Syllabus:
Getting Started with Node: Introduction to NodeJS, NPM, and Node Module System – Path, OS, FS, and http modules.
Introduction to ExpressJS: Lifecycle and routing of Express App, first web server, reading configuration parameters, Handling request and response parameters. **----------Page no:33**

Getting Started with Node:

Introduction to Node.js

Node.js is an open-source, cross-platform, JavaScript runtime environment that allows you to run JavaScript outside the browser.

- It uses the **V8 JavaScript Engine** (from Chrome).
- Designed for **server-side programming**.
- Non-blocking, event-driven architecture → handles many connections efficiently.

Example: **File name**: **Hello.js**

```
console.log("Hello, Node.js!");
```

To run:
```
node Hello.js
```
Output:
```
Hello, Node.js!
```

Features of Node.js

Feature	Description
Asynchronous	Node.js uses non-blocking I/O. It doesn't wait for one operation to complete before starting another.
Event-driven	Everything runs on events, using the Event Loop.
Single-threaded	Node uses a single thread but handles multiple requests asynchronously.
Fast execution	Uses Google's V8 engine for fast performance.
Cross-platform	Runs on Windows, macOS, Linux.

Web Applications: A **web application** is software that runs on a server and is rendered by a client browser that accesses all of the application's resources through the Internet.
A typical web application consists of the following components:

- **Client:** A client refers to the user interacting with the server by sending out requests.
- **Server:** The server is responsible for receiving client requests, performing appropriate tasks, and returning results to the clients. It bridges the **front end** and the stored data, allowing clients to perform operations on the data.
- **Database:** A database is where a web application's data is stored. The data can be created, modified, and deleted depending on the client's request.

Node.js Architecture:

- Node.js offers a **"Single-Threaded Event Loop"** architecture to manage concurrent requests without creating multiple threads and using fewer threads to utilize fewer resources.
- That's why developers prefer Node.js architecture to take the advantages Node.js has.
- Another reason for the popularity of Node.js is **its callback mechanism** and **JavaScript-event-based Model**.
- Node.js event loop architecture enables **Node.js to execute blocking I/O operations in a non-blocking way.**
- Also, you can easily scale your Node.js application with its single thread than multi-thread per request under ordinary web loads.
- Actually, Node.js has two types of threading:
    i. **Multi-Threading**
    ii. **Single Threading**

i. Multi-threading is a program execution model that creates multiple threads within a process.

- ✓ These threads execute independently but concurrently share process resources.
- ✓ During this multiple-threading process, a thread is chosen each time a request is made until all the allotted threads are exhausted.
- ✓ While a thread is busy, the server can't proceed; it must wait until a free thread is available.
- ✓ As a result, poor and slow applications can negatively impact customer experience.

➢ However, Node.js uses a **single-threaded processing model**.

ii. Single-Threading:

- ✓ Single threaded architecture handles Input/Output operations by using event loops to handle blocking operations in a non-blocking way, whereas multi-thread architecture processes every request as a single thread.
- ✓ This is what the creator of Node.js, Ryan Dahl, had in mind and showcased to the dev community when he introduced Node.js. And it's one of the main reasons why developers choose Node.js for app development.
- ✓ This was the basics of Node.js project architecture. Now let's go further to understand the Node.js architecture pattern and its components.

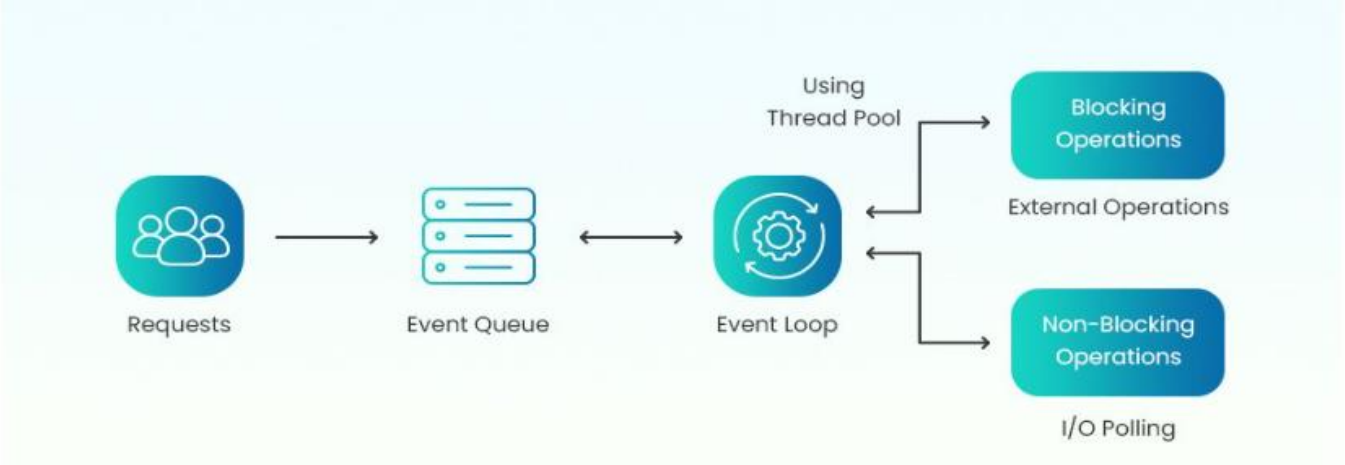


Figure: Node.js Architecture

Components of the Node.js Server Architecture

- Generally, server-side technologies like **ASP.NET**, **PHP**, **Java**, and **Ruby** use multi-threaded models. For each client request, traditional architectures create threads.
- With Node.js Single Threaded Event Loop Model Architecture, Node.js prevents creating multiple threads for each request.
- As a result, all client requests are processed through a single thread.

Moreover, this single-threaded architecture is also referred to as the event-driven architecture of Node.js. Hence, Node.js is capable of managing several clients at once.

Node.js adheres to two fundamentals of its architecture:

- **Non-blocking I/O operations**
- **Asynchronous paradigm**

However, **these two concepts are comparable to JavaScript's event-based model**.

Now is the time to understand the components of Node.js application architecture and how it functions.

Node.js architecture is made up of six elements, which are:

1. **Requests:** The incoming requests can be blocking (complex) or non-blocking (simple), depending upon the specific tasks users want to perform in a web application.
2. **Node.js Server:** Node.js server is the foundation of the architecture. As a server-side platform, the Node.js server not only accepts requests from users but also processes these requests and sends those responses to corresponding users.
3. **Event Queue:** Event Queue in the Node.js server stores the incoming client requests and passes them one-by-one into Event Loop.
4. **Event Loop:** This is an infinite loop – that never ends. It continues to receive requests from the event queue, process them, and return the corresponding response to the clients.
   This **event loop has six phases** that are repeated until no code is left to execute. The six phases of the event loop are:
   1) Timers
   2) I/O Callbacks
   3) Waiting / Preparation
   4) I/O Polling
   5) setImmediate() callbacks
   6) Close events

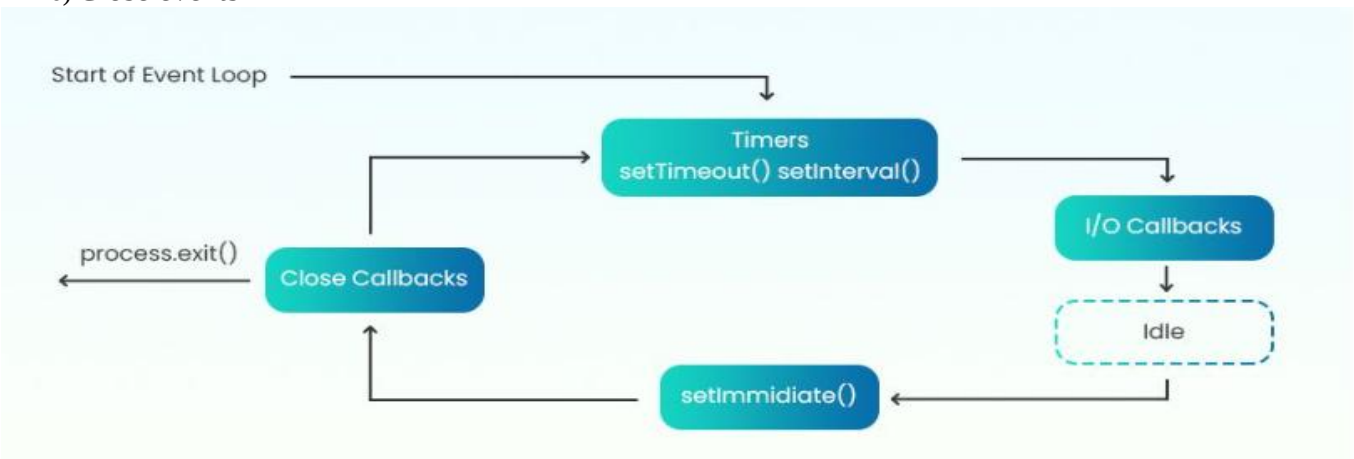


Figure: Event Loop Architecture

5) Thread Pool: The Thread Pool in Node.js backend architecture contains the threads for carrying out tasks required to process client requests.
6) External Resources: These External Resources are used for blocking client requests. They generally handle multiple blocking requests, like data storage, computation, etc.

The above-mentioned components of Node.js modular architecture play the most important role in **Node.js web development**. Let's understand how these components function and what flow Node.js architecture follows.

Workflow of Node.js Server Architecture

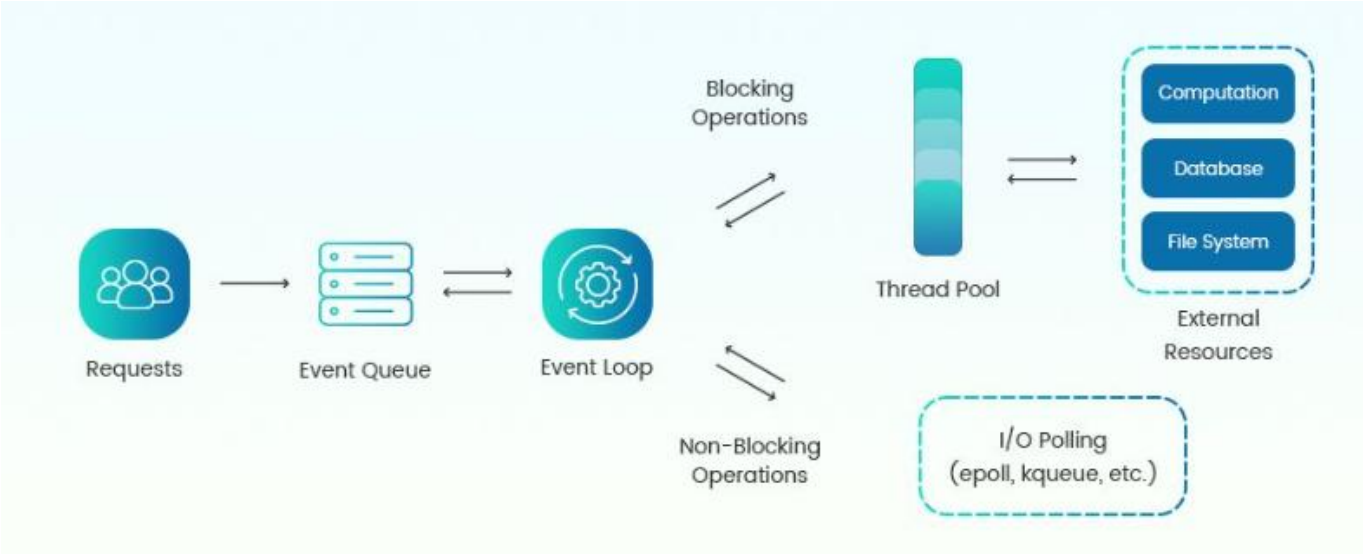


Figure: Work flow Architecture.

As we can see in the Node.js architecture diagram, incoming and outgoing requests fall under two categories.

Here, the incoming requests can either be **basic**, which equates to **non-blocking**, or **complex**, which equates to **blocking**. It actually depends on the tasks that web applications or software users are expected to perform.

Types of Operations

- **Queries** are used to find specific data within databases.
- **Delete queries** and requests to delete specific pieces of data are examples of eliminating data.
- When you want to **update data**, you either send a request to edit or update a specific piece of data or execute an update query on a particular table row to modify a database record.
- **Node.js** collects client requests as they arrive and adds them to its **Event Queue**.
- The **Event Loop** then receives each incoming request and decides whether it requires external resources.
  - If it does, the request is assigned to external resources.
  - If not, the process moves to the next stage.
- The **Event Loop** handles non-blocking (simple) requests using **I/O polling** and returns responses directly to clients.

Handling Complex Requests

- After the above phases, each **complex request** is granted a single thread from the **thread pool**. This thread allocates external resources — such as **file systems, databases**, etc. — to handle specific blocking requests.
- Once the task is finished, the response is sent back to the **Event Loop**, which then transmits it to the client. This approach makes Node.js a **single-threaded event loop model**.

Single threaded Event Loop Architecture in Node.js

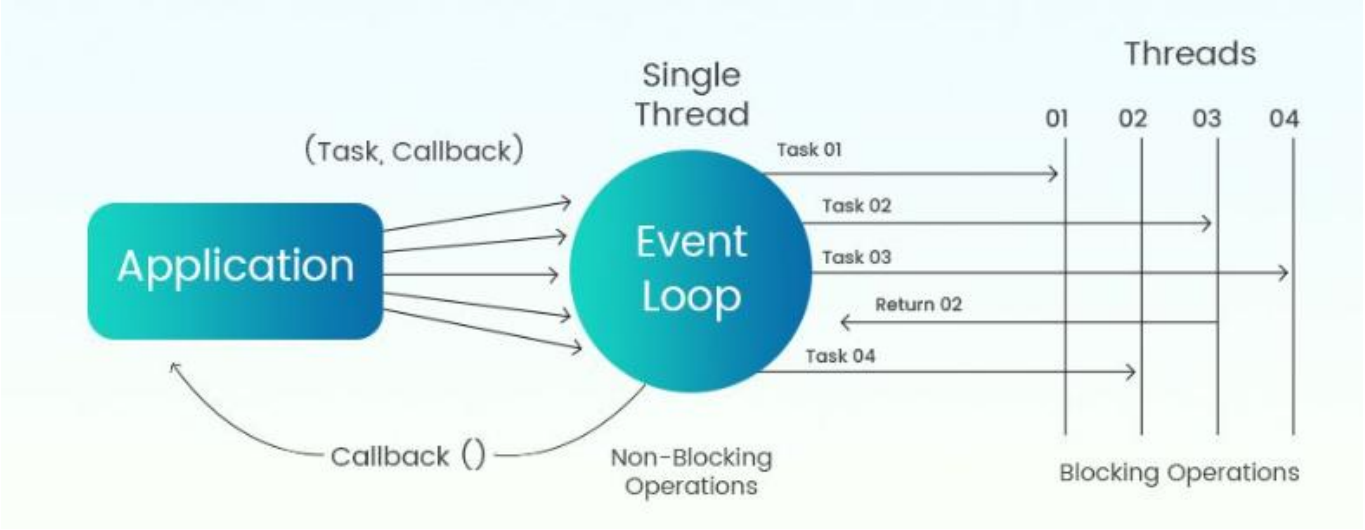


Figure: Single threaded Event Loop Architecture.

Advantages of Node.js Server Architecture

- The Node.js server can efficiently handle a high number of requests by employing the use of Event Queue and Thread Pool.
- There is no need to establish multiple threads because Event Loop processes all requests one at a time, therefore a single thread is sufficient.
- The entire process of serving requests to a Node.js server consumes less memory and server resources since the requests are handled one at a time.

Disadvantages of Node.js Server Architecture:

here are the disadvantages of Node.js server architecture in a more concise format:

1. **Single-Threaded**: Limited to one thread; can be a bottleneck for CPU-intensive tasks.
2. **Callback Hell**: Complex nesting of callbacks can lead to hard-to-maintain code.
3. **Performance Bottlenecks**: Not optimal for heavy computational tasks due to non-blocking I/O model.
4. **Dependency on Outside Libraries**: Heavy reliance on third-party libraries can impact stability and security.
5. **Inconsistent API**: Frequent API changes can lead to backward compatibility issues.
6. **Lack of Strong Typing**: JavaScript's lack of strong typing can lead to runtime errors and bugs.

Installing Node.js: Node.js can be installed in several ways depending on your OS. The easiest method is to use the **official Node.js installer** from the Node.js website.

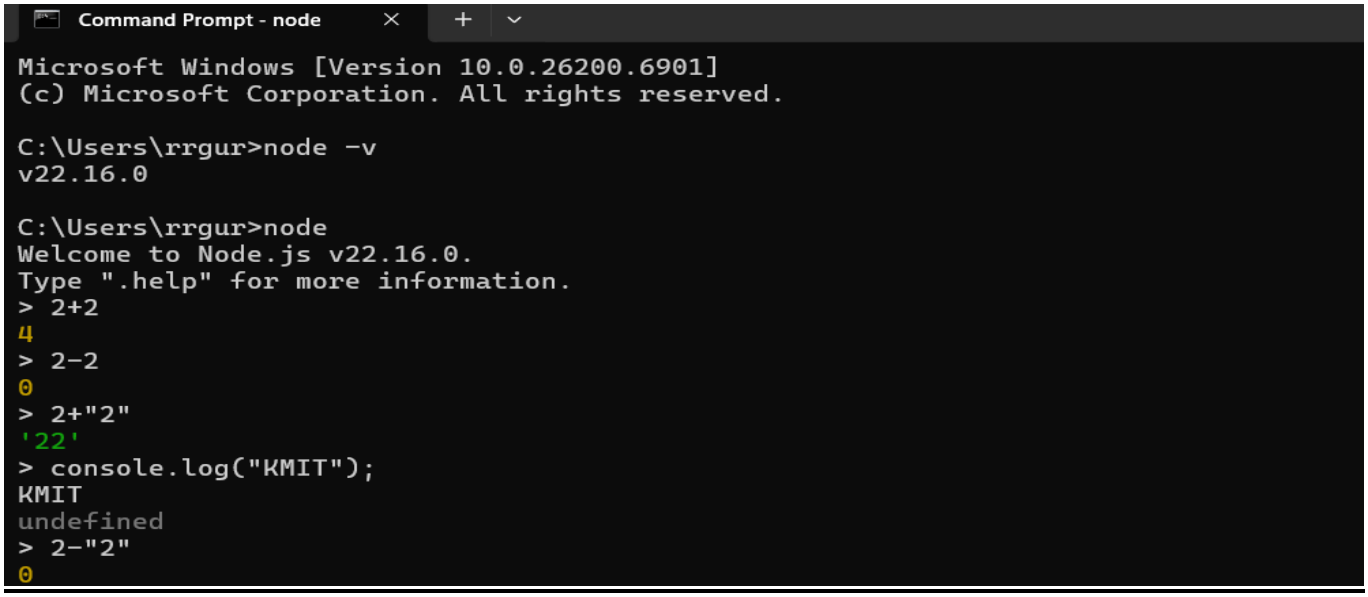
For Windows

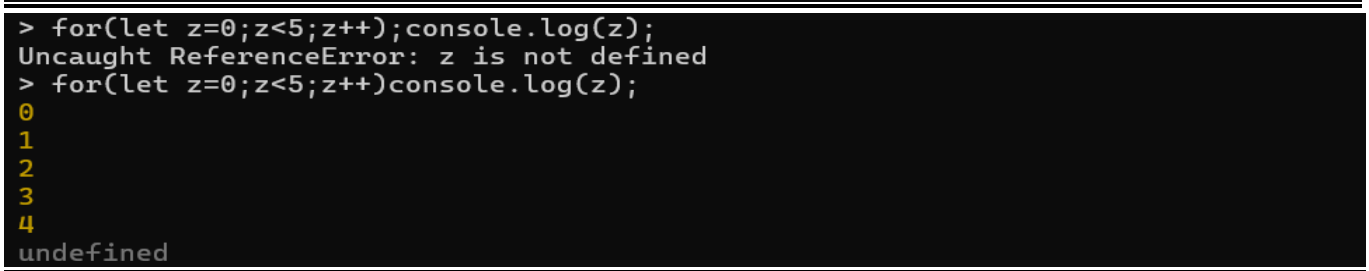
1. **Go to the official Node.js website:**
   🔗 https://nodejs.org
2. **Choose a version:**
   - **LTS (Long-Term Support):** Stable, recommended for most users.

- **Current:** Has the latest features but may be less stable.

3. **Download the Windows Installer (.msi)**
   - Choose 64-bit if your system supports it.
4. **Run the installer:**
   - Follow the setup wizard.
   - Keep the default options (it includes npm, the Node Package Manager).
5. **Verify installation:**
   Open **Command Prompt (cmd)** or **PowerShell**, then type:
6. node -v
   npm -v

This displays the installed **Node.js** and **npm** versions.





NPM (Node Package Manager):

The **NPM (Node Package Manager)** is the default package manager for Node.js that helps developers to install, manage, and share JavaScript modules easily.

NPM allows open-source web developers to share and borrow packages for app development. It also works as a command-line utility for installing packages in a project, managing dependencies, and handling version management.

Components of NPM

1. Website:
You can find packages from the official NPM website for your project.
Also, you can create and set up profiles to manage and access all types of packages.

2. Command Line Interface (CLI):
To interface with NPM packages and repositories, the CLI runs from your computer's terminal.

3. Registry:
It has a huge database of JavaScript projects and metadata.
This allows you to use any supported NPM registry. You can also utilize someone else's registry as per their terms of use.

Install / Verify

- NPM is usually installed with Node.js.
- Check versions:
  Command: `node -v`
  Command: `npm -v`

Initialize a project (package.json)

- Create `package.json` (interactive): **command:** `npm init`
- Use defaults: **command** : `npm init -y`
- Important **`package.json`** fields:
  - `name`, `version`, `description`
  - `main` (entry file)
  - `scripts` (runable commands, e.g. `start`, `test`)
  - `dependencies` (runtime)
  - `devDependencies` (development)
  - `peerDependencies`, `optionalDependencies`
  - `engines` (node engine version constraints)
  - `private: true` (prevents accidental `npm publish`)

package.json file:

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

```
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": ""
}
```

Install packages:

- **Install locally (default):** `npm install <package>` **(or)** `npm i <package>`
- **Install a specific version:** `npm i express@4.17.1`
- **Install as devDependency:** `npm i --save-dev <package>` **(or)** `npm i -D <package>`

It is used to **install a package as a development dependency** in your Node.js project.

`--save`

- This flag tells NPM to **add the installed package to your `package.json` file** under the `"dependencies"` section.
- It means the package is **required for your app to run in production**.

Example: npm install express –save This will update your `package.json` as:

```
"dependencies": {
  "express": "^5.1.0"
}
```

package.json file:

```
{
  "name": "bookstore_api",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": "",
  "dependencies": {
    "express": "^5.1.0"
  }
}
```

`--save-dev` : `dev` stands for **development**.

- This flag tells NPM to save the package in the `"devDependencies"` section of your `package.json`.
- It's used for packages that are **only needed during development**, not in production.

Example: command: npm i --save-dev nodemon

It means: "Install the nodemon package and record it under devDependencies in my package.json, because I only need it while developing — not when deploying my app."

package.json file:

```
{
  "name": "bookstore_api",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": "",
  "dependencies": {
    "express": "^5.1.0"
  },
  "devDependencies": {
    "nodemon": "^3.1.10"
  }
}
```

What is Nodemon?

- ✓ **Nodemon** (short for *Node Monitor*) is a **development tool** that automatically **restarts your Node.js server** whenever you make changes to your code.
- ✓ It's not part of Node.js itself — it's an **external package** that helps during **development**.

Why We Use Nodemon?

When you build a Node.js app (for example, using `app.js file name`), you normally start it with:

```
node app.js
```

- ✓ But every time you make a small change — like fixing a bug or updating a route — you have to **stop** the server (`Ctrl + C`) and **restart it manually**.
- ✓ That's slow and repetitive , So instead, we use **Nodemon**: **`nodemon app.js`**
- ✓ Now Nodemon watches your files. Whenever you **save** a file, it **automatically restarts** your app.

How to Install Nodemon:

- ✓ Since it's only needed during development, install it as a **dev dependency**:

Command: npm install --save-dev nodemon or npm i -D nodemon

How to Use It:

Option 1: Run Directly: `npx nodemon app.js`

Option 2: Add to package.json Scripts

In your `package.json`, add:

```
"scripts": {
  "start": "node app.js",
  "dev": "nodemon app.js"
}
```

Now you can run: `npm run dev`

- **Install globally (system-wide):** `npm i -g <package>`
- **Install from `package.json` (after cloning project):** `npm install`

Save locations & types

- dependencies: needed to run your app in production.
- devDependencies: needed only for development (tests, build tools).
- peerDependencies: packages expected to be provided by the host (used for libraries/plugins).
- optionalDependencies: installs but won't fail the whole install if it fails.

package-lock.json & npm-shrinkwrap.json

- `package-lock.json` (**auto-generated**) locks exact versions to reproduce installs. **Commit it**.
- `npm-shrinkwrap.json` similar but intended for published packages (rarely used).

npm scripts:

- Define scripts in `package.json` under `"scripts"`.

```
"scripts": {
  "start": "node app.js",
  "dev": "nodemon app.js",
  "test": "mocha",
  "build": "webpack --mode production"
}
```

Run:

```
npm run dev    # for custom scripts
npm start      # shortcut for `start`
npm test       # shortcut for `test`
```

Useful commands

- **Uninstall:**

```
npm uninstall <package>
npm uninstall -D <package>    # remove devDependency
```

- **Update:** `npm update <package>`
- **View package info:**

```
npm view <package>            # metadata
npm view <package> versions   # available versions
```

- **List installed packages:**

```
npm list         # local
npm list -g      # global
```

- **Search registry:** `npm search <term>`
- **Audit for vulnerabilities:**

```
npm audit
npm audit fix    # automatic fixes when available
```

- **Clear cache:**

```
npm cache verify
npm cache clean --force
```

npx (execute binaries without global install):

`npx` runs package binaries from either local `node_modules/.bin` or temporarily fetches them.

```
npx create-react-app my-app
npx eslint . –fix
```

Useful to avoid global installs for one-off tools.

Node Module System – Path, OS, FS, and http modules.

1. module.exports:

- ✓ In Node.js, every **JavaScript file** is treated as a **separate module**.
- ✓ Each module has a special object called **`module.exports`**.
  Whatever you assign to `module.exports` is what other files will get when they use `require()` to import your file.

Purpose of module.exports:

- ✓ The purpose is to **share code between files** — functions, variables, classes, or entire objects.
- ✓ It helps keep your project **modular**, **organized**, and **reusable**.

"Think of `module.exports` as the *stuff you want to make public* from one file, so another file can use it."

How It Works

1. Each Node.js file is wrapped in its own "module scope".
   So variables/functions in one file are **not automatically visible** to others.
2. The `module.exports` object is what **gets returned** when you call `require()`.
3. You can export **a single value** or **multiple values** using `module.exports`.

Example Program: (for run all Programs install npm, **command: npm init –y)**

first_module.js

```
function add(a,b)
{
    return a+b;
}
function sub(a,b)
{
    return a-b;
}
function divide(a,b)
{
    if(b==0)
    {
        throw new Error("divide by zero is not allowed")
    }
    return a/b;
}
module.exports={add,sub,divide};
```

index.js:

```
const firstmodule= require("./first_module");
/*./ means: "current directory" – the folder where your current JavaScript file is
located.
*/
console.log(firstmodule.add(10,20))

try
{
```

```
            console.log("trying divide by zero")
            let result=firstmodule.divide(10,0);
            console.log(result)
        }
        catch(error)
        {
            console.log("caught an error", error.message);
        }
        try
        {
            let result=firstmodule.divide(0,10);
            console.log(result)
        }
        catch(error){
            console.log("caught an error", error.message);
        }
```

Run: node index.js

Output: 30

```
        trying divide by zero
        caught an error divide by zero is not allowed
        0
```

Explanation:

```
Internal Working:
-----------------
When Node.js loads a file, it wraps your code like this:

(function (exports, require, module, __filename, __dirname) {
    // your code here
});

That's why every file automatically has:
----------------------------------------
exports
module
require
__filename
__dirname
Here, module.exports starts as an empty object {}.
```

1.1 import and export In ES6 (ECMAScript 2015) :

ES6 introduced a **native module system** for JavaScript — a more modern and readable way to **share code** between files.

Instead of: module.exports = ...
`const x = require('...');`

We now use: export ...
import ...

Purpose: Just like module.exports, ES6 export and import let you:

- **Split code** into multiple files (modules)
- **Reuse** functions, classes, and constants
- **Organize** code better in large applications

Types of Exports: There are **two main types**:

i. **Named Exports** — you can export multiple things.
ii. **Default Export** — you export one main thing per file.

i. Named Export

math.js

```
// Named exports
export function add(a, b) {
  return a + b;
}
export function subtract(a, b) {
  return a - b;
}
export const pi = 3.14159;
```

app.js

```
// Import named exports (must use same names or rename them)
import { add, subtract, pi } from './math.js';

console.log(add(10, 5));       // 15
console.log(subtract(10, 5));  // 5
console.log(pi);               // 3.14159
```

You can also rename while importing:

```
import { add as sum } from './math.js';
console.log(sum(2, 3)); // 5
```

ii. Default Export

user.js

```
export default class User {
  constructor(name) {
    this.name = name;
  }
  greet() {
    console.log(`Hello, ${this.name}!`);
  }
}
```

app.js

```
// Import default export — name can be anything
import User from './user.js';

const user1 = new User('John');
user1.greet(); // Hello, John!
```

Note: Only **one default export** is allowed per file.

Example: Combine Both:

utils.js

```
export const appName = 'My App';

export function greet(name) {
  return `Welcome, ${name}!`;
}
export default function sayBye(name) {
  return `Goodbye, ${name}!`;
}
```

main.js

```
import sayBye, { appName, greet } from './utils.js';

console.log(appName);        // My App
console.log(greet('John')); // Welcome, Rakesh!
console.log(sayBye('John')); // Goodbye, Rakesh!
```

2. Special Variables in Node.js

i. __filename:

- __filename is a **global variable** in Node.js.
- It stores the **absolute path of the current file**, including the file name.
- Useful to know **exactly which file is running**.

Example: console.log("filename is :", __filename);

Output: filename is : C:\Users\Rakesh\Documents\nodejs\example1.js

Use Cases:

- Logging the current file path for debugging.
- Creating paths **relative to the current file**.
- Combining with path.dirname() to get directory paths.

ii. __dirname

- __dirname is a **global variable** in Node.js.
- It stores the **absolute path of the directory** that contains the current file.
- **Does not include the file name.**

Example: console.log("Directory name is :", __dirname);

Output: Directory name is :C:\Users\Rakesh\Documents\nodejs

Note: Node.js automatically provides these two special global variables inside every module (file):

These are very helpful when you need to:

Load files relative to your current script
Build correct file paths
Avoid hardcoding full paths

3. Path Module:

- ✓ The path module in Node.js is one of the most important built-in (core) modules for handling and working with file and directory paths in a clean, platform-independent way.
- ✓ The path module helps you work with file and folder paths easily in Node.js. It is a core module, so you don't need to install it — just require('path').
- ✓ It makes your code cross-platform (works on both Windows \ and Linux / paths).

To use it, you must **import** it like this:

```
const path=require("path");          (or)      const path = require('node:path');
```

Commonly Used Path Methods:

Method	Description	Example
`path.basename()`	Returns the last portion (file name) of a path	`path.basename('/folder/file.txt')` → `'file.txt'`
`path.dirname()`	Returns the directory name of a path	`path.dirname('/folder/file.txt')` → `'/folder'`
`path.extname()`	Returns the file extension	`path.extname('index.html')` → `'.html'`
`path.join()`	Joins multiple path segments into one normalized path	`path.join('folder', 'sub', 'file.txt')` → `'folder/sub/file.txt'`
`path.resolve()`	Resolves paths into an absolute path	`path.resolve('folder', 'file.txt')` → `C:\Users\...`
`path.parse()`	Returns an object with details like root, dir, base, ext, name	
`path.format()`	Opposite of `parse()` → builds a path from an object	
`path.isAbsolute()`	Checks if the path is absolute	
`path.normalize()`	Normalizes a path (fixes extra slashes, etc.)	

Index.js

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

//get directory
console.log("Directory name is :", path.dirname(__filename));

//get file names
console.log("file name is:", path.basename(__filename));

// get file extensions
console.log("file extensions",path.extname(__filename));

//join path:It automatically adds or removes slashes (/ or \) as needed.
```

```
const joinpath=path.join("/users","documents","node","projects");

console.log("joine path is:",joinpath);

//resolve path:Resolves a sequence of paths into an absolute path.
const resolvepath=path.resolve("user","documents","node","projects");

console.log("Resolve path is:",resolvepath);
/*
Differences:
join() → joins relative paths.
resolve() → always gives absolute path starting from root.
*/
//Normalizing paths:Cleans up a messy path (removes .., ., and duplicate slashes).
const normalizepath=path.normalize("/user/.documents/../node/projects");
console.log("normalize path:",normalizepath);

//parse(path):Returns an object with useful details about the file path.
const info = path.parse('/user/local/test/data.txt');
console.log(“path details”,info);
//we can also rebuild a path using:
const rebuilt = path.format(info);
console.log(“Rebuilt path is :”,rebuilt); // /user/local/test/data.txt

//isAbsolute(path): Checks if the given path is absolute.
console.log(path.isAbsolute('/user/local')); // true
console.log(path.isAbsolute('data/file.txt')); // false
```

Run: **node index.js**

Output:

```
Directory name is : D:\WT 2025-26 III YR I SEM\DAY WISE TOPICS\NodeJS\nodeJS_Porgrams\4.path-module
file name is: index.js
file extensions .js
joine path is: \users\documents\node\projects
Resolve path is: D:\WT 2025-26 III YR I SEM\DAY WISE TOPICS\NodeJS\nodeJS_Porgrams\4.path-module\user\documents\node\projects
normalize path: \user\node\projects
path details: {
  root: '/',
  dir: '/user/local/test',
  base: 'data.txt',
  ext: '.txt',
  name: 'data'
}
Rebuilt path is : /user/local/test\data.txt
true
false
```

4. OS Module:

- ✓ The **os** module in Node.js provides **operating system-related utility methods and properties**.
- ✓ It allows you to get information about the system your Node.js app is running on, like CPU, memory, network, and platform details.
- ✓ It is a **built-in module**, so you don't need to install anything.

You can import it using: **const os = require('os');**

Commonly Used Methods of OS Module:

Method	Description	Example Output
`os.platform()`	Returns OS platform	`'win32', 'linux', 'darwin'`
`os.arch()`	Returns CPU architecture	`'x64', 'arm'`
`os.cpus()`	Returns array of CPU info objects	Array of CPU cores with model & speed
`os.freemem()`	Returns free memory in bytes	`2100000000`
`os.totalmem()`	Returns total system memory in bytes	`8192000000`
`os.homedir()`	Returns user home directory	`'C:\Users\Rakesh'`
`os.tmpdir()`	Returns OS temp directory	`'C:\Users\Rakesh\AppData\Local\Temp'`
`os.hostname()`	Returns system hostname	`'DESKTOP-123ABC'`
`os.networkInterfaces()`	Returns network interfaces	Object with IP addresses

Index.js

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

//1.Getting OS Platform:  Method: os.platform(): Returns the platform name (win32, linux, darwin, etc.).

console.log("Operating System Platform:", os.platform());

//2. Getting OS Type: Method: os.type(): Returns the name of the operating system.

console.log("Operating System Type:", os.type());

//3. Getting OS Release: Method: os.release():Returns the version of the operating system.

console.log("OS Release:", os.release());

//4. Getting System Architecture: Method: os.arch(): Returns the CPU architecture (x64, arm, etc.)

console.log("System Architecture:", os.arch());

/*5. Getting Total and Free Memory: Methods: os.totalmem() → total system memory in bytes

os.freemem() → free memory in bytes*/

console.log("Total Memory:", os.totalmem() / (1024 * 1024 * 1024), "GB");
console.log("Free Memory:", os.freemem() / (1024 * 1024 * 1024), "GB");
```

```
/*6. Getting Home Directory: Method: os.homedir(): Returns the path of the current user's home directory.
*/
console.log("Home Directory:", os.homedir()); //

/*7. Getting System Uptime: Method: os.uptime(): Returns the system uptime in seconds (how long the system has been running).*/

console.log("System Uptime:", os.uptime(), "seconds");// convert to hours: os.uptime() / 3600

//8. Getting Current User Info: Method: os.userInfo(): Returns details about the current logged-in user.
console.log("User Info:", os.userInfo());

//9. Getting CPU Information: Method: os.cpus():Returns details about each logical CPU core.
const cpus = os.cpus();
console.log("CPU Information:");
console.log(cpus);
console.log("Number of CPU cores:", cpus.length);

/* 10. Getting Network Interfaces: Method: os.networkInterfaces():Returns details of network adapters (IP addresses, MAC address, etc.)*/

console.log("Network Interfaces:", os.networkInterfaces());

//11. Getting Temporary Directory: Method: os.tmpdir(): Returns the default temp folder path.
console.log("Temporary Directory:", os.tmpdir());

/*12. Getting Endianness: Method: os.endianness(): Returns 'BE' (Big Endian) or 'LE' (Little Endian).
It shows how bytes are ordered in memory.*/

console.log("System Endianness:", os.endianness());

//13. Getting Hostname: Method: os.hostname(): Returns the computer's hostname.
console.log("Host Name:", os.hostname());

/*14. Checking Load Average (Linux/macOS only) Method: os.loadavg()
Returns CPU load average for the last 1, 5, and 15 minutes.
(Works on Linux/macOS, not on Windows.) */
console.log("Load Average:", os.loadavg());
```

Output:

```
Operating System Platform: win32
Operating System Type: Windows_NT
OS Release: 10.0.26200
System Architecture: x64
Total Memory: 7.784477233886719 GB
Free Memory: 1.9734420776367188 GB
Home Directory: C:\Users\rrgur
System Uptime: 261338.718 seconds
User Info: [Object: null prototype] {
```

```
  uid: -1,
  gid: -1,
  username: 'rrgur',
  homedir: 'C:\\Users\\rrgur',
  shell: null
}
CPU Information:
[
  {
    model: '11th Gen Intel(R) Core(TM) i3-1115G4 @ 3.00GHz',
    speed: 2995,
    times: {
      user: 8387734,
      nice: 0,
      sys: 7076968,
      idle: 96357328,
      irq: 818640
    }
  },
  {
    model: '11th Gen Intel(R) Core(TM) i3-1115G4 @ 3.00GHz',
    speed: 2995,
    times: {
      user: 7545140,
      nice: 0,
      sys: 5465437,
      idle: 98811421,
      irq: 314218
    }
  },
  {
    model: '11th Gen Intel(R) Core(TM) i3-1115G4 @ 3.00GHz',
    speed: 2995,
    times: {
      user: 7859390,
      nice: 0,
      sys: 5464015,
      idle: 98498593,
      irq: 309062
    }
  },
  {
    model: '11th Gen Intel(R) Core(TM) i3-1115G4 @ 3.00GHz',
    speed: 2995,
    times: {
      user: 6724578,
      nice: 0,
      sys: 4821968,
      idle: 100275453,
      irq: 259234
    }
```

```
  }
]
Number of CPU cores: 4
Network Interfaces: {
  'Wi-Fi': [
    {
      address: '2405:201:c005:806d:2b1a:406a:9a1f:efdb',
      netmask: 'ffff:ffff:ffff:ffff::',
      family: 'IPv6',
      mac: 'f8:9e:94:ae:bf:d2',
      internal: false,
      cidr: '2405:201:c005:806d:2b1a:406a:9a1f:efdb/64',
      scopeid: 0
    },
    {
      address: '2405:201:c005:806d:15be:1377:fb:184a',
      netmask: 'ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff',
      family: 'IPv6',
      mac: 'f8:9e:94:ae:bf:d2',
      internal: false,
      cidr: '2405:201:c005:806d:15be:1377:fb:184a/128',
      scopeid: 0
    },
    {
      address: 'fe80::563d:bf01:5e5a:25ee',
      netmask: 'ffff:ffff:ffff:ffff::',
      family: 'IPv6',
      mac: 'f8:9e:94:ae:bf:d2',
      internal: false,
      cidr: 'fe80::563d:bf01:5e5a:25ee/64',
      scopeid: 17
    },
    {
      address: '192.168.29.5',
      netmask: '255.255.255.0',
      family: 'IPv4',
      mac: 'f8:9e:94:ae:bf:d2',
      internal: false,
      cidr: '192.168.29.5/24'
    }
  ],
  'Loopback Pseudo-Interface 1': [
    {
      address: '::1',
      netmask: 'ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff',
      family: 'IPv6',
      mac: '00:00:00:00:00:00',
      internal: true,
      cidr: '::1/128',
      scopeid: 0
    },
```

```
    {
      address: '127.0.0.1',
      netmask: '255.0.0.0',
      family: 'IPv4',
      mac: '00:00:00:00:00:00',
      internal: true,
      cidr: '127.0.0.1/8'
    }
  ]
}
```

Temporary Directory: C:\Users\rrgur\AppData\Local\Temp
System Endianness: LE
Host Name: rrgurrala
Load Average: [0, 0, 0]

5. FS(File System):

- ✓ **The fs (File System) module in Node.js** provides an **API for interacting with the file system**.
- ✓ It allows you to perform operations such as reading, writing, updating, and deleting files and directories, which are essential for server-side applications and scripts.

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

Uses:

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

Key Features

- **Asynchronous and Synchronous Methods:** Provides both non-blocking and blocking methods for various file operations.
- **Error Handling:** Includes robust error handling to manage issues such as file not found or permission errors.
- **Directory Management:** Allows creation, deletion, and listing of directories.

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.

Use cases:

If your operations are not doing very heavy lifting like querying huge data from DB then go ahead with Synchronous way otherwise Asynchronous way.
In an Asynchronous way, you can show some progress indicator to the user while in the background you can continue with your heavyweight works. This is an ideal scenario for GUI based apps.

Note: In Node.js, when you work with the File System (fs) module,

- ✓ the term **utf-8** refers to the character encoding used when reading or writing text files.
- ✓ **UTF-8 (Unicode Transformation Format – 8-bit)** is a standard text encoding that represents characters from almost all languages.

- ✓ It's the most common encoding on the web and ensures that your file's text (like Hindi, Telugu, emojis, or special characters) is correctly stored and displayed.

Why use utf-8 in fs?

- ✓ By default, fs reads and writes binary buffers (raw bytes).
- ✓ If you want to handle text (strings), you must tell Node.js which encoding to use — and 'utf-8' is the most common one.

*/

Example-1: Synchronous file: create a directory(we use 'path.join' method)

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

const datafolder=path.join(__dirname,"data");

//if data folder is not existed, mkdirSync() this function creates folder name is 'data'
if(!fs.existsSync(datafolder))
{
    fs.mkdirSync(datafolder);
    console.log("data folder created");
}
//create new file inside cureent folder(Directory)
const filepath=path.join(datafolder,"example.txt");

//synchronous way of creating file with some text
fs.writeFileSync(filepath,"Keshav Memorial Institute of Technology");
console.log("file created successfully with some text");
```

```
//read data from file using synchronous way
const readfromfile=fs.readFileSync(filepath,"utf8");
console.log("File content:",readfromfile);
```

output:

```
file created successfully with some text

File content: Keshav Memorial Institute of Technology
```

exmple.txt: Keshav Memorial Institute of Technology

Example:2: append data into new line of existing file

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

const datafolder=path.join(__dirname,"data");
if(!fs.existsSync(datafolder))
{
    fs.mkdirSync(datafolder);
    console.log("data folder created");
}
const filepath=path.join(datafolder,"example.txt");
//synchronous way of creating file with append data

fs.appendFileSync(filepath,"\n autonomous institution ");

console.log("new file content added");
```

output: new file content added

exmple.txt:

Keshav Memorial Institute of Technology

autonomous institution

Example-3: async way of creating the file,read data from file, write data inot fiel and append data

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

const path=require('path');

//join method used to safely combine (join) multiple path segments into a single,
properly formatted file path.

const datafolder=path.join(__dirname,"data");

if(!fs.existsSync(datafolder))
{
    fs.mkdirSync(datafolder);
```

```
    console.log("data folder created");
}

const asyncfilepath=path.join(datafolder,"async-example.txt");

fs.writeFile(asyncfilepath,"hello, KMIT ,narayanguda",
    (err)=>{
    if(err)
       throw err;
    console.log("async file is created successfully");

})
//read text from existing file using async way

fs.readFile(asyncfilepath,"utf8",(err,data)=>{
    if(err)
        throw err;
    console.log("async file content:",data)
});
//adding data into  another line (async file)

fs.appendFile(asyncfilepath,`\n Departments:CSE,CSM,IT and CSD`,(err)=>{
    if(err) throw err;
    console.log("new line added to async file");
})

//reading updated data from file
fs.readFile(asyncfilepath,"utf8",(err,updateddata)=>{
    if(err)
        throw err;
    console.log("updated file content:",updateddata)
});
```

Output:

sync file is created successfully
new line added to async file
updated file content: hello, KMIT ,narayanguda
async file content: hello, KMIT ,narayanguda
Departments:CSE,CSM,IT and CSD

async-example.txt:

hello, KMIT ,narayanguda
Departments:CSE,CSM,IT and CSD

Example-4: Delete File

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

// Asynchronous
fs.unlink('example.txt', (err) => {
  if (err) console.error("Async Delete Error:", err);
  else console.log("Async Delete: File deleted");
});

// Synchronous
try {
  fs.unlinkSync('example.txt'); // make sure the file exists
  console.log("Sync Delete: File deleted");
} catch (err) {
  console.error("Sync Delete Error:", err);
}
```

Output:

```
Async Delete: File deleted
Sync Delete: File deleted
```

Example-5: Create Directory

```
// Asynchronous
fs.mkdir('myFolder', { recursive: true }, (err) => {
  if (err) console.error("Async Create Dir Error:", err);
  else console.log("Async Create: Directory created");
});

// Synchronous
try {
  fs.mkdirSync('myFolderSync', { recursive: true });
  console.log("Sync Create: Directory created");
} catch (err) {
  console.error("Sync Create Dir Error:", err);
}
```

Output:

Async Create: Directory created
Sync Create: Directory created

Example-6: Remove Directory

```
// Asynchronous
fs.rmdir('myFolder', (err) => {
  if (err) console.error("Async Remove Dir Error:", err);
  else console.log("Async Remove: Directory removed");
});

// Synchronous
```

```
try {
  fs.rmdirSync('myFolderSync');
  console.log("Sync Remove: Directory removed");
} catch (err) {
  console.error("Sync Remove Dir Error:", err);
}
```

Output:

```
Async Remove: Directory removed
Sync Remove: Directory removed
```

Example-7: **Read Directory**

```
// Asynchronous
fs.readdir('.', (err, files) => {
  if (err) console.error("Async Read Dir Error:", err);
  else console.log("Async Directory Read:", files);
});

// Synchronous
try {
  const files = fs.readdirSync('.');
  console.log("Sync Directory Read:", files);
} catch (err) {
  console.error("Sync Read Dir Error:", err);
}
```

Output:

Async Directory Read: ['fs_examples.js', 'node_modules', ...]
Sync Directory Read: ['fs_examples.js', 'node_modules', ...]

6. Http Module:

- ✓ The **http module** is a **built-in Node.js module** that allows you to create **web servers and handle HTTP requests**.
- ✓ Using this module, you can build simple web servers, APIs, or respond to client requests.
- ✓ No installation is required; it comes with Node.js.

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

Creating a Simple HTTP Server:

- ✓ for creating http server we require http model Here import http and use createServer method which contains req&res,
- ✓ When you use http.createServer(callback):
  ---
  req = Incoming HTTP Request object
  res = HTTP Response object
- ✓ These are automatically passed by Node.js when someone accesses your server (for example, from a browser).
- ✓ we require listen method which contains port number(example: 3000)
- ✓ now http server is created (open browser type-localhost:3000)
- ✓ **in this porgram , if u run in browser (localhost:3000) it is loading continuosly**

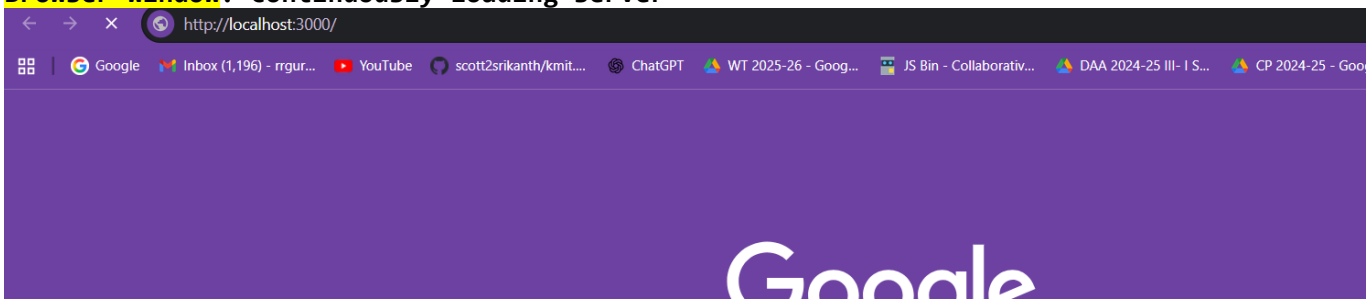
Example-1:create httpserver **Filename:server.js**

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

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

    console.log(req,"req");

});

const port=3000;
server.listen(port,()=>{
    console.log(`server is now listening to port ${port}`)
})
```

Console output:

```
server is running on  port number:3000
```

Browser window: continuously loading server



Common HTTP Status Codes:

Code	Meaning	Description
200	OK ✓	Request was successful, and the server is returning data.
201	Created	A new resource was created (used for POST).
400	Bad Request ✗	The client sent invalid data.
401	Unauthorized 🔒	Login/authentication required.
404	Not Found ⊘	The requested page or resource does not exist.
500	Internal Server Error ✹	Something went wrong on the server.

What is { "content-type": "text/plain" }

This is called the HTTP header — specifically, the Content-Type header.
It tells the browser what kind of data is being sent in the response body.

Content-Type	Meaning
"text/plain"	Simple plain text
"text/html"	HTML document
"application/json"	JSON data
"image/png"	PNG image
"text/css"	CSS stylesheet
"application/javascript"	JavaScript code

Why Content-Type is Mandatory

- ✓ It's not always technically required, but it's strongly recommended (almost mandatory) because:
- ✓ Browsers need to know how to interpret the data.
- ✓ If you send HTML but don't specify "text/html", the browser might show
- ✓ it as plain text instead of rendering the webpage.
- ✓ APIs and apps need to parse data correctly.

For example, if a client expects JSON,
it reads based on the Content-Type: application/json header.

Example-2: For getting responses on the browser Filename:server.js

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

const server=http.createServer((req,res)=>
{
//Returns the HTTP method used by the client
console.log("Method:", req.method);
//Returns the requested path (URL) after the domain name.
console.log("URL:", req.url);
//Returns an object containing all HTTP headers sent by the client.
console.log("Headers:", req.headers);

res.writeHead(200,{"Content-Type":"text/plain"});//sets the response headers.
res.end("hello node.js from http modules")


//Server with Different HTTP Methods
if (req.method === "GET") {
    res.end("This is a GET request");
  } else if (req.method === "POST") {
    res.end("This is a POST request");
  } else {
    res.end("Other HTTP method");
  }
});
const port=3002;
server.listen(port,()=>{
    console.log(`server is now listening to port ${port}`)
})
```

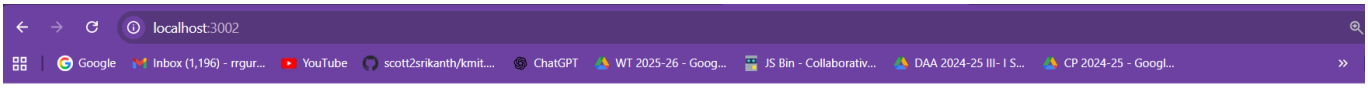
Run: node server.js

Console output: server is now listening to port 3002

Method: GET
URL: /
Headers: {
 host: 'localhost:3002',
 connection: 'keep-alive',
 'cache-control': 'max-age=0',

```
  'sec-ch-ua': '"Google Chrome";v="141", "Not?A_Brand";v="8", "Chromium";v="141"',
  'sec-ch-ua-mobile': '?0',
  'sec-ch-ua-platform': '"Windows"',
  'upgrade-insecure-requests': '1',
  'user-agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/141.0.0.0 Safari/537.36',
  accept:
'text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,ap
plication/signed-exchange;v=b3;q=0.7',
  'sec-fetch-site': 'none',
  'sec-fetch-mode': 'navigate',
  'sec-fetch-user': '?1',
  'sec-fetch-dest': 'document',
  'accept-encoding': 'gzip, deflate, br, zstd',
  'accept-language': 'en-US,en;q=0.9'
}
node:events:496
    throw er; // Unhandled 'error' event
    ^
Error [ERR_STREAM_WRITE_AFTER_END]: write after end
   at ServerResponse.end (node:_http_outgoing:1097:15)
   at Server.<anonymous> (D:\WT 2025-26 III YR I SEM\DAY WISE TOPICS\NodeJS\nodeJS_Porgrams\6.http-
module\server2.js:29:9)
   at Server.emit (node:events:518:28)
   at parserOnIncoming (node:_http_server:1153:12)
   at HTTPParser.parserOnHeadersComplete (node:_http_common:117:17)
Emitted 'error' event on ServerResponse instance at:
   at emitErrorNt (node:_http_outgoing:927:9)
   at process.processTicksAndRejections (node:internal/process/task_queues:91:21) {
 code: 'ERR_STREAM_WRITE_AFTER_END'
}
Node.js v22.16.0
```

Browser output:




```
hello node.js from http modules
```

```
/* in terminal you are getting error because

You already ended the response once using res.end("hello node.js from http modules");
```

Then, immediately after that, you again call res.end() inside your if conditions. Once a response is ended, you cannot write or end again – that's why Node throws ERR_STREAM_WRITE_AFTER_END.
To avoid this use res.write() method
*/

Example-3: using write() method **Filename: Server.js**

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

const server = http.createServer((req, res) => {
  res.writeHead(200, {"Content-Type": "text/plain"});

  res.write("hello node.js from http modules \n")
  let message = "";

  if (req.method === "GET") {
    message = "This is a GET request";
  } else if (req.method === "POST") {
    message = "This is a POST request";
  } else {
    message = "Other HTTP method";
  }

  // Send only once
  res.end(message);
});

const port = 3007;
server.listen(port, () => {
  console.log(`Server is now listening on port ${port}`);
});
```

Run: **node server.js**

Console output: **server is now listening to port 3007**

Browser output:


```
hello node.js from http modules
This is a GET request
```

Routing in NodeJS

✓ Routing is the process of deciding how a server should respond to different requests made by users.
✓ When you visit a website or use an app, your browser sends a request to the server.
✓ Routing determines how the server handles these requests based on the URL you visit and the type of request (such as viewing a page, submitting a form, or deleting something).

Routing with the Native HTTP Module

✓ In NodeJS, routing is done by directly using the built-in http module to create a server that listens for client requests.
✓ You manually handle different HTTP methods and URLs. This gives you full control over the request/response cycle, but it requires more boilerplate code and manual handling of request data.

How It Works

- You create an HTTP server with http.createServer().
- Inside the server's callback function, you inspect the incoming request (req), including its URL (req.url) and HTTP method (req.method).
- Based on these parameters, you can determine which functionality to execute (i.e., which route to trigger).

Example-4: creating routes using url variable

```
const http=require('http')
const server=http.createServer((req,res)=>{
    const URL=req.url;
    if(URL=="/")
    {
        res.writeHead(200,{"Content-Type":"text/plain"})
          res.write('Hello World\n');
        res.end("home page")
    }
    else if (URL=="/projects")
    {
        res.writeHead(200,{"Content-Type":"text/html"})
       // res.end("will get porjetcs directory data")
        res.end("<h1>Hello Node.js</h1><p>This is an HTML res/p>");
    }
    else if (URL=="/jsondata")
    {
        const data = { name: "John", city: "Hyderabad" };
        res.writeHead(200, { "content-type": "application/json" });
        res.end(JSON.stringify(data));
    }
    else
        res.writeHead(404,{"Content-Type":"text/plain"})
        res.end("this page is can not be found")
})

const port=3009;
```

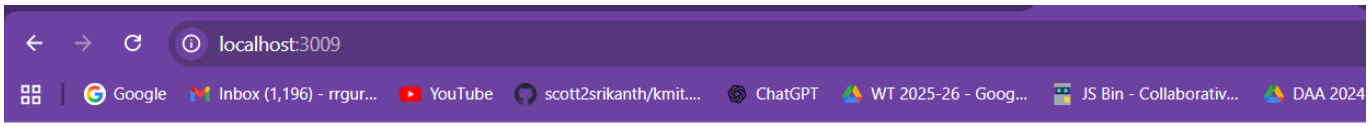
```
server.listen(port,()=>{
    console.log(`server is listening to port number ${port} `)
})
```

Run: node server.js

Console output:

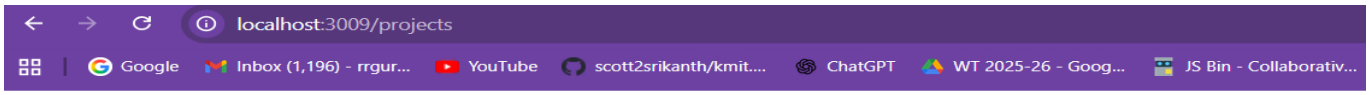
server is listening to port number 3009

Browser output: **url: http://localhost:3009/**



Hello World
home page

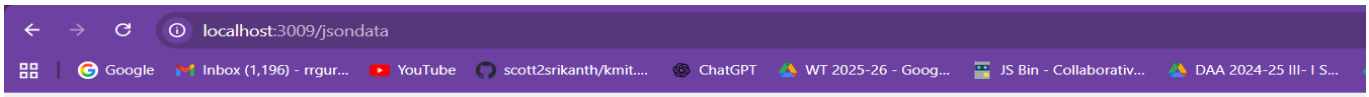
Browser output: **url: http://localhost:3009/projeccts**



Hello Node.js

This is an HTML res

Browser output: **url: http://localhost:3009/jsondata**

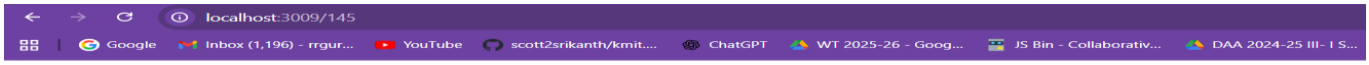


Pretty-print ☑


```
{
  "name": "John",
  "city": "Hyderabad"
}
```


Browser output: **url: http://localhost:3009/145**

Note: entered wring route name so else block is executed

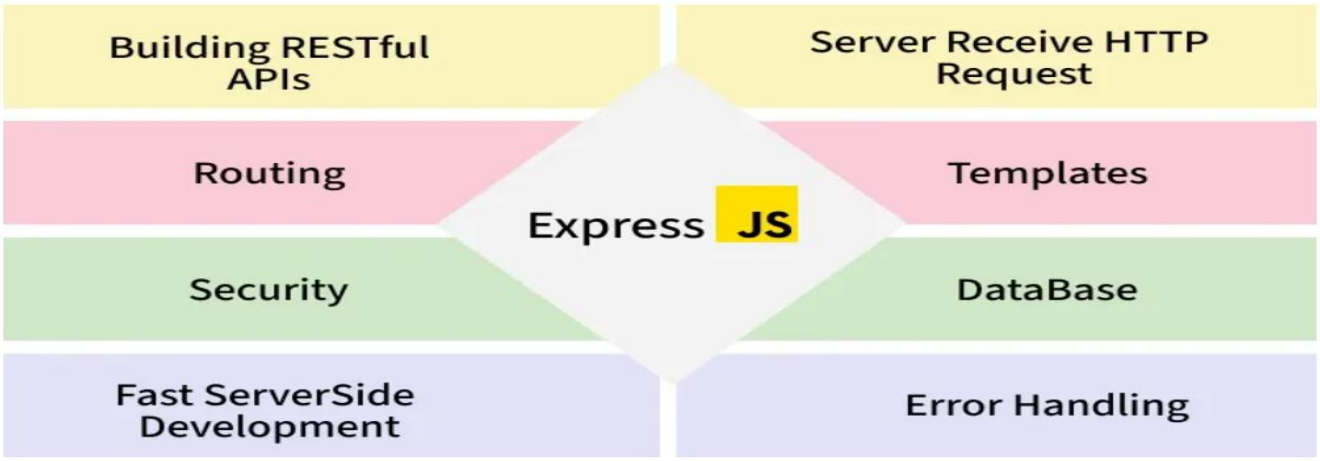


this page is can not be found

Introduction to ExpressJS: Lifecycle and routing of Express App, first web server, reading configuration parameters, Handling request and response parameters.

Introduction to ExpressJS:

Express.js is a minimal and flexible Node.js web application framework that simplifies building server-side applications and APIs.



- **Middleware-based**: Uses middleware functions to handle requests, responses, and add features like authentication, logging, or error handling.
- **Routing system**: structure—developers can design apps as they want.
- **Integration**: Works well with databases (MongoDB, MySQL, PostgreSQL) and front-end frameworks.
- **Scalable**: Suitable for small apps to large-scale enterprise APIs.

Problem in node.js:

- Routing logic is **manual** (`if-else` for each URL).
- No automatic body parsing for JSON or forms.
- No built-in error handling.
- Managing static files (HTML, CSS, JS) is complex.
- Large projects become **messy and hard to maintain**.

Advantage of Express.js:

✓ **Express.js** was built to make web development with Node.js **simple, structured, and efficient**.
✓ It sits **on top of the Node.js HTTP module** and provides features like:

Feature	Description
Routing	Handles multiple routes easily
Middleware	Processes requests before reaching routes
JSON & Form Parsing	Reads body data automatically
Static Files	Serves HTML, CSS, JS easily
Error Handling	Provides a clean way to handle errors
Modular Code	Split code into routers, controllers, etc.

Purpose of Express.js

Express.js is used to build web applications and RESTful APIs quickly and efficiently.

You use it when you want to:

1. Create a **web server** (serve HTML pages).
2. Build a **REST API** (for mobile apps or frontend frameworks).
3. Serve **static files** (images, CSS, JS).
4. Connect to **databases** (MongoDB, MySQL, etc.).
5. Build **backend logic** (authentication, routing, validation).

When Do We Use Express.js?

Situation	Should You Use Express?	Why
Building a simple website	✅ Yes	You can serve HTML/CSS easily
Creating an API for frontend (React, Angular, Vue)	✅ Yes	Express is ideal for RESTful APIs
Working with databases	✅ Yes	Express integrates easily with MongoDB/MySQL
Building a large project	✅ Definitely	You can organize routes, middleware, and controllers
Just testing Node basics	❌ Not necessary	You can use plain Node.js for simple "Hello World"

Eample-1: Create an Express server in Node.js

---

1. Make sure Node.js is installed

---

2. Create a new project folder: Open your terminal and type below commands

---

```
mkdir ExpressJS
cd ExpressJS
```

3. install package.json file: This creates a file that stores your app info and dependencies.

---

command: npm install -y

we will get below format output:

```
PS D:\WT 2025-26 III YR I SEM\DAY WISE TOPICS\ExpressJS\ExpressJS Programs\ExpressJS>
npm init -y
Wrote to D:\WT 2025-26 III YR I SEM\DAY WISE TOPICS\ExpressJS\ExpressJS
Programs\ExpressJS\package.json:

{
```

```
  "name": "expressjs",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": ""
}
```

4.install Express: command: npm install express

we will get below format output:

```
PS D:\WT 2025-26 III YR I SEM\DAY WISE TOPICS\ExpressJS\ExpressJS Programs\ExpressJS>
npm install express

added 68 packages, and audited 69 packages in 5m

16 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

5.Create the Express Server

```
*/
// Import express module
const express = require("express");

// Create an Express app
const app = express();

/*above line creates an Express application object.
->express() is a function that initializes a new Express app.
->The variable 'app' will be used to configure routes, middleware, and server settings.

Think of it like:
----------------
You are creating the “main controller” for your web server.
'app' represents your whole web server.

*/
// Define a route (when user visits http://localhost:3000)
app.get("/", (req, res) => {
res.send("Hello from Express server!");
});
```

```
/*above line defines a route handler for GET requests to the path /.
```

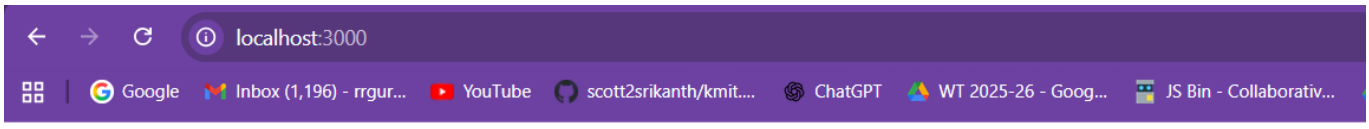
Part	Meaning
`app.get()`	Method to define a route for **HTTP GET requests** (like when someone opens a URL in a browser).
`"/"`	The **path or endpoint**. Here `/` means the **home page** (root URL).
`(req, res) => { ... }`	This is a **callback function** that runs when someone visits that route.
`req`	The **request object** – contains details about the incoming request (like headers, parameters, body, etc.).
`res`	The **response object** – used to send a reply back to the client/browser.
`res.send("Hello from Express server!")`	Sends the text **"Hello from Express server!"** as the HTTP response to the browser.

```
*/

// Define a port
const PORT = 3000;

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

/*purpose:
-----------
app.listen(PORT, ...)
This command starts your Express server and makes it listen for incoming requests (like
from a browser or API client).
The first argument PORT tells which port number the server should listen on (e.g.,
3000).
*/
```

Run: node server.js

Server is running on http://localhost:3000

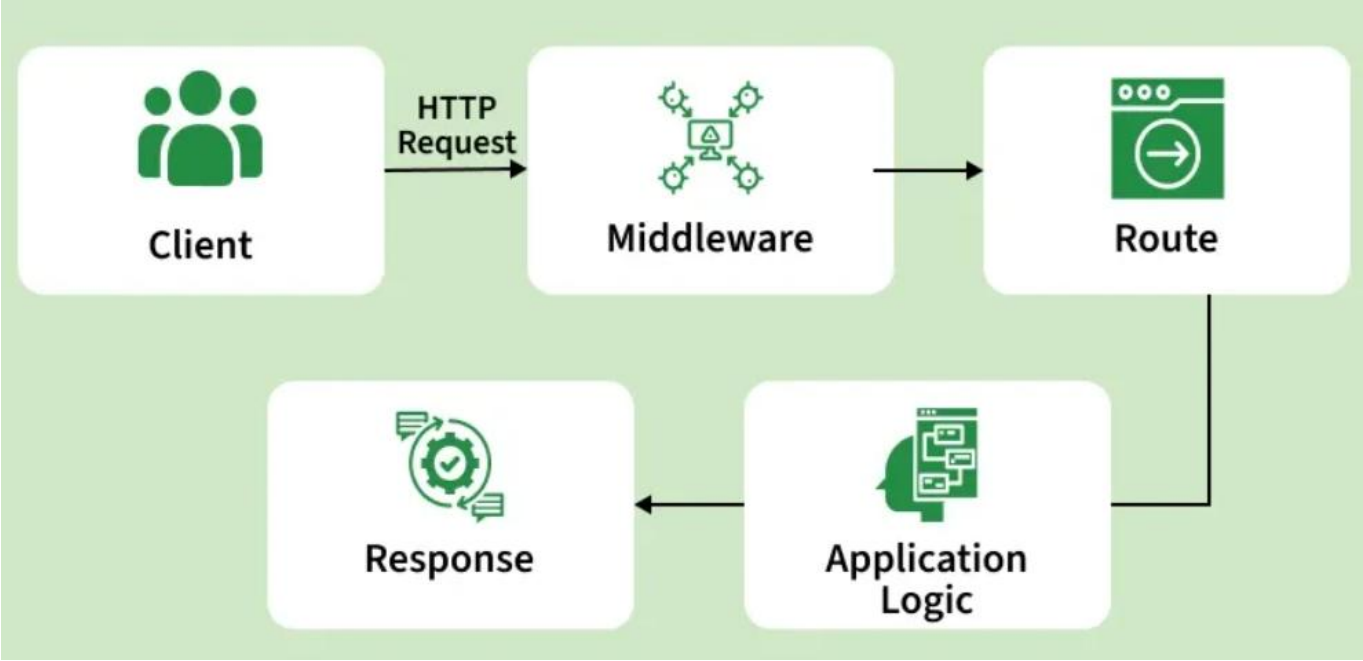
Browser output:



Hello from Express server!

How Express.js Works?

Express.js works by handling client requests through a series of steps called the **request-response cycle**, using middleware and routing to process requests and send responses efficiently.



- A client sends a request (e.g., browser or app) to the Express server.
- The request passes through middleware functions that can modify the request, perform logging, authentication, or other tasks.
- Express matches the request to a route handler based on the URL and HTTP method.
- The route handler processes the request and prepares a response.
- The server sends the response back to the client.
- If any error occurs, it is caught by error-handling middleware to ensure the server remains stable.

Express.js application lifecycle:

The lifecycle of an Express.js application can be broken down into **two main phases**: **the application lifecycle** and **the request-response lifecycle**. The application lifecycle involves starting and stopping the server, while the request-response lifecycle details the journey of a single request from the moment it's received to when a response is sent.

1. Express.js application lifecycle

This is the overall lifespan of your server from when you launch it until you shut it down.

1. **Server initialization**: You begin by importing the `express` library and creating an `app` instance. You also define the port your application will listen on.

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

2. **Configuration and middleware setup**: This is where you configure your application and set up any application-level middleware. You use **`app.use()`** to define middleware that will be executed for every incoming request. Examples include:

- **express.json()** for parsing JSON request bodies.
- Third-party middleware like morgan for logging requests.
- Custom authentication or logging middleware.

3. **Define routes**: You create routes using methods like **app.get()**, **app.post()**, etc., to handle specific HTTP methods and URL paths. Route handlers are another type of middleware that Express uses to complete the request.

4. **Start the server**: You use app.listen() to bind your application to a specific port and begin accepting incoming requests. This makes the server accessible. The listen method can take an optional callback function that executes once the server is successfully running, which is useful for logging that the application is ready.

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

5. **Server termination**: The server continues to run until it is explicitly shut down. This can be done manually (e.g., using Ctrl+C in the terminal) or programmatically **via process.exit()**.

2. Express.js request-response lifecycle

This process describes what happens when a single client request is processed by a running Express application.

1. **Request reception**: A client, such as a web browser, sends an HTTP request to the server, including information like the HTTP method (GET, POST), the URL, headers, and body data.
2. **Request and response objects creation**: Express receives the request and creates two objects:
   - req (request object): Contains all the information from the incoming request.
   - res (response object): Used by Express to build and send the response back to the client.
3. **Middleware pipeline**: The request and response objects are passed through a stack of middleware functions, which are executed in the order they were defined. Each middleware function can:
   - Execute code (e.g., logging, validation).
   - Modify the req or res objects (e.g., adding a user to req.user after authentication).
   - Pass control to the next middleware by calling the next() function.
4. **Route handler execution**: If a middleware function calls next(), the process continues until it finds a route handler that matches the request's URL and HTTP method. The handler function is responsible for executing the business logic for that specific endpoint.
5. **Response generation**: The route handler uses the res object to prepare a response for the client. This can involve fetching data from a database, performing calculations, and setting the response status and data.
6. **Sending the response**: The final response is sent back to the client using a method like res.send(), res.json(), or res.end(). Once the response is sent, the request-response cycle is complete.

7. **Error handling (optional)**: If any part of the middleware or route handling process throws an error, Express will look for an error-handling middleware, which is defined with four arguments: `(err, req, res, next)`. If defined, this middleware can send a proper error response instead of letting the application crash.

Routing of Express App:

Routing in Express.js is the process of mapping incoming HTTP requests (defined by method and URL) to specific handler functions. It enables developers to configure endpoints for various paths and operations, such as rendering views, processing form data, or performing CRUD actions on resources.

Syntax: `app.METHOD(PATH, HANDLER);`

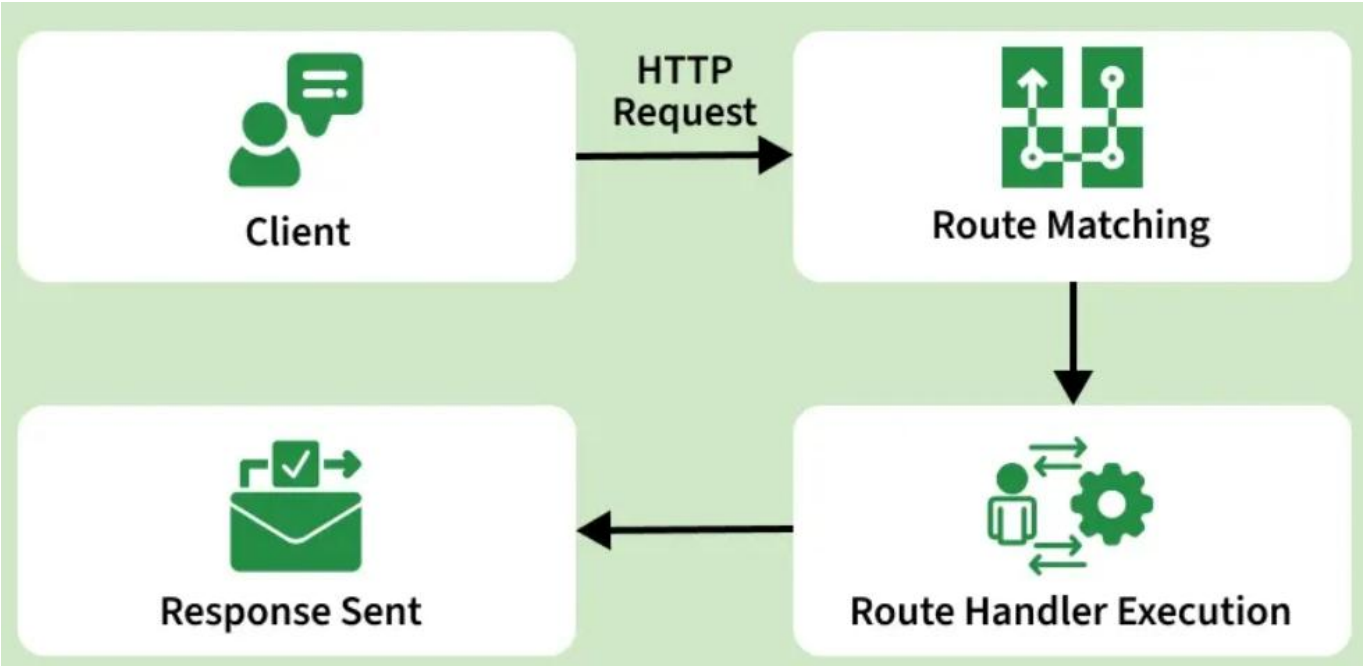
In the above syntax:

- **app:** Represents an instance of an Express application.
- **METHOD:** Represents an HTTP method like GET, POST, PUT, DELETE.
- **PATH:** Defines the endpoint (route) where the request will be handled.
- **HANDLER:** A function that executes when the route is accessed.

How Routing Work in ExpressJS?

Routing in Express.js follows a simple flow to handle client requests:

- **HTTP Request:** Shows the incoming request with method and path (e.g., GET /users).
- **Route Matching:** Lists all defined routes and shows which route matches the request (green check) and which don't (red cross).
- **Handler Execution:** Indicates that the matching route's handler function is executed (req, res).
- **Sending Response:** Shows how the server sends a response back to the client using methods like res.send(), res.json(), or res.render().



Now let's understand this with the help of example:

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

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

app.listen(3000, () => {
    console.log('Server is running on port 3000');
});
```

In this example

- express is imported and an app instance is created.
- app.get('/', ...) defines a route for GET requests to the root URL / and sends a response "Welcome to Express Routing!".
- app.listen(3000, ...) starts the server on port 3000 and logs a message when it's running.

Types of Routes:

1.Basic Routes in ExpressJS: Basic routing involves defining a URL and specifying an HTTP method (GET, POST, PUT, DELETE, etc.).

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

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

app.listen(3000, () => {
    console.log('Server is running on port 3000');
});

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

app.post('/submit', (req, res) => {
    res.send('Form Submitted Successfully!');
});
```

Explanation:

/home handles GET requests.
/submit handles POST requests.

2.Route Parameters in ExpressJS:

Route parameters allow capturing dynamic values from URLs, making routes flexible and reusable.

```
app.get('/users/:userId', (req, res) => {
    res.send(`User ID: ${req.params.userId}`);
});
```

Explanation- :userId is a route parameter.
req.params.userId extracts the value.

3.Optional and Multiple Route Parameters

Express allows defining optional parameters and multiple parameters in a single route.

```
app.get('/products/:productId?', (req, res) => {
    res.send(`Product ID: ${req.params.productId || 'No product selected'}`);
});
```

Explanation: products/123 → Product ID: 123
/products/ → No product selected

4.Multiple Route Parameters

```
app.get('/posts/:category/:postId', (req, res) => {
    res.send(`Category: ${req.params.category}, Post ID: ${req.params.postId}`);
});
```

Explanation: **/posts/tech/456 → Category: tech, Post ID: 456**

5.Query Parameters in ExpressJS: Query parameters are used for filtering or modifying requests. They appear after the ? symbol in URLs.

```
app.get('/search', (req, res) => {
    res.send(`Search results for: ${req.query.q}`);
});
```

Explanation: **req.query.q extracts q from the URL.**

6.Route Handlers in ExpressJS:

Route handlers define how Express responds to requests.

```
app.get('/example', (req, res, next) => {
    console.log('First handler executed');
    next();
}, (req, res) => {
    res.send('Response from second handler');
});
```

Explanation: The next() function passes control to the next handler.

7.Route Chaining in ExpressJS

Chaining allows defining multiple handlers for a route using .route().

```
app.route('/user')
    .get((req, res) => res.send('Get User'))
    .post((req, res) => res.send('Create User'))
    .put((req, res) => res.send('Update User'))
    .delete((req, res) => res.send('Delete User'));
```

Explanation: /user supports multiple HTTP methods using .route().

Example:1 **Create different types of routes using ExpressJs**

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

const app=express();

const port=3090;

app.listen(port,()=>{
    console.log(`server is runningh on port number ${port}`);

})
//root route
app.get("/",(req,res)=>{
    res.send("welcome to home page")
});

//Add More Routes (Optional)

app.get("/about", (req, res) => {
  res.send("About Page");
});

app.get("/contact", (req, res) => {
  res.send("Contact Us");
});


// Example: get all prodcuts
app.get("/products",(req,res)=>{
    const products=[
        {
            id:1,
            label: 'prodcut1'
        },
        {
            id:2,
            label: 'prodcut2'
        },
```

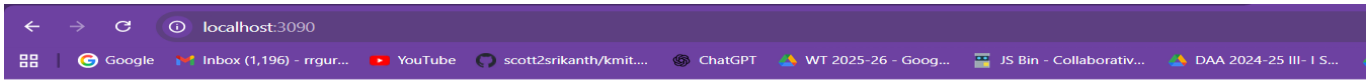
```
        {
            id:3,
            label: 'prodcut3'
        }
    ]
    res.json(products)

})

//example: get dynamic prodcuts

app.get("/products/:id",(req,res)=>{

//on above line :id:-id not fixed, we will use any name
/*
req.params is an object in Express that contains all the route parameters (also called URL parameters) defined in your route path.
 It is a built-in property of the request (req) object, so yes — the name params is fixed in Express.
 */
   //chacking param values

    console.log("req.params:",req.params);
    //output:req.params: [Object: null prototype] { id: '2' }

    const productID=parseInt(req.params.id);
    const products=[
        {
            id:1,
            label: 'prodcut 1'
        },
        {
            id:2,
            label: 'prodcut 2'
        },
        {
            id:3,
            label: 'prodcut 3'
        }
    ]
    const getSinngleProduct=products.find((product)=>product.id===productID) ;

    if(getSinngleProduct)
    {
        res.json(getSinngleProduct)
    }else
    {
        res.status(400).send("product is not found try different with ID")
    }
});
```

Run: **nodemorn server.js**

Terminal output:

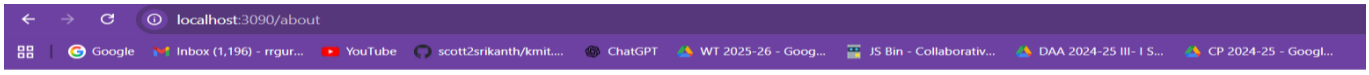
```
[nodemon] 3.1.10
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node index.js`
server is runningh on port number 3090
[nodemon] clean exit - waiting for changes before restart
```

Browser output: url: http://localhost:3090/ or **http://127.0.0.1:3090/**



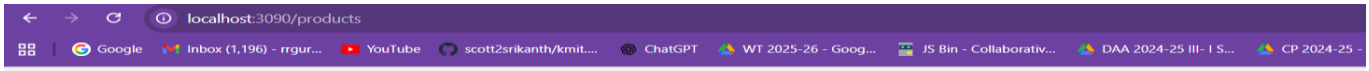
welcome to home page

url: http://localhost:3090/about or **http://127.0.0.1:3090/about**



About Page

url: http://localhost:3090/products

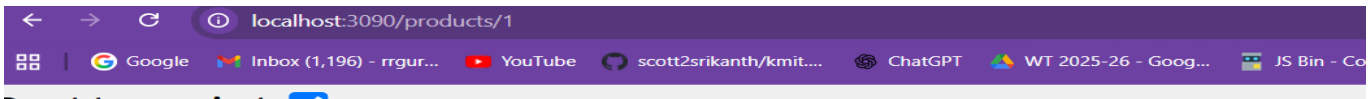


Pretty-print ☑


```
[
  {
    "id": 1,
    "label": "prodcut1"
  },
  {
    "id": 2,
    "label": "prodcut2"
  },
  {
    "id": 3,
    "label": "prodcut3"
  }
]
```


---

url: http://localhost:3090/products/1



Pretty-print ☑


```
{
  "id": 1,
  "label": "prodcut 1"
}
```

Example:2 Router-level Routing (Modular Routing)

```
/* When your project grows, you separate routes into different files using
express.Router().

✳ Example – Modular Routing

project structure:
-----------------
project/
├── server.js
└── routes/
    └── userRoutes.js

*/
```

Filename: userRoutes.js

```
//Router-level Routing (Modular Routing)

const express = require('express');
const router = express.Router();

// Define routes
router.get('/', (req, res) => {
  res.send('All Users');
});

router.get('/:id', (req, res) => {
  res.send(`User with ID: ${req.params.id}`);
});

module.exports = router;  // Export router
```

Filename: Server.js

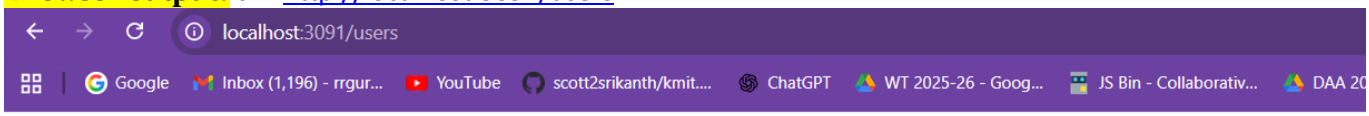
```
const express = require('express');
const app = express();

// Import router
const userRoutes = require('./routes/userRoutes');

// Use router middleware
app.use('/users', userRoutes);

app.listen(3002, () => {
  console.log('Server running on http://localhost:3002');
});


/*run
http://localhost:3002/users/ → “All Users”
```

```
http://localhost:3002/users/101 → "User with ID: 101"
*/
```

Run: nodemorn server.js

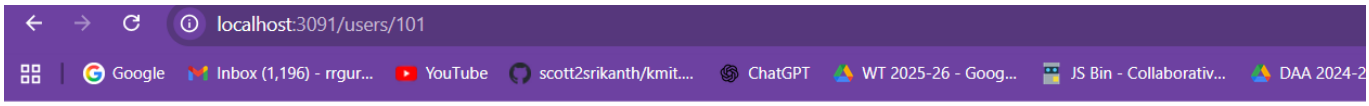
Terminal output: **Server running on http://localhost:3091**

Browser output: url: http://localhost:3091/users



All Users

url: http://localhost:3091/users/101



User with ID: 101

Example:3- Using app.route() Method

```
/*app.route() allows chaining multiple HTTP methods for the same route path – cleaner
and avoids  repetition

*/
```

Filename: app.js

```
const express = require('express');
const app = express();
app.use(express.static(__dirname));

app.route('/student')
  .get((req, res) => {
    res.send('GET request - Get student info');
  })
  .post((req, res) => {
    res.send('POST request - Add new student');
  })
  .put((req, res) => {
    res.send('PUT request - Update student');
  });

app.listen(3092, () => {
  console.log('Server running on http://localhost:3092');
});
```

Filename: index.html

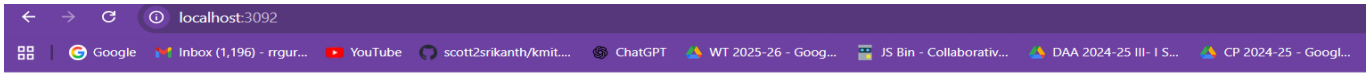
```
<!DOCTYPE html>
<html>
  <head>
    <title>Test Express Routes</title>
  </head>
  <body>
    <h2>Test /student route</h2>
    <button onclick="sendRequest('GET')">GET</button>
    <button onclick="sendRequest('POST')">POST</button>
    <button onclick="sendRequest('PUT')">PUT</button>

    <p id="output"></p>

    <script>
      async function sendRequest(method)
      {
        const response = await fetch('http://localhost:3092/student', {
          method: method,
        });
        const text = await response.text();
        document.getElementById('output').innerText = text;
      }
    </script>
  </body>
</html>
```

Run: nodemorn app.js

Terminal output:

[nodemon] 3.1.10
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node app.js`
Server running on http://localhost:3092
[nodemon] clean exit - waiting for changes before restart

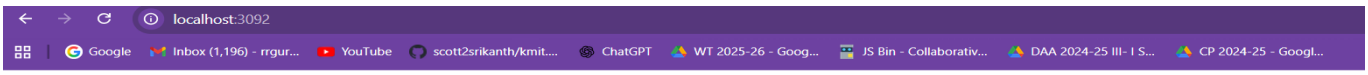
Browser output: url: http://localhost:3092/



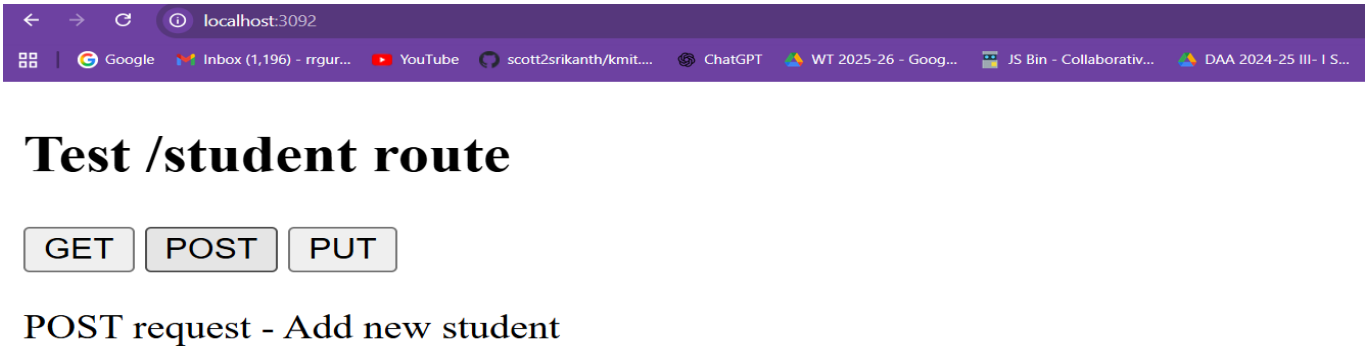
Test /student route

GET POST PUT

Click on GET Button it calling get Mehtod



Click on POST Button it calling get Mehtod



Click on PUT Button it calling get Mehtod

Example:4: Middleware routing Example(next method)

/* An Express application is essentially a series of middleware function calls.

✓ Middleware functions are functions that have access to the request object (req), the response object (res), and the next middleware function in the application's request-response cycle.

✓ The next middleware function is commonly denoted by a variable named 'next'.

✓ Middleware functions can perform the following tasks:
- Execute any code.
- Make changes to the request and the response objects.
- End the request-response cycle.
- Call the next middleware function in the stack.

- ✓ If the current middleware function does not end the request-response cycle, it must call next() to pass control to the next middleware function. Otherwise, the request will be left hanging.
- ✓ An Express application can use the following types of middleware:
  - Application-level middleware
  - Router-level middleware
  - Error-handling middleware
  - Built-in middleware
  - Third-party middleware
- ✓ You can load application-level and router-level middleware with an optional mount path.
- ✓ You can also load a series of middleware functions together, which creates a sub-stack of the middleware system at a mount point.

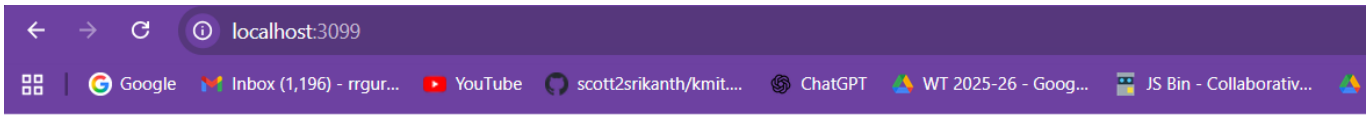
*/

Run:

Terminal output:

Server running on http://localhost:3099

Browser output: url: http://localhost:3099/

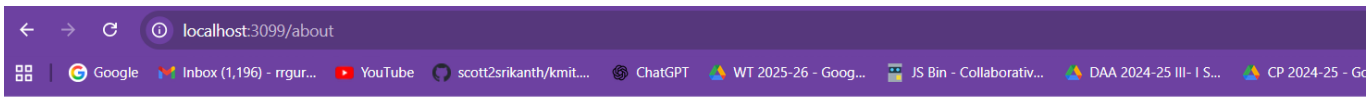


Welcome to Home Page

now check Terminal output: you will get time

Server running on http://localhost:3099
GET / at 2025-10-26T16:43:30.487Z
GET /favicon.ico at 2025-10-26T16:43:30.521Z

Browser output: url: http://localhost:3099/about



About Page

now check Terminal output: you will get time of about route

Terminal output:

Server running on http://localhost:3099
GET / at 2025-10-26T16:43:30.487Z

GET /favicon.ico at 2025-10-26T16:43:30.521Z
GET /about at 2025-10-26T16:45:21.309Z

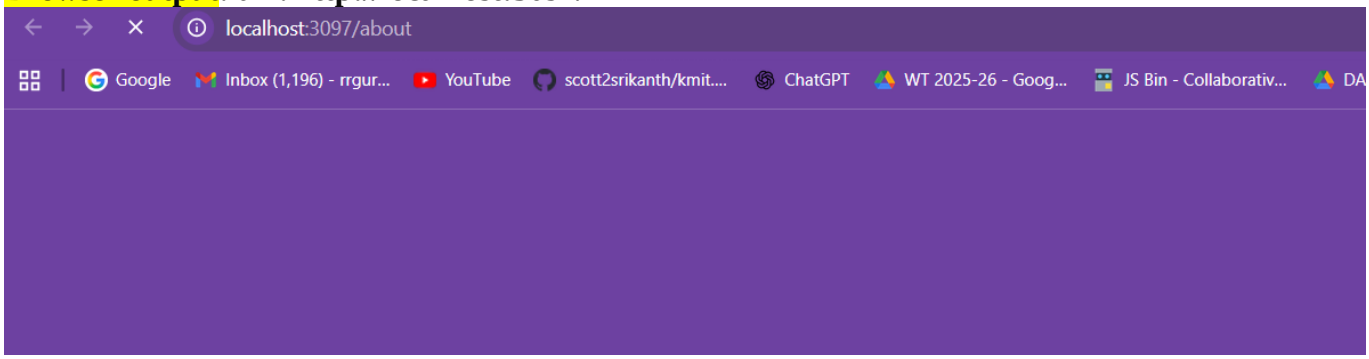
Note : if comment the next () , we ar e not working routes -loading only but it is working geeting output on terminal

```
console.log(`${req.method} ${req.url} at ${new Date().toISOString()}`);
```

Terminal output:

Server running on http://localhost:3097
GET / at 2025-10-26T16:50:13.868Z
GET /about at 2025-10-26T16:50:36.824Z

Browser output: url: http://localhost:3097/



First Web Server in Express.js

Let's create your first Express web server.

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

// Define a route
app.get('/', (req, res) => {
  res.send('Hello World! My first Express Web Server');
});

// Start the server
app.listen(3000, () => {
  console.log('Server running on http://localhost:3000');
});
```

Explanation

Code Line	Description
`const express = require('express')`	Imports the Express module
`const app = express()`	Creates an Express app object
`app.get('/', ...)`	Handles GET requests for the home route
`res.send()`	Sends a text response to the client
`app.listen(3000)`	Starts the server on port 3000

Reading Configuration Parameters in Express.js

In real-world Express.js applications, we **shouldn't hardcode** values like:

- Port numbers
- Database URLs
- API keys
- Secret tokens

Instead, we use **configuration parameters**, which can be **read from environment variables** or **config files**.

This makes your app:

- **Flexible** – works in multiple environments (development, test, production)
- **Secure** – sensitive data isn't stored directly in code
- **Easier to maintain**

What are Environment Variables?

Environment variables are **key-value pairs** stored outside your code.

They define configuration for your app — like:

```
PORT=5000
DB_URL=mongodb://localhost:27017/studentDB
```

You can define them:

- In the terminal before running the app
- In `.env` files (using the `dotenv` package)

Example-1: Filename: app.js

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

// Read port number from environment variable
const PORT = process.env.PORT || 3000; // Default is 3000 if not provided

app.get('/', (req, res) => {
  res.send(`Server is running on port ${PORT}`);
});

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

Explanation:

Code	Description
`process.env`	Built-in Node.js object that stores environment variables
`process.env.PORT`	Reads the value of variable `PORT`
`app.listen(PORT)`	Starts server on the configured port

Run the Program:

Case 1: Without Environment Variable **node app.js**

Output (Console): Server started on http://localhost:3000

Browser Output: Server is running on port 3000

Example 2: **Using .env file (Recommended for Projects)**

Step 1: Install dotenv

Command : **npm install dotenv**

Step 2: Create a .env file

PORT=4000
APP_NAME=StudentApp

File name: **app.js**

```
require('dotenv').config(); // Load variables from .env

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

const PORT = process.env.PORT;
const APP_NAME = process.env.APP_NAME;

app.get('/', (req, res) => {
  res.send(`Welcome to ${APP_NAME} running on port ${PORT}`);
});

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

Explanation:

Line	Description
`require('dotenv').config()`	Loads `.env` file variables into `process.env`
`.env` file	Stores app configuration securely
`process.env.PORT`	Reads value defined in `.env` file
`process.env.APP_NAME`	Reads app name from `.env` file

Run the Program: **nodemorn app.js**
Output (Console): StudentApp started on http://localhost:4000
Browser Output:

Welcome to StudentApp running on port 4000

Benefits of Using Configuration Parameters:

Benefit	Description
Security	Sensitive data (like passwords) not visible in source code
Portability	Same app can run in different environments
Flexibility	Easily change port, DB URL, or API keys without modifying code
Maintainability	Cleaner, well-organized application setup

Summary:

Concept	Description	Example
`process.env`	Stores environment variables	`process.env.PORT`
`.env` file	Stores config values	`PORT=4000`
`dotenv`	Loads `.env` file variables	`require('dotenv').config()`
Default value	Fallback if variable not set	`process.env.PORT

Example:

File name: `.env File`

```
# Application Configuration
PORT=5000
DB_URL=mongodb://localhost:27017/studentDB
APP_NAME=StudentManagementApp
```

File name:app.js

```
require('dotenv').config();
const express = require('express');
const app = express();

const PORT = process.env.PORT || 8080;
const DB_URL = process.env.DB_URL || "mongodb://localhost:27017/myDB";

app.get('/', (req, res) => {
  res.send(`App running on port ${PORT} and connected to ${DB_URL}`);
});

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

Output: App running on port 8080 and connected to mongodb://localhost:27017/myDB

Handling Request and Response Parameters:

Express gives two objects for every route:

- **`req`** → Request (contains data from the client)
- **`res`** → Response (used to send data back)

a) Route Parameters (req.params)

Example: server.js

```
const express=require('express')
const app=express();
const port=3090;

app.listen(port,()=>{
    console.log(`server is runningh on port number ${port}`);

})
// Example: get all prodcuts
app.get("/products",(req,res)=>{
    const products=[
        {
            id:1,
            label: 'prodcut1'
        },
        {
            id:2,
            label: 'prodcut2'
        },
        {
            id:3,
            label: 'prodcut3'
        }
    ]
    res.json(products)

})
//example: get dynamic prodcuts

app.get("/products/:id",(req,res)=>{

//on above line :id:-id not fixed, we will use any name
/*
req.params is an object in Express that contains all the route parameters (also called
URL parameters) defined in your route path.
 It is a built-in property of the request (req) object, so yes — the name params is
fixed in Express.
 */
   //chacking param values

    console.log("req.params:",req.params);
    //output:req.params: [Object: null prototype] { id: '2' }

    const productID=parseInt(req.params.id);
    const products=[
        {
```

```
            id:1,
            label: 'prodcut 1'
        },
        {
            id:2,
            label: 'prodcut 2'
        },
        {
            id:3,
            label: 'prodcut 3'
        }
    ]
    const getSinngleProduct=products.find((product)=>product.id===productID) ;

    if(getSinngleProduct)
    {
        res.json(getSinngleProduct)
    }else
    {
        res.status(400).send("product is not found try different with ID")
    }
});
```

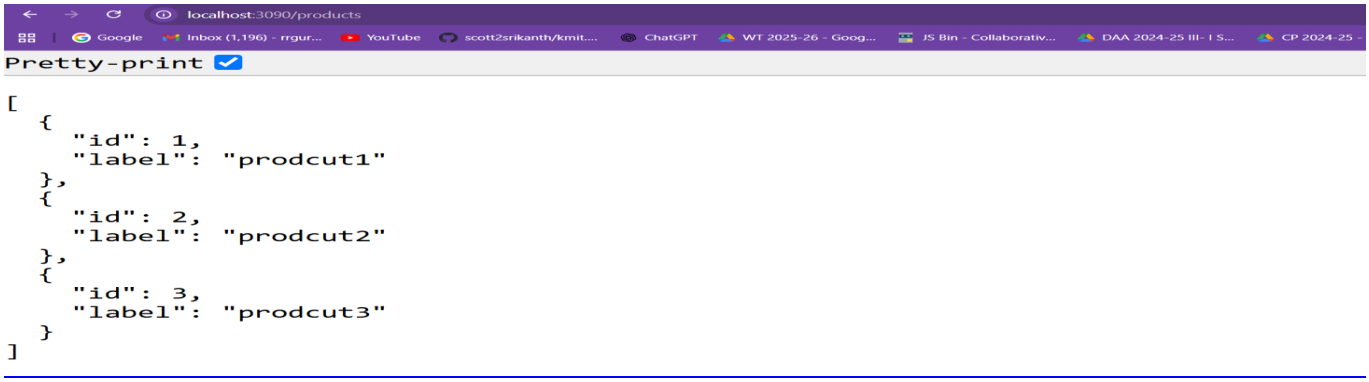
Run: **nodemorn server.js**

Terminal output:

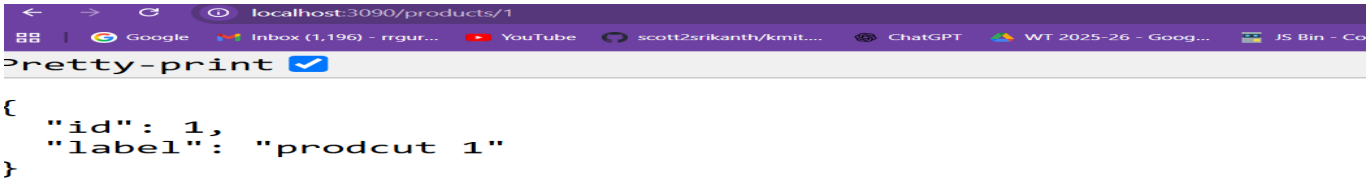
[nodemon] 3.1.10
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node index.js`
server is runningh on port number 3090
[nodemon] clean exit - waiting for changes before restart

Browser Output:

url: http://localhost:3090/products



url: http://localhost:3090/products/1



(b) Query Parameters (req.query)

Filename: app.js

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

// Route with query parameter
app.get('/search', (req, res) => {
  const keyword = req.query.q; // Access query parameter 'q'
  res.send(`You searched for: ${keyword}`);
});

const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server running at http://localhost:${PORT}`);
});
```

Explanation:

Part	Description
`/search`	URL path of the route
`req.query`	Object containing all query parameters
`req.query.q`	Access the value of the query parameter `q`
`res.send()`	Sends response back to the client

How to Give URL with Query Parameter

Syntax: url: http://localhost:3000/<route>?<parameter name>=<value>

In this example:

- Route: `/search`
- Query parameter: `q`
- Value: `nodejs`

Full URL: http://localhost:3000/search?q=nodejs

Browser Output:

You searched for: nodejs

(c) Body Parameters (req.body)

We use **POST or PUT requests** to send data to the server in the **body** of the request.
The body usually contains **JSON**, **form data**, or **other formats**.

We must use this statement : app.use(express.json());//

Explant ion:

- ✓ express.json():-Whenever someone sends data to your server (like through a POST or PUT request)
- ✓ in JSON format,this middleware converts it into a JavaScript object automatically —
- ✓ so you can easily access it using req.body.

Example -1 using postman: app.js

```
// Import Express
const express = require('express');
const app = express();

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

// POST route to receive student data
app.post('/student', (req, res) => {
  const { name, age } = req.body;  // Access JSON body
  res.send(`Received Student Name: ${name}, Age: ${age}`);
});

// Start server
const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server running at http://localhost:${PORT}`);
});
```

Explanation:

Line	Explanation
`app.use(express.json())`	Middleware that parses JSON request body and converts it into `req.body` object
`app.post('/student', (req, res) => {...})`	Defines a POST route at `/student` URL
`const { name, age } = req.body`	Destructures the data sent in request body
`res.send(...)`	Sends response back to client

1.URL: http://localhost:3000/student

2.Method: POST

3.Body (JSON)

```
{
  "name": "John",
  "age": 20
}
```

4□. Testing with Postman

1. Open **Postman**.
2. Select **POST** method.
3. Enter URL: http://localhost:3000/student
4. Go to **Body → raw → JSON** and enter:

```
{
 "name": "John",
 "age": 20
}
```

5. Click **Send**

Terminal Output: (optional): Server running at http://localhost:3000
Output in Postman / Browser: **Received Student Name: John, Age: 20**

Example-2: Testing Without Postman (Optional: Using HTML + Fetch)
File name: index.html

```
<html>
<body>
  <h2>Send Student Data</h2>
  <button onclick="sendStudent()">Send</button>
  <p id="output"></p>
  <script>
    async function sendStudent() {
      const response = await fetch('http://localhost:3000/student', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ name: 'Rakesh', age: 20 })
      });
      const text = await response.text();
      document.getElementById('output').innerText = text;
    }
  </script>
</body>
</html>
```

Clicking the button will send JSON body to **/student and display the server response.**

(d) Response Methods

Method	Description	Example
res.send()	Sends text or HTML	res.send('Hello')
res.json()	Sends JSON response	res.json({ name: 'John' })
res.status()	Sets HTTP status code	res.status(404).send('Not Found')
res.sendFile()	Sends files	res.sendFile(__dirname + '/index.html')