
Enzigma Software Pvt. Ltd.

Introduction

Web applications were written in a client/server model where the client would demand resources from the server and the server would respond with the resources. The server only responded when the client requested and would close the connection after each response.

This pattern is efficient because every request to the server takes time and resources(memory, CPU etc). So, it's wiser to close a connection after serving the requested asset so the server could respond to other requests too. So how do servers like these respond to millions of requests coming in at the same time? If you asked this question, you understand that it wouldn't be nice if requests to servers would be delayed till all other requests were responded to.

Imagine visiting Facebook and you were told to wait for 5 minutes, for thousands of people requesting before you. There has to be a way to run thousands or at least hundred of requests at once. Good news!! There's a thing as Threads. Threads are a way for systems to run multiple operations concurrently. So every request would open a new thread, every thread had all it required to run the same code to completion.

Each thread is a new employee and the browsers, well, hungry people. I'm sure you can get the point now. But this system has a downside. It would get to a point where there's a lot of requests and starting up new threads would consume a whole lot of memory and system resources. Just like in our example, employing more and more people to serve food would cost more money and space.

But it's also a good thing that immediately the server responds to the browser, the connection is closed and all resources (memory etc) are retrieved.

A benefit of this system is that it excels at CPU intensive applications. CPU intensive operations require a lot of logic and take more time to compute. Since every new request is handled by a new thread, it takes serious work off the main thread therefore making the system faster and able to perform important computations. It's quite efficient because each operation executes code off the main thread. But could things get better?

Here comes NodeJs.

Index

1. Training Guidelines	6
Which Naming Convention is to be followed & Why?	6
What are the Best Practises & Everything needed?	6
Why choose Version Control?	7
2.Estimated Time of Training	8
3. Module 1 - Introduction	9
Prerequisites	9
What is Node.js?	9
What is V8 engine?	10
Why Node.js?	10
Difference between .js and other server side platforms	10
Components of Node.js	11
What is the difference between save and save-dev?	14
What is the difference between dependencies and devDependencies ?	15
Footnotes	15
4. Module 2 - Typescript	16
TypeScript	16
5. Module 3 - Node Module	17
Node Package Manager (NPM) ?	17
Create Node.js application with TypeScript	17
Folder Structure of Node.js App	18
How to Loading core module	19
Loading core module	19
Why not to use Require to load module?	19
Why to use import to load module?	20

How to create local module?	20
How to Load local module?	21
How to load third party modules?	22
Summary	22
Footnotes	22
Assignments - Timeframe - 1 day	23
6. Module 4 - Asynchronous Node.js	25
Blocking I/O and Non-Blocking process	25
Synchronous Web Application Processing	25
Asynchronous Web Application Processing	25
What is the Event Loop?	25
Event-driven Programming	25
File System Module	26
Callback Method	27
How Callback Functions Work?	28
Basic Principles when Implementing Callback Functions	28
Why not to use Callback function?	30
What is Promise?	31
Async and Await	33
Sending Requests to External API	40
Summary	41
Footnotes	41
Assignments - Timeframe -1 day	41
7. Module 5 - Testing Application	43
What is Test Driven Development(TDD) And Behavioral Driven Development(BDD)	43
Testing Frameworks In Js (use mocha and chai)	43
Mocha And Basic Testing	43
Assertion Libraries	45
Testing Asynchronous Code	45
Using Promise and Async/Await	46
Testing API's	46
Summary	49
Footnotes	49
Assignments - Timeframe - 1/2 day	49

8. Module 6 - Routing	50
How to Serve request with Routing?	50
How to create Server Without Express using Http?	51
What is Express Framework?	52
What is Middleware?	55
How to manage Static files?	56
What View Engine(Template) Node.Js supports ?	56
Submitting Data using Body-parser	58
Summary	59
Assignments - Timeframe -1 day	61
9. Module 7 - MongoDB	62
What is MongoDB?	62
Features of MongoDB?	61
Aggregation Pipeline	61
Installing MongoDB and Compass.	64
How do I start MongoDB from windows?	64
MongoDB have primary key?	67
Summary	67
Footnotes	67
Assignments - Timeframe -1 day	67
10. Module 8 - Rest Api	70
What is Mongoose	70
Mongoose Schema vs Model	71
What is MongoDB schema validator ?	71
Rest API's in details	71
Defining the Schema in Node.js	71
Defining Model in Node.js	72
Rest Application Using Mongoose,Express	73
Testing Rest api in Node.Js	77
Summary	77
Footnotes	77
Assignments - Timeframe - 1 days	78
11. Module 9 - Authentication and Authorization	80

Commented [1]: newly added

Authentication	80
Authorization	80
Why to Authorize User?	80
JWT And Hashing	81
When Should i use JWT?	82
Summary	82
Footnote	82
Assignments - Timeframe - 1 day	83
12. Mini project - Quiz App	86
Summary	86
Steps	86
API Description	88
Scenarios to check (5h)	91

1. Training Guidelines

Which Naming Convention is to be followed & Why?

Naming conventions are fundamental as it helps to encourage consistency overall, and makes it more easily understandable & readable. The Naming Convention that we would be as follows...

- *camelCase* is used for variable & function names (*var lastName*)
- *camelCase* is used for filename (modules,
- Variable names Always start with *letter* (*var firstName*)
- *Spaces* around operator (*var x = y + z*)
- Encourage Code Readability & usage of Indentation
- Global variables & Constants should be written in UPPERCASE
- Add *//comments* where needed.
- Choose Meaningful names while naming file names & folders.
- Use lowercase for file and folders.
- Use dash(-) or underscore separator.

What are the Best Practises & Everything needed?

Best Practises ensure along with timely delivery ensure quality end-product making the code more readable, easily understand & ensure better maintenance.

- Declarations on the top
 - Single point to look on-to.
 - Reduced possibility of unwanted re-declarations.
- Follow Naming Convention mentioned above
 - Helps to have consistency overall.
 - Improves Readability.
- Encourage the use of Version Control.
 - Helps in reverting back to changes in-case of accidental changes.
- Implement the Folder Structure systematically.
- Follow the Modular approach.
 - Avoid having one single file having more than 600 lines of code.
 - Created Separate Folders.
- Follow throughout the whole application a Consistent Pattern.

-
- File/Folders Naming.
 - Modules Naming.

Why choose Version Control?

In this whole course module we would be heavily using version control in order to store,manage our code, the platform which we would be using for this would be [@Gitlab](#). Gitlab is no more different than github but instead offers us more features.

Prerequisites:

Need to ensure that you have a gitlab account created, after doing so.

Create a gitlab repo which has access to required members.

While creating a gitlab repo, ensure Naming Convention

- "node-"(followed by module name)

EQ: Node_Module name

Make sure all the changes have been committed to the repo.

2.Estimated Time of Training

Module	Module Timeframe
Module 1: Introduction to Node.Js	1 - day
Module 2: Typescript	2 - day
Module 3: Node Modules	1 - day
Module 4: Asynchronous Nodejs	1 - day
Module 5: Testing Application	1 - day
Module 6: Routing	1- day
Module 7: Mongodb	2- day
Module 8: Rest Api	1- day
Module 9: Authentication and Authorization	1- day
Mini Project	4 - days

Grand Total = 15 days

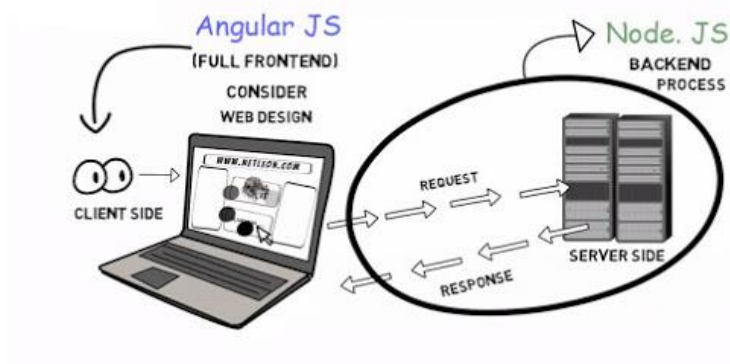
3. Module 1 - Introduction

Prerequisites

- Basic knowledge of JavaScript
- Basic HTML/CSS Knowledge
- Assume you know how to install node

What is Node.js?

Node.js is an open source framework of a different set of libraries and a run-time environment. It also includes a package management system (NPM) and modules which simplifies the development of server-side web applications and networking applications. Node.js applications are written in JavaScript, and can be run within the Node.js runtime on OS X, Microsoft Windows, and Linux.



What is a V8 engine?

Chrome V8, or simply **V8**, is an open-source JavaScript engine developed by Couchbase, MongoDB and Node.js that are used server-side.

V8 compiles JavaScript directly to native machine code before executing it. The Chromium Project for Google Chrome and Chromium web browsers. The project's creator is Lars Bak. The first version of the V8 engine was released at the same time as the first version of Chrome: September 2, 2008.

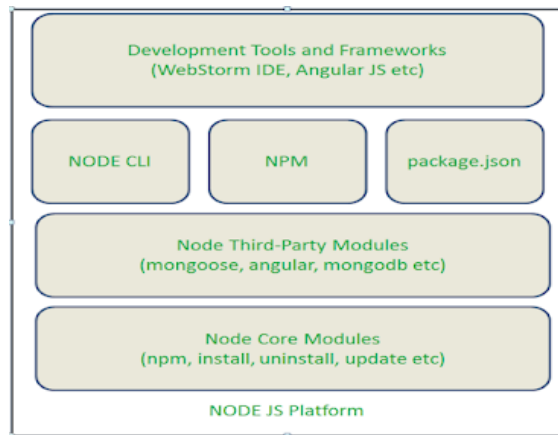
Why Node.js?

- Node.js enables developers to use Javascript on the front- and backend as well.
- Node.js is the perfect tool for developing server-side applications.
- Being an open-source technology
- Asynchronous and Event Driven.
- Very Fast - Being built on Google Chrome's V8 JavaScript Engine, Node.js library is very fast in code execution.
- Node.js uses a single threaded model with event looping.

Difference between .js and other server side platforms

Any Web Application developed without Node JS, typically follows “c” model. Simply we can call this model as Request/Response Model. Client sends request to the server, then server do some processing based on clients request, prepare response and send it back to the client.

Components of Node.js



● Installing Node.js to get started

You need the following two softwares available on any platform.

- Node.js
- Node Package Manager (NPM)
- IDE (Integrated Development Environment)/Visual Studio code or Text Editor

To get the latest Node.js library you can visit the official Node.js website:

<https://nodejs.org/en/download/>.

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

● Node CLI

The **CLI** runs from a terminal. This is how most developers interact with npm.

Ex : **npm install**

Install the dependencies in the local **node_modules** folder.

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

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

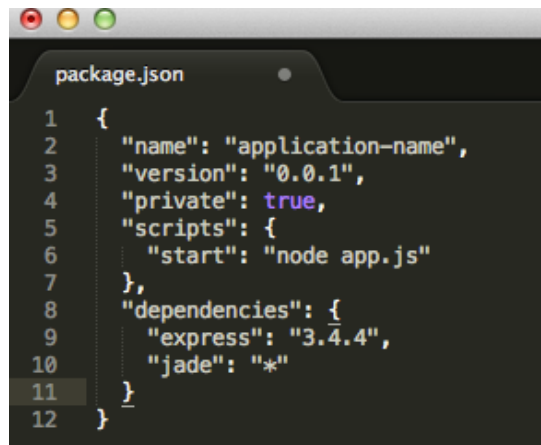
- **NPM**

Node Package Manager (NPM) is a command line tool .It let's you install and uninstall software consist module/libraries, plugins, frameworks and applications. Typically this software is installed to build Node applications.

Ex: **npm install** **npm init**

- **Package.json**

The package.json file is normally located at the root directory of a Node.js project.This file is used to give information to npm that allows it to identify the project as well as handle the project's dependencies(modules). It can also contain other metadata such as a project description, the version of the project in a particular distribution, license information, even configuration data - all of which can be understood to both npm and to the end users of the package.



```
1  {
2    "name": "application-name",
3    "version": "0.0.1",
4    "private": true,
5    "scripts": {
6      "start": "node app.js"
7    },
8    "dependencies": {
9      "express": "3.4.4",
10     "jade": "*"
11   }
12 }
```

Creating "package.json"

Open Command Prompt or in terminal of Visual Studio Code
Navigate to the directory in which you want to create package.json.

Run the following command

npm init

OR

`npm init -y` or `yes`

Second command retrieves or extract information from current directory like Name,version etc.

● **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.

Node.js includes three types of modules:

- **Core Modules** : Node.js has a set of built-in modules which you can use without any further installation
- **Local Modules** : Local modules are modules created locally in your Node.js application by programmer itself. These modules includes functionality in .js, if you are using Typescript to create module includes functionality in .ts file present in separate folder.
- **Third Party Modules** : The third party module can be downloaded by NPM (Node Package Manager). These type of modules are developed by others and we can use that in our project. Some of the best third party module examples are listed as follows: express, gulp, lodash, async, socket.io, mongoose, mongodb, underscore, pm2, bower, q, debug, react, mocha etc.

Install the dependencies in the local **node_modules** folder either locally or globally.

In npm there are two ways to install things:

Locally: If you're installing something that you want to use *in* your program, using `require('whatever')` or `import`, then install it locally, at the root of your project.

Ex: `npm install module_name`

Globally: if you are going to re-use the same library in a bunch of different projects install globally . (later if you want to use in different projects use **npm link** in a future installment.)

Ex: `npm install -g module_name`

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

Ex :: `\Users\username\AppData\Roaming\npm\node_modules`

By default, **npm install** will install all modules listed as **dependencies** or **devDependencies** in **package.json** or you can use **--save** to list in **package.json**.

Ex: `npm install -g module_name --save`

OR

Ex: `npm install -g module_name --save-dev`

What is the difference between save and save-dev?

In case of `npm install -g module_name --save` list all installed modules in **dependencies** in **package.json** as

```
"dependencies": {  
  "@types/express": "^4.16.0",  
  "@types/mocha": "^5.2.5",  
  "@types/supertest": "^2.0.6",  
  "body-parser": "^1.18.3",  
  "express": "^4.16.4",  
  "mongodb": "^3.1.8",  
  "mongoose": "^5.3.3",  
  "superset": "^1.0.1",  
  "ts-node": "^7.0.1",  
  "tsd": "^0.6.5",  
  "typescript": "^3.1.3"  
},
```

In case of `npm install -g module_name --save-dev` list all installed modules in `devDependencies` in `package.json` as

```
"devDependencies": {
  "@types/body-parser": "^1.17.0",
  "@types/mongodb": "^3.1.12",
  "@types/mongoose": "^5.2.19",
  "mocha": "^5.2.0",
  "should": "^13.2.3",
  "supertest": "^3.3.0",
  "tsc": "^1.20150623.0"
}
```

What is the difference between dependencies and devDependencies ?

The difference between these two, is that `devDependencies` are modules which are only required during **development**, while `dependencies` are modules which are also required at **runtime**.

Footnotes

- [Node.js Application Life cycle](#)
- [Difference between Angularjs and Nodejs](#)
- [Introduction to XML and JSON Format](#)
- [Difference between XML and JSON](#)
- [Core Modules](#)

4. Module 2 - Typescript

TypeScript

Typescript is open source, strongly type , object-oriented programming language.It is developed by microsoft.It is used for developing large web based applications which is compiled not interpreted to javascript.Typescript compiler compiles ts code in javascript i.e. why it is superset of javascript.It is used to develop javascript application executed at both client side(angularJs) and server side(Nodejs).

For Resource/Assignment refer to the Document please click [Training Document-Typescript](#)

5. Module 3 - Node Module

Create Node.js application with TypeScript

Step 1) Create folder with name Node_TypeScript

Step 2) Open the folder where to create typescript file followed by .ts extension to start coding.

Step 3) Add Package.json file. Run the following command

```
npm init
```

Step 4) Install typescript in project. Run following command.

```
npm install typescript --save-dev
```

Step 5) Add tsconfig.json file. Run the following command.

```
tsc -init
```

Step 6) Run the following command

```
npm install
```

Step 7) Install core modules in application. Run the following command

You also need to install

```
npm install express
```

Along with

```
npm install @types/express
```

Step 8) Create folder with name modules and add **myfirstModule.ts** file.

Step 9) Compile the module. Run the command. It creates a myFirstModule.js file.

Ex: tsc modules/myfirstModule.ts

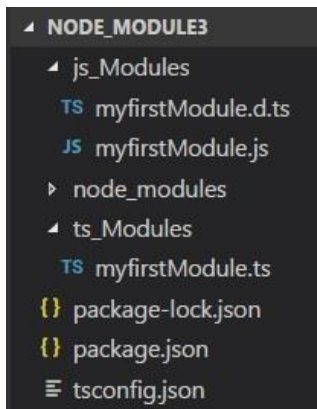
Step 10) To run the .js module, Run the command.

```
node modules/myfirstModule.js
```

To run ts modules, Run the command

```
ts-node modules/myfirstModule.ts
```

Folder Structure of Node.js App



To save all .js files generated after compiling .ts file configure tsconfig.json file as.

```
{
  "compilerOptions": {
    /* Basic Options */
    "target": "es5",
    "module": "commonjs",
    "declaration": true,
    "outDir": "./js_Modules",
    "rootDir": "./ts_Modules",
  }
}
```

Once compile automatically .js files to save in out directory configure in tsconfig.json file do some configurations in package.json file

```
6   "scripts": {
7     "test": "echo \"Error: no test specified\" && exit 1",
8     "build": "tsc"
9   },
```

Run the following command
npm run-script build

How to Loading core module

To load a module in your node application you can just use **"import"** keyword. syntax is given below.

```
import module from 'module_name';
```

There are several ways to reference modules, this depends on what type of module we have installed in the project and we are going to load.

Loading core module

Core modules can be loaded as follows.

```
import * as express from 'express';  
//Imports all exports from module  
OR  
import { express } from 'express';  
//Imports specified methods from module
```

Why not use Require to load modules?

- Load module dynamically which doesn't check if the loaded module is present or not at specific path statically.
- Even though it is not required to be loaded dynamically.
- Loading is done synchronously. If multiple modules are loaded one by one , which hampers performance.
- Require is javascript specific.

Why to use import to load modules?

- which check loaded module is present or not at specific path statically if not throws error as where require doesn't.

```
2
3 let myServer = require('./myServer');
4
5 import * as myServer from './myServer';
6
```

```
2
3 let myServer = require('./myServer');
4 [ts] Cannot find module './myServer'.
5 import * as myServer from './myServer';
6
7
```

- Load only the piece which is required.that can save memory.
- Load module asynchronously. If multiple modules are loaded simultaneously , which performs little better.
- Import is Typescript specific.

How to create a local module?

In our node project add **firstModule.ts** file with functionality may be dependent.

Ex:

```
8 export class FirstClass {  
9  
10     public welcome(): string {  
11         return 'Welcome to First Node.js App';  
12     }  
13 }  
14  
15 export default FirstClass
```

The **module.exports**, **exports**, **export default** is a special object which is included in every ts file in the Node.js application by default. *module* is a variable that represents current module and *exports* is an object that will be exposed as a module. So, whatever you assign to *module.exports* or *exports*, *exports default* will be exposed as a module.

How to Load local module?

To use local modules in your application, you need to load it using **import** keyword in the same way as core module. However, you need to specify the relative path(relative to current file) of typescript file without extension.

```
2 //absolute path  
3 //import { FirstClass } from './Users/varsha.kone/Desktop/NodeJs/ty_ModulesApp/modules/firstmodule'  
4  
5 //relative path of module  
6 import {FirstClass} from '../modules/firstmodule';  
7  
8 let firstClass = new FirstClass();  
9  
10 console.log(firstClass.welcome());
```

```
C:\Users\varsha.kone\Desktop\NodeJs\ty_ModulesApp>tsc ./modules/load.ts
```

```
C:\Users\varsha.kone\Desktop\NodeJs\ty_ModulesApp>node ./modules/load.js  
Welcome to First Node.js App
```

How to load third party modules?

Third party Node.js module can be downloaded using NPM (node package manager) which you can download locally or globally. To download globally we use the following command.

```
npm install <module_name>
```

here we use -g to install package globally. If you want to install locally then use the following command.

```
npm install <module_name> -g
```

In order to access MongoDB database, we need to install MongoDB drivers. To install native mongodb drivers using NPM, open command prompt and write the following command to install MongoDB driver in your application.

```
npm install mongodb
```

Above command will download node package inside node_modules folder and then you can directly use import keyword to load node module.

```
import module from 'module_name';
```

Summary

“Modules are javascript libraries support some functionality can be included in application”.

Footnotes

- [Command line arguments through Process](#)
- [Passing command line argument using Yargs](#)
- [REPL](#)
- [Global Objects](#)
- [Callback functions](#)
- [Arrow functions](#)
- [Debugging Node Application](#)
- [Restarting App with Nodemon](#)
- [How node file executed?](#)

Assignments - Timeframe - 1 day

Assignment 1 : Create a Node Application with name **Node_Modules**.

Assignment 2 : Create folder one for ts module with name **modules**. Create all respective modules in folders only.

Assignment 3 : Create all those given modules in modules folder. [Go through this link](#)

- Create **stringHandler.ts** module
- Define function named **nameHandler()** with string parameter **name** in class **StringClass**
- Pass parameter through command line. **"I am member of Enigma"**. Here you need to import readline module to input values.

```
npm install readline
npm install @types/readline
import * as readline from readline;
```

- Show index number of string Enigma on console.
- Show only **am** on console.
- Convert the string into array of string and return .

Assignment 4 : Create **arrayHandler.ts** module.

- Define class **ArrayClass** with **trainees** as array field.
- Declare array as **trainees**.
- Define function **newTrainees()** with signature :
newTrainees(newJoinser : string) : string[]
- Initialize some items in array.
- Define function **noOfTrainees()** return length of trainees array
noOfTrainees() : number
- Define function **addAtTop()** to push new trainee at top of trainees array :
addAtTop(trainee : string) : string[]
- Define function to add one more name of trainee at end of array .
addTrainee(trainee : string) : any . before adding trainee verify trainee name already exist or not.
if exist return message : Trainee already exist

if not exist add and return list.

- Define method **removeTrainee()** to delete two names of trainee from index 2.
removeTrainee() : String[]
- Define function to sort names of trainees.
sortTrainee() : String[]

Assignment 5 : Create **objectHandler.ts** module.

- Define function **startTraining()** in Objectclass class with following signature.
startTraining(task : string):any
- Declare and export object **training** with properties - **traineeName: string, duration: string , noOfModules: number, moduleList: Array<String> ,trainingStatus(callback :Function).**
- Import training object in objectHandler.ts module.
- Initialize object with values and print on console in **startTraining()**.
e.g: training.traineeName= "Trainee Name";
- Add module name dynamically in moduleList.
- Define the function **trainingStatus** in training object.
- Define function **newModule()** function with signature.
Ex: **newModule(module : string):any**
- Write functionality in **newModule()** as once **noOfModules** and size of **moduleList** meets invoke **trainingStatus** function returns message as **"Training Done Successfully with Modules"** or return moduleList details.

Assignment 6 : Create **scheduler.ts** module.

- Define method as **scheduleTask()** with parameter **task: string** and **callback: any** and export it.
- In main.ts module define method **startTask()** which invokes **scheduleTask()**.
- After two seconds **scheduleTask()** invokes callback method of **startTask()** to show the result of task to be done.

Assignment 7 : Create **main.ts** local module load all given modules and write menu driven program to test all given manipulation .Use readline to read option

Assignment 8 : Commit the code in Git , push new commit on GitLab Immediately change in code.

6. Module 4 - Asynchronous Node.js

Blocking I/O and Non-Blocking process

Blocking methods execute synchronously and non-blocking methods execute asynchronously. Please refer [blocking and non-blocking process](#) for more information.

Synchronous Web Application Processing

Synchronous is when the execution of additional JavaScript in the Node.js process must wait until a non-JavaScript operation(e.g - I/O request etc.) completes. This happens because the event loop is unable to continue running JavaScript while a blocking operation is occurring.

Asynchronous Web Application Processing

Asynchronous is when the execution of JavaScript in the Node.js process will continue to execute further JavaScript code after a non-JavaScript operation. This happens because the event loop is continuing running JavaScript code and it will not wait for completion of non-JavaScript operation.

What is the Event Loop?

The event loop is what allows Node.js to perform non-blocking I/O operations — despite the fact that JavaScript is single-threaded — by offloading operations to the system kernel whenever possible.

Since most modern kernels are multi-threaded, they can handle multiple operations executing in the background. When one of these operations completes, the kernel tells Node.js so that the appropriate callback may be added to the poll queue to eventually be executed.

Event-driven Programming

Event-driven programming is a programming paradigm in which the flow of program execution is determined by events - for example a user action such as a mouse click, key press, or a message from the operating system or another program. An event-driven application is designed to detect events as they occur, and then deal with them using an appropriate event-handling procedure. Javascript is a event driven programming language.

File System Module

Blocking methods execute synchronously and non-blocking methods execute asynchronously. File System Module supports both synchronous and asynchronous functionality to manipulate file.

- Synchronous Example:

```
3 import * as filesystem from 'fs';
4
5 console.log("Synchronous Operation Started at ", new Date().toTimeString());
6
7 let data: any = filesystem.readFileSync("myfirstexample.txt", "utf8");
8
9 console.log(`Result is generated at :${new Date().toTimeString()} \n`, data);
10
11 console.log("Synchronous Operation Ended at ", new Date().toTimeString());
```

```
C:\Users\varsha.kone\Desktop\NodeJs\ty_ModulesApp>node ./modules/fssynch.js
Synchronous Operation Started at 12:36:39 GMT+0530 (India Standard Time)
Result is generated at :12:36:39 GMT+0530 (India Standard Time)
First Example of reading file content Synchronously and asynchronously.
Synchronous Operation Ended at 12:36:39 GMT+0530 (India Standard Time)
```

- Asynchronous Example:

```
24 import * as filesystem from 'fs';
25
26 let start : number =Date.now();
27
28 console.log('\n');
29 console.log("Asynchronous Operation Started at : \t ", new Date().toTimeString());
30 filesystem.readFile("myfirstexample.txt", "utf8", function (error:
31 any, data: any) {
32
33     if (error) {
34
35         console.log("Can't process file because :", error);
36     }
37     else {
38         console.log('-----\n');
39         console.log('Result is generated at :          ${new Date().toTimeString()} \n \n', data);
40         console.log('\n');
41         console.log('Result is:          \n', data);
42         console.log('\n');
43         console.log('Time requires to finish process:  ${Math.floor(Date.now()-start)/1000} seconds \n');
44         console.log('\n');
45         console.log('-----\n');
46     }
47
48 });
49 console.log('\n');
50 console.log("Asynchronous Operation Ended at : \t ", new Date().toTimeString());
```

```
Asynchronous Operation Started at :      14:01:44 GMT+0530 (India Standard Time)

Asynchronous Operation Ended at :        14:01:44 GMT+0530 (India Standard Time)
-----

Result is generated at :                  14:01:44 GMT+0530 (India Standard Time)

First Example of reading file content Synchronously and asynchronously.

Result is:
First Example of reading file content Synchronously and asynchronously.

Time requires to finish process:          0.017 seconds
```

Callback Method

A callback function, also known as a higher-order function, is a function that is passed to another function (let's call this other function "otherFunction") as a parameter, and the callback function is called (or executed) inside the otherFunction.

How Callback Functions Work?

When we pass a callback function as an argument to another function, we are only passing the function definition. We are not executing the function in the parameter. And since the containing function has the callback function in its parameter as a function definition, it can execute the callback anytime.

Note that the callback function is not executed immediately. It is “called back” (hence the name) at some specified point inside the containing function’s body

Basic Principles when Implementing Callback Functions

we used anonymous functions that were defined in the parameter of the containing function. That is one of the common patterns for using callback functions. Another popular pattern is to declare a named function and pass the name of that function to the parameter.

Ex: In **callback.ts** file call callback method

```
24 public additionAsynch(number1:any,number2:any,callback:Function)
25 {
26     let result = 0;
27     setTimeout(function(){
28
29         if(typeof number1 === "number" && typeof number2 === "number")
30         {
31             result = number1 + number2;
32             callback( undefined, result);
33         }
34         else{
35             callback("argument value should be of type number",undefined);
36         }
37
38     },10000)
39 }
40
41 }
42
43
44 export default { CallbackClass };
```

Ex: In **main.ts** file define callback method

```
1
2 import {CallbackClass} from '../src/callback';
3
4 let callbackClass=new CallbackClass();
5
6 callbackClass.additionAsynch('10', '12', function (error : any, result : any) {
7
8     console.log("Addition is : ", result);
9     if (result) {
10
11         updateAddition(result);
12     }
13     else {
14         someError(error);
15     }
16 }
17 })
18
19 function updateAddition(result: any) {
20     console.log("Addition is : ", result );
21 }
22 function someError(error: any) {
23     console.log("Error is : ", error);
24 }
```

When we run main.ts file result on failure :

```
C:\Users\varsha.kone\Desktop\NodeJs\callback>node ./src/main.js
Error is : argument value should be of type number
```

When we run main.ts file result on success :

```
C:\Users\varsha.kone\Desktop\NodeJs\callback>node ./src/main.js
Addition is : 22
```

Why not to use Callback function?

As given in example The second code block **updateAddition** is gets executed when the callback function **additionAsynch** is actually called. Second callback just update value by 2 . following are reasons why not to use callback function.

- updation should take place when addition result is generated.

-
- There is no concept of a return value when working with normal callbacks in Node.js. Because of the return value, we have more control of how the callback function can be defined.
 - there may be an instance where you would need to nest multiple callback functions together which is very messy task .
 - promise is an enhancement to callbacks that looks towards solve these problems.

What is Promise?

A **Promise** is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers to an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of the final value, the asynchronous method returns a *promise* for the value at some point in the future.

In callback.ts file. Before using promise install it .Run the command
npm install promise OR npm install @types/promise

```
1 import * as Promise from 'promise';
2
3 export class CallbackClass{
4
5     //using promise
6
7     public divisionAsynch(number1: any, number2: any) {
8
9
10        return new Promise(function (onResolve, onReject) {
11
12            setTimeout(function () {
13
14                if (typeof number1 === 'number' && typeof number2 === 'number' && number2 > 0)
15                    onResolve(number1 / number2);
16                else {
17                    onReject("number1 and number 2 should be number type and non-negative");
18                }
19            }, 10000)
20        })
21    }
22
23 }
```

Ex: In main.ts file

```
1
2 import {CallbackClass} from '../src/callback';
3
4 let callbackClass=new CallbackClass();
5
6 // using promise
7 callbackClass.divisionAsynch(12, 0).then(function (result) {
8     console.log("Division is :", result);
9 },
10 function (error) {
11     console.log("Error is :", error);
12 }
13
14 }
```

When we run main.ts file Result on success state is:

```
C:\Users\varsha.kone\Desktop\NodeJs\callback>node ./src/main.js
Division is : 1.2
```

Result on failure state is:


```
C:\Users\varsha.kone\Desktop\NodeJs\callback>node ./src/main.js
Error is : number1 and number 2 should be number type and non-negative
```

In this example to handle result and error we don't need to call nested functions. when callback is called from **divisionAsync** return promise. Promise returns either success or failure result by calling onResolve and onReject functions, to handle we call **then** nested function of promise which reduces messy task and code.

Async and Await

There may be an instance where you would need to nest multiple callback functions together which is a very messy task. In nesting of function the callback functions, which would fire (or run) after the first function (to which the callbacks were assigned) run is completed. As given in example.

In callback.ts file

```
42 public additionAsync(number1:any,number2:any,callback:Function)
43 {
44     let result = 0;
45     setTimeout(function(){
46
47         if(typeof number1 === "number" && typeof number2 === "number")
48         {
49             result = number1 + number2;
50             callback(undefined,result);
51         }
52         else{
53             callback("argument value should be of type number",undefined);
54         }
55
56     },10000)
57 }
58
59
60 }
61
62 export default { CallbackClass };
```

In main.ts file

```

2  import {CallbackClass} from '../src/callback';
3  let callbackClass=new CallbackClass();
4  callbackClass.additionAsynch(10, 12, function (error : any, result : any) {
5
6      if (result) {
7
8          updateAddition(result,function(updateResult : any ){
9
10             console.log("Addition is : ",updateResult);
11             if(updateResult)
12             {
13                 Multiplication(updateResult,function( multiplicationResult : any ){
14                     if (multiplicationResult)
15                     {
16                         console.log("Multiplication is : ", multiplicationResult);
17                     }
18                 })
19             }
20
21             });
22         }
23         else {
24             someError(error);
25         }
26     })
27
28
29  function Multiplication(result: any, callback : Function) {
30      let mresult : any = result *2;
31      callback(mresult);
32  }
33  function updateAddition(result: any , callback :Function){
34      let uresult=result +2;
35      callback(uresult);
36  }
37  function someError(error: any) {
38      console.log("Error is : ", error);
39  }

```

In this example updateAddition, Multiplication and someError are nested functions calls callback functions. The problem with this kind of code is that this kind of situations can cause a lot of trouble and the code can get messy when there are several functions. This situation is called what is commonly known as a **callback hell**.

So, to find a way out, the idea of **Promises and function chaining** was introduced.

In promiseFile.ts file

```
25 public additionAsync(number1: any, number2: any) {  
26  
27     return new Promise(function (onResolve, onReject) {  
28  
29         if (typeof number1 === 'number' && typeof number2 === 'number' && number2 > 0) {  
30             onResolve(number1 + number2);  
31         }  
32         else {  
33             onReject("number1 and number 2 should be number type and non-negative");  
34         }  
35     })  
36 }  
37  
38 }
```

In

main.ts file where call above method.

```
62 callbackClass.additionAsync('10','12').then(  
63     function(result : any){  
64         let additionResult=result+2;  
65         console.log("Addition is : ", additionResult);  
66         return additionResult;  
67     }  
68 ).catch(  
69     function(error: any)  
70     {  
71         console.log("Error is : ", error);  
72         return error;  
73     }  
74 ).  
75  
76 then(  
77     function(mulResult : any){  
78  
79         let multiplicationResult=mulResult*2;  
80         console.log("Multiplication is : ", multiplicationResult);  
81         return multiplicationResult;  
82     }  
83 ).catch(  
84     function(error : any ){  
85         console.log("Error is : ",error);  
86     }  
87 );
```

The

above code demos a function implemented with **function chaining** instead of callbacks. It can be observed that the code is now more easy to understand and readable. The code basically says that **updateAddition** catch the error if there is any; if there is no error **then** implement the following statement:

```
console.log("Addition is :", 24);
```

```
console.log("Multiplication is :", 48);
```

With Node v8, the async/await feature was officially rolled out by the Node to deal with Promises and function chaining. The functions need not to be chained one after another, simply **await** the function that returns the Promise. But the function **async** needs to be declared before **awaiting** a function returning a Promise. The code now looks like below.

Ex: In `async_await.ts` file

```
3 import * as Promise from 'promise';
4 export function additionAsync(number1: any, number2: any) {
5
6     return new Promise(function (onResolve: any, onReject: any) {
7
8         if (typeof number1 === 'number' && typeof number2 === 'number' && number2 > 0) {
9             onResolve(number1 + number2);
10         }
11         else {
12             onReject("number1 and number 2 should be number type and non-negative");
13         }
14     })
15 }
16
17 export function Multiplication(result: any) {
18     return new Promise(function (onResolve: any, onReject: any) {
19         onResolve(result * 2);
20     })
21 }
22
23 export function updateAddition(result: any) {
24     return new Promise(function (onResolve: any, onReject: any) {
25         onResolve(result + 2);
26     });
27 }
28 export default { updateAddition, Multiplication, additionAsync}
```

In `async_await_main.ts` file call methods.

```
1 import { updateAddition, Multiplication, additionAsync } from '../src/async_await';
2 async function main() {
3   let additionResult: any = await additionAsync(10, 12);
4   if (additionResult.err) {
5     console.log("Error is :", additionResult.err);
6   }
7   else {
8     console.log("\nAddition is:\t", additionResult);
9   }
10
11   let updateResult: any = await updateAddition(additionResult);
12   if (updateResult.err) {
13     console.log("Error is :", updateResult.err);
14   }
15   else {
16     console.log("Addition after update is :\t", updateResult);
17   }
18   let multiplicationResult: any = await Multiplication(updateResult);
19   if (multiplicationResult.err) {
20     console.log("Error is :", multiplicationResult.err);
21   }
22   else {
23     console.log("Multiplication is :\t", multiplicationResult);
24   }
25   return additionResult;
26 }
27
28 console.log("\n \nAsynchronous call started");
29 main();
30 console.log("without waiting to end asynchronous call execute next statement ");
31
```

In the above code main is called asynchronously.

The code above basically asks the javascript engine running the code to wait for the **additionAsync** function to complete before moving on to the next line to execute it. The **additionAsync** function returns a Promise for which user will **await** . Before async/await, if it needs to be made sure that the functions are running in the desired sequence, that is one after the another, chain them one after the another as we call **updateAddition** and **Multiplication**. All those functions run synchronously and receiving updated result from one another. Code writing and understanding becomes easy with async/await as can be observed from both the examples. Result of given example is :

```
C:\Users\varsha.kone\Desktop\NodeJs\callback\src>node Async_wait_main.js
```

```
Asynchronous call started  
without waiting to end asynchronous call execute next statement
```

```
Addition is      :      22  
Addition after update is :    24  
Multiplication is :      48
```

Sending Requests to External API

Connecting to external APIs is easy in Node. You can just require the `core` module and start sending requests. Request module makes it simple to perform these HTTP requests. request allows you to make all types of HTTP requests, including GET, POST, PUT, and DELETE. Its flexibility makes the request module ideal for interacting with RESTful APIs. You can install request using the following npm command.

```
1 import * as request from 'request';
2
3 //key:248776b6718ccc9d57ec05c0e13860db
4 request({ uri: "https://api.darksy.net/forecast/248776b6718ccc9d57ec05c0e13860db/18.4088,76.5604",
5   json:true },
6
7   (error, response, body) => {
8
9     if(error)
10     {
11       console.log("Unable to connect with forecast darksky.net");
12     }
13     else if(response.statusCode===404)
14     {
15       console.log("Resource is not found");
16     }
17     else
18     {
19
20       console.log('Time Zone      :\\t', body.timezone);
21       console.log('about Weather :\\t', body.daily.summary);
22       console.log('Temperature is :\\t', body.currently.temperature);
23
24     }
25
26   });
```

C:\Users\varsha.kone\Desktop\NodeJs\ty_ModulesApp\request_Api>node external.js
Time Zone : Asia/Kolkata
about Weather : No precipitation throughout the week, with high temperatures falling to 83°F next Wednesday.
Temperature is : 86.97

Summary

“ Node.js support some library functions can execute asynchronously to avoid blocking of UI for faster performance of application “.

Footnotes

- [Location API Guide](#) (Refer this link for assignment 6)
- [Axios](#)
- [SuperAgent](#)
- [Got](#)

Assignments - Timeframe -1 day

Assignment 1 :

- Create application with name **Node_Asynchronous**.
- Create folder each for **ts_Modules** and **js_Modules**.
- Create folder **model** and create **user.ts** file with **user** object.
- user model will have properties like username: string, password: string ,firstName :string, lastName : string , Address : string etc.
- Input all details using module readline .

Assignment 2 :

Create **register.ts** local module with **newUser** function with signature : **newUser(user : any):promise**

Save user details in file **userDetails.json** synchronously and return Promise back to **home.ts** module and show result on console.

Assignment 3 :

Input user details like username and password for registration and save in **user** object and pass it to respective functions.

Assignment 4 :

Create **retrieveUser .ts** local module with signature :

retrieveUser() : promise

Retrieve user details from **userDetails.json** file and return to home.ts where we call method Asynchronously

-
- Assignment 5 :** Create module **googleapi.ts** with one function with signature :
getLocation(address: string). On invocation of method send request for google map api which returns latitude and longitude as an response. Display latitude and longitude on console. (Use async/await to call api)
- Assignment 6 :** Create home.ts local module with user input to ask for new registration , retrieve details or call api , accordingly call functions from **register.ts** , **retrieveUser .ts** and **googleapi.ts** (user switch case) . call all methods asynchronously using async and await.

7. Module 5 - Testing Application

What is Test Driven Development(TDD) And Behavioral Driven Development(BDD)

Test driven development is a methodology for writing the tests first for a given module and for the actual implementation afterward. If you write your tests before your application code, that saves you from the cognitive load of keeping all the implementation details in mind, for the time you have to write your tests.

BDD allows you to write software requirements as specifications in a human-readable format, and use those specifications to run tests that make sure that the software does what is expected.

Testing Frameworks In Js

- **Mocha** : Mocha is a Javascript test framework running on Node Js. It tests run serially, allowing for flexible and accurate reporting, while mapping uncaught exceptions to the correct test cases.

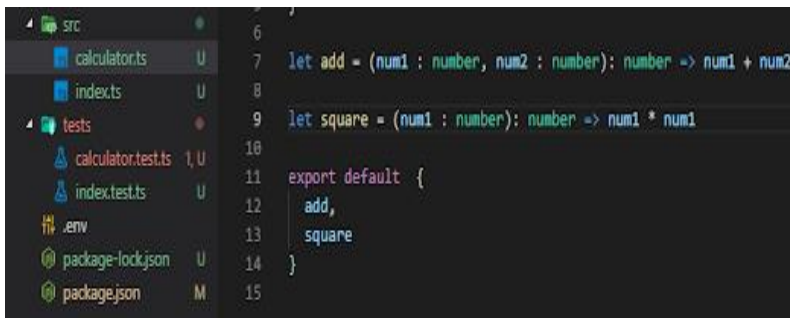
Mocha And Basic Testing

As mocha is a popular unit test framework we are going to use it as a testing framework throughout our course. [TDD in terms of mocha](#). [BDD in terms of mocha](#).

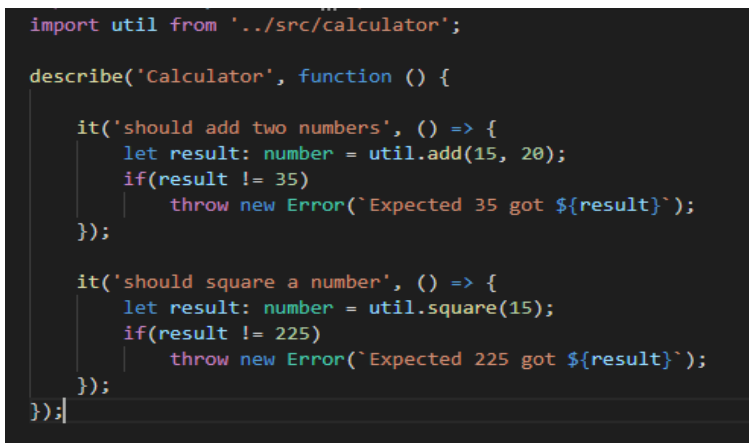
- **Installation :**
Install with [npm](#) globally :
 - Run `npm install --global @types/mocha`
- Or Install with `tsd` to get all type definitions :
 - Run `tsd install mocha --save-dev`
- Resolve dependencies :
 - Run `tsd install mocha -s -r`

- **Unit Testing :**

- It is a level of software testing where individual units/ components of a software are tested. The purpose is to validate that each unit of the software performs as designed.
- Example - Following example has a calculator.ts file with some functionality and calculator.test.ts test file for unit testing.



```
6
7 let add = (num1 : number, num2 : number): number => num1 + num2
8
9 let square = (num1 : number): number => num1 * num1
10
11 export default {
12   add,
13   square
14 }
15
```



```
import util from '../src/calculator';

describe('Calculator', function () {

  it('should add two numbers', () => {
    let result: number = util.add(15, 20);
    if(result != 35)
      throw new Error(`Expected 35 got ${result}`);
  });

  it('should square a number', () => {
    let result: number = util.square(15);
    if(result != 225)
      throw new Error(`Expected 225 got ${result}`);
  });

});
```

Assertion Libraries

- Please refer [Assertion Libraries](#)
- **Installation :**
 - Run `npm install --global @types/expect`
 - Run `tsd install expect -s -r`
- **Example** - In above example we are checking results, because mocha doesn't come with inbuilt assertion libraries, we need to externally add those. In our course we are going to use [expect](#) for assertions.

Testing Asynchronous Code

- Please go through [asynchronous code testing](#) in mocha.
- **Example** - In following example we are going to test `asyncSquare()` method which is a asynchronous method.

```
1
2 import * as Promise from 'promise';
3 export function asyncSquare(number : any)
4 {
5     return new Promise(function(onResolve : any, onReject : any){
6
7         let square= number*number;
8         onResolve(square);
9
10    })
11 }
12 export default { asyncSquare }
```

Test and its result -

```
1
2 import {describe} from 'mocha';
3 import * as expect from 'expect';
4 import * as supertest from 'supertest';
5 import * as should from 'should';
6 import {asynchSquare} from '../src/asynchSquare';
7 describe("Call asynchSquare method",function(){
8   it("Should Calculate Square of Number", async function() {
9
10     let result = await asynchSquare(10);
11     if (result) {
12       console.log(result);
13       expect(result).toBe(100);
14       return result;
15     }
16   })
17 })
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

Call asynchSquare method
100
✓ Should Calculate Square of Number

1 passing (39ms)

Using Promise and Async/Await

- Please go through [Promise](#) and [Async/Await](#) in mocha.

Testing API's

- Now a days most of the node applications are using node api in backend and before going to production we have to check existing functionality is working as expected or not, Therefore testing of api is important task.
- There are different npm modules are available for testing of requests in node js such as Chai, Supertest etc.
- In this course we are going to use supertest for http assertions.
- Example - Following example has a server.js file with all routes we are going to test using supertest and mocha.

```
//POST /users
router.post('/users/register', async (req: Request, res: Response) => {
  try{
    let body: object = _.pick(req.body, Object.keys(req.body));
    if((body['email'] != undefined) && body['password'] != undefined){
      let user: any = new User(body);
      user = await user.save();
      res.send(user);
    }
    else res.status(404).send('Empty data not allowed');
  } catch(e) {res.status(404).send(e)}
});

//GET /users
router.get('/users', authenticate, async (req: Request, res: Response) => {
  try{
    var users = await User.find();
    res.send(users);
  } catch(e) {res.status(404).send(e)}
});
```

- Server.test.js is a test file, which has test cases for users post and get requests.

```
describe('/POST /users/register', function() {
  it('should create user', function(done) {
    this.timeout(10000);
    request(app)
      .post("/users/register")
      .send(user)
      .expect(200)
      .expect((res) => {
        expect(res.body).not.toBeNull();
      })
      .end(async (err, res) => {
        if(err) return done(err);
        let retrivedUser: any = await User.findOne({ email: user['email'] })
        expect(retrivedUser._id).not.toBeNull();
        expect(retrivedUser['email']).toBe(user['email']);
        done();
      });
  });
});

describe('/GET /users', function() {

  it('should return all users', function(done) {
    this.timeout(10000);
    request(app)
      .get("/users")
      .set('x-auth', factory.users[0].tokens[0].token)
      .expect(200)
      .expect((res) => {
        expect(res.body.length).toBeGreaterThan(0);
      })
      .end(done);
  });
});
```

- Please refer [Supertest](#) for more information.

Summary

“ Before deploying application on production environment need to check code produces expected output or not using test framework introduced in node.js as mocha, chai,supertest, assert and expect “.

Footnotes

- [Integration testing in node js](#)
- [Mocha framework](#)

Assignments - Timeframe - 1/2 day

Assignment 1 : Write unit tests for assignments given for Node_module, and Node_Asynchronous.

8. Module 6 - Routing

How to Serve request with Routing?

- **URL** : A URL is an address that shows where a particular page can be found on the server.
Example : `http://localhost:1212/home.html`
- **Route** : Place where to register endpoints (IP address:port/name of page).
Example : `/home.html`
- **Route Handler** : Whenever a request is received , it is the job of the **routing** engine to match the request URL with the registered **routes**. After finding the matched **route**, the **route handler** function(Callback function) for the **route** is being called to render response.
- **Routing** : Routing refers to determining how an application responds to a client request to a particular endpoint, which is a URI (or path) and a specific HTTP request method (GET, POST, and so on).Each route can have one or more handler functions, which are executed when the route is matched. So how Nodejs allows to create Route, Route Handler and start Routing where Node modules comes in picture. Node framework provides diff two core modules to create web server and start routing. Http and Express.
 - **Express** :
The express framework provides an abstraction layer above the vanilla http module to make handling web traffic and APIs a little easier. There's also tons of middleware available for express (and express-like) frameworks to complete common tasks such as: CORS, XSRF, POST parsing, cookies etc.Express uses the http module internally, `app.listen()` returns an instance of http
 - **Http** :
The http api is very simple and is used to to setup and manage incoming/outgoing ,HTTP connections. Node does most of the heavy lifting here but it does provide things you'll commonly see throughout most node web framework such as: request/response objects etc.

How to create Server Without Express using Http?

- Please refer [create server](#) for server creation guide. before creating server install http module run the command. **npm i http --save-dev** and **npm i @types/http --save-dev**

```
import { IncomingMessage, OutgoingMessage, Server, createServer } from 'http';

let hostname : string = '127.0.0.1';

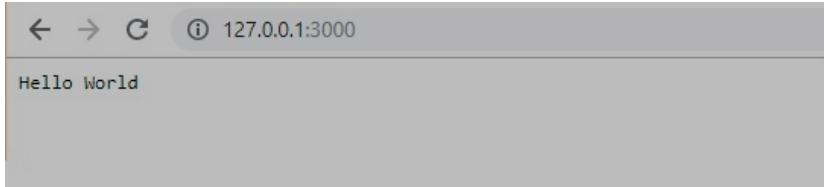
let port : number = 3000;

let server : Server = createServer(function(request: IncomingMessage, response: OutgoingMessage){
  if (request.url === '/') {
    response.setHeader('content-type', 'text/plain');
    response.end("Hello World");
  }
});

server.listen(port, hostname, function () {
  console.log('Server running at : ${hostname}:${port}');
})
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
Server running at : 127.0.0.1:3000
█
```



What is Express Framework?

- Please refer [express framework](#).
- Express is a popular Node.js web framework. designed for building single-page, multi-page, and hybrid web applications. It provides a wrapper for Routing. The Express Router helps in the creation of Routes and route handlers and start routing process. Following are some of the core features of Express framework -
- Allows to set up middlewares to respond to HTTP Requests.
- Defines a routing table which is used to perform different actions based on HTTP Method and URL.
- Allows to dynamically render HTML Pages based on passing arguments to templates.
- Allows to create server where listen to request.

Follow the steps to create server and serve http get, post request.

Step 1) Create instance of Express. before creating instance install express module run the command. `npm i express--save-dev` and `npm i @types/express --save-dev`

```
3 import * as express from 'express';
4
5 const application: any = express();
```

Step 2) Call Route methods. Express supports methods that correspond to all HTTP request methods: get, post, and so on. To create a route, we can use either the express module or

express.Router both serves same intention, but router not able to start server instead use express.

If you are using express

`App.method('path', 'handler');`

```
//Register all routes
application.get("/home",function(request : Request, response : Response){
    response.render("home.hbs");
})

application.post("/home", function (request: Request, response: Response) {
    response.render("home.hbs", { message: "Welcome Home"});
})
```

If using Router

`router.method('path', 'handler');`

```
1
2
3 import * as express from 'express';
4 import {Request,Response} from 'express';
5
6 let router:any=express.Router();
7
8 router.get('/home', function (request: Request, response: Response, next: Function) {
9
10     response.render('home.hbs');
11 })
12 router.post('/home', function (request: Request, response: Response, next: Function) {
13
14     response.render('home.hbs');
15 })
16 router.get('/home/:key', function (request: Request, response: Response, next: Function) {
17
18     response.render('home.hbs');
19 })
20
```

Where:

- App is an instance of express.
- METHOD is an HTTP request method, in lowercase.
- PATH is a path on the server.
- HANDLER is the function executed when the route is matched.

Step 3) Route paths, in combination with a request method, define the endpoints at which requests can be made. Route paths can be strings, string patterns, or regular expressions.

```
16 router.get('/home/:key', function (request: Request, response: Response, next: Function) {  
17  
18     response.render('home.hbs');  
19 })
```

Step 4) Configure serve at port number as environment specific where you deploy application
Example : Azure, Heroku, Kubernetes ect.

Use **Process.env.PORT** variable to get default port of deployment environment or go with local default as

const port: any = process.env.PORT || 3000

```
34 let port: any = process.env.port || 8000;  
35  
36 application.listen(port,function() {  
37  
38     console.log(`server started at ${port}`);  
39  
40 })
```

```
C:\Users\varsha.kone\Desktop\NodeJs\TypeScript\RestApi\Routing>node startserver.js  
server started at 8000
```

← → ↻ ⓘ localhost:8000/home

Home Page

Step 5) Route parameters are named URL segments that are used to capture the values specified at their position in the URL. The captured values are populated in the req.params object, with the name of the route parameter specified in the path as their respective keys.

Route path: /users/:userId

Request URL: http://localhost:3000/users/34

req.params: { "userId": "34" }

```
16 router.get('/home/:key', function (request: Request, response: Response, next: Function) {  
17  
18     response.render('home.hbs');  
19 })
```

← → ↻ ⓘ localhost:8000/home/enigma

Home Page

What is Middleware?

Middleware in node. Js Handles request(routing), creates session, cookie for user,Create instance of route and register it, sets default view engine ,Handle Error occurred while processing request. Parse the request parameters in json, text or xml.Handle all Custom process in application etc. **Middleware** functions are functions that have access to the request object (req), the response object (res), and the next function in the application's request-response cycle. The next function is

a function in the Express router which, when invoked, executes the middleware succeeding the current middleware. Please refer [middleware](#).

How to manage Static files?

Static files are data on server side which we want to show on client side. E.g - Images, CSS files, html files and JavaScript files etc.

To serve static files, we can use the `express.static` built-in middleware function in express.

Example: In following example the files (login.html, bootstrap.css) which we want to serve to client are in public folder and public folder is served to client using `express.static` middleware function. before setting path install path module run the command. **npm i path--save-dev** and **npm i @types/path --save-dev**

And import it **import * as path from 'path';**

```
20
21 //set default static resource path
22 application.use(express.static(path.join(__dirname, 'public')));
23
```

What View Engine(Template) Node.Js supports ?

Template engine produces the final HTML using template and data in client's browser. However, some HTML templates process data and generate final HTML page at server side also. The following is a list of important (but not limited) template engines for Node.js.

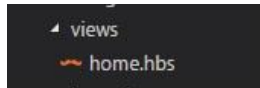
Please refer [view engine](#) in node.

- [Jade](#)
- [Vash](#)
- [EJS](#)
- [Mustache](#)
- [Dust.js](#)
- [Nunjucks](#)
- [Handlebars](#)

- o [atpl](#)
- o [haml](#)

Follow steps to create web page using view Engine

Step 1) Create folder with name “**views**”. Create file with extension **.hbs**, depend on what view engine to use. Install handlebars module.



Step 2) In this case we are using handlebars view engine. Create page by adding html tags. Use salesforce slds style to make UI responsive.

Download SLDS at [Download the Salesforce Lightning Design System CSS framework](#)

And save the folder in public folder.add it's link in page.

```
<!DOCTYPE html><html><head>
  <link rel="stylesheet" type="text/css" href-
    "/Styles/salesforce-lightning-design-system-2.7.4/assets/styles/salesforce-lightning-design-system.css"
  />
</head><body><form action="/home" method="post">
  <div class="demo-only" style="height: 640px;">
    <section role="dialog" tabindex="-1" aria-labelledby="modal-heading-01" aria-modal="true" aria-describedby-
      class="slds-modal slds-fade-in-open">
      <div class="slds-modal__container">
        <header class="slds-modal__header">
          <button class="slds-button slds-button__icon slds-modal__close slds-button__icon-inverse" title='
            <svg class="slds-button__icon slds-button__icon_large" aria-hidden="true">
              </svg>
            <span class="slds-assistive-text">Close</span>
          </button>
          <h2 id="modal-heading-01" class="slds-text-heading_medium slds-hyphenate">Login</h2>
        </header>
        <div class="slds-modal__content slds-p-around_medium" id="modal-content-id-1">
          <p> <label class="slds-form-element__label" for="text-input-id-1">First Name</label>
            <div class="slds-form-element__control">
              <input id="textfnn" class="slds-input_borders" type="text" value="" /></div>
          </p>
          <p> <label class="slds-form-element__label" for="text-input-id-1">Last Name</label>
            <div class="slds-form-element__control">
              <input id="textlnm" class="slds-input_borders" type="text" value="" /></div>
          </p>
          <p>
            <div class="slds-form-element__control">
              <label class="slds-form-element__label" for="text-input-id-1">Full Name : {{fullName}}</label>
            </div>
          </p>
        </div>
      </div>
    </div>
  </div>
</body></html>
```

Step 3) middleware has to set default view engine for your app. before setting default view engine install express-handlebars module run the command. `npm i express-handlebars --save-dev` and `npm i @types/express-handlebars --save-dev`

```
15 import * as path from 'path';
16
17 //set default view engine
18 application.engine('.hbs', hbs({ extname: '.hbs' }));
19 application.set('view engine', '.hbs');
20
21 //set default static resource path
22 application.use(express.static(path.join(__dirname, 'public')));
23
```

Step 4) Register route using middleware which return response on post request.

```
router.post('/home', function (request: Request, response: Response) {
    let firstName: any = request.body.firstName;
    let lastName: any = request.body.lastName;
    response.render('home.hbs', {fullName : firstName+' '+lastName});
})
```

Submitting Data using Body-parser

In order to read HTTP POST data , we have to use "body-parser" node module. body-parser is a piece of express middleware that reads a form's input and stores it as a javascript object accessible through **req.body**. With express you can read any data inside HTTP request, such as headers **req.headers** (array), you can read the body of the http packet as **req.body** and you can read as query parameter **req.query.variable**, It helps since express automatically transforms the request in javascript objects. This body-parser module parses the JSON, buffer, string and URL encoded data submitted using HTTP POST request. Install body-parser using NPM before setting default view engine install body-parser module run the command.

```
npm i body-parser --save-dev and  
npm i @types/body-parser --save-dev
```

```
24 //set default result type json/xml  
25 application.use(bodyParser.json());  
26 application.use(bodyParser.urlencoded({extended:true}));  
27  
28 application.use('/',router.router);  
29
```

```
router.post('/home', function (request: Request, response: Response) {  
    let firstName: any = request.body.firstName;  
    let lastName: any = request.body.lastName;  
    response.render('home.hbs',{fullName : firstName+' '+lastName});  
})
```

Login

First Name

Enzigma

Last Name

Software Pvt Ltd

Full Name : Enzigma Software Pvt Ltd

Login

Summary

“ Routing Maps URL pattern with routes register and start processing response “.

Assignments - Timeframe -1 day

Assignment 1 : Create application(package) with name **Node_Routing**

Assignment 2: Create folder with name **public** to save all static resources (CSS, JS)

Assignment 3 : Create folder with name **views** and save all pages given in it.

- Design a login page with textbox for username, password. Give link for forgot password, Sign Up for new user.
- Design Registration page for creating new user.
 - Create **forgotPassword** page to send password at email_id.
 - Create **changePassword** page to change password.
 - Design Profile page to show details of user. Put a link for change password.

Assignment 4 : Create folder with name **Routing** to create server.

Create **route.ts** module to create server and route for every page with respected route methods using Middleware.

Assignment 5 : Use bootstrap/salesforce slds to design the UI.

9. Module 7 - MongoDB

What is MongoDB?

MongoDB is an open source, document oriented database that stores data in form of documents (key and value pairs).that document based databases are one of types of NoSQL databases. Please refer [mongodb](#) for more information.

What is the Aggregation pipeline?

Refer [mongodb aggregation](#) (Refer any link to learn)

Features of MongoDB?

MongoDB provides high performance,auto replication,Horizontal scaling vs vertical scaling, Load balancing,High Availability,Indexing etc. MongoDB is a schemaless NoSQL document database. It means you can store JSON documents in it, and the structure of these documents can vary as it is not enforced like SQL databases. This is one of the advantages of using NoSQL as it speeds up application development and reduces the complexity of deployments. Please refer [mongodb features](#) for more information.

- **Collections**

'Collections' in Mongo are equivalent to tables in relational databases. They can hold multiple JSON documents.

- **Documents**

'Documents' are equivalent to records or rows of data in SQL. While a SQL row can reference data

in other tables, Mongo documents usually combine that in a document.

- **Fields**

'Fields' or attributes are similar to columns in a SQL table.

- **Schema**

While MongoDB is schema-less, SQL defines a schema via the table definition. A Mongoose 'schema' is a document data structure (or shape of the document) that is enforced via the application layer.

- **Models**

'Models' are higher-order constructors that take a schema and create an instance of a document equivalent to records in a relational database.

Below is an example of how data is stored in Mongo vs. SQL Database:

SQL

PERSON

Id	FirstName	LastName	Email
1	Ada	Lovelace	ada.lovelace@gmail.com
2	Grace	Hopper	grace.hopper@gmail.com
3	Kathy	Sierra	kathy.sierra@gmail.com

PHONE NUMBER

PersonId	PhoneId	Phone Number	Type
1	1	+1.123.456.7890	Home
1	2	+1.111.222.3333	Work



MONGO

PEOPLE

```
{
  "Id": 1,
  "FirstName": "Ada",
  "LastName": "Lovelace",
  "Email": "ada.lovelace@gmail.com",
  "Phone": [{
    "Home": "+1.123.456.7890"
  }, {
    "Work": "+1.111.222.3333"
  }]
}

{
  "Id": 2,
  "FirstName": "Grace",
  "LastName": "Hopper",
  "Email": "grace.hopper@gmail.com"
}
```

Installing MongoDB and Compass.

- [Download MongoDB and Experience it for different platform.](#)

How do i start MongoDB from windows?

Step 1) Download the [mongodb](#). Follow normal setup instructions

Step 2) Create the following folder as C:\Desktop\MongoData

Step 3) Open the cmd and run following command .

cd to C:\Program Files\MongoDB\Server\3.2\bin>

enter command **mongod**

by default, mongodb server will start at port 27017

Step 4) download MongoDB [Compass](#) and follow normal setup instructions.

Step 5) Start MongoDB Compass and create a new connection on **localhost:27017**

Core module to connect mongo from node.

Install first mongodb core module in **newUser.ts** file

npm install @types/mongodb and import as given.

```
3 import * as mongo from 'mongodb';
4
5 let mongoClient: any = mongo.MongoClient;
6 let mongoPromise = mongoClient.connect("mongodb://localhost:27017/UsersDB",
7   { useNewUrlParser: true });
8
9 export class EmployeeManagement {
10
11   public newEmployee(employee: Object) {
12
13     return mongoPromise.then(function (client: any) {
14       let database = client.db("UsersDB");
15
16       return database.collection("EmployeeCollection").insertOne(employee);
17     })
18   }
19 }
20
21 export default EmployeeManagement
```

Once query written to create new record in database test it using mocha as given in **test.ts** file in test folder.

```
2 import { describe } from 'mocha';
3 import * as expect from 'expect';
4 import { EmployeeManagement } from '../Controller/newUser';
5
6 describe("Employee Manipulation",function(){
7
8     it("Creating new Employee", async function(){
9
10         let employeeManagement = new EmployeeManagement();
11         let employee : Object ={empId :123, empName: "Enzian", empAddress:"pune"};
12         let response= await employeeManagement.newEmployee(employee);
13         //test getting expected result
14         expect(response).not.toEqual(response.error,"Not expected result");
15
16     })
17
18 })
19
```

To run test method on specific environment do configuration in package.json file as

```
"scripts": {
  "test": "SET NODE_ENV=test/starttest && node node_modules/mocha/bin/_mocha --recursive"
},
```

Run the command

npm test

```
C:\Users\varsha.kone\Desktop\NodeJs\TypeScript\RestApi>npm test

> restapi@1.0.0 test C:\Users\varsha.kone\Desktop\NodeJs\TypeScript\RestApi
> SET NODE_ENV=test/starttest && node node_modules/mocha/bin/_mocha --recursive

Employee Manipulation
  ✓ Creating new Employee (1349ms)

1 passing (1s)
```

MongoDB have primary key?

In MongoDB, the primary key is automatically created and assigned to the `_id` field. If you do not specify the `_id` field, it will be auto-generated for you. The `_id` field is always indexed and always unique. `_id` is of type `ObjectId` class in mongodb. You can create your own `objectId` while inserting new document(record). Please refer [primary key](#) concept in mongodb.

[CRUD operations on document\(don't forget visit link to go through in detail \).](#)

Summary

“ MongoDB is one of the most popular NoSql database used as database application in Node.js “.

Footnotes

- [What is NoSQL?](#)
- [Features of NoSQL.](#)
- [Why NoSQL](#)
- [Difference between RDBMS and NoSql](#)
- [Documentbase Database](#)
- [MongoClient Collection and Methods](#)

Assignments - Timeframe -1 day

Assignment 1: Create node application with name **Node_Mongodb**.

Assignment 2: Create folder with name **config** where to create file **config.json** to save connection string for connection.ts module.

Assignment 3: Create folder with name **object** where create object **userId** in module **userId.ts** with schema `{_id : "userid"}`

Assignment 4: Create folder with name **dbcontroller** where to create modules **connection.ts** and write CRUD functionality.

- Define functionality **createUser** with signature :
public createUser (userCollectionName: string, user: any) . Write query to create new user in collection **allUserdb**. Before creating new document check user already exist or not , if present return error “ **user already Exist**”.
- Define functionality **verifyUser** with signature :
public verifyUser (userCollectionName: string, user: any) . Write query to check user present in collection. **Once user verified save userid in userid object by setting it's property _Id**
- Define functionality **removeUser** with signature :
public removeUser (userCollectionName: string, user: any) . Write query to delete user from collection.
- Define functionality **editUser** with signature :
public editUser (userCollectionName: string, user: any) . Write query to edit user password, email ,mobilenumber in collection.
- Define functionality **returnUser** with signature :
Public returnUser (userCollectionName: string, userId: any) . Write query to return user details by verifying user with it's user id in collection. **(Use userid saved in userid object.user id saved in this object when user is verified in verifyUser functionality)**

Assignment 5: Create folder with name **controller** where to create modules **user.ts** with handler functionality.

- Define function **newUser** with signature :
public newUser(user: any) and call createUser method from connection.ts module.
- Define function **validateUser** with signature :
public validateUser (user: any) and call verifyUser method from connection.ts module. Pass username and password in user object to verify user
- Define function **removeProfile** with signature :

public removeProfile (user: any) and call removeUser method from connection.ts module.

- Define function **updateProfile** with signature :
public updateProfile (user: any) and call editUser method from connection.ts module.
- Define function **userProfile** with signature :
public userProfile (user: any) and call returnUser method from connection.ts module.

Assignment 6: Create folder with name **unitTest** where to create modules **user.test.ts** and write negative and positive test method for **user.ts** functionality.

Assignment 7: Commit changes to Git and push it to GitLab.

10. Module 8 - Rest Api

What is Mongoose?

[Click on link to get details of Mongoose](#)

Mongoose is an Object Document Mapper (ODM). This means that Mongoose allows you to define objects with a strongly-typed schema that is mapped to a MongoDB document.

Mongoose provides an incredible amount of functionality around creating and working with schemas. Mongoose currently contains eight SchemaTypes that a property is saved as when it is persisted to MongoDB. They are:

- String
- Number
- Date
- Buffer
- Boolean
- Mixed
- ObjectId
- Array

Each data type allows you to specify:

- a default value
- a custom validation function
- indicate a field is required
- a get function that allows you to manipulate the data before it is returned as an object
- a set function that allows you to manipulate the data before it is saved to the database
- create indexes to allow data to be fetched faster

Why Mongoose?

Here are four reasons why using Mongoose with MongoDB is generally a good idea.

1. Schemas
2. Validation

3. Instance Methods

4. Returning results

Please refer [mongoose](#) for more information.

Mongoose Schema vs. Model

A Mongoose model is a wrapper on the Mongoose schema. A Mongoose schema defines the structure of the document, default values, validators, etc., whereas a Mongoose model provides an interface to the database for creating, querying, updating, deleting records, etc.

Defining the Schema in Node.js

A schema defines document properties through an object where the key name corresponds to the property name in the collection. Before using mongoose install it run the command. **npm install mongoose** and **npm i @types/mongoose**

Example : 1.1

```
TS employeeSchema.ts x
1  import * as mongoose from 'mongoose';
2
3  const schema = mongoose.Schema;
4
5  let employeeSchema = new schema({
6    empId: Number,
7    empName: String,
8    empAddress: String
9  })
10
11  export default employeeSchema
```

What is Mongodb schema validator

Refer [Schema validator](#) or (Refer any link to learn)

Rest api's in details

Refer any link to learn.

Defining Model in Node.js

We need to call the model constructor on the Mongoose instance and pass it the name of the collection and a reference to the schema definition.

Example : 1.2

```
TS employeeCollection.ts ✕
1  import * as mongoose from 'mongoose';
2  import employeeSchema from '../Schema/employeeSchema';
3
4  //connection with DB
5  mongoose.connect("mongodb://localhost:27017/EmployeeDB", { useNewUrlParser: true });
6
7  //Create model
8  let employeeModel = mongoose.model("employeeCollection", employeeSchema);
9
10 export default employeeModel
11
12
```

Define Handler to insert new model in db.

Example : 1.3


```
$ newEmployee.ts x
1  import employeeModel from '../Models/employeeCollection';
2
3  export function newEmployee(employee: Object) {
4
5      //Instantiate model
6      let newEmployeeModel = new employeeModel(employee);
7
8      //Start insertion operation
9      return newEmployeeModel.save();
10 }
11
12 export default newEmployee
13
```

[CRUD Operations in Mongoose](#). (Don't forget to click on link to go in detail)

Rest Application Using Mongoose, Express

REST APIs handle the server side of the web application. That means that the backend is built once and can provide content or data for frontend, mobile, or other server-side apps. A great example is the google calendar API.

REST stands for Representational State Transfer and is a way how a web server should respond to requests. This allows to not only read data, but also do different things like updating, deleting or creating data

REST stands for Representational State Transfer. REST is web standards based architecture and uses HTTP Protocol. It revolves around resource where every component is a resource and a resource is accessed by a common interface using HTTP standard methods. A REST Server simply provides access to resources and REST client accesses and modifies the resources using HTTP

protocol. Here each resource is identified by URIs/ global IDs. REST uses various representation to represent a resource like text, JSON, XML but JSON is the most popular one HTTP methods

Following four HTTP methods are commonly used in REST based architecture.

- GET - This is used to provide a read only access to a resource.
- PUT - This is used to create a new resource.
- DELETE - This is used to remove a resource.
- POST - This is used to update a existing resource or create a new resource

To develop Rest Applications using Express, Mongoose is pretty Simple in Node.Js.

Follow Steps to create Rest Api

Step 1) Define Schema of model in Schema folder and export it as given in above **Example : 1.1.**

Step 2) Define model in separately in model folder and export it as given in above **Example : 1.2.**

Step 3) Connect with database as specified above

Step 4) Write CRUD Operations in module using mongoose as given in above **Example : 1.3.**

Step 5) Register routes for each function where create, edit , delete

operations are defined (according to requirements) at different route methods using Express.

```
TS route.ts x
1
2 import * as express from 'express';
3 import { Request, Response } from 'express';
4 import { newEmployee } from '../Controller/newEmployee';
5
6 let router: any = express.Router();
7 router.get('/newemployee', function (request: Request, response: Response) {
8     // response.type("text");
9     // response.contentType("application/text");
10    response.contentType("application/json");
11    response.json("new employee");
12
13 })
14 router.post('/newemployee', function (request: Request, response: Response) {
15     let result = newEmployee(request.body);
16     result.then(function (result: any) {
17         if (result._id) {
18             // response.send(result);
19             response.contentType("application/json");
20             response.json("new employee created successfully");
21         }
22     }).catch(function (error: any) {
23         response.sendStatus(404);
24     })
25 })
```

Step 6) Test api function writing test methods or through test tool

Postman with different Http methods..

```
startTest.ts x
1 import { describe } from 'mocha';
2 import * as expect from 'expect';
3 import * as request from 'supertest';
4 import * as app from "../Routing/startserver";
5 import { EmployeeManagement } from '../Controller/newUser';
6 import { Response } from "express";
7
8 describe("Start Request", function () {
9   it("new get request to create new Employee", async function () {
10     this.timeout(10000);
11     let response : any = await request(app).get("/newemployee").
12       set('Accept', 'application/json').
13         expect(200);
14     expect(response.body).toBe("new employee");
15   })
16   it("new post request to create new Employee", async function () {
17     this.timeout(10000);
18     let employee = { empId: 900, empName: "Mahendra Dhoni", empAddress: "India" };
19     let response : any = await request(app).post('/newemployee').send(employee).
20       set('Accept', 'application/json').expect(200);
21     expect(response.body).toBe("new employee created successfully");
22   })
23 })
24
25
26
27 });
```

Test Result

```
C:\Users\varsha.kone\Desktop\NodeJs\TypeScript\RestApi>npm test

> restapi@1.0.0 test C:\Users\varsha.kone\Desktop\NodeJs\TypeScript\RestApi
> SET NODE_ENV=test/starttest && node node_modules/mocha/bin/_mocha --recursive

Start Request
  ✓ new get request to create new Employee (125ms)
  ✓ new post request to create new Employee (1011ms)

2 passing (1s)
```

Activ

Testing Rest api in Node.Js

Test Rest Applications by writing mocha test cases or using api testing tool like Postman.

Summary

“ Mongoose ODM framework given for mongoDB applications. Mongoose, MongoDB and Express Frameworks can be used to develop Rest api in node.js, those api can be used by any third party applications “.

Footnotes

- [Mongoose Queries](#)
- [Mongoose Validators.](#)
- [Mongoose Middleware.](#)
- [Mongoose Populate](#)
- [Mongoose Plugin](#)
- [Mongoose Discriminators.](#)

Assignments - Timeframe - 1 days

Assignment 1 : Create node application (package) with name **Node_RestApi**.

Assignment 2 : Going to create node application for **newly joined trainees in enzigma**.

Assignment 3 : Create **objects** folder separately and create schema for model **trainee** as

Property name	type	required
traineeName	String	yes
traineeQualification	String	yes
traineeJoiningdate	Date	yes
traineeTrainings	Array	Set default training as salesforce admin and Apex
traineeTrainingStatus	String	Values- Training, Passed, Failed
traineeAllocatedproject	String	
traineeTrainingScore	Number	

Assignment 4 : Create **model** folder separately and create model **traineesCollection** in **traineeModel.ts** module.

Assignment 5 : Create **connection** folder separately and create TraineeManipulation class in traineeManipulation.ts module and add following functionality.

- Define method **createTrainee** with signature :
public createTrainee(trainee: any) and write mongoose query to create new trainee in collection.
- Define method **getAllTrainees** with signature :
public getAllTrainees() and write mongoose query to return trainees collection.
- Define method **updateTrainee** with signature :
public updateTrainee (trainee: any) and write mongoose query to update traineeAllocated project, trainingStatus, trainingScore in traineesCollection.

- Define method **cancelTrainee** with signature :
public cancelTrainee (traineeId: any) and write mongoose query to remove trainee in collection whose score is less than 5 and allocated project is empty.

Assignment 4 : Create folder with **controller** in root app folder and create **TraineeClass** in **traineeClass.ts** module with following functionality.

- Define method **newTrainee** with signature :
public newTrainee(newTrainee: any) and call method **createTrainee** from **TraineeManipulation** class.(call methods **Asynchronously using Async**)
- Define method **allTrainee** with signature :
public allTrainee () and call method **getAllTrainees** from **TraineeManipulation** class.
- Define method **editTrainee** with signature :
public editTrainee (newTrainee: any) and call method **updateTrainee** from **TraineeManipulation** class.
- Define method **removeTrainee** with signature :
public removeTrainee (traineeId: any) and call method **cancelTrainee** from **TraineeManipulation** class.

Assignment 5 : Create folder with **routing** in root folder.

- Create **routes.ts** module to register routes for all controller handler at route methods. Configure the server either at production environment or local.
- Create **startServer.ts** file to start server by importing routes.ts module.

Assignment 6 : Create unitTest folder in root folder. Create **trainee.test.ts** , write test method with positive and negative response for TraineeClass functionality.

Assignment 7 : Create integrationTest folder in root folder. Create **routes.test.ts** , write test method for endpoint with positive and negative response for routes.ts module functionality.

Assignment 8 : Commit changes to Git and push it to Gitlab.

11. Module 9 - Authentication and Authorization

Authentication

It is the verification of the credentials of the connection attempt. This process consists of sending the credentials from the remote access client to the remote access server in an either plaintext or encrypted form by using an authentication protocol.

Authorization

It is the verification that the connection attempt is allowed. Authorization occurs after successful authentication. We can authorize user using JWT node library.

Why to Authorize User?

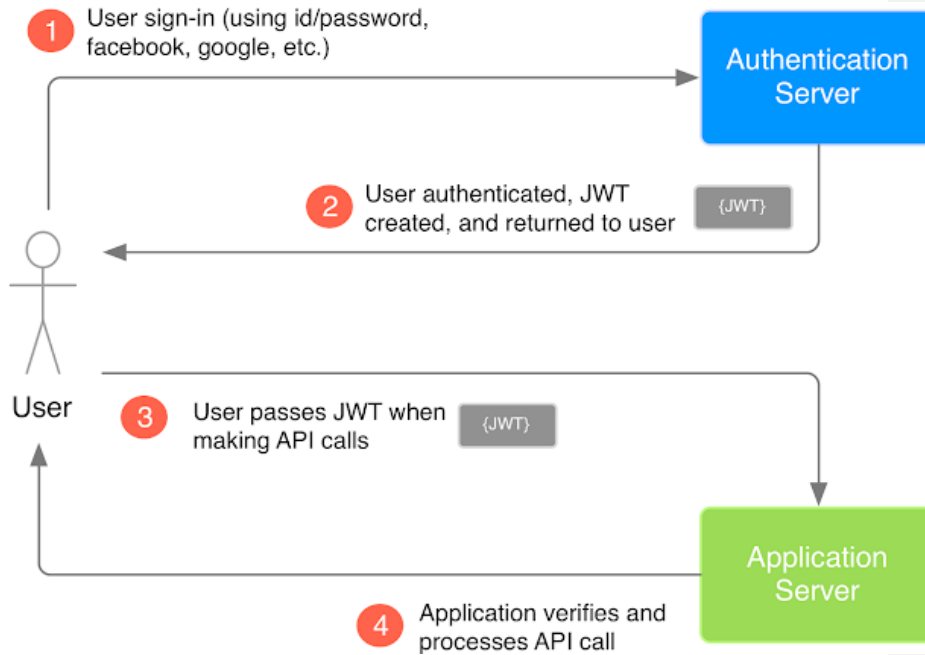
When we are using third party api in our project ex: google map by using endpoint url given by google, to protect APIs from unauthenticated users, Subsequent request coming from the same user for any other secured resource of server has to verify user again and again is called authorization. instead of verifying user with its username and password server used to use keys/token to ensure authorizing access to a resource..

User Authorization using token :

1. We have to login first with username and password , google authenticate user and generate one token (public/private key).
2. Return token back with response to user.
3. When same user requesting for same endpoint api again google has to ensure user is right or not while instead of using username and password use token send by user with request. for every subsequent request same process goes at server side.
4. Meanwhile can set token with expiry also. If user token expires not allowed to request for same endpoint api.
5. To achieve this type of authorization node provides library JWT which digitally sign(Token) and verifies token.

JWT And Hashing

To go in details of JWT please click [here](#)



When Should i use JWT?

JWTs are typically used to replace session identifiers. For example: if you're using a session system which stores an opaque ID on the client in a cookie while also maintaining session in a database for that ID. With JWTs you can replace both the session data and the opaque ID.

You'll still use a cookie to store the access token. With the token stored in a secure cookie, the user's client will supply the token on every subsequent request to your server. This allows the server to authenticate the request, without having to ask for credentials a second time (until the token expires, that is).

[How to Create a JWT ?](#)

- 1) Generate public and private key.
- 2) Generate claim/payload.
- 3) Generate token.

Summary

“ Before allow to access any secured api or resource to third party users we have to check user is right or not (authenticate) and authorise. Node.js comes with jwt library allows to generate token to make sure user is authorised or not “.

Footnote

- [Authentication basics](#)
- [Public Routes](#)
- [Private Routes](#)
- [User Authentication](#)
- [Session Management](#)
- [Node-uuid](#)

Assignments - Timeframe - 1 day

Assignment 1: Create node application (package) with name **Node_Authentication_Authorization**.

Assignment 2: Create folder with name **schema**.

- Create schema for userSchema in **userSchema.ts** file with schema as

```
userSchema
{
  userName(type String),
  userPassword(type String),
  userEmail(type String),
  userMobilenummer(type Number)
}
```
- Create object for userToken in **userToken .ts** file with property token to store generated token.
-

Assignment 3: Create folder with name **model**.

- Create userCollection.ts file .
- Create **userModel** for userSchema.
- Expose userModel to use externally.

Assignment 4: Create folder with name **Keys**.

Assignment 5: Create folder with name **Controller**.

- Create class user in **user.ts** file and expose it.
- Define functionality **newUser()**, here write mongoose query to create new user in database with given schema.
- Define the function **validateUser()** with user object parameter with properties **userName** and **userPassword** which return Promise. Write mongoose query to verify user is registered user or not.
- Define the function **updateUser()** with user object parameter with properties **userName** and **userPassword** which return Promise. Write mongoose query to change password of user with filter value user name.
- Define the function **removeUser()** with user object parameter with properties **userName** and **userPassword** which return Promise. Write mongoose query to remove the user from collection.

Assignment 4: Create folder with name **token**. Create module **token.ts**.

- Define class **token** with following functionality and expose it.
- Define **saveSecretKey()** functionality with string return type which returns path of folder. Write code to generate private key and public key and save in respective files as private.key.txt and public.key.txt in Keys folder. Write code to generate private key and public key using **keypair** node core module.
- Define **generateToken()** functionality as object parameter with properties **userName** and **userPassword**, string as return type. Write code to generate token using JWT node module with claim/payload and private key (read from file) and algorithm details. return generated token.
- Define functionality **verifyToken** as string parameter token, ,verifyOptions with properties payload. Write a code to verify token using JWT node module. to verify token retrieve public key from file, claim/payload details and algorithm details.

Assignment 4: create folder with name **routes**. Create module **userRoutes.ts**.

- Create Route for post method with URL pattern **"/newuser"**. Write code to call function **newUser** by passing user details from request body and return response **"User registered successfully"**.
- Create Route for post method with URL pattern **"/validateuser"**. Call function **validateUser** by passing username, password in request body. Once user gets validated generate token by calling generateToken method from token module. return response **as token** in response header.
- Create route for put request with URL pattern **"/edituser"**. call **verifyToken** function by passing token from request header. Once token gets verified call updateUser function from user module by passing object present user details from request body.
- Create route for delete request with URL pattern **"/removeuser"**. call **verifyToken** function by passing token from request header. Once token gets verified call removeUser function from user module by passing object present user details from request body.

Assignment 4: Create folder with name **routing**. Create module **startServer.ts**. Write code register routes created in userRoutes.ts module and start server.

Assignment 5 : Create folder with name **unitTest**. Write test suit for user module with name user.test.ts, Write test suite for token.test.ts for token module.

Assignment 6 : Create folder with name **apiTest**. Write test methods for all endpoint created in userRoutes.ts module.

Assignment 7 : Commit code in git and push it on Github.

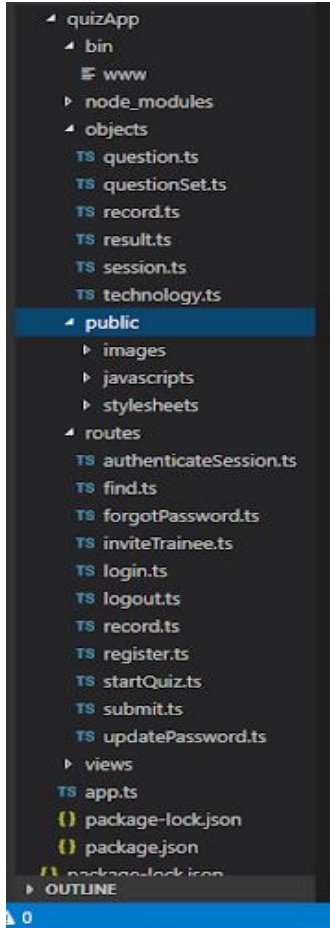
12. Mini project - Quiz App

Summary

- In this app, the admin can create a quiz test which will consist of MCQs and he can invite trainees for that test by email.
- Admin can add his/her questions with options and one correct answer to that question. Later on, he/she can use those questions to create a question set for a quiz.
- The trainee can give the test by clicking on the invitation link in the email.
- The result of the test will be calculated when the trainee submits the test.

Steps

- Install **express-generator** and set default view engine **pug** and create your **quizApp** folder **(15m)**
 - `npm install express-generator -g;`
 - `express -view = pug quizApp;`
- This will automatically create a folder structure for your quizApp. Now install the **dependencies** and check whether your server is up and running. **(15m)**
 - `npm install;`
 - `npm start;`
- Install **mongoose** and import it in your **app.ts** to connect to your local database using the **connection string**. **(15m)**
- Create **objects folder** under the quizApp and create files for each model. The files name should be the same as the name of the mongoose model followed by **.ts**. **(1h)**
- In each file **import mongoose** and define the model schema. Create the **mongoose model** using this schema except record object and **export** it. The schema should contain all the fields and validations for each field. **(2h)**
[Please refer model structure](#)
- Under routes folder create one file for each api mentioned. The name of the file should be same as the name of the api. **(30m)**
- In the api file implement the **route handler** for that api route using express router and export the router.
- Register all the routes in app.ts **(1h)**
- **Project structure for quizApp**



API Description

- **register: (1h)**
 - Route => "/"
 - Type => post
 - Body => username and password.
 - This api will accept username and password parameter in body.
 - Check if the **username is valid** (email validation) and unique then create a user record.
 - The password field should be **encrypted** i.e. **SHA256** before saving the record in database.
- **login: (30 m)**
 - Route => "/"
 - Type => post
 - Body => username and password.
 - This api will accept username and password parameter in body.
 - Encrypt the password and check if the given username and password exist in the database.
 - If exist then create the session record for that user.
 - The session should be valid for one hour.
- **authenticateSession: (30 m)**
 - Route => "/"
 - Type => get
 - cookie/queryParam /header => sid.
 - This api will accept session Id as parameter in.
 - Check if the **sid** exist in the database.
- **logout: (30 m)**
 - Route => "/"
 - Type => post
 - cookie/queryParam /header=> sid.
 - This api will accept session Id as parameter in.
 - Delete the session record using sid.
- **forgotPassword: (1h)**
 - Route => "/"

- Type => get
- QueryParam => username.
- This api will accept username as a parameter in queryParam.
- Check if the username exist in the database.
- Generate a link which will consist the the **username** and **6 digit code**
- Send this link in email to the user
- Return message “forgotten password send on mail”.

- **updatePassword: (1h)**

- Route => “/”
- Type => post
- Body => username, oldPassword, newPassword.
- This api will accept username, oldPassword and newPassword in body.
- Check if the user exist in the database using the username and oldPassword.
- If exist then the encrypt the new password and update user record
- Return message “password updated successfully”

- **questionSet: (1h)**

- Route => “/”
- Type => post
- Body => technology, noOfQuestions , username.
- This api will accept technology, noOfQuestions , username in body.
- Check if the user exist in the database using the username and retrieve ObjectId.
- Create question set .
- Return question set Id.

- **question: (1h)**

- Route => “/”
- Type => post
- Body => technology, username, question, options, answer and marks
- Check if the user exist in the database using the username and retrieve ObjectId used as OwnerId.
- Create question in question collection .
- Return message “question created successfully”.

- **inviteTrainee: (2h)**

- Route => “/”

- Type => post
- Body => sid and traineeEmail and questionSetID.
- This apis accepts sid and trainee Email and Question Set ID as parameter in body .
- Authenticate the sid and send an link on email which will consist the trainee Email and Question Set ID.
- Return message “Invitation send to Trainee/Trainees”

- **startTest: (2h)**

- Route => “/”
- Type => get
- QueryParams=> email and questionSetID.
- This api accepts trainee Email and Question Set ID as parameters in body.
- Using these parameters return a questionSet record.

- **submitTest: (2h)**

- Route => “/”
- Type => post
- Body => resultRecord
- Result record should consist the questionSetID, key value pair of question and its answer given by trainee and email of trainee .
- Calculate the marks and populate the marks field in result record .
- Insert the record into the database.
- Return message “ you have passed/fail with “ +score

Scenarios to check (5h)

- User can register using username, password
- User can login using username, password
- User can update password
- User can create question/questionSet records after signup
- User can update question/questionSet records after signup
- User can invite someone for quiz using email
- User can give a quiz test by clicking on the invitation link
- Marks are calculated after submitting

[Use case diagram and model structure](#)