UNIT – I

INTRODUCTION TO MEAN STACK DEVELOPMENT

The MEAN stack would be a web application development system based on JavaScript. MongoDB, Express, Angular, and Node are the four major components that build up the structure of the stack, and MEAN is called after them.

- MongoDB is a content database system.
- Express.js Node.js web framework
- Angular.js is a JavaScript framework for the client.
- The most popular JavaScript web host is Node(.js).

MEAN stack variants include MERN (changing Angular.js by React.js) and MEVN (trying to replace Angular.js with React.js) (using Vue.js). Among the most prominent technical paradigms for developing web apps is the MEAN stack.

MEAN Stack Developer

A MEAN stack programmer would be someone who specializes in one or more areas. The term "MEAN stack development" provides a step-by-step system that includes these components. NodeJS, MongoDB, ExpressJs, AngularJs MEAN stack programmers are similar to JS-based software developers.

A decent MEANstack developer should be well-versed in the language.

Becoming a Mean Stack Programmer is a highly sought and difficult job. As a result, if you're willing to work on a wide variety of skills and have expertise in many high-level programming languages and apps, you'll be a competent MEAN stack programmer.

Back-end, Web application, Database, Validation, and Mobile Apps developers are accessible in several gasification processes. Previously, you would seek a speciality and accomplish your

goals in that area as you gain expertise and training. The pattern has shifted due to modern best practices.

Enhancing abilities and gaining proficiency in:

- Cascading Style Sheets
- HTML (Hypertext Markup Language)
- Scripting in Java
- Universally useful languages such as Ruby, PHP, or Python.
- Database management systems, such as Postgres, Oracle, or MySQL.
- Any web server, such as Nginx or Apache
- Implementation OS: FreeBSD, Linux, or CentOS
- Any revision control system, such as GIT.

Components of MEAN Stack

The MEAN stack would be a developing trend in front-end and back-end JavaScript programming. This architecture is the only technology that can fulfil all of the most efficient implementation standards. MEAN would be an acronym that stands for the following words:

- MongoDB is abbreviated as M,
- Express is abbreviated as E,
- AngularJS is abbreviated as A,
- and NodeJS is abbreviated as N.

1. MongoDB – A JavaScript time to establish that it is open-source and cross-platform.

- Uses a JSON document that is relevant to the data model.
- A schema-free, self-contained NoSQL database
- Highly scalable design designed to process massive amounts of data that is both cost-effective and beneficial in transporting data between the server and a client
- Assists with file storage, index preparation, and bandwidth allocation.

2. Express JS –The server-side JavaScript framework is a minimalist framework.

- Reduces the amount of work involved in developing secure online and mobile apps.
- Developers can add additional innovations and developments.
- Back-end development is mostly done with this framework.
- It is easier to develop server-side apps with Nodejs.
- Database connectivity, template processors, and multiple basic routing options.

3. AngularJS – a JavaScript framework for online front-end development.

- Google maintains and builds a computer MVC JavaScript data conditional UI framework that is flexible in structure, development, and testing.
- Reduce the time it takes for dynamic and SPAs to emerge.
- For a full-stack web front-end framework, it's easy to learn and scale.

4. Node JS – a JavaScript-based execution framework that is open-source and cross-platform.

- Built using the JS V8engine from Google Chrome
- Aids in the development of feature-rich real-time web-based applications.
- Before the operation, convert JavaScript software to native machine code.
- It enables the creation of server-based applications that are both accessible and safe.
- It has a diverse ecosystem of open-source modules and features.

Working of Mean Stack

MEAN would be an open-source software stack for building cloud-hosted apps.

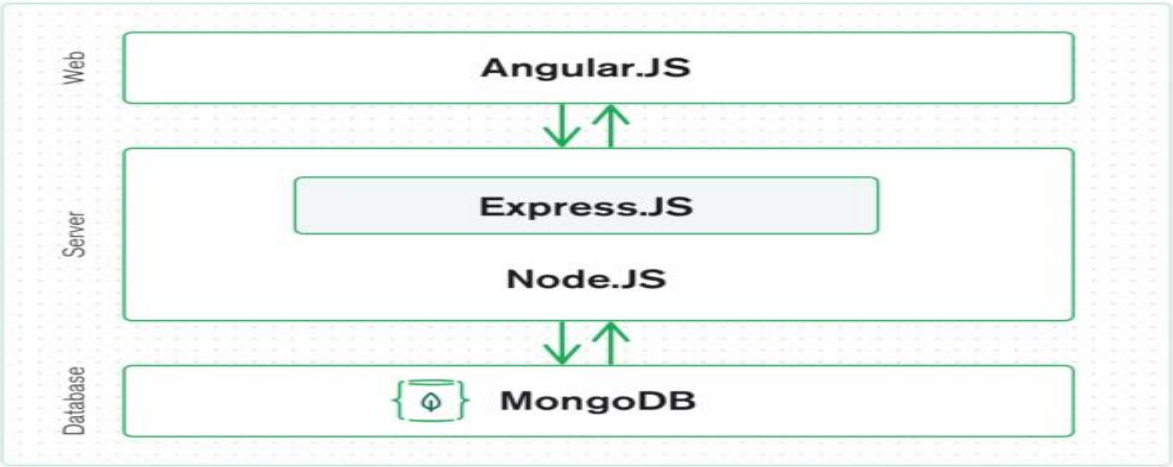
MEAN stack apps are scalable, adaptable, and versatile, making them ideal for hosting services. The stack comes with its web server, making it simple to install, and the databases could be expanded on demand to handle transient consumption surges.

A MEAN application is released into the world tailored to take full advantage of the cloud's efficiency gains and speed enhancements.

JavaScript has remained a popular front-end web development language because it is versatile, interactive, and simple. However, it's been only available for front-end and database design for a few years, enabling developers to design apps utilizing end-to-end JavaScript. You can simplify your development teams since MEAN employs the same language throughout.

MEAN eliminates the need to engage many professionals to create distinct parts of an app. Alternatively, you may leverage a unified pool of JavaScript programmers to work on-demand, wherever and whenever they are needed. Mandating on JavaScript also allows you to share code throughout the app, decreasing the amount of time you spend reinventing the wheel.

Mean Stack Architecture



MEAN.js was created to enable programmers to utilize better practices when dealing with open-source Javascript elements, construct a solid framework for meeting daily external technological, and solve frequent challenges by linking to MongoDB, Express.js, Angular.js, as well as Node.js platforms.

The MEAN stack remains incredibly basic and straightforward to do using on both the back-end and frontend. In both server-side and client execution, numerous technologies employ distinct languages. We utilize just one vocabulary for both server and client in MEAN technology.

Among the most significant advantages is that the programmer may create all of the software in JavaScript, from client and server. That would be a huge relief for Java programmers who have spent time and money learning the language for client-side jobs.

The MVC (Model – view – controller) paradigm is supported.The MEAN elements are free software, which means that the stack is changed on a constant schedule. Furthermore, it is simple to use, adaptable to comprehend, and aids developers in customizing according to their needs. Other aspects of MEAN include a large node.js component library and JSON usage to transmit data.

The following are the characteristics of the MEAN stack architecture:

- The programmer creates the complete code again by sending the computer in JavaScript, one of the essential elements of a MEAN stack design.
- The MEAN stack framework supports the Model View Controller (MVC) framework.
- The MEAN elements are independent and accessible.
- It’s simple to learn and put into practice.
- It allows developers to customize according to their needs.
- It transfers data via JSON and has a large node.js module library.

Advantages of Mean Stack

With a cross-platform start writing methodology, MEAN apps may be utilized in various ways. MEAN is best matched to real-time applications, but it can also be used for other scenarios, such as:

- Tools for managing workflows.
- News aggregation website.
- Applications for to-do lists and calendars.
- Interactive forums.

Since all of the elements are built on JavaScript and JSON, integrating the stack's elements is simple and intuitive.

Furthermore, MEAN's E and A (Express and Angular) remain two of the most widely used and well-supported Java applications for the back-end, including front programming. Express makes it simple to route and manage Html pages and answers, and it offers excellent middleware support for JSON endpoints, including form submissions. Angular is a sophisticated framework for creating interactive HTML pages that connect with a server on the back end.

Disadvantages of Mean Stack

Although JavaScript is a fantastic contemporary language, it would not be created with back-end servers in mind. Because the MEAN stack, along with the back-end server, is built on JavaScript, it may experience scalability and management standards at scale owing to JavaScript's nature.

Furthermore, because the area of project management is so quick, company and server functionality may result in insufficient segregation, resulting in software applications and improper practices.

Finally, while several manuals and courses are available, they rarely contain precise JS code standards for this framework. As a result, anything which works great in one implementation may cause problems in another.

Uses of Mean Stack

PayPal

PayPal is a sophisticated money-transfer service. It was created using HTML, Templates, and Javascript on both the application and database sides. PayPal employs two very different front-end and back-end programmers who utilize AngularJS and NodeJS.

Netflix

Netflix is a well-known subscription model that utilizes MEAN technologies. The streaming site uses AngularJS to explore a range of patterns.

The back end of Netflix was written in Java, and the front end was written in JS. However, managing many programming skills appeared to be problematic.

The Weather Channel

The Weather Channel is a television network that broadcasts weather forecasts. The Weather Channel also uses weather.com to conduct its business. The website, as well as the software, rely significantly on MEAN features to function properly.

The Weather Report also uses AngularJS for simple weather predictions and functionality. It utilizes AngularJS for multiple themes that differ based on the places and regions because it delivers worldwide predictions through the site.

Automatic Data Processing

Automatic Database Design (ADP) is a recruitment and selection system that relies heavily on huge datasets. It has more than 41000 clients and a million users.

The program keeps track of important confidential documents such as wages, health, and insurance. The ADP Mobile Applications application was built with the aid of MongoDB.

Forbes

Within two months, Forbes created a bespoke CMS based on MongoDB. Within a fortnight, Forbes additionally developed a special site.

Gmail

Google backs AngularJS as the ideal option for creating fast and scalable online apps. According to Google, developers are encouraged to adopt the Angular basis for establishing interface design. In reality, Gmail is a conventional SPA that was created with AngularJS. Gmail launches the computer for a particular HTML page and refreshes it uniquely.

YouTube

YouTube's software was available to all users on PS3 in 2012. This made it possible for users to use their cell phones to manage YouTube on their PS3. YouTube relied on Angular's framework to successfully use HTML data. YouTube programmers could utilize Angular to run flexible scripts in the user experience.

Angular made it possible for Youtube clips to play smoothly on the PS3 while preserving their performance and effectiveness.

LinkedIn

LinkedIn was founded on a Javascript on Rails process-based platform. As the firm grew, it needed a flexible solution for a sustainable app.

As a result, LinkedIn migrated to Node.js and realized its many advantages. The MEAN element aided in improving the web platform's functionality. LinkedIn serves approximately 760 million daily users and over 260 million unique visitors.

Trello

Trello is a web-based assignment and product development application. It has become an Atlassian company that provides a Kanban-style project-listing tool.

Trello's stemming is based on Node.js. Trello's team sought preconditions that were event-driven, transparent, and non-blocking.

NODE.JS

INTRODUCTION

- Node.js is an open source server environment
- Node.js is free
- Node.js runs on various platforms (Windows, Linux, Unix, Mac OS X, etc.)
- Node.js uses JavaScript on the server

Node.js files

- Node.js files contain tasks that will be executed on certain events
- A typical event is someone trying to access a port on the server
- Node.js files must be initiated on the server before having any effect
- Node.js files have extension ".js"

A common task for a web server can be to open a file on the server and return the content to the client.

PHP or ASP handles a file request:

1. Sends the task to the computer's file system.
2. Waits while the file system opens and reads the file.
3. Returns the content to the client.
4. Ready to handle the next request.

Here is how Node.js handles a file request:

1. Sends the task to the computer's file system.
2. Ready to handle the next request.
3. When the file system has opened and read the file, the server returns the content to the client.

Node.js eliminates the waiting, and simply continues with the next request.

Node.js runs single-threaded, non-blocking, asynchronous programming, which is very memory efficient.

Install Node.js

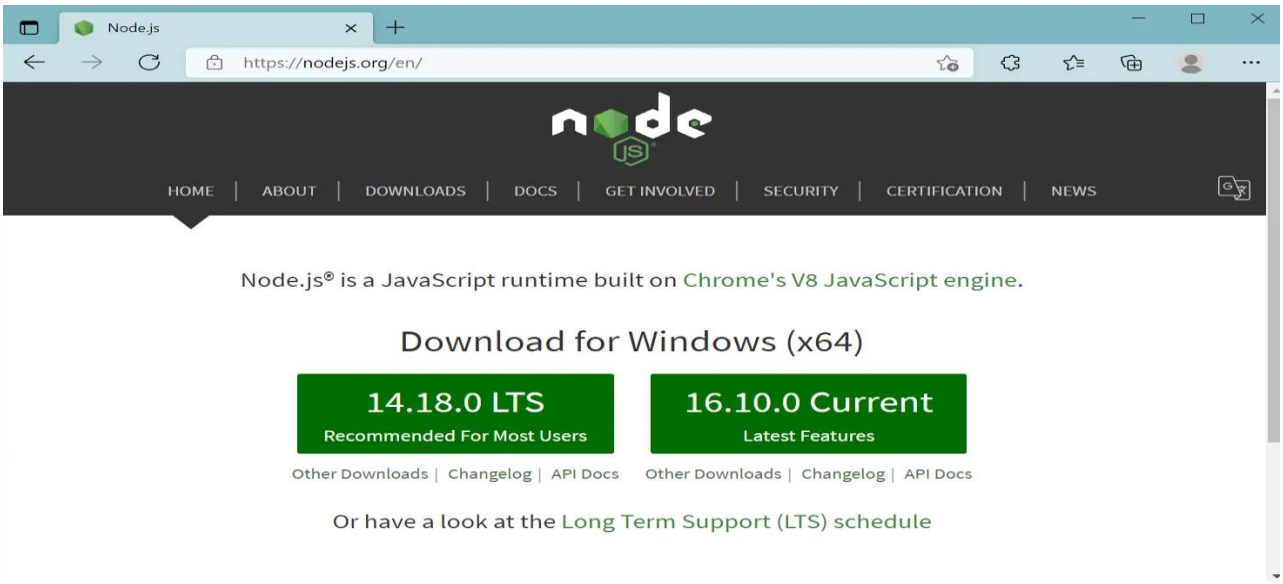
Node.js development environment can be setup in Windows, Mac, Linux and Solaris. The following tools/SDK are required for developing a Node.js application on any platform.

1. Node.js
2. Node Package Manager (NPM)
3. IDE (Integrated Development Environment) or TextEditor

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

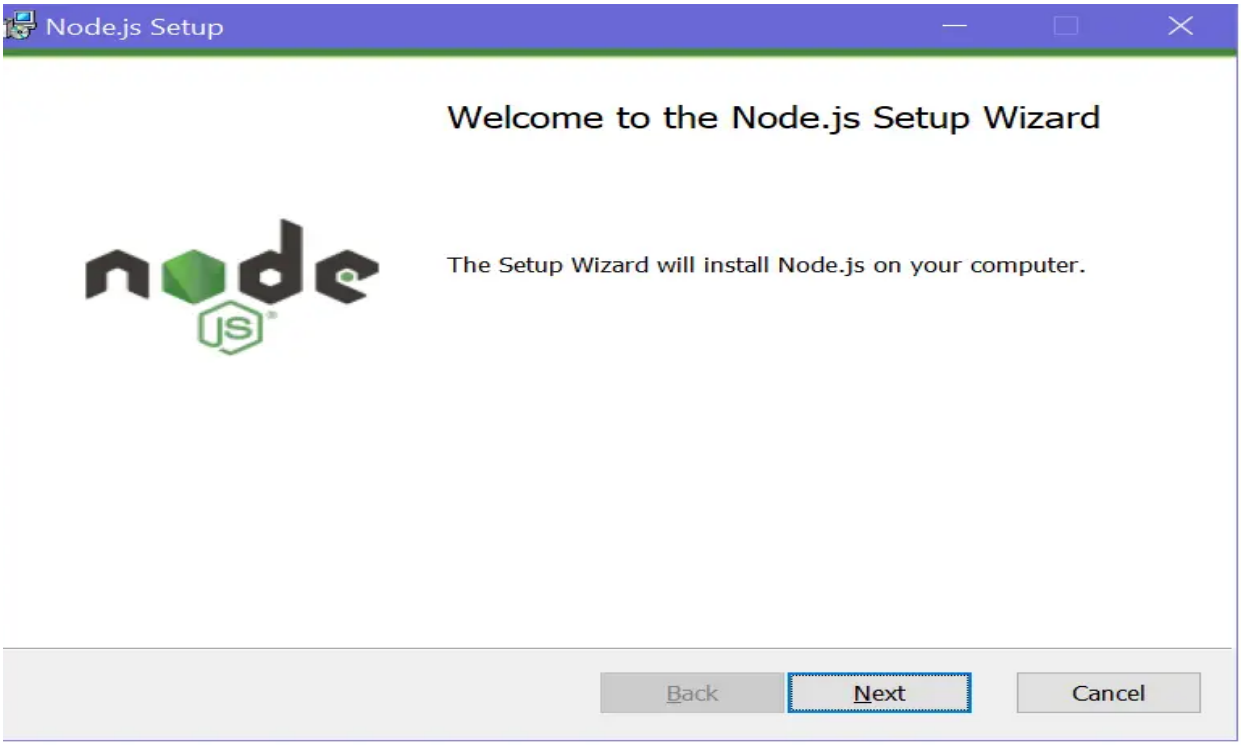
Install Node.js on Windows

Visit Node.js official web site https://nodejs.org.



Download the installer for windows by clicking on LTS or Current version button. Here, we will install the latest version LTS for windows that has long time support. However, you can also install the Current version which will have the latest features.

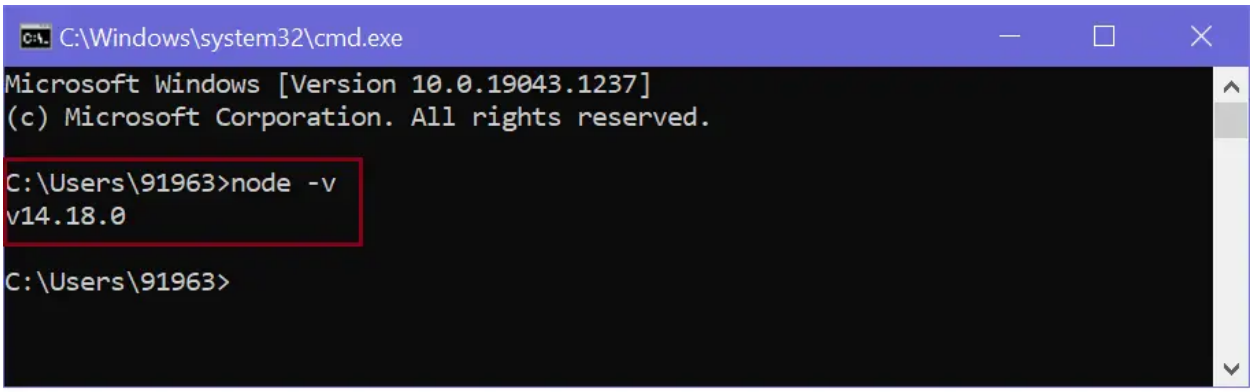
After you download the MSI, double-click on it to start the installation as shown below.



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

Verify Installation

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



Executing and Running a NODE.JS

The official Node.js website has installation instructions for Node.js: **https://nodejs.org.** Install Node.js on your computer.

Program 1: Write a Node.js application to display "Hello World" in a web browser.

Create a Node.js file named "myfirst.js".

myfirst.js

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

http.createServer(function (req, res) {

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

  res.end('Hello World!');
```

}).listen(8080);

Save the file on your computer: C:\Users*Your Name*\myfirst.js

The code tells the computer to write "Hello World!" if anyone (e.g. a web browser) tries to access your computer on port 8080.

Command Line Interface

Node.js files must be initiated in the "Command Line Interface" program of your computer.

How to open the command line interface on your computer depends on the operating system. For Windows users, press the start button and look for "Command Prompt", or simply write "cmd" in the search field.

Navigate to the folder that contains the file "myfirst.js", the command line interface window should look something like this:

```
C:\Users\Your Name>_
```

Initiate the Node.js File

The file you have just created must be initiated by Node.js before any action can take place.

Start your command line interface, write node myfirst.js and hit enter:

Initiate "myfirst.js":

C:\Users*Your Name*>node myfirst.js

Now, your computer works as a server!

If anyone tries to access your computer on port 8080, they will get a "Hello World!" message in return!

Start your internet browser, and type in the address: http://localhost:8080

ADVANTAGES OF NODE.JS

The popularity of Node.js can be attributed to a couple of stand-out advantages,

- Node.js is open-source meaning it has been made freely available to developers all over the world. Anyone is able to proposed modifications or improvements.
- Using Node.js we are able to build entire server-side applications using only JavaScript.
- The runtime environment itself is very lightweight. It includes only the core modules needed to function. Any other modules required by the application may be easily installed.
- The asynchronous nature of Node.js leads to increased performance over other frameworks.
- Node.js may be run on Windows, MAC, or Linux systems.
- Node.js can generate dynamic page content
- Node.js can create, open, read, write, delete, and close files on the server.
- Node.js can collect form data.
- Node.js can add, delete, modify data in your database

NODE.JS PROCESS MODEL

The Traditional Web Server Model

In the traditional web server model, each request is handled by a dedicated thread from the thread pool. If no thread is available in the thread pool at any point of time then the request waits till the next available thread. Dedicated thread executes a particular request and does not return to thread pool until it completes the execution and returns a response.

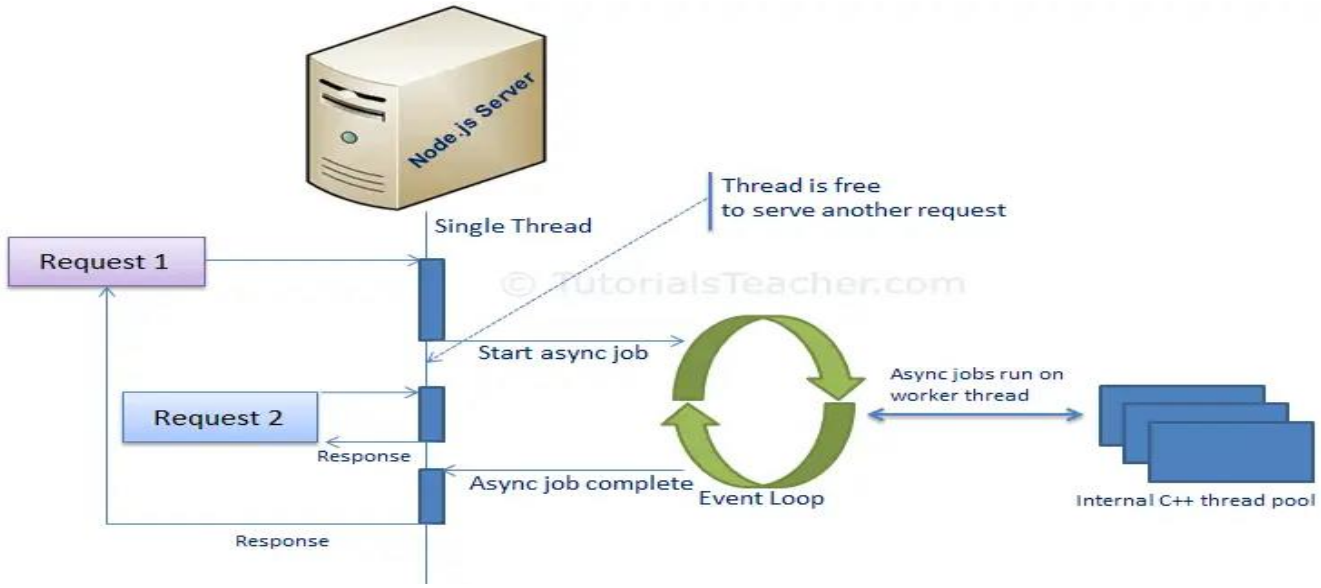
The Node.js Process Model

Node.js processes user requests differently when compared to a traditional web server model. Node.js runs in a single process and the application code runs in a single thread and thereby needs less resources than other platforms.

All the user requests to your web application will be handled by a single thread and all the I/O work or long running job is performed asynchronously for a particular request. So, this single thread doesn't have to wait for the request to complete and is free to handle the next request. When asynchronous I/O work completes then it processes the request further and sends the response.

An event loop is constantly watching for the events to be raised for an asynchronous job and executing callback function when the job completes. Internally, Node.js uses libev for the event loop which in turn uses internal C++ thread pool to provide asynchronous I/O.

The following figure illustrates asynchronous web server model using Node.js.



Node.js process model increases the performance and scalability with a few caveats. Node.js is not fit for an application which performs CPU-intensive operations like image processing or other heavy computation work because it takes time to process a request and thereby blocks the single thread.

NODE PACKAGE MANAGER

Node Package Manager (NPM) is a command line tool that installs updates or uninstalls Node.js packages in your application. It is also an online repository for open-source Node.js packages. The node community around the world creates useful modules and publishes them as packages in this repository.

NPM is a package manager for Node.js packages, or modules if you like.

www.npmjs.com hosts thousands of free packages to download and use.

The NPM program is installed on your computer when you install Node.js

Package in Node.js

A package in Node.js contains all the files you need for a module. Modules are JavaScript libraries you can include in your project.

Downloading a package is very easy.

Open the command line interface and tell NPM to download the package you want.

I want to download a package called "upper-case":

Download "upper-case":

Download "upper-case":

```
C:\Users\Your Name>npm install upper-case
```

Now you have downloaded and installed your first package.

NPM creates a folder named "node_modules", where the package will be placed. All packages you install in the future will be placed in this folder.

My project now has a folder structure like this:

C:\Users*My Name*\node_modules\upper-case

Using a Package

Once the package is installed, it is ready to use.

Include the "upper-case" package the same way you include any other module:

```
var uc = require('upper-case');
```

Program 2: Create a Node.js file that will convert the output "Hello World!" into upper-case letters.

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

var uc = require('upper-case');

http.createServer(function (req, res) {

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

  /*Use our upper-case module to upper case a string:*/

  res.write(uc.upperCase("Hello World!"));

  res.end();

}).listen(8080);
```

Save the code above in a file called "demo_uppercase.js", and initiate the file.

Initiate demo_uppercase:

```
C:\Users\Your Name>node demo_uppercase.js
```

OUTPUT

NODE.JS MODULES

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

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

Node.js Module Types

Node.js includes three types of modules:

1. Core Modules or Built-in modules
2. Local Modules
3. Third Party Modules

1) NODE.JS CORE MODULES (OR) BUILT-IN MODULES

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

a) Include Modules

To include a module, use the require() function with the name of the module:

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

Now your application has access to the HTTP module, and is able to create a server

```
http.createServer(function (req, res) {

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

```
  res.end('Hello World!');

}).listen(8080);
```

Create Your Own Modules

You can create your own modules, and easily include them in your applications. Create Node.js module that returns a date and time object

```
exports.myDateTime = function () {
  return Date(); };
```

Use the exports keyword to make properties and methods available outside the module file.

Save the code above in a file called **"myfirstmodule.js"**

Include Your Own Module

Now you can include and use the module in any of your Node.js files.

Program 3: Create a Node.js module that returns a date and time object. Use the module "myfirstmodule" in a Node.js file.

demo_module.js

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

var dt = require('./myfirstmodule');

http.createServer(function (req, res) {

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

  res.write("The date and time is currently: " + dt.myDateTime());

  res.end();
```

}).listen(8080);

Notice that we use . / to locate the module, that means that the module is located in the same folder as the Node.js file.

Save the code above in a file called "demo_module.js", and initiate the file:

Initiate demo_module.js:

C:\Users*Your Name*>node demo_module.js

OUTPUT

b) The Built-in HTTP Module

Node.js has a built-in module called HTTP, which allows Node.js to transfer data over the Hyper Text Transfer Protocol (HTTP).

To include the HTTP module, use the require() method:

var http = require('http');

Node.js as a Web Server

The HTTP module can create an HTTP server that listens to server ports and gives a response back to the client.

Use the createServer() method to create an HTTP server:

Program 4: Write a Node.js application to create an HTTP server.

demo_http.js

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

//create a server object:

http.createServer(function (req, res) {

  res.write('Hello World!'); //write a response to the client

  res.end(); //end the response

}).listen(8080); //the server object listens on port 8080
```

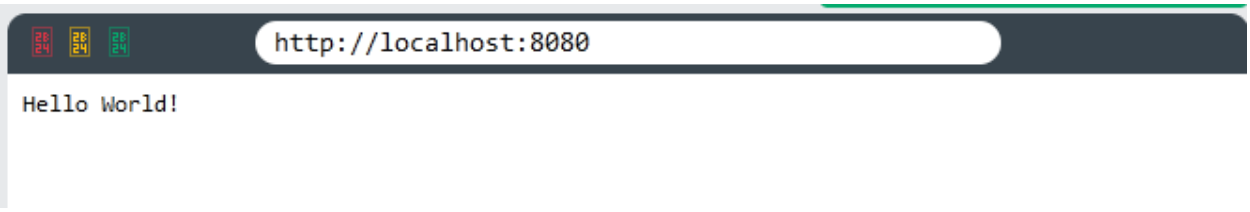
The function passed into the http.createServer() method, will be executed when someone tries to access the computer on port 8080.

Save the code above in a file called "demo_http.js", and initiate the file:

Initiate demo_http.js:

C:\Users*Your Name*>node demo_http.js

OUTPUT



Add an HTTP Header

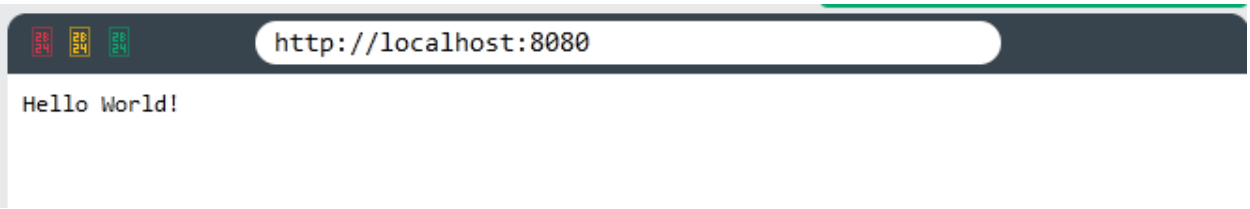
If the response from the HTTP server is supposed to be displayed as HTML, you should include an HTTP header with the correct content type.

Program 5: Write a Node.js application to add an HTTP header to the HTTP server.

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

http.createServer(function (req, res) {

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

  res.write('Hello World!');

  res.end();

}).listen(8080);
```

The first argument of the res.writeHead() method is the status code, 200 means that all is OK, the second argument is an object containing the response headers.

OUTPUT



Read the Query String

The function passed into the http.createServer() has a req argument that represents the request from the client, as an object (http.IncomingMessage object).

This object has a property called "url" which holds the part of the url that comes after the domain name.

demo_http_url.js

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

http.createServer(function (req, res) {

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

  res.write(req.url);

  res.end();

}).listen(8080);
```

Save the code above in a file called "demo_http_url.js" and initiate the file:

Initiate demo_http_url.js:

C:\Users*Your Name*>node demo_http_url.js

If you have followed the same steps on your computer, you should see two different results when opening these two addresses:

http://localhost:8080/summer

Will produce this result:

/summer

http://localhost:8080/winter

Will produce this result:

/winter

Split the Query String

There are built-in modules to easily split the query string into readable parts, such as the URL module.

Program 6: Write a Node.js application to split the query string into readable parts.

demo_querystring.js

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

var url = require('url');

http.createServer(function (req, res) {

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

  var q = url.parse(req.url, true).query;

  var txt = q.year + " " + q.month;

  res.end(txt);

}).listen(8080);
```

Initiate demo_querystring.js:

C:\Users*Your Name*>node demo_querystring.js

The address:

http://localhost:8080/?year=2017&month=July

OUTPUT

Will produce this result:

2017 July

C) Node.js File System Module

Node.js as a File Server

The Node.js file system module allows you to work with the file system on your computer.

To include the File System module, use the require() method:

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

Common use for the File System module:

- Read files
- Create files
- Update files
- Delete files
- Rename files

Read Files

The fs.readFile() method is used to read files on your computer.

Program 6: Create a Node.js file that reads the HTML file, and return the content

Assume we have the following HTML file (located in the same folder as Node.js):

demofile1.html

```
<html>
<body>
<h1>My Header</h1>
<p>My paragraph.</p>
</body>
</html>
```

demo_readfile.js

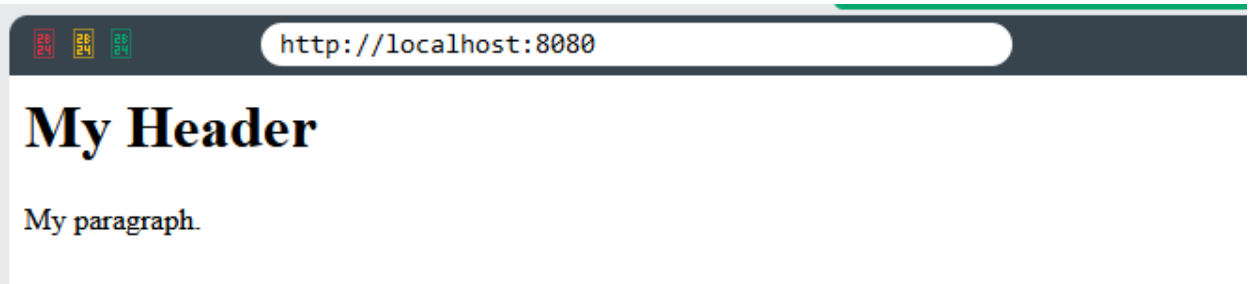
```
var http = require('http');

var fs = require('fs');

http.createServer(function (req, res) {

  //Open a file on the server and return its content:

  fs.readFile('demofile1.html', function(err, data) {

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

    res.write(data);

    return res.end();

  });

}).listen(8080);
```

Save the code above in a file called "demo_readfile.js", and initiate the file:

Initiate demo_readfile.js:

C:\Users*Your Name*>node demo_readfile.js

OUTPUT



Create Files

The File System module has methods for creating new files:

- fs.appendFile()
- fs.open()
- fs.writeFile()

The fs.appendFile() method appends specified content to a file. If the file does not exist, the file will be create

Program 7: Create a new file using the appendFile() method.

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

//create a file named mynewfile1.txt:

fs.appendFile('mynewfile1.txt', 'Hello content!', function (err) {

  if (err) throw err;

  console.log('Saved!');

});
```

OUTPUT

```
Saved!
```

The fs.open() method takes a "flag" as the second argument, if the flag is "w" for "writing", the specified file is opened for writing. If the file does not exist, an empty file is created.

Program 8: Create a new, empty file using the open() method.

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

fs.open('mynewfile2.txt', 'w', function (err, file) {

  if (err) throw err;

  console.log('Saved!');

});
```

OUTPUT

The fs.writeFile() method replaces the specified file and content if it exists. If the file does not exist, a new file, containing the specified content, will be created.

Program 9: Create a new file using the writeFile() method.

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

fs.writeFile('mynewfile3.txt', 'Hello content!', function (err) {

  if (err) throw err;

  console.log('Saved!');

});
```

OUTPUT

Update Files

The File System module has methods for updating files:

fs.appendFile()

fs.writeFile()

The fs.appendFile() method appends the specified content at the end of the specified file:

Program 10: Append "This is my text." to the end of the file "mynewfile1.txt".

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

fs.appendFile('mynewfile1.txt', ' This is my text.', function (err) {
  if (err) throw err;
  console.log('Updated!');
});
```

OUTPUT

The fs.writeFile() method replaces the specified file and content.

Program 11: Replace the content of the file "mynewfile3.txt".

```
var fs = require('fs');
fs.writeFile('mynewfile3.txt', 'This is my text', function (err) {
  if (err) throw err;
  console.log('Replaced!');
});
```

OUTPUT

```
Replaced!
```

Delete Files

To delete a file with the File System module, use the fs.unlink() method.

The fs.unlink () method deletes the specified file.

Program 12: Delete "mynewfile2.txt".

```
var fs = require('fs');
fs.unlink('mynewfile2.txt', function (err) {
  if (err) throw err;
  console.log('File deleted!');
});
```

OUTPUT

```
File deleted!
```

Rename Files

To rename a file with the File System module, use the fs.rename() method.

The fs.rename() method renames the specified file.

Program 13: Rename "mynewfile1.txt" to "myrenamedfile.txt".

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

fs.rename('mynewfile1.txt', 'myrenamedfile.txt', function (err) {

  if (err) throw err;

  console.log('File Renamed!');

});
```

OUTPUT

```
File Renamed!
```

D) The Built-in URL Module

The URL module splits up a web address into readable parts.

To include the URL module, use the require() method:

```
var url = require('url');
```

Parse an address with the url.parse() method, and it will return a URL object with each part of the address as properties.

Program 14: Split a web address into readable parts.

```
var url = require('url');

var adr = 'http://localhost:8080/default.htm?year=2017&month=february';

//Parse the address:

var q = url.parse(adr, true);

/*The parse method returns an object containing url properties*/

console.log(q.host);

console.log(q.pathname);

console.log(q.search);

/*The query property returns an object with all the querystring parameters as properties:*/

var qdata = q.query;

console.log(qdata.month);
```

OUTPUT

```
localhost:8080
/default
?year=2017&month=february
february
```

2) Local Modules

Unlike built-in and external modules, local modules are created locally in your Node.js application.

Program 15: Create a simple calculating module that calculates various operations.

calc.js

```
exports.add = function (x, y) {
   return x + y;
};

exports.sub = function (x, y) {
   return x - y;
};

exports.mult = function (x, y) {
   return x * y;
};

exports.div = function (x, y) {
   return x / y;
};
```

Since this file provides attributes to the outer world via exports, another file can use its exported functionality using the require() function.

index.js

```
const calculator = require('./calc');

let x = 50, y = 10;

console.log("Addition of 50 and 10 is "
        + calculator.add(x, y));

console.log("Subtraction of 50 and 10 is "
        + calculator.sub(x, y));

console.log("Multiplication of 50 and 10 is "
```

```
        + calculator.mult(x, y));

console.log("Division of 50 and 10 is "
        + calculator.div(x, y));
```

Step to run this program: Run the index.js file using the following command:

node index.js

Output:

Addition of 50 and 10 is 60

Subtraction of 50 and 10 is 40

Multiplication of 50 and 10 is 500

Division of 50 and 10 is 5

This module also hides functionality that is not needed outside of the module.

Third-party modules

Third-party modules are modules that are available online using the Node Package Manager(NPM). These modules can be installed in the project folder or globally. Some of the popular third-party modules are Mongoose, express, angular, and React.

Example

npm install express

npm install mongoose

npm install -g @angular/cli

ASYNCHRONUS PROGRAMMING IN NODE.JS

Asynchronous Programming is a design pattern that is extensively used in computer programming. It is particularly relevant in environments that handle numerous operations concurrently, such as Node.js.

In synchronous programming, tasks are executed one after another, meaning each task must wait for the previous one to complete before it can start. This can lead to inefficiency and wasted

resources, especially in I/O operations where the system often has to wait for data to be read from or written to a disk or over a network.

On the other hand, asynchronous programming allows tasks to be executed concurrently. This means that while one task is waiting for an I/O operation to complete, other tasks can continue to execute. This leads to a more efficient use of resources and can significantly improve the performance of your applications.

A simple comparison of synchronous and asynchronous programming

Synchronous	Asynchronous
Tasks are executed one after another	Tasks are executed concurrently
Each task must wait for the previous one to complete	While one task is waiting, other tasks can continue
Can lead to inefficiency in I/O operations	More efficient use of resources

Asynchronous programming techniques

In the context of Node.js, asynchronous programming is implemented using a few different techniques, including

1) Callbacks,

2) Promises, and

3) Async/await.

These techniques allow Node.js to handle multiple I/O operations concurrently, making it ideal for building scalable network applications.

1) Callbacks

Callbacks are a fundamental part of asynchronous programming in Node.js. They are functions that are passed as arguments to other functions and are invoked after some operation has been completed. In other words, they are a way to ensure that certain code doesn't execute until other code has already finished execution.

Program 16: Create a Node.js function that reads a file asynchronously.

```
fs.readFile('/path/to/file', function(err, data) {
  if (err) {
    console.error('Error:', err);
    return;
  }
  console.log('File data:', data);
});
```

fs.readFile is a Node.js function that reads a file asynchronously. The second argument is a callback function that is called after the file has been read. If an error occurred, it's logged to the console. Otherwise, the file data is logged.

Here's a summary of how callbacks work:

Step	Description
1	An asynchronous function is called
2	A callback function is passed as an argument
3	The asynchronous function performs some operation
4	Once the operation is complete, the callback function is invoked

While callbacks are a powerful tool for asynchronous programming, they can lead to complex and hard-to-read code when used excessively, a situation often referred to as callback hell. In the next sections, we'll explore some alternatives to callbacks, such as promises and async/await, that can help make your asynchronous code more readable and manageable.

2) Promises

A Promise is an object that represents the eventual completion or failure of an asynchronous operation. It returns a value which is either a resolved value or a reason why it's rejected. Promises have three states:

Pending: The Promise's outcome hasn't yet been determined.

Fulfilled: The Promise has completed, and a resulting value is available.

Rejected: An error occurred during the operation.

```
let promise = new Promise(function(resolve, reject) {
  // some asynchronous operation
});
promise.then(function(result) {
  // handle the result
}).catch(function(error) {
  // handle the error
});
```

Program 17: Create a Node.js application to implement Promise function which is used to simulate a delay. The 'Promise created…' message is logged immediately, and the 'Promise resolved' message is logged after the delay.

```
let promise = new Promise(function(resolve, reject) {
  // simulate a delay
  setTimeout(function() {
    resolve('Promise resolved');
  }, 2000);
});
promise.then(function(value) {
```

```
  console.log(value);
});
console.log('Promise created...');
```

3) Async/await

Async/await is a syntactic sugar on top of Promises that makes asynchronous code look and behave more like synchronous code.

```
async function asyncFunction() {
  let promise = new Promise(function(resolve, reject) {
    setTimeout(function() {
      resolve('Promise resolved');
    }, 2000);
  });
  let result = await promise;
  console.log(result);
}
asyncFunction();
console.log('Async function called...');
```

In this async/await is used to wait for a Promise to resolve. The 'Async function called…' message is logged immediately, and the 'Promise resolved' message is logged after the delay.

Promises and async/await are modern approaches to handle asynchronous operations in Node.js. They provide a cleaner and more manageable way to handle asynchronous code compared to traditional callbacks.

Promises and async/await provide a powerful way to handle asynchronous operations in Node.js. They can make your code cleaner, more readable, and easier to manage. In the next sections, we'll explore how to use these features in more detail and how to handle errors in asynchronous code.

Handling Errors in Asynchronous Code

Handling errors in asynchronous code is crucial for building robust Node.js applications. Unhandled errors can lead to unexpected behavior, making your application unreliable and hard to debug. Here are some strategies for handling errors in asynchronous code.

Callbacks

In the callback pattern, the first argument is usually reserved for an error object. If an error occurred, this object will contain details about the error. Otherwise, it will be null.

```
fs.readFile('/path/to/file', function(err, data) {
  if (err) {
    console.error('Error:', err);
    return;
  }
  console.log('File data:', data);
});
```

Promises

Promises use the .catch() method to handle errors. Any errors that occur in a promise chain will be caught by the nearest .catch() handler.

```
doSomething()
.then(result => doSomethingElse(result))
.then(newResult => doAnotherThing(newResult))
.catch(error => console.error('Error:', error));
```

Async/Await

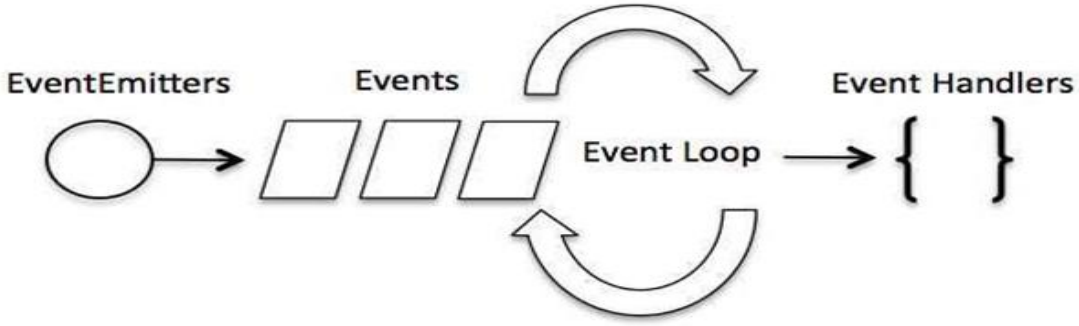
In async/await, you can use try/catch blocks to handle errors, just like in synchronous code.

```
async function asyncFunction() {
  try {
    let result = await someAsynchronousOperation();
    // handle the result
  } catch (error) {
    console.error('Error:', error);
  }}
```

EVENTS AND EVENT LOOPS IN NODE.JS

Node.js uses event driven programming. It means as soon as Node starts its server, it simply initiates its variables, declares functions and then simply waits for event to occur. It is the one of the reason why Node.js is pretty fast compared to other similar technologies.

There is a main loop in the event driven application that listens for events, and then triggers a callback function when one of those events is detected.



To bind event handler with an event:

```
1. // Bind event and even handler as follows
2. eventEmitter.on('eventName', eventHandler);
```

To fire an event:

```
1. // Fire an event
2. eventEmitter.emit('eventName');
```

Program 18: Write a NODE.JS application to implement Node.js events

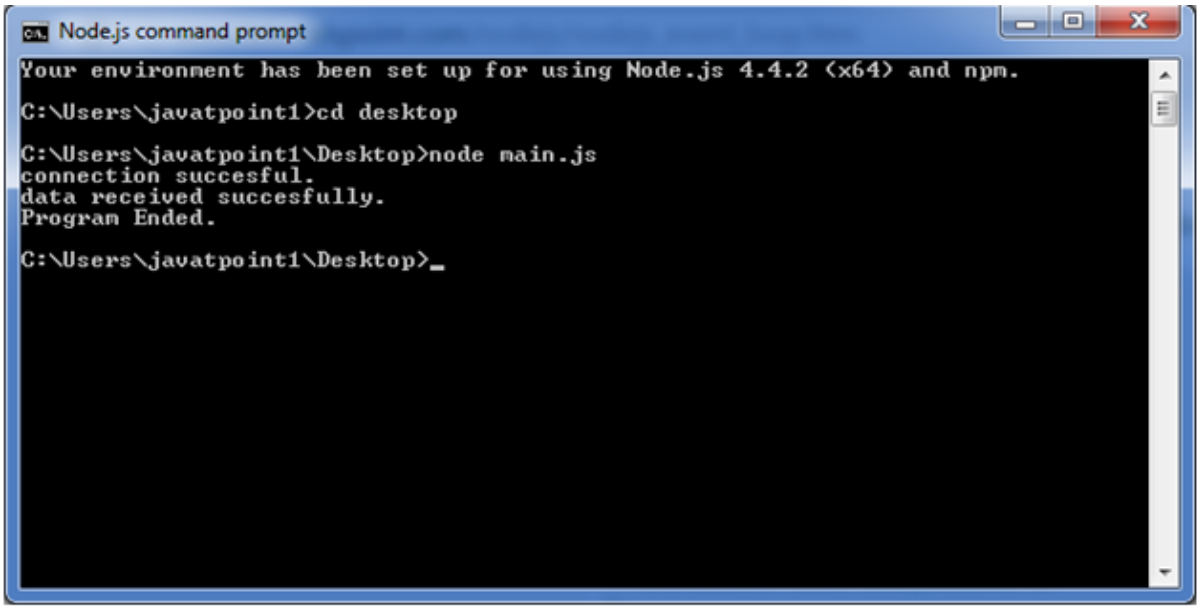
main.js

```
// Import events module
var events = require('events');
// Create an eventEmitter object
var eventEmitter = new events.EventEmitter();
// Create an event handler as follows
var connectHandler = function connected() {
  console.log('connection succesful.');
  // Fire the data_received event
```

```
    eventEmitter.emit('data_received');
}
// Bind the connection event with the handler
eventEmitter.on('connection', connectHandler);
 // Bind the data_received event with the anonymous function
eventEmitter.on('data_received', function(){
    console.log('data received succesfully.');
});
// Fire the connection event
eventEmitter.emit('connection');
console.log("Program Ended.");
```

OUTPUT

node main.js



NODE.JS EVENT LOOP

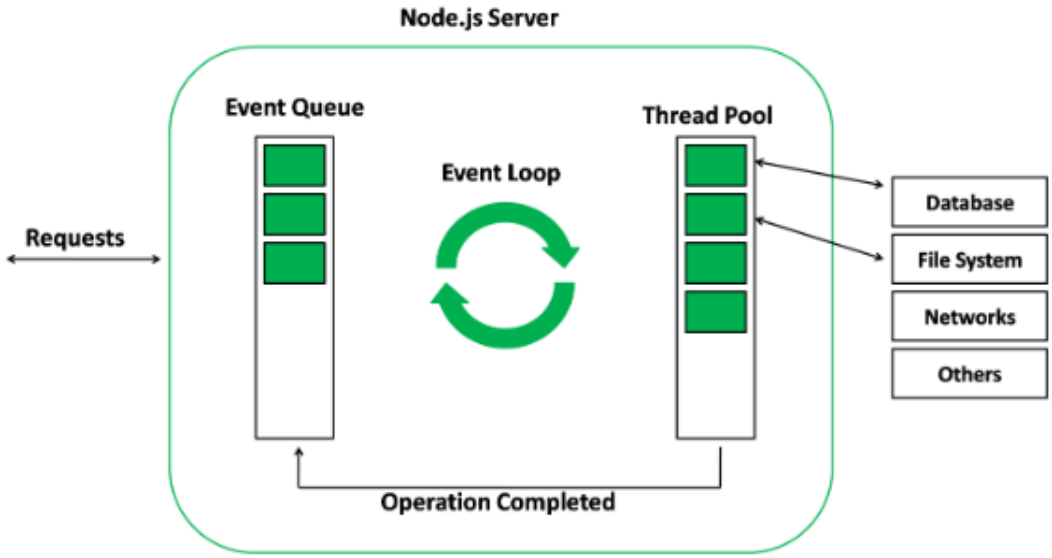
An event loop is an endless loop, which waits for tasks, executes them, and then sleeps until it receives more tasks.

The event loop executes tasks from the event queue only when the call stack is empty i.e. there is no ongoing task.

The event loop allows us to use callbacks and promises.

The event loop executes the tasks starting from the oldest first.

Working of Event Loop



When Node.js starts, it initializes the event loop, processes the provided input script which may make async API calls, schedules timers, then begins processing the event loop. In the previous example, the initial input script consisted of console.log() statements and a setTimeout() function which schedules a timer.

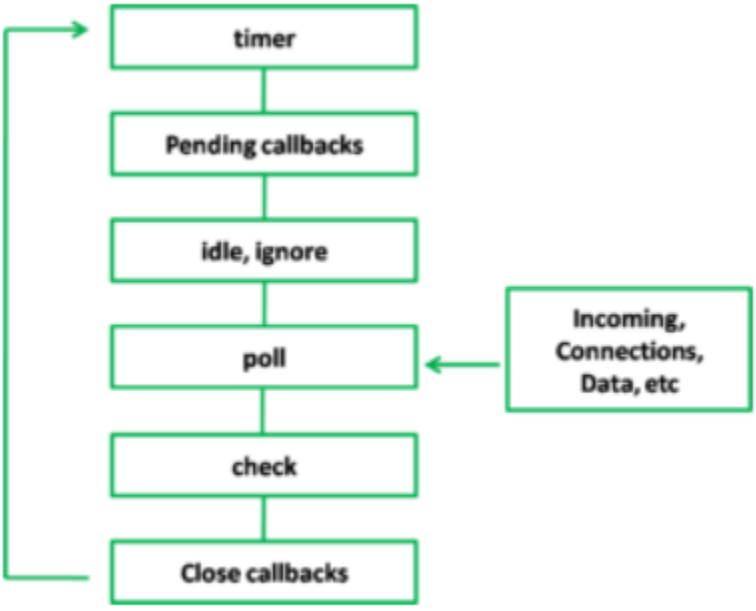
When using Node.js, a special library module called libuv is used to perform async operations. This library is also used, together with the back logic of Node, to manage a special thread pool called the libuv thread pool. This thread pool is composed of four threads used to delegate operations that are too heavy for the event loop. I/O operations, Opening and closing connections, setTimeouts are examples of such operations.

When the thread pool completes a task, a callback function is called which handles the error(if any) or does some other operation. This callback function is sent to the event queue. When the call stack is empty, the event goes through the event queue and sends the callback to the call stack.

Phases of the Event loop

The event loop in Node.js consists of several phases, each of which performs a specific task. These phases include:

The following diagram shows a simplified overview of the event loop order of operations:



1) Timers

This phase processes timers that have been set using setTimeout() and setInterval().

Program 19: Write a NODE.JS application which illustrates the working of Timers phase.

```
console.log('Start');
setTimeout(() => {
   console.log('Timeout callback');
}, 2000);
console.log('End');
```

The setTimeout() function is called with a callback that will print "Timeout callback" to the console after 2000 milliseconds (2 seconds). This function is added to the message queue in the timers phase, and the event loop will process it after the synchronous code is executed.

OUTPUT

Start

End

Timeout callback

The "Timeout callback" is printed after 2 seconds, after the "Start" and "End" are printed, because the setTimeout() function is non-blocking and its callback is processed by the event loop after the execution of the synchronous code.

2) Pending Callbacks

This phase processes any callbacks that have been added to the message queue by asynchronous functions.

Program 20: Write a NODE.JS application which illustrates the working Callbacks phase.

```
console.log('Start');

setImmediate(() => {

   console.log('Immediate callback');

});

console.log('End');
```

OUTPUT

Start

End

Immediate callback

The setImmediate() function is called with a callback that will print "Immediate callback" to the console. This function is added to the message queue in the pending callbacks phase, and the event loop will process it after the timers phase.

3) Idle, Prepare

The “idle.ignore” phase is not a standard phase of the event loop in Node.js. It means it’s Used internally only. The “idle” phase is a period of time during which the event loop has nothing to do and can be used to perform background tasks, such as running garbage collection or checking for low-priority events.

“idle.ignore” is not an official phase of the event loop, it is a way to ignore the idle phase, meaning that it will not use the time of the idle phase to perform background tasks.

```
const { idle } = require('idle-gc');

idle.ignore();
```

Here we are using the idle-gc package, which allows you to ignore the idle phase. This can be useful in situations where you want to ensure that the event loop is always busy and that background tasks are not performed.

It’s worth mentioning that, in general, the use of idle.ignore is not recommended, as it could cause performance issues, we should only use this if we have a very specific use case that requires it.

4. Poll

This phase is used to check for new I/O events and process any that have been detected.

Program 21: Write a NODE.JS application which illustrates the working of the poll phase using a readStream.

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

const readStream = fs.createReadStream('./file.txt');

console.log('Start');

readStream.on('data', (chunk) => {

   console.log(chunk.toString());

});
```

```
console.log('End');
```

A readStream is created to read the contents of a file. The ‘data’ event is added to the message queue in the poll phase, and the event loop will process it after the pending callbacks phase. The output will be the content of the file.

5) Check

This phase processes any setImmediate() callbacks that have been added to the message queue.

Program 21: Write a NODE.JS application which illustrates the working of the Check phase.

```
console.log('Start');

setImmediate(() => {

   console.log('Immediate callback');

});

console.log('End');
```

OUTPUT

```
Start

End

Immediate callback
```

The setImmediate() function is called with a callback that will print “Immediate callback” to the console. This function is added to the message queue in the check phase, and the event loop will process it after the poll phase.

6) Close Callbacks

This phase processes any callbacks that have been added to the message queue by the close event of a socket. This means that any code that needs to be executed when a socket is closed is placed in the message queue and processed during this phase.

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

const server = net.createServer((socket) => {
   socket.on('close', () => {
      console.log('Socket closed');
   });
});

server.listen(8000);
```

A server is created using the net module, and the 'close' event is added to the message queue in the close callbacks phase with a callback that will print "Socket closed" to the console. This event will be triggered when the server's socket is closed by the client and the callback will be processed by the event loop during the close callbacks phase.

It's important to note that the order of execution of these phases can vary depending on the specific implementation of the event loop, but generally, the event loop will process them in the order mentioned above.

Each phase is executed in order, and the event loop will continue to cycle through these phases until the message queue is empty.

STREAMS AND BUFFERS IN NODE.JS

STREAMS

Streams are objects that let you read data from a source or write data to a destination in continuous fashion.

In Node.js, there are four types of streams.

1) Readable − Stream which is used for read operation.

2) Writable − Stream which is used for write operation.

3) Duplex − Stream which can be used for both read and write operation.

4) Transform − A type of duplex stream where the output is computed based on input.

Each type of Stream is an EventEmitter instance and throws several events at different instance of times. Some of the commonly used events are:

- data − This event is fired when there is data is available to read.
- end − This event is fired when there is no more data to read.
- error − This event is fired when there is any error receiving or writing data.
- finish − This event is fired when all the data has been flushed to underlying system.

Reading from a Stream

Program 22: Write a Node.js application which reads the data from a stream.

Create a text file named input.txt having the following content

Tutorials Point is giving self learning content

to teach the world in simple and easy way!!!!!

main.js

```
var fs = require("fs");

var data = '';

// Create a readable stream

var readerStream = fs.createReadStream('input.txt');

// Set the encoding to be utf8.

readerStream.setEncoding('UTF8');

// Handle stream events --> data, end, and error

readerStream.on('data', function(chunk) {
```

```
   data &plus;= chunk;
});
readerStream.on('end',function() {
   console.log(data);
});
readerStream.on('error', function(err) {
   console.log(err.stack);
});
console.log("Program Ended");
```

Run the main.js to see the result

```
node main.js
```

OUTPUT

```
Program Ended
Tutorials Point is giving self learning content
to teach the world in simple and easy way!!!!!
```

Writing to a Stream

Program 23: Write a Node.js application which writes the data to the stream.

```
var fs = require("fs");
var data = 'Simply Easy Learning';
// Create a writable stream
var writerStream = fs.createWriteStream('output.txt');
// Write the data to stream with encoding to be utf8
```

```
writerStream.write(data,'UTF8');

// Mark the end of file

writerStream.end();

// Handle stream events --> finish, and error

writerStream.on('finish', function() {

   console.log("Write completed.");

});

writerStream.on('error', function(err) {

   console.log(err.stack);

});

console.log("Program Ended");
```

Run the main.js to see the result

node main.js

OUTPUT

```
Program Ended

Write completed.
```

Now open output.txt created in your current directory; it should contain the following.

Simply Easy Learning

Piping the Streams

Piping is a mechanism where we provide the output of one stream as the input to another stream. It is normally used to get data from one stream and to pass the output of that stream to another

stream. There is no limit on piping operations. Now we'll show a piping example for reading from one file and writing it to another file.

Program 24: Write a NODE.JS application to implement the Piping concept in the Streams.

main.js

```
var fs = require("fs");

// Create a readable stream

var readerStream = fs.createReadStream('input.txt');

// Create a writable stream

var writerStream = fs.createWriteStream('output.txt');

// Pipe the read and write operations

// read input.txt and write data to output.txt

readerStream.pipe(writerStream);

console.log("Program Ended");
```

Run the main.js to see the result

node main.js

OUTPUT

Program Ended

Open output.txt created in your current directory; it should contain the following:

Tutorials Point is giving self learning content

to teach the world in simple and easy way!!!!!

Chaining the Streams

Chaining is a mechanism to connect the output of one stream to another stream and create a chain of multiple stream operations. It is normally used with piping operations. Now we'll use piping and chaining to first compress a file and then decompress the same.

Program 25: Write a NODE.JS application to implement Chaining concept in Streams.

main.js

```
var fs = require("fs");

var zlib = require('zlib');

// Compress the file input.txt to input.txt.gz

fs.createReadStream('input.txt')

  .pipe(zlib.createGzip())

  .pipe(fs.createWriteStream('input.txt.gz'));

console.log("File Compressed.");
```

Run the main.js to see the result

node main.js

OUTPUT

File Compressed.

We will find that input.txt has been compressed and it created a file input.txt.gz in the current directory.

To decompress the same file

main.js

```
var fs = require("fs");

var zlib = require('zlib');

// Decompress the file input.txt.gz to input.txt

fs.createReadStream('input.txt.gz')

  .pipe(zlib.createGunzip())

  .pipe(fs.createWriteStream('input.txt'));

console.log("File Decompressed.");
```

Run the main.js to see the result

node main.js

OUTPUT

File Decompressed.

BUFFERS

Node.js provides Buffer class which provides instances to store raw data similar to an array of integers but corresponds to a raw memory allocation outside the V8 heap.

Buffer class is a global class that can be accessed in an application without importing the buffer module.

Creating Buffers

Node Buffer can be constructed in a variety of ways.

Method 1

Following is the syntax to create an uninitiated Buffer of 10 octets.

```
var buf = new Buffer(10);
```

Method 2

Following is the syntax to create a Buffer from a given array.

```
var buf = new Buffer([10, 20, 30, 40, 50]);
```

Method 3

Following is the syntax to create a Buffer from a given string and optionally encoding type.

```
var buf = new Buffer("Simply Easy Learning", "utf-8");
```

Though "utf8" is the default encoding, you can use any of the following encodings "ascii", "utf8", "utf16le", "ucs2", "base64" or "hex".

Writing to Buffers

Syntax

Following is the syntax of the method to write into a Node.js Buffer −

```
buf.write(string[, offset][, length][, encoding])
```

Parameters

Here is the description of the parameters used:

string − This is the string data to be written to buffer.

offset − This is the index of the buffer to start writing at. Default value is 0.

length − This is the number of bytes to write. Defaults to buffer.length.

encoding − Encoding to use. 'utf8' is the default encoding.

Return Value

This method returns the number of octets written. If there is not enough space in the buffer to fit the entire string, it will write a part of the string.

Example

```
buf = new Buffer(256);

len = buf.write("Simply Easy Learning");

console.log("Octets written : "+  len);
```

OUTPUT

```
Octets written: 20
```

Reading from Buffers

Following is the syntax of the method to read data from a Node Buffer:

```
buf.toString([encoding][, start][, end])
```

Parameters

Here is the description of the parameters used −

encoding − Encoding to use. 'utf8' is the default encoding.

start − Beginning index to start reading, defaults to 0.
end − End index to end reading, defaults is complete buffer.

Return Value

This method decodes and returns a string from buffer data encoded using the specified character set encoding.

Example

```
buf = new Buffer(26);
for (var i = 0 ; i < 26 ; i++) {
  buf[i] = i + 97; }
```

```
console.log( buf.toString('ascii'));       // outputs: abcdefghijklmnopqrstuvwxyz
console.log( buf.toString('ascii',0,5));   // outputs: abcde
console.log( buf.toString('utf8',0,5));    // outputs: abcde
console.log( buf.toString(undefined,0,5)); // encoding defaults to 'utf8', outputs abcde
```

OUTPUT

```
abcdefghijklmnopqrstuvwxyz

abcde

abcde

abcde
```

Convert Buffer to JSON

Syntax

Following is the syntax of the method to convert a Node Buffer into JSON object:

```
buf.toJSON()
```

Return Value

This method returns a JSON-representation of the Buffer instance.

Example

```
var buf = new Buffer('Simply Easy Learning');
var json = buf.toJSON(buf);
console.log(json);
```

OUTPUT

```
{ type: 'Buffer',
  data:
```

[

83,

105,

109,

112,

108,

121,

32,

69,

97,

115,

121,

32,

76,

101,

97,

114,

110,

105,

110,

```
    103
  ]
}
```

Concatenate Buffers

Syntax

Following is the syntax of the method to concatenate Node buffers to a single Node Buffer −

```
Buffer.concat(list[, totalLength])
```

Parameters

Here is the description of the parameters used:

- **list** − Array List of Buffer objects to be concatenated.
- **totalLength** − This is the total length of the buffers when concatenated.

Return Value

This method returns a Buffer instance.

Example

```
var buffer1 = new Buffer('TutorialsPoint ');

var buffer2 = new Buffer('Simply Easy Learning');

var buffer3 = Buffer.concat([buffer1,buffer2]);

console.log("buffer3 content: " + buffer3.toString());
```

OUTPUT

buffer3 content: TutorialsPoint Simply Easy Learning

Compare Buffers

Following is the syntax of the method to compare two Node buffers:

buf.compare(otherBuffer);

Parameters

Here is the description of the parameters used:

otherBuffer − This is the other buffer which will be compared with buf

Return Value

Returns a number indicating whether it comes before or after or is the same as the otherBuffer in sort order.

Example

```
var buffer1 = new Buffer('ABC');

var buffer2 = new Buffer('ABCD');

var result = buffer1.compare(buffer2);

if(result < 0) {

   console.log(buffer1 &plus;" comes before " &plus; buffer2);

} else if(result === 0) {

   console.log(buffer1 &plus;" is same as " &plus; buffer2);
```

```
} else {
   console.log(buffer1 &plus;" comes after " &plus; buffer2);
}
```

OUTPUT

```
ABC comes before ABCD
```

Copy Buffer

Syntax

Following is the syntax of the method to copy a node buffer −

```
buf.copy(targetBuffer[, targetStart][, sourceStart][, sourceEnd])
```

Parameters

Here is the description of the parameters used −

- **targetBuffer** − Buffer object where buffer will be copied.
- **targetStart** − Number, Optional, Default: 0
- **sourceStart** − Number, Optional, Default: 0
- **sourceEnd** − Number, Optional, Default: buffer.length

Return Value

No return value. Copies data from a region of this buffer to a region in the target buffer even if the target memory region overlaps with the source. If undefined, the targetStart and sourceStart parameters default to 0, while sourceEnd defaults to buffer.length.

Example

```
var buffer1 = new Buffer('ABC');

//copy a buffer

var buffer2 = new Buffer(3);

buffer1.copy(buffer2);

console.log("buffer2 content: " + buffer2.toString());
```

OUTPUT

```
buffer2 content: ABC
```

Slice Buffer

Syntax

Following is the syntax of the method to get a sub-buffer of a node buffer −

```
buf.slice([start][, end])
```

Parameters

Here is the description of the parameters used −

- **start** − Number, Optional, Default: 0
- **end** − Number, Optional, Default: buffer.length

Return Value

Returns a new buffer which references the same memory as the old one, but offset and cropped by the start (defaults to 0) and end (defaults to buffer.length) indexes. Negative indexes start from the end of the buffer.

Example

```
var buffer1 = new Buffer('TutorialsPoint');

//slicing a buffer

var buffer2 = buffer1.slice(0,9);

console.log("buffer2 content: " + buffer2.toString());
```

OUTPUT

```
buffer2 content: Tutorials
```

Buffer Length

Syntax

Following is the syntax of the method to get a size of a node buffer in bytes −

```
buf.length;
```

Return Value

Returns the size of a buffer in bytes.

Example

```
var buffer = new Buffer('TutorialsPoint');

//length of the buffer

console.log("buffer length: " &plus; buffer.length);
```

OUTPUT

```
buffer length: 14
```

CONNECTING NODE.JS TO DATABASE

CONNECTING NODE.JS TO A MONGODB DATABASE

MongoDB is the most popular NoSQL database. MongoDB stores data in collections. The individual records in the collections are called documents, which have a key-value structure that is like JSON data. MongoDB is preferred because of its performance, flexibility, and scalability features.

Mongoose

Mongoose is a promise-based Object Data Modeling (ODM) library for the Node.js framework. Mongoose simplifies how you interact with a MongoDB database. It allows you to create and model MongoDB schema. This way, you avoid complex writing of database queries/schemas. Mongoose gives you the ability to model the data you want to store in MongoDB.

Setting the Node.js Mongoose Project

To create a Node.js project, run npm init -y in your desired folder. Then install the following packages:

- Express - to create a Node.js HTTP server.
- Mongoose - MongoDB ODM for Node.js

npm install express mongoose

Establish Mongoose MongoDB Node.js connection

To establish a connection to MongoDB using Mongoose, create a db.js file and connect Mongoose as follows:

Import the Mongoose library:

const mongoose = require('mongoose');

Create a connectDB as follows:

```
const connectDB = async () => {
  try {
    const conn = await mongoose.connect(`mongodb://localhost:27017/test`, {
      useNewUrlParser: true,
    });
    console.log(`MongoDB Connected: {conn.connection.host}`);
  } catch (error) {
    console.error(error.message);
    process.exit(1);
  }
}
```

The MongoDB used here is running locally. Thus, Mongoose will establish a connection to mongodb://localhost:27017test, where test is your database name. Ensure you enter the correct URI that connects to either the locally installed MongoDB or cloud MongoDB Atlas.

Finally, export the connectDB function.

```
module.exports = connectDB;
```

Create a server.js file to run this function as follows:

import Express and db.js file:

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

const connectDB = require('./db');
```

Add Express middleware and parser:

```
// express

const app = express();

//body parser

app.use(express.json());
```

Execute the connectDB() function:

```
//connect to database

connectDB();
```

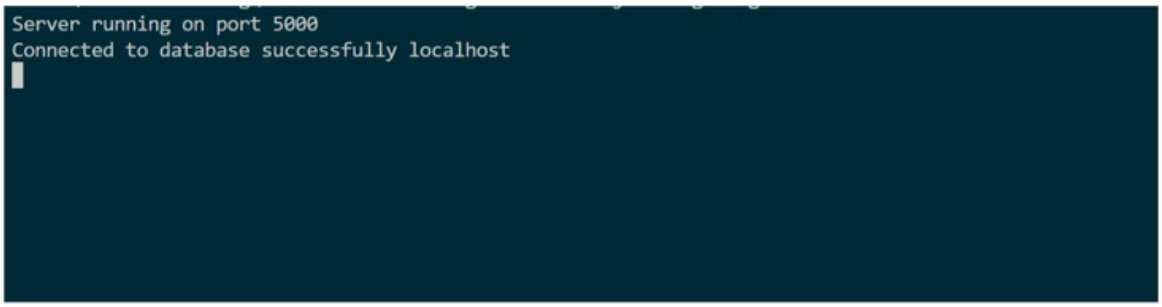
Add server routes:

```
// routes
```

Run the application on a local host port:

```
const PORT = process.env.PORT || 5000;

app.listen(PORT, () => console.log(`Server running on port {PORT}`));
```

To test this setup, run node server.js. This should log the following message if a connection to the database was established successfully.



Model a Schema Using Mongoose

A model defines a collection schema inside a MongoDB database. Create a model.js file and a Mongoose collection schema as follows:

```
const {
  Schema,
  model
} = require("mongoose");
const MySchema = new Schema({
  name: {
    type: String,
    required: true,
    maxlength: 50
  },
  createdAt: {
```

```
    type: Date,
    default: Date.now,
  },
});
const TaskModel = model("test", MySchema)
module.exports = TaskModel
```

Here, we are creating a MySchema function that executes the mongoose.Schema method. This method sets the schema that Mongoose will run to MongoDB. To create the schema, add the files that your document will have. In this case, we are adding two fields:

A name - As the above code inside, this field will be created as a string with a maxlength of fifty characters. required is set true to indicate that every document being created must have this field.

A createdAt field - Sets the time the document was created. It is executed as a date type created by default with the current date.

Add Data to the Database Using the Mongoose Schema

To add data to the database, create a controller.js file.

Import MySchema from the model file

```
const Task = require('./model');
```

Add a task to a task collection. This will define the POST route.

```
exports.createTask = async (req, res) => {
  try {
```

```
// get the task from the body
const taskData = await req.body;
//create a new task then save
await Task.create(taskData)
  .then((createdTask) => {
    if (!createdTask) return res.status(404)
      .json({
        success: false,
        message: "Task creation failed",
        error: "Unable get created task"
      })
    res.status(201)
      .json({
        success: true,
        createdTask
      })
  })
  .catch((error) => {
```

```
        res.status(404)
          .json({
            success: false,
            error: error.message
          })
      })
  } catch (error) {
    res.status(500)
      .json({
        success: false,
        message: "Internal server error"
      })
  }
}
```

Retrieve All Tasks

Add this code to the controller.js file. This will define the GET route.

```
exports.createTask = async (req, res) => {
  try {
```

```
// get the task from the body
const taskData = await req.body;
//create a new task then save
await Task.create(taskData)
  .then((createdTask) => {
    if (!createdTask) return res.status(404)
      .json({
        success: false,
        message: "Task creation failed",
        error: "Unable get created task"
      })
    res.status(201)
      .json({
        success: true,
        createdTask
      })
  })
  .catch((error) => {
```

```
        res.status(404)
          .json({
            success: false,
            error: error.message
          })
      })
  } catch (error) {
    res.status(500)
      .json({
        success: false,
        message: "Internal server error"
      })
  }
}
exports.getTasks = async (req, res) => {
  //get all the data in the model and return it as response
  try {
    Task.find()
```

```
    .then((allTasks) => {
      res.status(200)
        .json({
          success: true,
          allTasks
        })
    })
    .catch((error) => {
      res.status(404)
        .json({
          success: false,
          message: "Cant fined ",
          error
        })
    })
} catch (error) {
  res.status(500)
    .json({
```

```
        success: false,
        message: "Internal server error",
        error: error.message
      })
  }
}
```

Create Routes

Create routes to execute the above controllers. Create a new file and call it routes.js and add the following code:

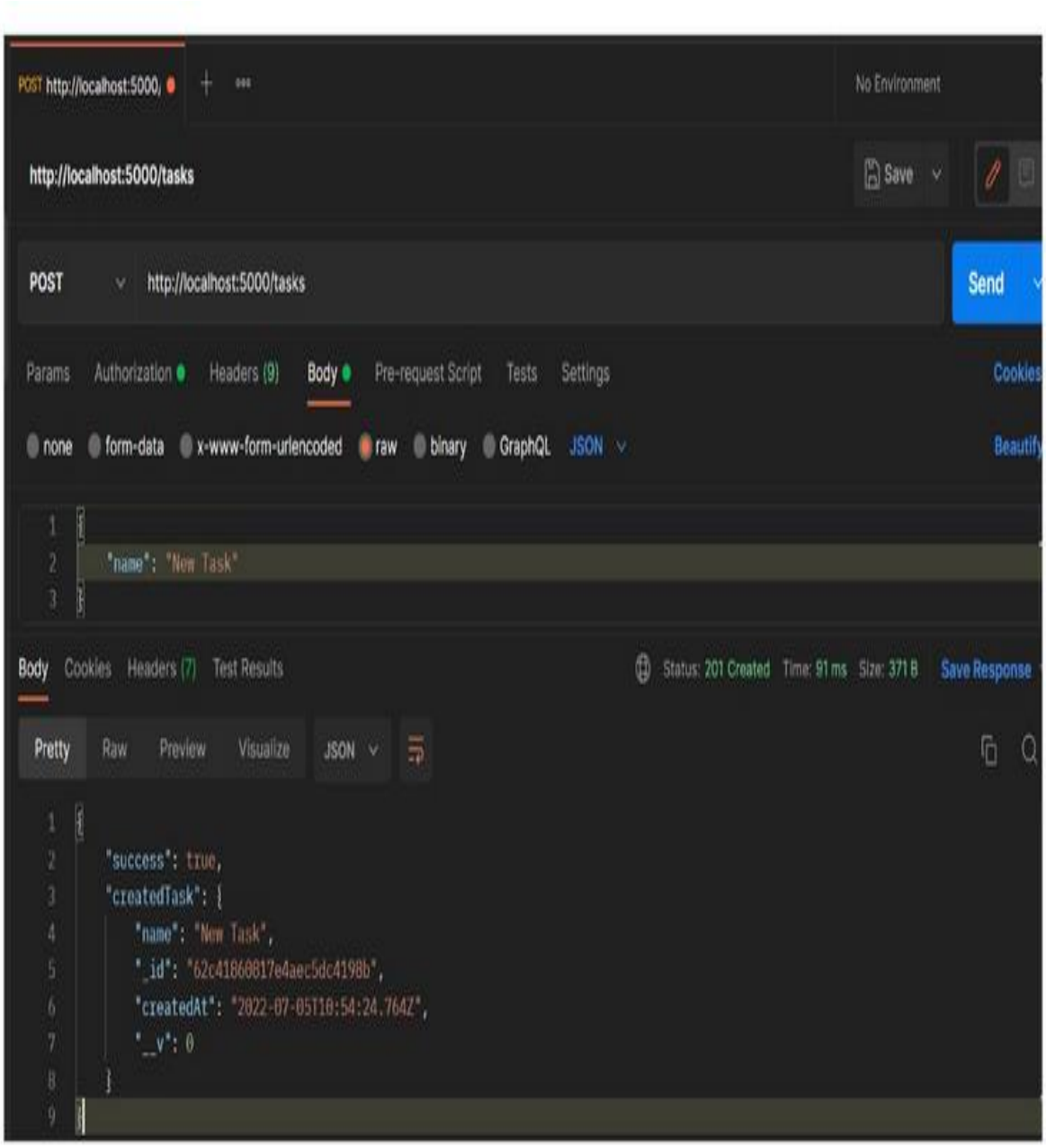
```
const router = require("express")
  .Router()
const controller = require('./controller')
router
  .post('/', controller.createTask)
  .get('/', controller.getTasks)
module.exports = router
```

Finally, execute these routes inside the server.js file.

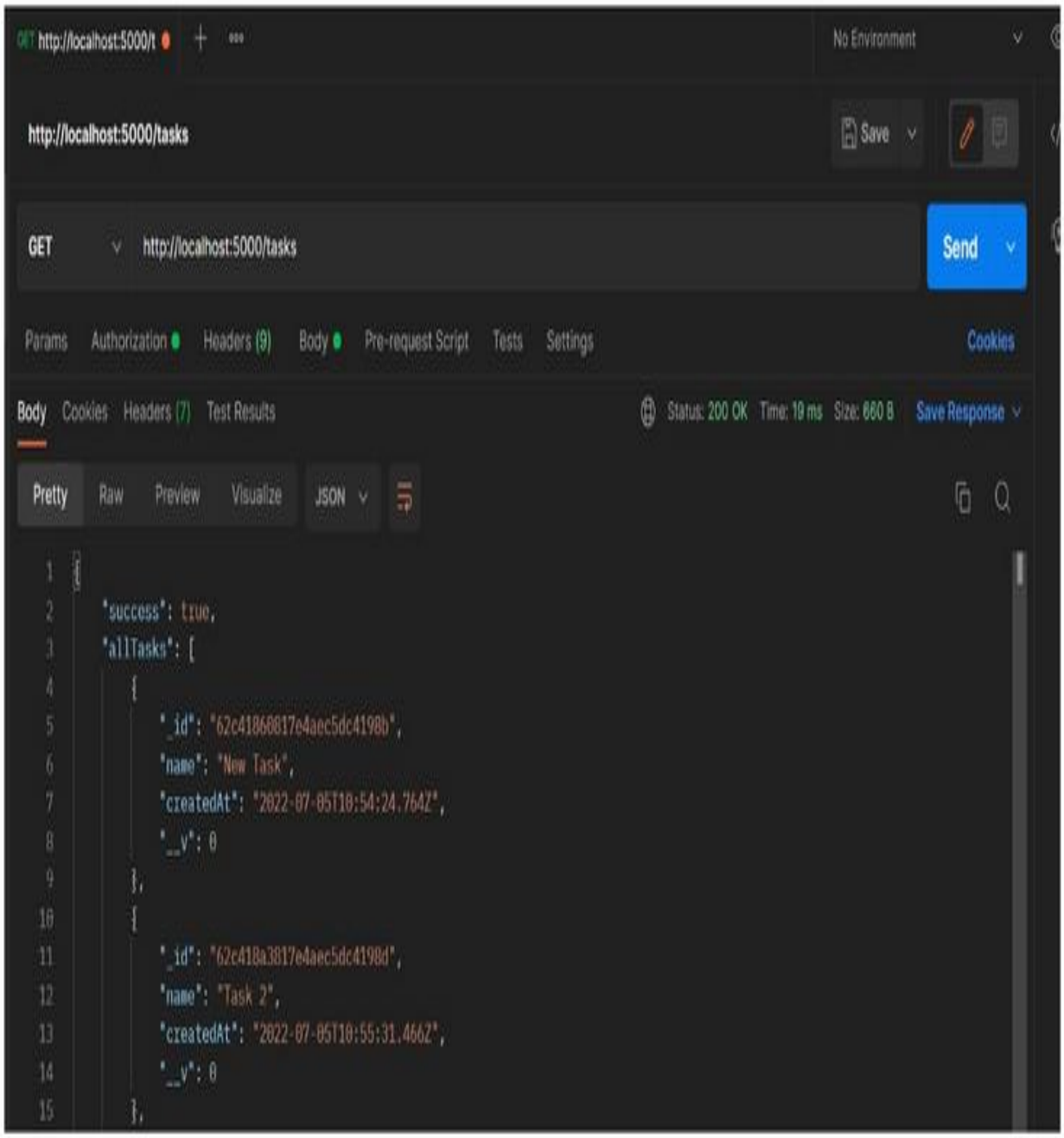
```
const router = require('./routes')
app.use('/tasks', router)
```

Testing

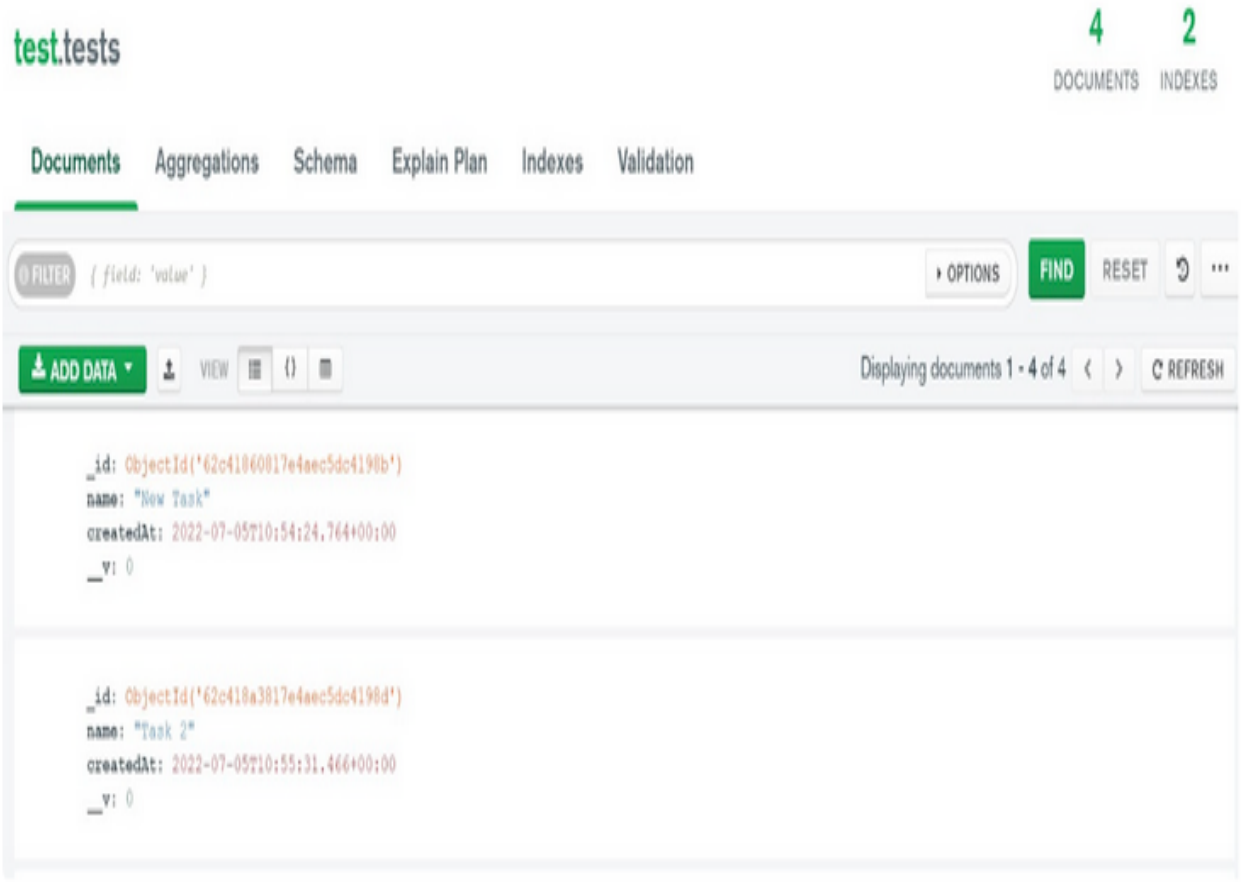
To add a new task, send a POST request using postman as follows:



To retrieve the added tasks, send a GET request using postman as follows:



You can also view the added tasks by accessing your MongoDB database:



CONNECTING NODE.JS TO A MYSQL DATABASE

Node.js can be used in database applications. One of the most popular databases is MySQL.

Once you have MySQL up and running on your computer, you can access it by using Node.js.

To access a MySQL database with Node.js, you need a MySQL driver. This tutorial will use the "mysql" module, downloaded from NPM.

To download and install the "mysql" module, open the Command Terminal and execute the following:

C:\Users\Your Name>npm install mysql

Now you have downloaded and installed a mysql database driver.

Node.js can use this module to manipulate the MySQL database:

```
var mysql = require('mysql');
```

Create Connection

Start by creating a connection to the database.

Use the username and password from your MySQL database.

Program 27: Write a NODE.JS application to connect to the MySQL database. Use the username and password from your MySQL database.

demo_db_connection.js

```
var mysql = require('mysql');

var con = mysql.createConnection({
  host: "localhost",
  user: "myusername",
  password: "mypassword"
});

con.connect(function(err) {
  if (err) throw err;
  console.log("Connected!");
```

});

OUTPUT

```
Connected!
```

Creating a Database

To create a database in MySQL, use the "CREATE DATABASE" statement.

Program 28: Write a NODE.JS application to create a database named "mydb".

demo_create_db.js

```
var mysql = require('mysql');

var con = mysql.createConnection({

  host: "localhost",

  user: "myusername",

  password: "mypassword"

});

con.connect(function(err) {

  if (err) throw err;

  console.log("Connected!");

  /*Create a database named "mydb":*/
```

```
  con.query("CREATE DATABASE mydb", function (err, result) {
    if (err) throw err;
    console.log("Database created");
  });
});
```

Run "demo_create_db.js"

C:\Users*Your Name*>node demo_create_db.js

OUTPUT

```
Connected!
Database created
```

WEB SOCKETS IN NODE.JS

A WebSocket is a computer communications protocol providing duplex communication channels over a single TCP connection. In other words, it allows internet-capable devices to communicate with each other, one acting as the client and the other acting as the server, with both able to initiate communication.

WebSockets require a persistent connection between the client and the server, which means that the server needs to be able to handle a large number of concurrent connections. Node.js uses an event-driven, non-blocking I/O model which makes it well suited to handle the resource requirements of WebSocket-based applications.

Node.js is a great choice for implementing websockets as it provides a fast and scalable server-side environment that supports the popular ws library

Implementing Web Sockets in NODE.JS

1) First, you'll need to install the "websocket" package using npm.

$ npm install websocket

2) Next we'll setup the server and client.

Example

The server will listen for connections and reply "hello" to any and everything the client sends then another message shortly after.

Create a websocket server in NodeJS

```
var server = require('websocket').server, http = require('http');

var socket = new server({
httpServer: http.createServer().listen(1337)
});

socket.on('request', function(request) {
var connection = request.accept(null, request.origin);

connection.on('message', function(message) {
console.log(message.utf8Data);
connection.sendUTF('hello');
setTimeout(function() {
connection.sendUTF('this is a websocket example');
}, 1000);
});

connection.on('close', function(connection) {
console.log('connection closed');
```

```
});

});
```

Once the server has been started, you can use the code below in any HTML5 browser that carries websocket support to establish a connection to the server. In this application the client sends a "hello" message when it opens the connection and puts anything it receives into the #content div.

```
<div id="content"></div>

<script type="text/javascript">
var content = document.getElementById('content');
var socket = new WebSocket('ws://localhost:1337');
socket.onopen = function () {
socket.send('hello from the client');
};

socket.onmessage = function (message) {
content.innerHTML += message.data +'<br />';
};

socket.onerror = function (error) {
console.log('WebSocket error: ' + error);
};

</script>
```

EXPRESS.JS

INTRODUCTION TO EXPRESS FRAMEWORK

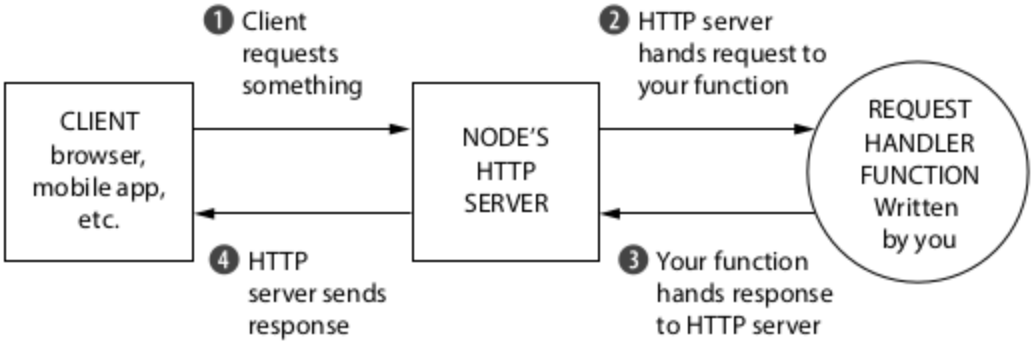
Express.js is a web application framework for Node.js. It provides various features that make web application development fast and easy which otherwise takes more time using only Node.js.

Express.js is based on the Node.js middleware module called connect which in turn uses http module. So, any middleware which is based on connect will also work with Express.js.

Express.js

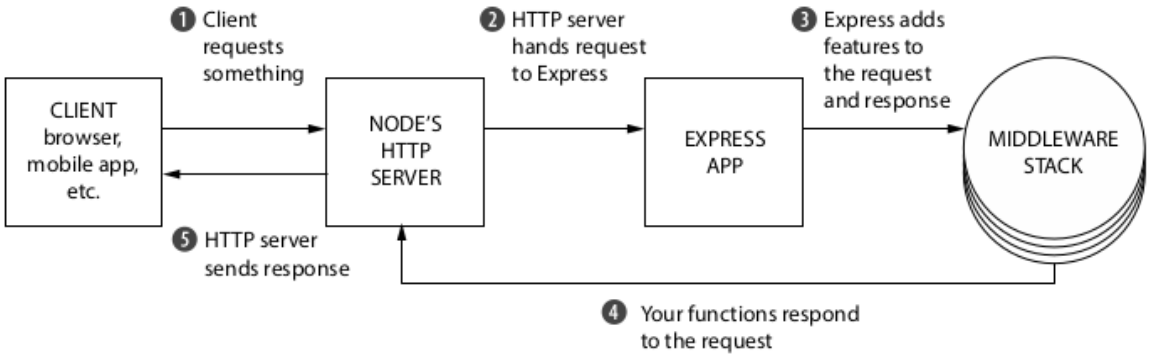
Express.js (Express) is a light web framework which sits on top of Node.js and it adds functionality like middleware**,** routing, etc. and simplicity to Node.js.

When creating a Node.js web application, we write a single JavaScript application which listens to requests from the browser, based on the request, the function will send back some data or an HTML web page.



A request handler is a JavaScript function which takes a request and sends an appropriate response.

Node.js APIs can get complex and writing how to handle a single request can end up being over 50 lines of code. Express makes it easier to write Node.js web applications.



Advantages of using Express with Node.js

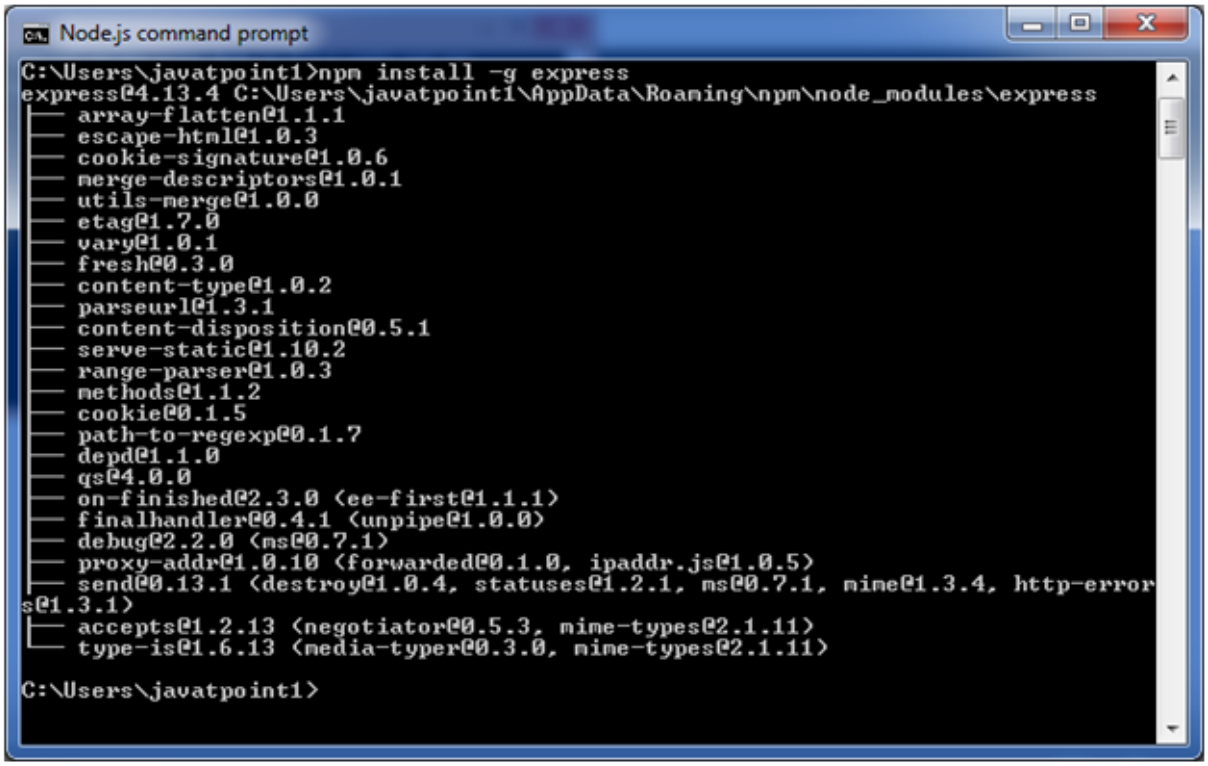
1. Makes Node.js web application development fast and easy.
2. Easy to configure and customize.
3. Allows you to define routes of your application based on HTTP methods and URLs.
4. Includes various middleware modules which you can use to perform additional tasks on request and response.
5. Easy to integrate with different template engines like Jade, Vash, EJS etc.

6. Allows you to define an error handling middleware.
7. Easy to serve static files and resources of your application.
8. Allows you to create REST API server.
9. Easy to connect with databases such as MongoDB, Redis, MySQL

Install Express.js

You can install express.js using npm. The following command will install latest version of express.js globally on your machine so that every Node.js application on your machine can use it.

```
npm install -g express
```



Installing Express

Use the following command to install express:

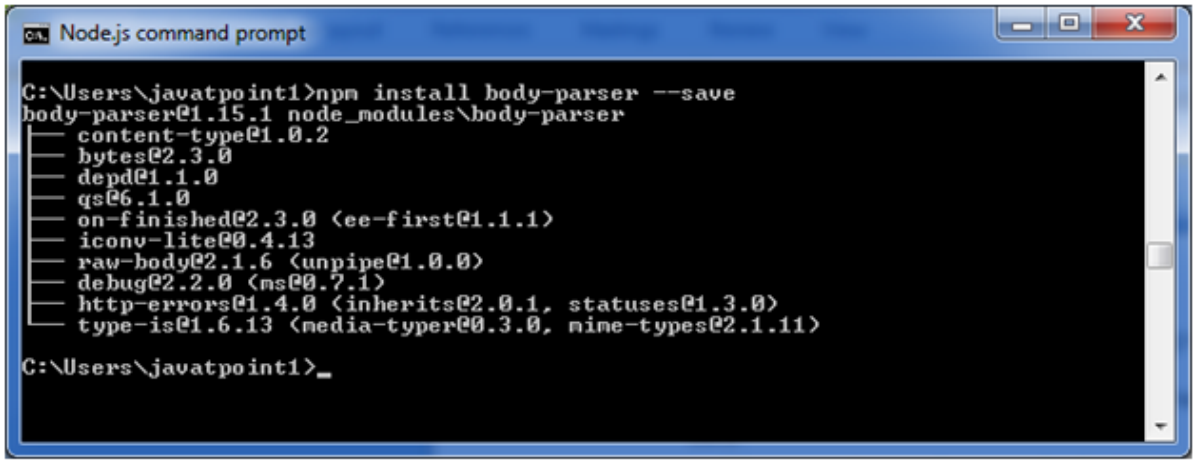
npm install express --save

As you know, --save will update the package.json file by specifying express.js dependency.

The above command install express in node_module directory and create a directory named express inside the node_module. You should install some other important modules along with express. Following is the list:

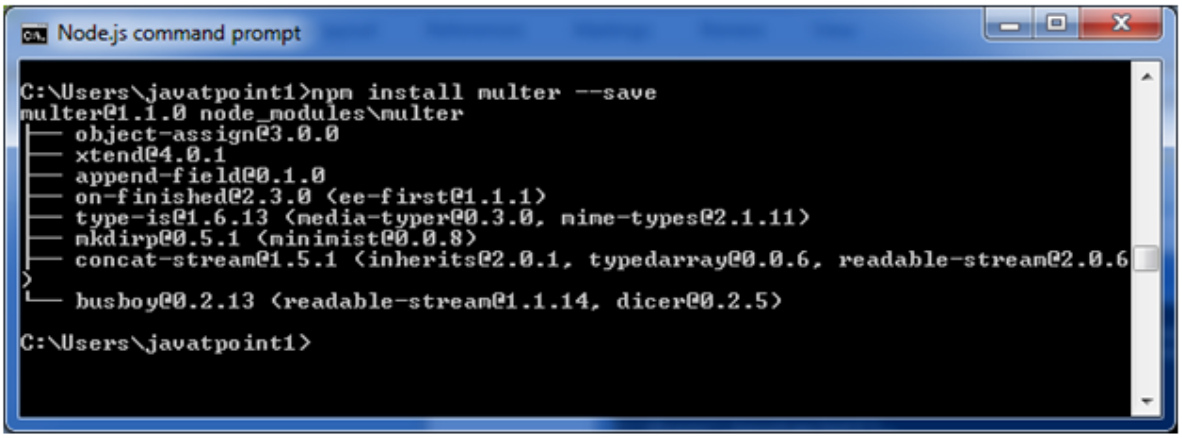
- **body-parser:** This is a node.js middleware for handling JSON, Raw, Text and URL encoded form data.
- **cookie-parser:** It is used to parse Cookie header and populate req.cookies with an object keyed by the cookie names.
- **multer:** This is a node.js middleware for handling multipart/form-data.

npm install body-parser --save



npm install cookie-parser --save

npm install multer --save



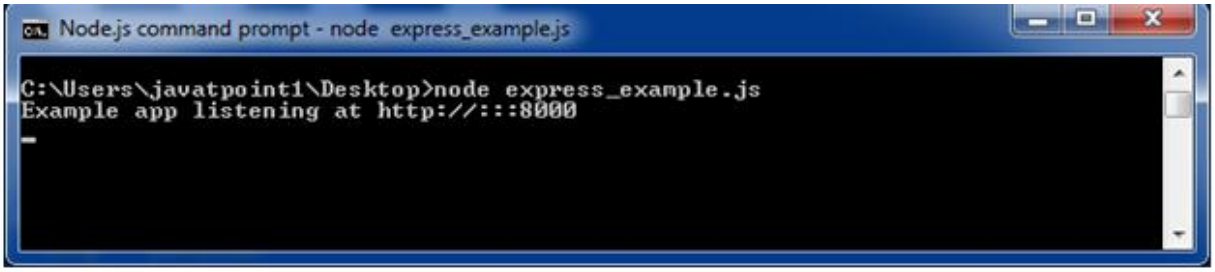
Program 1: A simple Express.js app which starts a server and listen on a local port. It only responds to homepage. For every other path, it will respond with a 404 Not Found error.

express_example.js

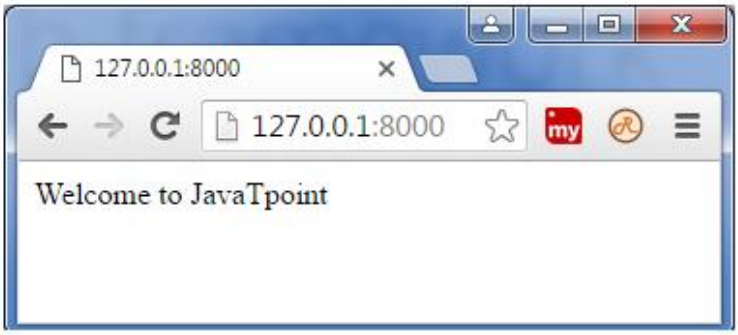
```
var express = require('express');

var app = express();

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

   res.send('Welcome to JavaTpoint!');

});
```

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

var host = server.address().address

  var port = server.address().port

 console.log("Example app listening at http://%s:%s", host, port)

})
```

OUTPUT



Open http://127.0.0.1:8000/ in your browser to see the result.



EXPRESS.JS ROUTING

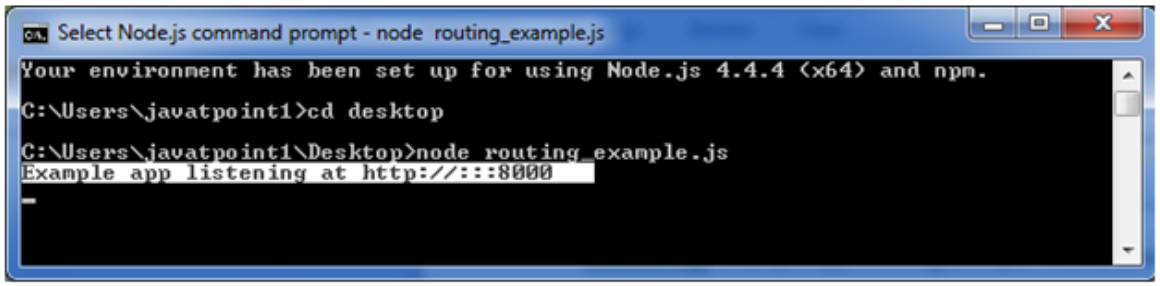
Routing is made from the word route. It is used to determine the specific behavior of an application. It specifies how an application responds to a client request to a particular route, URI or path and a specific HTTP request method (GET, POST, etc.). It can handle different types of HTTP requests.

Program 2: Write a Express.js application to implement Routing.

routing_example.js

```
var express = require('express');
var app = express();
app.get('/', function (req, res) {
  console.log("Got a GET request for the homepage");
  res.send('Welcome to JavaTpoint!');
})
app.post('/', function (req, res) {
  console.log("Got a POST request for the homepage");
  res.send('I am Impossible! ');
})
app.delete('/del_student', function (req, res) {
  console.log("Got a DELETE request for /del_student");
  res.send('I am Deleted!');
})
app.get('/enrolled_student', function (req, res) {
  console.log("Got a GET request for /enrolled_student");
```

```
  res.send('I am an enrolled student.');
})
// This responds a GET request for abcd, abxcd, ab123cd, and so on
app.get('/ab*cd', function(req, res) {
  console.log("Got a GET request for /ab*cd");
  res.send('Pattern Matched.');
})
var server = app.listen(8000, function () {
var host = server.address().address
 var port = server.address().port
console.log("Example app listening at http://%s:%s", host, port)
})
```

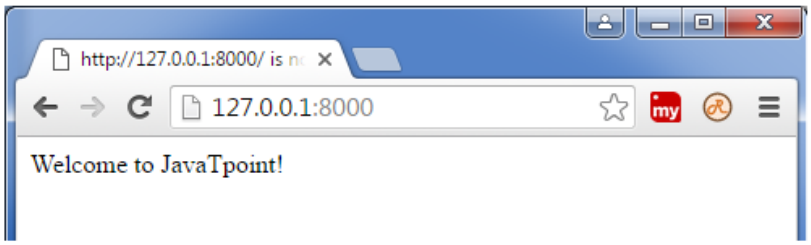


You see that server is listening.

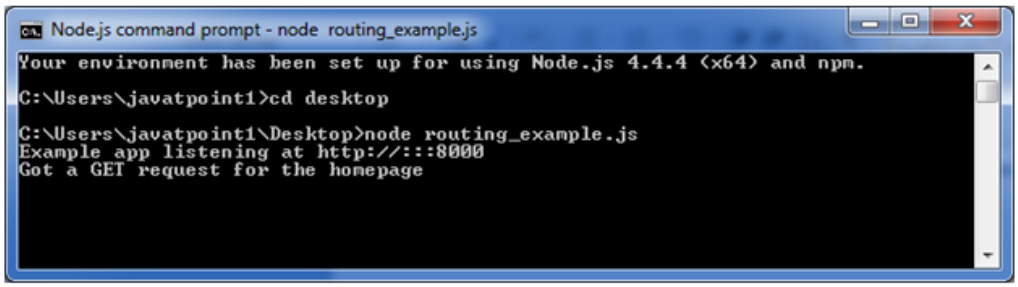
Now, you can see the result generated by server on the local host http://127.0.0.1:8000

Output:

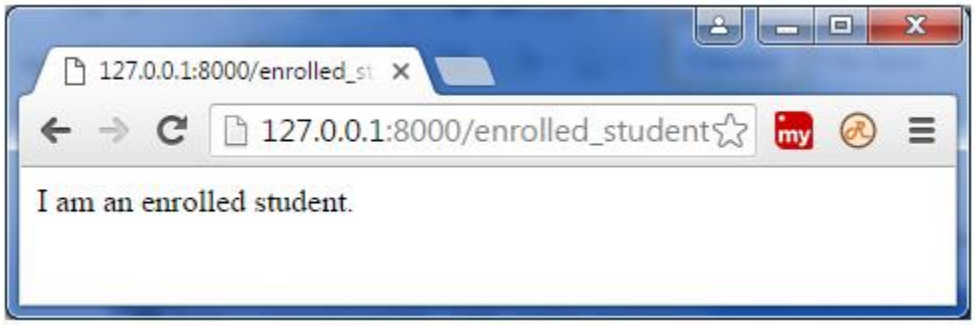
This is the homepage of the example app.



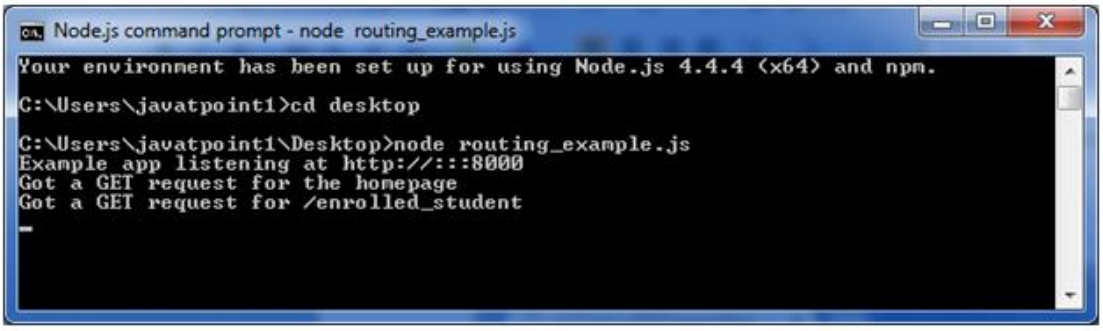
Note: The Command Prompt will be updated after one successful response.



You can see the different pages by changing routes. **http://127.0.0.1:8000/enrolled_student**

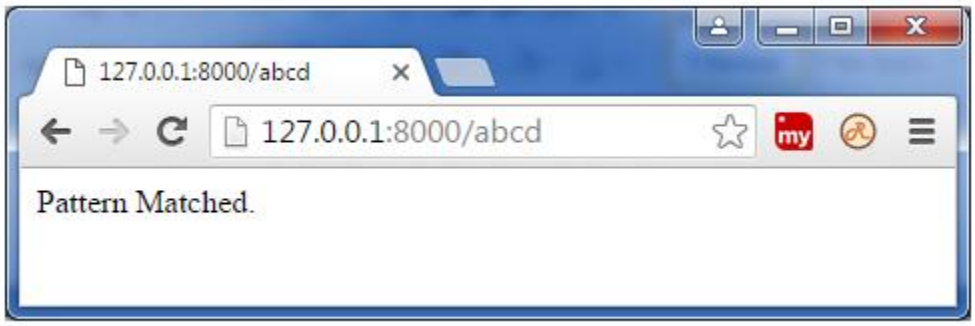


Updated command prompt

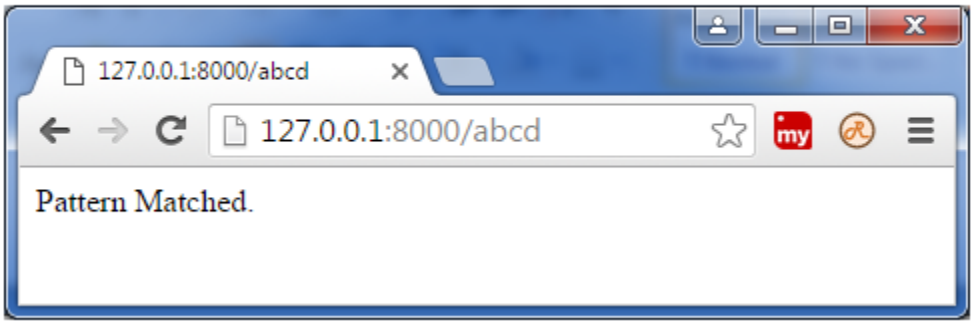


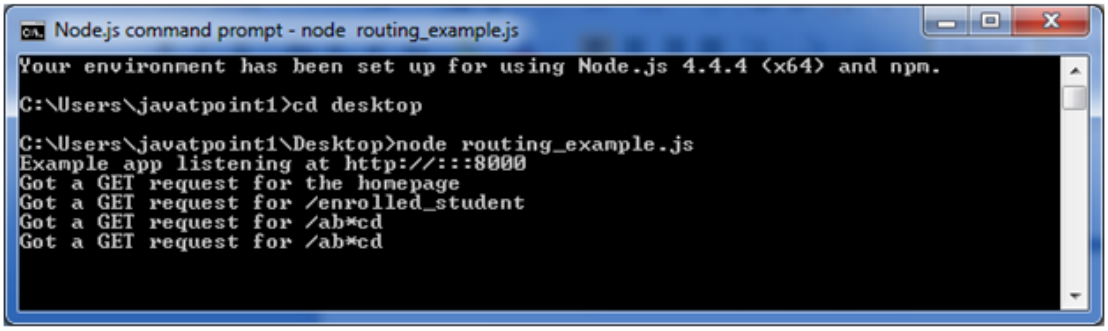
This can read the pattern like abcd, abxcd, ab123cd, and so on.

Next route http://127.0.0.1:8000/abcd



Next route http://127.0.0.1:8000/ab12345cd

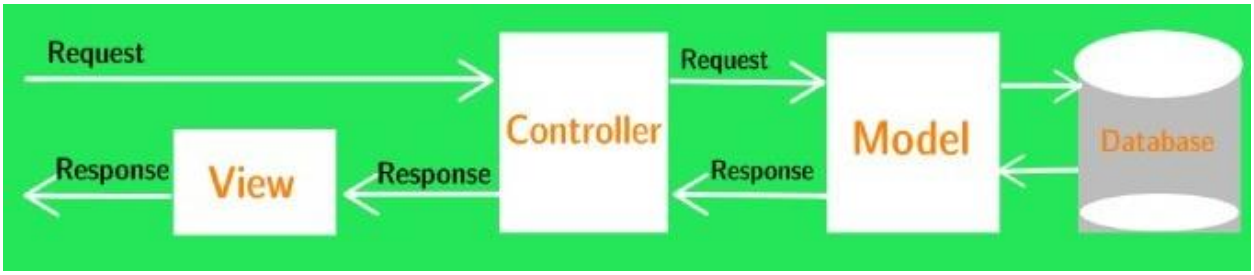




IMPLEMENTING MVC APPLICATION IN EXPRESS.JS

MVC is the most popular & useful structure for web application and it describes as

- **Model –** It can handle the database
- **View –** It can handle the client-side web pages
- **Controller –** It can control the request & response of Model & View



First of all, install Express app using the Express Generator tool. After installing it, you will get the following basic folder structure of express.

```
myapp/
  |__bin/
  |__node_modules/
  |__public/
  |__routes/
  |__views/
  |__package-lock.json
  |__package.json
```

Follow the given Steps to create the MVC structure of Express.

```
myapp/
  |__bin/
  |__controllers/
  |     |__crud-controller.js
  |__models/
  |     |__crud-models.js
  |__node_modules/
  |__public/
  |__routes/
  |     |__crud-route.js
  |__views/
  |     |__crud-operation.ejs
  |__package-lock.json
  |__package.json
```

Express – Model

Model

In this folder, you can write the functionality & logics related to the Database like insert, fetch, update, delete queries. Even It takes the query request from the controller & sends the response back to the controller.

You can create a model in the myapp application through the following steps

- Create a folder models in the the myapp application.
- Create a file crud-model.js in the models. Even you can create more controller files.
- Define some functionality in the crud-model.js as the following script

File Path- models/crud-model.js

File Name – crud-model.js

```
module.exports={
  createCrud:function(){
       data="Form data was inserted";
       return data;
  },
  fetchCrud:function(){
    data="data was fetched";
    return data;
  },
  editCrud:function(editData){
    data= "Data is edited by id: "+editData;
    return data;
  },
  UpdateCrud:function(updateId){
    data= "Data was updated by id: "+updateId;
    return data;
  },
  deleteCrud:function(deleteId){
    data= "Data was deleted by id: "+deleteId;
    return data;
  }
}
```

Include model file **crud-model.js** in the controller file **crud-controller.js** of controllers using the following script.

```
var crudModel=require('../models/crud-model');
```

Express – View

View

In this folder, you can write HTML code for displaying a web page on the web browser. Even you can send the data from the controller to view for displaying data dynamically.

The view will be generated with the Basic Structure of Express App and It contains views folder

- You can create web pages in the views folder through the following steps
- Create a file crud-operation.ejs in the views folder.
- Write the HTML code as the following script. Even you can create another HTML file & write HTML code based on your project requirement.

File Name – crud-operation.ejs

```
<!DOCTYPE html>

<html>

<head> <title>CRUD Operation</title>

<link rel='stylesheet' href='/stylesheets/style.css' />

<style>

table, td, th {

    border: 1px solid #ddd;

   text-align: left;}

table {

border-collapse: collapse;
```

```
width: 50%;}
.table-data{
position: relative;
 left:150px;
top:100px;}
th, td {
padding: 15px;}
</style>
</head>
<body>
    <% if(typeof editData!='undefined'){ %>
          <h1><%= editData %></h1>
         <form method="POST" action="/crud/edit/<%=editId %>">
              <input type="submit" value="Update Data">
          </form>
  <% } else{ %>
      <h1>Crud Operation</h1>
      <h3>This is View Page</h3>
```

```
 <h4>Create Data</h4>

<form method="POST" action="/crud/create">

      <input type="submit" value="Create Data">

 </form>

 <% } %>

<br><br> <br><br>

<table border="1" >

  <tr>

   <th><a href="/crud/form">Crud Form</a></th>

   <th><a href="/crud/fetch">Fetch Data</a></th>

    <th><a href="/crud/edit/5">Edit Data</a></th>

    <th><a href="/crud/delete/5">Delete Data</a></th>

    </tr>

    </table>

</body>

</html>
```

You can load a view file crud-operation.ejs in the controller file crud-controller.js of the controllers' folder using the following script.

```
res.render('crud-operation');
```

Express – Controller

Controller - In this folder, you can write the functionality & logic to develop dynamic web applications. Even it takes the data request from the views & sends it to the model and sends the response back to the views.

You can create a model in the myapp application through the following steps

- Create a folder controllers in the the myapp application.
- Create a file crud-controller.js in the controllers. Even you can create more controller files.
- Define some functionality in the crud-controller.js as the following script

Path – controllers/crud-controller.js

File Name – crud-controller.js

```
var crudModel=require('../models/crud-model');

module.exports={

crudForm:function(req, res) {

res.render('crud-operation');

},

createCrud:function(req,res){

const createData=crudModel.createCrud();

res.send('<h1>'+createData+'</h1>');

},

fetchCrud:function(req,res){
```

```
const fetchData=crudModel.fetchCrud();
res.send('<h1>'+fetchData+'</h1>');
},
editCrud:function(req,res){
const editId=req.params.id;
const editData= crudModel.editCrud(editId);
res.render('crud-operation',{editData:editData,editId:editId});
},
UpdateCrud:function(req,res){
const updateId=req.params.id;
const updateData= crudModel.UpdateCrud(updateId);
res.send('<h1>'+updateData+'</h1>');
},
deleteCrud:function(req,res){
const deleteId=req.params.id;
const deleteData= crudModel.deleteCrud(deleteId);
res.send('<h1>'+deleteData+'</h1>');
}
}
```

Include model file crud-controller.js in the controller file crud-route.js of Route folder using the following script.

```
var crudController=require('../controllers/crud-controller');
```

Express – Route

Route – In the route folder, you can create a custom route/link to execute the dynamic web pages.

The Route will be generated with the Basic Structure of Express App and It contains routes folder

- You can create routes in the routes folder through the following steps
- Create a file crud-route.js in the routes folder.
- Define some functionality in the crud-route.js as the following script

Path – routes/crud-route.js

File Name – crud-route.js

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

var crudController=require('../controllers/crud-controller');

var router = express.Router();

// curd form route

router.get('/form', crudController.crudForm );

// create data route

router.post('/create', crudController.createCrud);

// display data route
```

```
router.get('/fetch', crudController.fetchCrud);

// edit data route

router.get('/edit/:id', crudController.editCrud);

// update data route

router.post('/edit/:id', crudController.UpdateCrud);

// delete data route

router.get('/delete/:id', crudController.deleteCrud);

module.exports = router;
```

You have to load a route file crud-route.ejs in the root file app.js of the myapp app using the following script.

```
var crudRouter = require('./routes/crud-route');

app.use('/crud', crudRouter);
```

Start the Node.js server & run the following route

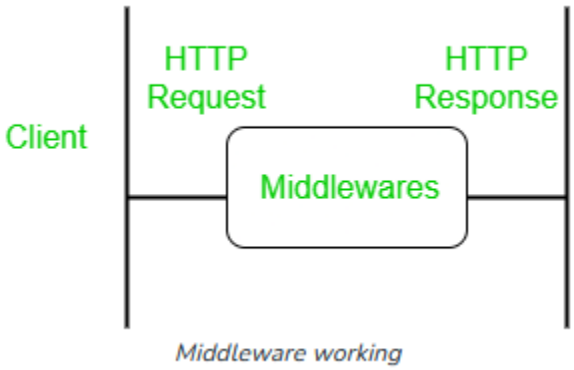
http://localhost:3000/crud/form

MIDDLEWARE IN EXPRESS.JS

Express.js is a routing and Middleware framework for handling the different routing of the webpage and it works between the request and response cycle.

Middleware gets executed after the server receives the request and before the controller actions send the response.

Middleware has the access to the request object, responses object, and next, it can process the request before the server send a response. An Express-based application is a series of middleware function calls.



Middleware working

Advantages of using middleware

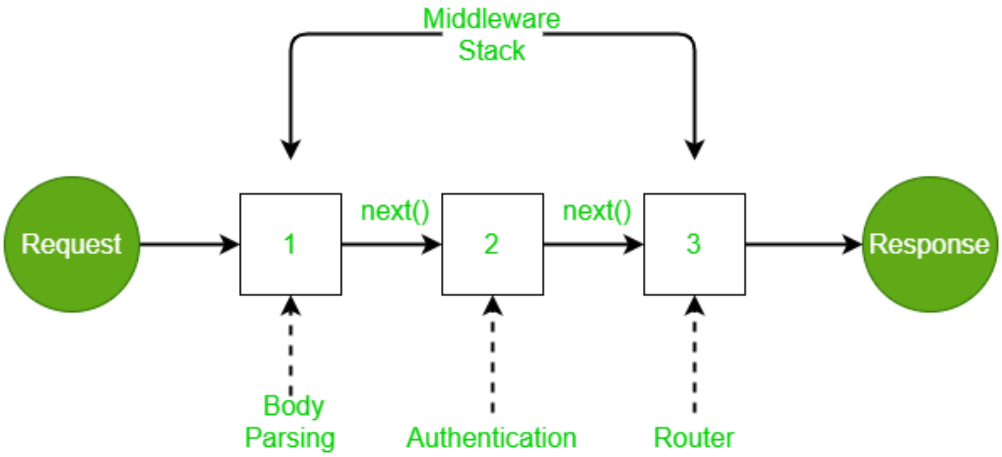
- Middleware can process request objects multiple times before the server works for that request.
- Middleware can be used to add logging and authentication functionality.
- Middleware improves client-side rendering performance.
- Middleware is used for setting some specific HTTP headers.
- Middleware helps for Optimization and better performance.

Middleware Chaining

Middleware can be chained from one to another, Hence creating a chain of functions that are executed in order. The last function sends the response back to the browser. So, before sending the response back to the browser the different middleware process the request.

The next() function in the express is responsible for calling the next middleware function if there is one.

Modified requests will be available to each middleware via the next function.



In the above case, the incoming request is modified and various operations are performed using several middlewares, and middleware is chained using the next function. The router sends the response back to the browser.

Middleware Syntax

The basic syntax for the middleware functions are as follows –

app.get(path, (req, res, next) => {}, (req, res) => {})

Middleware functions take 3 arguments: the request object, the response object, and the next function in the application's request-response cycle, i.e., two objects and one function.

Middleware functions execute some code that can have side effects on the app, and usually add information to the request or response objects. They are also capable of ending the cycle by sending a response when some condition is satisfied. If they don't send the response when they are done, they start the execution of the **next function** in the stack. This triggers calling the 3rd argument, next().

The middle part **(req,res,next)=>{}** is the middleware function. Here we generally perform the actions required before the user is allowed to view the webpage or call the data and many other functions. So let us create our own middleware and see its uses.

Create our middleware and see that how it executes.

Step 1: Go to your project directory and enter the following command to create a NodeJs project. Make sure that NodeJs is installed in your machine.

npm init -y

It will create a package.json file.

Step 2: Install two dependencies using the following command.

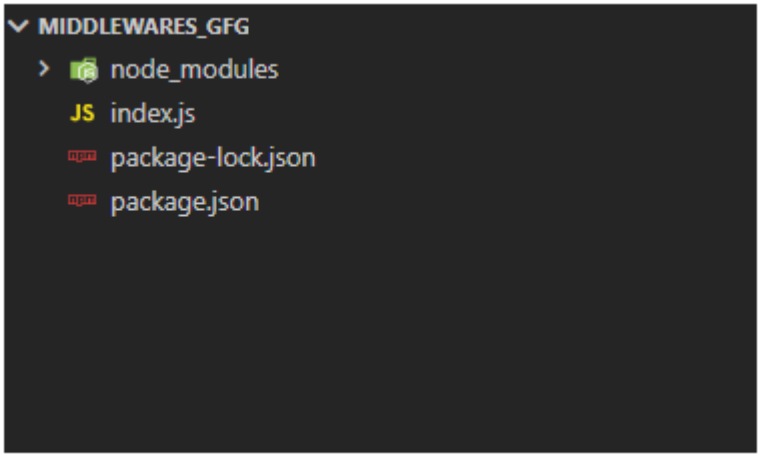
npm install express nodemon

Step 3: In the scripts section of the package.json file, add the following code line.

"start": "nodemon index.js",

Step 4: Create an index.js file in the directory. Make sure that it is not inside any subdirectories of the directory you are working in.

Project Structure: It will look like the following.



Project Structure

Now we will set up our express app and send a response to our **server**.

Here is the code for the index.js file.

index.js

```
const express = require("express");
const app = express();


const port = process.env.port || 3000;
app.get("/", (req, res) => {
  res.send(`<div>
    <h2>Welcome to GeeksforGeeks</h2>
    <h5>Tutorial on Middleware</h5>
  </div>`);
});
app.listen(port, () => {
  console.log(`Listening to port ${port}`);
});
```

Step to run the application

Run the code by entering the following command on the terminal.

npm start

Output

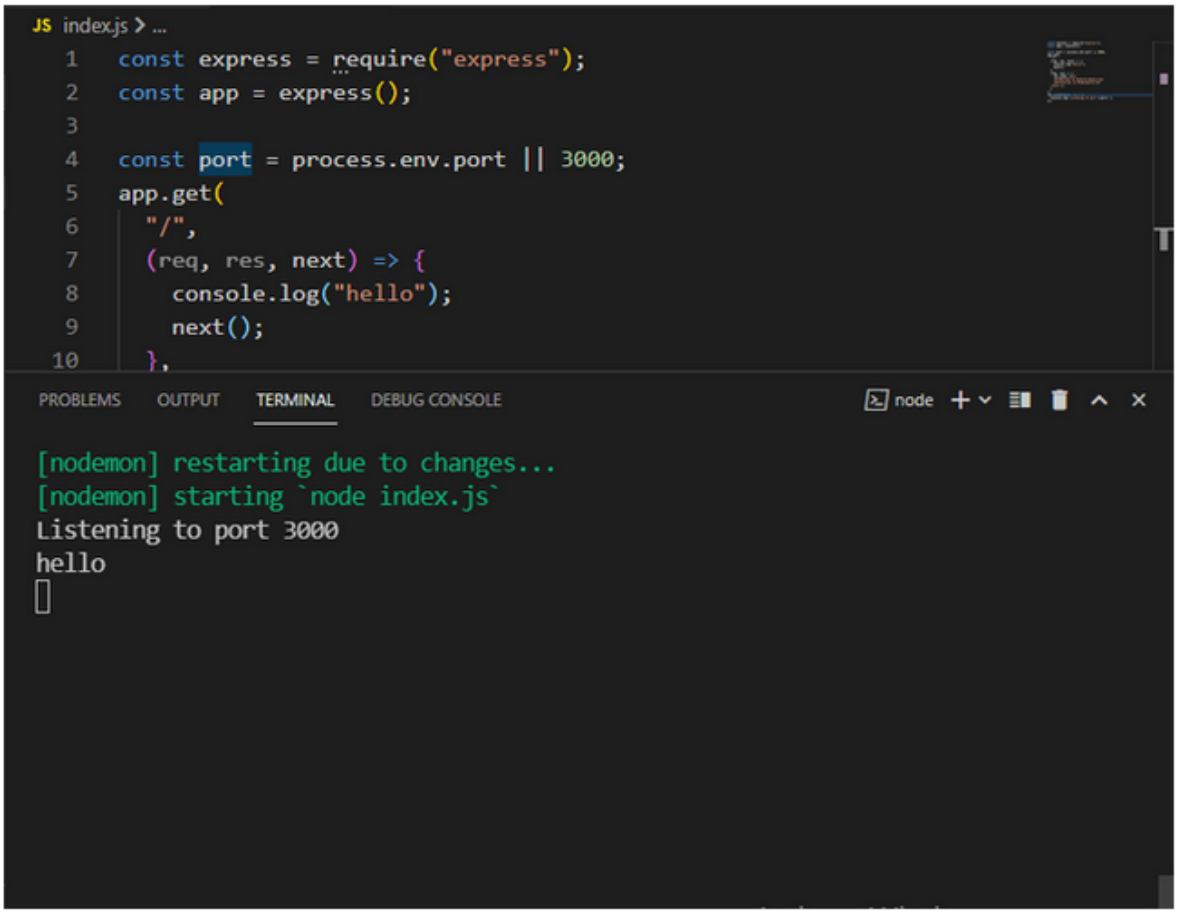
Create a Middleware

In the **app.get()** function, modify according to the following code.

index.js

```
app.get(
  "/",
  (req, res, next) => {
    console.log("hello");
    next();
  },
  (req, res) => {
    res.send(`<div>
    <h2>Welcome to GeeksforGeeks</h2>
    <h5>Tutorial on Middleware</h5>
  </div>`);
  }
);
```

Output



USING TEMPLATE ENGINES WITH EXPRESS.JS

A template engine facilitates you to use static template files in your applications. At runtime, it replaces variables in a template file with actual values, and transforms the template into an HTML file sent to the client. So this approach is preferred to design HTML pages easily.

Following is a list of some popular template engines that work with Express.js:

- Pug (formerly known as jade)
- mustache
- dust
- atpl
- eco

- ect
- ejs
- haml
- haml-coffee
- handlebars
- hogan
- jazz
- jqtpl
- JUST
- liquor
- QEJS
- swig
- templayed
- toffee
- underscore
- walrus
- whiskers

In the above template engines, pug (formerly known as jade) and mustache seems to be most popular choice. Pug is similar to Haml which uses whitespace. According to the template-benchmark, pug is 2x slower than Handlebars, EJS, Underscore. ECT seems to be the fastest. Many programmers like mustache template engine mostly because it is one of the simplest and versatile template engines.

Using template engines with Express

Template engine makes you able to use static template files in your application. To render template files you have to set the following application setting properties:

- **Views:** It specifies a directory where the template files are located.

  Example: app.set('views', './views').

- **View engine:** It specifies the template engine that you use. For example, to use the Pug template engine: app.set('view engine', 'pug').

Pug Template Engine

Pug is a template engine that works with NodeJS and browsers. Pug makes it easier to display data from a database or API and create reusable HTML.

Pug node js express templates allow the server-side data that must be sent to the application to be rendered quickly.

Syntax of a pug is as follows:

p # { name }

The advantages of using pug node js express include better separation of concerns, increased readability, better performance, simplicity, reusability, and dynamic templates.

Better separation of concerns:

Pug gives developers the ability to separate the presentation and functionality of their applications. As a result, it is simpler to manage and update an application's front end without affecting its back end.

Improved performance:

Performance has been improved because the server no longer needs to handle Pug templates for each request because they are compiled into HTML before even being served to the client. This may result in enhanced functionality and quicker website loads.

Dynamic templates:

Pug makes it simple for developers to build dynamic templates that can be rendered with different data by allowing them to use variables and looping constructs.

Greater clarity:

Since Pug templates are much simpler to read than conventional HTML code, developers can work on complicated projects and communicate with others more easily.

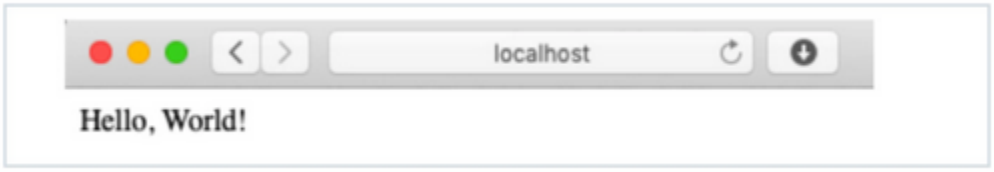
Reusability:

Pug enables writers to make mixins and reusable templates that can be incorporated into other templates. This enables developers to reuse common components across various templates, resulting in a more modular and effective development process.

Simplicity:

Pug makes it easier for programmers to write HTML code by using indentation and whitespace to specify the hierarchy of components. This makes HTML code simpler to understand and maintain, particularly for more significant projects.

Program: Create a Node.js and Express.js application integrated with Pug Template engine. Show a one-line Welcome to Express.js + Pug template.



Step - 1:

Start the node application and open port 3000 in the browser.

$ npm start

Step - 2:

Install pug engine:

npm i pug --save

Step - 3:

Create a subdirectory called /views inside the main directory to store the .pug template files. Now create a simple index.pug file. Note that it has a .pug extension and save it inside the freshly created directory /views. Add the following code to it to show a one-line Welcome to Express.js + Pug template.

index.pug

```
doctype = html

html

  head

    title Express.js + Pug

  body

    h1 Welcome to Express.js + Pug
```

Step 4:

We need to add two more lines of code to our Express.js main file, both of which use the app.set() function with the views and view engine properties. The view engine property specifies the template engine to use, while the views property refers to the directory where the templates are stored. The template is rendered by the app.get() method's res.render() function.

```
// pug node js express

const express = require("express");

// creating the express app

const app = express();

// setting the pug as view engine in our app

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

app.set("view engine", "pug");
```

```
// home page route
app.get("/", (req, res) => {
  res.render("index");
});
const port = 3000;
const host = "localhost";
// listening to the server
app.listen(port, host, () => {
  console.log(`Server started at ${host} port ${port}`);
});
```

The template gets rendered as:

ERROR HANDLING IN EXPRESS.JS

Error handling in Express is done using middleware. But this middleware has special properties. The error handling middleware are defined in the same way as other middleware functions, except that error-handling functions MUST have four arguments instead of three err, req, res, next.

Example

To send a response on any error, we can use

```
app.use(function(err, req, res, next) {

  console.error(err.stack);

  res.status(500).send('Something broke!');

});
```

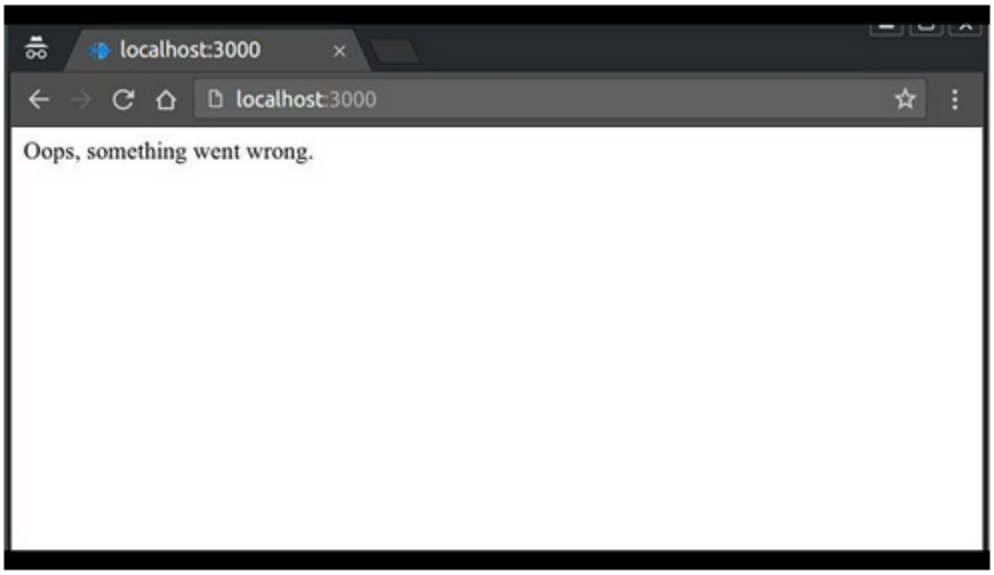
The error handling middleware allows us to separate our error logic and send responses accordingly. The next() method we discussed in middleware takes us to next middleware/route handler.

For error handling, we have the next(err) function. A call to this function skips all middleware and matches us to the next error handler for that route. Let us understand this through an example.

```
var express = require('express');
var app = express();
app.get('/', function(req, res){
  //Create an error and pass it to the next function
  var err = new Error("Something went wrong");
  next(err);
```

```
});
/*
 * other route handlers and middleware here
 * ....
 */
//An error handling middleware
app.use(function(err, req, res, next) {
  res.status(500);
  res.send("Oops, something went wrong.")
});
app.listen(3000);
```

This error handling middleware can be strategically placed after routes or contain conditions to detect error types and respond to the clients accordingly. The above program will display the following output.



API HANDLING IN EXPRESS.JS

REST (Representational state transfer) is a popular architecture that is used to create web services.

API (Application Programming Interface is a code that allows two software programs to communicate with each other.

REST API is a software that allows two apps to communicate with one another over the internet and through numerous devices.

HTTP Request Types

HTTP Requests are simply messages that are sent by the client to do some tasks on the server

- GET - Get command is used to request data from the server, but mainly this method is used to read data
- PATCH - This command is used to update, change or replace the data
- POST - The post method is used to create new or to edit already existing data
- Delete - This delete command is used to delete the data completely from the server

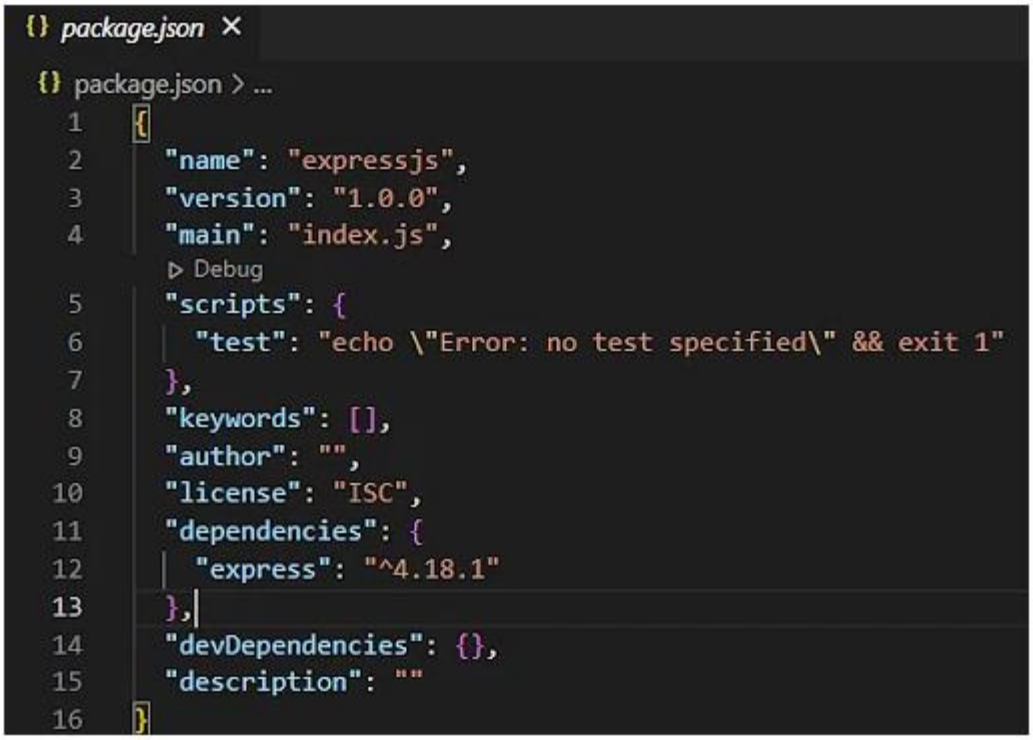
API Using Express and Its Architecture

A request is sent by the client in the form of a JSON file, and with the help of an HTTP request which gets a patch post and delete, it will go to the server first then, the server sends back the response to the client in the form of a message to tell what happened to your request.

Step 1: First, open your editor. Now open your terminal and write a command npm init -y

```
PS C:\expressjs> npm init -y
Wrote to C:\expressjs\package.json:

{
  "name": "expressjs",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
```

This command will create a JSON file.



Step 2: Now we will install express for that write command

npm install express

```
PS C:\expressjs> npm install express
[########.........] \ idealTree: timing idealTree Completed in 383ms
```

This will install dependencies in the package.json file

Step 3: Now we will create a new file to write all our code index.js

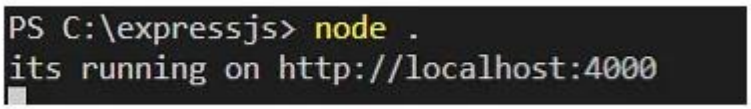
Write a few lines of code in the first line. We will import the express packages

Step 4:

This app.listen will respond to the server to listen on a specific port which we have defined already as a variable 4000

Step 5: Then we will write a second argument as a callback to tell the API is working, now lets run this code to check our API is working or node

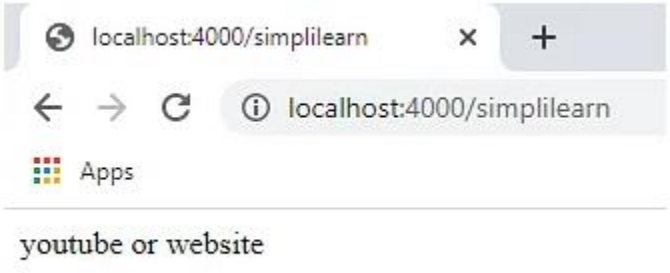
To run this code, we will write a command node.



Step 6: As our code is running fine, let's open our search engine and write localhost:4000, this will not run,

Step 7: Now we need an add the second argument here

localhost:4000/simplilearn



Our Simple API is running Successfully.

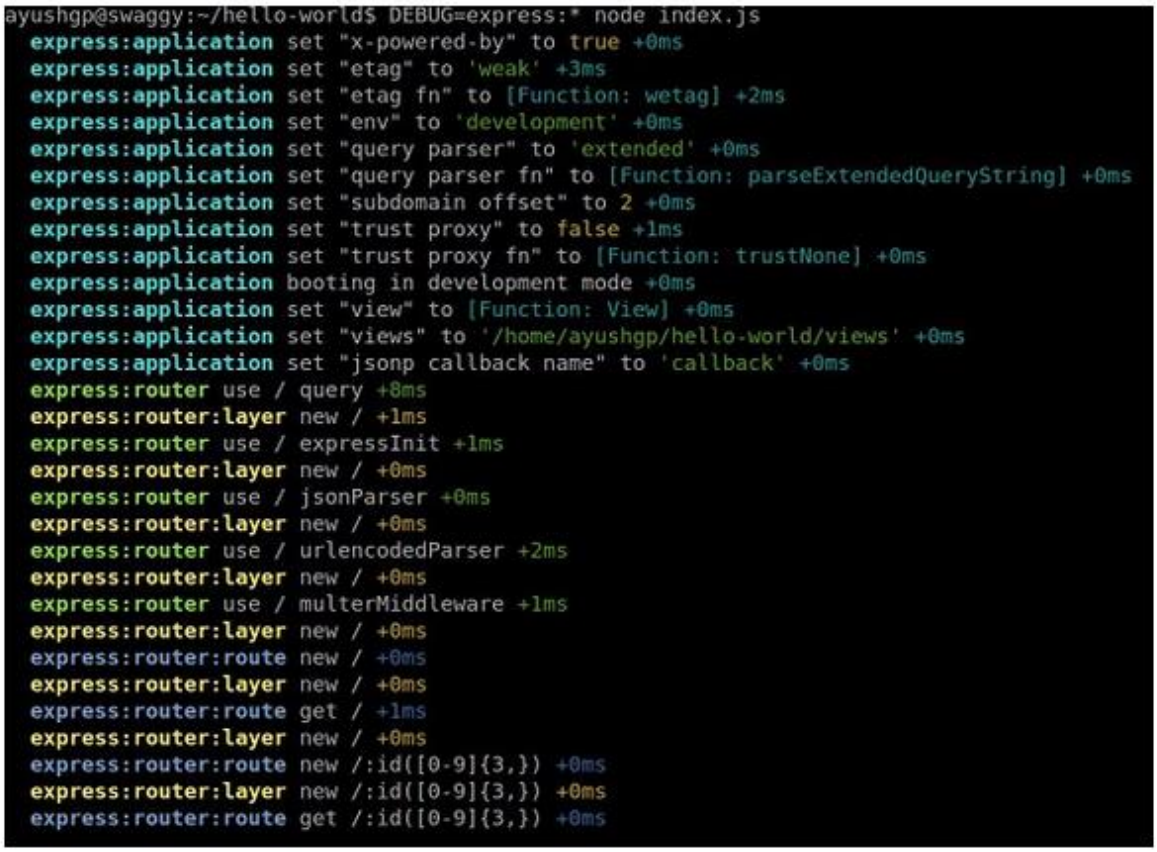
DEBUGGING IN EXPRESS.JS

Express uses the Debug module to internally log information about route matching, middleware functions, application mode, etc.

To see all internal logs used in Express, set the DEBUG environment variable to **Express:*** when starting the app.

DEBUG = express:* node index.js

The following output will be displayed.



These logs are very helpful when a component of your app is not functioning right. This verbose output might be a little overwhelming. You can also restrict the DEBUG variable to specific area to be logged. For example, if you wish to restrict the logger to application and router, you can use the following code.

DEBUG = express:application,express:router node index.js

Debug is turned off by default and is automatically turned on in production environment. Debug can also be extended to meet your needs, you can read more about it at its npm page.

DEVELOPING TEMPLATE ENGINES FOR EXPRESS.JS

Use the app.engine(ext, callback) method to create your own template engine. ext refers to the file extension, and callback is the template engine function, which accepts the following items as parameters: the location of the file, the options object, and the callback function.

Example

Program: Create a Express.JS application for implementing a very simple template engine for rendering .ntl files.

index.ntl

```
const fs = require('fs') // this engine requires the fs module

app.engine('ntl', (filePath, options, callback) => { // define the template engine

  fs.readFile(filePath, (err, content) => {

    if (err) return callback(err)

    // this is an extremely simple template engine

    const rendered = content.toString()

      .replace('#title#', `<title>${options.title}</title>`)

      .replace('#message#', `<h1>${options.message}</h1>`)

    return callback(null, rendered)

  })

})
```

```
app.set('views', './views') // specify the views directory

app.set('view engine', 'ntl') // register the template engine
```

Your app will now be able to render .ntl files. Create a file named index.ntl in the views directory with the following content.

```
#title#

#message#
```

Then, create the following route in your app.

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

  res.render('index', { title: 'Hey', message: 'Hello there!' })

})
```

When you make a request to the home page, index.ntl will be rendered as HTML.

USING PROCESS MANAGERS IN EXPRESS.JS

When you run Express apps for production, it is helpful to use a *process manager* to:

- Restart the app automatically if it crashes.
- Gain insights into runtime performance and resource consumption.
- Modify settings dynamically to improve performance.
- Control clustering.

A process manager is somewhat like an application server: it's a "container" for applications that facilitates deployment, provides high availability, and enables you to manage the application at runtime.

The most popular process managers for Express and other Node.js applications are:

- **Forever**: A simple command-line interface tool to ensure that a script runs continuously (forever). Forever's simple interface makes it ideal for running smaller deployments of Node.js apps and scripts.
- **PM2**: A production process manager for Node.js applications that has a built-in load balancer. PM2 enables you to keep applications alive forever, reloads them without downtime, helps you to manage application logging, monitoring, and clustering.
- **StrongLoop Process Manager (Strong-PM)**: A production process manager for Node.js applications with built-in load balancing, monitoring, and multi-host deployment. Includes a CLI to build, package, and deploy Node.js applications to a local or remote system.
- **SystemD**: The default process manager on modern Linux distributions, that makes it simple to run a Node application as a service.

SECURITY AND DEPLOYMENT IN EXPRESS.JS

Security in Express.js

There are 7 simple measures to take for the purpose of data security:

1. Use reliable versions of Express.js
2. Secure the connection and data
3. Protect your cookies
4. Secure your dependencies
5. Validate the input of your users
6. Protect your system against brute force
7. Control user access

1. Use reliable versions of Express.js

Deprecated or outdated versions of Express.js are a no go. The 2nd and 3rd versions of Express are no longer supported. In these, safety or performance issues are not fixed anymore.

As a developer, you absolutely have to migrate to Express 4. This version is a revolution! It is quite different in terms of the routing system, middleware, and other minor aspects.

2. Secure the connection and data

To secure HTTP headers, you can make use of Helmet.js – a helpful Node.js module. It is a collection of 13 middleware functions for setting HTTP response headers. In particular, there are functions for setting Content Security Policy, handling Certificate Transparency, preventing clickjacking, disabling client-side caching, or adding some small XSS protections.

npm install helmet --save

Even if you do not want to use all the functions of Helmet, the absolute minimum that you must do is to disable X-Powered-By header:

Even if you do not want to use all the functions of Helmet, the absolute minimum that you must do is to disable X-Powered-By header:

app.disable('x-powered-by')

This header can be used to detect that the application is powered by Express, which lets hackers conduct a precise attack. Surely, X-Powered-By header is not the only way to identify an Express-run application, but it is probably the most common and simple one.

To protect your system from HTTP parameter pollution attacks, you can use HPP. This middleware puts aside such parameters as req.query and req.body and selects the latest parameter value instead. The installation command looks as follows:

npm install hpp --save

To encrypt data which is being sent from the client to the server, use Transport Layer Security (TLS). TLS is a cryptographic protocol for securing the computer network, the descendant of the Secure Socket Layer (SSL) encryption. TLS can be handled with Nginx – a free but effective HTTP server – and Let's Encrypt – a free TLS certificate.

3. Protect your cookies

In Express.js 4, there are two cookie session modules:

- express-session (in Express.js 3, it was express.session)
- cookie-session (in Express.js 3, it was express.cookieSession)

The express-session module stores session ID in the cookie and session data on the server. The cookie-session stores all the session data to the cookie.

In general, cookie-session is more efficient. Yet, if the session data you need to store is complex and likely to exceed 4096 bytes per cookie, use express-session. Another reason to use express-session is when you need to keep the cookie data invisible to the client.

Besides, you should set cookie security options, namely:

- secure
- httpOnly
- domain
- path
- expires

If "secure" is set to "true", the browser will send cookies only via HTTPS. If "httpOnly" is set to "true", the cookie will be sent not via client JS but via HTTP(S). The value of "domain" indicates the domain of the cookie. If the cookie domain matches the server domain, "path" is used to indicate the cookie path. If the cookie path matches the request path, the cookie will be sent in the request. Finally, as the name itself suggests, the value of "expires" stands for the time when the cookies will expire.

Another important recommendation is not to use the default session cookie name. It may enable hackers to detect the server and to run a targeted attack. Instead, use generic cookie names.

4. Secure your dependencies

No doubt, npm is a powerful web development tool. However, to ensure the highest level of security, consider using only the 6th version of it – npm@6. The older ones may contain some serious dependency safety vulnerabilities, which will endanger your entire app. Also, to analyze the tree of dependencies, use the following command:

npm audit

npm audit can help to fix real problems in your project. It checks all your dependencies in dependencies, devDependencies, bundledDependencies, and optionalDependencies, but not your peerDependencies. Here you can read about all current vulnerabilities in any npm packages.

Another tool to ensure dependency safety is Snyk. Snyk runs the application check to identify whether it contains any vulnerability listed in Snyk's open-source database. To conduct the check, run three simple steps.

Step 1. Install Snyk

npm install -g snyk
cd your-app

Step 2. Run a test

snyk test

Step 3. Learn how to fix the issue

snyk wizard

Wizard is a Snyk method, which explains the nature of the dependency vulnerability and offers ways of fixing it.

5. Validate the input of your users

Controlling user input is an extremely important part for server-side development. This is a no less important problem than unauthorized requests, which will be described in the seventh part of this article.

First of all, wrong user input can break your server when some values are undefined and you do not have error handling for a specific endpoint. However, different ORM systems can have unpredictable behavior when you try to set undefined, null, or other data types in the database.

For example, destroyAll method in Loopback.js ORM (Node.js framework) can destroy all data in a table of the database: when it does not match any records it deletes everything as described here. Imagine that you can lose all data in a production table just because you have ignored input validation.

Use body/object validation for intermediate inspections

To start with, you can use body/object validation for intermediate inspections. For example, we use ajv validator which is the fastest JSON Schema validator for Node.js.

```
const Ajv = require('ajv');
const ajv = new Ajv({allErrors: true});
const speaker = {
  'type': 'object',
  'required': [
    'id',
    'name'
  ],
  'properties': {
    'id': {
      'type': 'integer',
    },
```

```
    'name': {
      'type': 'string',
    },
  },
};
const conversation = {
  type: 'object',
  required: [
    'duration',
    'monologues'
  ],
  properties: {
    duration: {
      type: 'integer',
    },
    monologues: {
      type: 'array',
      items: monolog,
    },
  },
};
const body = {
  type: 'object',
  required: [
    'speakers',
    'conversations'
  ],
  properties: {
    speakers: {
      type: 'array',
```

```
      items: speaker,
    },
    conversations: {
      type: 'array',
      items: conversation,
    },
  },
};
const validate = ajv.compile(body);
const isValidTranscriptBody = transcriptBody => {
  const isValid = validate(transcriptBody);
  if (!isValid) {
    console.error(validate.errors);
  }
  return isValid;
};
```

Handle errors

Now, imagine that you forgot to check a certain object and you do some operations with the undefined property. Or you use a certain library and you get an error. It can break your instance, and the server will crash. Then, the attacker can ping a specific endpoint where there is this vulnerability and can stop your server for a long time.

The simplest way to do an error handling is to use try-catch construction:

```
try {
  const data = body;
  if (data.length === 0) throw new Error('Client Error');
  const beacons = await  this.beaconLogService.filterBeacon(data);
  if (beacons.length > 0) {
    const max = beacons.reduce((prev, current) => (prev.rssi > current.rssi) ? prev : current);
```

```
    await this.beaconLogService.save({
      ...max,
      userId: headers['x-uuid']
    });
    return {
      data: {
        status: 'Saved',
        position: max
      },
    };
  }
  return {
    data: {
      status: 'Not valid object,
    },
  };
}
catch(err) {
  this.logger.error(err.message, err.stack);
  throw new HttpException('Server Error',     HttpStatus.INTERNAL_SERVER_ERROR);
}
```

Feel free to use a new Error('message') constructor for error handling or even extend this class for your own purpose!

Use JOI

The main lesson here is that you should always validate user input so you don't fall victim to man-in-the-middle attacks. Another way to do it is with the help of @hapi/joi – a part of the hapi ecosystem and a powerful JS data validation library.

Pay attention here that the module joi has been deprecated. For this reason, the following command is a no go:

npm install joi

Instead, use this one:

npm install @hapi/joi

Use express-validator

One more way to validate user input is to use express-validator – a set of express.js middlewares, which comprises validator.js and function sanitizer. To install it, run the following command:

npm install --save express-validator

Sanitize user input

Also, an important measure to take is to sanitize user input to protect the system from a MongoDB operator injection. For this, you should install and use express-mongo-sanitize:

npm install express-mongo-sanitize

Protect your app against CSRF

Besides, you should protect your app against cross-site request forgery (CSRF). CSRF is when unauthorized commands are sent from a trusted user. You can do this with the help of csurf. Prior to that, you need to make sure that session middleware for cookies is configured as described earlier in this article. To install this Node.js module, run the command:

npm install csurf

6. Protect your system against brute force

A brute force attack is the simplest and most common way to get access to a website or a server. The hacker (in most cases automatically, rarely manually) tries various usernames and passwords repeatedly to break into the system.

These attacks can be prevented with the help of rate-limiter-flexible package. This package is fast, flexible, and suitable for any Node framework.

To install, run the following command:

npm i --save rate-limiter-flexible
yarn add rate-limiter-flexible

This method has a simpler but more primitive alternative: express-rate-limit. The only thing it does is limiting repeated requests to public APIs or to password reset.

npm install --save express-rate-limit

7. Control user access

Among the authentication methods, there are tokens, Auth0, and JTW. Let's focus on the third one! JTW (JSON Web Tokens) are used to transfer authentication data in client-server applications. Tokens are created by the server, signed with a secret key, and transferred to a client. Then, the client uses these tokens to confirm identity.

Express-jwt-permissions is a tool used together with express-jwt to check permissions of a certain token. These permissions are an array of strings inside the token:

```
"permissions": [
  "status",
  "user:read",
  "user:write"
```

]

To install the tool, run the following command:

npm install express-jwt-permissions --save

Deployment of Express.js applications

We will Deploy the **Node.js** and **Express.js** application on render.

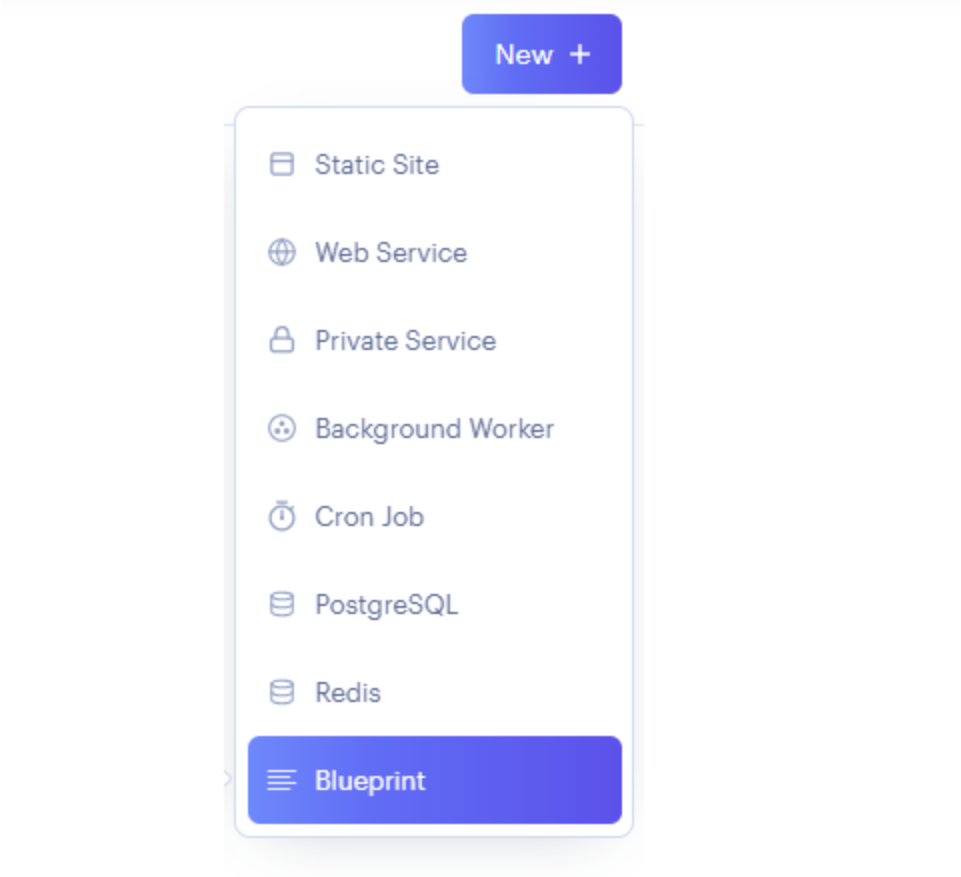
Render is the website that provides hosting services. Render is a cloud-based platform that provides a simple and scalable way to host and manage websites and web applications. With Render, users can quickly deploy websites and applications to the cloud, without having to worry about infrastructure management or server maintenance. It provides the free service to host up to the

Step-by-Step Implementation:

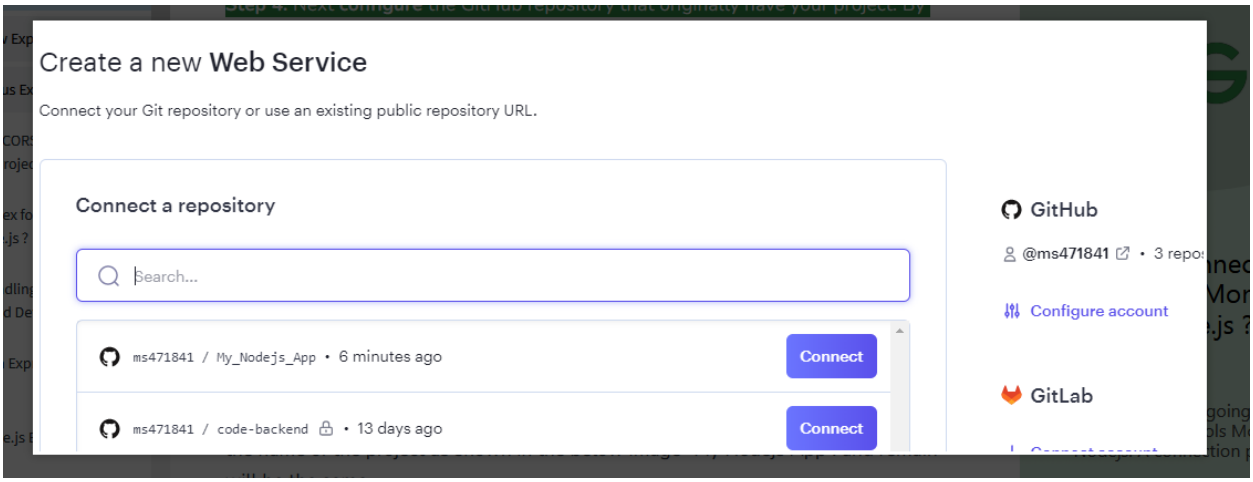
Step 1: Deploy the code on GitHub: You need to deploy the code on **GitHub** first.

Step 2: Now create the account on Render. You can Sign in with **GitHub**.

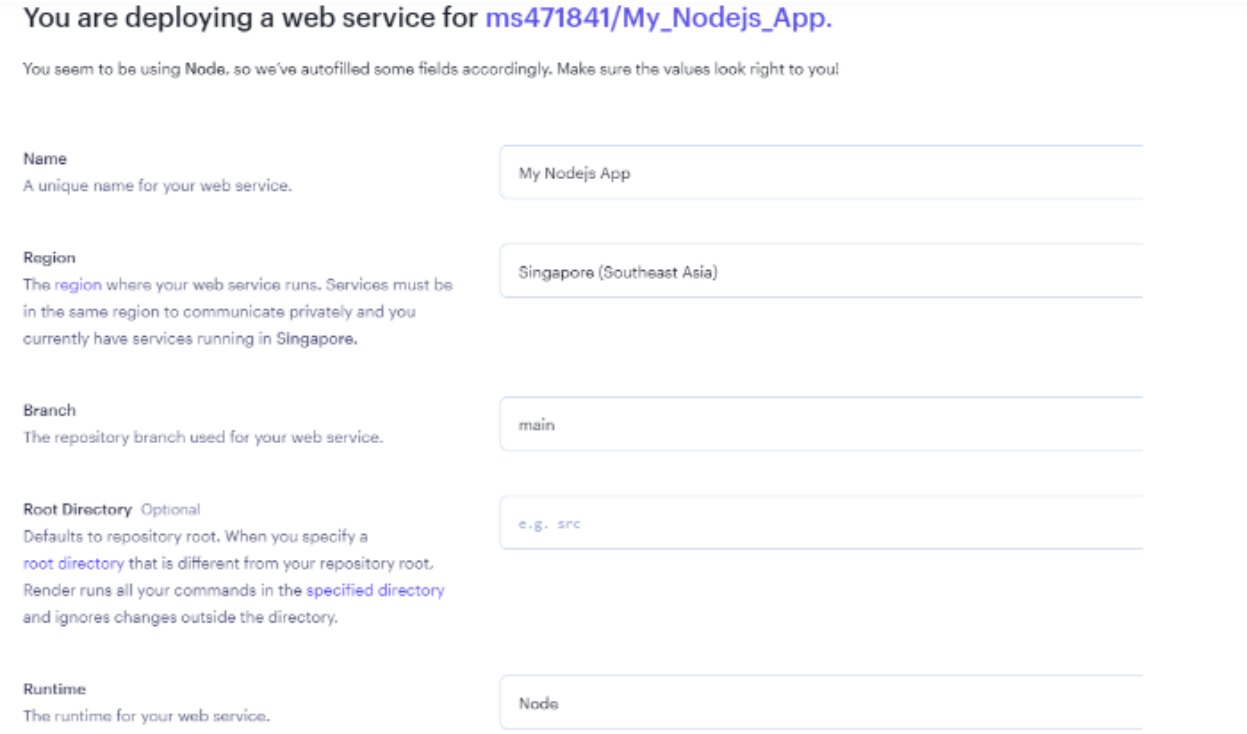
Step 3: On the Dashboard, click on the **new** and then **web service** as shown in the below image.



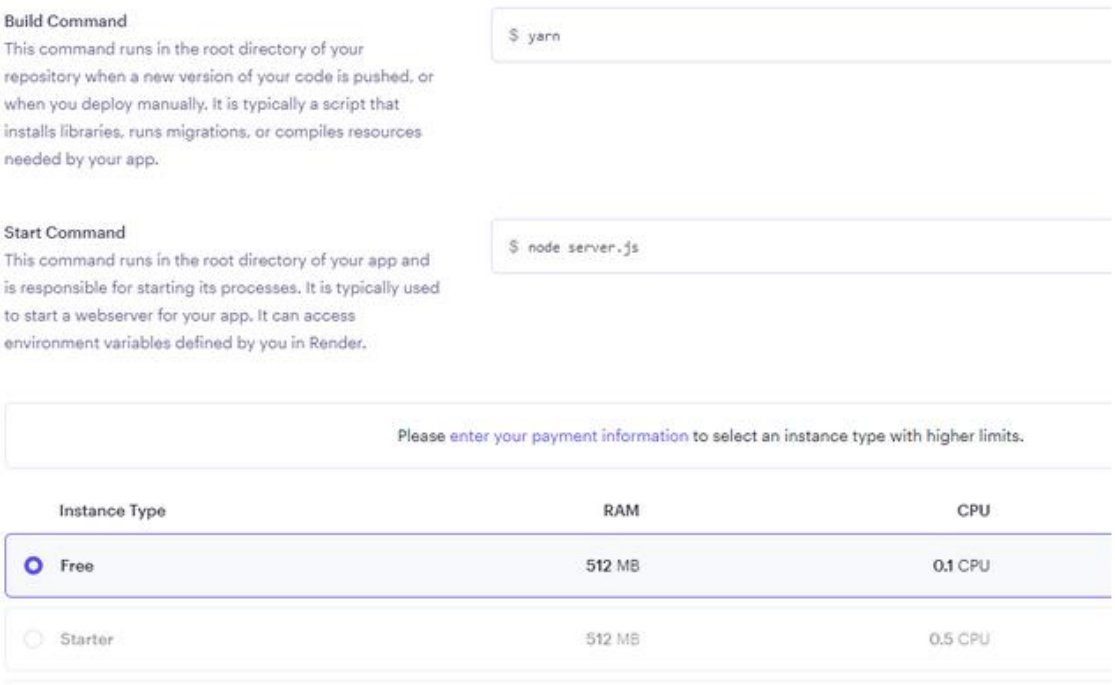
Step 4: Next **configure** the GitHub repository that originally have your project. By clicking on the **Configure account** as shown in the below image. After configuring click on the **connect** button.



Step 5: Now you are on the screen "You are deploying a web service", After Type the name of the project as shown in the below image "My Nodejs App". and remain will be the same.



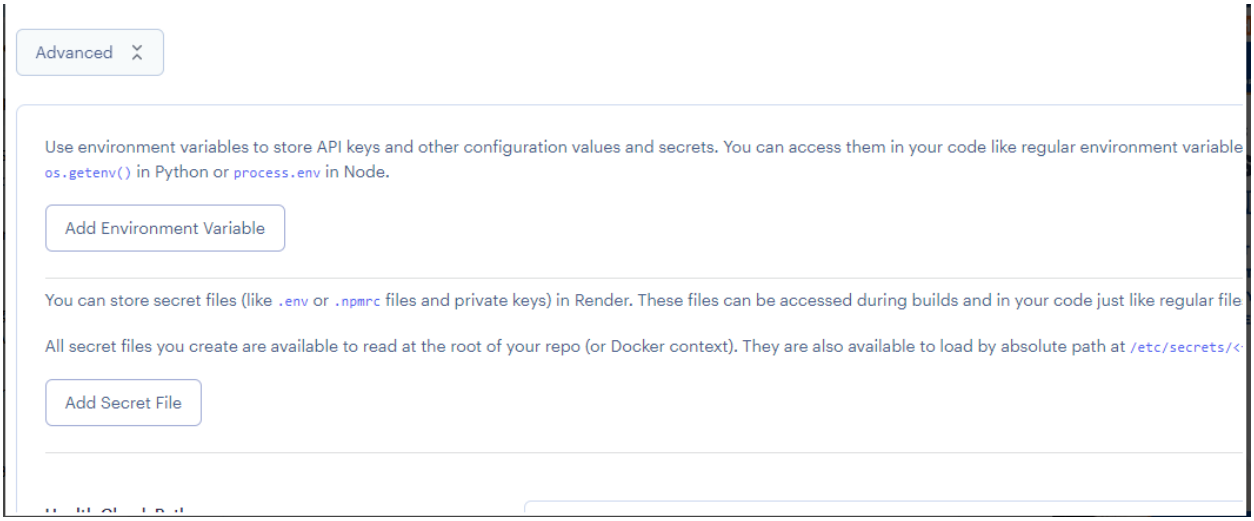
Step 6: Next, this is the main part, in **Start Command**, you have to write the **Start Command** for your app that going to be exactly the same as in the packages Json file scripts. And then click on free and as you want but we are going with free.



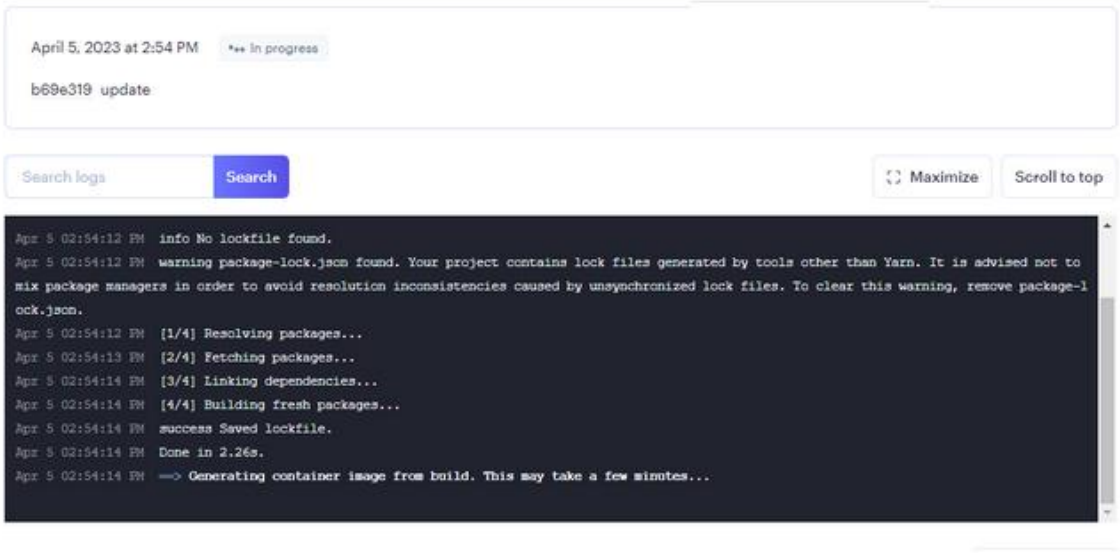
My Package Json file looks like and sees the **start scripts that is node server.js**.

```
{
    "name": "sample",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
    "type": "module",
    "scripts": {
        "test": "echo \"Error: no test specified\" && exit 1",
        "start": "node server.js"
    },
    "author": "",
    "license": "ISC",
    "dependencies": {
          "express": "^4.18.2"
    }
}
```

Step 7: If your **Node.js** application has the **environment** variables then add all of them by clicking on advance and then **Add Environment Variable**.



Step 8: Next Click on Create and wait for a moment until it is deployed.



Now finally your **Node.js** app is deployed on render, you can check it by clicking on the link, that appears something like that.

Output: