Node.js Tutorial

Node.js tutorial provides basic and advanced concepts of Node.js. Our Node.js tutorial is designed for beginners and professionals both.

Node.js is a cross-platform environment and library for running JavaScript applications which is used to create networking and server-side applications.

Our Node.js tutorial includes all topics of Node.js such as Node.js installation on windows and linux, REPL, package manager, callbacks, event loop, os, path, query string, cryptography, debugger, URL, DNS, Net, UDP, process, child processes, buffers, streams, file systems, global objects, web modules etc. There are also given Node.js interview questions to help you better understand the Node.js technology.

What is Node.js

Node.js is a cross-platform runtime environment and library for running JavaScript applications outside the browser. It is used for creating server-side and networking web applications. It is open source and free to use. It can be downloaded from this link https://nodejs.org/en/

The definition given by its official documentation is as follows:

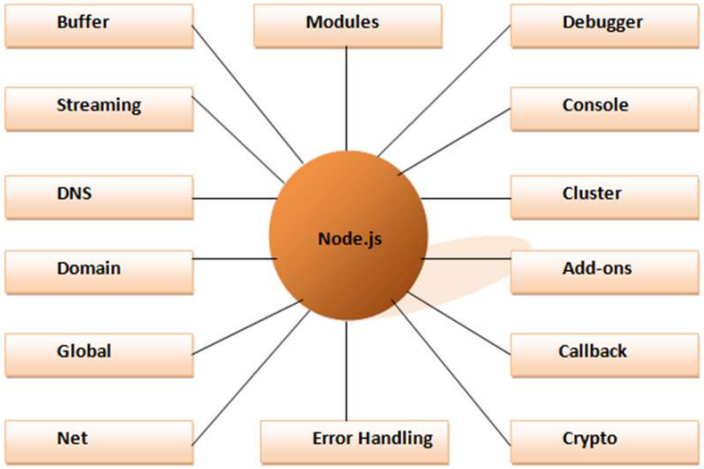
Node.js is a platform built on Chrome's JavaScript runtime for easily building fast and scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.

Node.js also provides a rich library of various JavaScript modules to simplify the development of web applications.

1. Node.js = Runtime Environment + JavaScript Library

Different parts of Node.js

The following diagram specifies some important parts of Node.js:



Features of Node.js

Following is a list of some important features of Node.js that makes it the first choice of software architects.

1. **Extremely fast:** Node.js is built on Google Chrome's V8 JavaScript Engine, so its library is very fast in code execution.
2. **I/O is Asynchronous and Event Driven:** All APIs of Node.js library are asynchronous i.e. non-blocking. So a Node.js based server never waits for an API to return data. The server moves to the next API after calling it and a notification mechanism of Events of Node.js helps the server to get a response from the previous API call. It is also a reason that it is very fast.
3. **Single threaded:** Node.js follows a single threaded model with event looping.
4. **Highly Scalable:** Node.js is highly scalable because event mechanism helps the server to respond in a non-blocking way.
5. **No buffering:** Node.js cuts down the overall processing time while uploading audio and video files. Node.js applications never buffer any data. These applications simply output the data in chunks.
6. **Open source:** Node.js has an open source community which has produced many excellent modules to add additional capabilities to Node.js applications.
7. **License:** Node.js is released under the MIT license.

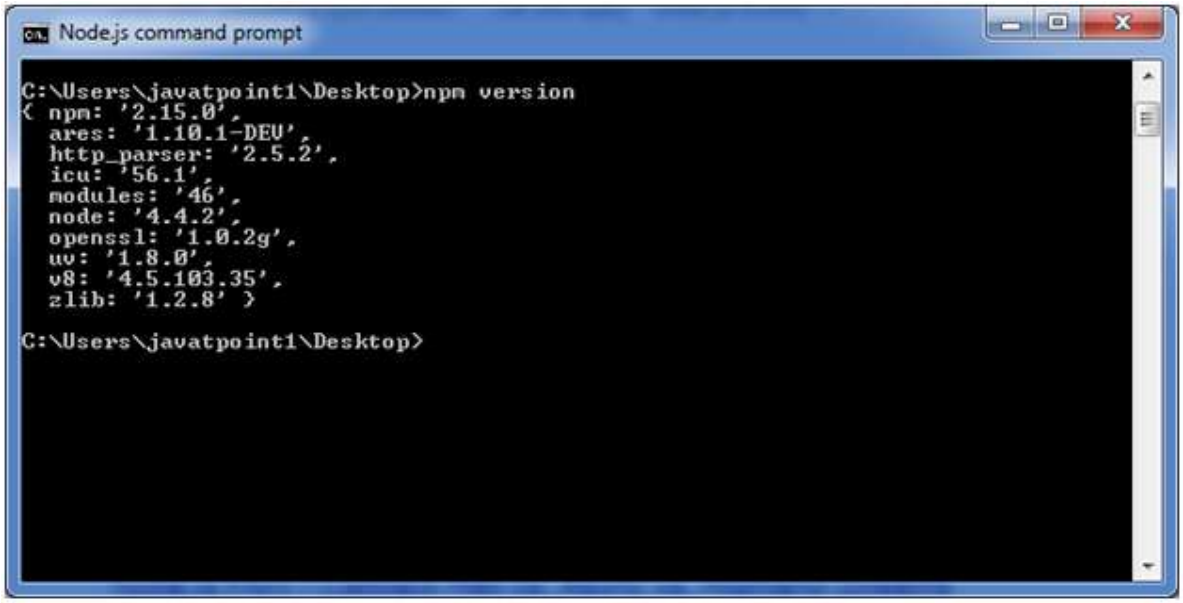
Node.js Package Manager(NPM)

Node Package Manager provides two main functionalities:

- It provides online repositories for node.js packages/modules which are searchable on search.nodejs.org
- It also provides command line utility to install Node.js packages, do version management and dependency management of Node.js packages.

The npm comes bundled with Node.js installables in versions after that v0.6.3. You can check the version by opening Node.js command prompt and typing the following command:

```
1. npm  version
```



Installing Modules using npm

Following is the syntax to install any Node.js module:

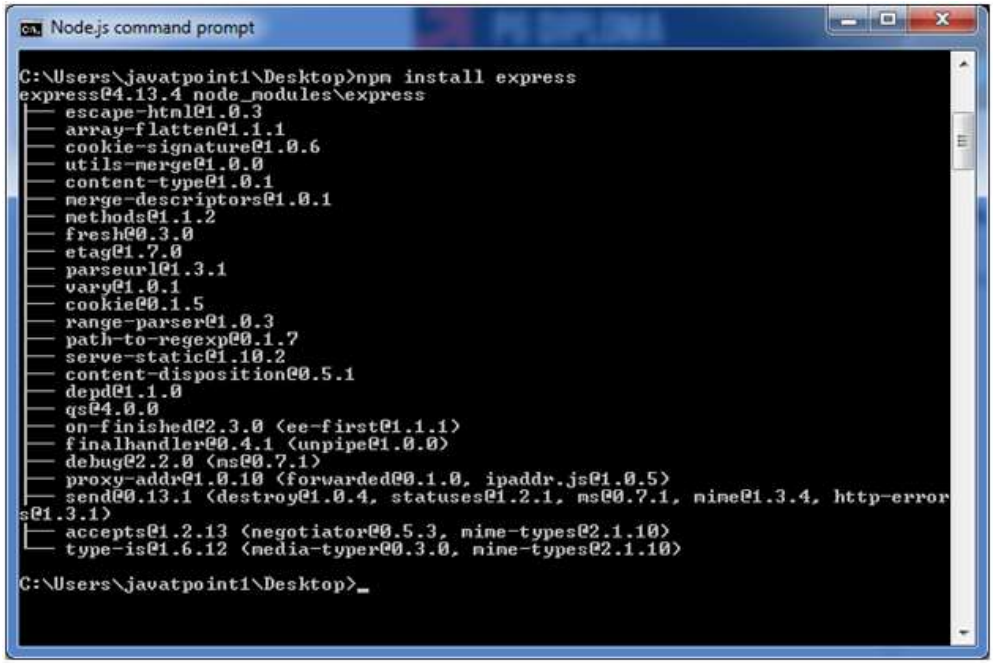
1. npm install **<Module** Name**>**

Let's install a famous Node.js web framework called express:

Open the Node.js command prompt and execute the following command:

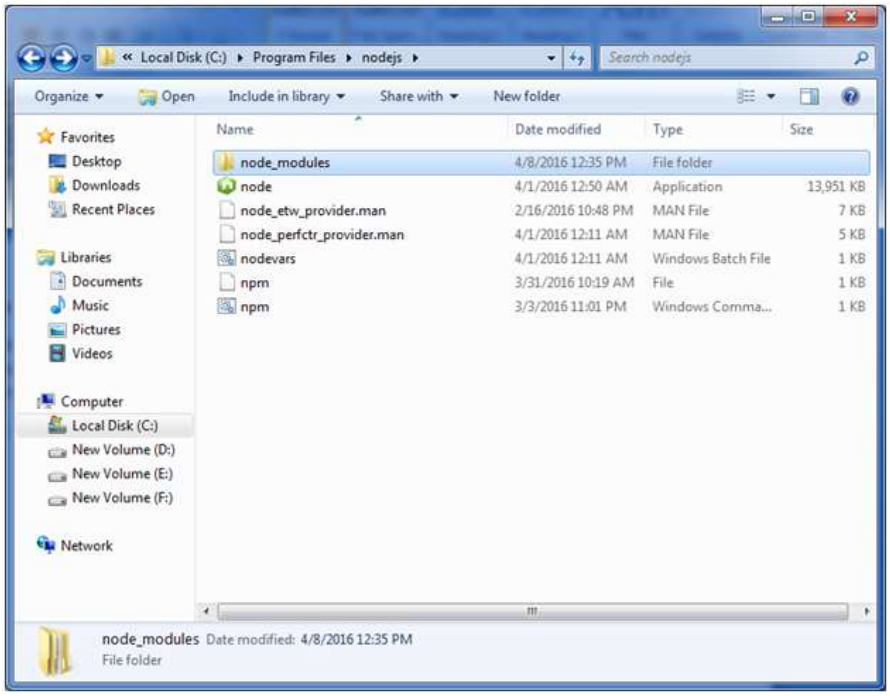
1. npm install express

You can see the result after installing the "express" framework.



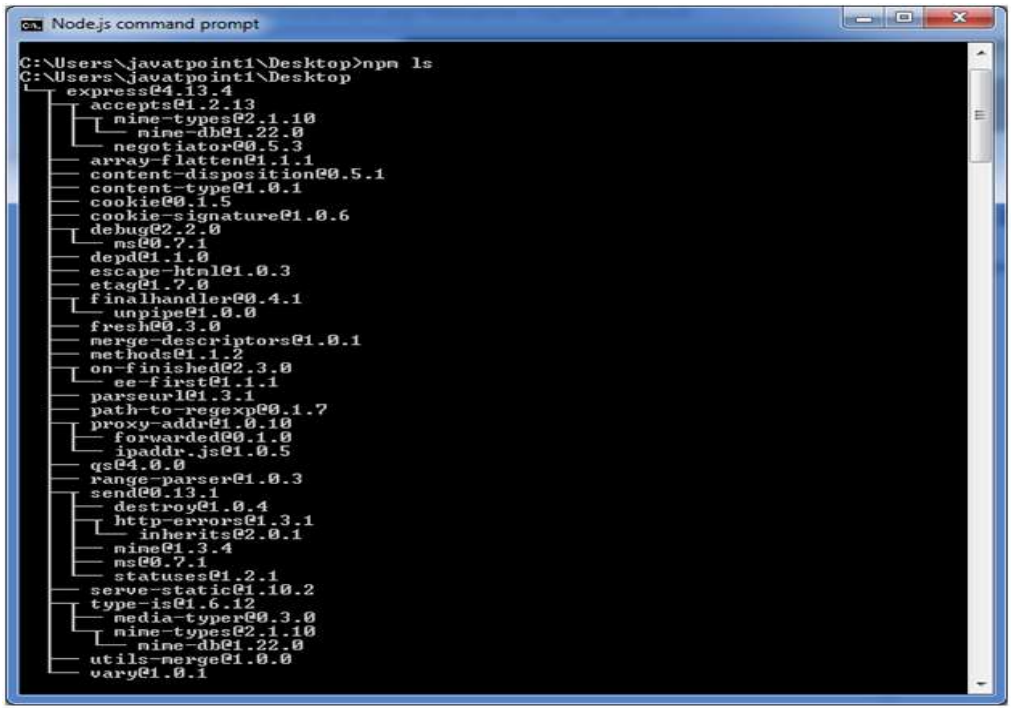
Global vs Local Installation

By default, npm installs dependency in local mode. Here local mode specifies the folder where Node application is present. For example if you installed express module, it created node_modules directory in the current directory where it installed express module.



You can use npm ls command to list down all the locally installed modules.

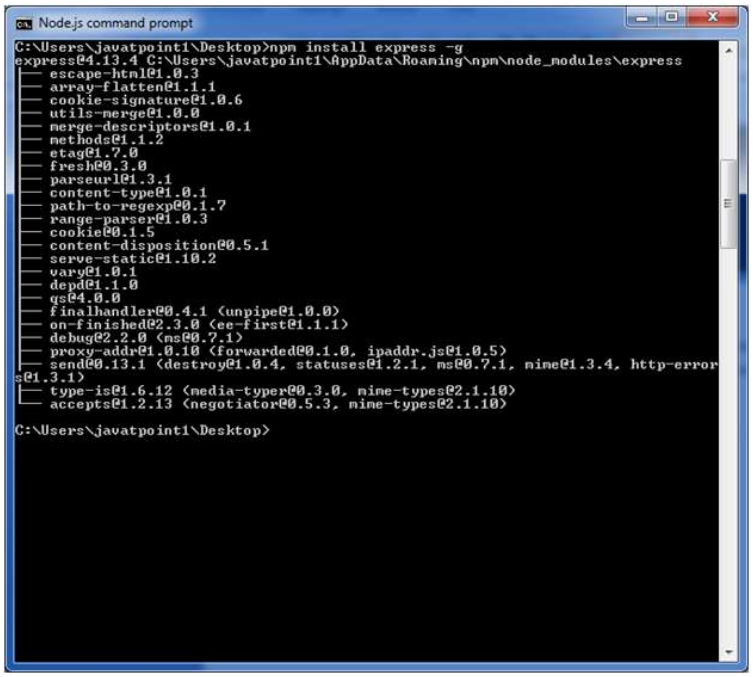
Open the Node.js command prompt and execute "npm ls":



Globally installed packages/dependencies are stored in system directory. Let's install express module using global installation. Although it will also produce the same result but modules will be installed globally.

Open Node.js command prompt and execute the following code:

1. npm install express -g

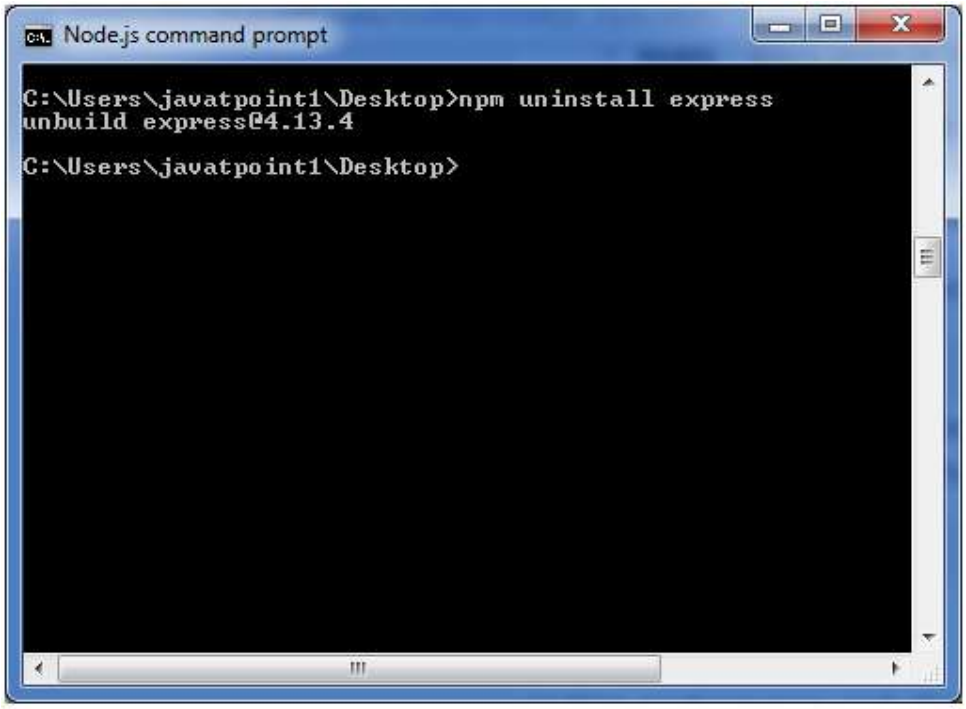


Here first line tells about the module version and its location where it is getting installed.

Uninstalling a Module

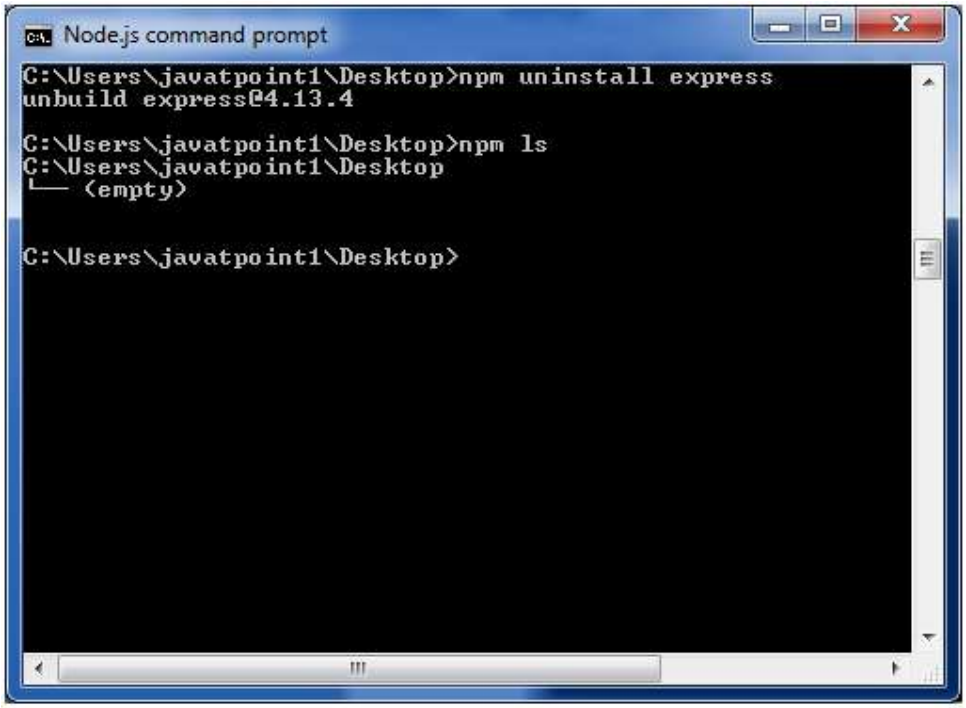
To uninstall a Node.js module, use the following command:

1. npm uninstall express



The Node.js module is uninstalled. You can verify by using the following command:

1. npm ls

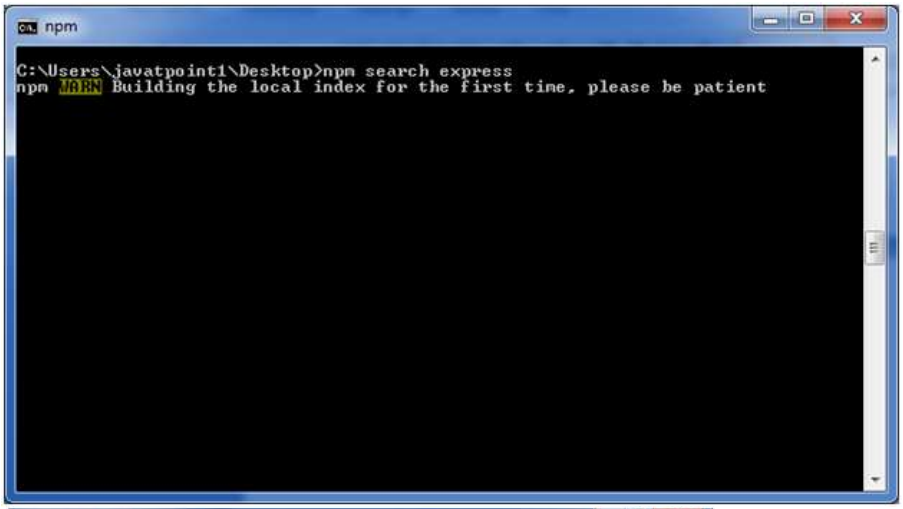


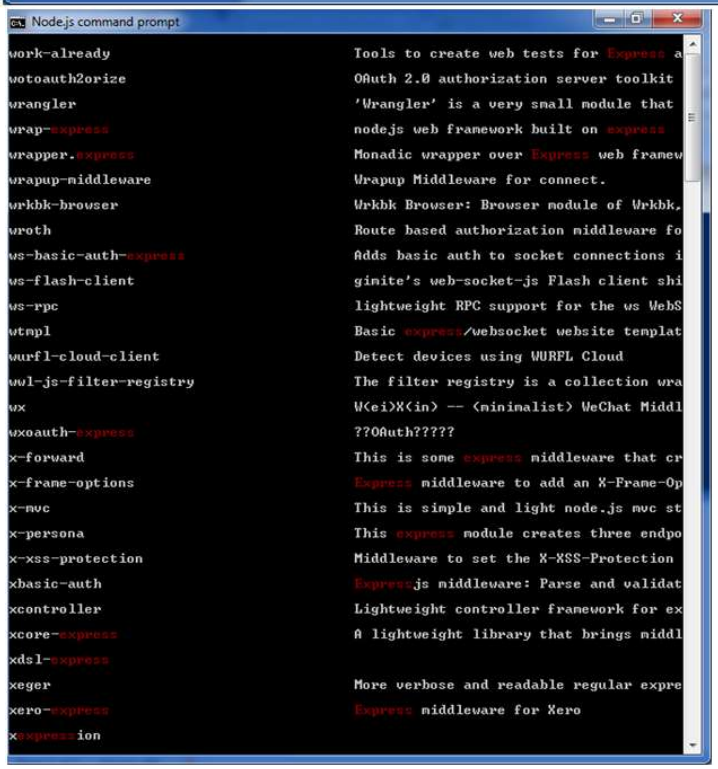
You can see that the module is empty now.

Searching a Module

"npm search express" command is used to search express or module.

1. npm search express





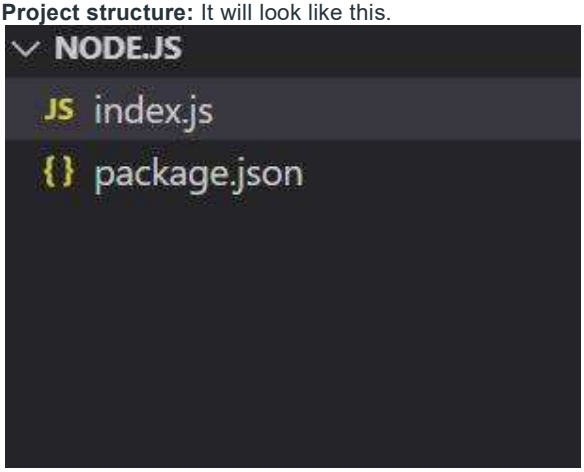
Build a Simple Web Server

Introduction: Node.js is an open-source and cross-platform runtime environment for executing JavaScript code outside a browser. You need to remember that **NodeJS is not a framework, and it's not a programming language.** Node.js is mostly used in server-side programming. In this article, we will discuss how to make a web server using node.js.

Creating Web Servers Using NodeJS: There are mainly two ways as follows.

1. Using http inbuilt module
2. Using express third party module

Using http module: HTTP and HTTPS, these two inbuilt modules are used to create a simple server. The HTTPS module provides the feature of the encryption of communication with the help of the secure layer feature of this module. Whereas the HTTP module doesn't provide the encryption of the data.
Project structure: It will look like this.

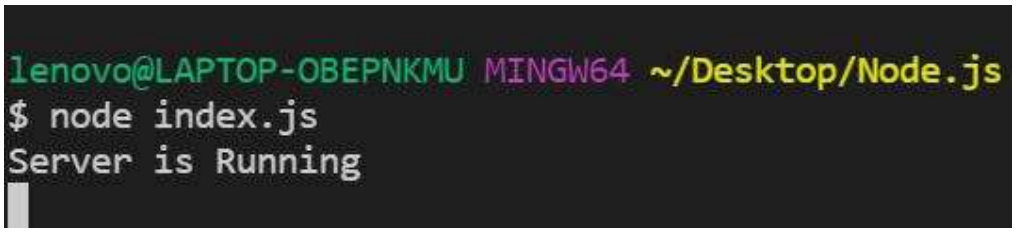


index.js

```
// Importing the http module

const http = require("http")



// Creating server

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

    // Sending the response

    res.write("This is the response from the server")

    res.end();

})



// Server listening to port 3000

server.listen((3000), () => {

    console.log("Server is Running");

})
```

Run **index.js** file using below command:

```
node index.js
```



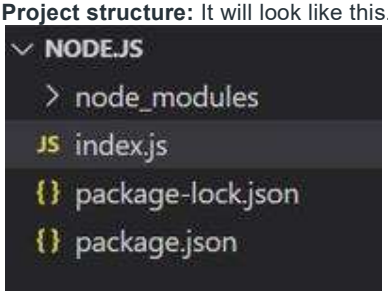
Output: Now open your browser and go to ***http://localhost:3000/***, you will see the following output:

Using express module: The *express.js* is one of the most powerful frameworks of the node.js that works on the upper layer of the http module. The main advantage of using *express.js* server is filtering the incoming requests by clients.

Installing module: Install the required module using the following command.

```
npm install express
```

Project structure: It will look like this.



index.js

```
// Importing express module

const express = require("express")

const app = express()



// Handling GET / request

app.use("/", (req, res, next) => {

    res.send("This is the express server")

})
```

```
// Handling GET /hello request

app.get("/hello", (req, res, next) => {

    res.send("This is the hello response");

})


// Server setup

app.listen(3000, () => {

    console.log("Server is Running")

})
```

Run **the index.js** file using the below command:

```
node index.js
```

Output: Now open your browser and go to ***http://localhost:3000/***, you will see the following output:

This is the express server

Introduction

Learning how to make HTTP requests in Node.js can feel overwhelming as dozens of libraries are available, with each solution claiming to be more efficient than the last. Some libraries offer cross-platform support, while others focus on bundle size or developer experience.

In this post, we'll explore five of the most popular ways to make HTTP requests in Node.js, with step-by-step instructions for each method.

First, we'll cover HTTP requests and HTTPS requests using the standard library. After that, we'll show you how to use alternatives like Node Fetch, Axios, and SuperAgent.

How to Make HTTP Requests in Node.js

In Node.js, you can send and handle HTTP requests using different libraries, with the most common ones being `http` (built-in module), `axios`, and `node-fetch`. Below are examples of how to send and handle HTTP requests using these options:

1. Using `http` (Built-in Node.js module)

The `http` module allows you to send HTTP requests without installing any additional libraries. However, it's more low-level compared to libraries like Axios or Fetch.

Example: Sending a GET request with `http`

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

const options = {
  hostname: 'jsonplaceholder.typicode.com',
  port: 80,
  path: '/posts/1',
  method: 'GET',
  headers: {
    'Content-Type': 'application/json'
  }
};

const req = http.request(options, (res) => {
  let data = '';

  // A chunk of data has been received.
  res.on('data', (chunk) => {
    data += chunk;
  });

  // The whole response has been received.
  res.on('end', () => {
    console.log('Response: ', JSON.parse(data));
  });
});

req.on('error', (error) => {
  console.error('Error: ', error);
});

req.end();
```

2. Using `axios` (Popular 3rd-party library)

`axios` is a widely used library in the Node.js ecosystem because of its simplicity and features like automatic JSON parsing and promise-based API.

Install `axios`:

```
npm install axios
```

Example: Sending a GET request with `axios`

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

```
axios.get('https://jsonplaceholder.typicode.com/posts/1')
  .then(response => {
    console.log('Response: ', response.data);
  })
  .catch(error => {
    console.error('Error: ', error);
  });
```

Example: Sending a POST request with `axios`

```
axios.post('https://jsonplaceholder.typicode.com/posts', {
  title: 'foo',
  body: 'bar',
  userId: 1
})
  .then(response => {
    console.log('Response: ', response.data);
  })
  .catch(error => {
    console.error('Error: ', error);
  });
```

3. Using `node-fetch` (Promise-based Fetch API)

`node-fetch` is a lightweight module that brings the browser's `fetch` functionality to Node.js.

Install `node-fetch`:

```
npm install node-fetch
```

Example: Sending a GET request with `node-fetch`

```
const fetch = require('node-fetch');

fetch('https://jsonplaceholder.typicode.com/posts/1')
  .then(res => res.json())
  .then(data => console.log('Response: ', data))
  .catch(error => console.error('Error: ', error));
```

Example: Sending a POST request with `node-fetch`

```
fetch('https://jsonplaceholder.typicode.com/posts', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({
    title: 'foo',
    body: 'bar',
    userId: 1
  })
})
  .then(res => res.json())
  .then(data => console.log('Response: ', data))
```

```
  .catch(error => console.error('Error: ', error));
```

4. Handling HTTP Requests in Node.js with `express`

For handling incoming HTTP requests (like a server receiving requests), `express` is commonly used.

Install `express`:

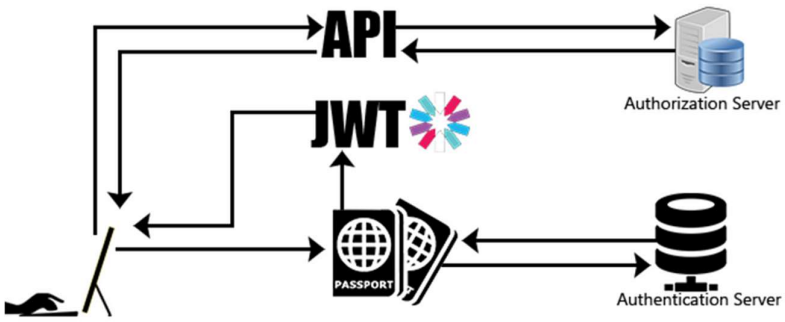
```
npm install express
```

Example: Basic `express` server handling GET and POST requests

```
const express = require('express');
const app = express();
const port = 3000;

// Middleware to parse JSON body
app.use(express.json());

// Handle GET request
app.get('/api/posts/:id', (req, res) => {
  res.json({ id: req.params.id, title: 'Sample Post', body: 'This is a
post' });
});

// Handle POST request
app.post('/api/posts', (req, res) => {
  const post = req.body;
  res.status(201).json({ message: 'Post created', post });
});

// Start server
app.listen(port, () => {
  console.log(`Server is running on http://localhost:${port}`);
});
```

Handling User authentication with NodeJS

Authentication and authorization are two fundamental concepts in web application security. They ensure that users have the right level of access to resources while protecting sensitive data. In this guide, we'll explore how to implement authentication and authorization in Node.js applications, covering best practices, popular libraries, and strategies for securing your Node.js projects.



Understanding Authentication

What Is Authentication?

Authentication is the process of verifying the identity of a user, ensuring they are who they claim to be. This is typically achieved through the use of credentials, such as usernames and passwords.

Authentication Best Practices

1. Use HTTPS: Always use HTTPS to secure data transmission between the client and server, especially when handling login credentials.
2. Password Hashing: Store passwords securely by hashing and salting them. Libraries like `bcrypt` can help with this.
3. Multi-Factor Authentication (MFA): Implement MFA to add an extra layer of security. This could involve something the user knows (password) and something they have (e.g., a mobile app token).
4. Session Management: Use secure and random session tokens to manage user sessions.

Popular Authentication Libraries

1. Passport.js

Passport.js is a widely-used authentication library for Node.js. It supports various authentication strategies, including local (username and password), OAuth, and OpenID. Here's how to set up Passport.js for local authentication:

```
const passport = require('passport');
const LocalStrategy = require('passport-
local').Strategy;
```

```
passport.use(
  new LocalStrategy((username, password, done) => {
    // Verify user credentials here
    if (username === 'user' && password === 'password')
{
      return done(null, { id: 1, username: 'user' });
    } else {
      return done(null, false, { message: 'Invalid
credentials' });
    }
  })
);
```

2. JSON Web Tokens (JWT)

JSON Web Tokens are a popular way to implement authentication and authorization in Node.js. Users receive a token upon login, which they include in subsequent requests. Here's a simple example using the `jsonwebtoken` library:

```
const jwt = require('jsonwebtoken');

// Create a token
const token = jwt.sign({ userId: 1 }, 'secret_key', {
expiresIn: '1h' });

// Verify a token
jwt.verify(token, 'secret_key', (err, decodedToken) =>
{
  if (err) {
    console.error('Token verification failed');
  } else {
    console.log('Decoded token:', decodedToken);
  }
});
```

`jwt.sign(payload, secretOrPrivateKey, options)`:

- **`payload`**: An object with data you want to encode into the token. Here, it's `{ userId: 1 }`, representing a user ID that can be used to identify the user.

- **`secretOrPrivateKey`**: The secret key (`'secret_key'` in this case) is used to sign the token. This key should be kept secure on the server, as it is used to verify the token's validity.
- **`options`**:
    - `{ expiresIn: '1h' }` specifies that the token will expire in **1 hour**. After this time, the token will no longer be valid.

Node.js File System

Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js helps developers to write JavaScript code to run on the server-side, to generate dynamic content and deliver to the web clients. The two features that make Node.js stand-out are:

- Event-driven
- Non-blocking I/O model

About Node.js file system: To handle file operations like creating, reading, deleting, etc., Node.js provides an inbuilt module called FS (File System). Node.js gives the functionality of file I/O by providing wrappers around the standard POSIX functions. All file system operations can have synchronous and asynchronous forms depending upon user requirements. To use this File System module, use the require() method:

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

Common use for File System module:

- Read Files
- Write Files
- Append Files
- Close Files
- Delete Files

What is Synchronous and Asynchronous approach?

- **Synchronous approach:** They are called **blocking functions** as it waits for each operation to complete, only after that, it executes the next operation, hence blocking the next command from execution i.e. a command will not be executed until & unless the query has finished executing to get all the result from previous commands.
- **Asynchronous approach:** They are called **non-blocking functions** as it never waits for each operation to complete, rather it executes all operations in the first go itself. The result of each operation will be handled once the result is available i.e. each command will be executed soon after the execution of the previous command. While the previous command runs in the background and loads the result once it is finished processing the data.

  Example of asynchronous and synchronous: Create a text file named **input.txt** with the following content:

```
A computer science portal
```

-
- Now let us create a js file named **main.js** with the following code:

javascript

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



// Asynchronous read

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

```
   if (err) {

      return console.error(err);

   }

   console.log("Asynchronous read: " + data.toString());

});
```

- **Output:**

```
Asynchronous read A computer science portal
```

javascript

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



// Synchronous read

var data = fs.readFileSync('input.txt');

console.log("Synchronous read: " + data.toString());
```

- **Output:**

```
Synchronous read A computer science portal
```

Open a File: The fs.open() method is used to create, read, or write a file. The fs.readFile() method is only for reading the file and fs.writeFile() method is only for writing to the file, whereas fs.open() method does several operations on a file. First, we need to load the fs class which is a module to access the physical file system. **Syntax:**

```
fs.open(path, flags, mode, callback)
```

Parameters:

1. **`path`**:
   - This is the **path** to the file you want to open.
   - It can either be the file name (if the file is in the current directory) or the full path (if the file is in another directory).
   - Example: `'./file.txt'` or `'/usr/local/file.txt'`.
2. **`flags`**:
   - **Flags** determine how the file should be opened, such as for reading, writing, or appending. Each flag has a different behavior.
   - Here's a breakdown of the most commonly used flags:
     - **`r`**: Open the file for **reading**. An error will be thrown if the file does not exist.
     - **`r+`**: Open the file for **reading and writing**. An error will be thrown if the file does not exist.
     - **`w`**: Open the file for **writing**. If the file does not exist, it is created. If the file exists, it is **truncated** (emptied) before writing.
     - **`wx`**: Like `w` but fails if the file **already exists**.

- **`w+`**: Open the file for **reading and writing**. If the file exists, it is truncated. If the file does not exist, it is created.
- **`a`**: Open the file for **appending** (adding to the end of the file). If the file does not exist, it is created.
- **`ax`**: Like `a`, but fails if the file **already exists**.
- **`a+`**: Open the file for **reading and appending**. If the file does not exist, it is created.
- **`ax+`**: Like `a+`, but fails if the file **already exists**.

Flags control whether you're opening the file for reading, writing, both, or whether you want to overwrite the file, append to it, or throw errors if it already exists.

3. **`mode`** (optional):
   - The `mode` specifies the **file permissions** if the file is created. It is only used when a new file is created.
   - Default mode is `0o666`, which translates to **read and write permission** for the owner, group, and others.
   - Example:
     - `0o644`: Owner has read-write permission, others have read-only.
     - `0o777`: All permissions for everyone.
   - If the file exists, `mode` is ignored.
4. **`callback`**:
   - This is a function that gets executed after the file is opened.
   - The callback has two parameters: `err` and `fd` (file descriptor).
     - **`err`**: If there's an error opening the file (e.g., file not found, permission issues), it will contain the error object.
     - **`fd`**: The file descriptor. This is a reference to the opened file and can be used in other operations such as reading, writing, or closing the file.

Example: Let us create a js file named **main.js** having the following code to open a file **input.txt** for reading and writing.

javascript

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



// Asynchronous - Opening File

console.log("opening file!");

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

   if (err) {

      return console.error(err);

   }
```

```
   console.log("File open successfully");

});
```

Output:

```
opening file!
File open successfully
```

Reading a File: The fs.read() method is used to read the file specified by fd. This method reads the entire file into the buffer. **Syntax:**

```
fs.read(fd, buffer, offset, length, position, callback)
```

Parameters:

- **fd:** This is the file descriptor returned by fs.open() method.
- **buffer:** This is the buffer that the data will be written to.
- **offset:** This is the offset in the buffer to start writing at.
- **length:** This is an integer specifying the number of bytes to read.
- **position:** This is an integer specifying where to begin reading from in the file. If the position is null, data will be read from the current file position.
- **callback:** It is a callback function that is called after reading of the file. It takes two parameters:
    - **err:** If any error occurs.
    - **data:** Contents of the file.

Example: Let us create a js file named **main.js** having the following code:

javascript

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

var buf = new Buffer(1024);



console.log("opening an existing file");

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

   if (err) {

      return console.error(err);

   }

   console.log("File opened successfully!");

   console.log("reading the file");



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

      if (err){

         console.log(err);

      }
```

```
      console.log(bytes + " bytes read");



      // Print only read bytes to avoid junk.

      if(bytes > 0){

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

      }

   });

});
```

Output:

```
opening an existing file
File opened successfully!
reading the file
40 bytes read
A computer science portal
```

Writing to a File: This method will overwrite the file if the file already exists. The fs.writeFile() method is used to asynchronously write the specified data to a file. By default, the file would be replaced if it exists. The 'options' parameter can be used to modify the functionality of the method.

Syntax:

```
fs.writeFile(path, data, options, callback)
```

Parameters:

- **path:** It is a string, Buffer, URL, or file description integer that denotes the path of the file where it has to be written. Using a file descriptor will make it behave similarly to fs.write() method.
- **data:** It is a string, Buffer, TypedArray, or DataView that will be written to the file.
- **options:** It is a string or object that can be used to specify optional parameters that will affect the output. It has three optional parameters:
    - **encoding:** It is a string value that specifies the encoding of the file. The default value is 'utf8'.
    - **mode:** It is an integer value that specifies the file mode. The default value is 0o666.
    - **flag:** It is a string value that specifies the flag used while writing to the file. The default value is 'w'.
- **callback:** It is the function that would be called when the method is executed.
    - **err:** It is an error that would be thrown if the operation fails.

Example: Let us create a js file named **main.js** having the following code:

javascript

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



console.log("writing into existing file");

fs.writeFile('input.txt', 'Hello Students', function(err) {
```

```
   if (err) {
      return console.error(err);
   }

   console.log("Data written successfully!");
   console.log("Let's read newly written data");

   fs.readFile('input.txt', function (err, data) {
      if (err) {
         return console.error(err);
      }
      console.log("Asynchronous read: " + data.toString());
   });
});
```

Output:

```
writing into existing file
Data written successfully!
Let's read newly written data
Asynchronous read: Hello Students
```

Appending to a File:

The fs.appendFile() method is used to synchronously append the data to the file.

Syntax:

```
fs.appendFile(filepath, data, options, callback);
```

or

```
fs.appendFileSync(filepath, data, options,callback);
```

Parameters:

- **filepath:** It is a String that specifies the file path.
- **data:** It is mandatory and it contains the data that you append to the file.
- **options:** It is an optional parameter that specifies the encoding/mode/flag.
- **Callback:** Function is mandatory and is called when appending data to file is completed.

Example 1: Let us create a js file named **main.js** having the following code:

javascript

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


var data = "\nLearn Node.js";


// Append data to file

fs.appendFile('input.txt', data, 'utf8',


    // Callback function

    function(err) {

        if (err) throw err;


        //  If no error

        console.log("Data is appended to file successfully.")

});
```

Output:

```
Data is appended to file successfully.
```

Example 1: For synchronously appending

javascript

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


var data = "\nLearn Node.js";


// Append data to file

fs.appendFileSync('input.txt', data, 'utf8');

console.log("Data is appended to file successfully.")
```

Output:

```
Data is appended to file successfully.
```

- Before Appending Data to input.txt file:

```
A computer science portal
```

- After Appending Data to input.txt file:

```
A computer science portal
Learn Node.js
```

Closing the File: The fs.close() method is used to asynchronously close the given file descriptor thereby clearing the file that is associated with it. This will allow the file descriptor to be reused for other files. Calling fs.close() on a file descriptor while some other operation is being performed on it may lead to undefined behavior. **Syntax:**

```
fs.close(fd, callback)
```

Parameters:

- **fd:** It is an integer that denotes the file descriptor of the file for which to be closed.
- **callback:** It is a function that would be called when the method is executed.
    - **err:** It is an error that would be thrown if the method fails.

Example: Let us create a js file named **main.js** having the following code:

javascript

```
// Close the opened file.

fs.close(fd, function(err) {

   if (err) {

      console.log(err);

   }

   console.log("File closed successfully.");

}
```

Output:

```
File closed successfully.
```

Delete a File:

The fs.unlink() method is used to remove a file or symbolic link from the filesystem. This function does not work on directories, therefore it is recommended to use fs.rmdir() to remove a directory. **Syntax:**

```
fs.unlink(path, callback)
```

Parameters:

- **path:** It is a string, Buffer or URL which represents the file or symbolic link which has to be removed.
- **callback:** It is a function that would be called when the method is executed.
    - **err:** It is an error that would be thrown if the method fails.

Example: Let us create a js file named **main.js** having the following code:

javascript

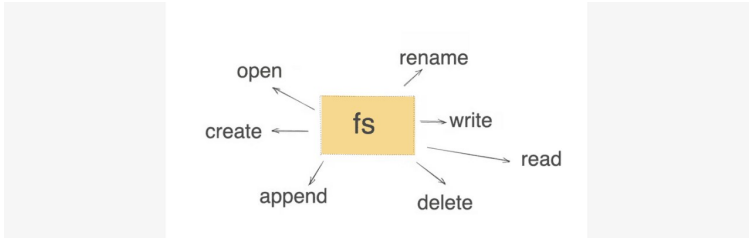
```
var fs = require("fs");
```

```
console.log("deleting an existing file");

fs.unlink('input.txt', function(err) {

   if (err) {

      return console.error(err);

   }

   console.log("File deleted successfully!");

});
```

Output:

```
deleting an existing file
File deleted successfully!
```

What is Node JS 'fs' Module?



'fs' Module is inbuilt provided by Node.js to perform different actions. The 'fs' Module is a file system module that helps you work with files, like read files, node.js write files, append files, and many more. It operates both, synchronously and asynchronously. Let's look at an example of how to use the fs module with node js write to file capabilities.

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

Node.js Events

The Node.js API is based on an event-driven architecture. It includes the events module, which provides the capability to create and handle custom events. The event module contains EventEmitter class. The EventEmitter object emits named events. Such events call the listener functions. The Event Emitters have a very crucial role the Node.js ecosystem. Many objects in a Node emit events, for example, a net.Server object emits an event each time a peer connects to it, or a connection is closed. The fs.readStream object emits an event when the file is opened, closed, a read/write operation is performed. All objects which emit events are the instances of events.EventEmitter.

Since the EvenEmitter class is defined in events module, we must include in the code with require statement.

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

To emit an event, we should declare an object of EventEmitter class.

```
var eventEmitter = new events.EventEmitter();
```

When an EventEmitter instance faces any error, it emits an 'error' event. When a new listener is added, 'newListener' event is fired and when a listener is removed, 'removeListener' event is fired.

Sr.No.	Events & Description
1	**newListener(event, listener)** ■ **event** – String: the event name ■ **listener** – Function: the event handler function This event is emitted any time a listener is added. When this event is triggered, the listener may not yet have been added to the array of listeners for the event.
2	**removeListener(event, listener)** ■ **event** – String The event name ■ **listener** – Function The event handler function This event is emitted any time someone removes a listener. When this event is triggered, the listener may not yet have been removed from the array of listeners for the event.

Following instance methods are defined in EventEmitter class –

Sr.No.	Events & Description
1	**addListener(event, listener)** Adds a listener at the end of the listeners array for the specified event. Returns emitter, so calls can be chained.
2	**on(event, listener)** Adds a listener at the end of the listeners array for the specified event. Same as addListener.
3	**once(event, listener)** Adds a one time listener to the event. This listener is invoked only the next time the event is fired, after which it is removed
4	**removeListener(event, listener)** Removes a listener from the listener array for the specified event. If any single listener has been added multiple times to the listener array for the specified event, then removeListener must be called multiple times to remove each instance.
5	**removeAllListeners([event])** Removes all listeners, or those of the specified event. It's not a good idea to remove listeners that were added elsewhere in the code, especially when it's on an emitter that you didn't create (e.g. sockets or file streams).
6	**setMaxListeners(n)** By default, EventEmitters will print a warning if more than 10 listeners are added for a particular event. Set to zero for unlimited.
7	**listeners(event)** Returns an array of listeners for the specified event.
8	**emit(event, [arg1], [arg2], [...])** Execute each of the listeners in order with the supplied arguments. Returns true if the event had listeners, false otherwise.
9	**off(event, listener)** Alias for removeListener

```
var events = require('events');
var eventEmitter = new events.EventEmitter();

// listener #1
var listner1 = function listner1() {
   console.log('listner1 executed.');
}

// listener #2
var listner2 = function listner2() {
   console.log('listner2 executed.');
}
```

Let us bind these listeners to a connection event. The first function listener1 is registered with addListener() method, while we use on() method to bind listener2.

```
// Bind the connection event with the listner1 function
eventEmitter.addListener('connection', listner1);

// Bind the connection event with the listner2 function
eventEmitter.on('connection', listner2);

// Fire the connection event
eventEmitter.emit('connection');
```

When the connection event is fired with emit() method, the console shows the log message in the listeners as above

```
listner1 executed.
listner2 executed.
```

Let us remove the listener2 callback from the connection event, and fire the connection event again.

```
// Remove the binding of listner1 function
eventEmitter.removeListener('connection', listner1);
console.log("Listner1 will not listen now.");

// Fire the connection event
eventEmitter.emit('connection');
```

The console now shows the following log –

```
listner1 executed.
listner2 executed.
Listner1 will not listen now.
listner2 executed.
```

The EventEmitter class also has a listCount() method. It is a class method, that returns the number of listeners for a given event.

Open Compiler

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

const myEmitter = new events.EventEmitter();
// listener #1
var listner1 = function listner1() {
   console.log('listner1 executed.');
}

// listener #2
var listner2 = function listner2() {
  console.log('listner2 executed.');
```

```
}

// Bind the connection event with the listner1 function
myEmitter.addListener('connection', listner1);

// Bind the connection event with the listner2 function
myEmitter.on('connection', listner2);

// Fire the connection event
myEmitter.emit('connection');
console.log("Number of Listeners:" + myEmitter.listenerCount('connection'));
```

```
Output:

listner1 executed.
listner2 executed.
Number of Listeners:2
```

Inheriting Events and Returning event emitter:

When inheriting from the `EventEmitter` class, you can create your own custom class that extends `EventEmitter`. This allows your class to emit events and listen for them internally or externally. Additionally, you can return an instance of the `EventEmitter` to allow event-driven behavior outside the class.

Here's how you can inherit from `EventEmitter` and return it:

Inheriting from `EventEmitter`

1. **Create a Class**: You can create a class that extends the `EventEmitter` class.
2. **Emit Events**: The class can emit events when certain methods are called.
3. **Return the Event Emitter**: You can return the `this` context or the instance of the `EventEmitter` for chaining purposes.

Example: Inheriting and Returning `EventEmitter`

```
const EventEmitter = require('events');

// Create a custom class that extends EventEmitter
class MyClass extends EventEmitter {
  constructor() {
    super(); // Call the parent constructor (EventEmitter)
  }

  // A method that emits an event
  doSomething() {
    console.log('Doing something...');

    // Emit an event
    this.emit('done', 'Task completed');
```

```
    // Return the event emitter instance for chaining
    return this;
  }

  // Another method
  doAnotherThing() {
    console.log('Doing another thing...');

    // Emit another event
    this.emit('anotherDone', 'Another task completed');

    // Return the event emitter instance for chaining
    return this;
  }
}

// Create an instance of the class
const myInstance = new MyClass();

// Listen for the 'done' event
myInstance.on('done', (message) => {
  console.log(message);
});

// Listen for the 'anotherDone' event
myInstance.on('anotherDone', (message) => {
  console.log(message);
});

// Invoke methods and emit events
myInstance.doSomething().doAnotherThing();
```

Key Points:

1. **Inheritance**: The `MyClass` class extends the `EventEmitter` class, so it inherits all its methods, such as `on`, `emit`, and `once`.
2. **Chaining Methods**: By returning `this` in each method, you allow for method chaining. In the example, `myInstance.doSomething().doAnotherThing()` calls both methods in sequence.
3. **Event Emission**: Events are emitted within the methods using `this.emit(eventName, args...)`, allowing external listeners to handle them.

Output:

```
Doing something...
Task completed
Doing another thing...
Another task completed
```

Express.js Tutorial

-

Express.js is a fast, flexible and minimalist web framework for Node.js. It's effectively a tool that simplifies building web applications and APIs using JavaScript on the server side. Express is an open-source that is developed and maintained by the Node.js foundation.
Express.js offers a robust set of features that enhance your productivity and streamline your web application. It makes it easier to organize your application's functionality with middleware and routing. It adds helpful utilities to Node HTTP objects and facilitates the rendering of dynamic HTTP objects.

Why learn Express?

Express is a user-friendly framework that simplifies the development process of Node applications. It uses JavaScript as a programming language and provides an efficient way to build web applications and APIs. With Express, you can easily handle routes, requests, and responses, which makes the process of creating robust and scalable applications much easier. Moreover, it is a lightweight and flexible framework that is easy to learn and comes loaded with middleware options. Whether you are a beginner or an experienced developer, Express is a great choice for building your application.

Getting Started Express

1. Installation: Install Express using npm:

```
npm install express
```

2. Basic Example of an Express App:

Node

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


// Define routes and middleware here
// ...


const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
    console.log(`Server running on port ${PORT}`);
});
```

Explanation:

- Import the '**express**' module to create a web application using Node.js.
- Initialize an Express app using **`const app = express();`.**
- Add **routes (endpoints)** and **middleware** functions to handle requests and perform tasks like authentication or logging.
- Specify a port **(defaulting to 3000)** for the server to listen on.

Key Features of Express

1. **Middleware and Routing**: Define clear pathways (routes) within your application to handle incoming HTTP requests (GET, POST, PUT, DELETE) with ease. Implement reusable functions (middleware) to intercept requests and create responses, adding functionalities like authentication, logging, and data parsing.
2. **Minimalistic Design**: Express.js follows a simple and minimalistic design philosophy. This simplicity allows you to quickly set up a server, define routes, and handle HTTP requests efficiently. It's an excellent choice for building web applications without unnecessary complexity.
3. **Flexibility and Customization**: Express.js doesn't impose a strict application architecture. You can structure your code according to your preferences. Whether you're building a RESTful API or a full-fledged web app, Express.js adapts to your needs.
4. **Templating Power:** Incorporate templating engines like Jade or EJS to generate dynamic HTML content, enhancing user experience.
5. **Static File Serving:** Effortlessly serve static files like images, CSS, and JavaScript from a designated directory within your application.
6. **Node.js Integration:** Express.js seamlessly integrates with the core functionalities of Node.js, allowing you to harness the power of asynchronous programming and event-driven architecture.

Applications of Express

Express.js empowers you to construct a wide array of web applications. Here are some captivating examples:

- **RESTful APIs:** Develop robust APIs that adhere to the REST architectural style, enabling communication with other applications and front-end interfaces.
- **Real-time Applications:** Leverage Express.js's event-driven nature to create real-time applications like chat or collaborative editing tools.
- **Single-Page Applications (SPAs):** Craft SPAs that fetch and update content dynamically on the client-side, offering a seamless user experience.

Node.js - Express Framework

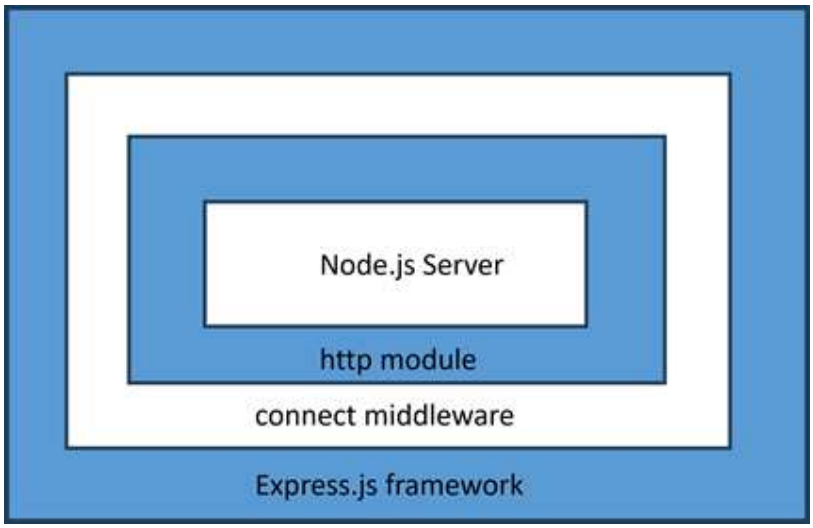
Express.js is a minimal and flexible web application framework that provides a robust set of features to develop Node.js based web and mobile applications. Express.js is one of the most popular web frameworks in the Node.js ecosystem. Express.js provides all the features of a modern web framework, such as templating, static file handling, connectivity with SQL and NoSQL databases.

Node.js has a built-in web server. The createServer() method in its http module launches an asynchronous http server. It is possible to develop a web application with core Node.js features. However, all the low level manipulations of HTTP request and responses have to be tediously handled. The web application frameworks take care of these common tasks, allowing the developer to concentrate on the business logic of the application. A web framework such as Express.js is a set of utilities that facilitates rapid, robust and scalable web applications.

Following are some of the core features of Express framework –

- Allows to set up middlewares to respond to HTTP Requests.
- Defines a routing table which is used to perform different actions based on HTTP Method and URL.
- Allows to dynamically render HTML Pages based on passing arguments to templates.

The Express.js is built on top of the connect middleware, which in turn is based on http, one of the core modules of Node.js API.



Installing Express

The Express.js package is available on npm package repository. Let us install express package locally in an application folder named ExpressApp.

```
D:\expressApp> npm init
D:\expressApp> npm install express --save
```

The above command saves the installation locally in the node_modules directory and creates a directory express inside node_modules.

Hello world Example

Following is a very basic Express app which starts a server and listens on port 5000 for connection. This app responds with Hello World! for requests to the homepage. For every other path, it will respond with a 404 Not Found.

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

app.get('/', function (req, res) {
   res.send('Hello World');
})

var server = app.listen(5000, function () {
   console.log("Express App running at http://127.0.0.1:5000/");
})
```

Save the above code as index.js and run it from the command-line.

```
D:\expressApp> node index.js
Express App running at http://127.0.0.1:5000/
```

Visit http://localhost:5000/ in a browser window. It displays the Hello World message.

Application object

An object of the top level express class denotes the application object. It is instantiated by the following statement –

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

The Application object handles important tasks such as handling HTTP requests, rendering HTML views, and configuring middleware etc.

The app.listen() method creates the Node.js web server at the specified host and port. It encapsulates the createServer() method in http module of Node.js API.

```
app.listen(port, callback);
```

Basic Routing

The app object handles HTTP requests GET, POST, PUT and DELETE with app.get(), app.post(), app.put() and app.delete() method respectively. The HTTP request and HTTP response objects are provided as arguments to these methods by the NodeJS server. The first parameter to these methods is a string that represents the endpoint of the URL. These methods are asynchronous, and invoke a callback by passing the request and response objects.

GET method

In the above example, we have provided a method that handles the GET request when the client visits '/' endpoint.

```
app.get('/', function (req, res) {
  res.send('Hello World');
})
```

- Request Object – The request object represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, and so on.
- Response Object – The response object represents the HTTP response that an Express app sends when it gets an HTTP request. The send() method of the response object formulates the server's response to the client.

You can print request and response objects which provide a lot of information related to HTTP request and response including cookies, sessions, URL, etc.

The response object also has a sendFile() method that sends the contents of a given file as the response.

```
res.sendFile(path)
```

Save the following HTML script as index.html in the root folder of the express app.

```
<html>
<body>
<h2 style="text-align: center;">Hello World</h2>
</body>
</html>
```

Change the index.js file to the code below –

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

app.get('/', function (req, res) {
  res.sendFile(path.join(    dirname,"index.html"));
})

var server = app.listen(5000, function () {

  console.log("Express App running at http://127.0.0.1:5000/");
})
```

Run the above program and visit http://localhost:5000/, the browser shows Hello World message as below:

Let us use sendFile() method to display a HTML form in the index.html file.

```
<html>
  <body>

    <form action = "/process_get" method = "GET">
      First Name: <input type = "text" name = "first_name"> <br>
      Last Name: <input type = "text" name = "last_name"> <br>
      <input type = "submit" value = "Submit">
    </form>

  </body>
</html>
```

The above form submits the data to /process_get endpoint, with GET method. Hence we need to provide a app.get() method that gets called when the form is submitted.

```
app.get('/process_get', function (req, res) {
  // Prepare output in JSON format
  response = {
    first_name:req.query.first_name,
    last_name:req.query.last_name
  };
  console.log(response);
  res.end(JSON.stringify(response));
})
```

The form data is included in the request object. This method retrieves the data from request.query array, and sends it as a response to the client.

The complete code for index.js is as follows –

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

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

app.get('/', function (req, res) {
  res.sendFile(path.join(   dirname,"index.html"));
})

app.get('/process_get', function (req, res) {
  // Prepare output in JSON format
  response = {
```

```
    first_name:req.query.first_name,
    last_name:req.query.last_name
  };
  console.log(response);
  res.end(JSON.stringify(response));
})

var server = app.listen(5000, function () {
  console.log("Express App running at http://127.0.0.1:5000/");
})
```

Visit http://localhost:5000/.

Now you can enter the First and Last Name and then click submit button to see the result and it should return the following result –

```
{"first_name":"John","last_name":"Paul"}
```

POST method

The HTML form is normally used to submit the data to the server, with its method parameter set to POST, especially when some binary data such as images is to be submitted. So, let us change the method parameter in index.html to POST, and action parameter to "process_POST".

```
<html>
  <body>

    <form action = "/process_POST" method = "POST">
      First Name: <input type = "text" name = "first_name">  <br>
      Last Name: <input type = "text" name = "last_name">  <br>
      <input type = "submit" value = "Submit">
    </form>

  </body>
</html>
```

To handle the POST data, we need to install the body-parser package from npm. Use the following command.

```
npm install body-parser –save
```

This is a node.js middleware for handling JSON, Raw, Text and URL encoded form data.

This package is included in the JavaScript code with the following require statement.

```
var bodyParser = require('body-parser');
```

The urlencoded() function creates application/x-www-form-urlencoded parser

```
// Create application/x-www-form-urlencoded parser
var urlencodedParser = bodyParser.urlencoded({ extended: false })
```

Add the following app.post() method in the express application code to handle POST data.

```
app.post('/process_post', urlencodedParser, function (req, res) {
   // Prepare output in JSON format
   response = {
      first_name:req.body.first_name,
      last_name:req.body.last_name
   };
   console.log(response);
   res.end(JSON.stringify(response));
})
```

Here is the complete code for index.js file

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

var bodyParser = require('body-parser');
// Create application/x-www-form-urlencoded parser
var urlencodedParser = bodyParser.urlencoded({ extended: false })

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

app.get('/', function (req, res) {
   res.sendFile(path.join(__dirname,"index.html"));
})

app.get('/process_get', function (req, res) {
   // Prepare output in JSON format
   response = {
      first_name:req.query.first_name,
      last_name:req.query.last_name
   };
   console.log(response);
   res.end(JSON.stringify(response));
})
app.post("/process_post", )
var server = app.listen(5000, function () {
   console.log("Express App running at http://127.0.0.1:5000/");
})
```

Run index.js from command prompt and visit http://localhost:5000/.

Now you can enter the First and Last Name and then click the submit button to see the following result –

```
{"first_name":"John","last_name":"Paul"}
```

Serving Static Files

Express provides a built-in middleware express.static to serve static files, such as images, CSS, JavaScript, etc.

You simply need to pass the name of the directory where you keep your static assets, to the express.static middleware to start serving the files directly. For example, if you keep your images, CSS, and JavaScript files in a directory named public, you can do this –

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

We will keep a few images in public/images sub-directory as follows –

```
node_modules
index.js
public/
public/images
public/images/logo.png
```

Let's modify "Hello Word" app to add the functionality to handle static files.

```
var express = require('express');
var app = express();
app.use(express.static('public'));

app.get('/', function (req, res) {
   res.send('Hello World');
})

var server = app.listen(5000, function () {
   console.log("Express App running at http://127.0.0.1:5000/");
})
```

Save the above code in a file named index.js and run it with the following command.

```
D:\expressApp> node index.js
```

Now open http://127.0.0.1:5000/images/logo.png in any browser and see observe following result.

To learn Express.js in details, visit our ExpressJS Tutorial (ExpressJS)

Express.js Routing

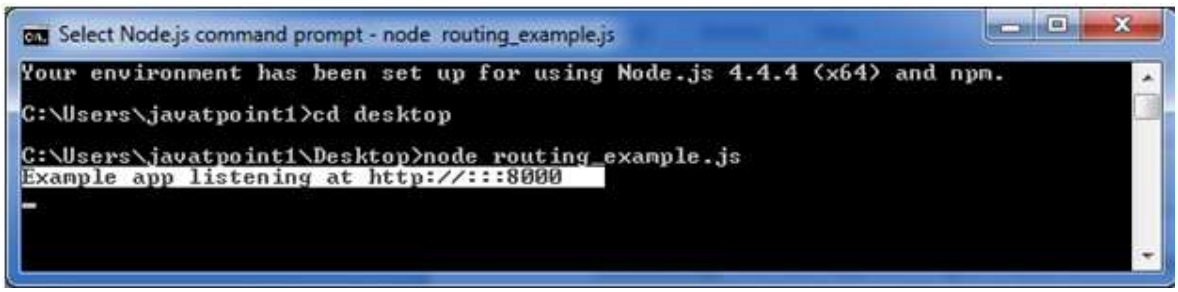
Routing is made from the word route. It is used to determine the specific behavior of an application. It specifies how an application responds to a client request to a particular route, URI or path and a specific HTTP request method (GET, POST, etc.). It can handle different types of HTTP requests.

Let's take an example to see basic routing.

File: 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.');
```

```
23. })
24. var server = app.listen(8000, function () {
25. var host = server.address().address
26.   var port = server.address().port
27. console.log("Example app listening at http://%s:%s", host, port)
28. })
```



You see that server is listening.

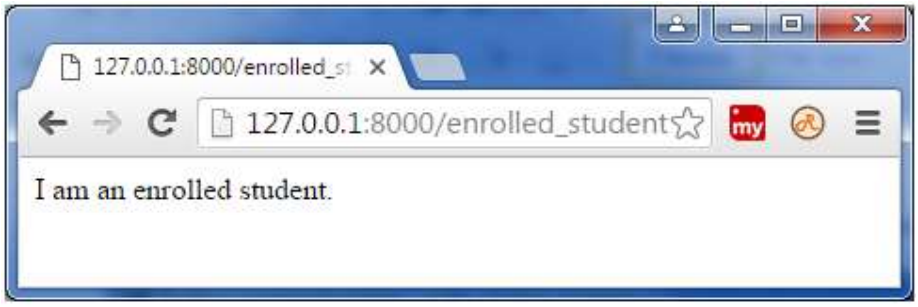
Now, you can see the result generated by server on the local host http://127.0.0.1:8000

Output:

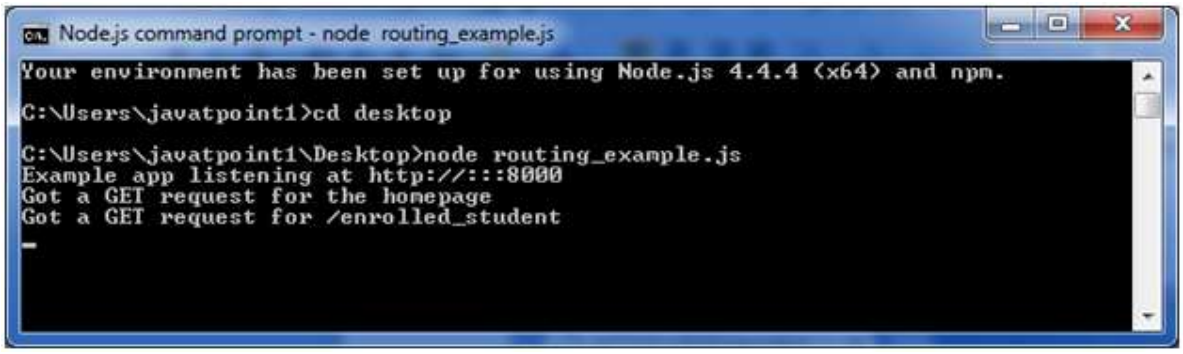
This is the homepage of the example app.

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:

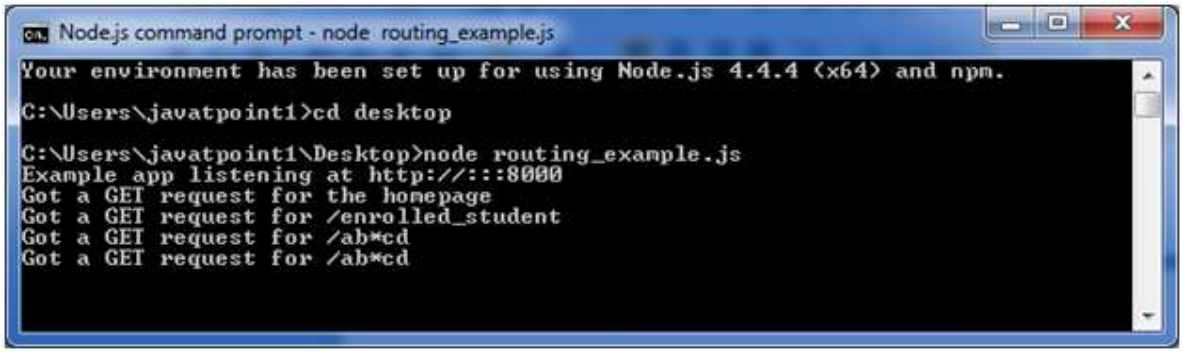


This can read the pattern like abcd, abxcd, ab123cd, and so on.

Next route **http://127.0.0.1:8000/abcd**

Next route http://127.0.0.1:8000/ab12345cd

Updated command prompt:



Express.js Middleware

Express.js Middleware are different types of functions that are invoked by the Express.js routing layer before the final request handler. As the name specified, Middleware appears in the middle between an initial request and final intended route. In stack, middleware functions are always invoked in the order in which they are added.

Middleware is commonly used to perform tasks like body parsing for URL-encoded or JSON requests, cookie parsing for basic cookie handling, or even building JavaScript modules on the fly.

What is a Middleware function

Middleware functions are the functions that access to the request and response object (req, res) in request-response cycle.

A middleware function can perform the following tasks:

- It can execute any code.
- It can make changes to the request and the response objects.
- It can end the request-response cycle.
- It can call the next middleware function in the stack.

Express.js Middleware

Following is a list of possibly used middleware in Express.js app:

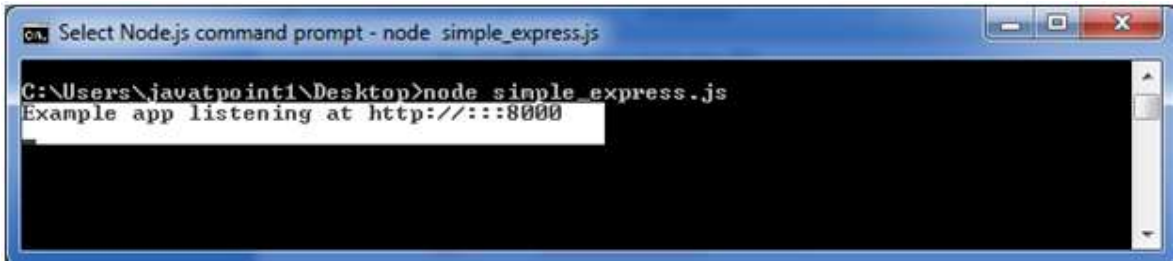
- Application-level middleware
- Router-level middleware
- Error-handling middleware
- Built-in middleware
- Third-party middleware

Let's take an example to understand what middleware is and how it works.

Let's take the most basic Express.js app:

File: simple_express.js

```
1. var express = require('express');
2. var app = express();
3.
4. app.get('/', function(req, res) {
5.   res.send('Welcome to JavaTpoint!');
6. });
7. app.get('/help', function(req, res) {
8.   res.send('How can I help You?');
9. });
10. var server = app.listen(8000, function () {
11.   var host = server.address().address
12.   var port = server.address().port
13. console.log("Example app listening at http://%s:%s", host, port)
14. })
```



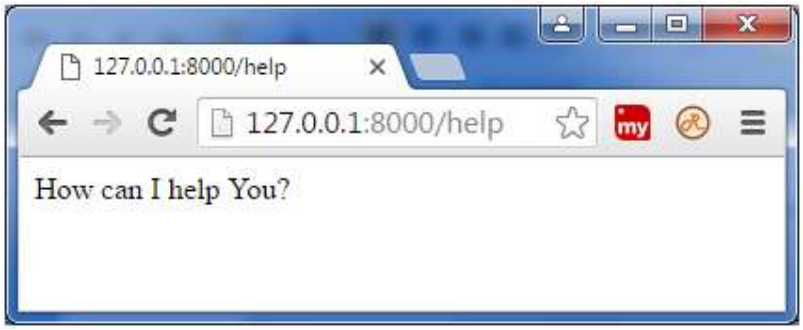
You see that server is listening.

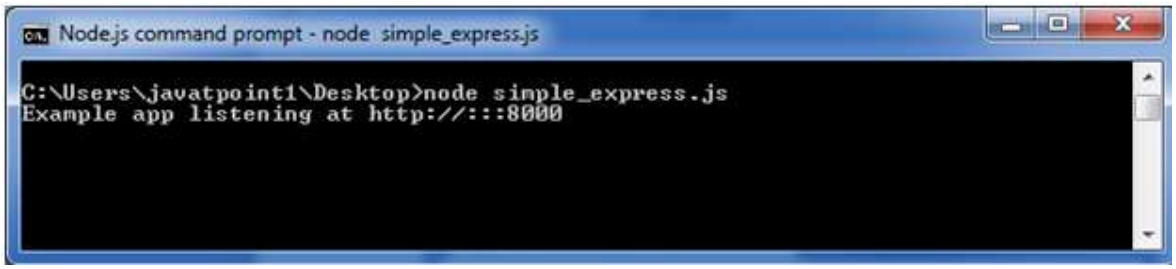
Now, you can see the result generated by server on the local host **http://127.0.0.1:8000**

Output:

Let's see the next page: **http://127.0.0.1:8000/help**

Output:





Note: You see that the command prompt is not changed. Means, it is not showing any record of the GET request although a GET request is processed in the **http://127.0.0.1:8000/help page**.

Use of Express.js Middleware

If you want to record every time you a get a request then you can use a middleware.

See this example:

File: simple_middleware.js

```
var express = require('express');
var app = express();
app.use(function(req, res, next) {
  console.log('%s %s', req.method, req.url);
  next();
});
app.get('/', function(req, res, next) {
  res.send('Welcome to JavaTpoint!');
});
app.get('/help', function(req, res, next) {
  res.send('How can I help you?');
});
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)
})
```

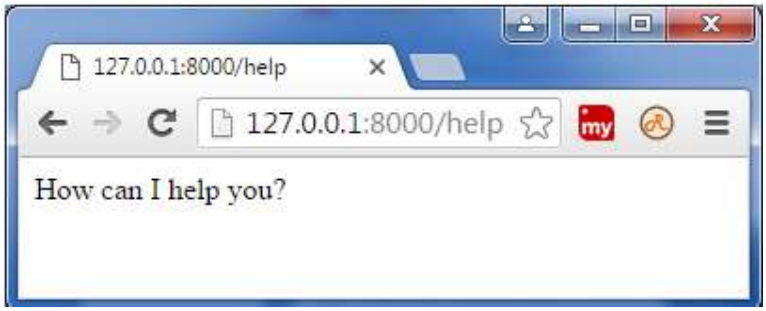
You see that server is listening.

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

Output:

You can see that output is same but command prompt is displaying a GET result.

Go to **http://127.0.0.1:8000/help**



As many times as you reload the page, the command prompt will be updated.

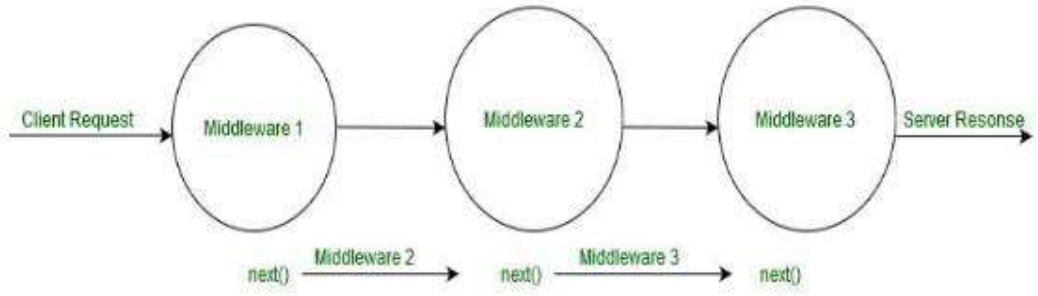
Note: In the above example next() middleware is used.

Middleware example explanation

- In the above middleware example a new function is used to invoke with every request via **app.use()**.
- Middleware is a function, just like route handlers and invoked also in the similar manner.
- You can add more middlewares above or below using the same API.

Express.js is the most powerful framework of the node.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. Express.js use different kinds of middleware functions in order to complete the different requests made by the client for e.g. client can make get, put, post, and delete requests these requests can easily handle by these middleware functions.

Working of the middleware functions:



Custom Middlewares:
We can create multiple Custom middleware using express.js according to the routing of the request and also forward the request to the next middleware.

Syntax:

```
app.<Middlewaretype>(path,(req,res,next))
```

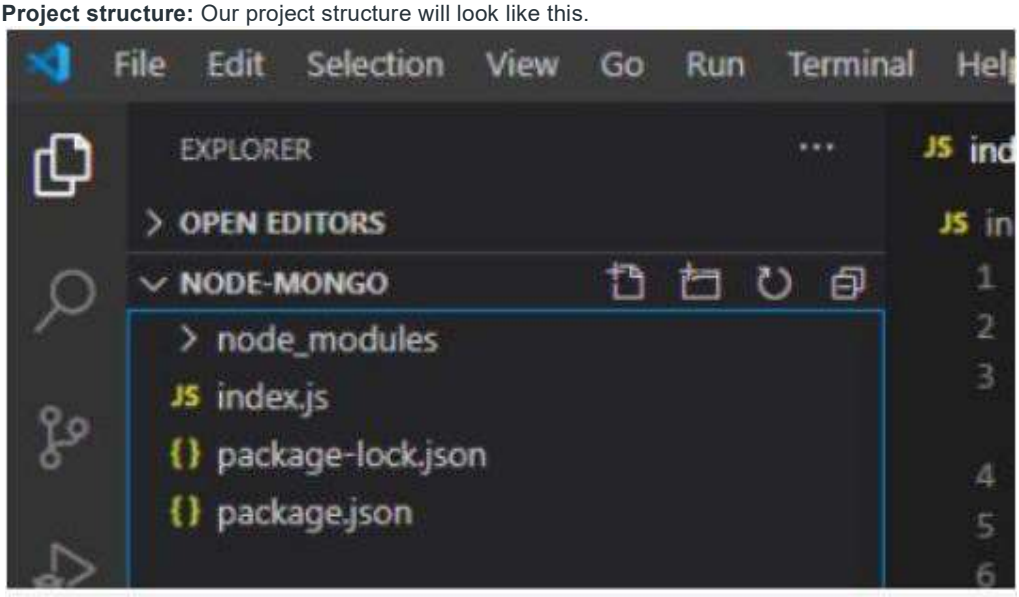
Parameters: Custom Middleware takes the following two parameters:

- **path:** The path or path pattern or in regular expression by which particular Middleware will be called.
- **callback:** The second parameter is the callback function that takes three parameters request, response, and next() function as an argument.

Installing module: Install the express module using the following command.

```
npm install express
```

Project structure: Our project structure will look like this.



- index.js

```
// Requiring module

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

```
// Creating express app object

const app = express();



app.post("/check",(req,res,next)=>{

  res.send("This is the post request")

  next()

})



app.get("/gfg",(req,res,next)=>{

  res.send("This is the get request")

  res.end()

})



// Server setup

app.listen(3000, () => {

  console.log("Server is Running");

})
```

Run **index.js** file using below command:

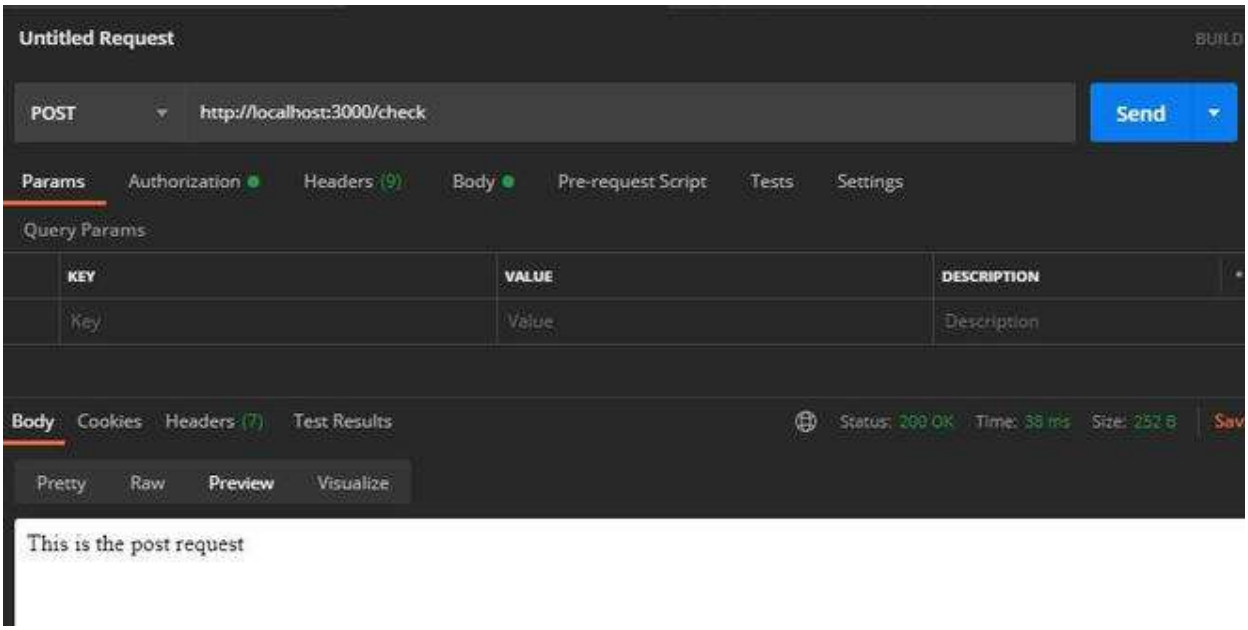
```
node index.js
```

Output:

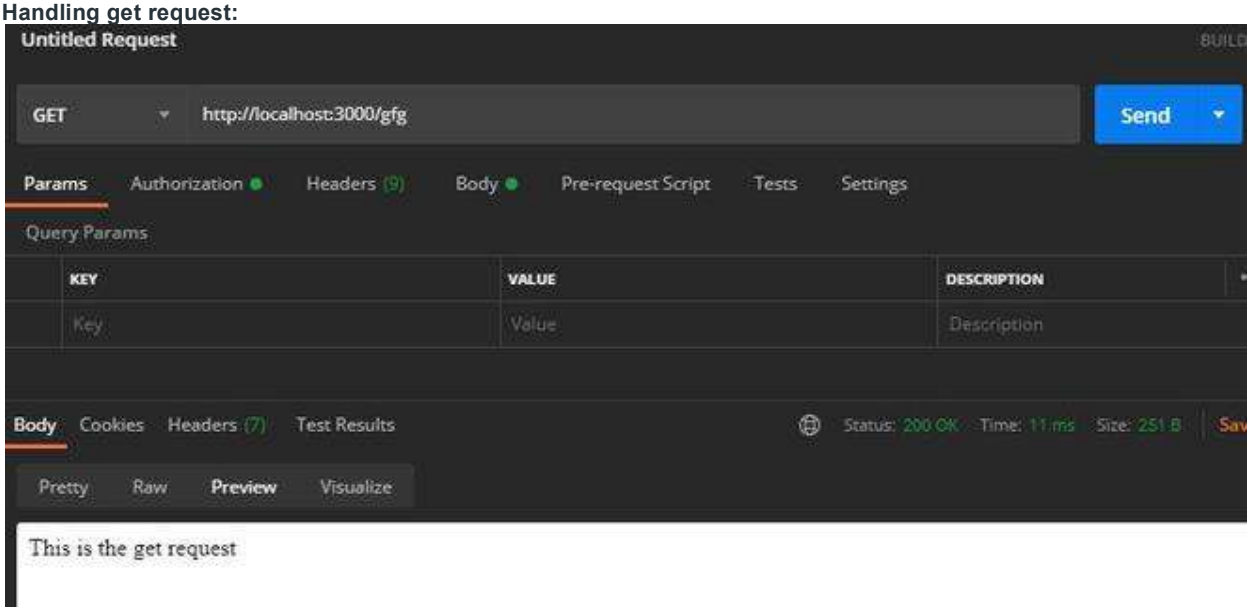
```
Server is Running
```

Now open the postman tool and send the following requests:

- **Handling Post request:**



- **Handling get request:**



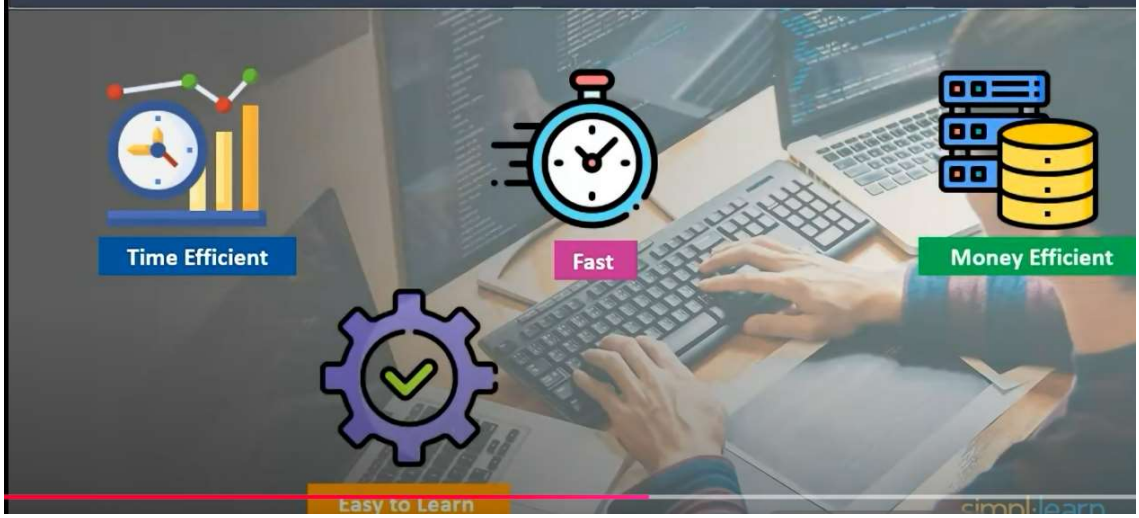
What is Express JS ?

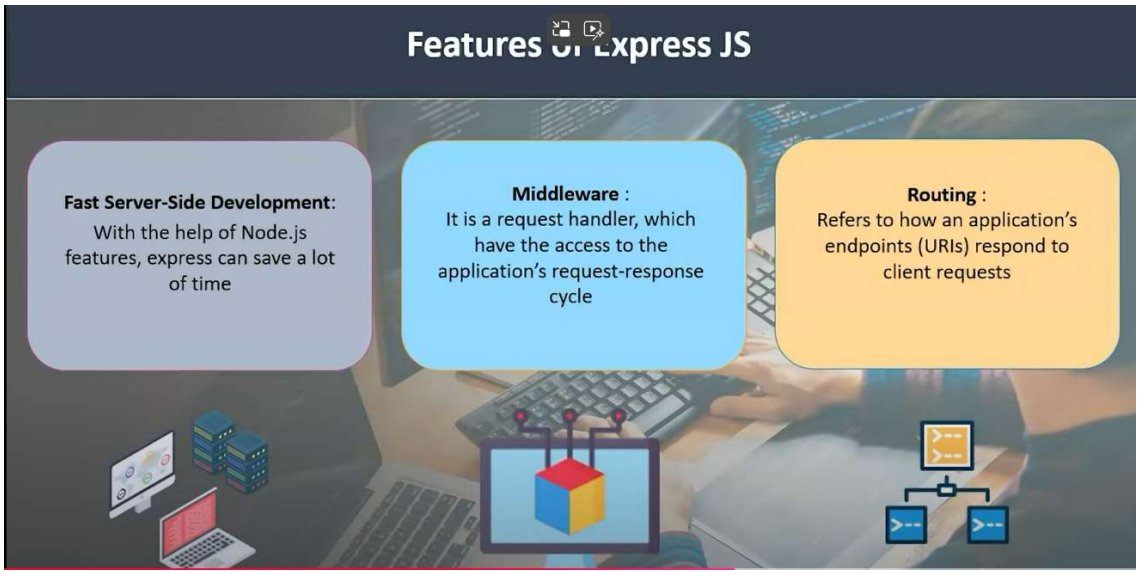
Express is a Node.js framework designed for building APIs, web applications and cross-platform mobile apps

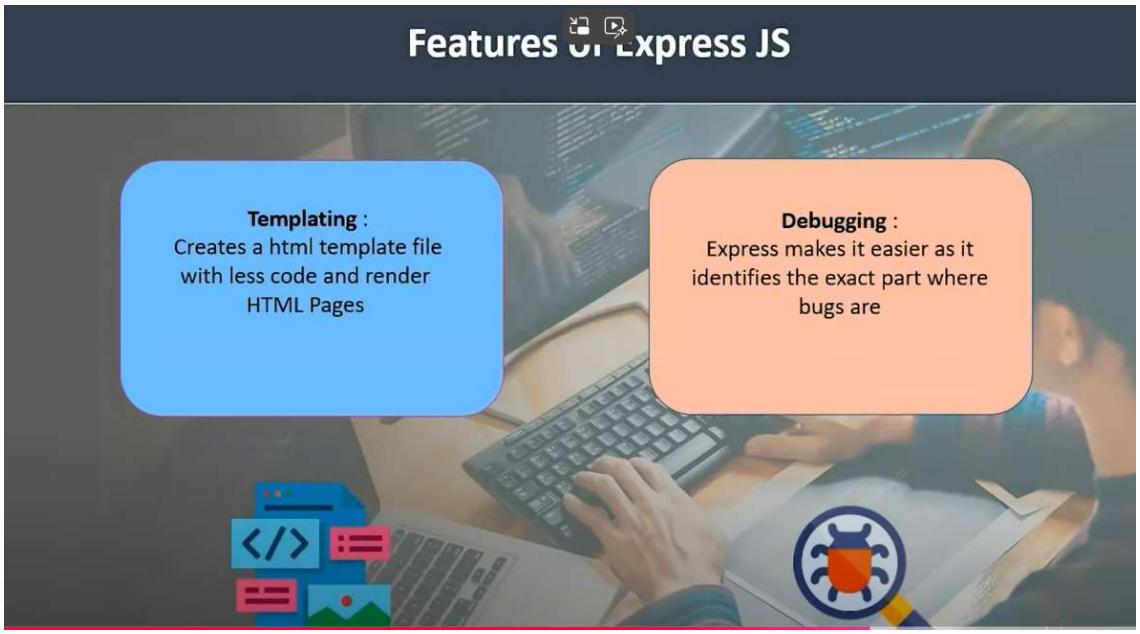
Express is high performance, fast, unopinionated, and lightweight

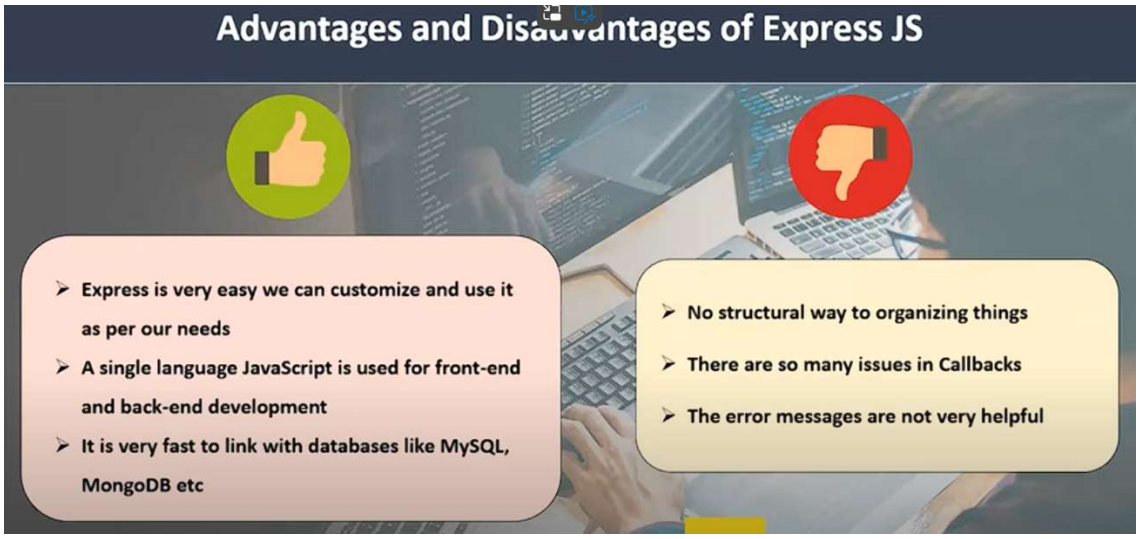
It is used as a server-side scripting language

Need of Express JS




Features of Express JS
Fast Server-Side Development:
With the help of Node.js features, express can save a lot of time
Middleware :
It is a request handler, which have the access to the application's request-response cycle
Routing :
Refers to how an application's endpoints (URIs) respond to client requests


Features of Express JS
Templating :
Creates a html template file with less code and render HTML Pages
Debugging :
Express makes it easier as it identifies the exact part where bugs are



Using template engines with Express

A ***template engine*** enables you to use static template files in your application. At runtime, the template engine replaces variables in a template file with actual values, and transforms the template into an HTML file sent to the client. This approach makes it easier to design an HTML page.

Some popular template engines that work with Express are Pug, Mustache, and EJS. The Express application generator uses Jade as its default, but it also supports several others.

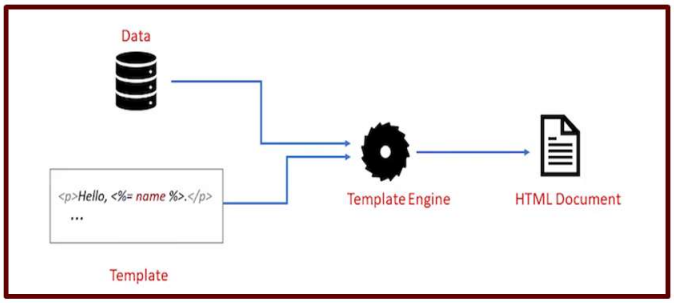
See Template Engines (Express wiki) for a list of template engines you can use with Express. See also Comparing JavaScript Templating Engines: Jade, Mustache, Dust and More.

Server-side rendering involves generating HTML on the server and sending it to the client, as opposed to generating it on the client side using JavaScript. This improves initial load time, and SEO, and enables dynamic content generation. Express is a popular web application framework for NodeJS, and EJS is a simple templating language that lets you generate HTML with plain JavaScript.

Template Engine in Express JS

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

- **EJS Embedded JavaScript Template**
- **Pug (formerly known as jade)**
- mustache
- dust
- atpl
- eco
- ect
- ejs
- haml
- haml-coffee
- handlebars
- hogan



EJS Template Engine in Express JS

- 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.

  For example: app.set('views', './views').
- **view engine:** It specifies the template engine that you use. **For example**, to use the EJS template engine: **app.set('view engine', 'ejs').**

Steps to Create Application (And Installing Required Modules):

Step 1: Create a new directory for your project:

```
mkdir express-ssr
```

Step 2: Navigate into the project directory:

```
cd express-ssr
```

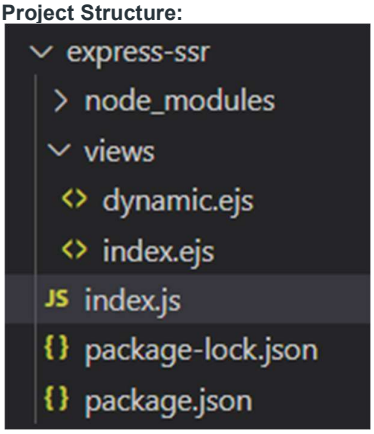
Step 3: Initialize npm (Node Package Manager) to create a package.json file:

```
npm init -y
```

Step 4: Install required modules (Express and EJS) using npm:

```
npm install express ejs
```

Project Structure:



The updated dependencies in **package.json** file will look like:

```
"dependencies": {
    "ejs": "^3.1.9",
    "express": "^4.18.3"
}
```

```ejs
//views/index.ejs
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Express SSR with EJS</title>
</head>
<body>
    <h1>Hello, <%= name %>!</h1>
</body>
</html>
```

This file is an EJS template used for rendering a webpage with dynamic content passed from the server i.e (index.js)

Components

1. **`<!DOCTYPE html>`**:
    - This declares the document type and version (HTML5 in this case) to the browser.
2. **`<html lang="en">`**:
    - Defines the root of the document with the language set to English.
3. **`<head>` section**:
    - **`<meta charset="UTF-8">`**: Ensures that the page uses UTF-8 encoding for handling text and special characters.
    - **`<meta name="viewport" content="width=device-width, initial-scale=1.0">`**: Makes the page responsive by setting the width to the device's screen width.
    - **`<title>Express SSR with EJS</title>`**: Sets the title of the page.
4. **`<body>` section**:
    - **`<h1>Hello, <%= name %>!</h1>`**: This is where EJS renders dynamic content.
        - **`<%= name %>`** is EJS syntax for embedding the value of `name` in the HTML. When you use the `render` method in Express, this variable is passed from the server-side code.

```ejs
//views/dynamic.ejs
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Express SSR with EJS</title>
</head>
<body>
    <h1>Dynamic Content: <%= data %></h1>
</body>
</html>
```

The code you provided is another EJS template, but this time it's used to render dynamic content based on the `data` variable passed from the index.js server.

Components

1. **`<!DOCTYPE html>`**:
   - Specifies the document type as HTML5.
2. **`<html lang="en">`**:
   - Defines the document language as English.
3. **`<head>` section**:
   - **`<meta charset="UTF-8">`**: Ensures the page uses UTF-8 encoding for character representation.
   - **`<meta name="viewport" content="width=device-width, initial-scale=1.0">`**: Makes the page responsive by setting the width to match the device's screen.
   - **`<title>Express SSR with EJS</title>`**: Sets the title of the page.
4. **`<body>` section**:
   - **`<h1>Dynamic Content: <%= data %></h1>`**: Displays the dynamic content passed to the template.
     - **`<%= data %>`**: EJS syntax for outputting a value. This is where the `data` value (passed from Express) will be displayed.

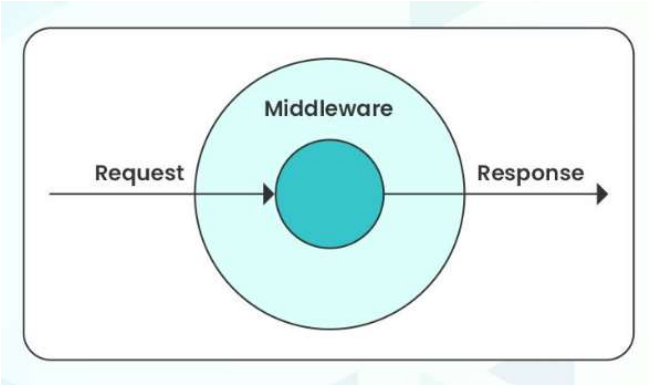
```
const express = require('express');
const app = express();
const path = require('path');

// Set the view engine to EJS
app.set('view engine', 'ejs');
app.set('views', path.join(__dirname, 'views'));

// Basic Approach: Render EJS template with static data
app.get('/', (req, res) => {
    res.render('index', { name: 'World' });
});

/*
Advanced Approach: Render EJS template
with asynchronously fetched data
*/
app.get('/dynamic', async (req, res) => {
    const dynamicData = await fetchData();
    res.render('dynamic', { data: dynamicData });
});

// Function to simulate asynchronous data fetching
async function fetchData() {
    return new Promise(resolve => {
        setTimeout(() => {
            resolve('Dynamic Content');
        }, 1000);
    });
}

// Start the server
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
    console.log(`Server is running on port ${PORT}`);
});
```

Feature	Node.js	Express.js
Type	Runtime environment	Web application framework
Purpose	Running JavaScript on the server, handling core tasks	Simplifying web server development on top of Node.js
Routing	Not built-in	Built-in, easy routing with `app.get`, `app.post`, etc.
Middleware	No built-in middleware support	Middleware-based architecture for flexibility
Level of Abstraction	Low level (gives full control)	Higher level (simplifies common tasks)
Development Speed	Slower (more boilerplate required)	Faster (common functionality built-in)
Use Case	Low-level server development	Web applications and APIs development

Middleware:

What is Middleware?

Middleware refers to software that lies between the application and the server to communicate with each other. Basically, it intercepts requests and responses, and performs additional practices before passing them on to the final route handler. You can utilize it for various purposes, including authentication, logging, error handling, and data parsing.



Middleware is also accessible to change the requests and response objects, and can also terminate the response cycle, if necessary. This simplicity and flexibility make it a vital function of Express.js framework.

What is Express.js Middleware?

In Express.js, middleware is a function that has access to the request object (req), the response object (res), and the next middleware function in the application's request-response cycle. Middleware functions are used to execute code, modify the request and response objects, end the request-response cycle, or call the next function in the stack.

Since installing Express.js has confined functionalities of its own, it utilizes multiple middleware functions by using the next () function to simplify and improve web application operation. However, developers can also write their own Express Custom Middleware for their Express.js web application.

For a comprehensive understanding, let's examine a practical express middleware example:

A user visits your website and logs in with some credentials. Now, the request element has been sent to your web server. Meanwhile, you are required to verify the user also his/her login database, such as a particular <user> land up at the given <time> via the <IP> and <used device> before preparing or sending the response object to the browser.

Furthermore, you are also required to check your website's page that the user searched for the last time and modify the request object accordingly.

And how do you do this entire process? With one and only Middleware Function!

Different Types of Express.js Middleware

You can use the following types of middleware in an Express.js application:

1. Application-level middleware

It runs for all routes in an application. When a user receives an authentication request program, the middleware moves toward authentication code logic to approve the authentication. And use the next () function to call the rest of the route. If authentication fails, the middleware will show an error and the cycle will not respond to the next route.

2. Router-level middleware

This middleware function works the same way as the application level but can generate or limit an instance using Express.route() function or router.use() function. It runs for all routes in a router object only.

3. Built-in middleware

This function helps Express to act as a module. You can utilize the Express.json function to compute and add payloads and the Express. static function to act as application static assets.

4. Error-handling middleware

This middleware function helps to handle errors in the req-res cycle. It defines the middleware function in the same way with three arguments.

5. Third-party middleware

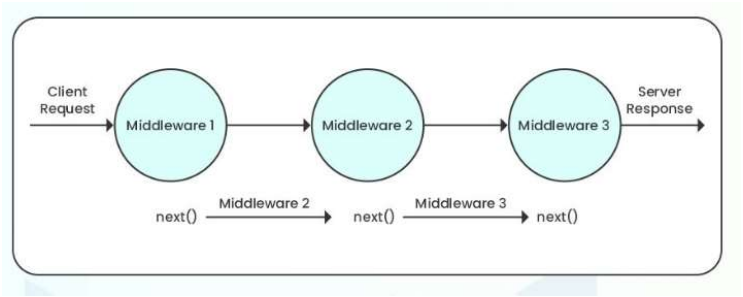
The user needs to install the node.js technology and load it to the application level at the router level to let the third-party middleware function work properly. ExpressJS supports lots of third-party middleware functions, such as

The bodyParser function is used to parse the requests with payloads and can be installed using npm install.

The cookieParser function parses the cookie header by cookie names and can be mounted by the npm install function.

How to Write and Create Express Middleware?

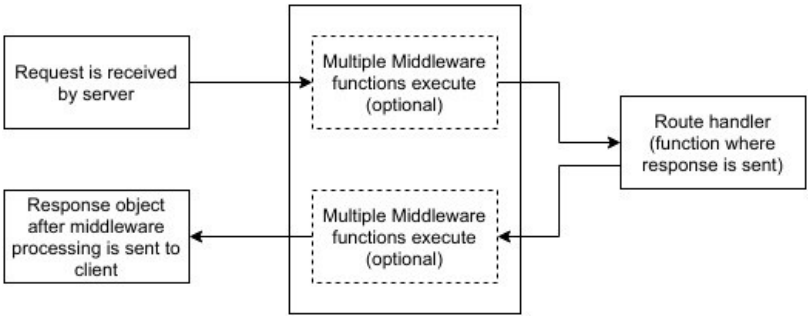
Middleware functions are written in chain format to execute a code order with the next () function that calls the next express middleware function if needed as shown in the below image:



The basic Express middleware syntax is

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

{}, (req, res) => {})



Steps to Implement Middleware in Express

Step 1: Initialize the node in the project using the following command.

```
npm init -y
```

Step 2: Install the required dependencies.

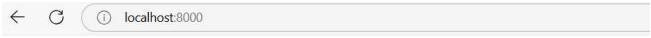
```
npm install express nodemon
```

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

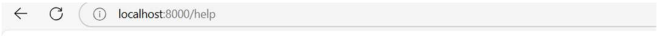
```
"start": "nodemon index.js",
```

Project Structure


```
index.js > ...
1   var express = require('express');
2   var app = express();
3
4   app.get('/', function(req, res) {
5     res.send('Welcome to CBIT!');
6   });
7   app.get('/help', function(req, res) {
8     res.send('How can I help You?');
9   });
10  var server = app.listen(8000, function () {
11    var host = server.address().address
12    var port = server.address().port
13  console.log("Example app listening at http://%s:%s", host, port)
14  })
```



Welcome to CBIT!



How can I help You?

```
PS C:\Users\Abdul Ahad\Documents\CBIT Academics\EAD\EAD Lab\express-middleware> node index.js
Example app listening at http://:::8000
```

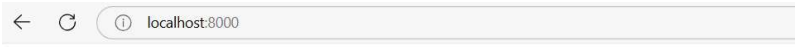
Note: You see that the command prompt is not changed. Means, it is not showing any record of the GET request although a GET request is processed

Use of Express.js Middleware

If you want to record every time you a get a request then you can use a middleware.

index_midlleware.js > app.get('/help') callback

```
var express = require('express');
var app = express();
app.use(function(req, res, next) {
  console.log('%s %s', req.method, req.url);
  next();
});
app.get('/', function(req, res, next) {
  res.send('Welcome to AI&DS!');
});
app.get('/help', function(req, res, next) {
  res.send('How can I help you in AI&DS Department?');
});
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)
})
```



Welcome to AI&DS!

How can I help you in AI&DS Department?

You can see that output is same but terminal is displaying a GET result. As many times as you reload the page, the terminal will be updated.

Note: In the above example next() middleware is used.

Middleware example explanation

- In the above middleware example a new function is used to invoke with every request via **app.use()**.
- Middleware is a function, just like route handlers and invoked also in the similar manner.
- You can add more middlewares above or below using the same API.

Routing:

Routing in Express.js is the mechanism for handling HTTP requests (GET, POST, DELETE, etc.) at different URL paths (or endpoints). Express provides a flexible and powerful routing system, making it easy to define and organize routes

Each route can have one or more handler functions, which are executed when the route is matched.

Route definition takes the following structure:

```
app.METHOD(PATH, HANDLER)
```

Where:

- app is an instance of express.
- METHOD is an HTTP request method, in lowercase.
- PATH is a path on the server.
- HANDLER is the function executed when the route is matched.
- The following examples illustrate defining simple routes.
- Respond with Hello World! on the homepage:

```
app.get('/', (req, res) => {
  res.send('Hello World!')
})
```

- Respond to POST request on the root route (/), the application's home page:

```
app.post('/', (req, res) => {
  res.send('Got a POST request')
})
```

- Respond to a PUT request to the /user route:

```
app.put('/user', (req, res) => {
  res.send('Got a PUT request at /user')
})
```

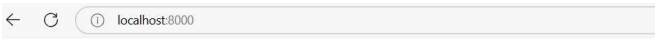
- Respond to a DELETE request to the /user route:

```
app.delete('/user', (req, res) => {
  res.send('Got a DELETE request at /user')
```

- })

Example:

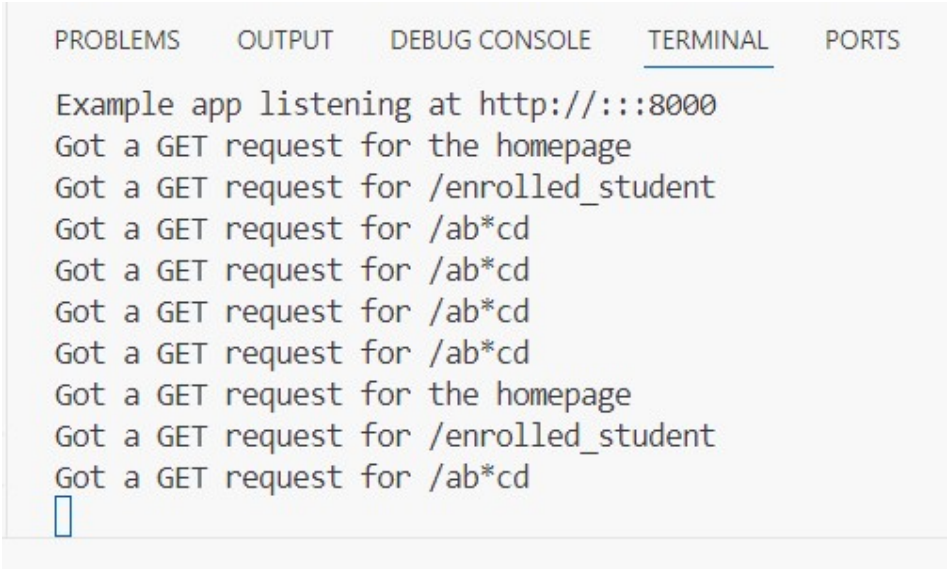
```
var express = require('express');
var app = express();
app.get('/', function (req, res) {
   console.log("Got a GET request for the homepage");
   res.send('Welcome to Website!');
})
app.post('/', function (req, res) {
   console.log("Got a POST request for the homepage");
   res.send('I am Impossible!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



Welcome to Website!

I am an enrolled student.

Pattern Matched.



Static Files:

In Express.js, static files (such as images, CSS files, and JavaScript files) are served using the built-in middleware express.static. This middleware allows you to specify one or more directories from which to serve static assets.

```
my-app/
├── public/
│   ├── images/
│   ├── css/
│   └── js/
└── app.js
```

Use `express.static` Middleware:

- In your `app.js` file, use the `express.static` middleware to serve files from the `public` directory.

  Example:

const express = require('express');

const app = express();

// Use express.static to serve static files from the 'public' directory

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

app.get('/', (req, res) => {
  res.send('Welcome to the homepage!');
});


const PORT = 3000;

app.listen(PORT, () => {
  console.log(`Server is running on http://localhost:${PORT}`);
});
```

Accessing Static Files:

- Once set up, any file in the `public` directory can be accessed directly by specifying its path relative to `public`.
- For example:
  - An image file located at `public/images/logo.png` would be accessible at `http://localhost:3000/images/logo.png`.
  - A CSS file at `public/css/styles.css` would be accessible at `http://localhost:3000/css/styles.css`.

Custom Mount Path (Optional):

- You can specify a custom URL path by providing an additional argument to `express.static`.
- For example, if you want to serve files from `public` under the path `/static`, you can do the following:

```
app.use('/static', express.static('public'));
```

Now, `public/images/logo.png` would be accessible at `http://localhost:3000/static/images/logo.png`.

Example of Serving HTML with Static Assets

In public, you might have an HTML file that links to CSS and JS files:

```
<!-- public/index.html -->

<!DOCTYPE html>

<html lang="en">

<head>
```

```
    <meta charset="UTF-8">
    <title>My App</title>
    <link rel="stylesheet" href="/css/styles.css">
</head>
<body>
    <h1>Hello, world!</h1>
    <img src="/images/logo.png" alt="Logo">
    <script src="/js/app.js"></script>
</body>
</html>
```