Project -12

“Node.js”

SUBMITTED BY:

Manaswi M. Patil

INDEX

1. Introduction

A. SUMMARY OF NODE.JS:

Node.js is an open-source, cross-platform JavaScript runtime environment that allows developers to run JavaScript code outside of a web browser. It is run on server side .(backend purpose).Node.js **uses asynchronous programming** it means Node.js is designed to execute operations in a non-blocking way.(doesn't wait to perform previous operation it run simultaneously with previous one). Node.js is the runtime environment for javascript which is run on server. Means php, flask, django ,python can run using node.js.

Synchronous it means run sequentially. Synchronous code refers to code that is executed in a sequential and blocking manner. In a synchronous programming model, each operation must be completed before the next one starts, and the program waits for each operation to finish before moving on to the next one.

Asynchronous: Node.js allows tasks to run independently, without waiting for each other to complete. This enables efficient handling of multiple operations simultaneously.

Non-Blocking I/O (Input/Output): It is run on Asynchronous non-blocking IO/ model. Instead of waiting for one operation to finish before starting another, Node.js executes I/O operations in a non-blocking manner. This means it can initiate an operation and move on to the next task without waiting for the previous one to complete.

Single thread: This allows it to efficiently manage multiple operations simultaneously by initiating tasks and continuing with other work while waiting for operations to complete.

Working of node.js:

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

Features of node.js:

1. **Asynchronous and Event-Driven:** Node.js can do many things at the same time without waiting for one operation to finish before starting another. It's like a chef cooking multiple dishes simultaneously instead of one after another.

2. **Server-Side Execution:** Node.js lets developers use JavaScript to build the "back-end" of websites or applications. It is a one type of engine behind the process that handles data, calculations, and communication with databases.
3. **Package Management (npm):** Node.js has a helpful system called npm that acts like a giant app store for code. Developers can easily find and use pre-made pieces of code (packages) to add specific features to their projects, saving time and effort.
4. **Cross-Platform:** Node.js works on different types of computer systems, just like a book that can be read on various devices like phones, tablets, and computers. This flexibility makes it easy for developers to create software that runs on Windows, macOS, or Linux.
5. **Scalability:** Node.js can handle a lot of tasks simultaneously, making it great for building apps that need to serve many users at once. It's like a traffic cop efficiently managing a busy intersection. As more cars (requests) come in, Node.js directs traffic without causing a jam.

B. WHY WE USED NODE.JS:

1. **JavaScript Everywhere:** Node.js allows developers to use JavaScript on both the client and server sides of a web application. This unification simplifies development by using a single language throughout the entire stack.
2. **Asynchronous I/O:** Node.js is designed to be non-blocking and event-driven, which makes it well-suited for handling asynchronous operations. This is particularly beneficial for handling a large number of concurrent connections without the need for threads, which can improve the overall performance of applications.
3. **Fast Execution:** Node.js is built on the V8 JavaScript runtime, developed by Google for the Chrome browser. V8 compiles JavaScript directly into machine code, resulting in fast execution. This performance is crucial for building scalable and efficient applications.
4. **Large Ecosystem (NPM):** Node Package Manager (NPM) is a vast repository of open-source libraries and modules that developers can easily integrate into their projects. This extensive ecosystem simplifies development, as developers can leverage existing packages to add functionality and features to their applications.
5. **Community Support:** Node.js has a large and active community of developers. This means that there are numerous resources, tutorials, and forums available for support. The community's engagement also results in continuous improvements and updates to the Node.js environment.
6. **Real-time Applications:** Node.js is well-suited for building real-time applications, such as chat applications and online gaming, where low latency and high concurrency are essential.

Its event-driven architecture and asynchronous capabilities make it suitable for handling multiple simultaneous connections.

7. **Cross-Platform:** Node.js is cross-platform and can be run on various operating systems like Windows, macOS, and Linux. This makes it versatile and adaptable for different deployment environments.
8. **Scalability:** Due to its non-blocking, event-driven architecture, Node.js is scalable, making it suitable for applications that need to handle a large number of concurrent connections. It can efficiently handle a high volume of requests without the need for additional resources.

C. HISTORY OF NODE.JS:

Node.js was created by Ryan Dahl and was first released in 2009. Here's a brief history:

- 2009: Ryan Dahl introduced Node.js, a runtime environment for executing JavaScript code on the server side, at the European JSConf.
- 2010: The initial stable version, Node.js 0.1.100, was released. The Node Package Manager (NPM) was introduced, becoming a crucial part of the Node.js ecosystem.
- 2011: Joyent, a cloud infrastructure company, adopted Node.js, and it gained significant attention and popularity in the developer community.
- 2012: The Node.js Foundation was established to manage the project's development and foster community collaboration.
- 2014: The io.js fork was created due to concerns about the governance of Node.js. It introduced a faster release cycle and additional features.
- 2015: Node.js and io.js merged to form the Node.js Foundation, leading to a unified and stronger development effort.
- 2016: Node.js 6, the first Long-Term Support (LTS) version under the Node.js Foundation, was released.
- 2018: The Node.js project joined forces with the JavaScript Foundation, creating the OpenJS Foundation to support and facilitate collaboration in the JavaScript ecosystem.
- 2019: Node.js 12, another LTS version, was released, bringing improvements in performance, security, and developer experience.
- 2020: Node.js 14 (LTS) introduced enhanced diagnostics and an improved JavaScript engine.
- 2021: Node.js 16 (LTS) was released with features like V8 engine upgrades, improved ECMAScript module support, and enhanced diagnostics.

Throughout its history, Node.js has grown to become a widely adopted technology, particularly in building scalable and performant server-side applications. Its asynchronous, event-driven

architecture, along with a vibrant ecosystem of libraries and modules, has contributed to its popularity in web development.

D. NODEJS VS BROWSER:

Node.js and web browsers are both environments that can execute JavaScript code, but they serve different purposes and have different capabilities.

Node.js:

1. **Server-Side Execution:** Node.js is primarily used for server-side development. It allows developers to write JavaScript code that runs on the server, handling tasks such as processing data, interacting with databases, and managing server-side logic.
2. **File System Access:** Node.js provides APIs for file system access, allowing server-side applications to read and write files, manipulate directories, and perform other file-related operations.
3. **No DOM Manipulation:** Unlike in the browser, Node.js does not have a Document Object Model (DOM) because it doesn't need to manipulate web pages. Instead, it focuses on providing a runtime environment for server-side tasks.
4. **Access to Operating System Features:** Node.js has APIs that allow interaction with the operating system, enabling tasks like spawning child processes, working with the network, and accessing the system's resources.

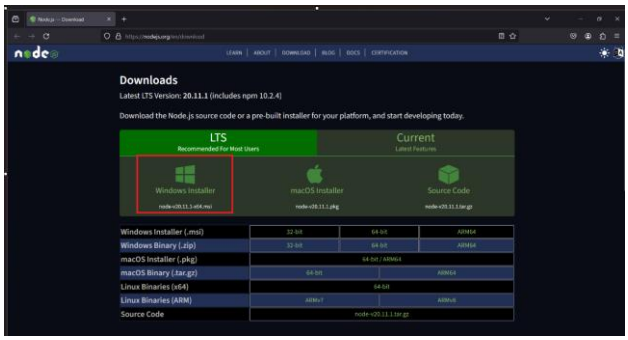
Browser:

1. **Client-Side Execution:** Web browsers execute JavaScript code on the client side, allowing developers to create dynamic and interactive user interfaces. In the browser, JavaScript interacts with the DOM to manipulate the structure and appearance of web pages.
2. **DOM Manipulation:** Browsers have a Document Object Model (DOM) that represents the structure of a web page. JavaScript in the browser can manipulate this DOM to dynamically update content, handle user events, and create responsive interfaces.
3. **Web APIs:** Browsers provide various web APIs that allow JavaScript to interact with the browser environment. These APIs include those for handling events, making AJAX requests, working with the browser's storage, and more.
4. **Graphics and Animation:** Browsers support graphics rendering and animation capabilities, allowing developers to create visually appealing and interactive web applications.
5. **User Interface Interactivity:** JavaScript in the browser is essential for creating responsive user interfaces, validating forms, handling user input, and providing a seamless browsing experience.

The two environments have different APIs and capabilities tailored to their respective roles in web development.

2. INSTALLATION

- Go on chrome →type node.js install in chrome search bar→ https://nodejs.org/en/download → choode windows installer→go ahead to next process.



- Open vs code editor→first create folder in your desktop files→open with vs code →run your all project files in it→ create index.js file→run with command node (file name).

Example:

```
JS index.js > ...
1   const mau = require("./second")
2   console.log('hello words')
3
```

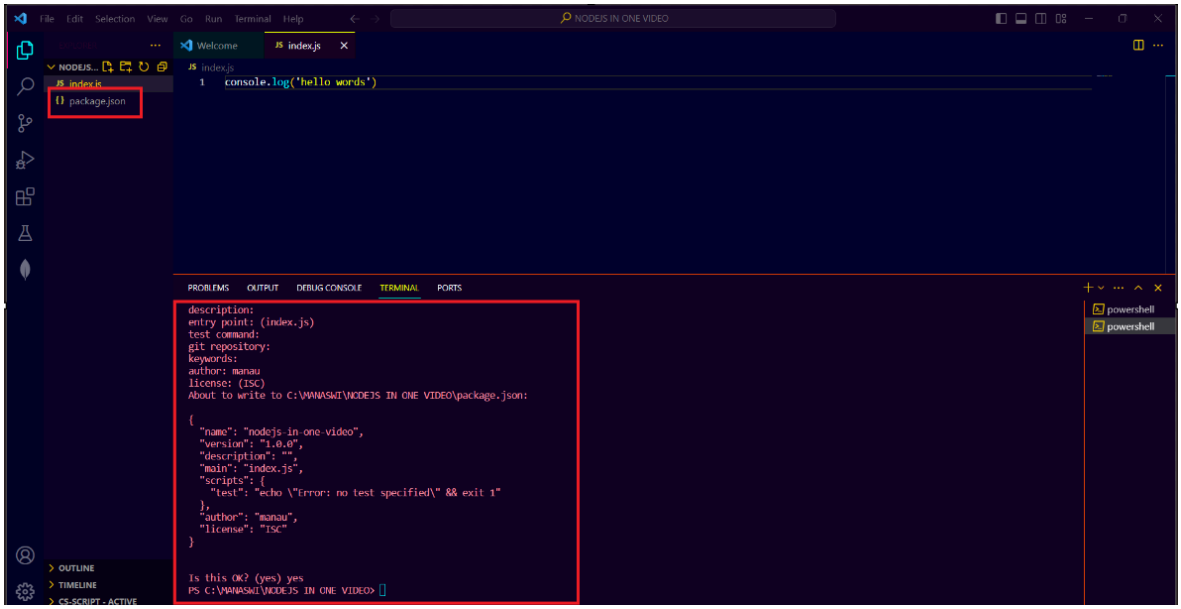
```
} C:\MANASWI\NODEJS IN ONE VIDEO\second.js C:\MANASWI\NODEJS IN ONE VIDEO
hello words
PS C:\MANASWI\NODEJS IN ONE VIDEO>
```

- To check node.js version type command as **"node –version"**.
- Node packages manger "**npm"** →manages node packages → A **package in Node.js contains all the files you need for a module**.
- **Modules are JavaScript libraries** you can include in your project(files which you created known as modules).

```
PS C:\MANASWI\NODEJS IN ONE VIDEO> npm --version
9.5.1
PS C:\MANASWI\NODEJS IN ONE VIDEO> node --version
v18.16.1
PS C:\MANASWI\NODEJS IN ONE VIDEO>
```

- **Npm init command:** The npm init command is used to initialize a new Node.js project and create a '**package.json file'**. The package.json file contains metadata about the project, such as its name, version, description, entry point (main file), test command, repository, dependencies, and other configuration details. after entering this command it will ask you name ,author all details to users.

```
PS C:\MANASWI\NODEJS IN ONE VIDEO> npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (nodejs-in-one-video)
```



- Express.js and angular.js are the packages of node.js→to install packages type **npm install** or used shortcut **npm i or go on express.js website →type in terminal 'npm install express --save'**→run **'npm fund'** for more details→In previous we need to add in dependencies so we used that **'- - save'** to install add package.
- **npm i**→creates **'node modules'** folder in our previous files→it contains all dependencies which are came through internet.
- We can install package globally using command→"**npm i -g nodemon**"→now we can used nodemon command.
- **Nodemon: It will shows the live changes which you made in your code and display live output.** changes between the code which you made in code and display fresh output. →'**node index'** to see output. nodemon is a tool that helps in the development of Node.js applications by monitoring for any changes in the source code and automatically restarting the server.

Node.js http modules:

- createServer creates an HTTP server, and the callback function is executed whenever a request is received.
- res is the response object, and req is the request object.
- writeHead sets the response header, and end sends the response.
- It is built on the V8 JavaScript runtime engine, which is also used by the Chrome web browser.

Dev dependencies:'npm install –save-dev nodemon'. n Node.js and npm (Node Package Manager), dependencies in a project can be categorized into two main types: regular dependencies and development dependencies.

Regular Dependencies: These are the dependencies that are required for the application to run in a production environment. They are typically modules or libraries that your application relies on to function correctly.These dependencies are listed in the dependencies section of your package.json file.Examples include web frameworks (like Express.js), database drivers, utility libraries, etc.

Development Dependencies*:* These are dependencies that are only needed during the development phase of the project. They include tools, libraries, or testing frameworks that help with development and testing but are not required for the deployed application to run. Development dependencies are listed in the devDependencies section of your package.json file. Examples include testing libraries (like Mocha or Jest), linters, transpilers, and tools like nodemon for development server auto-restart.

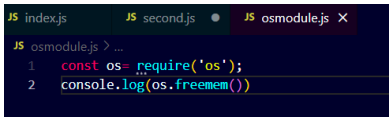
To uninstall: "npm uninstall nodemon".

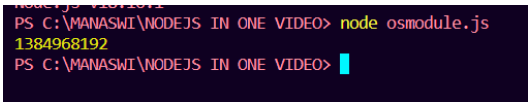
- npm i express→ 'package lock json' this file contain all dependencies trees.

Some important functions:

Search→node.js doc→go →Some important functions.

I. **Create file as 'osmodule.js' in vs code .**



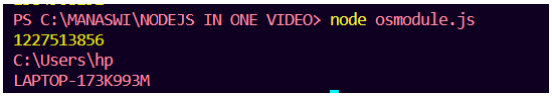


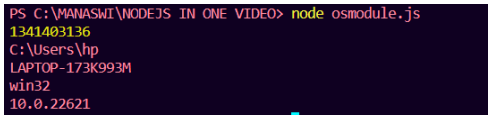
i. **os.Homedir()**

ii. **Os.Hostname()**

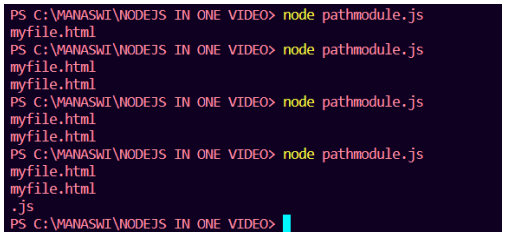
iii. **os.platform()**

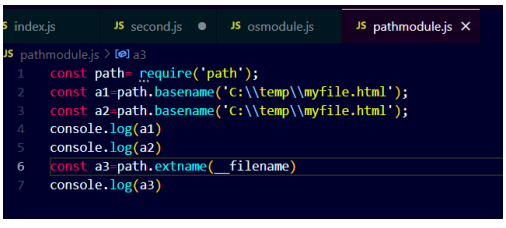
iv. **Os.release()**





II. **Create file as 'pathmodule.js':**

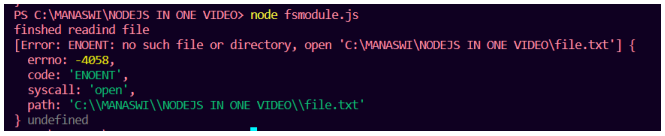




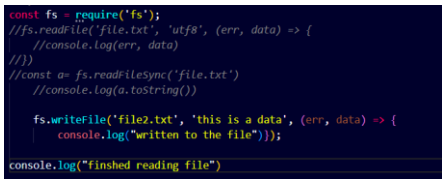
III. **File system:**

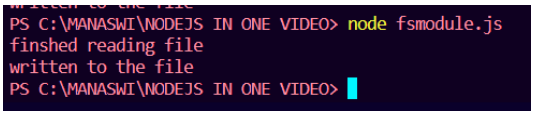
This below file throw error because it doesn't valid.

readFile



WriteFile:





Node.js File System: the Node.js File System (fs) module is a built-in module that allows you to work with the file system on your computer. It provides methods to performing various operations, such as reading from and writing to files, creating and deleting files and directories.

Require the module: To use the file system module, you need to include it in your Node.js script. You do this by using the require function:

Reading a File: You can use the fs.readFile method to read the contents of a file. It takes the file path and a callback function as parameters:

Writing to a File: To write data to a file, you can use the fs.writeFile method:

Creating and Deleting Directories: The fs.mkdir method is used to create a new directory, and fs.rmdir is used to remove a directory.

3. MODULES

Module creation: Each file is treated as a module.

require() Function: To include functionality from another module, you use the **require()** function.

There are two main types of modules :

I. Common js module:

- Extension is ".js"
- Syntax : `const mau = require("./second")`
- **Const declaration variable or word = require("./file name which u want to importing")** ./ used for relative path.
- Common.JS uses require() for importing modules and module.exports for exporting.

II. ECMAScript (ES) modules:

Extension is **" .mjs"**

```
export  function add(a,b){
    return a+b

}
```

Export function declaration like variable or word() {create array with curly braces}then console.log("content what u want to print")→it eill done in mjs file→in main file where u want to import it in that file you will code like → `import {mau } from './test.mjs'` declaration word in curly braces if you had used default in .mjs file then no need variable declaration in curly braces .

Example:

```
JS test.mjs > ...
1  export  function add(a,b){
2      return a+b
3  }
4  console.log(add(3,5))
5
6  export default function mul(a,b){
7      return a*b
8
9  }
10 console.log(mul(3,5))
```

```
JS test2.js
1  import {mau} from './test.mjs'
2  // add()
3  // mul()
4  console.log (mau)
```

```
{} package.json > ...
1  {
2    "name": "nodejs-in-one-video",
3    "version": "1.0.0",
4    "description": "",
5    "type":"module",
6    "main": "index.js",
     Debug
7    "scripts": {
8      "test": "echo \"Error: no test specified\" && exit 1"
9    },
10   "author": "manau",
11   "license": "ISC"
12 }
```

➢ If you used **"* as**" it will show all details in your file.

```
JS test2.js
1   import * as mau  from './test.mjs'
2   //add()
3   //mul()
4   console.log (mau)
```

```
[Running] node "c:\MANASWI\NODEJS IN ONE VIDEO\test2.js"
8
15
[Module: null prototype] {
  add: [Function: add],
  default: [Function: mul]
}

[Done] exited with code=0 in 0.19 seconds
```

Creating Modules:

To create a module, you simply create a new JavaScript file. For example, you can create a file named '**myModule.js.**'

```
const myVariable = 'Hello, World!';

function myFunction() {
  console.log('This is my function.');
}

module.exports = { myVariable, myFunction };
```

Exporting from Modules: The '**module.exports**' or **exports** object is used to make variables, functions, or objects available for other modules to use.

To use the exported functionality in another module, you use the require function.

Importing Modules: To use the exported functionality in another module, you use the '**require function.**'

Core modules: Node.js has several built-in core modules that provide essential functionality. These modules can be imported using require**. "const fs = require('fs');"**

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

III. Third-Party Modules:

- **Express:** A popular web framework for building web applications and APIs. It is used in the process of handling routes, requests, and responses.
- **Mongoose**: An ODM (Object Data Modeling) library for MongoDB, making it easier to interact with MongoDB databases.
- **Socket.io:** Enables real-time bidirectional communication between clients and servers using WebSockets.
- **Customs hooks:** This helps in organizing and reusing pieces of code. consisder, you have a set of tasks you need to do every time you process some data. Instead of copying and pasting the same code everywhere, you can create a "hook" function that contains that code. Then, whenever you want to perform those tasks, you just call your hook, and it takes care of everything for you.

4. Working with files

- **Module wrapper function:** In Node.js, every module (JavaScript file) is wrapped in a function called the "module wrapper function" to provide a private scope for the module's code. Every Node.js module is wrapped by this function to create a private scope and provide certain variables and functions.

```
(function(exports, require, module, __filename, __dirname){
```

  - **exports**: It is an object that is initially empty and is used to define exports from the module. Anything assigned to exports will be accessible from outside the module when it is required in other files.
  - **require**: It's a function used to include other modules or files within the current module.
  - **module*:*** It is an object that represents the current module. It includes information about the module, and you can use it to export values or change the behavior of the module.
  - **__filename**: It is a string representing the absolute path of the file containing the code.
  - **__dirname*:*** It is a string representing the absolute path of the directory containing the module.

The module wrapper function ensures that variables and functions declared within a module are not automatically accessible from the global scope.

5. NPM(Node Package Manager)

Installing packages: global installation & local installation

Local Installation: npm install packageName →This installs the package in the **node_modules** directory of your current project. A locally installed package is specific to a particular project or directory. It is typically installed within the project's directory structure.

Global Installation: npm install -g packageName → This installs the package globally, making its commands available in the terminal. A globally installed package is available system-wide, meaning it can be used by any project on the computer.

Creating packages: npm init →This command guides you through creating a **package.json** file with information about your project.

Updating packages: npm update→This updates the packages in your **node_modules** directory according to the version constraints defined in your **package.json**

Using installed packages: After installing a package, you can use it in your Node.js code. For example, if you install the **lodash** library:

```
// Importing lodash in your Node.js code
const _ = require('lodash');

// Using lodash functions
const result = _.sum([1, 2, 3, 4, 5]);
console.log(result);  // Output: 15
```

Running scripts: npm start

Npm workspaces: NPM Workspaces allow you to manage multiple packages within a single top-level, root package.-->Create a root folder for your workspace→Inside the root folder, create a **package.json** file with **"private": true** and **"workspaces": ["packages/*"]**. Keep in array i.e {}.

6. Error handling

JavaScript uses try...catch blocks for handling errors. This is useful for synchronous code.

In try block synchronous code will handle.

```
index.html ● JS script.js ×
Users > hp > JS script.js > ...
1  setTimeout(()=>{
2      console.log("hacking Wifi...please Wait...")
3  },1000)
4  try{
5      console.log(mau)
6  }
7  catch(error){
8      console.log(error)
9  }
10 setTimeout(()=>{
11     console.log("fetchning username and password...please Wait...")
12 },2000)
13
14
15 setTimeout(()=>{
16     console.log("mau's id...please wait")
17 },3000)
18
19 setTimeout(()=>{
20     console.log("fetchning username and password of mau's(+235647453)fetched..please Wait...")
21 },4000)
22
```

```
ReferenceError: mau is not defined
    at Object.<anonymous> (c:\Users\hp\script.js:5:17)
    at Module._compile (node:internal/modules/cjs/loader:1256:14)
    at Module._extensions..js (node:internal/modules/cjs/loader:1310:10)
    at Module.load (node:internal/modules/cjs/loader:1119:32)
    at Module._load (node:internal/modules/cjs/loader:960:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:81:12)
    at node:internal/main/run_main_module:23:47
hacking Wifi...please Wait...
fetchning username and password...please Wait...
mau's id...please wait
fetchning username and password of mau's(+235647453)fetched..please Wait...
```

Using another try block within try block

```
index.html ● JS script.js ●
Users > hp > JS script.js > setTimeout() callback
1  setTimeout(()=>{
2      console.log("hacking Wifi...please Wait...")
3  },1000)
4  try{
5      setTimeout(()=>{
6          try{
7              console.log(mau)
8          }catch(error){
9  }
10             },100)
11 }
12 catch(error){ ···
14 }
15
16 setTimeout(()=>{ ···
18 },2000)
19
20
21 setTimeout(()=>{ ···
23 },3000)
24
25 setTimeout(()=>{ ···
27 },4000)
```

```
[Running] node "c:\Users\hp\script.js"
hacking Wifi...please Wait...
fetchning username and password...please Wait...
mau's id...please wait
fetchning username and password of mau's(+235647453)fetched..please Wait...
```

Call stack and stack trace : The call stack is a mechanism used to keep track of function calls in a program. When an error occurs.

```
C: > Users > hp > JS callstack.js > ...
Codeium: Refactor | Explain | Generate JSDoc | × | Codiumate: Options | Test this function
function functionA() {
    console.log("Function A");
    functionB();
}

Codeium: Refactor | Explain | Generate JSDoc | × | Codiumate: Options | Test this function
function functionB() {
    console.log("Function B");
    functionC();
}

Codeium: Refactor | Explain | Generate JSDoc | × | Codiumate: Options | Test this function
function functionC() {
    console.log("Function C");
    throw new Error("An error occurred in Function C");
}

// Trigger the functions
functionA();
```

```
[Running] node "c:\Users\hp\callstack.js"
Function A
Function B
Function C
c:\Users\hp\callstack.js:13
    throw new Error("An error occurred in Function C");
    ^

Error: An error occurred in Function C
    at functionC (c:\Users\hp\callstack.js:13:11)
    at functionB (c:\Users\hp\callstack.js:8:5)
    at functionA (c:\Users\hp\callstack.js:3:5)
    at Object.<anonymous> (c:\Users\hp\callstack.js:17:3)
    at Module._compile (node:internal/modules/cjs/loader:1256:14)
    at Module._extensions..js (node:internal/modules/cjs/loader:1310:10)
    at Module.load (node:internal/modules/cjs/loader:1119:32)
    at Module._load (node:internal/modules/cjs/loader:960:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:81:12)
    at node:internal/main/run_main_module:23:47
```

The stack trace is a detailed report of the active functions and their order at the time the error occurred. It lists the functions that were called, in order, leading up to the point where error is occurred.

```
C: > Users > hp > JS calltrace.js > ...
Codeium: Refactor | Explain | Generate JSDoc | × | Codiumate: Options | Test this function
function functionA() {
    console.log("Function A");
    functionB();
}

Codeium: Refactor | Explain | Generate JSDoc | × | Codiumate: Options | Test this function
function functionB() {
    console.log("Function B");
    functionC();
}

Codeium: Refactor | Explain | Generate JSDoc | × | Codiumate: Options | Test this function
function functionC() {
    console.log("Function C");
    try {
        throw new Error("An error occurred in Function C");
    } catch (error) {
        console.error(error.stack);
    }
}

// Trigger the functions
functionA();
```

```
[Running] node "c:\Users\hp\calltrace.js"
Function A
Function B
Function C
Error: An error occurred in Function C
    at functionC (c:\Users\hp\calltrace.js:14:13)
    at functionB (c:\Users\hp\calltrace.js:8:5)
    at functionA (c:\Users\hp\calltrace.js:3:5)
    at Object.<anonymous> (c:\Users\hp\calltrace.js:21:3)
    at Module._compile (node:internal/modules/cjs/loader:1256:14)
    at Module._extensions..js (node:internal/modules/cjs/loader:1310:10)
    at Module.load (node:internal/modules/cjs/loader:1119:32)
    at Module._load (node:internal/modules/cjs/loader:960:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:81:12)
    at node:internal/main/run_main_module:23:47
```

Debugging node.js: Insert the **debugger;** statement at the point in your code where you want to start debugging. Node.js has a built-in debugger and supports debugging with external tools. You can start a Node.js script in debugging mode using→node inspect yourScript.js →or **node --inspect yourScript.js** → You can then connect to the debugging session using Chrome DevTools or use the --inspect-brk flag for breakpoints.

Uncaught exceptions : An uncaught exception occurs when an error is thrown, but there is no surrounding try...catch block to handle it. Node.js will terminate the process if an unhandled exception occurs. "Uncaught exceptions" refer to errors in a computer program that occur during runtime but are not properly handled by the program.

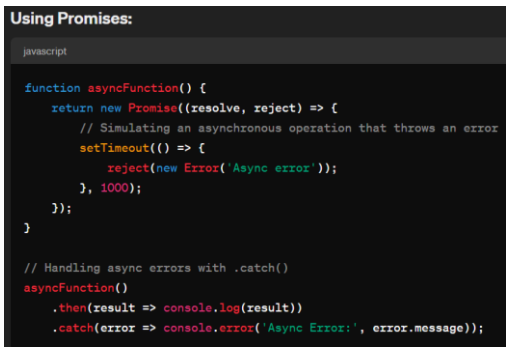
When an uncaught exception occurs:

1. **Termination:** The program may terminate abruptly.
2. **Error Message:** An error message is usually displayed, providing information about the type of exception and the location where it occurred.
3. **Default Handling:** In some environments, there might be default handlers that log or display information about the uncaught exception.

```
process.on('uncaughtException', (error) => {
    console.error('Uncaught Exception:', error.message);
    // Perform cleanup or other necessary actions
    process.exit(1);
});

// Triggering an uncaught exception
throw new Error('This is an uncaught exception');
```

Handling async errors:

Handling asynchronous errors in Node.js often involves using promises and the .catch() method or try...catch blocks in asynchronous functions.



Types of errors:

1. **JavaScript Errors:** JavaScript errors are those that are explicitly thrown by the JavaScript runtime or your code. These include the built-in **Error** object and its subclasses, a custom JavaScript error is thrown using the **throw** statement.
2. **System errors:** System errors are related to operations at the system level, and they may be thrown by Node.js itself or by modules that interact with the underlying system. **readFileSync** operation tries to read a file that doesn't exist, resulting in a system error.
3. **User-Specified Errors:** User-specified errors are custom errors created by developers to represent specific conditions or states in their application. Developers can create custom error classes by extending the built-in **Error** object. a custom error class **CustomError** is defined, and an instance of it is thrown with a custom message.
4. **Assertion Errors:** Assertion errors occur when an **assert** statement fails. Assertions are checks placed in the code to ensure that certain conditions are met. When an assertion fails, an assertion error is thrown. The **assert.strictEqual** statement fails, triggering an assertion error with a custom message.

7. Asynchronous programming

Event loop: The event loop is a fundamental concept in asynchronous programming that allows Node.js to handle multiple operations concurrently without blocking the execution of the program. It enables non-blocking I/O operations and supports an event-driven architecture.

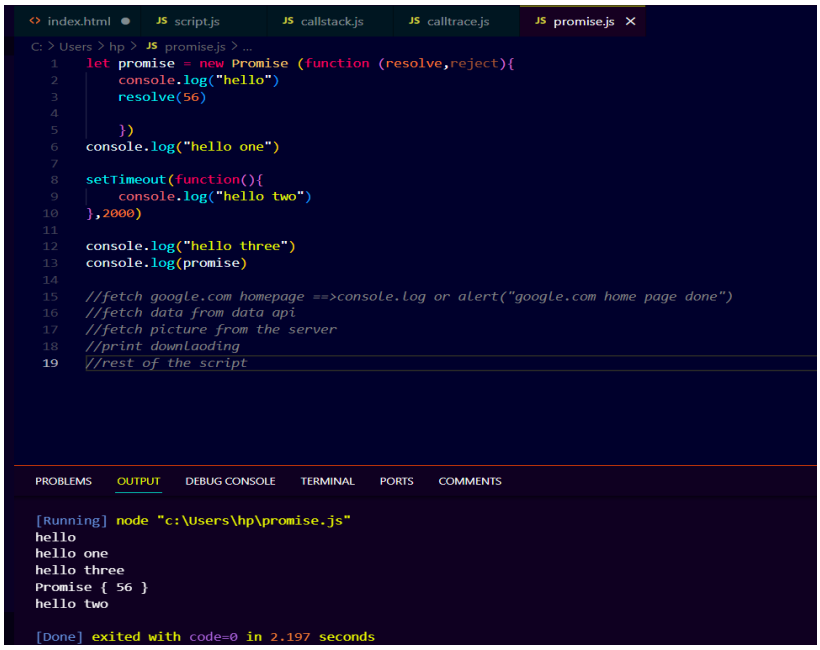
How the Event Loop Works:

- **Event Queue:** Asynchronous operations (such as I/O operations or timers) are executed in the background, and when they complete, their callback functions are added to the event queue.
- **Event Loop:** The event loop constantly checks the event queue. When the call stack is empty, it takes the first callback from the queue and pushes it onto the call stack for execution.
- **Callbacks:** Callback functions are used to handle the results of asynchronous operations. They are executed when the corresponding operation completes.
- **Two Most important methods .**then() and .catch()

Event emitter: The EventEmitter is a core module in Node.js that provides an implementation of the observer pattern. It allows objects to emit and handle events, an event-driven architecture.

Writing asynchronous code:

1.Promises: Promises used for parallel execution. are a mechanism for handling asynchronous operations in JavaScript. A promise represents a value that might be available now, or in the future, or never.



```
//fetch google.com homepage ==>console.log or alert("google.com home page done")
//fetch data from data api
//fetch picture from the server
//print downlaoding
//rest of the script
```

2. Async/Await: is a syntax for writing asynchronous code that looks synchronous. It is built on top of promises.

3. Callbacks: Callbacks are functions passed as arguments to another function, to be executed later when an asynchronous operation completes.

4. setTimeout: is a function that schedules a function or code snippet to be executed after a specified delay.

5. setInterval: setInterval is a function that repeatedly executes a function or code snippet at a specified interval.

6. setImmediate: setImmediate is a Node.js-specific function that schedules a function to be executed in the next iteration of the event loop.

7. process.nextTick: process.nextTick is a function that schedules a callback to be invoked on the next iteration of the event loop.

8. Command Line Apps

Environment variables: The dotenv package allows you to load environment variables from a .env file into process.env.

```
// Install dotenv package: npm install dotenv
require('dotenv').config();

// Access environment variables
console.log(process.env.API_KEY);
```

Command line args: Command-line arguments are parameters passed to a command-line application when it is executed.

```
// Access command-line arguments
const args = process.argv.slice(2);
console.log('Command-line arguments:', args);
```

commander.js provides a simple and expressive way to define and parse command-line options.

```
// Install commander package: npm install commander
const { program } = require('commander');

program
  .option('-p, --port <port>', 'Set the server port')
  .option('-H, --host <host>', 'Set the server host')
  .parse(process.argv);

console.log('Port:', program.port);
console.log('Host:', program.host);
```

Exiting and exit codes: Exiting a Node.js process can be done using process.exit(). Exit codes can be used to indicate the success or failure of the process.

Taking input:

i. process.stdin:

process.stdin is a readable stream that represents the standard input (stdin) in a Node.js application.

It allows you to read data entered by the user in the command-line interface.

Working in Node.js→Here's a basic example of using **process.stdin**.

This example prompts the user to enter their name and then displays a personalized greeting.

ii. prompts Package:

- **prompts** is a lightweight and versatile package for prompting users in a Node.js command-line application.
- It provides a simple API for creating prompts and validating user input.

- **Working in Node.js→**First, install the **prompts** package→npm install prompts →Then, use it in your code.
- This example prompts the user for their name using the **prompts** library and then displays a personalized greeting.

iii. inquirer Package:

- **inquirer** is a powerful and widely-used package for creating interactive command-line interfaces in Node.js.
- It supports a variety of prompt types and provides extensive customization options.
- **Working in Node.js→**First, install the **inquirer** package→npm install inquirer →Then, use it in your code.
- Both **prompts** and **inquirer** provide a wide range of prompt types, such as text input, password input, multiple-choice, etc.

Printing output:

i. **process.stdout→ process.stdout** is a writable stream in Node.js that represents the standard output (stdout) stream. It is used for writing regular output to the console.

process.stderr: process.stderr is a writable stream in Node.js that represents the standard error (stderr) stream. It is used for writing error messages and other critical information.

ii. chalk Package:

- **chalk** is a popular library for styling console output with colors.
- It allows you to add color to your console logs and make them more visually appealing.
- **Working in Node.js→** First, install the **chalk** package→ **npm install chalk→**Then, use it in your code.
- **chalk** provides various methods for styling text with colors, such as **chalk.blue**, **chalk.red**, etc.
- You wrap the text you want to style with these methods.

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

// Styling console output
console.log(chalk.blue('This is a blue message.'));
console.error(chalk.red('This is an error message.'));
```

iii. figlet Package:

- **figlet** is a package for creating ASCII art text using different fonts.
- It can be used to generate stylized text that can be displayed in the console.
- **Working in Node.js**→First, install the **figlet** package→npm install figlet→Then, use it in your code.
- **figlet** takes a text string and converts it into ASCII art.
- It provides different fonts for styling the ASCII art.

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

// Creating ASCII art
figlet('Hello Figlet!', (err, data) => {
  if (err) {
    console.error('Error:', err);
    return;
  }
  console.log(data);
});
```

iv. cli-progress Package:

- **cli-progress** is a library for creating progress bars in the console.
- It's useful when you want to show the progress of a long-running operation.
- **Working in Node.js**→ First, install the **cli-progress** package→npm install cli-progress →then, use it in your code:
- **cli-progress** provides a **SingleBar** class for creating a single-line progress bar.
- You start the progress bar with the total number of steps and update it as the task progresses.
- Prints the data to ***standard output*** using **console.log()**.
- Prints an error message to standard ***error using*** **console.error()**.

```
const { SingleBar } = require('cli-progress');

// Creating a progress bar
const progressBar = new SingleBar({}, cliProgress.Presets.shades_classic);
progressBar.start(100, 0);

// Simulating a task with progress
for (let i = 0; i <= 100; i++) {
  // Some task...
  progressBar.update(i);
}

progressBar.stop();
```

9. Working with APIS

A. Authentication:

Authentication is the process of verifying the identity of a user or system. In Node.js, authentication is often implemented in web applications, APIs, or other networked systems.

How it Works in Node.js:

- User Credentials: Users provide credentials (such as username and password) to prove their identity.
- Server Validation: The server validates the credentials against stored data (e.g., in a database).
- Session Management:Upon successful validation, the server creates a session or issues a token to represent the authenticated user.This session or token is used to maintain the user's identity during subsequent requests.
- Middleware: Middleware in Node.js frameworks like Express can be used for authentication. Middleware functions intercept incoming requests, verify authentication, and pass control to the next middleware or route handler.

1. **Jsonwebtoken Package**: "**const jwt = require('jsonwebtoken');**"
   - The jsonwebtoken package is a widely-used library for creating and verifying JSON Web Tokens (JWT) in Node.js applications.
   - JWT is a compact, URL-safe means of representing claims to be transferred between two parties.
   - Working in Node.js→First, install the jsonwebtoken package→**npm install jsonwebtoken.**
   - The jwt.sign method creates a JWT with a specified payload and a secret key.
   - The jwt.verify method verifies the JWT using the same secret key and returns the decoded payload if successful.

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

// Creating a JWT
const payload = { userId: 123 };
const secretKey = 'your-secret-key';
const token = jwt.sign(payload, secretKey, { expiresIn: '1h' });

console.log('JWT:', token);

// Verifying a JWT
jwt.verify(token, secretKey, (err, decoded) => {
  if (err) {
    console.error('JWT verification failed:', err.message);
  } else {
    console.log('Decoded JWT:', decoded);
  }
});
```

2. **<u>Passport.js Package:</u> const passport = require('passport'); and const LocalStrategy = require('passport-local').Strategy;**
   - can be used for various authentication strategies. Passport.js is configured with a local strategy, where it checks if the provided username and password match a predefined set. If the authentication is successful, it returns user information; otherwise, it returns an error.
   - Passport.js is a popular authentication middleware for Node.js. It provides a flexible and modular way to authenticate users in web applications.
   - It supports various authentication strategies, including local authentication, OAuth, and more.
   - Working in Node.js→First, install the passport package and any necessary authentication strategy (e.g., passport-local for local authentication)→npm install passport passport-local
   - Passport is configured to use the LocalStrategy for username/password authentication.
   - Express sessions are used to persist login state.
   - Routes are protected, and authentication is enforced using passport.authenticate.
   - serializeUser and deserializeUser functions manage user session serialization and deserialization.
   - Both jsonwebtoken and Passport.js are crucial in securing and managing authentication in Node.js applications. jsonwebtoken is specifically focused on token-based authentication using JWT, while Passport.js provides a broader set of tools for implementing authentication strategies in web applications

B. HTTP Server:

<u>Fastify Framework</u>: **"const fastify = require('fastify')();"** Fastify is used to create a simple HTTP server. A route is defined for the root path ('/') that sends a JSON response. The server listens on port 3000.

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

fastify.get('/', (request, reply) => {
  reply.send({ message: 'Hello, Fastify!' });
});

fastify.listen(3000, (err, address) => {
  if (err) throw err;
  console.log(`Server listening on ${address}`);
});
```

Fastify is a web framework for Node.js that is designed to be fast, efficient, and extensible. Let's explore how Fastify works in a Node.js application:

Initialization: To use Fastify, you start by installing it using npm or yarn: **npm install fastify**.

After installation, you import Fastify into your Node.js application:

Routing:

- Fastify uses a routing system to handle different HTTP methods and URL paths.
- Routes are defined by specifying the HTTP method and URL pattern:

Request and Response Objects:

- When a request is received, Fastify creates a **request** object and a **reply** object.
- The **request** object contains information about the incoming request (headers, parameters, query parameters).
- The **reply** object is used to construct and send the response back to the client.

Middleware:

- Fastify supports middleware functions that can be executed before or after handling a route.
- Middleware functions have access to the **request** and **reply** objects, allowing you to modify or augment the request or response:

Schema Validation:

- Fastify allows you to define the expected shape of incoming request data using JSON Schema.
- It automatically validates incoming requests against the specified schema:

Plugins:

- Fastify is highly extensible through plugins.
- You can use built-in plugins or create your own to add features like authentication, logging, and more.

Dependency Injection:

- Fastify uses a dependency injection system, allowing you to inject services and dependencies into your routes and plugins.

Lifecycle Hooks:

- Fastify provides lifecycle hooks that allow you to run code at specific points in the application's lifecycle:

Server Start:

- To start the Fastify server and listen for incoming requests, you call the **listen** method:

NestJS Framework: NestJS is used to define a basic controller with a route for the root path ('/'). The controller responds with the string 'Hello, NestJS!'. NestJS is a web framework for building scalable and maintainable server-side applications with

Node.js. It brings a set of architectural patterns and features to the table that make it suitable for a variety of projects.

```
import { Controller, Get } from '@nestjs/common';

@Controller()
export class AppController {
  @Get()
  getHello(): string {
    return 'Hello, NestJS!';
  }
}
```

1. **Modular and Organized Structure:**
   - NestJS encourages a modular and organized project structure, dividing your application into modules, each containing its own controllers, services, and other related components.
   - This structure helps in organizing and scaling your codebase, making it easier to understand and maintain.
2. **TypeScript Support:**
   - NestJS is built with TypeScript, a superset of JavaScript that adds static typing to the language.
   - TypeScript brings benefits like improved code quality, better tooling support, and enhanced developer productivity by catching potential errors during development.
3. **Decorator-based Programming:**
   - NestJS leverages decorators for defining modules, controllers, services, and other components. Decorators provide a declarative way to attach metadata to classes and functions, enhancing readability and maintainability.
4. **Dependency Injection:**
   - NestJS uses a powerful dependency injection system, making it easy to manage and inject dependencies into controllers, services, and other components.
   - Dependency injection helps in writing modular and testable code, promoting the separation of concerns.
5. **Expressive Routing:**
   - NestJS provides a powerful routing system that allows you to define routes with controllers and methods in a declarative manner.
   - The routing system supports various HTTP methods, parameters, query parameters, and route guards.
6. **Middleware Support:**

- NestJS supports middleware functions that can be executed before or after handling a request.
- Middleware functions have access to the request and response objects, allowing you to perform tasks such as authentication, logging, and request manipulation.

7. **Exception Handling:**
   - NestJS provides a robust exception handling mechanism. You can use exception filters to catch and handle exceptions globally or at a more granular level.
   - This helps in centralizing error handling logic and providing consistent error responses.
8. **Built-in Validation:**
   - NestJS integrates with class-validator and class-transformer libraries, allowing you to perform automatic request validation based on decorators.
   - You can define validation rules using decorators on your DTOs (Data Transfer Objects) to ensure that incoming data meets specific criteria.
9. **Interceptors:**
   - Interceptors allow you to intercept the execution flow of a request before it reaches the controller or after it leaves the controller.
   - Interceptors are useful for tasks such as logging, caching, and modifying the response.
10. **Testing Support:**
    - NestJS provides a testing module and utilities for unit testing your application.
    - You can easily create and run unit tests for controllers, services, and other components, ensuring the reliability of your codebase.
11. **WebSocket Support:**
    - NestJS has built-in support for handling WebSocket communication, making it easy to implement real-time features in your applications.
    - It provides decorators for WebSocket events, allowing you to handle connections, disconnections, and messages.
12. **Extensibility through Middleware and Guards:**
    - NestJS is highly extensible. You can create custom middleware functions and guards to add functionality such as authentication, authorization, and request processing.
13. **Community and Ecosystem:**
    - NestJS has a growing and active community, which means you can find a wealth of resources, tutorials, and third-party packages.
    - The ecosystem includes plugins, modules, and tools that can be integrated into your NestJS applications.

Express.js Framework: Express.js is a widely-used, minimalist framework that is great for building APIs and web applications. It provides a straightforward and flexible structure, making it easy to set up routes, middleware, and handle HTTP requests. It's a good choice for various types of web projects.Express.js is used to create an HTTP server with a route for the root path ('/') that responds with the string 'Hello, Express!'. The server listens on port 3000.
Express.js is a web application framework for Node.js that simplifies the development of web applications by providing a set of features and tools. Here's a concise overview of key functions.

1. **Routing:**
   - Defines how the application responds to different HTTP requests and URL patterns.
   - Uses HTTP methods (GET, POST, etc.) and URL patterns to create routes.
2. **Middleware:**
   - Functions executed during the request-response cycle.
   - Handle tasks like logging, authentication, and data parsing.
   - Use **app.use()** to apply middleware globally or locally to specific routes.
3. **Request and Response Objects:**
   - **req** (request) object represents the HTTP request.
   - **res** (response) object represents the server's response to the client.
4. **Route Parameters:**
   - Captures values from the URL to be used in route handlers.
   - Defined in route patterns using a colon syntax (**:parameter**).
5. **Static File Serving:**
   - Serves static files (images, stylesheets) using **express.static** middleware.
   - Allows easy integration of client-side assets.
6. **Template Engines (Optional):**
   - Supports various template engines like EJS or Pug for generating dynamic HTML content.
   - Enables embedding variables and logic in HTML templates.
7. **Error Handling:**
   - Defines middleware for handling errors during the request-response cycle.
   - Uses a special middleware function with four parameters (err, req, res, next).
8. **Server Configuration and Listening:**
   - Configures the server by specifying the port and other settings.
   - Starts the server using the **listen** method.

9. **Community and Ecosystem:**
   - Express has a large and active community, resulting in a wealth of tutorials, documentation, and third-party modules.
   - A rich ecosystem of middleware and extensions is available, providing solutions for common web development challenges.
10. **Performance:**

- Express is known for its performance and efficiency.
- It is lightweight and designed to handle a large number of concurrent requests, making it suitable for building scalable applications.

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

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

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

- **HTTP Module:** The HTTP module is part of the Node.js core, and it's used to create simple HTTP servers. It's suitable for basic server functionalities when a lightweight solution is needed and is often employed for quick prototypes or small-scale applications. The **http** module in Node.js is a core module that provides functionality for creating HTTP servers and making HTTP requests.
- The **http** module allows developers to create HTTP servers with ease. It provides the **createServer** method, enabling the handling of incoming HTTP requests.
- The **http** module can be used to make HTTP requests to other servers. The **request** method allows developers to send HTTP requests and handle the responses.
- `const http = require('http');`

C. Making API Calls:

1. **HTTP Module**: The HTTP module is utilized when making HTTP requests within a Node.js application. It's a low-level module suitable for basic scenarios where simplicity is key. However, for more advanced features and better ease of use, developers often turn to external libraries like Axios, Unfetch, or Got.

   The http module is a core module in Node.js for handling HTTP-related tasks. It allows you to create HTTP servers (createServer) and make HTTP requests (http.request).
2. **Axios Package**: `npm install axios` ,Axios is widely used for making HTTP requests in both browser and Node.js environments. It supports features like promise-based requests, request/response transformation, and automatic handling of request and response headers. It's often chosen for its simplicity and ease of integration.The response data is logged on success, and any errors are handled.

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

axios.get('https://example.com/api/data')
  .then((response) => {
    console.log(response.data);
  })
  .catch((error) => {
    console.error(error.message);
  });
```

3. **Unfetch Package**: **npm install unfetch** ,Unfetch is a lightweight library mainly used for making HTTP requests in *browser environments.* It's suitable for scenarios where a minimalistic approach is preferred, and dependencies need to be kept to a minimum. It provides a simple API for handling asynchronous requests.The response is processed as JSON, and the data is logged on success. Errors are also handled.

4. **Got Package:** Got is used for making HTTP requests in Node.js environments. It offers a simple and flexible API, supporting asynchronous requests with Promises. Got is known for its ease of use and is chosen when developers need a reliable library with minimal configuration for handling HTTP calls.The response body is logged on success, and any errors are handled.

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

got('https://example.com/api/data')
  .then((response) => {
    console.log(response.body);
  })
  .catch((error) => {
    console.error(error.message);
  });
```

10.Keeping application running

- **Nodemon package: nodemon** is a tool in the Node.js ecosystem that helps developers in the development workflow by automatically restarting a Node.js application whenever changes are detected in the source code.
- **nodemon** monitors the files in the directory where it is run. If any file with specific extensions (like **.js**, **.json**, etc.) changes, **nodemon** automatically restarts the Node.js application.
- npm install -g nodemon → for local project →npm install --save-dev nodemon →After installation, instead of running your Node.js application with the **node** command, you use →'**nodemon your-app.js'**.
- **PM2:** For long-running applications or services, process management tools like PM2 can be used.
- PM2 monitors your application, restarts it in case of failures, and provides features for easy scaling. →# Install PM2 globally: **npm install -g pm2**→ # Start your application with PM2: **pm2 start app.js**

- **Graceful Shutdown:** Implementing a graceful shutdown mechanism allows your application to handle termination signals (SIGINT, SIGTERM) gracefully.
- This involves closing server connections, cleaning up resources, and ensuring a smooth shutdown process.

```
process.on('SIGINT', () => {
  // Perform cleanup tasks
  console.log('Received SIGINT signal. Shutting down gracefully...');
  process.exit(0);
});
```

- **Keep-Alive and Timeout Settings:** Configure server settings related to Keep-Alive and timeouts to control how long connections are kept open and how long the server waits for activity before closing idle connections.
- These settings can be crucial for handling a large number of concurrent connections efficiently.

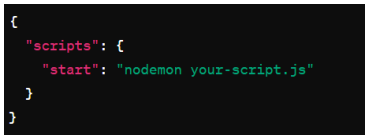
```
const server = http.createServer(app);

server.keepAliveTimeout = 65 * 1000; // Set Keep-Alive timeout to 65 seconds
server.headersTimeout = 70 * 1000;  // Set headers timeout to 70 seconds
```

Installation:

- You can install **nodemon** globally or as a development dependency in your project→**npm install -g nodemon**→Local Installation (as a development dependency)→**npm install --**

save-dev nodemon →After installation, you can use **nodemon** by simply replacing the **node** command with **nodemon** when running your script.

- **Global Usage: nodemon your-script.js** .
- **Local Usage:** Add a script in your **package.json** file→npm start →



Features:

1. **Automatic Restart: nodemon** monitors files for changes and automatically restarts the Node.js application.
2. **Custom Configuration:** Configuration can be specified in a **nodemon.json** file or in the **package.json** file under the **"nodemonConfig"** key.
3. **Ignore Files:** You can specify files or directories to be ignored by **nodemon**→ `nodemon --ignore lib/`
4. **Events and Extensibility: nodemon** emits events that can be used for custom scripts or integrations. Custom functionality can be added using the **--exec** option to run a custom script or command→ `nodemon --exec "npm run lint && node server.js"`
5. **Environment Variables: nodemon** allows you to set environment variables through the command line→ `nodemon --env PORT=3000`
6. **Delay and Quiet Mode: --delay** option sets a delay time before restarting to accommodate multiple file changes. → **--quiet** option suppresses unnecessary output. → `nodemon --delay 2 --quiet`
7. **Interactive Mode: nodemon** can run in interactive mode (**-i** or **--interactive**), which allows you to control the application through a REPL. → **nodemon -i**

Example Usage:

- Assuming you have a file named **app.js**, you can run it using **nodemon** like this→ **nodemon app.js**
- Now, whenever you make changes to your **app.js** file, **nodemon** will automatically restart the application.
- **nodemon** is a handy tool during development as it improves the developer experience by saving time and effort that would otherwise be spent manually restarting the server after every code change.

11.Templating engines

EJS (Embedded JavaScript): EJS is a simple templating language that lets you embed JavaScript code directly into your templates. → `npm install ejs`

Pug: Pug uses indentation to represent the structure of the document. Unlike traditional HTML that relies on opening and closing tags, Pug uses whitespace to indicate nesting. → `npm install pug`

Marko: `npm install marko express` →Marko provides additional tools to aid development, such as a browser extension for Chrome and Firefox. This extension allows developers to visually inspect Marko components in the browser's developer tools, making it easier to debug and understand the structure of the UI. Marko is a templating engine, allowing developers to define HTML templates with dynamic content. These templates can include conditional logic, loops, and other control flow constructs, making it easy to generate HTML dynamically based on data or application state.

12.Working with databases

Documents: A document database is a type of nonrelational database that is designed to store and query data as JSON-like documents. Document databases make it easier for developers to store and query data in a database by using the same document-model format they use in their application code. The flexible, semistructured, and hierarchical nature of documents and document databases allows them to evolve with applications' needs.

Mongoose package:

Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node.js. It provides a higher-level, schema-based abstraction over the MongoDB driver, making it easier to interact with MongoDB databases using JavaScript or TypeScript.

- Install mongoose: npm install mongoose
- Run the MongoDB Server.
- Define the Schema (Create userModel.js)

1. Referencing Mongoose

let mongoose = require('mongoose');

2.Define the Schema (Create userModel.js):A schema defines document properties through an object where the key name corresponds to the property name in the collection.

3. Exporting a Model

We need to call the model constructor on the Mongoose instance and pass it the name of the collection and a reference to the schema definition. Mongoose is a powerful ODM (Object Data Modeling) library for MongoDB and Node.js. It provides a variety of features and functionalities to simplify the interaction between Node.js applications and MongoDB databases.

- Mongoose is an ODM (Object Data Modeling) library for MongoDB and Node.js. It provides a higher-level, schema-based abstraction over the MongoDB driver, making it easier to interact with MongoDB databases in a Node.js application.
- Here are some key aspects and features of the Mongoose package:

1. **Schema Definition:**

- Mongoose allows you to define the structure of your MongoDB documents using a schema. A schema defines the fields, types, and constraints for each document.

2. **Model Creation:**

- Once you define a schema, you can create a model using Mongoose. Models are constructor functions that allow you to create, read, update, and delete documents in the MongoDB collection.

3. **Document Creation and Saving:**

- Mongoose models can be used to create new documents and save them to the database.

4. **Querying the Database:**

- Mongoose provides a rich set of methods for querying the database, allowing you to find documents that match specific conditions.

5. **Updating Documents:**

- Mongoose provides methods for updating documents in the database.

6. **Deleting Documents:**

- You can use Mongoose to delete documents from the database.

7. **Middleware (Hooks):**

- Mongoose supports middleware functions that allow you to run custom logic before or after certain operations (e.g., before saving a document).

8. **Validation:**

- Mongoose provides built-in and custom validators for data validation. You can define validation rules within your schema.

9. **Population:**

- Mongoose supports population, allowing you to reference documents from other collections and populate them in query results.

These features make Mongoose a powerful and widely used tool for working with MongoDB in Node.js applications. It simplifies the interaction with the database and provides additional features for data modeling and validation.

Prisma package:

- Prisma supports database migrations, allowing you to evolve the database schema over time while preserving data. **→npx prisma migrate save --name init –experimental→npx prisma migrate up –experimental**
- Prisma enables you to work with transactions, ensuring atomicity and consistency in database operations.
- Prisma supports connecting to multiple databases and switching between them in your application.

- Prisma provides typescripts functionality. It means typescripts code is transper into JavaScript using the TypeScript Compiler (**tsc**). This allows developers to write code with advanced features while ensuring compatibility with browsers and Node.js.

Native drivers

- Another way to connect to different databases in Node.js is to use the official native drivers provided by the database.
- **Various list of drivers in mongodb:**
- **C:** You can add the driver to your application to work with MongoDB in C. Download the required libraries, libmongoc and libbson.
- **C++:** You can add the driver to your application to work with MongoDB using the C++11 or later standard. Download the library, `mongocxx`, from mongocxx.org.
- **NET/C# Driver:** It provides an API for performing CRUD operations, querying, and other MongoDB-specific tasks using C# →You can install the MongoDB .NET driver using NuGet, the package manager for .NET→Install-Package MongoDB.Driver
- **Go Driver:** The Go driver for MongoDB allows Go developers to connect to and interact with MongoDB databases. It provides functions for CRUD operations, querying, and managing MongoDB-specific features in a Go application. →Install the Go driver using the go get command:→**go get go.mongodb.org/mongo-driver/mongo**
- **Java Drivers:**Java developers can use MongoDB Java drivers to connect to MongoDB databases. The drivers offer a Java API for performing operations such as inserts, updates, queries, and index management→You can include the Java driver in your project using Maven or Gradle.
- **Kotlin Drivers:** Kotlin developers can use MongoDB Java drivers directly, as Kotlin is interoperable with Java. Alternatively, there may be Kotlin-specific extensions or libraries for more idiomatic MongoDB interactions in Kotlin.
- **Node.js Driver:** The Node.js driver for MongoDB allows JavaScript developers (Node.js) to interact with MongoDB. It provides asynchronous and non-blocking access to MongoDB, supporting CRUD operations, queries, and more. →Install the Node.js driver using npm→ **npm install mongodb**
- **PHP Driver**: The MongoDB PHP driver enables PHP developers to interact with MongoDB databases. It provides functions for connecting to MongoDB, performing CRUD operations, and handling BSON data→ Install the PHP driver using the pecl command→**pecl install mongodb**

- **Python Drivers:** Python developers can use MongoDB drivers like PyMongo. These drivers provide a Pythonic interface for connecting to MongoDB, executing queries, and handling BSON data. →Install PyMongo using pip→pip install pymongo
- **Ruby Drivers:** The MongoDB Ruby driver allows Ruby developers to connect to and interact with MongoDB databases. It provides a Ruby API for performing operations such as inserts, updates, queries, and more. →Install the Ruby driver using gem→gem install mongo
- **Rust Driver:** The Rust driver for MongoDB allows Rust developers to interact with MongoDB databases. It provides a Rust-friendly API for CRUD operations and other MongoDB-specific features. →Add the MongoDB driver to your Cargo.toml file.
- **Scala Driver:** Scala developers can use MongoDB Scala drivers to interact with MongoDB. These drivers provide a Scala API for connecting to MongoDB, executing queries, and handling BSON data. →Include the Scala driver in your project dependencies.
- **Swift Driver:** The MongoDB Swift driver allows Swift developers to interact with MongoDB databases. It provides a Swift API for CRUD operations and other MongoDB-specific feature→Add the Swift driver to your Swift Package Manager dependencies.

Relational:

A relational database is a (most commonly digital) database based on the relational model of data, as proposed by E. F. Codd in 1970.

Knex package: Knex.js is a "batteries included" SQL query builder for PostgreSQL, CockroachDB, MSSQL, MySQL, MariaDB, SQLite3, Better-SQLite3, Oracle,and Amazon Redshift designed to be flexible, portable, and fun to use.

It features both traditional node style callbacks as well as a promise interface for cleaner async flow control, a stream interface, full-featured query and schema builders, transaction support (with savepoints), connection pooling and standardized responses between different query clients and dialects.

TypeORM package:

An ORM **is** known as Object Relational Mapper. This is a tool or a level of abstraction which maps(converts) data in a relational database into programmatic objects that can be manipulated by a programmer using a programming language(usually an OOP language). ORMs solely exist to map the details between two data sources which due to a mismatch cannot coexist together.

- TypeORM is an ORM that can run in NodeJS, Browser, Cordova, PhoneGap, Ionic, React Native, NativeScript, Expo, and Electron platforms and can be used with TypeScript and JavaScript (ES5, ES6, ES7, ES8).
- Its goal is to always support the latest JavaScript features and provide additional features that help you to develop any kind of application that uses databases - from small applications.
- TypeORM is an ORM that can run in NodeJS, Browser, Cordova, PhoneGap, Ionic, React Native, NativeScript, Expo, and Electron platforms and can be used with TypeScript and JavaScript (ES2021).
- Its goal is to always support the latest JavaScript features and provide additional features that help you to develop any kind of application that uses databases - from small applications with a few tables to large-scale enterprise applications with multiple databases.
- TypeORM supports both Active Record and Data Mapper patterns.
- **Active Record Pattern:**
- Definition: In the Active Record pattern, each database table is represented by a corresponding model class, and instances of these classes directly interact with the database.
- Characteristics: Models contain both data and behavior (CRUD operations).
- The model is responsible for its own persistence.
- **Data Mapper Pattern:**
- Definition: In the Data Mapper pattern, there is a clear separation between the domain objects (models) and the database mapping logic. A mapper class handles the communication between the domain objects and the database.
- Characteristics: Models are plain objects without database-related logic.
- Mappers handle database interactions and mapping data to/from models.
- TypeORM is highly influenced by other ORMs, such as Hibernate, Doctrine and Entity Framework.

Sequelize package: **npm install sequelize sqlite3→**or **yarn add sequelize sqlite3**

Installing

- Sequelize is available via npm (or yarn).
- npm install --save sequelize
- You'll also have to manually install the driver for your database of choice:
- **Install One of the following:**

- $ npm install --save pg pg-hstore # Postgres
- $ npm install --save mysql2
- $ npm install --save mariadb
- $ npm install --save sqlite3
- $ npm install --save tedious # Microsoft SQL Server
- $ npm install --save oracledb # Oracle Database Installing

- Connecting to a database
- Testing the connection
- Closing the connection
- Terminology convention
- Tip for reading the docs
- New databases versus existing databases
- Logging

Prisma Package:

ORM (Object-Relational Mapping): Prisma acts as an ORM, providing a way to interact with databases using TypeScript. It allows developers to work with databases using a type-safe API generated based on the database schema. This helps catch errors at compile time and improves code maintainability.

Schema Definition: Developers define the data model using a schema file (often named schema.prisma). This schema serves as the source of truth for the database structure, and Prisma uses it to generate TypeScript types and database query builders.

Code Generation: Prisma generates TypeScript code based on the database schema. This generated code includes TypeScript types for database entities and query builders for performing CRUD (Create, Read, Update, Delete) operations.

Database Migrations: Prisma supports database migrations, allowing developers to evolve the database schema over time. Migrations help manage changes to the database structure while preserving existing data.

Native Drivers:

Database Connectivity: Prisma uses native drivers or connectors to establish connections and communicate with various types of databases. These native drivers are specific to each supported database system (e.g., MySQL, PostgreSQL, SQLite, and others).

Optimized Performance: Native drivers are typically optimized for performance and efficiency. They understand the intricacies of the underlying database system and can leverage native features and optimizations provided by the database.

Query Translation: The native drivers are responsible for translating Prisma's high-level queries into the specific SQL or database commands understood by the target database. This ensures that the interactions between Prisma and the database are efficient and accurate.

Connection Pooling: Many native drivers include connection pooling mechanisms to efficiently manage and reuse database connections, reducing the overhead of establishing new connections for each query.

Testing:

Software testing is the process of verifying that what we create is doing exactly what we expect it to do. The tests are created to prevent bugs and improve code quality.

The two most common testing approaches are unit testing and end-to-end testing. In the first, we examine small snippets of code, in the second, we test an entire user flow.

1.End-to-endTesting:

End-to-end testing is the type of software testing used to test entire software from starting to the end along with its integration with the external interfaces. The main purpose of end-to-end testing is to identify the system dependencies and to make sure that the data integrity and communication with other systems, interfaces, and databases to exercise complete productions.

Advantages of End-to-End Testing :

- Comprehensive Testing: End-to-end testing ensures that all components of an application work together as expected, thereby providing comprehensive testing coverage.
- Realistic Scenarios: This type of testing simulates realistic process, which helps to identify issues that may not be detected in other forms of testing.
- Customer-Centric Testing: End-to-end testing focuses on the user's experience, and ensures that the application meets the customer's requirements.
- Early Bug Detection: By testing the entire system, including all its components, end-to-end testing helps detect bugs and defects early in the software development lifecycle.
- Streamlined Testing Process: End-to-end testing reduces the time and effort required to perform separate unit tests, integration tests, and system tests.

Disadvantages of End-to-End Testing :

- Time-Consuming: End-to-end testing is often time-consuming due to the complexity of the test scenarios, and may require significant resources.
- Costly: This type of testing can be expensive, as it involves testing the entire system, which may require a significant investment in time, personnel, and infrastructure.
- Limited Test Coverage: End-to-end testing may not cover all scenarios, as it is impossible to test every possible combination of inputs, especially in large systems.
- Difficult to Isolate Issues: When a test fails, it can be difficult to isolate the cause of the failure, as it may be due to issues in multiple components.
- Maintenance: End-to-end tests are often fragile and require frequent updates as the application evolves, leading to additional maintenance costs.

2. **UnitTesting:**

Unit Testing is the type of software testing level in which each individual component of software are tested separately. It is generally performed by a developer. It can't be used for those systems which have a lot of interdependence between different modules. It does not allow for parallel testing.

Advantages of Unit Testing :

- Early Bug Detection: Unit tests detect bugs early in the development cycle when they are easier and less costly to fix. Since each unit is tested in isolation, it is easier to identify and isolate issues.
- Increased Quality of Code: Unit tests ensure that each unit or component works as expected and meets the requirements, leading to higher code quality and more reliable software.
- Faster Feedback: Unit testing provides fast feedback to developers, allowing them to quickly identify and fix issues, reducing the overall development time.
- Cost-effective: Unit testing is a cost-effective testing approach as it requires fewer resources compared to other types of testing, such as end-to-end testing or integration testing.
- Simplified Debugging: Since each unit is tested in isolation, debugging is easier and less time-consuming, as developers can quickly identify the cause of the problem.

Disadvantages of Unit Testing :

- Limited Scope: Unit testing only tests individual units in isolation, and it may not capture the interactions and dependencies between the units, which can lead to issues in the integrated system.
- Time-consuming: Writing and maintaining unit tests can be time-consuming, especially for complex systems with many units.

- Skill Required: Writing effective unit tests requires a certain level of skill and experience in software testing, which may not be present in every developer.
- False Sense of Security: Unit tests only test individual units and do not guarantee that the system will work correctly as a whole.
- Overhead: Maintaining a comprehensive suite of unit tests can create additional overhead, which can be burdensome, particularly in large and complex systems.

Jest: npm install --save-dev jest→ Jest is a delightful JavaScript Testing Framework with a focus on simplicity. It works with projects using: Babel, TypeScript, Node, React, Angular, Vue and more!

Mocha: `npm i mocha` → Mocha is an open source JavaScript test framework running on Nodejs and in the browser, making asynchronous testing simple and fun, and it's a great candidate for BDD (Behavior Driven Development).

Cypress: Cypress is a new front end testing tool built for the modern web. It enables you to write faster, easier and more reliable tests.

13.Logging

- Logging is an essential part of understanding the complete application life cycle of the Node.js application. We can much more easily and quickly fix errors by looking at logs throughout the development process, from creating to debugging to designing new features. Error, warn, info, and debug are the four basic logging levels in Node.js. Logging involves persistently collecting information about an application's runtime behaviour.
- Operations engineers and developers use logs for debugging. Product managers and UX designers use logs for planning and design.
- Node.js logging is an important part of supporting the complete application life cycle. From creation to debugging to planning new features, logs support us all the way. By analysing the data in the logs, we can glean insights, resolve bugs much quicker, and detect problems early and as they happen.
- The original method of logging is console.log. It has variants like console.error, console.info, and console.warn. Those are just convenience methods over the base function which is: console.log(level, message)
- You can write to console.log, but you have little control over where things log from outside the code.

Winston: Winston is designed to be a simple and universal **logging library with support for multiple transports.** A transport is essentially a storage device for your logs. Each winston logger can have multiple transports configured at different levels. For example, one may want error logs to be stored in a persistent remote location (like a database), but all logs output to the console or a local file.

Morgan: Is a NodeJS and express.js middleware(Middleware functions can perform various tasks, modify the request or response, and control the flow of the application) to log the HTTP request and error, simplifying the debugging process. It provides flexibility in defining the format of log messages and helps override the output destination for your logs.

- morgan is a Node.js and Express middleware to log HTTP requests and errors, and simplifies the process. In Node.js and Express, middleware is a function that has access to the request and response lifecycle methods, and the next() method to continue logic in your Express server.

- morgan is used in Node.js applications, particularly with web frameworks like Express.js, for logging HTTP request information. Here are some reasons why developers use morgan:
- Request Logging: morgan logs details about incoming HTTP requests, providing insights into the behavior of your web server. It captures information such as the HTTP method, URL, status code, response time, and other relevant details.
- Debugging: During development and debugging, morgan helps developers trace the flow of requests and identify any issues or unexpected behavior in their application.The logs can be especially useful when troubleshooting errors or understanding how requests are processed.
- Monitoring and Analytics:Logging request information is valuable for monitoring server performance and understanding traffic patterns.
- Middleware Integration: morgan is easy to integrate into Express.js applications as middleware.
- Customization: morgan supports various predefined formats ('combined', 'common', 'dev', etc.) as well as the ability to create custom log formats.
- Performance Analysis: By logging response times, developers can identify slow-performing routes and optimize their application for better responsiveness.
- Security Auditing:Logging request details can aid in security auditing by providing a record of incoming requests.
- Compliance and Documentation:Request logs can serve as a form of documentation for your API or web service, helping developers understand how to interact with your application.

14. Keeping app running

Pm2 package:

- PM2 lets you run your nodejs scripts forever. In the event that your application crashes, PM2 will also restart it for you. →Install PM2 globally to manager your nodejs instances.
- PM2 is a process manager for Node.js applications. It allows you to manage, monitor, and deploy Node.js applications in production environments. PM2 provides features like process clustering, load balancing, automatic application restarts.

➢ To use PM2, you need to install it globally on your machine→npm install -g pm2

- After installing PM2, you can start and manage your Node.js applications using various PM2 commands:
- **Start an Application:** To start a Node.js application using PM2, navigate to the directory containing your application and run→pm2 start app.js →Replace **app.js** with the entry file of your application.
- **List Running Processes:** To view a list of running processes managed by PM2, use→ pm2 list
- **Stop and Restart:** To stop or restart a process, you can use→ pm2 stop <app_name_or_id> →pm2 restart <app_name_or_id>
- **View Logs:** To view logs of a specific process, you can use→ pm2 logs <app_name_or_id>

This command shows real-time logs. Press **Ctrl + C** to exit the log view.

- **Monitoring:** To monitor CPU and memory usage of processes, use→pm2 monit
- **Startup Script:** To ensure that PM2 starts your applications on system startup, use→pm2 startup

This command generates a script that you can add to your system's startup configuration.

Forever package: `npm install -g forever` Forever is a node.js package for ensuring that a given script runs continuously (i.e. forever) even when the server crash/stops.

- **pm2** and **forever** are both process managers for Node.js applications, designed to keep your Node.js processes running continuously, manage process scaling, and provide additional features like logging and monitoring.

Nohup: nohup (short for "no hang up") is a Unix command that is used to run a command in the background and detach it from the current shell session. It is particularly useful when you

want a process to continue running even after you log out of a terminal or close your SSH session→ useful for background process, running scripts, long running process.

Threads:

Node.js is a single-threaded language and gives us ways to **work parallelly** to our main process.

Child process: The child_process module gives the node the ability to run the child process, established through IPC (inter-process communication) by accessing operating system commands.The **three** main methods inside this module are :

child_process.spawn(): The **spawn()** method is used to launch a new process with the given command and arguments.

child_process.fork() : The **fork()** method is a special case of **spawn()** specifically designed for running Node.js modules as separate processes. It creates a new Node.js process and runs the specified module in that process. Communication between the parent and child process is possible through inter-process communication (IPC).

child_process.exec():exec() is suitable for running simple shell commands and capturing their output. The **exec()** method is used to run shell commands.

Cluster: One of the main purposes of the **cluster** module is to allow multiple Node.js processes to run in parallel.Each process created by the cluster represents a worker, and these workers can handle incoming requests concurrently.

Worker thread: worker threads provide a way to perform parallel computation and take advantage of multi-core systems. Worker threads allow you to run JavaScript code in parallel, using separate threads, and communicate with the main thread through a messaging system. This is different from the event-driven, single-threaded nature of the main Node.js thread.

Streams: Streams are a type of data handling methods and are used to read, write or transform chunks of data piece by piece without keeping it in memory all at once. There are four **types** of streams in Node.js.

- Readable: streams from which data can be read.
- Writable: streams to which we can write data.
- Duplex: streams that are both Readable and Writable.
- Transform: streams that can modify or transform the data as it is written and read.

Multiple streams can be chained together using pipe() method.

More debugging: Debugging is a concept to identify and remove errors from software applications.

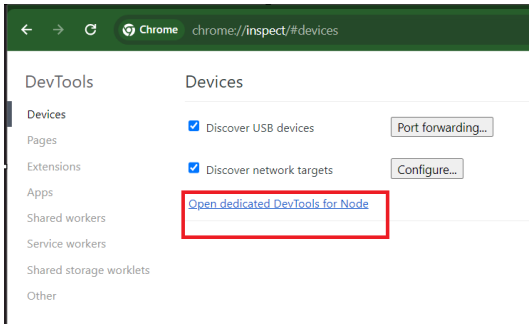
Why not to use console.log() for debugging?

Using console.log to debug the code generally dives into an infinite loop of "stopping the app and adding a console.log, and start the app again" operations. Besides slowing down the development of the app, it also makes the writing dirty and creates unnecessary code. Finally, trying to log out variables alongside with the noise of other potential logging operations, may make the process of debugging difficult when attempting to find the values you are debugging.

Memory leaks:

- Memory leaks are caused when your Node.js app's CPU and memory usage increases over time for no apparent reason. In simple terms, a Node.js memory leak is an orphan block of memory on the Heap that is no longer used by your app because it has not been released by the garbage collector. It's a useless block of memory. These blocks can grow over time and lead to your app crashing because it runs out of memory.
- **Global variables:**Global variables in Node.js are the variables referenced by the root node, which is global. It's the equivalent of window for JavaScript running in the browser.
- Global variables are never garbage collected throughout the lifetime of an app. They occupy memory as long as the app is running. Here's the kicker: this applies to any object referenced by a global variable, and all their children, as well. Having a large graph of objects referenced from the root can lead to a memory leak in Node.js applications.
- **Multiple references:** If you reference the same object from multiple objects, it can lead to a memory leak if one of the references is garbage collected while the other one is left dangling.
- **Closures:** Closures memorize their surrounding context. When a closure holds a reference to a large object in heap, it keeps the object in memory as long as the closure is in use.This implies easily ending up in situations where a closure holding such a reference can be improperly used leading to a memory leak.
- **Timers & Events:** The use of setTimeout, setInterval, Observers, and event listeners can cause memory leaks when heavy object references are kept in their callbacks without proper handling.

Node –inspect: click on this link → chrome://inspect/#devices→ Node.js provides a built-in DevTools-based debugger to allow debugging Node.js applications.



Using APM : These tools send your logs from your running application into a single location. They often come with high-powered search and query utilities so that you can easily parse your logs and visualize them. APM stands for Application Performance Monitoring, and it is a set of practices, tools, and services designed to help developers monitor and manage the performance and availability of their applications. Logging is an essential component of APM, as it allows you to gather and analyse information about your application's behavior and performance.

Garbage collection:

- Garbage collection is the automated process of identifying and reclaiming memory occupied by unreachable or unused objects.
- The garbage collector scans the heap (memory area where dynamically allocated objects reside) to identify objects that are no longer reachable.Node.js uses Chrome's V8 engine to run JavaScript. Within V8, memory is categorized into Stack and Heap memory.
- **Stack**: Stores static data, method and function frames, primitive values, and pointers to stored objects. The stack is managed by the operating system.
- **Heap**: Stores objects. Because everything in JavaScript is an object this means all dynamic data like arrays, closures, etc. The heap is the biggest block of memory and it's where **Garbage Collection (GC)** happens.

Garbage collection frees up memory in the Heap used by objects that are no longer referenced from the Stack, either directly or indirectly. The goal is to create free space for creating new objects. Garbage collection is generational. Objects in the Heap are grouped by age and cleared at different stages.

There are two stages and two algorithms used for garbage collection.

The heap has two main stages:

New Space: Where new allocations happen. Garbage collection is quick. Has a size of between 1 and 8 MBs. Objects in the New Space are called the Young Generation.

Old Space: Where objects that survived the collector in the New Space are promoted to. Objects in the Old Space are called the Old Generation. Allocation is fast, however, garbage collection is expensive and infrequent.

Common built in modules:

Built-in modules are already installed with **Node.js**, so you don't need to install them with any package manager (yarn, npm, etc.).

- **fs**: dealing with the system files.
- **os**: provides information about the operation system.
- **net**: to build clients and servers.
- **path**: to handle file paths.
- **url**: help in parsing URL strings.
- **http**: making Node.js transfer data over HTTP.
- **console**: to log information in the console.
- **assert**: provides a set of assertion tests.
- **process**: provides information about, and control over, the current process.
- **cluster**: able to creating child processes that runs simultaneously and share the same server port.
- **perf_hooks**: provides APIs for performance measurement
- **crypto**: to handle OpenSSL cryptographic functions.
- **Buffer**: provides APIs to handling streams of binary data.
- **DNS**: enables name resolution.
- **events**: for handling existing events and creating custom events.
- **child_processes**: provides the ability to spawn subprocesses.
- **REPL**: provides a Read-Eval-Print-Loop (REPL) implementation that is available both as a standalone program or includible in other applications.
- **readline**: provides an interface for reading data from a Readable stream one line at a time.
- **util**: supports the needs of Node.js internal APIs.
- **querystring**: provides utilities for parsing and formatting URL query strings.
- **string_decoder**: provides an API for decoding Buffer objects into strings.
- **tls**: provides an implementation of the Transport Layer Security (TLS) and Secure Socket Layer (SSL) protocols.