.EventEmitter();

// register a listener for the 'knock' event  
em.on('knock', function (data) {  
 console.log('Received the knock event: ' + data);  
});

// trigger an event called 'knock'  
em.emit('knock', "who's there?");

In NodeJs, any object that emits an event is an instance of the EventEmitter class

If you run this script the response in terminal will be:

Received the knock event: who’s there?

# Node Package Manager (NPM)

What is NPM?  
It is the official package manager for Node and is bundled & installed automatically with the environment.

Here is how you would install a specific package via terminal:

npm install — save package\_name  
npm update

\*\*by writing ‘save’ it will save the package in to your *package.json* folder which handles your dependencies.

**What is***package.json***?**

{  
 “name”: “Node101”,  
 “version”: “0.0.0”,  
 “description”: “**Sample Code**”,  
 “main”: “hello\_world.js”,  
 “author”: {  
 “name”: “Linda Haviv”,  
 “email”: “”  
 }  
}

There are lots of great modules on [NPM](https://www.npmjs.com/):

Popular NPM modules:

- 7053 [underscore](https://www.npmjs.com/package/underscore)  
- 6458 [async](https://www.npmjs.com/package/async)  
- 5591 [request](https://www.npmjs.com/package/request)  
- 4931 [lodash](https://www.npmjs.com/package/lodash)  
- 3630 [commander](https://www.npmjs.com/package/commander)  
- 3543 [express](https://www.npmjs.com/package/express)  
- 2708 [yargs](https://www.npmjs.com/package/yargs)  
- 2634 [coffee-script](https://www.npmjs.com/package/coffee-script)

*How does it work?*  
The best way to manage locally installed npm packages is to create a**package.json** file.

A **package.json** file affords you a lot of great things:

It serves as documentation for what packages your project depends on.

It allows you to specify the versions of a package that your project can use using [semantic versioning rules](https://docs.npmjs.com/getting-started/semantic-versioning).

Makes your build reproducible which means that its *way* easier to share with other developers.

**\*\*\*(**source:[**npmjs docs**](https://docs.npmjs.com/getting-started/using-a-package.json)**)\*\*\***

In general, Node is a great tool to quickly get an api up and running

Note: The first thing you do when you get any kind of Node project is that you navigate to the directory that it’s in and in terminal write:

npm install

This command will look at everything you have in your package.json file and it will install all of those things

## How to use Modules in Node.js?

Now that we listed some popular modules, you might be wondering how to use them in your projects. To do that, we need to use a function in Node called **require.**The demo server we previously built used the built-in **require**module.

const http = require(‘http’);

**Require** will let us do 3 things:

load in modules that come with Node.js — example: [http](https://nodejs.org/api/http.html) module, [FS](https://nodejs.org/api/fs.html)module

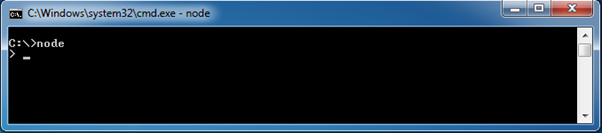
load in third party libraries that allows us to write less code — example: [express.js](https://www.npmjs.com/package/express)

require our very own files. This will allow us to break up our application to multiple smaller files which is essential for building real-world apps.

# Node.js Console - REPL

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.

[](http://www.tutorialsteacher.com/Content/images/nodejs/node-repl.png)Launch Node.js REPL

You can now test pretty much any Node.js/JavaScript expression in REPL. For example, if your 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.

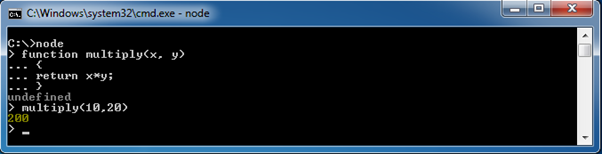
> "Hello" + "World"   
Hello World

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 next line. Write. break to get out of continuity mode.

For example, you can define a function and execute it as shown below.

[](http://www.tutorialsteacher.com/Content/images/nodejs/nodejs-example2.png)

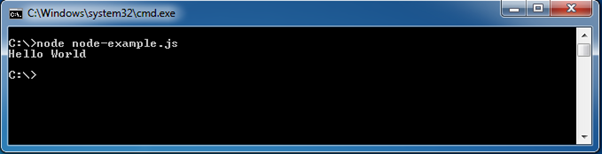
## Node.js Example in REPL

You can execute an external JavaScript file by writing node fileName command. For example, assume that node-example.js is on C drive of your PC with following code.

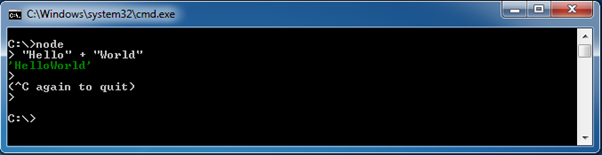
node-example.js

**console.log("Hello World");**

Now, you can execute node-exampel.js from command prompt as shown below.

[](http://www.tutorialsteacher.com/Content/images/nodejs/run-nodejs-external-file.png)Run External JavaScript file

To exit from the REPL terminal, press Ctrl + C twice or write .exit and press Enter.

[](http://www.tutorialsteacher.com/Content/images/nodejs/quit-repl.png)Quit from REPL

Thus, you can execute any Node.js/JavaScript code in the node shell (REPL). This will give you a result which is similar to the one you will get in the console of Google Chrome browser.

Note: ECMAScript implementation in Node.js and browsers is slightly different. For example, {}+{} is '[object Object][object Object]' in Node.js REPL, whereas the same code is NaN in the Chrome console because of the automatic semicolon insertion feature. However, mostly Node.js REPL and the Chrome/Firefox consoles are similar.

# REPL Commands

The following table lists important REPL commands.

| REPL Command | Description |
| --- | --- |
| .help | Display help on all the commands |
| tab Keys | Display the list of all commands. |
| Up/Down Keys | See previous commands applied in REPL. |
| .save filename | Save current Node REPL session to a file. |
| .load filename | Load the specified file in the current Node REPL session. |
| ctrl + c | Terminate the current command. |
| ctrl + c (twice) | Exit from the REPL. |
| ctrl + d | Exit from the REPL. |
| .break | Exit from multiline expression. |
| .clear | Exit from multiline expression. |

# Node.js Basics

Node.js supports JavaScript. So, JavaScript syntax on Node.js is like the browser's JavaScript syntax.

## Primitive Types

Node.js includes following primitive types:

*String*

*Number*

*Boolean*

*Undefined*

*Null*

*RegExp*

Everything else is an object in Node.js.

## Loose Typing

JavaScript in Node.js supports loose typing like the browser's JavaScript. Use var keyword to declare a variable of any type.

## Object Literal

Object literal syntax is same as browser's JavaScript.

Example: Object

var obj = {

authorName: ‘Ankit Kumar’,

language: 'Node.js'

}

## Functions

Functions are first class citizens in Node's JavaScript, like the browser's JavaScript. A function can have attributes and properties also. It can be treated like a class in JavaScript.

Example: Function

function Display(x) {

console.log(x);

}

Display(300);

## Buffer

Node.js includes an additional data type called Buffer (not available in browser's JavaScript). Buffer is mainly used to store binary data, while reading from a file or receiving packets over the network.

## Process Object

Each Node.js script runs in a process. It includes process object to get all the information about the current process of Node.js application.

The following example shows how to get process information in REPL using process object.

> process.execPath

'C:\\Program Files\\nodejs\\node.exe'

> process.pid

1652

> process.cwd()

'C:\\'

## Defaults to local

Node's JavaScript is different from browser's JavaScript when it comes to global scope. In the browser's JavaScript, variables declared without var keyword become global. In Node.js, everything becomes local by default.

## Access Global Scope

In a browser, global scope is the window object. In Node.js, global object represents the global scope.

To add something in global scope, you need to export it using export or module.export. The same way, import modules/object using require () function to access it from the global scope.

For example, to export an object in Node.js, use exports.name = object.

Example:

exports.log = {

console: function(msg) {

console.log(msg);

},

file: function(msg) {

// log to file here

}

}

Now, you can import log object using require() function and use it anywhere in your Node.js project.

# Node.js Module

Module in Node.js is a simple or complex functionality organized in single or multiple JavaScript files which can be reused throughout the Node.js application.

Each module in Node.js has its own context, so it cannot interfere with other modules or pollute global scope. Also, each module can be placed in a separate .js file under a separate folder.

Node.js implements CommonJS modules standard. CommonJS is a group of volunteers who define JavaScript standards for web server, desktop, and console application.

## Node.js Module Types

Node.js includes three types of modules:

Core Modules

Local Modules

Third Party Modules

Node.js Core Modules

Node.js is a light weight framework. The core modules include bare minimum functionalities of Node.js. These core modules are compiled into its binary distribution and load automatically when Node.js process starts. However, you need to import the core module first to use it in your application.

The following table lists some of the important core modules in Node.js.

|  |  |
| --- | --- |
| **Core Module** | **Description** |
| http | http module includes classes, methods and events to create Node.js http server. |
| url | url module includes methods for URL resolution and parsing. |
| querystring | querystring module includes methods to deal with query string. |
| path | path module includes methods to deal with file paths. |
| util | util module includes utility functions useful for programmers. |

## Loading Core Modules

In order to use Node.js core or NPM modules, you first need to import it using require() function as shown below.

var module = require('module\_name');

As per above syntax, specify the module name in the require() function. The require() function will return an object, function, property or any other JavaScript type, depending on what the specified module returns.

The following example demonstrates how to use Node.js http module to create a web server.

### Example: Load and Use Core http Module

var http = require('http');

var server = http.createServer(function(req, res){

//write code here

});

server.listen(5000);

In the above example, require() function returns an object because http module returns its functionality as an object, you can then use its properties and methods using dot notation e.g. http.createServer().

In this way, you can load and use Node.js core modules in your application.

# Node.js Web Server

To access web pages of any web application, you need a web server. The web server will handle all the http requests for the web application e.g IIS is a web server for ASP.NET web applications and Apache is a web server for PHP or Java web applications.

Node.js provides capabilities to create your own web server which will handle HTTP requests asynchronously. You can use IIS or Apache to run Node.js web application but it is recommended to use Node.js web server.

## Create Node.js Web Server

Node.js makes it easy to create a simple web server that processes incoming requests asynchronously.

The following example is a simple Node.js web server contained in server.js file.

server.js

var http = require('http'); // 1 - Import Node.js core module

var server = http.createServer(function (req, res) { // 2 - creating server

//handle incomming requests here..

});

server.listen(5000); //3 - listen for any incoming requests

console.log('Node.js web server at port 5000 is running..')

In the above example, we import the http module using require() function. The http module is a core module of Node.js, so no need to install it using NPM. The next step is to call createServer() method of http and specify callback function with request and response parameter. Finally, call listen() method of server object which was returned from createServer() method with port number, to start listening to incoming requests on port 5000. You can specify any unused port here.

Run the above web server by writing node server.js command in command prompt or terminal window and it will display message as shown below.

C:\> node server.js

Node.js web server at port 5000 is running..

This is how you create a Node.js web server using simple steps. Now, let's see how to handle HTTP request and send response in Node.js web server.

## Handle HTTP Request

The http.createServer() method includes request and response parameters which is supplied by Node.js. The request object can be used to get information about the current HTTP request e.g., url, request header, and data. The response object can be used to send a response for a current HTTP request.

The following example demonstrates handling HTTP request and response in Node.js.

server.js

var http = require('http'); // Import Node.js core module

var server = http.createServer(function (req, res) { //create web server

if (req.url == '/') { //check the URL of the current request

// set response header

res.writeHead(200, { 'Content-Type': 'text/html' });

// set response content

res.write('<html><body><p>This is home Page.</p></body></html>');

res.end();

}

else if (req.url == "/student") {

res.writeHead(200, { 'Content-Type': 'text/html' });

res.write('<html><body><p>This is student Page.</p></body></html>');

res.end();

}

else if (req.url == "/admin") {

res.writeHead(200, { 'Content-Type': 'text/html' });

res.write('<html><body><p>This is admin Page.</p></body></html>');

res.end();

}

else

res.end('Invalid Request!');

});

server.listen(5000); //6 - listen for any incoming requests

console.log('Node.js web server at port 5000 is running..')

In the above example, req.url is used to check the url of the current request and based on that it sends the response. To send a response, first it sets the response header using writeHead() method and then writes a string as a response body using write() method. Finally, Node.js web server sends the response using end() method.

Now, run the above web server as shown below.

C:\> node server.js

Node.js web server at port 5000 is running..

To test it, you can use the command-line program curl, which most Mac and Linux machines have pre-installed.

curl -i http://localhost:5000

You should see the following response.

**HTTP/1.1 200 OK**

**Content-Type: text/plain**

**Date: Tue, 8 Sep 2015 03:05:08 GMT**

**Connection: keep-alive**

**This is home page.**

For Windows users, point your browser to http://localhost:5000 and see the result.

The same way, point your browser to http://localhost:5000/student and see the result.

It will display "Invalid Request" for all requests other than the above URLs.

## Sending JSON Response

The following example demonstrates how to serve JSON response from the Node.js web server.

**server.js**

**var http = require('http');**

**var server = http.createServer(function (req, res) {**

**if (req.url == '/data') { //check the URL of the current request**

**res.writeHead(200, { 'Content-Type': 'application/json' });**

**res.write(JSON.stringify({ message: "Hello World"}));**

**res.end();**

**}**

**});**

**server.listen(5000);**

**console.log('Node.js web server at port 5000 is running..')**

So, this way you can create a simple web server that serves different responses.

# Node.js File System

Node.js includes fs module to access physical file system. The fs module is responsible for all the asynchronous or synchronous file I/O operations.

Let's see some of the common I/O operation examples using fs module.

## Reading File

Use fs.readFile() method to read the physical file asynchronously.

fs.readFile(fileName [,options], callback)

Parameter Description:

* filename: Full path and name of the file as a string.
* options: The options parameter can be an object or string which can include encoding and flag. The default encoding is utf8 and default flag is "r".
* callback: A function with two parameters err and fd. This will get called when readFile operation completes.

The following example demonstrates reading existing TestFile.txt asynchronously.

Example: Reading File

var fs = require('fs');

fs.readFile('TestFile.txt', function (err, data) {

if (err) throw err;

console.log(data);

});

The above example reads TestFile.txt (on Windows) asynchronously and executes callback function when read operation completes. This read operation either throws an error or completes successfully. The err parameter contains error information if any. The data parameter contains the content of the specified file.

The following is a sample TextFile.txt file.

**TextFile.txt**

This is test file to test fs module of Node.js

Now, run the above example and see the result as shown below.

C:\> node server.js   
This is test file to test fs module of Node.js

Use fs.readFileSync() method to read file synchronously as shown below.

## Example: Reading File Synchronously

var fs = require('fs');

var data = fs.readFileSync('dummyfile.txt', 'utf8');

console.log(data);

## Writing File

Use fs.writeFile() method to write data to a file. If file already exists then it overwrites the existing content otherwise it creates a new file and writes data into it.S:

fs.writeFile(filename, data[, options], callback)

## Parameter Description:

**filename:** Full path and name of the file as a string.

**Data:** The content to be written in a file.

**options:** The options parameter can be an object or string which can include encoding, mode and flag. The default encoding is utf8 and default flag is "r".

**callback:** A function with two parameters err and fd. This will get called when write operation completes.

The following example creates a new file called test.txt and writes "Hello World" into it asynchronously.

Example: Creating & Writing File

var fs = require('fs');

fs.writeFile('test.txt', 'Hello World!', function (err) {

if (err)

console.log(err);

else

console.log('Write operation complete.');

});

In the same way, use fs.appendFile() method to append the content to an existing file.

Example: Append File Content

var fs = require('fs');

fs.appendFile('test.txt', 'Hello World!', function (err) {

if (err)

console.log(err);

else

console.log('Append operation complete.');

});

# Open File

Alternatively, you can open a file for reading or writing using fs.open() method.

Signature:

fs.open(path, flags[, mode], callback)

Parameter Description:

* path: Full path with name of the file as a string.
* Flag: The flag to perform operation
* Mode: The mode for read, write or readwrite. Defaults to 0666 readwrite.
* callback: A function with two parameters err and fd. This will get called when file open operation completes.

### **Flags**

The following table lists all the flags which can be used in read/write operation.

| Flag | Description |
| --- | --- |
| r | Open file for reading. An exception occurs if the file does not exist. |
| r+ | Open file for reading and writing. An exception occurs if the file does not exist. |
| rs | Open file for reading in synchronous mode. |
| rs+ | Open file for reading and writing, telling the OS to open it synchronously. See notes for 'rs' about using this with caution. |
| w | Open file for writing. The file is created (if it does not exist) or truncated (if it exists). |
| wx | Like 'w' but fails if path exists. |
| w+ | Open file for reading and writing. The file is created (if it does not exist) or truncated (if it exists). |
| wx+ | Like 'w+' but fails if path exists. |
| a | Open file for appending. The file is created if it does not exist. |
| ax | Like 'a' but fails if path exists. |
| a+ | Open file for reading and appending. The file is created if it does not exist. |
| ax+ | Like 'a+' but fails if path exists. |

The following example opens an existing file and reads its content.

Example:File open and read

var fs = require('fs');

fs.open('TestFile.txt', 'r', function (err, fd) {

if (err) {

return console.error(err);

}

var buffr = new Buffer(1024);

fs.read(fd, buffr, 0, buffr.length, 0, function (err, bytes) {

if (err) throw err;

// Print only read bytes to avoid junk.

if (bytes > 0) {

console.log(buffr.slice(0, bytes).toString());

}

// Close the opened file.

fs.close(fd, function (err) {

if (err) throw err;

});

});

});

# Delete File

Use fs.unlink() method to delete an existing file.

Signature:

fs.unlink(path, callback);

The following example deletes an existing file.

## Example:File Open and Read

var fs = require('fs');

fs.unlink('test.txt', function () {

console.log('write operation complete.');

});

# Important method of fs module

| Method | Description |
| --- | --- |
| fs.readFile(fileName [,options], callback) | Reads existing file. |
| fs.writeFile(filename, data[, options], callback) | Writes to the file. If file exists then overwrite the content otherwise creates new file. |
| fs.open(path, flags[, mode], callback) | Opens file for reading or writing. |
| fs.rename(oldPath, newPath, callback) | Renames an existing file. |
| fs.chown(path, uid, gid, callback) | Asynchronous chown. |
| fs.stat(path, callback) | Returns fs.stat object which includes important file statistics. |
| fs.link(srcpath, dstpath, callback) | Links file asynchronously. |
| fs.symlink(destination, path[, type], callback) | Symlink asynchronously. |
| fs.rmdir(path, callback) | Renames an existing directory. |
| fs.mkdir(path[, mode], callback) | Creates a new directory. |
| fs.readdir(path, callback) | Reads the content of the specified directory. |
| fs.utimes(path, atime, mtime, callback) | Changes the timestamp of the file. |
| fs.exists(path, callback) | Determines whether the specified file exists or not. |
| fs.access(path[, mode], callback) | Tests a user's permissions for the specified file. |
| fs.appendFile(file, data[, options], callback) | Appends new content to the existing file. |

# Express

$ npm install express –save

## Install ExpressJS

After Successfully installing NodeJS you can proceed with installing ExpressJS.

Start by:

Opening Terminal, this should open your home directory if you’re on Mac

Change directory to your Documents Folder by typing

cd documents

Create a new folder by typing

mkdir express-practice

you can name the folder whatever you want.

Change Directory to the newly created folder by executing command:

cd express-practice

now run

npm install express --save

This would install Express in the express-practice directory and save it in the dependencies list

Step 3: Getting it to Run in the Browser

Now you need your ExpressJS Application to run in the Browser, you can achieve this by following these steps:

run npm init , this command prompts you for a number of things, such as the name and version of your application. For now, you can simply hit RETURN/ENTER to accept the defaults for all of them.

Open express-practice project folder in your Favorite Text Editor.

Create a new file named index.js

put the code below in that file

var express = require('express');

var app = express();

app.get('/', function (req, res) {

res.send('Hello World!');

});

app.listen(4000, function () {

console.log('Example app listening on port 4000!');

})

5. Save the file.

6. Now go to Terminal, make sure you’re in the express-practice folder directory.

7. Run node index.js

8. Now open http://localhost:4000 on your browser

At this point you have successfully set up your first ExpressJS Application

Express is the most popular framework for Node apps, and it features middleware using continuation passing. When you want to run the same code for potentially many different routes, the right place for that code is probably middleware.

Middleware is a function that gets passed the request and response objects, along with a continuation function to call, called next(). Imagine you want to add a requestId to each request/response pair so that you can easily trace them back to the individual request when you’re debugging or searching your logs for something.

You can write some middleware like this:

require('dotenv').config();

const express = require('express');

const cuid = require('cuid');

const app = express();

// request id middleware

const requestId = (req, res, next) => {

const requestId = cuid();

req.id = requestId;

res.id = requestId;

// pass continuation to next middleware

next();

};

app.use(requestId);

app.get('/', (req, res) => {

res.send('\n\nHello, world!\n\n');

});

module.exports = app;

## Database integration

Adding the capability to connect databases to Express apps is just a matter of loading an appropriate Node.js driver for the database in your app. This document briefly explains how to add and use some of the most popular Node.js modules for database systems in your Express app:

* Cassandra
* Couchbase
* CouchDB
* LevelDB
* MySQL
* MongoDB
* Neo4j
* Oracle
* PostgreSQL
* Redis
* SQL Server
* SQLite
* ElasticSearch

### MongoDB

**Module:** mongodb

**Installation**

$ npm install mongodb

Example (v2.\*)

var MongoClient = require('mongodb').MongoClient

MongoClient.connect('mongodb://localhost:27017/animals', function (err, db) {

if (err) throw err

db.collection('mammals').find().toArray(function (err, result) {

if (err) throw err

console.log(result)

})

})

Example (v3.\*)

var MongoClient = require('mongodb').MongoClient

MongoClient.connect('mongodb://localhost:27017/animals', function (err, client) {

if (err) throw err

var db = client.db('animals')

db.collection('mammals').find().toArray(function (err, result) {

if (err) throw err

console.log(result)

})

})

# Structuring Your API

The way you choose to structure your API is one of the most important decisions you’ll make. You must ensure that it’s smart, flexible, and easy to use – this is necessary. If it’s not easy to use, other developers will not understand what you’re building nor will they be able to figure out how to build on top of it. Think before you build (I know. Planning sucks. Especially when you are excited to get going, but it \*pays off\*).

├── build.sh

├── dist

│ ├── …

├── package.json

├── src

│ ├── config

│ │ └── index.js

│ ├── controllers

│ │ ├── …

│ ├── models

│ │ ├── …

│ ├── package.json

│ ├── routes

│ │ ├── …

│ ├── server.js

│ ├── utils

│ │ ├── …

All source code is stored in /src. It compiles down from ES6+ to ES5 into the /dist directory for execution on the server. You’re probably asking yourself why you’d take the extra step to write in something that is just going to be compiled down? Good question. ES6+ standards provide some killer additional functionalities, such as arrow functions, modified scoping, destructuring, rest/spread parameter handling, and more!

Let’s have a look at the compilation that takes place in the build.sh file:

#!/bin/bash

rm -rf dist && mkdir dist

npx babel src --out-dir dist --ignore node\_modules

cp src/package.json dist

cd dist && yarn install --production --modules-folder node\_modules

#!/bin/bash

Denotes that this is an executable bash file

rm –rf dist && mkdir dist

Removes the /dist directory if it exists (cleanup).

Creates a new /dist directory.

npx babel src —out–dir dist —ignore node\_modules

Compiles every file to ES5 and moves the files to the /dist directory, with the exception of node\_modules (those are already compiled).

cp src/package.json dist

By design, npx doesn’t migrate json files, so we need to copy it ourselves using the cp command.

cd dist && yarn install —production —modules–folder node\_modules

Move into the /dist directory and install the npm modules using yarn

Running the build is as simple as running the following command from your terminal:

$ ./build.sh

Note: You will need to ensure that the build.sh file is executable.

You can create an npm script like:

"scripts": {

"build": "./build.sh"

}

Which can be executed by running the following from your terminal:

$ yarn build

**The Main File**

The following file, server.js, contains the most important logic and sits on the top-level of our codebase. The beginning portion imports all of the necessary npm modules, followed by our config and logger utility.

Next, we take advantage of the Express use method to invoke several of our imported middleware libraries (cors, compression, and our body-parser). \*\*Please note\*\* that there are several other middleware libraries that we include for additional functionality (e.g. email, logging, jwt authentication, etc.). Finally, after a bit of initialization, we dynamically include all routes and pass the API context to the route for binding.

// import npm modules

import fs from 'fs';

import path from 'path';

import express from 'express';

import bodyParser from 'body-parser';

import cors from 'cors';

import winston from 'winston';

import compression from 'compression';

import expressWinston from 'express-winston';

import winstonPapertrail from 'winston-papertrail';

import jwt from 'express-jwt';

// import custom configuration and utilities

import config from './config';

import logger from './utils/logger';

// initialize the api

const api = express();

// initialize middleware

api.use(cors());

api.use(compression());

api.use(bodyParser.urlencoded({ extended: true }));

api.use(bodyParser.json());

// ignore authentication on the following routes

api.use(

jwt({ secret: config.jwt.secret }).unless({

path: [

'/',

'/auth/signup',

'/auth/login',

'/auth/forgot-password',

'/auth/reset-password',

],

}),

);

// throw an error if a jwt is not passed in the request

api.use((err, req, res, next) => {

if (err.name === 'UnauthorizedError') {

res.status(401).send('Missing authentication credentials.');

}

});

// initialize our logger (in our case, winston + papertrail)

api.use(

expressWinston.logger({

transports: [

new winston.transports.Papertrail({

host: config.logger.host,

port: config.logger.port,

level: 'error',

}),

],

meta: true,

}),

);

// listen on the designated port found in the configuration

api.listen(config.server.port, err => {

if (err) {

logger.error(err);

process.exit(1);

}

// require the database library (which instantiates a connection to mongodb)

require('./utils/db');

// loop through all routes and dynamically require them – passing api

fs.readdirSync(path.join(\_\_dirname, 'routes')).map(file => {

require('./routes/' + file)(api);

});

// output the status of the api in the terminal

logger.info(`API is now running on port ${config.server.port} in ${config.env} mode`);

});

module.exports = api;

Note: The customer logger module can be used with most logging services (Papertrail, Loggly, etc.)

# Routing

To keep things tidy and organized, all routing logic (e.g. GET /users) is kept in its own route file inside of a /routes directory.

import User from '../controllers/user';

module.exports = api => {

api.route('/users').get(User.list);

api.route('/users/:userId').get(User.get);

api.route('/users).post(User.post);

api.route('/users/:userId').put(User.put);

api.route('/users/:userId').delete(User.delete);

};

As you can see, the contents of the route file above hold all references to the controllers for GET, POST, PUT, and DELETE operations. This works because we import and reference the User Controller, passing along the necessary parameters and/or data with every API call.

# Controllers

Controllers include the database model associated with the data that they will be handling, receiving data from the routes, and then making an informed decision on how to handle the data. Finally, the controllers communicate through the models which then talk to the database, and return a status code with a payload.

// import npm modules

import async from 'async';

import validator from 'validator';

// import user model

import User from '../models/user';

// import custom utilities

import logger from '../utils/logger';

// retrieve a list of all users

exports.list = (req, res) => {

const query = req.query || {};

User.apiQuery(query)

// limit the information returned (server side) – e.g. no password

.select('name email username bio url twitter background')

.then(users => {

res.json(users);

})

.catch(err => {

logger.error(err);

res.status(422).send(err.errors);

});

};

// retrieve a specific user using the user id (in our case, the user from the jwt)

exports.get = (req, res) => {

const data = Object.assign(req.body, { user: req.user.sub }) || {};

User.findById(data.user)

.then(user => {

user.password = undefined;

user.recoveryCode = undefined;

res.json(user);

})

.catch(err => {

logger.error(err);

res.status(422).send(err.errors);

});

};

// update a specific user

exports.put = (req, res) => {

const data = Object.assign(req.body, { user: req.user.sub }) || {};

if (data.email && !validator.isEmail(data.email)) {

return res.status(422).send('Invalid email address.');

}

if (data.username && !validator.isAlphanumeric(data.username)) {

return res.status(422).send('Usernames must be alphanumeric.');

}

User.findByIdAndUpdate({ \_id: data.user }, data, { new: true })

.then(user => {

if (!user) {

return res.sendStatus(404);

}

user.password = undefined;

user.recoveryCode = undefined;

res.json(user);

})

.catch(err => {

logger.error(err);

res.status(422).send(err.errors);

});

};

// create a user

exports.post = (req, res) => {

const data = Object.assign({}, req.body, { user: req.user.sub }) || {};

User.create(data)

.then(user => {

res.json(user);

})

.catch(err => {

logger.error(err);

res.status(500).send(err);

});

};

// remove a user record (in our case, set the active flag to false to preserve data)

exports.delete = (req, res) => {

User.findByIdAndUpdate(

{ \_id: req.params.user },

{ active: false },

{

new: true,

},

)

.then(user => {

if (!user) {

return res.sendStatus(404);

}

res.sendStatus(204);

})

.catch(err => {

logger.error(err);

res.status(422).send(err.errors);

});

};

# Mongoose Models (MongoDB)

Mongoose is a wonderful ODM (Object Data Modeling) library for Node.js and MongoDB. If you’re familiar with the reference ORM (Object Resource Mapping) and libraries for Node.js, such as Sequelize and Bookshelf, Mongoose is straightforward. The massive benefit with Mongoose is how easy it is to structure MongoDB schemas – there’s no need to fuss around with custom business logic.

What’s even more exciting are the many goodies like middleware, plugins, object population, and schema validation either baked in, or one yarn (I love yarn) or one npm install away. It’s truly remarkable how popular the project has become among developers who use MongoDB.

When it comes to Mongoose models, I tend to keep things somewhat flat (or at least a maximum of 3 deeply nested objects) to avoid confusion. Here’s an example of a user model pulled directly from a project currently under development here at Stream:

// import npm modules

import mongoose, { Schema } from 'mongoose';

import bcrypt from 'mongoose-bcrypt';

import timestamps from 'mongoose-timestamp';

import mongooseStringQuery from 'mongoose-string-query';

// import custom utilities

import logger from '../utils/logger';

import email from '../utils/email';

import events from '../utils/events';

// build user schema

export const UserSchema = new Schema(

{

email: {

type: String,

lowercase: true,

trim: true,

index: true,

unique: true,

required: true,

},

username: {

type: String,

lowercase: true,

trim: true,

index: true,

unique: true,

required: true,

},

password: {

type: String,

required: true,

bcrypt: true,

},

name: {

type: String,

trim: true,

required: true,

},

bio: {

type: String,

trim: true,

default: '',

},

url: {

type: String,

trim: true,

default: '',

},

twitter: {

type: String,

trim: true,

default: '',

},

background: {

type: Number,

default: 1,

},

interests: {

type: Schema.Types.Mixed,

default: [],

},

preferences: {

notifications: {

daily: {

type: Boolean,

default: false,

},

weekly: {

type: Boolean,

default: true,

},

follows: {

type: Boolean,

default: true,

},

},

},

recoveryCode: {

type: String,

trim: true,

default: '',

},

active: {

type: Boolean,

default: true,

},

admin: {

type: Boolean,

default: false,

},

},

{ collection: 'users' },

);

// pre-save hook that sends welcome email via custom email utility

UserSchema.pre('save', function(next) {

if (!this.isNew) {

next();

}

email({

type: 'welcome',

email: this.email,

})

.then(() => {

next();

})

.catch(err => {

logger.error(err);

next();

});

});

// pre-save hook that sends password recovery email via custom email utility

UserSchema.pre('findOneAndUpdate', function(next) {

if (!this.\_update.recoveryCode) {

return next();

}

email({

type: 'password',

email: this.\_conditions.email,

passcode: this.\_update.recoveryCode,

})

.then(() => {

next();

})

.catch(err => {

logger.error(err);

next();

});

});

// require plugins

UserSchema.plugin(bcrypt); // automatically bcrypts passwords

UserSchema.plugin(timestamps); // automatically adds createdAt and updatedAt timestamps

UserSchema.plugin(mongooseStringQuery); // enables query capabilities (e.g. ?foo=bar)

UserSchema.index({ email: 1, username: 1 }); // compound index on email + username

module.exports = exports = mongoose.model('User', UserSchema); // export model for use

Note: When it comes to hosting and running MongoDB, I like to use MongoDB Atlas. It’s a database as a service provided by the makers of MongoDB themselves. If you don’t want to use a free MongoDB Atlas instance, you’re welcomed to use a local version. Additionally, if you want to monitor your data, MongoDB Compass is an excellent choice!

# Utilities

Custom utilities can be used for a variety of things – basically, anything you want. Some examples include establishing database connections, sending emails, logging to an external service, and even communicating with HTTP based service here at Stream.

Here’s an example of a utility called the Stream Personalization REST API. This integration was completed in about a dozen lines of code:

// import npm modules

import axios from 'axios';

import jwt from 'jsonwebtoken';

// import custom utilities

import config from '../../config';

const personalization = data => {

// setup promise

return new Promise((resolve, reject) => {

// build jwt for signing the API call

const token = jwt.sign(

{

action: '\*',

feed\_id: '\*',

resource: '\*',

user\_id: '\*',

},

config.stream.apiSecret,

{ algorithm: 'HS256', noTimestamp: true },

);

// initiate call via axios (http module)

return axios({

baseURL: config.stream.baseUrl,

headers: {

Authorization: token,

'Content-Type': 'application/json',

'Stream-Auth-Type': 'jwt',

},

method: 'GET',

params: {

api\_key: config.stream.apiKey,

user\_id: data.userId,

},

url: data.endpoint,

})

.then(res => {

// map over results and deserialize

const data = res.data.results.map(result => {

return result.foreign\_id.split(':')[1];

});

// successfully resolve call and return deserialized data

resolve(data);

})

.catch(err => {

// catch and reject with error

reject(err);

});

});

};

export default personalization;

The code above can now be called from any file like so:

personalization({

endpoint: '/user\_recommendations',

userId: req.user.sub, // id is extracted from the jwt

})

.then(users => {

// iterate over users and enrich

users.map(user => {

// do something with the user data

});

})

.catch(err => {

res.status(503).send(err.response.data);

});

# How to Create a Node.js Cluster for Speeding Up Your Apps

Node.js is becoming increasingly popular as a server-side run-time environment, especially for high traffic websites, as statistics show. Also, the availability of several frameworks makes it a good environment for rapid prototyping. Node.js has an event-driven architecture, leveraging a non-blocking I/O API that allows requests being processed asynchronously.

One of the important and often less highlighted features of Node.js is its scalability. In fact, this is the main reason some large companies with heavy traffic are integrating Node.js in their platform (e.g., Microsoft, Yahoo, Uber, and Walmart) or even completely moving their server-side operations to Node.js (e.g., PayPal, eBay, and Groupon).

Each Node.js process runs in a single thread and by default it has a memory limit of 512MB on 32-bit systems and 1GB on 64-bit systems. Although the memory limit can be bumped to ~1GB on 32-bit systems and ~1.7GB on 64-bit systems, both memory and processing power can still become bottlenecks for various processes.

The elegant solution Node.js provides for scaling up the applications is to split a single process into multiple processes or workers, in Node.js terminology. This can be achieved through a cluster module. The cluster module allows you to create child processes (workers), which share all the server ports with the main Node process (master).

## Node.js Cluster Module: what it is and how it works

A cluster is a pool of similar workers running under a parent Node process. Workers are spawned using the fork() method of the child\_processes module. This means workers can share server handles and use IPC (Inter-process communication) to communicate with the parent Node process.

He masters process is in charge of initiating workers and controlling them. You can create an arbitrary number of workers in your master process. Moreover, remember that by default incoming connections are distributed in a round-robin approach among workers (except in Windows). There is another approach to distribute incoming connections, that I won’t discuss here, which hands the assignment over to the OS (default in Windows). Node.js documentation suggests using the default round-robin style as the scheduling policy.

Although using a cluster module sounds complex in theory, it is very straightforward to implement. To start using it, you must include it in your Node.js application:

var cluster = require('cluster);

A cluster module executes the same Node.js process multiple times. Therefore, the first thing you need to do is to identify what portion of the code is for the master process and what portion is for the workers. The cluster module allows you to identify the master process as follows:

if(cluster.isMaster) { ... }

The master process is the process you initiate, which in turn initialize the workers. To start a worker process inside a master process, we’ll use the fork() method:

cluster.fork();

This method returns a worker object that contains some methods and properties about the forked worker. We’ll see some examples in the following section.

A cluster module contains several events. Two common events related to the moments of start and termination of workers are the online and the exit events. online is emitted when the worker is forked and sends the online message. exit is emitted when a worker process dies. Later, we’ll see how we can use these two events to control the lifetime of the workers.

Let’s now put together everything we’ve seen so far and show a complete working example.

Examples

This section features two examples. The first one is a simple application showing how a cluster module is used in a Node.js application. The second one is an Express server taking advantage of Node.js cluster module, which is part of a production code I generally use in large-scale projects.

## How a Cluster Module is Used in a Node.js App

In this first example, we set up a simple server that responds to all incoming requests with a message containing the worker process ID that processed the request. The master process forks four workers. In each of them, we start listening the port 8000 for incoming requests.

The code that implements what I’ve just described, is shown below:

var cluster = require('cluster');

var http = require('http');

var numCPUs = 4;

if (cluster.isMaster) {

for (var i = 0; i < numCPUs; i++) {

cluster.fork();

}

} else {

http.createServer(function(req, res) {

res.writeHead(200);

res.end('process ' + process.pid + ' says hello!');

}).listen(8000);

}

You can test this server on your machine by starting it (run the command node simple.js) and accessing the URL http://127.0.0.1:8000/. When requests are received, they are distributed one at a time to each worker. If a worker is available, it immediately starts processing the request; otherwise it’ll be added to a queue.

There are a few points that are not very efficient in the above example. For instance, imagine if a worker dies for some reason. In this case, you lose one of your workers and if the same happens again, you will end up with a master process with no workers to handle incoming requests. Another issue is related to the number of workers. There are different number of cores/threads in the systems that you deploy your application to. In the mentioned example, to use all the system’s resources, you must manually check the specifications of each deployment server, find how many threads there are available, and update it in your code. In the next example, we’ll see how to make the code more efficient through an Express server.

## How to Develop a Highly Scalable Express Server

Express is one the most popular web application frameworks for Node.js (if not the most popular). On Site Point we have covered it a few times.

This second example shows how we can develop a highly scalable Express server. It also demonstrates how to migrate a single process server to take advantage of a cluster module with few lines of code.

var cluster = require('cluster');

if(cluster.isMaster) {

var numWorkers = require('os').cpus().length;

console.log('Master cluster setting up ' + numWorkers + ' workers...');

for(var i = 0; i < numWorkers; i++) {

cluster.fork();

}

cluster.on('online', function(worker) {

console.log('Worker ' + worker.process.pid + ' is online');

});

cluster.on('exit', function(worker, code, signal) {

console.log('Worker ' + worker.process.pid + ' died with code: ' + code + ', and signal: ' + signal);

console.log('Starting a new worker');

cluster.fork();

});

} else {

var app = require('express')();

app.all('/\*', function(req, res) {res.send('process ' + process.pid + ' says hello!').end();})

var server = app.listen(8000, function() {

console.log('Process ' + process.pid + ' is listening to all incoming requests');

});

}

The first addition to this example is getting the number of the CPU cores using the Node.js os module. The os module contains a cpus() function, which returns an array of CPU cores. Using this approach, we determine the number of the workers to fork dynamically, based on the server specifications to maximize the utilization.

A second and more important addition is handling a worker’s death. When a worker dies, the cluster module emits an exit event. It can be handled by listening for the event and executing a callback function when it’s emitted. You can do that by writing a statement like cluster.on('exit', callback);. In the callback, we fork a new worker to maintain the intended number of workers. This allows us to keep the application running, even if there are some unhandled exceptions.

In this example, I also set a listener for an online event, which is emitted whenever a worker is forked and ready to receive incoming requests. This can be used for logging or other operations.

## Advanced Operations

While using cluster modules is relatively straightforward, there are other operations you can perform using workers. For instance, you can achieve (almost!) zero down-time in your application using cluster modules. We’ll see how to perform some of these operations in a while.

## Communication Between Master and Workers

Occasionally you may need to send messages from the master to a worker to assign a task or perform other operations. In return, workers may need to inform the master that the task is completed. To listen for messages, an event listener for the message event should be set up in both master and workers:

worker.on('message', function(message) {

console.log(message);

});

The worker object is the reference returned by the fork() method. To listen for messages from the master in a worker:

process.on('message', function(message) {

console.log(message);

});

Messages can be strings or JSON objects. To send a message from the master to a specific worker, you can write a code like the on reported below:

worker.send('hello from the master');

Similarly, to send a message from a worker to the master you can write:

process.send('hello from worker with id: ' + process.pid);

In Node.js, messages are generic and do not have a specific type. Therefore, it is a good practice to send messages as JSON objects with some information about the message type, sender, and the content itself. For example:

worker.send({

type: 'task 1',

from: 'master',

data: {

// the data that you want to transfer

}

});

An important point to note here is that message event callbacks are handled asynchronously. There isn’t a defined order of execution.

## Zero Down-time

One important result that can be achieved using workers is (almost) zero down-time servers. Within the master process, you can terminate and restart the workers one at a time, after you make changes to your application. This allows you to have older version running, while loading the new one.

To be able to restart your application while running, you must keep two points in mind. Firstly, the master process runs the whole time, and only workers are terminated and restarted. Therefore, it’s important to keep your master process short and only in charge of managing workers.

Secondly, you need to notify the master process somehow that it needs to restart workers. There are several methods for doing this, including a user input, or watching the files for changes. The latter is more efficient, but you need to identify files to watch in the master process.

Suggestion for restarting your workers is to try to shut them down safely first; then, if they did not safely terminate, forcing to kill them. You can do the former by sending a shutdown message to the worker as follows:

workers[wid].send({type: 'shutdown', from: 'master'});

And start the safe shutdown in the worker message event handler:

process.on('message', function(message) {

if(message.type === 'shutdown') {

process.exit(0);

}

});

To do this for all the workers, you can use the workers property of the cluster module that keeps a reference to all the running workers. We can also wrap all the tasks in a function in the master process, which can be called whenever we want to restart all the workers.

function restartWorkers() {

var wid, workerIds = [];

for(wid in cluster.workers) {

workerIds.push(wid);

}

workerIds.forEach(function(wid) {

cluster.workers[wid].send({

text: 'shutdown',

from: 'master'

});

setTimeout(function() {

if(cluster.workers[wid]) {

cluster.workers[wid].kill('SIGKILL');

}

}, 5000);

});

};

We can get the ID of all the running workers from the workers object in the cluster module. This object keeps a reference to all the running workers and is dynamically updated when workers are terminated and restarted. First, we store the ID of all the running workers in a workerIds array. This way, we avoid restarting newly forked workers.

Then, we request a safe shutdown from each worker. If after 5 seconds the worker is still running and it still exists in the workers object, we then call the kill function on the worker to force it shutdown.

# Conclusions

Node.js applications can be parallelized using cluster modules to use the system more efficiently. Running multiple processes at the same time can be done using few lines of code and this makes the migration relatively easy, as Node.js handles the hard part.

In addition to performance, you can increase your application’s reliability and uptime by restarting workers while your application is running.

You need to be careful when considering the use of a cluster module in your application. The main recommended use for cluster modules is for web servers. In other cases, you need to study carefully how to distribute tasks between workers, and how to efficiently communicate progress between the workers and the master. Even for web servers, make sure a single Node.js process is a bottleneck (memory or CPU), before making any changes to your application, as you might introduce bugs with your change.