NODE.JS

## Introduction

* Node.js or Node is an open source and cross-platform runtime environment for executing JavaScript code, outside of a browser.
* We often use Node to build back end services also called API (Application Programming Interfaces), this are the services that power client application.
* Node is ideal for building Highly-Scalable, Data- intensive and real – time Apps backend services that power our client application.
* Benefits of Node with respect to other backend application like (ASP.NET, Rails, and Django etc.):-

1. **Great for prototyping and agile development**.
2. **Superfast and highly scalable services**. (It’s used in large production such as PayPal, Uber, and Netflix etc.)
3. **Use JavaScript everywhere**.
4. **Because we can use JavaScript in frontend and backend, code will be cleaner and more consistent codebase**.
5. **Large ecosystem of open-source library’s available**.

## Architecture

Before Node we use JavaScript only to use applications that runs inside our browser, so every browser has JavaScript engine that takes our JavaScript code and converts the code that the computer can understand. For example IE uses **Chakra**, Firefox uses **SpiderMonkey** andChrome uses **v8** because of these varieties of engine, sometimes JavaScript code can behave differently in one browser or another.

In 2009 Ryan Dahl creator of Node came up with a brilliant idea to run **JavaScript code outside the browser, so he embedded v8 engine inside C++ program and call the Node.exe**.

**So Node is a program that includes the v8 JavaScript engine plus some additional module that give us capabilities not available inside browsers, we can work with file system or Network and so on..**

We cannot compare Node with C# or any fundamental programming language because Node is not a Programming language and also Node should compare with framework such as ASP.NET or Rails or Django and so on this are framework to build web application but Node are not framework.

**It’s a runtime environment for executing JavaScript code.**

## How Node works

Node.js is highly scalable this is because of **NON-Blocking or Asynchronous** nature of Node (i.e. a Single Thread is use to handle multiple request).

**But the application build in ASP.NET, Rails are work in Blocking or Synchronous in nature**.

In contrast Node applications are Asynchronous by default. We have single thread to handle all requests and Node is continuously monitoring the Event Queue in back ground. This kind of Architecture is ideal for application using lot of disks or network access; we can serve more clients without the need or throwing more hardware. (I.e. it’s more scalable.).

Node should not be used for CPU-intensive apps like Video encoding or Image manipulation service because Node use only one Thread , so it should only be use application for Data intensive or Real time applications.

## Installing Node

Install from <https://nodejs.org//> and check the version in cmd by typing Node –version.

## Node Module System

### Global Object

There are many global object in JavaScript like console.log (), setTimeout (), clearTimeout (), setInterval (), clearInterval() these all are couple global object in JavaScript. In node we have couple other global which we are going to learn later.

In JavaScript all the global object are called using window.globalobjectName(console.log()) or call it directly.

However in Node we don’t have this window object instead we have another object called global like global.console.log(). But if we declared a variable they are not added in global object , so if we do console.log(global.variablename) we will get undefined, that means the scope of this variable is only scope to particular (.js) file and not available outside of this file. Here comes the Node modular system.

### Modules

If we variable and function as global like window.sayHello(), then same function sayHello() can be present in other js file also with global scope that will leads to ambiguity or overwrite.

So it is best practice we should not declare function variables in global scope. Instead we need module we declare variables or functions as private same as OOPS concept and if we want to use variables outside a module, then we have to explicitly export it outside the module.

And every Node will have one module or main module like app.js. In short in NODE every file is a module and the variables and functions define are scope in that module.

### Creating a Module

Export function or variable to another .js file

Example:

Var url =’htttp://google.com’

function log(message){}

**module.exports.log = log;**

In real world application we will have lots of variables and functions, but we will export subset of these members to outside.

### Load a Module

To load a module we use the require function, this function takes one argument i.e. the path of target module.

**const logger = require(‘./logger.js’) or require(‘./logger’)**

**logger.log(‘Message’);**

In recent version of JavaScript we have the ability to define constant, so as a best result loading the module variable should be declare as constant because we don’t want accidentally overwrite the value of module to something else. So that by **jshint** we can find the errors (npm install -g jshint)

**Example jshint app.js**

Sometimes instead of exporting object from a module, we want to export single function

**Example:**

**module.exports.log = log; // it is empty object**

**module.exports = log; // reset to just a function**

### Path Module

In Node we have many build in module. Some important build in module:

**1) File System**

**2) HTTP**

**3) OS**

**4) Path**

**5) Process**

**6) Query Strings**

**7) Stream**

**Example of Path Module**

//Path Module

const path = require('path');

var pathObj = path.parse(\_\_filename); // \_\_fileName is the argument in modular wrapper function

console.log(pathObj);

### OS Module

Is used to get information about the current operating system.

//OS Module

const os = require('os');

var totalMemory = os.totalmem();

var freeMemory = os.freemem();

console.log('Total Memory: ' + totalMemory);

//Template String which is available in more recent version of javascript(ES6/ ES2015 : ECMAScript 6)

//which the replacement of concatenation (+)

console.log(`Total Memory: ${totalMemory}`);

console.log(`Free Memory: ${freeMemory}` );

**Template String is the replacement of concatenation (+)**

Before NODE we could get the information of OS using JavaScript, because JS runs only inside our browser and we can work only with window or document objects.

### File Module

Is use to work with file system

Examples:

//File Module

const fs = require('fs');

//we have synchronous or asynchronous methos always use asynchronous

//as we know NODE architecture is Asynchronous

//Synchronous example

const filesSyns = fs.readdirSync('./');

console.log(filesSyns);

//Asynchronous example

fs.readdir('./',function(err,files){

if(err){

console.log('Error: ', err);

}

else{

console.log('Results: ',files);

}

});

### Events Module

Event is a signal that indicates something has happened in our application

//Event Module

const EventEmitter = require('events');

// here EventEmitter is a class (is a container for branch of related methods and properties)

//To use EventEmitter First we have to create object of this class

const emitter = new EventEmitter();

//Register a listener

// emitter.on('messageLogged',function(arg){

// console.log('Listener called',arg);

// });

//Use arrow function which is available ES6 instead of function

// arg is the parameters/arguments passed from emit function

emitter.on('messageLogged',(arg) => {

console.log('Listener called',arg);

});

//Raise an Event

//emitter.emit('messageLogged'); // is used to raise and events(pass agrument as name of events)

//A listener is an instance that will be called when the event is raise

// And listener should register before emit methods

//Event Arguments

//emitter.emit('messageLogged',1,'url');

// Here the parameters is little bit confusing, so if we want pass multiple values

//about an event it always best practice to encapsulate those values inside an object

emitter.emit('messageLogged',{id: 1,url: 'http://'});

Event Arguments

Quite often when we raise an event we also send some data about that event

In real world it is quite rear that we will world with event emitter object directly, instead we will create a class that has the capabilities of all the event emitter , then we will use that class in our code.

In app.js

//Event Module

const EventEmitter = require('events');

const emitter = new EventEmitter();

//Use arrow function which is available ES6 instead of function

// arg is the parameters/arguments passed from emit function

emitter.on('messageLogged',(arg) => {

console.log('Listener called',arg);

});

const log = require('./logger');

log('Message');

In logger.js

var url = 'http://mylogger.io/log';

const EventEmitter = require('events');

const emitter = new EventEmitter();

function log(message){

//send an HTTP request

console.log(message);

emitter.emit('messageLogged',{id: 1,url: 'http://'});

}

module.exports = log;

Here we only got the message after running the app.js, but the event listener was not called because here we are working with two different event emitter.

That’s why we have discussed that we will rarely work directly on event emitter instead we will create a class that has all the capabilities of this event emitter but it has additional capabilities.

If we want to raise an event in your application to signal that something has happened, we need to:

1. Create a class that extents EventEmitter with this that class will have all the functionality defined in EventEmitter or we can add additional functionality and inside that whether we want to raise an event we emit

In app.js

//Event Module

const EventEmitter = require('events');

//const emitter = new EventEmitter();

// we no longer this event emitter here we want to directly with the below logger object

//Register an insterner

// emitter.on('messageLogged',(arg) => {

// console.log('Listener called',arg);

// });

//Remove the register insterner after the creating the objects as below

const Logger = require('./logger');

const logger = new Logger();

logger.on('messageLogged',(arg) => {

console.log('Listener called',arg);

});

logger.log('Message');

In logger.js

var url = 'http://mylogger.io/log';

const EventEmitter = require('events');

//const emitter = new EventEmitter(); // no longer need this actual object

class Logger extends EventEmitter{ // it has all the functionalities that define in EventEmitter

log(message){

//send an HTTP request

console.log(message);

//Raise an event

this.emit('messageLogged',{id: 1,url: 'http://'});

// in this class we can directly emit or raise events

}

}

module.exports = Logger;

### Http Module

This Module is used for creating networking application. For example we can create a web server that listens for Http Request on a given port (with this we can easily create a backend servers for our client application like web application that we build React or Angular or Mobile application running on mobile device)

//HTTP Module

const http = require('http');

//Creating web server and assigning to object

//This server is an event emitter(it has all the capabilities of ever emitter)

//in this instead of working with socket, we can actually work

//with actual request or response object

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

if(req.url === '/'){

res.write('Hello world');

res.end();

}

// In real world application we are not writing route like this

//Instead we are using framework to handle various routes

//Internally Express framework is built on top HTTP Module on Node

if(req.url === '/api/courses'){

res.write(JSON.stringify([1, 2, 3]));

res.end();

}

});

//so every time we create a server it raises an event

//so use on method to handle that event

//In real world application we are not using like this

// server.on('connection',(socket)=> {

// console.log('new connection');

// });

server.listen(3000);

console.log('Listening on port 3000...');

## Node Package Manager

### Introduction

NPM is a command line tool is used to registry third party tool to add to our node application.

Install NPM (particular version to our project): “**npm i –g npm@5.5.1”**

In mac **“sudo npm i –g npm@5.5.1”**

### Package.json

Package.json is basically a json file that includes some basic information of our application like (name, version, author, dependency etc.).It basically bunch of Meta data about our application and all node application by standard should have this package.json file.

Adding package.json from NPM “**npm init” (it will ask some questions) or “npm init --yes”(to directly create ). Before adding any packages to our application we need to create a package.json file**

### Installing a Node Package

Let install a package underscore (**npm i underscore** or **npm i underscore --save**) when we run this in package.json under dependencies section underscore package name with version will be added and will download the underscore package under the folder node\_modules

### Using a package

How to use the package underscore in our project.

//Load the underscore package

var \_ = require('underscore');

//How require functions looks the code

//1.first it assume this module is a core module

//2.But in Node we don't have core module with underscore

//so the require function may be this underscore package

// is a file or folder in this project(if it is './underscore')

// 3.require moves to third step it assume package we will present

// inside a node\_modules folder

// use the module

var result = \_.contains([1, 2, 3],2);

console.log(result);

### Package Dependencies

If we try to install package mongoose, it download mongoose with other dependencies package automatically in node\_modules. But suppose out of dependency packages one package is use in our project of other functionally

If the same version is used in both the places i.e. locally and mongoose, then same package with be present in node\_modules folder otherwise another package version will download outside the node\_modules folder.

### NPM Packages and Source Control

In Node project we will have many dependencies package, but dependencies name with version are present in package.json file, then it is well enough to run the project with this dependencies.

And we can ignore the folder from GIT by create a file with .gitignore extension and then mention the folder.

### Semantic Versioning

"underscore": "^1.9.1"

In semantic versioning we have three numbers:

1. First Number (1) – we called as major version.
2. Second Number (19) – minor version.
3. Third Number (1) – Patch version (which is used for bug fixes).

So when they fix a bug, they will increment the third number by 1 (**1.9.1 to 1.9.2**)

Minor version is change, for adding new features that don’t break the existing API (**1.9.1 to 1.20.0**).

Major version is change which the existing functionality breaks for this package (**1.9.1 to 2.0.0**).

**Now comes to caret (^) symbol tells us that we have interested in any version of underscore as long as major version is constant. (^1.9.1 = 1.x).**

**In some real world application instead of ^ we may see ~ this means major and minor version is constant. (~1.9.1 = 1.9x)**

**Some times when new patch release or bug fix happens it can introduce some other bug, so in that case we have to specially mention the version (1.9.1)**

### Listing the installed packages

If we want to see all the list of installed dependencies under exact version (**npm list**) command

And (**npm list --depth=0**) to exclude the dependency packages

### Viewing Registry Info for a package

To view the package registry (**npm view mongoose**) and

To view only the dependencies

(**npm view mongoose dependencies**)

And if we want see the package version release so far (**npm view mongoose versions**)

### Installing a specific version of a package

Command is (**npm i mongoose@2.1.1**)

### Updating Local Packages

In the real world as we build our application, their might be newer version of the dependencies that we installed so run the command (**npm outdated**) and update to wanted version by (**npm update**).

If you want update the dependencies to very latest version, we need to install this package

(**npm i –g npm-check-updates**) in mac (**sudo** **npm i –g npm-check-updates)**, then run

(**npm-check-updates or ncu**) and install with (**npm i**)

### DevDependencies

Some dependencies we used only on development like ( tools we used for running unit testing, tools for static analysis of our code, tools for bundling of JavaScript and so on) and these dependencies should not along with production deployment

(**npm i jshint --save-dev**) we have install with –save-dev and package.json it will show in devdependencies , this tells Node this is development dependencies and it should not go with production code.

### Uninstalling a Package

(**npm un jshint**) command use to uninstalled a package.

### Working with Global Packages

(npm install –g npm) it will upgrade to latest version and if you want to installed particular version of NPM (npm i –g [npm@5.1.1](mailto:npm@5.1.1))

(npm –g outdated) to check all the global packages installed in this system.

(**npm un –g “nameofpackage”**) for uninstalled the global package.

### Publishing a Package

1. First create a folder with new package name
2. Create a package.json file (**npm init --yes**)
3. Create a file index.js because i.e. the entry point to our package.
4. Add a user to npm (**npm adduser**) or if already user is created then (**npm login**)
5. Publish the package with (**npm publish**) it will give some permission issue if same name present is already present
6. Now you can installed in our project

### Updating a published package

(**npm version major**) or (**npm version minor**) or (**npm version patch**)

## Building RESTful Services with Express

### RESTful Services

REST stands for Representational State Transfer is basically a conventional for building HTTP services.

For Example an endpoint <http://vidly.com/api/customer> so the client can send a http requests to this endpoint to talk to our service.

**First** the endpoint should start with **http** or **https**, **second** the **domain name**, **third** **api** is not compulsory but lot of companies flow this convention to expose the RESTful services, **fourth** **customers** which refers the collection of customers in our application (in REST world we refer this fourth part as a resource, we can expose our resources such as customers, movies, retails and various endpoints.

All the operations of customers such as create, update, delete, get will be done by sending an http requests to this endpoints

HTTP Methods:

1. GET – Getting data

Eg: GET Request

1. **/api/customers** services will send repose of array of customers object

**[{id: 1, name:’ ‘}, {id: 2, name:’ ‘} … ]**

1. Getting single customer **/api/customers/1** we should include the id of the customer with request and server will response customer object like **{id: 1, name: ‘ ‘}**
2. POST – Creating data

**/api/customers**, here we are working with collections of customers and **we should include customer object in the body of request**.

1. PUT – Updating data

Eg: we should send an http put request to this endpoint

**/api/customers/1** and **also we should include customers object in the body of the request**

1. DELETE – Deleting data

We should send an http delete request to this endpoint

**/api/customers/1** and here **we should not include customers object in the body of the request.**

### Introducing Express

Express is very popular framework for building web application and web server on top of Node, it also very fast, light weight and perfectly documented.

(**npm i express**) for installing express js in project after creating package.json file (**npm init –yes**).

### Building Your First Web Server

1. Create a js fille index.js or app.js
2. Load the express module

// load the express module, this returns a function express

const express = require('express');

// Now call the express function and assign to constant app

const app = express();

//this get method takes two arguments

//1.url or path

//2. call back function this will call when http get is called

// (with two argument request and response)

// Here we are not using any if else

// statement for route

// instead we used get which is provided by Express

// so as the route got expand, we can use some other js file to specify routes

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

res.send('Hello World');

});

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

res.send([1, 2, 3]);

});

app.listen(3000, () =>console.log('Listening on port 3000...'));

### Nodemon

(**npm i –g nodemon**) is used to install nodemon is used for Node monitoring instead of cancelling the process of node till will automatically starts . And use with nodemon index.js

### Environment Variables

The ports (3000) which we have used for listening purpose need not to be hardcoded, instead we can use environment variable.

//PORT is an environment variable

// an environment variable basically a variable that is part

// of an environment in which the process runs, it values is set

// outside the application.

const port = **process.env.PORT** || 3000;

app.listen(port, () =>console.log(`Listening on port ${port}...`));

In mac we have use export and in windows set (**set PORT=5000**) in terminal.

### Route Parameter

We use **route parameter for essential or required values** and use **query string parameters for anything that is optional** (query string parameter is used after ? mark in the url)

// id is the name of the parameter we can use anything

// req.params.id is used to read the value of route parameter

app.get('/api/courses/:id',(req,res)=>{

res.send(req.params.id);

});

// also it can have multiple parameter

app.get('/api/posts/:year/:month',(req,res)=>{

res.send(req.params);

});

// for reading query string parameter we use req.query

//http://localhost:500/api/posts/1?sortBy=ID

app.get('/api/posts/:id',(req,res)=>{

res.send(req.query);

});

### Handling HTTP Get Request

http://localhost:500/api/courses/1

const express = require('express');

const app = express();

const courses = [

{id:1, name: 'courses1'},

{id:2, name: 'courses2'},

{id:3, name: 'courses3'}

];

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

res.send('Hello World!!');

});

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

res.send(courses);

});

// id is the name of the parameter we can use anything

// req.params.id is used to read the value of paramter

app.get('/api/courses/:id',(req,res)=>{

// find is a method that is available in every array of javascript

// as a arguments we pass a function, this function will be use to find

// the matches of the given criteria (here we use arrow function)

// always assign values to let or const instead of var

// we use let if your assigned values can be change later

const course = courses.find(c => c.id == parseInt(req.params.id));

if(!course)

{s

// if object not found we send 404 and optionally we can message also

res.status(404).send('The course with the given Id was not found');

}

res.send(course);

});

const port = process.env.PORT || 3000;

app.listen(port, () =>console.log(`Listening on port ${port}...`));

### Handling HTTP Post Request

const express = require('express');

const app = express();

app.use(express.json());

const courses = [

{id:1, name: 'courses1'},

{id:2, name: 'courses2'},

{id:3, name: 'courses3'}

];

//Here post to the collection of courses

app.post('/api/courses', (req,res) => {

// in this route handle we need to read the course object that should be

// in the body of request, use this properties to create a new course object

// then add the course object to the courses array

// here we are assign new course but real time it will from database

const course = {

id: courses.length + 1,

// in order to read value from body to work,

// we need to enable parsing of JSON object in the body of the request(see at top)

// by default it is not enable in Express(app.use(express.json());)

name: req.body.name

};

courses.push(course);

// finally by convention when we post an object to a server

// server creates a new resource, we should return that object in the body of response

res.send(course);

});

const port = process.env.PORT || 3000;

app.listen(port, () =>console.log(`Listening on port ${port}...`));

HTTP Request can be tested by PostMan, fiddler etc

### Input Validation

As security we should not trust what the client sent, always validate the input.

For complex object validation we need JOI (npm i joi)

// here Joi is a class so we use pascal naming convention

// best partice to put all the dependencies at top

// so we can say this module is dependent on two joi and express

const Joi = require('joi');

const express = require('express');

const app = express();

app.use(express.json());

const courses = [

{id:1, name: 'courses1'},

{id:2, name: 'courses2'},

{id:3, name: 'courses3'}

];

//Here post to the collection of courses

app.post('/api/courses', (req,res) => {

// here we are using Joi which validate the inputs and return proper message

const schema = {

name: Joi.string().min(3).required()

};

const result = Joi.validate(req.body,schema);

// Here we are doing the validation

// however if the object is complex, then we should npm joi

// if(! req.body.name || req.body.name.length <3 ){

// //Bad Request

// //How

// res.status(400).send('Name is required and should be minimum 3 character');

// return;

// }

// here if result returns error then send the message for first element using details[0]

if(result.error)

return res.status(400).send(result.error.details[0].message);

const course = {

id: courses.length + 1,

name: req.body.name

};

courses.push(course);

res.send(course);

});

const port = process.env.PORT || 3000;

app.listen(port, () =>console.log(`Listening on port ${port}...`));

### Handling HTTP PUT request

// here Joi is a class so we use pascal naming convention

// best partice to put all the dependencies at top

// so we can say this module is dependent on two joi and express

const Joi = require('joi');

const express = require('express');

const app = express();

app.use(express.json());

const courses = [

{id:1, name: 'courses1'},

{id:2, name: 'courses2'},

{id:3, name: 'courses3'}

];

app.put('/api/courses/:id',(req,res) =>{

//Look up the course

//If not existing, return 404 to client

const course = courses.find(c => c.id === parseInt(req.params.id))

if(!course) return res.status(404).send

//Validate

//If invalid, return 400 - Bad request

// this part is duplicated in two route handler

// so we will put this part in different function and will be called here

// const schema = {

// name: Joi.string().min(3).required()

// };

// const result = Joi.validate(req.body, schema);

// const result = validateCouse(req.body);

//here we can use object desctructor in javascript

// instead of result.error repeat

const { error } = validateCouse(req.body);

if(error){

res.status(400).send(error.details[0].message);

}

//Update the course

course.name = req.body.name;

//Return updated course to client

res.send(course);

});

function validateCouse(course){

const schema = {

name: Joi.string().min(3).required()

};

return Joi.validate(course, schema);

}

const port = process.env.PORT || 3000;

app.listen(port, () =>console.log(`Listening on port ${port}...`));

### Handling HTTP Delete Request

// here Joi is a class so we use pascal naming convention

// best partice to put all the dependencies at top

// so we can say this module is dependent on two joi and express

const Joi = require('joi');

const express = require('express');

const app = express();

app.use(express.json());

const courses = [

{id:1, name: 'courses1'},

{id:2, name: 'courses2'},

{id:3, name: 'courses3'}

];

app.delete('/api/courses/:id',(req,res) =>{

//Look up the course

// If not exists , return 404

const course = courses.find( c => c.id === parseInt(req.params.id));

if(!course) return res.status(404).send('The course with the Id not found');

//Delete the course

const index = courses.indexOf(course);

courses.splice(index,1);

//return the same course

res.send(course);

});

const port = process.env.PORT || 3000;

app.listen(port, () =>console.log(`Listening on port ${port}...`));

## Express – Advanced Topics

### Middleware

A Middleware function is basically a function that takes a request object and either returns the response to the client or passes control to another middleware function.

On example we saw is the route handler function, because it terminates the request response cycle by sending response to clients.

So every request that we get under the server will go through a middleware function.

### Custom Middleware

Custom Middleware is used for doing authentication, authorization, logging etc.

All the custom middleware should be present in different module/js file.

Function log which will be used in middle present in logger.js file

function log (req,res,next)

{

console.log('logging...');

// to pass control in next middleware function

// if we don't do this as we are not terminating the req res cycle it leads to hanging

next();

}

module.exports = log;

In index.js load the log module

// here Joi is a class so we use pascal naming convention

// best partice to put all the dependencies at top

// so we can say this module is dependent on two joi and express

const Joi = require('joi');

// import the logger module

const logger = require('./logger');

const express = require('express');

const app = express();

app.use(express.json());

// Custom middle ware

// next is a reference of next middleware function in the pipeline

app.use(logger);

### Built-In Middleware

We have many Built-In Middleware function in express.js like we use previously.

app.use(express.json());

We have another similar middleware function i.e. URL encoded

// this middleware function parses the income request with the URL encoded payroll

app.use(express.urlencoded({ extended:true }));

And finally we have last built middleware function, static and we use them to serve static files

// Parameter name public is folder, where we placed all the

// static files like css, images etc

app.use(express.static('public'))

### Third-party Middleware

We can get all the third-party middleware in express

<http://expressjs.com/en/resources/middleware.html>

And that does not means we can use all the third party middleware in our project, which will slow down the performance of your application. If you don’t a functionality that comes with a middle function don’t use it (because it is going to slow down the request processing).

1. Helmet (Helps secure your apps by setting various HTTP Headers)

**npm i helmet**

const helmet = require('helmet');

app.use(helmet());

1. Morgan (HTTP request logger.)

**npm i morgan**

const morgan = require('morgan');

app.use(morgan('tiny'));

### Environments

In more complex application we need what environment our application is running, because we may require to enable or disable certain features based on current environment

//Two ways we can get the environment

// First one will return undefined if NOD\_ENV is not defined

console.log(`NOD\_ENV: ${process.env.NOD\_ENV}`);

// Second will return by default development

console.log(`app: ${app.get('env')}`);

Setting environment variable to production: **set NODE\_ENV=production**

**Here checking the environment and enabling Morgan only for development environment**

if(app.get('env') === 'development'){

app.use(morgan('tiny'));

console.log('Morgan enabled');

}

### Configuration

There are various Node packages available for managing configuration, most popular on is **rc** (<https://www.npmjs.com/package/rc>) this package is good but it gives many issue.

We have one more package **config** which will be using for configuration install it (**npm i config**)

Step to do configuration setting in NODE project

1. Create a folder, inside the file create a default.json file

{

"name": "My Express App"

}

1. Inside the same folder create a json file development.json which will be specify to development environment

{

"name": "My Express App - Development",

"mail":{

"host": "dev-mail-server"

}

}

1. Similarly we can json file for production (production.json)

{

"name": "My Express App - Production",

"mail":{

"host": "prod-mail-server"

}

}

1. Load the config module in our module (index.js)

const config = require('config');

//Configuration

console.log(`Application Name: ${config.get('name')}`);

console.log(`Mail Server: ${config.get('mail.host')}`);

**However we should store application secret in configuration file (like password of database, mail server etc) because we will store this code in source control repository which will be visible to everyone.**

**We can drill with this sensitive data by storing them in environment variables**

// app\_(can use application name) we have use so that it should not clash with another environment variable

**set app\_password=1234**

* Create a json file in same config folder name as **custom-environment-variables.json,** in this file we will map configuration setting with environment variable only.

{

"mail":{

**"password": "app\_password"**

}

}

* Now read the value in your module

console.log(`Mail Password: ${config.get('mail.password')}`);

### Debugging

In JavaScript we mostly use console.log() for debugging purpose but this approach is very tedious.

The better way for log message for the purpose of debugging is to use the **debug package** in node

In your load the module and replace console.log()

// this require function returns a function

// we call this function and give an argument(arbitary namespace)

const startupDebugger = require('debug')('app:startup');

// it will be use for db debugging

const dbDebugger = require('debug')('app:db');

if(app.get('env') === 'development'){

app.use(morgan('tiny'));

startupDebugger('Morgan enabled');

}

//Db work..

dbDebugger('Connected to the database');

In console we can set the environment to specify what kind of debugger information we want to see in console

**set DEBUG=app:startup** ( that means we can see only the debugging message of startup only)

**set DEBUG=** (here no log message will show as we set the DEBUG to empty)

**set DEBUG=app:startup,app:db** ( with this we can see debugging message of both the namespaces)

**set DEBUG=app:\*** ( to show all the debugging namespace with app namespace

we can also set and execute project at same time **DEBUG=app:db nodemon index.js**

### Templating Engine

In all the endpoint implemented so far we return JSON objects in the response, something we have to return HTML marker to the client so we use Templating Engine.

Several Templating Engine present in Express most popular are :

1. PUG
2. Mustache
3. EJS

Each templating engine has different syntax for generating dynamic HTML returning to client.

Here we will use PUG

* First install PUG ( npm i pug)
* Set the view engine in index.js

//set the view engine for our application

//name of the property we need to set view engine

app.set('view engine','pug');

app.set('views','./views');

* Create a folder name view and create a file with index.pug , write the code

Here title and message is parameter which h we will be set in index.js while rendering

html

head

title= title

body

h1= message

* In index.js

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

// now we will this html marker and return to client

// instead of send we use render

// First argument is name of the view here it is index

//second argument we pass an object , where values will the parameter defined in file

res.render('index', { title: 'My Express App', message: 'Hello'});

});

### Database Integration

This section will be covered later, when we will explain MongoDB

<http://expressjs.com/en/guide/database-integration.html>

### Authentication

Authentication is out of scope in express. Later we will discuss about authentication and authorization for securing from endpoints.

### Structuring Express Application